

Basiswissen für Softwarearchitekten

Aus- und Weiterbildung nach iSAQB®-Standard zum
Certified Professional for Software Architecture –
Foundation Level

» Hier geht's
direkt
zum Buch

DIE LESEPROBE

1 Einleitung

Software ist allgegenwärtig. Dies gilt sowohl für kommerzielle Unternehmenssoftware als auch für nahezu alle anderen Bereiche des beruflichen, öffentlichen und privaten Alltags: Fliegen, Telefonieren, Überweisen, Autofahren – all das wäre ohne Software kaum noch möglich. In jedem Haushalt und in vielen Alltagsgegenständen, von der Waschmaschine bis zum Auto, werden softwaregesteuerte Bestandteile verwendet [BJ++06]. Software steht in der Regel nicht autark für sich, sondern ist in Geräte mit Hardware und Elektronik oder in Geschäftsprozesse, mit denen Unternehmen ihre Wertschöpfung erzielen, eingebettet [TTL00].

Der Nutzen und wirtschaftliche Erfolg von Unternehmen und Produkten wird zunehmend von Software und deren Qualität bestimmt (siehe [BM++96], [SV99], [TTL00]). Als Folge stehen Softwareingenieure und damit die Disziplin Software Engineering vor der Herausforderung, immer komplexere Anforderungen immer schneller und kostengünstiger bei gleichzeitig hoher Softwarequalität umzusetzen.

Die kontinuierliche Steigerung der Größe und Komplexität von softwareintensiven Systemen hat inzwischen dazu geführt, dass sie zu den komplexesten von Menschen geschaffenen künstlichen Systemen überhaupt zählen. Bestes Beispiel ist das Internet: ein auf Software basierendes weltumspannendes System. Inzwischen ist das Internet sogar auf der internationalen Raumstation ISS verfügbar und hat damit die Grenzen der Erde überschritten.

Nur ein strukturiertes und systematisches Herangehen kann dabei gesichert zum Erfolg führen. Trotz Anwendung etablierter Softwareentwicklungsmethoden bleibt die Anzahl der fehlgeschlagenen Softwareprojekte seit Jahren erschreckend hoch. Um dem entgegenzuwirken, versucht man in den frühen Phasen des Software Engineering bereits möglichst viele Fehler zu vermeiden bzw. dort zu identifizieren und auszumerzen. Zu diesen Phasen zählen insbesondere das Requirements Engineering sowie die Softwarearchitektur. Getreu den Worten von Ernst Denert, einem der Väter der methodischen Softwareentwicklung, wollen wir uns hier mit Softwarearchitektur beschäftigen, der »Königdisziplin des Software Engineering« (zitiert aus dem Geleitwort von Ernst Denert in [Sie04]).

1.1 Softwarearchitektur als Disziplin im Software Engineering

Bereits in den 60er-Jahren wurden die Probleme mit Softwareprojekten unter dem Stichwort Softwarekrise bekannt. 1968 fand in Garmisch eine NATO-Konferenz hochrangiger Forscher und Praktiker statt, um unter dem Titel »Software Engineering« über die Zukunft der Softwareentwicklung nachzudenken. Heute gilt diese Konferenz als Geburtsstunde des Software Engineering [Dij72].

Im Vergleich zu traditionellen Ingenieurdisziplinen wie beispielsweise dem Bauwesen, das auf mehrere Tausend Jahre Erfahrung zurückblicken kann, ist Software Engineering mit dem Geburtsjahr 1968 noch sehr jung. So erscheint es auch nicht verwunderlich, dass dessen Teildisziplin Softwarearchitektur noch deutlich jünger ist. Abbildung 1.1 demonstriert dies deutlich: Das Web of Knowledge, eine der großen und renommierten Publikationsdatenbanken, verzeichnet erst ab den 90er-Jahren eine wachsende Anzahl von Publikationen zum Thema Softwarearchitektur [Reu12].

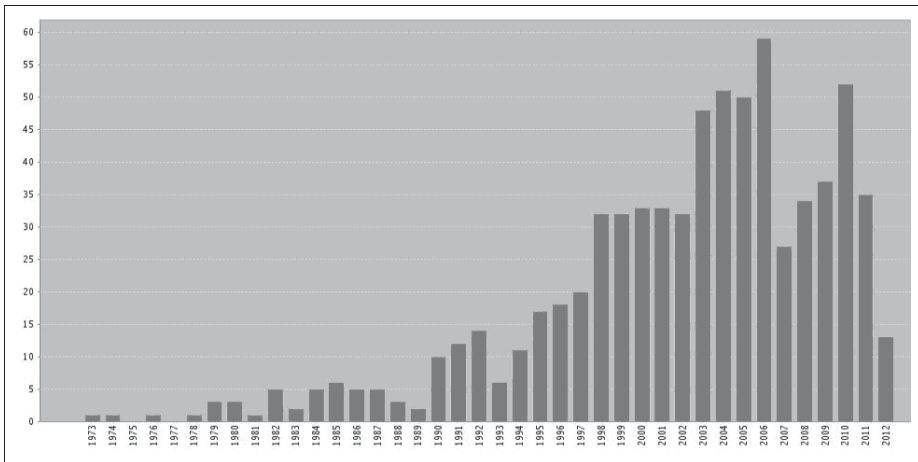


Abb. 1.1 Veröffentlichungen zu Softwarearchitektur seit 1973 [Reu12]

Betrachten wir hingegen die klassische Architektur im Bauwesen, so können wir auf eine bereits Jahrtausende währende Tradition zurückblicken. Ein wichtiger Vordenker war hier Marcus Vitruvius Pollio, ein römischer Architekt aus dem ersten Jahrhundert vor Christus. Er ist Autor des Werkes »De architectura«, das heute unter dem Titel »Ten Books on Architecture« bekannt ist [Vit60]. Vitruvius vertrat die These, dass gute Architektur durch eine kunstvolle Kombination der folgenden Elemente zu erreichen sei:

- **utilitas (Nützlichkeit):**
Das Gebäude erfüllt seine Funktion.
- **firmitas (Festigkeit):**
Das Gebäude ist stabil und langlebig.
- **venustas (Schönheit):**
Das Gebäude ist ästhetisch gestaltet.

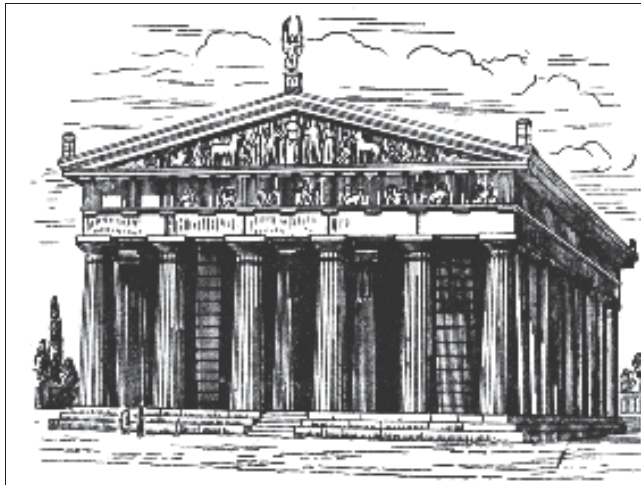


Abb. 1.2 *Architektur im alten Rom*

Diese These lässt sich direkt auf die Disziplin Softwarearchitektur übertragen. Ziel der Softwarearchitektur und damit Aufgabe eines Softwarearchitekten ist es, ein System zu konstruieren, das in einem kunstvoll ausgewogenen Dreiklang die drei folgenden Eigenschaften vereint:

- **utilitas (Nützlichkeit):**
Die Software erfüllt die funktionalen und nicht funktionalen Anforderungen der Nutzer und Kunden.
- **firmitas (Festigkeit):**
Die Software ist stabil im Hinblick auf die geforderten Qualitätseigenschaften, z. B. die Anzahl der gleichzeitig zu bedienenden Nutzer, und langlebig, da zukünftige Weiterentwicklungen möglich sind, ohne das System komplett neu bauen zu müssen.
- **venustas (Schönheit):**
Die Software ist sowohl nach außen gegenüber dem Nutzer als auch nach innen gegenüber demjenigen, der die Software pflegen und weiterentwickeln soll, gut strukturiert. Dadurch ist sie intuitiv nutzbar und die internen Strukturen sind leicht verständlich, sodass die Aufgaben gut erledigt werden können.

1.2 iSAQB® – International Software Architecture Qualification Board

Softwarearchitektur ist eine junge Disziplin, über deren Umfang und Ausgestaltung in der Informatik trotz vieler Publikationen immer noch unterschiedliche Meinungen kursieren. Aufgaben und Verantwortungsbereiche von Softwarearchitekten werden unterschiedlich definiert und in Softwareprojekten ständig neu verhandelt.

Für andere Disziplinen im Software Engineering hingegen, wie z.B. beim Projektmanagement, Requirements Engineering oder Testen, gibt es inzwischen einen deutlich ausgereifteren Wissenskanon. Dafür bieten unabhängige Organisationen Lehrpläne an, die klar beschreiben, welche Kenntnisse und Fähigkeiten eine entsprechende Ausbildung vermitteln soll (Testen: www.istqb.org [ISTQB], Requirements Engineering: www.ireb.de [IREB], Projektmanagement: www.pmi.org [PMI]).

Vor diesem Hintergrund haben Anfang 2008 verschiedene Softwarearchitekturexperten aus Wirtschaft und Wissenschaft das »International Software Architecture Qualification Board« als eingetragenen Verein (iSAQB e.V., www.isaqb.org) gegründet. Dessen Ziel ist es, Standards für die Ausbildung und Zertifizierung von Softwarearchitekten zu definieren. Bewusst wird im iSAQB® jegliche Hersteller- oder Produktorientierung vermieden. Zertifizierungen auf den unterschiedlichen Stufen Foundation Level, Advanced Level und Expert Level ermöglichen es Softwarearchitekten, sich den Stand ihrer Kenntnisse und Fähigkeiten durch ein anerkanntes Verfahren bescheinigen zu lassen (siehe Abbildung 1.3).

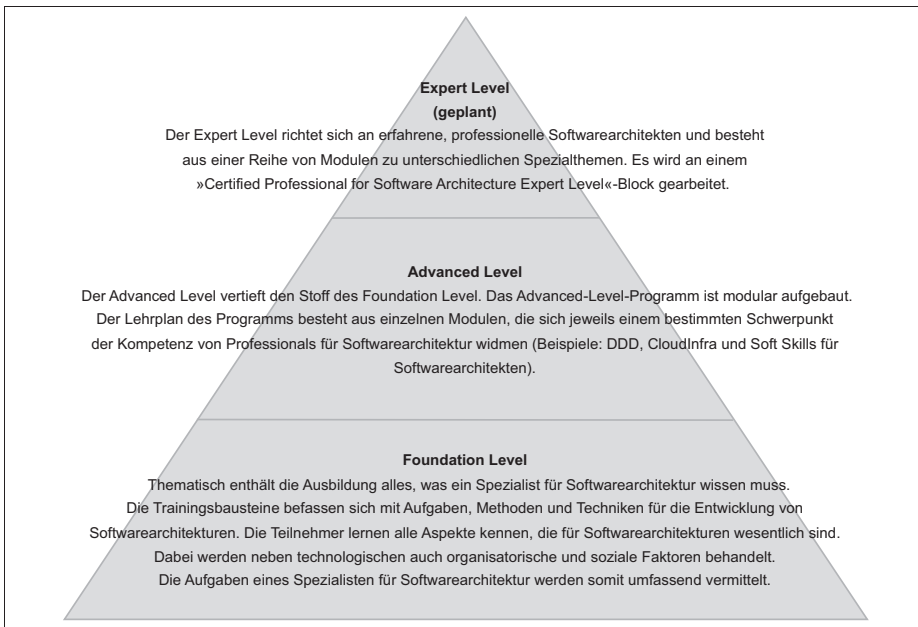


Abb. 1.3 iSAQB®-Zertifizierungsstufen (www.isaqb.org)

Von diesem standardisierten Lehr- und Ausbildungsplan profitieren sowohl etablierte als auch angehende Softwarearchitekten und ebenso Unternehmen oder auch entsprechende Aus- und Weiterbildungseinrichtungen, da er die eingangs geschilderte begriffliche Unsicherheit beseitigt. Nur auf Basis von präzisen Lehr- und Ausbildungsplänen kann eine Prüfung und Zertifizierung angehender Softwarearchitekten stattfinden und so letztlich ein qualitätsgesicherter Ausbildungsstand von Softwarearchitekten mit einem entsprechend akzeptierten Wissenskanon etabliert werden.

