4 Prozess der Architektur- und Komponentenentwicklung

Matthias Riebisch

Die Entwicklung einer Architektur ist ein sehr komplexer Prozess, bei dem viele Tätigkeiten miteinander verwoben sind. Zahlreiche Aspekte greifen bei der Entscheidungsfindung ineinander, eine Reihe von Prinzipien ist gleichzeitig zu beachten. Dies ist eine der Ursachen dafür, dass dieser Prozess schwer verständlich ist. Häufig wird dann gesagt, dass viel Erfahrung notwendig ist, um Architekturen zu entwickeln. Erfahrungen allein sind jedoch nicht ausreichend für die Entwicklung von guten Software-Architekturen – Programmierer mit viel Erfahrung sind nicht automatisch gute Architekten. Ein Erlernen der grundlegenden Prinzipien und Zusammenhänge der Software-Architekturen kann und muss das Sammeln von Erfahrungen ergänzen. Außerdem sind Entscheidungen möglichst rational zu treffen, obwohl häufig deren Ziele und die Auswirkungen der Alternativen nicht vollständig klar sind. Hierbei soll dieses Kapitel unterstützen, indem der Architekturentwicklungs-Prozess zerlegt und seine grundlegenden Elemente stark vereinfacht und modellhaft beschrieben werden. Die Darstellung in diesem Kapitel entspricht damit nicht mehr dem Ablauf der Entwicklung einer Architektur in der Realität; durch kleine Beispiele soll der Bezug zu realen Prozessen aufrechterhalten bleiben.

4.1 Charakter des Prozesses

Betrachtet man den Ablauf des Prozesses der Architekturentwicklung auf der obersten Ebene, wird zunächst eine Folge von Iterationen und Inkrementen deutlich. In einer solchen Folge wird die Fertigstellung Stück für Stück und Stufe für Stufe vorgenommen, indem in jeder Iteration eine Teilaufgabe gelöst wird. Zur Bildung solcher Teilaufgaben werden die Prinzipien der Zerlegung (siehe Abschnitt **??**) angewendet, um die komplexen Aufgaben besser beherrschen zu können. Tätigkeiten bauen dabei aufeinander auf oder werden wiederholt. Zur Verbesserung des Verständnisses werden

sie hier untersucht. Die tatsächliche Folge von Aktivitäten hängt vom Einzelfall ab. Hier sollen deshalb die grundsätzlichen Muster solcher Folgen untersucht werden.

4.2 Aktivitäten innerhalb einer Iteration

Für das Verständnis des Prozesses ist es wichtig, welche Folge von Aktivitäten innerhalb einer Iteration abläuft und welche Tätigkeiten dabei ausgeführt werden. Jede Iteration soll zum Erfolg eines Entwicklungsprojekts beitragen, indem sie einen substantiellen, nachweisbaren Beitrag zur Verbesserung leistet, sei es die Realisierung einer Anforderung, die Verbesserung eines Qualitätsmerkmals der Architektur oder auch nur die Machbarkeit einer bestimmten technischen Lösung. Dazu müssen in jeder Iteration Aktivitäten durchgeführt werden, die sich in folgende Phasen gliedern lassen:

- Analyse der Ziele, Anforderungen, Mängel und Rahmenbedingungen,
- Bewertung von Alternativen und Behandlung der Risiken,
- Entscheidungen über Strategie, Vorgehen und Reihenfolge, der Entwicklung von Strukturen und der Umsetzung der Ziele in konkrete Lösungen, sowie
- Prüfung und Bewertung der Ergebnisse.

Diese Aktivitäten werden in den folgenden Abschnitten untersucht. Ihre Gliederung in vier Phasen zeigt eine starke Ähnlichkeit zum Spiralmodell von Boehm [**?**]. Die Aktivitäten der letzten beiden Phasen werden bei Boehm zur Entwicklung zusammengefasst. Bei Boehm erfolgt zusätzlich dazu eine Planung der nächsten Stufe vor Analyse der Ziele. Damit ergibt sich für eine Iteration die Folge von Aktivitäten wie in Abbildung **??** dargestellt.

Die Aktivitäten während einer Iteration werden in den folgenden Abschnitten so vorgestellt, dass ihre jeweiligen Gemeinsamkeiten sichtbar werden, um die Struktur der Iterationen zu verdeutlichen. In der Praxis unterscheiden sich die Aktivitäten je nach Ebene der Verfeinerung sowie nach Domäne. Bei eingebetteten Systemen müssen die möglichen Entscheidungen über Hardware-Lösungen stärker und früher berücksichtigt werden als bei betrieblichen Informationssystemen; für Komponenten einer Web-Applikation stehen andere Kriterien der Schnittstellengestaltung im Vordergrund als für Plugin-Komponenten eines mobilen Handheld-Kommunikationsgeräts. Die Wahl einer Leitidee stellt für die Architektur eines großen, eher homogenen Systems eine wichtige Entscheidung dar, weil sie für alle Bestandteile gültig sein soll. In anderen, eher heterogen aufgebauten Systemen wird für Subsysteme jeweils eine separate Entscheidung über Leitideen und Architekturstile getroffen.