Die Zertifizierung zum Certified Professional for Software Architecture (CPSA) wird von unabhängigen Zertifizierungsstellen durchgeführt. Basis für die Zertifizierung zum CPSA (Foundation Level) ist ein anspruchsvoller, vom iSAQB® in Einklang mit dem Lehrplan entwickelter, nicht öffentlicher Fragenkatalog, aus dem eine Teilmenge als Prüfungsfragen ausgewählt wird. Für die Zertifizierung zum Advanced Level werden neben der Erfordernis des Besuchs von lizenzierten Schulungen bzw. der Anerkennung eines anderen, nicht durch den iSAQB® definierten Zertifikats praktische Aufgaben gestellt. Der Expert Level befindet sich derzeit noch in Entwicklung.

Auf Basis dieses Lehrplans bieten verschiedene lizenzierte Schulungsveranstalter mehrtägige Kurse an, die Wissen in diesen Themengebieten auffrischen und vielfach deutlich vertiefen. Die Teilnahme an einem Kurs wird zwar nachdrücklich empfohlen, ist jedoch nicht Bedingung für die Prüfungsanmeldung zur Zertifizierung.

1.3 Certified Professional for Software Architecture – Foundation und Advanced Level

Der iSAQB® hat inzwischen nicht nur die Zertifizierungsrichtlinien für den CPSA Foundation Level, sondern auch für den Advanced Level definiert.

Der Advanced Level ist modular aufgebaut und besteht aus einzelnen Schulungen, die sich jeweils einem bestimmten Schwerpunkt der Kompetenz eines IT-Professionals widmen:

- **Methodische Kompetenz:**
Wissen und Fähigkeiten im Bereich des systematischen Vorgehens bei IT-Projekten, unabhängig von Technologien
- **Technische Kompetenz:**
Wissen und Fähigkeiten im Bereich des Einsatzes von Technologien zur Lösung von Entwurfsaufgaben
- **Kommunikative Kompetenz:**
Wissen und Fähigkeiten im Bereich der Kommunikation, Präsentation, Rhetorik und Moderation zur effektiven Wahrnehmung der Rolle im Softwareentwicklungsprozess

Voraussetzungen für den Advanced Level sind:

- Ausbildung und Zertifizierung zum CPSA-F (Foundation Level)
- Mindestens 3 Jahre Berufserfahrung in der IT-Branche
- Mitarbeit an Entwurf und Entwicklung von mindestens zwei verschiedenen IT-Systemen
- Für die Prüfung: mindestens 70 Credit Points aus allen drei Kompetenzbereichen (jeweils mindestens 10 Credit Points)

Die Prüfung besteht aus der Bearbeitung einer Prüfungsaufgabe in Eigenregie und der anschließenden Besprechung der Lösung mit zwei unabhängigen Prüfern in einem Interview.

Für den Foundation Level wurden die Bereiche, in denen ein Softwarearchitekt über fundiertes Wissen und geeignete Fähigkeiten verfügen sollte, im Rahmen eines öffentlich zugänglichen Lehrplans beschrieben [isaqb-lehrplan]. Danach soll angehenden Softwarearchitekten folgendes Spektrum an Inhalten vermittelt werden:

- der Begriff und die Bedeutung von Softwarearchitektur,
- die Aufgaben und Verantwortungsbereiche von Softwarearchitekten,
- die Rolle des Softwarearchitekten in Projekten,
- State-of-the-Art-Methoden und -Techniken zur Entwicklung von Softwarearchitekturen.

Im Mittelpunkt steht der Erwerb folgender Fähigkeiten:

- mit anderen Projektbeteiligten aus den Bereichen Anforderungsmanagement, Projektmanagement, Test und Entwicklung wesentliche Softwarearchitekturentscheidungen abzustimmen,
- Softwarearchitekturen auf Basis von Sichten, Architekturmustern und technischen Konzepten zu dokumentieren und zu kommunizieren,
- die wesentlichen Schritte beim Entwurf von Softwarearchitekturen zu verstehen und für kleine und mittlere Systeme selbstständig durchzuführen.

Die Schulung zum Foundation Level vermittelt das notwendige Wissen, um für kleine und mittlere Systeme ausgehend von einer hinreichend detailliert beschriebenen Anforderungsspezifikation eine dem Problem angemessene Softwarearchitektur zu entwerfen und zu dokumentieren. Diese kann dann als Implementierungsgrundlage bzw. -vorlage genutzt werden. Teilnehmer erhalten das Rüstzeug, um problembezogene Entwurfsentscheidungen auf der Basis ihrer vorab erworbenen Praxiserfahrung zu treffen.

Abbildung 1.4 zeigt die inhaltliche Struktur und die Gewichtung der einzelnen Bereiche des Lehrplans für den iSAQB® Certified Professional for Software Architecture (CPSA), Foundation Level.

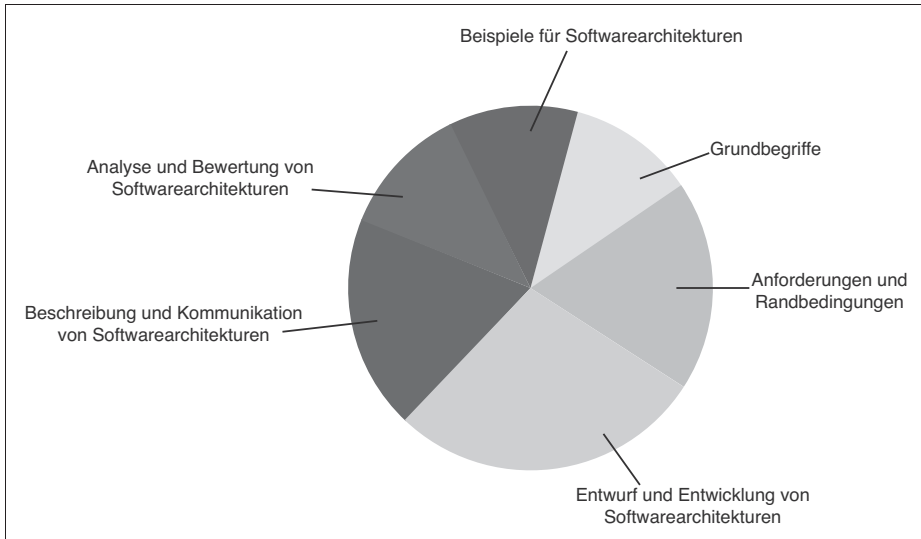


Abb. 1.4 Struktur des iSAQB®-Lehrplans für CPSA, Foundation Level

Sie haben die Möglichkeit, sich bei verschiedenen unabhängigen Anbietern durch eine Prüfung gemäß dem iSAQB®-Lehrplan zertifizieren zu lassen. Für die Zertifizierung setzen die Prüfungsanbieter standardisierte Prüfungsfragen ein, die der iSAQB® erarbeitet hat.

Für die Prüfungen wird ein Multiple-Choice-Verfahren verwendet. Entsprechend objektiv ist das Prüfungsergebnis messbar.

Mit der Prüfung können Sie somit Ihr notwendiges Grundlagenwissen als Softwarearchitekt nachweisen. Natürlich müssen Sie dann später in der Anwendung zeigen, dass Sie Ihr Wissen auch praktisch und erfolgreich in konkreten Architekturen einzusetzen wissen.

1.4 Zielsetzung des Buches

Wir, das Autorenteam des Buches, haben gemeinsam mit anderen iSAQB®-Mitgliedern am iSAQB®-Lehrplan für den Certified Professional for Software Architecture, Foundation Level, gearbeitet. Im Rahmen dieser Zusammenarbeit ist auch die Idee zu diesem Buch entstanden. Dementsprechend verfolgen wir darin die zentrale Zielsetzung, kompakt und prägnant das notwendige Wissen für die CPSA-Prüfung, Foundation Level, und somit das Fundament für den Wissenskanon in der Disziplin Softwarearchitektur bereitzustellen. Das Buch ist demzufolge die ideale Referenz für eine entsprechende Prüfungsvorbereitung. Wir empfehlen Ihnen ergänzend den Besuch entsprechender Schulungen, da dort das Lehrmaterial durch über dieses Buch hinausgehende praktische Beispiele von Softwarearchitekturen und persönliche Erfahrungen der Dozenten oder Trainer abgerundet wird.

Da der iSAQB® und somit auch das Buch primär auf methodische Fähigkeiten und Wissen fokussiert, gehören konkrete Implementierungstechnologien oder spezielle Werkzeuge explizit nicht zum standardisierten Lehrinhalt. Deshalb haben wir dieses Buch bewusst technologieneutral verfasst. Auch die von uns verwendeten Notationen, wie z.B. die UML, sind nur exemplarisch zu verstehen. Ebenso ist es nicht Ziel des Buches, ein einzelnes konkretes Vorgehensmodell oder einen spezifischen Entwicklungsprozess darzustellen. Vielmehr werden von uns an vielen Stellen mehrere Beispiele etwa für Notationen oder Vorgehensmodelle kurz vorgestellt.

In diesem Buch erklären wir vor allem wichtige Begriffe und Konzepte der Softwarearchitektur und stellen deren Bezug zu anderen Disziplinen dar. Darauf aufbauend führen wir die grundlegenden Techniken und Methoden für den Entwurf und die Entwicklung, die Beschreibung und Kommunikation sowie die Qualitätssicherung von Softwarearchitekturen ein. Schließlich betrachten wir die Rolle, die Aufgaben, das Umfeld und die Arbeitsumgebung von Softwarearchitekten und deren Einbettung in die umfassende Organisations- und Projektstruktur.

1.5 Voraussetzungen

Entsprechend der oben genannten Zielsetzung setzt das vorliegende Buch – wie auch der iSAQB®-Lehrplan – Erfahrung in der Softwareentwicklung voraus. Insbesondere gehören folgende Inhalte nicht zum Lehrplan und sind damit auch nicht Thema des Buches, obgleich sie zu den notwendigen Kompetenzen von Softwarearchitekten zählen:

- typischerweise mehrjährige praktische Erfahrung in der Softwareentwicklung, erworben durch Programmierung unterschiedlicher Systeme,
- vertiefte Kenntnisse und praktische Erfahrung mit mindestens einer höheren Programmiersprache,
- Grundlagen der Modellierung und Abstraktion sowie der Modellierungssprache UML, insbesondere der Klassen-, Paket-, Komponenten- und Sequenzdiagramme sowie deren Bezug zum Quellcode,
- praktische Erfahrung in technischer Dokumentation, insbesondere in der Dokumentation von Quellcode, Systementwürfen oder technischen Konzepten,
- Kenntnisse über die Methodik beim Testen von Software in den verschiedenen Teststufen.

Darüber hinaus sind Kenntnisse und Erfahrung mit der Objektorientierung für das Verständnis einiger Konzepte hilfreich. Ebenfalls wünschenswert ist Erfahrung in der Konzeption und Implementierung verteilter ablaufender Anwendungen wie etwa Client-Server-Systeme oder Webanwendungen.

1.6 Leitfaden für den Leser

Der Aufbau dieses Buches orientiert sich vor allem an der Struktur und den Inhalten des iSAQB®-Lehrplans Foundation Level nach Abbildung 1.4 bzw. [isaqb-lehrplan]:

- In Kapitel 2 beschreiben wir grundlegende Begriffe und Inhalte des Themenbereichs Softwarearchitektur, die in den folgenden Kapiteln aufgegriffen und vertieft werden. Beispielsweise wird dort der Begriff der »Sicht« auf ein Softwaresystem im Rahmen einer Softwarearchitektur eingeführt.
- In Kapitel 3 diskutieren wir den Umgang mit Anforderungen und Randbedingungen. Dabei wird auf die Arbeit mit Randbedingungen und Einflussfaktoren, das Formulieren von Qualitätsanforderungen und den Einsatz von Szenarien eingegangen. Wichtige Begriffe: Softwarequalität, Qualitätsmodelle und Qualitätsmerkmale, Kompromisse zwischen Qualitätsmerkmalen, Taktiken und Praktiken zur Erreichung von Qualitätsmerkmalen, Szenarien.
- Aspekte des praktischen Entwurfs von Softwarearchitekturen erörtern wir in Kapitel 4. Themen sind dort unter anderem Varianten des Vorgehens bei der Architekturentwicklung, wichtige Architekturmuster wie Schichten, Pipes and Filters, Model View Controller sowie Entwurfsprinzipien wie Kopplung, Kohäsion und Trennung von Verantwortlichkeiten. Darüber hinaus werden Daten, Datenmodelle, Deployment und Betrieb behandelt.
- In Kapitel 5 werden ausgewählte praxisnahe Beschreibungsmittel sowie in der Praxis bewährte Richtlinien vorgestellt. Diese ermöglichen es Ihnen, Ihre Softwarearchitektur zu dokumentieren und zielgruppenorientiert anderen zu vermitteln. Das iSAQB®-Sichtenmodell, Querschnittsaspekte in Softwarearchitekturen sowie praktisch bewährte Richtlinien für die Dokumentation von Softwarearchitekturen sind Beispiele für den Inhalt des Kapitels.
- In Kapitel 6 werfen wir einen ersten Blick auf den Zusammenhang von Softwarearchitektur und Qualitätsfragestellungen. Wichtige Begriffe dieses Abschnitts sind u.a. bezogen auf Software: Architekturbewertung (qualitativ und quantitativ), ATAM (Architecture Tradeoff Analysis Method) und Risiken bezüglich der Erreichung von Qualitätsmerkmalen.
- Abschließend zeigen wir in Kapitel 7 (Exkurs) eine Reihe von Beispielen für Unterstützungswerkzeuge des Softwarearchitekten, wie z.B. solche zur Modellierung, Generierung oder Dokumentation.
- Einige beispielhafte Übungsfragen, ein Glossar sowie ein Quellenverzeichnis runden das Buch ab.

Speziell zur iSAQB®-Prüfungsvorbereitung sollten Sie vor allem die Kapitel 2 – Kapitel 6 gründlich durcharbeiten und ergänzend einen Blick in Kapitel 7 und weitere Exkurse werfen. Ansonsten empfiehlt es sich, zumindest das Kapitel 2

komplett zu lesen und anschließend die Themen zu vertiefen, die Sie besonders interessieren.

1.7 Zielpublikum

Als Zielpublikum dieses Buches sehen wir in erster Linie Zertifizierungsinteressierte, die es – ggf. neben Schulungen – als Vorbereitung zur Prüfung einsetzen wollen. Hinzu kommen Praktiker und Studierende, die die praktischen Grundbegriffe von Softwarearchitekturen kennenlernen wollen.

Interessant ist dieses Buch auch für Softwareprojektmanager, Softwareproduktmanager sowie Entscheider auf der mittleren Softwareentwicklungsebene als Einstieg Überblick zum Thema Softwarearchitektur.

1.8 Danksagungen

Wir möchten uns an dieser Stelle beim iSAQB-Verein für seine unterstützende Mitwirkung bedanken. Frau Ingrid Schindler vom Lehrstuhl für Software Systems Engineering der Technischen Universität Clausthal und Mitarbeiter der ITech Progress haben uns beim Anfertigen der Abbildungen sehr unterstützt. Die Erstellung der 6. Auflage wäre ohne die unermüdliche Unterstützung von Dimitri Blatner und Thomas Garratt von ITech Progress nicht möglich gewesen.

Unserer Betreuerin seitens des dpunkt.verlags, Frau Christa Preisendanz, danken wir für ihre Geduld.

Zu guter Letzt möchten wir ganz besonders unseren Familien und Partnern danken, die an zahllosen Tagen die Zeit und die Geduld aufgebracht haben, uns gemeinsam an diesem Buch arbeiten zu lassen.

2 Grundlagen von Softwarearchitekturen

Wie bereits eingangs beschrieben, ist Software heute fast allgegenwärtig. Nahezu rund um die Uhr verlassen wir uns auf das korrekte Funktionieren von Software, angefangen vom Klingeln des Weckers am Morgen über funktionierende Bremsen in Autos und bei der Bahn bis zur Verwaltung unseres Geldes auf Bankkonten.

Trotz dieser Allgegenwärtigkeit und unserer Abhängigkeit von Software haben wir Softwareingenieure es immer noch nicht in der notwendigen Tiefe verstanden, wie man Software wiederholbar erfolgreich baut: Softwareprojekte dauern zu lange, kosten zu viel, scheitern zu oft. Und selbst wenn ein Softwareprojekt erfolgreich in den Betrieb geht, ist das Ergebnis für die Beteiligten oft mangelhaft. Dies attestiert uns der CHAOS-Report der Standish Group in jährlichen Abständen immer wieder [Sta99]. Auch alle Kritik am CHAOS-Report im Speziellen kann nicht darüber hinwegtäuschen, dass andere Erfolgserhebungsmethoden ganz ähnliche, nicht schmeichelhafte Ergebnisse liefern (siehe [EK08], [EV10]): Unterm Strich ist unsere Fähigkeit, Softwareprojekte innerhalb des magischen Vierecks (siehe [Bal00], [Die00], [Dum01], [Lit05], [May05]) erfolgreich abzuwickeln, sehr begrenzt (vgl. Abbildung 2.1). Wir schaffen es immer noch nicht, wiederholbar qualitativ hochwertige Software zu erschwinglichen Kosten und im vorgegebenen Zeitfenster mit der notwendigen Funktionalität zu erstellen.

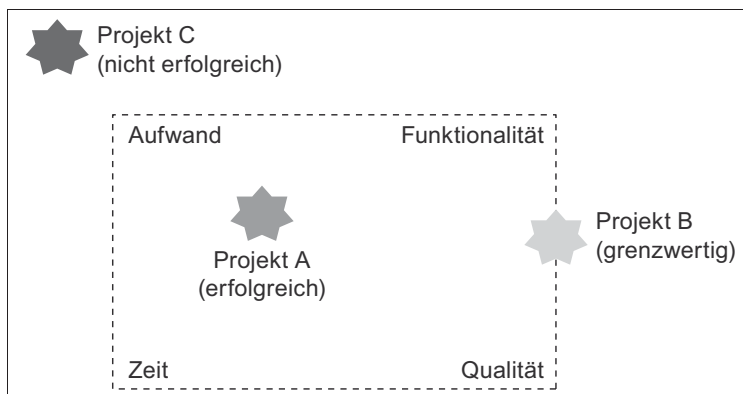


Abb. 2.1 Das magische Viereck erfolgreicher Softwareprojekte

Will man erfolgreich Software entwickeln, dann sind Requirements Engineering und Architekturentwurf nachweislich zwei zentrale Schlüsselfaktoren. Bei beiden ist das Risiko von gravierenden Fehlentwicklungen hoch, da früh – insbesondere bei noch eingeschränktem Wissensstand – Entscheidungen zu treffen sind, deren Auswirkungen weit reichen und teilweise erst deutlich später im Projektverlauf erkennbar werden (siehe [Nus01], [GEM04]).

Deshalb ist Softwarearchitektur einer der entscheidenden Erfolgsfaktoren in der Softwareentwicklung. Denn der Architekturentwurf entscheidet mit darüber, wie man Millionen von Programmzeilen großer softwareintensiver Systeme so strukturiert, dass im Ergebnis die geforderte Funktionalität mit der gewünschten Qualität bei Erfüllung der finanziellen Vorgaben im vereinbarten Zeitrahmen zur Verfügung steht (vgl. Abbildung 2.1).

Aber was ist Softwarearchitektur eigentlich? Was sind die Kernkonzepte in diesem so entscheidenden Teilgebiet des Software Engineering? Welche Vorgehensweisen und Ansätze gibt es für einen erfolgreichen Architekturentwurf?

Wie Software Engineering ist auch Softwarearchitektur ein junges Gebiet. Deshalb finden sich viele unterschiedliche Auffassungen zu den genannten Fragestellungen – und diese möchten wir hier nicht etwa abwerten oder schlicht als falsch disqualifizieren. Vielmehr möchten wir in diesem Kapitel unser grundlegendes Verständnis von Softwarearchitektur darlegen und so die Basis für die nachfolgenden Kapitel liefern.

Zunächst werden wir den Begriff des softwareintensiven Systems und den damit einhergehenden Zusammenhang mit der Softwarearchitektur vorstellen. Darauf aufbauend können wir dann die zentralen Grundbegriffe von Softwarearchitekturen vorstellen und definieren. Schließlich führen wir das grundlegende Vorgehen beim Architekturentwurf ein und stellen das Wechselspiel mit den anderen Disziplinen und Rollen vor.

2.1 Einbettung in den iSAQB®-Lehrplan

Nachfolgend finden Sie die Lernziele des Kapitels »Grundlagen von Softwarearchitekturen« aus dem iSAQB®-Lehrplan [isaqb-lehrplan].

2.1.1 Lernziele

- LZ 1-1:
Definitionen von Softwarearchitektur verstehen
- LZ 1-2:
Ziele und Nutzen von Softwarearchitektur verstehen und erläutern
- LZ 1-3:
Langfristige Auswirkungen von Softwarearchitektur kennen

- LZ 1–4:
Aufgaben und Verantwortung von Softwarearchitekt:innen verstehen
- LZ 1–5:
Abgrenzung zu anderen Architekturdomänen
- LZ 1–6:
Rolle von Softwarearchitekt:innen mit anderen Stakeholdern in Beziehung setzen¹

2.2 Softwareintensive Systeme und Softwarearchitekturen

Eine Softwarearchitektur manifestiert sich stets in dem zugehörigen System unabhängig davon, ob sie explizit entworfen wurde oder irgendwie gewachsen ist. Deshalb müssen wir zuerst ein klares Verständnis dafür entwickeln, was denn ein softwareintensives System überhaupt ist, bevor wir uns einer tiefer gehenden Betrachtung der Softwarearchitektur derartiger Systeme zuwenden können. Dazu werden wir in den folgenden Abschnitten zuerst den Begriff des softwareintensiven Systems präzisieren und dabei auch die unterschiedlichen Arten von softwareintensiven Systemen diskutieren. Auf dieser Basis können wir letztlich den Bezug zur Softwarearchitektur der betrachteten Systeme herstellen.

2.2.1 Was ist ein softwareintensives System?

Hier stellt sich zuerst die grundlegende Frage: Was ist überhaupt ein System? Eine Antwort auf diese Frage finden wir im IEEE-Standard 610.12-1990, dem »IEEE Standard Glossary of Software Engineering Terminology«. Dort ist ein System wie folgt definiert:

»**system.** A collection of components organized to accomplish a specific function or set of functions.«

([IEEE 610.12-1990], Seite 73)

Das charakterisiert recht treffend die wesentlichen Eigenschaften eines Systems, die auch intuitiv so zu erwarten sind. Ein System besteht aus Bausteinen bzw. Komponenten, wie z.B. Hardware-, Software- oder mechanischen Bausteinen. Bereits in diesem Systembegriff findet sich die Vorstellung wieder, dass Systeme in

1. Das Lernziel 1–7 »Bedeutung von Daten und Datenmodellen« ist in diesem Buch nach Kapitel 4 verschoben worden. Grund dafür ist, dass in dem entsprechenden Abschnitt auch der Entwurf von Datenmodellen diskutiert wird, was thematisch besser in das Kapitel 4 passt.

Bausteine zerlegt werden können. Darüber hinaus soll entsprechend der Definition ein System einem bestimmten Zweck dienen. Dies spiegelt das Grundverständnis des Ingenieurwesens wider, für den Menschen nutzbringende Dinge zu erschaffen (siehe NSPE Code of Ethics for Engineers [NSPE]).

Das zweite wesentliche Element in dem Ausdruck »softwareintensives System« ist der Begriff der Software selbst. Auch hier werden wir nach einem Blick in das IEEE Standard Glossary of Software Engineering Terminology fündig:

»**software.** Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system.«

([IEEE 610.12-1990], Seite 66)

Software ist dementsprechend mehr als nur eine Ansammlung von Programmdateien. Zur Software gehören auch weitere Prozeduren (z. B. Konfigurationsskripte), zugehörige Dokumentation (z. B. eine Architekturbeschreibung) und schließlich auch noch Daten (z. B. die initiale Befüllung einer Datenbank mit den notwendigen Metainformationen und Stammdaten).

Um nun den Begriff eines softwareintensiven Systems so zu formulieren, wie wir ihn in diesem Buch verwenden, müssen wir die zwei Definitionen zusammenbringen und dabei der Software die geforderte intensive Rolle im System zuordnen. Ein softwareintensives System ist demnach wie folgt definiert:

Ein **softwareintensives System** besteht aus einer Menge von Bausteinen, die so zusammengestellt sind, dass sie gemeinsam den Zweck des Systems erfüllen. Bausteine, die vollständig oder zu wesentlichen Teilen aus Software bestehen, übernehmen dabei essenzielle Aufgaben zur Erfüllung des Systemzwecks. Der Softwareanteil des Systems besteht dabei aus einer Menge von Programmen und weiteren Prozeduren und Daten sowie zugehöriger Dokumentation.

2.2.2 EXKURS: Ausprägungen von softwareintensiven Systemen

Es gibt unterschiedliche Ansätze, Software zu kategorisieren. Jeder davon stellt bestimmte Eigenschaften in den Vordergrund und ist dementsprechend auch nicht allgemeingültig. So unterscheidet das IEEE Standard Glossary of Software Engineering Terminology im Rahmen der Definition des Begriffs Software zwischen Anwendungssoftware, Unterstützungssoftware und Systemsoftware:

»**software.** ... See also: application software; support software; system software.«

([IEEE 610.12-1990], Seite 66)

Diese Unterscheidung ist abhängig vom Betrachtungskontext: Die Datenbank eines Versicherungssystems ist aus Sicht der Kunden Systemsoftware. Hingegen ist sie aus Sicht des Programmierers des Versicherungssystems Unterstützungssoftware. Oder betrachten wir einen Webbrowser: Aus Sicht eines PC-Nutzers, der im Internet surfen möchte, ist der Webbrowser Anwendungssoftware. Aus Sicht des Programmierers, der für den Webbrowser ein Plug-in erstellt, ist er Systemsoftware.

Eine andere häufig anzutreffende Unterscheidung ist die in Standardsoftware und Individualsoftware [Sie04]. Diese und andere Versuche, Software zu kategorisieren, beispielsweise bezüglich Größe oder Anwendungsdomäne, führen in der Konsequenz zu einer mehrdimensionalen Einteilung, wie Abbildung 2.2 zeigt. Dabei ist die Einteilung nicht immer eindeutig und meist abhängig vom Betrachter, wie im obigen Abschnitt dargestellt.

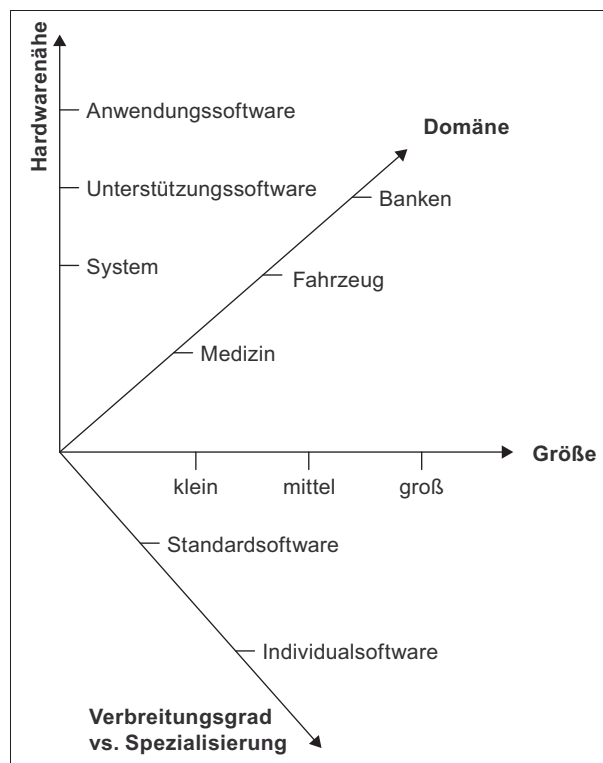


Abb. 2.2 Mehrdimensionale Kategorisierung von softwareintensiven Systemen

Für uns ist eine Kategorisierung von Software hilfreich, insoweit sie erste Rückschlüsse auf die Softwarearchitektur erlaubt. Idealerweise könnte für jede Softwarekategorie bereits eine Reihe von Architekturansätzen und architekturspezifischen Entwurfsproblemen vordefiniert sein. Diese Vordefi-

dition würde dann die gesammelten Entwurfserfahrungen der Softwarearchitekten widerspiegeln. Softwarearchitekten hätten so einen einfacheren und gesicherteren Start in die Entwurfsaufgabe.

Leider gibt es weder eine einheitliche Kategorisierung von Software noch eine zugeordnete und vollständige Sammlung des Entwurfserfahrungswissens. Es ist auch offen, ob es jemals möglich sein wird, in einem derartig dynamischen Umfeld wie in der Softwareindustrie eine solche Wissensbasis aufzubauen und zu nutzen. Deshalb muss sich heute jeder Softwarearchitekt und jede Softwareorganisation dieses selbst erarbeiten und aufbauen.

Nichtsdestotrotz sind der grundsätzliche Ansatz und das Ziel erstrebenswert. Deshalb wollen wir in diesem Buch beispielhaft eine einfache Kategorisierung vornehmen und ebenfalls beispielhaft sowie in Auszügen den Bezug zur Softwarearchitektur herstellen. Abbildung 2.3 zeigt eine mögliche Unterscheidung in die folgenden Kategorien:

- Bei **Informationssystemen** stehen die Verwaltung und die Verarbeitung von Informationen im Vordergrund. Große Datenmengen oder komplexe Datenstrukturen müssen verwaltet, bearbeitet, ausgewertet und berechnet werden. Dabei gilt es, unter Umständen mehrere Tausend Benutzer gleichzeitig und interaktiv zu bedienen. Beispiele für Informationssysteme sind das Kernversicherungsverwaltungssystem einer Versicherung, SAP-Systeme, CAD-Systeme, komplexe Simulationssysteme für Wettervorhersagen oder Simulationsberechnungen von Ingenieuren.
- **Eingebettete Systeme** beinhalten Software, die in physikalischen Gegenständen eingebettet ist. Unter starken Ressourceneinschränkungen der zur Verfügung stehenden Hardware realisieren sie daten- und funktionssicherheitskritische Aufgaben, die hohen funktionalen und qualitativen Ansprüchen gerecht werden müssen. Die Funktionen umfassen meist Regelungs-, Steuerungs- oder Kommunikationsfunktionen. Beispiele für eingebettete Systeme sind Waschmaschinen, Werkzeugmaschinen oder Produktionslinien in der Fertigungsindustrie, Funkzellen von Handynetzen, Airbag-Steuerungen oder Parkassistenten in Fahrzeugen.
- **Mobile Systeme** sind (semi-)autonome und personenspezifische Einheiten mit hohen Interaktionsanforderungen. Neben ihrer Eigenschaft, dass sie mobil sind, zeichnen sie sich insbesondere dadurch aus, dass sie lokale und ggf. auch (semi-)autonome Funktionen bereitstellen. Zusätzlich haben sie die Fähigkeit und teilweise auch die Notwendigkeit, mit zentralen, meist stationären Systemen zu interagieren, um sich abzugleichen oder Informationen und Aktionen mit anderen abzustimmen. Aufgrund der Mobilität ist die Verbindung aber nicht kontinuierlich verfü-

bar. Beispiele für derartige Systeme sind Smartphones, (semi-)autonome Transportroboter oder Sensor- und Aktuatorknoten von Ad-hoc-Netzwerken.

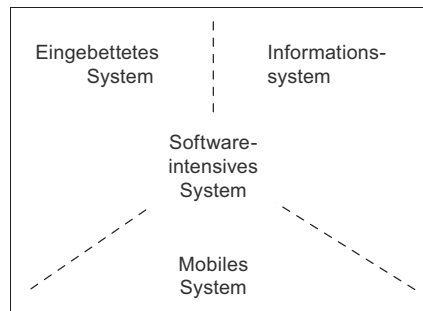


Abb. 2.3 Kategorien von softwareintensiven Systemen

Durch die zunehmend stärkere Vernetzung der Systeme gibt es softwareintensive Systeme, die nicht nur einer Kategorie zuzuordnen sind. So bindet ein Versicherungssystem über eine Anwendung auf einem Smartphone die Versicherungsmakler direkt in das System ein. Über Internetanbindung und Webbrowser können SAP-Auswertungsreports direkt im Fahrzeug analysiert werden. Und schon heute sind fast alle Produktionslinien einer Fabrik über das zugehörige Manufacturing Execution System an das SAP-System der Fabrik angeschlossen und umgekehrt.

Auch wenn viele der heutigen Systeme nicht eindeutig einer der oben dargestellten drei Kategorien zuzuordnen sind, so hat doch jedes System seinen Schwerpunkt. Im Zweifel ist jedem im Entwicklungsprojekt klar, ob das betrachtete System oder die anstehende Entwurfsaufgabe eher einem Informationssystem oder einem eingebetteten System zuzuordnen ist.

Dies ist für Softwarearchitekten von besonderer Bedeutung. Denn für jede dieser Kategorien gibt es bestimmte Grundmuster und -systematiken bezüglich Softwarearchitektur und Architekturentwurf. So findet man bei Informationssystemen meist eine Schichtenarchitektur vor, wie z.B. eine Aufteilung in Präsentations-, Anwendungs- und Datenhaltungsschicht. Bei eingebetteten Systemen findet man hingegen eher eine Architektur mit einer Reihe von zyklisch ausgeführten Prozessen, organisiert in Tasks, die zwar lose gekoppelt sind, trotzdem aber miteinander interagieren, z.B. über eine busbasierte Kommunikation. Bei mobilen Systemen hingegen hat man zwar ebenfalls eine Architektur mit aktiven Prozessen. Diese kommunizieren jedoch meist mittels Shared Memory, da sie auf einem Prozessor (ggf. Multicore-Prozessor) ausgeführt werden.

Darüber hinaus gibt es für jede Kategorie eine spezifische Menge von architekturrelevanten Entwurfsproblemen gerade im Hinblick auf spezifische, nicht funktionale Anforderungen. So muss man sich bei Informationssysteme-

men über die Art der Datenhaltung und der zugehörigen Transaktionssteuerung Gedanken machen. Hingegen ist bei eingebetteten Systemen eher die Frage des Scheduling der aktiven Prozesse oder der Kommunikationslast auf dem Netzwerk relevant. Bei den mobilen Geräten dagegen gilt es, auf dem Prozessor (auch bei einem Multicore-Prozessor) die Balance zu finden zwischen der geforderten hochwertigen, aber ressourcenintensiven grafischen Oberfläche und den hardwarenahen sensor- und aktuator-spezifischen Funktionen.

2.2.3 Bedeutung der Softwarearchitektur für ein softwareintensives System

Wie bereits dargestellt wurde, ist der Entwurf der Softwarearchitektur ein wichtiger und kritischer Schritt in der Softwareentwicklung. Die Softwarearchitektur hat unmittelbare Auswirkung auf die Parameter des in Abbildung 2.1 dargestellten magischen Vierecks: die gewünschte Funktionalität, die damit verbundenen Qualitätseigenschaften, der notwendige Aufwand zur Systemerstellung und die Zeitdauer bis zur Bereitstellung des Systems. Letztlich geht es um die Frage, wie große Systeme zu strukturieren sind, sodass die geforderten Parameter des magischen Vierecks erreicht werden können.

Aber hat denn jedes softwareintensive System überhaupt eine Softwarearchitektur? Ebenso wie auch Bauwerke, die ohne das Mitwirken eines Bauingenieurs entstanden sind, besitzt jedes softwareintensive System eine Architektur, auch wenn diese nicht explizit entworfen und umgesetzt wurde. Leider ist in Entwicklungsprojekten noch viel zu häufig zu beobachten, dass Softwarearchitektur nicht explizit entworfen wird. Die Konsequenzen sind oft folgeschwer.

Anforderungen an Software verändern sich während der Softwareentwicklung, aber auch über den ganzen Lebenszyklus hinweg, egal wie gut sie dokumentiert sind. Sich ändernde Anforderungen beeinflussen das Entwicklungsprojekt. So müssen beispielsweise Projektpläne geändert oder das Budget angepasst werden. Dies alles läuft allerdings ins Leere, wenn sich die bereits realisierten Softwareanteile den gewünschten Änderungen widersetzen. Eine gute Softwarearchitektur allerdings ermöglicht das leichte Ändern von bestehenden oder das einfache Einbringen von neuen Funktionalitäten, ohne die Qualitätseigenschaften der existierenden Software zu gefährden.

Softwarearchitektur ist somit von enormer Bedeutung für eine erfolgreiche Softwareentwicklung. Worauf aber sind diese Zusammenhänge begründet? Diese Frage wollen wir anhand von zwei Teilfragen diskutieren. Warum hat jede Software eine Architektur? Und warum ist die Softwarearchitektur mitentscheidend für eine erfolgreiche Softwareentwicklung?

Wesentlicher Bestandteil einer Softwarearchitektur ist die meist hierarchische Zerlegung des Systems in Teilsysteme oder Bausteine. Die Existenz einer solchen Zerlegungsstruktur ist auch bereits in der Natur von softwareintensiven Systemen enthalten. Wie in der Begriffsdefinition oben bereits aufgeführt, besteht ein

softwareintensives System aus einer Menge von Bausteinen, die entsprechend organisiert werden, um den Zweck des Systems zu erfüllen. Damit ist die Geburtsstunde der Architektur bereits mit dem Begriff eines Systems untrennbar verbunden. Somit hat jedes System und auch jedes softwareintensive System implizit oder explizit eine Architektur.

Diese inhärente Verzahnung, dass die Softwarearchitektur maßgeblich die Systemstruktur definiert und umgekehrt, ist auch der Grund dafür, dass Softwarearchitektur mitentscheidend für eine erfolgreiche Softwareentwicklung ist. Die Struktur von Bauwerken legt fest, welche Teile tragende Bestandteile sind und welche nicht. Will man an einem Gebäude etwas verändern, ohne dass dabei tragende Bestandteile betroffen sind, ist das in der Regel problemlos möglich. Müssen allerdings tragende Bestandteile verändert werden, so ist nicht absehbar, ob und wie das realisierbar ist und welche Kosten dadurch entstehen.

Analoges gilt für die Softwarearchitektur: Sie definiert über die Festlegung der Systemstruktur die tragenden Elemente in der Software. Damit gibt die Softwarearchitektur den Rahmen für zukünftige Veränderungen vor. Ergeben sich im Laufe des Entwicklungsprojekts oder darüber hinaus im Lebenszyklus der Software Änderungs- oder Neuerungswünsche, dann ist das problemlos möglich, solange die tragenden Grundpfeiler der Software erhalten bleiben. Andernfalls gilt das Gleiche wie bei Bauwerken: Kosten, Zeit und resultierende Qualität sind nur schwer abzuschätzen und stehen in der Regel nicht in einem vernünftigen Kosten-Nutzen-Verhältnis.

Dieser banale, aber fundamentale Zusammenhang zwischen dem softwareintensiven System, der inhärent existierenden Softwarearchitektur und den sich daraus ergebenden Einschränkungen bezüglich des magischen Vierecks der Softwareentwicklung begründen die enorme Bedeutung und Tragweite der Softwarearchitektur im Rahmen der Softwareentwicklung.

2.3 Grundlegende Konzepte von Softwarearchitekturen

Die Softwarearchitektur legt die wesentlichen Strukturen, die übergreifenden technischen Konzepte und Entwurfsentscheidungen eines Softwaresystems fest und bildet somit die Grundlage der gesamten Systementwicklung. Sie kann somit als Bauplan verstanden werden, der die Entwicklung, Auslieferung und den Betrieb komplexer und umfangreicher Software nachhaltig erleichtert. Eine Softwarearchitektur beschreibt keinen detaillierten Entwurf, vielmehr geht es darum, einen konstruktiven Lösungsweg ausgehend von den Anforderungen, die von außen an das System gestellt werden, hin zum fertig konstruierten System zu beschreiben, das dann beispielsweise in Form von Programmdateien vorliegt. Dabei sind nach Möglichkeit Begründungen für die Entwurfsentscheidung vorzuhalten, denn die Wahl einer bestimmten Architektur hat starken Einfluss auf die Qualitätseigenschaften und nicht funktionalen Eigenschaften, wie z.B. Wartbarkeit, Erweiterbarkeit oder Performance.

Trotz ihrer großen Bedeutung ist Softwarearchitektur immer noch eine junge Disziplin. Deshalb geben wir in diesem Kapitel eine allgemeine Einführung in die grundlegenden Konzepte von Softwarearchitekturen. Dazu definieren wir zuerst unser Verständnis des Begriffs Softwarearchitektur. Dieser basiert auf den Begriffen des Bausteins und der Schnittstelle, die ebenfalls eingeführt werden. Dies liefert uns die Grundlage, um darzustellen, wofür Softwarearchitekturen gut sind und welchen Nutzen sie generieren können. Schließlich runden wir dies mit Konzepten zur Beschreibung von Softwarearchitekturen ab.

2.3.1 Was ist eine Softwarearchitektur?

Eine einzelne, allgemein akzeptierte Definition für Softwarearchitektur gibt es nicht. Zur Einstimmung in den Wildwuchs der Softwarearchitekturdefinitionen hier eine der exotischeren Definitionen, die deshalb aber nicht weniger treffend ist:

»Software architecture is a framework for change.«

(Andreas Rausch, siehe auch [SEI Def])

Das Software Engineering Institute der Carnegie Mellon University (SEI) hat auf einer eigens dafür eingerichteten Website inzwischen über 150 Definitionen dazu gesammelt [SEI Def]. Bei den ausgearbeiteten Definitionen lässt sich zunehmend ein Konsens erkennen. Dieser Konsens, dem wir uns anschließen wollen, spiegelt sich auch in der Definition des ISO/IEC/IEEE-Standards 42010:2011, Systems and software engineering – Architecture description, wider:

»<system> fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution.«

[ISO/IEC/IEEE 42010:2011]

Statt eine grundsätzlich neue Definition für diesen zentralen Begriff im Rahmen des vorliegenden Buches einzuführen, orientieren wir uns an diesem Standard, ergänzen ihn aber an einigen wenigen Stellen, z. B. um den Begriff einer Schnittstelle oder den Bezug zur Entwicklungsorganisation, da diese aus unserer Sicht für Softwarearchitekturen wesentlich sind.

Die **Softwarearchitektur** definiert die grundlegenden Prinzipien und Regeln für die Organisation eines Systems sowie dessen Strukturierung in Bausteinen und Schnittstellen und deren Beziehungen zueinander wie auch zur Umgebung. Dadurch legt sie Richtlinien für den gesamten Systemlebenszyklus, angefangen bei Analyse über Entwurf und Implementierung bis zu Betrieb und Weiterentwicklung, wie auch für die Entwicklungs- und Betriebsorganisation fest.

Dieses Verständnis von Softwarearchitektur beinhaltet zwei wesentliche Aspekte. Zum einen gibt es den auf das System bezogenen konstruktiven Aspekt einer Softwarearchitektur, der den Aufbau bzw. die Zerlegung eines softwareintensiven Systems in Bausteine, Schnittstellen sowie deren Beziehungen zueinander und zur Umgebung beschreibt. Darüber hinaus enthält die Definition aber noch einen weiteren, auf die Vorgehensweise bezogenen Aspekt: Die Softwarearchitektur beeinflusst auch die Entwicklungsorganisation und den Systemlebenszyklus und gibt deshalb vor, welche Prinzipien und Regeln dabei zu beachten sind.

Daraus resultiert auch die Konsequenz, dass Softwarearchitekturziele durchaus auch längerfristige Ziele sein können, die über Projektziele und deren meist an die Projektlaufzeit gekoppelten zeitlichen Horizont hinausreichen. Softwarearchitektur kann somit auch eine Investition in den gesamten Systemlebenszyklus bedeuten, die sich unter Umständen erst nach Abschluss des zugehörigen Entwicklungsprojekts amortisiert.

2.3.2 Bausteine, Schnittstellen und Konfigurationen

Betrachten wir nun zunächst den systemkonstruktiven Aspekt der Softwarearchitektur. Dort werden die Begriffe Baustein, Schnittstelle und Beziehung zwischen diesen Elementen eingeführt. Was genau verstehen wir aber unter einer Schnittstelle oder einem Baustein und deren Beziehungen untereinander? Dieser Frage wollen wir in den folgenden Abschnitten auf den Grund gehen.

Schnittstelle und Baustein sind fundamentale Begriffe des Ingenieurwesens. In der Informatik erfreuen sie sich ebenfalls großer Beliebtheit, doch obgleich diese Begriffe dort fast täglich Verwendung finden, haben wir kein gemeinsames präzises Verständnis davon, was eine Schnittstelle oder ein Baustein ist.

Dass die Begriffe der Schnittstelle und des Bausteins nicht von allen gleich verstanden werden, merkt man recht schnell, wenn etwa an einem Systementwicklungsprojekt gleichzeitig Elektrotechniker, Maschinenbauer und Informatiker beteiligt sind. Unternehmen Sie doch einfach mal den Versuch, sich mit den Projektbeteiligten auf eine Schnittstelle für die zu entwickelnde Werkzeugmaschine zu einigen. Sie werden sehr schnell feststellen, dass hier fundamental andere Vorstellungen davon vorherrschen, was eine Schnittstelle ist und was nicht. Deshalb wollen wir hier den Begriff der Schnittstelle wie folgt definieren:

Eine **Schnittstelle** repräsentiert einen wohldefinierten Zugangspunkt zum System oder dessen Bausteinen. Dabei beschreibt eine Schnittstelle die Eigenschaften dieses Zugangspunkts, wie z.B. Attribute, Daten und Funktionen. Ziel ist es, diese Eigenschaften möglichst präzise mit allen notwendigen Aspekten zu definieren, wie z.B. Syntax, Datenstrukturen, funktionales Verhalten, Fehlerverhalten, nicht funktionale Eigenschaften, Nutzungsprotokoll der Schnittstelle, Technologien, Randbedingungen und Semantik.

Dieses Verständnis einer Schnittstelle zeigt uns deutlich, dass es sehr aufwendig sein kann, Schnittstellen in dieser vollständigen Form zu beschreiben. Programmiersprachen wie Java oder C# beinhalten Schnittstellenkonzepte. Mit diesen können in der Regel die Syntax, also der Name der Schnittstelle, die zur Verfügung gestellten Methoden und deren Argumente sowie die Rückgabewerte beschrieben werden. Die darüber hinausgehenden Aspekte einer Schnittstelle, wie z.B. funktionales Verhalten, müssen in zusätzlichen Dokumentationen abgeleitet werden.

Häufig findet man in den Programmen nur sehr unzureichende Schnittstellenbeschreibungen. Betrachtet man z.B. die Collection-Schnittstelle in Java, so ist diese sehr gut ausgearbeitet und dokumentiert. Allerdings werden Sie dort keine Aussagen zur Performance der Einfügeoperation eines Elements in eine Collection finden, wie z.B. die untere bzw. obere Schranke oder die durchschnittliche Dauer für das Einfügen eines Elements.

Für die Verwendung dieser Schnittstelle, insbesondere bei rechenintensiven Aufgaben über große Mengen von Elementen, kann diese Eigenschaft aber höchst relevant sein. Somit sind für die Entscheidung, ob die Collection von Java verwendet wird oder stattdessen eine andere Lösung gesucht werden muss, diese Eigenschaften maßgeblich.

In diesem speziellen Fall muss sich der Programmierer also der konkreten Implementierungen der Schnittstelle bewusst werden und daraus die geeignete auswählen. Er kann hier z.B. zwischen der ArrayList und der LinkedList wählen, die unterschiedliche Performance-Eigenschaften aufweisen. Somit kapselt die Schnittstelle nicht vollständig die Implementierung, obwohl es gemäß Definition ihre Aufgabe wäre.

Dieses kleine Beispiel zeigt auf, dass es im Allgemeinen unmöglich ist, vollständige Schnittstellenbeschreibungen zu erstellen. Vielmehr ist es Ermessenssache des Architekten, zu entscheiden, welche Aspekte einer Schnittstelle beschrieben werden müssen und welche im Zweifel vernachlässigt werden können. Trotzdem sollte man danach streben, für den konkreten Projektkontext möglichst – im Sinne der relevanten Eigenschaften – vollständige Schnittstellenbeschreibungen zu erstellen.

Nun können wir uns dem Begriff des Bausteins zuwenden. In der Literatur wird »Baustein« häufig synonym zu »Komponente« verwendet. Während »Baustein« jedoch als allgemeiner Begriff verstanden wird, sind »Komponenten« eine spezielle Ausprägung davon. Wir haben uns hier jedoch bewusst gegen den Begriff Komponente entschieden, da dieser oft individuell vorgebelegt ist. Manche verstehen darunter in erster Linie UML-Komponenten, andere hingegen Programmierkonstrukte wie Pakete oder JavaBeans.

Wir verwenden das umgangssprachliche Wort »Baustein«, um von der Vielzahl von Begriffen zu abstrahieren, die in Programmiersprachen, Modellierungsansätzen und Entwurfsmethoden als Bestandteile der Softwarearchitektur zum Einsatz kommen. Ein Baustein in unserem Sinne ist somit eine Abstraktion von speziellen Programmierkonstrukten oder Beschreibungselementen.

Der Baustein ist das zentrale Basiselement, aus dem die statische Struktur von Softwarearchitekturen aufgebaut ist. Er beinhaltet sämtliche Software- oder Implementierungsartefakte, die letztendlich Abstraktionen von Quellcode darstellen. Das reicht von kleinen Bausteinen (wie Funktionen oder Klassen) über mittelgroße Bausteine (wie Pakete oder Bibliotheken) bis zu großen Bausteinen (wie Subsysteme, Schichten oder Frameworks). Bausteine können sich somit in unterschiedlicher Form manifestieren. Abbildung 2.4 zeigt einige Beispiele für konkrete Ausprägungen von Bausteinen. Dabei ist zu beachten, dass Bausteine selbst wiederum aus Bausteinen bestehen können.

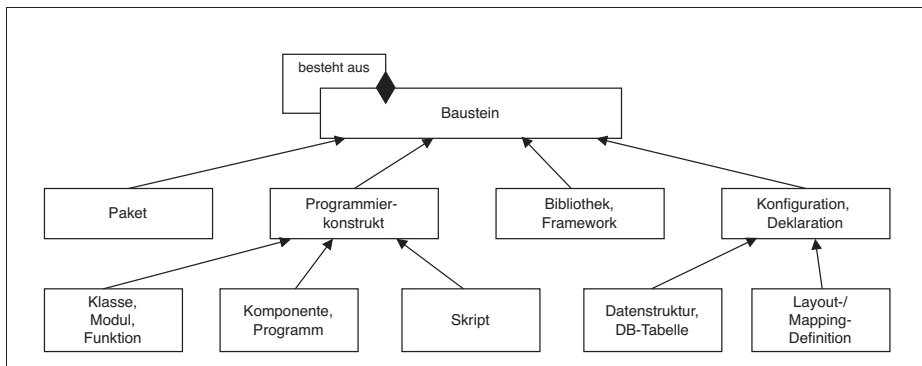


Abb. 2.4 Beispiele für Bausteine

Baustein ist somit einer der wichtigsten Begriffe in der Softwarearchitektur. Es bedarf dabei aber klarer Kriterien zur entsprechenden Abgrenzung: Was ist ein Baustein und was nicht? Unsere Definition von Baustein beinhaltet die drei folgenden wesentlichen Eigenschaften und subsumiert somit weitestgehend die in der Literatur vorherrschenden Definitionen ([Szy98], [D'SW98], [RQ++12]).

Ein **Baustein** bietet Schnittstellen an, die er im Sinne eines Vertrags garantiert. Diese Garantie gilt unter der Bedingung, dass die von ihm benötigten Schnittstellen im Rahmen einer entsprechenden Konfiguration zur Verfügung gestellt werden. (**Export und Import von Schnittstellen**)

Über die angebotenen und benötigten Schnittstellen kapselt der Baustein die Implementierung dieser Schnittstellen. Daher kann er durch andere Bausteine ersetzt werden, die dieselben Schnittstellen exportieren und ggf. importieren. (**Kapselung und Austauschbarkeit**)

Und schließlich sind Bausteine die Einheit der hierarchischen (De-)Komposition eines softwareintensiven Systems. Das heißt, ein (Super-)Baustein kann durch eine entsprechende Konfiguration von anderen (Sub-)Bausteinen und deren Beziehungen implementiert werden. Wir sagen dann auch, dieser (Super-)Baustein kapselt die (Sub-)Bausteine. Dabei kann diese Kapsel auch äußere Schnittstellen auf innere delegieren und umgekehrt. So werden die Beziehungen zwischen den Bausteinen definiert. (**Konfiguration und hierarchische (De-)Komposition**)

Beachten Sie: Aufgrund möglicher Seiteneffekte müssen bei einem Austausch von Bausteinen auch die importierten Schnittstellen berücksichtigt werden. Dementsprechend ist ein Baustein auch eine Einheit der Wiederverwendung. Weitere Annahmen, z.B. über die Umgebung der Bausteine – neben der Existenz der von dem Baustein importierten Schnittstellen –, sollten nicht existieren oder zumindest möglichst gering ausfallen und explizit dokumentiert sein.

Damit ist nicht nur der Begriff des Bausteins geklärt, sondern es sind auch die möglichen Beziehungen zwischen Bausteinen und Schnittstellen definiert. Die wesentlichen Eigenschaften von Bausteinen sind dadurch erfasst: Export und Import von Schnittstellen, Kapselung und Austauschbarkeit, Konfiguration und hierarchische (De-)Komposition. Abbildung 2.5 illustriert diese Begriffe und deren Zusammenhänge nochmals.

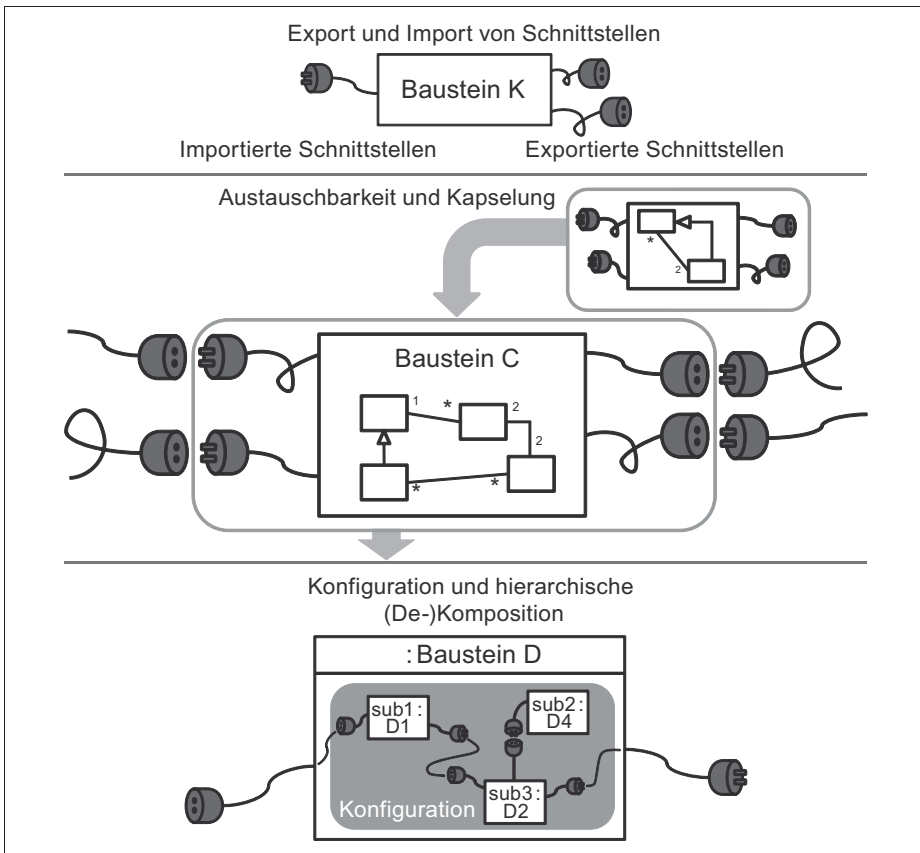


Abb. 2.5 Zusammenhang zwischen Bausteinen und Schnittstellen

Wie in Abbildung 2.6 dargestellt, können wir hierbei unterschiedliche Sichten auf einen Baustein unterscheiden:

- In der **Blackbox-Sicht** sieht man lediglich die von dem Baustein exportierten und importierten Schnittstellen. Das ist die Sicht des Bausteinnutzers. Diese Sicht respektiert das Geheimnisprinzip, d. h., sie verbirgt das (private) Innenleben des Bausteins. Die Blackbox-Sicht kann beispielsweise mit UML-Komponentendiagrammen beschrieben werden.
- Die **Greybox-Sicht** zeigt, welche zusätzlichen, meist eher technischen Schnittstellen von der Komponente benötigt werden. Zum Beispiel Konfigurationschnittstellen oder Schnittstellen zur Laufzeitumgebung, die erforderlich sind und verwendet werden [BW97]. Die Greybox-Sicht kann mit UML-Verteilungsdiagrammen beschrieben werden. Das ist die Sicht des Bausteinkonfigurators.
- Die **Whitebox-Sicht** (auch: Glassbox-Sicht) erlaubt einen Blick auf den inneren Aufbau des Bausteins, d. h. auf seine Zerlegung in die Konfiguration der Subbausteine oder eine andere Art der Implementierung. Dabei wird auch die Delegation seiner exportierten und importierten Schnittstellen auf das Innenleben des Bausteins sichtbar. Das ist die Sicht des Bausteinimplementierers. Die Whitebox-Sicht kann beispielsweise mit UML-Kompositionstrukturdiagrammen beschrieben werden.

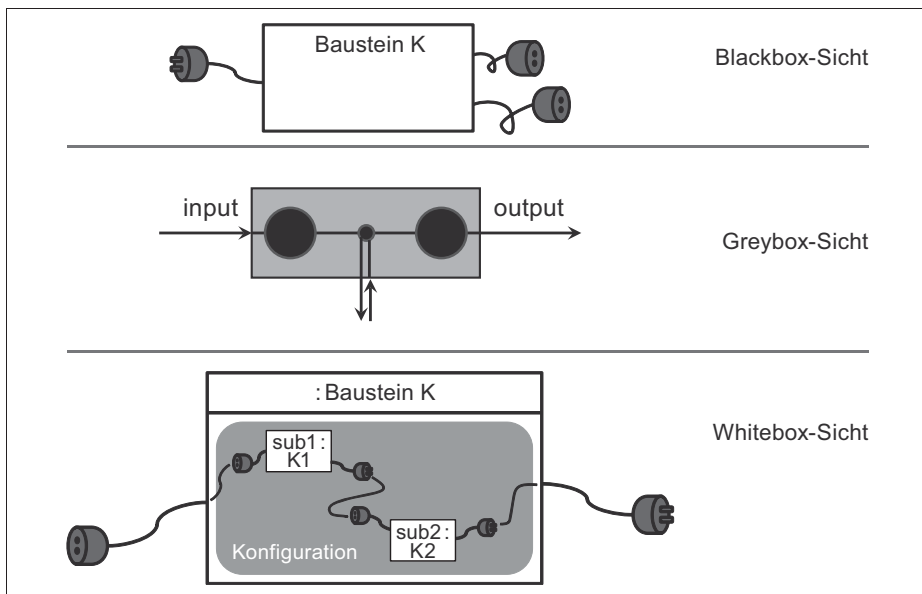


Abb. 2.6 Blackbox-, Greybox- und Whitebox-Sicht

Insbesondere im Zusammenspiel zwischen Blackbox und Whitebox wird die hierarchische (De-)Komposition einer Architektur und ihrer Bausteine deutlich. Wie Abbildung 2.7 zeigt, kann die Blackbox-Sicht eines Bausteins A in einer darunter liegenden Whitebox-Sicht hierarchisch dekomponiert werden. In dieser Whitebox-Sicht zerfällt der Baustein A in seiner Konfiguration in die Bestandteile B1, B2 und B3. Dabei ist zu beachten, dass die Bestandteile b1, b2 und b3 keine Bausteine sind. Sondern es sind Platzhalter (in der UML auch Parts genannt), die eine Instanz eines Bausteins nutzen. Wir nennen diese Platzhalter auch Bausteininstanzen oder, falls es nicht weiter von Belang ist, auch nur einfach Bausteine. Auch die Subbausteininstanzen b1 bis b3 sind Platzhalter für Bausteininstanzen in der Konfiguration des Bausteins A. Jedoch kann nicht jede beliebige Bausteininstanz an den Platzhalter b1, b2 oder b3 gesetzt werden, da die Platzhalter typisiert sind. Platzhalter sind vergleichbar mit Variablen: Sie haben einen Wert (= Bausteininstanz) und einen Typ (= Baustein). Somit kann an den Platzhalter »b1: Baustein B1« nur eine Instanz des Bausteins B1 gesetzt werden.

Da diese Bausteininstanz in der Konfiguration den Typ des Bausteins festlegt, gibt es für diesen Baustein wiederum eine Blackbox-Sicht.

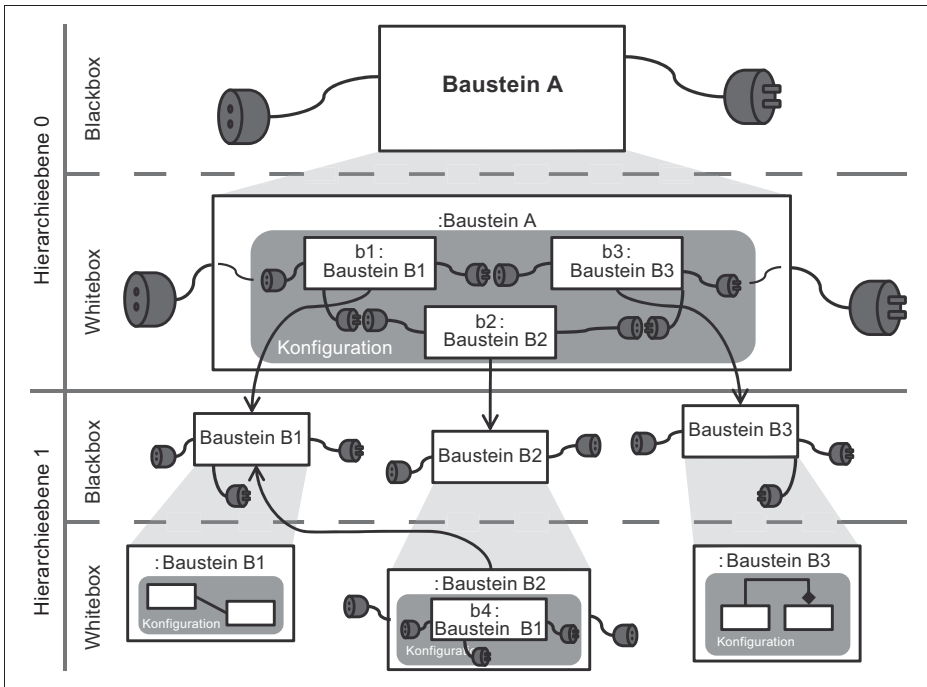


Abb. 2.7 Mit Blackbox- und Whitebox-Sichten hierarchisch (de-)komponieren

Die Darstellung zeigt aber nur eine Hierarchie im Sinne der Beschreibung der Architektur. Somit können Instanzen eines Bausteins in mehreren Konfigurationen anderer Bausteine, zu sehen in deren Whitebox-Sichten, verwendet werden, die

unter Umständen auf unterschiedlichen Hierarchieebenen anzuordnen sind. In Abbildung 2.7 taucht der Baustein B1 als Bausteininstanz in der Konfiguration der Whitebox-Sicht sowohl als Subbausteininstanz von A wie auch als Subbausteininstanz von B2 auf. B1 könnte ein XML-Parser-Baustein sein, der in den verschiedensten Bausteinen auf den unterschiedlichsten Ebenen verwendet wird.

Dabei ist zu beachten, dass diese hierarchische Zerlegung nicht nur für Bausteine, sondern auch für ihre Schnittstellen gilt. Das heißt, besitzt ein Baustein eine Schnittstelle zu einem anderen Baustein in der Greybox, so existiert auch eine entsprechende Schnittstelle auf der Blackbox- und Whitebox-Ebene und muss natürlich demgemäß implementiert werden, z.B. dadurch, dass die Schnittstelle auf einen Subbaustein delegiert wird.

Unabhängig davon sind Schnittstellen aber jeweils die Elemente, über die Bausteine verbunden werden. Sowohl der exportierende als auch der importierende Baustein müssen sich dabei an den Schnittstellenvertrag halten. Dieser wird in der Schnittstelle selbst definiert. Wer aber definiert die Schnittstelle? Hier gibt es unterschiedliche Möglichkeiten:

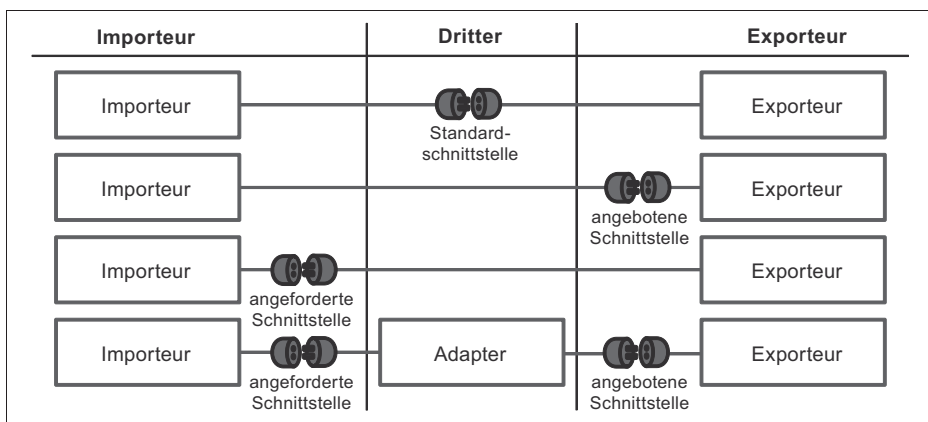


Abb. 2.8 Wer definiert die Schnittstelle und legt den Schnittstellenvertrag fest?

- **Standardschnittstelle:**
Diese Schnittstelle wird von außerhalb definiert. Sowohl der importierende als auch der exportierende Baustein halten sich daran.
- **Angebotene Schnittstelle:**
Hier definiert der Baustein, der die Schnittstelle exportiert, die Schnittstelle. Neben der Standardschnittstelle ist das die meistverwendete Schnittstellenart.
- **Angeforderte Schnittstelle:**
Hier definiert der Baustein, der die Schnittstelle importiert, die Schnittstelle. Gerade bei Frameworks ist diese Konstellation häufig anzutreffen. Über derartige Schnittstellen kann man Bausteine mit spezifischer Funktionalität in ein Programmgerüst einbringen.

■ Unabhängige Schnittstellen:

In diesem Fall haben der exportierende und der importierende Baustein jeweils eigene Schnittstellen definiert. Damit erhöht sich die Entkopplung der Bausteine. So können sie unabhängig voneinander entwickelt und getestet werden. Allerdings führt dies dazu, dass die Schnittstellen im Laufe der Zeit nicht mehr identisch sind. Deshalb muss ein Adapter dazwischengesetzt werden.

Jede Art von Schnittstellendefinition hat ihre spezifischen Eigenschaften und somit auch Vor- und Nachteile. So erhöht die letzte Art der Schnittstellendefinition die Entkopplung der Bausteine, allerdings muss das mit zusätzlichem Aufwand bei der Entwicklung und verlängerter Laufzeit bezahlt werden. Trotzdem kann diese Variante dauerhaft sinnvoll sein, z. B. bei Integrationsaufgaben.

Andererseits kann diese Variante auch nur übergangsweise genutzt werden, um einerseits maximale Parallelität beim Entwickeln sicherzustellen und andererseits inkonsistente Entwicklungen zu koppeln. In diesem Fall sollte man sich im Anschluss allerdings Zeit zum Refactoring und Redesign der Architektur nehmen, um die so entstandenen Adapter wieder auszubauen, und zwar mit neuen gemeinsamen Schnittstellenverträgen. Sonst wird aus der Übergangslösung unbeabsichtigt eine Dauerlösung.

2.3.3 Konzepte der Beschreibung von Softwarearchitekturen

Eine Architektur – egal ob explizit oder implizit entstanden – ist nur begrenzt hilfreich, wenn sie nicht dokumentiert ist. Nur eine angemessen dokumentierte Architektur kann nachhaltig kommuniziert, diskutiert und weiterentwickelt werden.

Dabei wird die Softwarearchitektur nicht nur mit anderen Architekten diskutiert. Vielmehr werden alle Aspekte der Softwarearchitektur unterschiedlichen Interessenvertretern (Stakeholder) vorgestellt, mit diesen diskutiert und gemeinsam weiterentwickelt. So können beispielsweise auch Kunden und Nutzer bei Architekturentscheidungen, die sie betreffen, mit einbezogen werden. Ebenso sollten Entwickler in die Diskussion involviert werden, insbesondere um umsetzungsrelevante Aspekte der Architektur zu kommunizieren und zu diskutieren.

Entsprechend dem Standard ISO/IEC/IEEE 42010:2011 [ISO/IEC/IEEE 42010:2011] beinhaltet eine Softwarearchitekturbeschreibung eine Menge von Artefakten, um eine Softwarearchitektur darzustellen. Dieser Standard definiert ein konzeptionelles Begriffsmodell für Architekturbeschreibungen. Abbildung 2.9 zeigt den für uns relevanten Ausschnitt aus dem Begriffsmodell.

Entscheidend jedoch ist, dass man sich regelmäßig für Refactoring und Re-design Zeit nimmt. Hier müssen bei der Projektkalkulation entsprechende Ressourcen vorgesehen werden.

4.4 Architekturzentrierte Entwicklungsansätze

In diesem Abschnitt werden einige architekturzentrierte Entwicklungsansätze und Konzepte vorgestellt, die ein Softwarearchitekt heute einsetzt, um Architekturen zu entwerfen und umzusetzen. Hier soll lediglich ein knapper Überblick von Entwicklungsansätzen, die architekturzentriert sind, vorgestellt werden. Die Liste ist keineswegs vollständig.

4.4.1 EXKURS: Domain-Driven Design

Beim domänengesteuerten Entwurf (engl. Domain-Driven Design, DDD) handelt es sich um eine Sammlung von Prinzipien und Mustern, die Entwicklern beim Entwurf von Objektsystemen helfen sollen. Der Begriff »Domain-Driven Design« wurde von Eric Evans geprägt. Auch für ein besseres Verständnis von Microservices ist Domain-Driven Design bedeutend, da es hilft, größere Systeme nach Fachlichkeiten zu strukturieren. Jedes Subsystem soll demnach eine eigene Einheit bilden.

4.4.1.1 Fachmodelle als Basis

Sie sollten Ihren Entwurf mit der Strukturierung der Fachdomäne beginnen [Sta24]. Entwerfen Sie ein projektweit akzeptiertes Domänenmodell, das auf rein fachlicher Basis strukturiert werden sollte. Dieses Modell verbessert die Kommunikation zwischen Fachexperten und Entwicklern und ermöglicht eine präzise Formulierung von Anforderungen. Durch die direkte Abbildung in Software lässt sich das Domänenmodell sehr leicht testen. Auf Grundlage dieses Modells entsteht eine gemeinsame, domänenspezifische Sprache, deren Elemente in ein Projektglossar aufgenommen werden sollten.

Diese allgemeingültige Sprache wird als »Ubiquitous Language« bezeichnet und ist ein zentrales Konzept des Domain-Driven Design. Sie sollte in allen Bereichen der Softwareentwicklung verwendet werden, d.h., alle Projektmitglieder nutzen dieselben Begriffe wie die Domänenexperten, z.B. im Quellcode oder in Datenbanken. Sie beschreibt die Fachlichkeit, die Elemente des Domänenmodells, die Klassen und Methoden etc.

Abbildung 4.5 zeigt die Bausteine eines Domänenmodells.

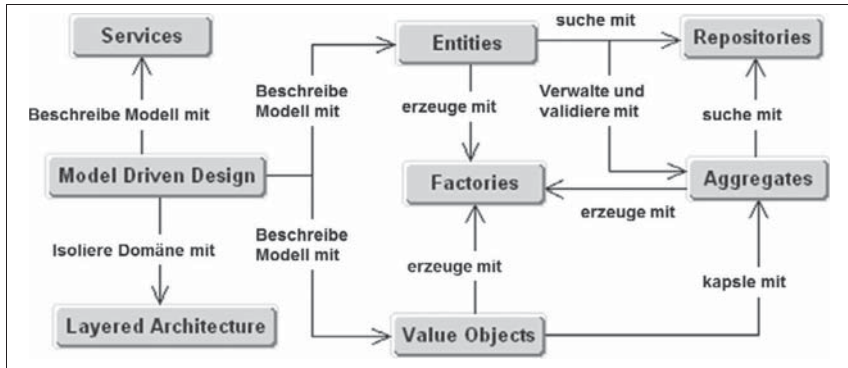


Abb. 4.5 Bausteine eines Domänenmodells

4.4.1.2 Systematische Verwaltung der Domänenobjekte

Entitäten stellen die Kernobjekte einer Fachdomäne dar und sind in der Regel persistent. Sie besitzen innerhalb der Domäne eine gleichbleibende Identität und einen klar definierten Lebenszyklus. Eine Entität ist ein »Ding« in Ihrem System. Es ist oft hilfreich, sich Entitäten in Form von Substantiven, wie z. B. Personen oder Orte, vorzustellen.

Die Wertobjekte (engl. Value Objects) beschreiben den Zustand anderer Objekte und verfügen über keine eigene Identität. Sie beschreiben einfach solche Objekte, die Identitäten haben. Sie können aus anderen Wertobjekten bestehen, aber niemals aus Entitäten. Wertobjekte sind im Gegensatz zu Entitäten unveränderlich.

Bei den Services handelt es sich um Operationen, die Abläufe oder Prozesse der Domäne darstellen, nicht von Entitäten wahrgenommen werden und für gewöhnlich über keinen eigenen Zustand verfügen. Die Ein- und Ausgaben dieser Operationen sind Entitäten, Aggregate oder Value Objects (also Domänenobjekte bzw. Fachobjekte).

Für die Verwaltung der Domänenobjekte schlägt Evans drei verschiedene Verwaltungsobjekte vor:

- **Aggregate:**
Ein Aggregat kapselt vernetzte Domänenobjekte und besitzt grundsätzlich genau eine Entität als Wurzelobjekt, die den einzigen Zugriff zum Aggregat darstellt. Externe Objekte dürfen nur Referenzen auf die Wurzelentität enthalten.
- **Fabrik:**
Fabriken (engl. Factories) kapseln die nicht triviale Konstruktion komplexer Objektstrukturen. Fabriken haben keinen Zugriff auf andere Schichten und dienen ausschließlich der Konstruktion von Fachobjekten.
- **Repository:**
Ein Repository bietet allen Arten von Objekten die Möglichkeit, Objektreferenzen anderer Objekte zu erhalten, und kapselt den Zugriff der Fachobjekte auf die darunter liegende Persistenztechnologie.

4.4.1.3 Strukturierung der Fachdomäne

Die Strukturierung der Fachdomäne kann üblicherweise nach zwei Aspekten erfolgen: nach Fachobjekten oder nach Benutzertransaktionen.

- Eine Zerlegung nach Fachobjekten ist dann sinnvoll, wenn
 - Wiederverwendung wichtig ist,
 - die Fachlogik komplex, umfangreich oder flexibel ist,
 - objektorientiertes Paradigma gut verstanden ist.

Diese Zerlegung entspricht im Großen und Ganzen der objektorientierten Zerlegung.

- Eine Strukturierung nach Benutzertransaktionen ist sinnvoll bei
 - simpler Datenbeschaffung und einfachen Operationen darauf,
 - Integration von Fremdsystemen,
 - einfacher oder wenig umfangreicher Fachlogik,
 - wenig Erfahrung mit objektorientierten Vorgehensweisen.

Eine Benutzertransaktion entspricht einer Aktion, die ein Benutzer des Systems ausführen kann, einschließlich aller systeminternen Operationen wie etwa der Prüfung der Eingabedaten.

Ein wichtiger Punkt ist vor allen Dingen die Bewahrung der konzeptionellen Integrität: Sie sollten möglichst alle Teile nach ähnlichen Aspekten zerlegen und dieses Konzept konsistent anwenden (und am besten dokumentieren).

4.4.1.4 Arten von Domänen

Im DDD kann ein System in folgende Domänen unterteilt werden:

- Core Domain
- Generic Domain
- Supporting Domain

Die Core Domain enthält die Kernfunktionalität des Systems und beschreibt den Grund für die Existenz des Systems. Sie sollte möglichst nur von den erfahrensten Entwicklern umgesetzt werden.

Die Generic Domain enthält Funktionalität, die wichtig für das Geschäft ist, aber nicht Teil der Core Domain ist, wie z.B. Rechnungen erstellen oder Briefe versenden. Sie kann hinzugekauft oder outgesourct werden.

Die Supporting Domain enthält unterstützende und untergeordnete Funktionalität und kann auch von weniger erfahrenen Entwicklern übernommen werden. Allerdings sollte sie streng von der Core Domain getrennt werden, z.B. durch einen Anti-Corruption Layer.

4.4.1.5 Integration von Domänen

Wie bereits erwähnt, sollte jedes Subsystem eine eigene Einheit bilden. Ein weiterer wichtiger Begriff im DDD ist der Bounded Context. Er hilft, die passende Granularität für Microservices zu finden. Jedes Modell hat einen Kontext, somit besteht eine anspruchsvolle Fachdomäne sicherlich aus mehreren Bounded Contexts.

Zur Integration von verschiedenen Domänen gibt es mehrere Möglichkeiten, wie z. B.:

- **Published Language:**
Zwischen zwei Domänen sollte es eine gemeinsame Sprache geben, über die sie interagieren können. Beispiele hierfür sind JSON oder XML.
- **Open Host Service:**
Dies ist eine klassische Interaktionsform im SOA-Umfeld. Eine Domäne spezifiziert ein Protokoll, über das andere Domänen diese nutzen können, z. B. RESTful Webservice.
- **Anti-Corruption Layer:**
Es werden Services einer anderen Domäne unter Verwendung einer Isolationsschicht genutzt.
- **Separate Ways:**
Zwei Domänen sind vollständig getrennt, eine Integration findet nicht statt.

4.4.2 EXKURS: Globale Analyse

Die globale Analyse ist ein systematischer Ansatz für den Entwurf von Architekturen. Dabei wird besonderer Wert auf die Einflussfaktoren auf eine Softwarearchitektur gelegt. Es wird postuliert, dass Einflussfaktoren Wechselwirkungen und Konflikte untereinander haben und deswegen als Gesamtheit betrachtet werden müssen. Zudem können Einflussfaktoren oder ihre Auswirkungen sich über die Zeit ändern, auch durch das eigene Handeln. Deswegen sieht die globale Analyse vor, alle Architekturentscheidungen »global« zu treffen, um die Einflussfaktoren in Einklang zu bringen und die Qualität des Systems zu steigern [HNS99].

4.4.3 EXKURS: Evolutionäre Architektur

Der Ansatz der evolutionären Architektur sieht vor, eine Architektur zu bauen, die »inkrementelle, geleitete Veränderungen über unterschiedliche Dimensionen hinweg unterstützt«. Dieser Ansatz stammt aus dem Buch »Building Evolutionary Architectures« von Neal Ford, Rebecca Parsons

und Patrick Kua [FPK17]. Ein zentrales Element ist die Definition einer Fitnessfunktion, die die Eignung der Architektur für die Aufgabenstellung möglichst objektiv beschreiben soll. Diese Funktion oder Funktionen sollen genutzt werden, um immer wieder die Eignung des Systems zu bestätigen oder Veränderungsbedarfe aufzudecken. Inkrementelle Veränderung hat einen hohen Stellenwert in einer evolutionären Architektur, da kleine Änderungen die Anpassbarkeit und damit die Evolutionsfähigkeit einer Architektur fördern. Beim Bau einer evolutionären Architektur sollen daher Continuous Delivery und Continuous Integration (siehe Abschnitt 4.9.3) genutzt werden, um inkrementelle Änderungen zu ermöglichen und eine Build Pipeline als Umfeld für die Ausführung von Fitnessfunktionen zur Verfügung zu haben [Par18].

4.4.3.1 Prinzipien

Im evolutionären Entwicklungsansatz werden Prinzipien postuliert, die das Erreichen einer evolutionären Architektur ermöglichen sollen.

Das erste dieser Prinzipien ist der sogenannte »last responsible moment«. Er besagt, dass eine Entscheidung zum spätmöglichen vertretbaren Zeitpunkt getroffen werden soll, sodass eine möglichst große Menge an Informationen vorliegt. Das Verzögern einer Entscheidung, solange es sinnvoll ist, erlaubt das Sammeln weiterer für die Entscheidung wichtiger Informationen und kann so die Qualität der Entscheidung verbessern. Außerdem werden technische Schulden durch frühe Entscheidungen, die im Nachgang angepasst werden müssen, verhindert.

Das zweite Prinzip besagt, dass Architekten und Entwickler die Entwicklungsfähigkeit des Systems fördern sollen. Dies soll erreicht werden, indem die Verantwortlichkeiten aus der Perspektive der Domäne oder der Fachexperten verteilt werden. Eine solche Verteilung macht es einfacher, die Software anzupassen, da die meisten Änderungswünsche einer Software ebenfalls durch die Domäne angestoßen werden. Die Änderbarkeit wird des Weiteren gefördert durch leichtgewichtige Werkzeuge und eine tiefgreifende Kenntnis von Datenhoheit und Lebenszyklen. Das Prinzip sieht ebenfalls vor, qualitativ hochwertigen und verständlichen Code zu erzeugen, da eine geringe Verständlichkeit zu längeren Änderungszeiten führt. Zuletzt sollen Abhängigkeiten durch den Fokus auf ihre Notwendigkeit bewusst gewählt werden. Durch die Vermeidung von unnötigen Abhängigkeiten sollen die beteiligten Teams einen möglichst hohen Grad an Eigenständigkeit erreichen und überflüssige Abstimmung und Kommunikation vermieden werden.

Das dritte Prinzip ist Postel's Law. Wir kennen es bereits aus Abschnitt 4.3.4.1. Für eine evolutionäre Architektur bekommt Postel's Law noch eine weitere Bedeutung. So gilt für das Senden von Informationen die

Faustregel, dass ein Schnittstellenpartner jede einmalig gesandte Information als verlässliche Schnittstelle ansieht. Eine Information zu veröffentlichen, kann folglich mit dem Eingehen einer vertraglichen Verpflichtung gleichgesetzt werden und ist für die Eigenständigkeit und infolgedessen Änderungsfreudigkeit der Architektur nicht förderlich. Insgesamt fördert das Befolgen des Robustheitsprinzips die Eigenständigkeit eines Systems und damit auch eine Evolutionsfähigkeit.

Prinzip Nummer vier sieht vor, eine Architektur hinsichtlich Testbarkeit zu optimieren. Ein testbares System weist eine bessere Modularität auf und dient als Sicherheitsnetz, insbesondere wenn Akzeptanztests eingesetzt werden. Dieses Sicherheitsnetz kann genutzt werden, um die Implikationen einer Änderung zu verstehen, ohne in einer produktiven Umgebung ein Risiko eingehen zu müssen. Schnittstellentests werden verwendet, um Abstimmungsaufwände zwischen Teams zu identifizieren.

Das letzte Prinzip ist Conway's Law oder die Arbeit um Conway's Law. Es besagt, dass die Struktur einer Software immer die Struktur der Kommunikation einer Organisation abbilden wird. Mangelnde Kommunikation ist ein Treiber für technische Komplexität und widerspricht daher einer evolutionären Architektur. Es gilt, wenn die Architektur und die Organisation (gemeint ist Kommunikation) nicht zusammenpassen, muss entweder die Architektur oder die Organisation sich ändern.

4.4.3.2 Fitnessfunktionen

Fitnessfunktionen haben ihren Ursprung eigentlich in der Bioinformatik und werden dort benutzt, um die Eignung eines Kandidaten für eine Aufgabenstellung zu bewerten. Kandidaten werden dabei durch evolutionäre Algorithmen vorgeschlagen.

In evolutionären Architekturen werden Fitnessfunktionen genutzt, um zu messen, ob eine Eigenschaft der Architektur im gewünschten Umfang vorhanden ist. Damit sind sie nicht nur in der Lage, die Eignung einer Lösung zu beschreiben, sondern auch Lösungen vergleichbar zu machen.

Die Erstellung von Fitnessfunktionen ist eine herausfordernde Tätigkeit, da für alle gewünschten Eigenschaften der Architektur ein messbares Merkmal gefunden werden muss, das die gewünschte Eigenschaft auf eine wirksame Art und Weise repräsentiert. In dieser Hinsicht sind Fitnessfunktionen und Szenarien (siehe Abschnitt 3.4) vergleichbar. Fitnessfunktionen können aber im Gegensatz zu Szenarien auch genutzt werden, um mehrere Eigenschaften gleichzeitig in einer sogenannten holistischen Fitnessfunktion zu messen.

Damit Fitnessfunktionen aussagekräftig bleiben, ist es notwendig, eine regelmäßige Überprüfung der Funktionen vorzunehmen. Die Funktionen sollten auf Aktualität, Vollständigkeit und Wirksamkeit geprüft werden.

Bei der Arbeit mit Fitnessfunktionen sollten Sie außerdem beachten, dass der Zeitpunkt der Messung in vielen Fällen Auswirkungen auf das Ergebnis haben kann und damit auch auf die Vergleichbarkeit von Messergebnissen. Es ist daher empfehlenswert, neben der eigentlichen Funktion auch zu beschreiben, unter welchen Umständen oder zu welchen Zeitpunkten eine Funktion angewandt wird.

Beispiele für Fitnessfunktionen können die zyklomatische Komplexität eines Moduls oder die Anzahl von externen Abhängigkeiten einer Komponente sein (siehe [Par18], [Ing18]).

4.4.4 EXKURS: Modellgetriebene Architektur

MDA steht für »Modellgetriebene Architektur« und bezeichnet ein Konzept zur Generierung von Anwendungen oder Anwendungsteilen aus (UML-)Modellen heraus. Es wurde durch die Object Management Group (OMG) definiert.

Bei der modellgetriebenen Softwareentwicklung (engl. Model Driven Software Development, MDSD) werden Softwarekomponenten automatisiert durch Transformationen aus Modellen generiert [RH06]. MDSD bezeichnet die Verwendung von Modellen und Generatoren zur Verbesserung der Softwareentwicklung.

Im Zentrum steht das Modell, das typischerweise mithilfe einer domänenspezifischen Sprache (engl. Domain Specific Language, DSL) formuliert wird. Die DSL kann entweder textuell oder grafisch sein. Eine DSL ist allerdings keine Pflicht. MDA, MDSD & Co. sind unabhängig von DSLs.

Um letzten Endes eine ausführbare Anwendung zu erhalten, gibt es zwei Alternativen. Bei der direkten Interpretation werden ausführbare Modelle durch eine virtuelle Maschine direkt interpretiert. Beispiel dafür wäre die ausführbare UML (engl. Executable UML) der OMG. Die zweite Möglichkeit ist der generative Ansatz, bei dem das Modell mittels einer oder mehrerer Transformationen in eine ausführbare Anwendung übersetzt wird.

Die Model Driven Architecture (MDA), wie sie die OMG definiert, ist im Grunde genommen nichts anderes als eine Spezialisierung des MDSD-Ansatzes und hat nichts mit Architektur zu tun. Während bei der modellgetriebenen Softwareentwicklung die Wahl der verwendeten Modellierungssprachen offengelassen wird und es keine Einschränkung hinsichtlich der Transformation in lauffähige Anwendungen gibt, hat die MDA hier konkretere Vorstellungen. Zum Beispiel sollte die zu verwendende DSL MDA-konform sein, d. h. mittels der MOF (Meta Object Facility) definiert werden. Die Meta Object Facility bildet das Metamodell. In der Praxis werden meistens UML-Profile verwendet.

Die plattformunabhängigen Aspekte werden im Rahmen des PIM (engl. Platform Independent Model) modelliert. Anschließend wird das