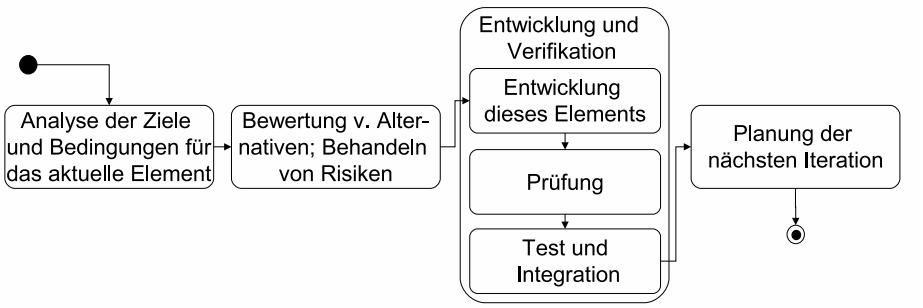

Abbildung 4.1: Folge von Aktivitäten während der Phasen einer Iteration

Analyse der Ziele und Bedingungen

Die bei der aktuell bearbeiteten Iteration zu erfüllenden Ziele werden von den jeweils zutreffenden Anforderungen bestimmt. Wird die Tätigkeit auf einer tieferen Verfeinerungsebene ausgeführt, ergeben sich viele der Ziele aus den Wirkungen der vorangegangenen Entscheidungen. Wie bereits in der Einleitung dargestellt, werden bei der Architekturentwicklung die folgenden Ziele verfolgt, die bei Entscheidungen als Präferenzen des Entscheiders auftreten:

- die funktionalen und qualitativen Ziele für das Produkt
- die organisatorischen und technologischen Rahmenbedingungen der Entwicklung
- die zu erwartende Variabilität der Ziele dieser beiden Kategorien.

Analyse der Ziele Bei Entscheidungen zwischen verschiedenen Alternativen stellen Anforderungen an das aktuell bearbeitete Element in gleicher Weise Ziele der Architekturentwicklung dar, wie die oben genannten. Gemeinsam treten diese Ziele als gewünschte Konsequenzen von Entscheidungen auf. Sie helfen dem Entwickler, aus der Menge möglicher Alternativen auszuwählen. Stehen die Ziele in Konflikt zueinander und / oder existiert keine Lösungsmöglichkeit, die alle Ziele erfüllt, muss eine Priorisierung durchgeführt werden, bei der ein Kompromiss zwischen den konkurrierenden Zielen festgelegt wird. Häufig müssen die Fakten dafür erst zusammengetragen werden. Dabei hilft vor allem die Beantwortung solcher Fragen wie:

- Welche Nutzerprofile bzw. Nutzungsszenarien gibt es?

- Wie wichtig sind bestimmte Ziele für die verschiedenen Nutzerprofile?
- Welchen Einfluss haben bestimmte Ziele auf die Architektur, insbesondere:
- Welches sind die höchsten Risiken?

Langfristige Ziele erfordern eine Planung der Entwicklung, bei der entsprechende Ziele verfolgt werden. Es handelt sich dabei häufig um Unternehmens-strategische und / oder betriebswirtschaftliche Ziele. Sie müssen auf oberen Verfeinerungsebenen einer Architektur berücksichtigt werden, weil sie viele nachfolgende Entscheidungen beeinflussen.

Angenommen es ist eine Architektur für ein betriebliches Informationssystem einer mittelständischen deutschen Firma zu entwickeln. Wenn Outsourcing und Unternehmensexpansion zu den strategischen Zielen dieses Unternehmens gehören, sollte die Architektur für Mehrsprachigkeit vorbereitet werden, auch wenn die erste Version dieses System zunächst in deutscher Sprache entwickelt wird, weil eine spätere Veränderung der Architektur sehr hohen Aufwand erfordern würde.

Ziele des Projektmanagements müssen bei Entscheidungen über die Architektur berücksichtigt werden, weil die Architektur unmittelbaren Einfluss auf die Möglichkeiten der Arbeitsteilung hat. Dies trifft für alle Verfeinerungsebenen zu.

Um auch diesen Zusammenhang zu verdeutlichen, soll angenommen werden, dass im Software-Zweig eines Logistikunternehmens nur wenige Entwickler qualifiziert sind, Problemlösungen mit Echtzeitanforderungen und Nebenläufigkeit zu erarbeiten. Aus diesem Grunde sollten die Echtzeitanforderungen auf möglichst wenige Module konzentriert werden, wenn ein eingebettetes Steuerungssystem für ein Fördersystem entwickelt wird.

Variabilitätsanalyse Variabilität als weiteres Ziel der Architekturentwicklung bezeichnet die Möglichkeit, Eigenschaften eines Systems mit geringem Aufwand zu ändern. Dies kann z.B. durch Veränderungen von Parametern oder Konfigurationen sowie durch Austausch von Komponenten erfolgen. Alle diese Möglichkeiten erfordern Vorkehrungen in der Architektur. Angenommen, Variabilität soll erreicht werden, indem eine Plugin-Schnittstelle einen einfachen Austausch von Komponenten gestattet. Dann muss diese Schnittstelle so gestaltet werden, dass sie sowohl die minimale als auch die maximale Funktionalität des betreffenden Systems ermöglicht.

Informationen über die Variabilität von Anforderungen und Zielen werden bei der üblicherweise bei der Domänenanalyse erhoben und häufig in Merkmalmodellen dargestellt. Merkmalmodelle wurden für die Methode FODA [**?**] entwickelt, einen neueren Stand beschreibt beispielsweise [**?**].

Bewertung von Alternativen und Behandlung der Risiken

Innerhalb jeder Iteration ist es zunächst notwendig, die Entwicklungsstrategie, das Vorgehen und die Reihenfolge zu planen. Bezogen auf die anstehenden Entscheidungen bedeutet dies, dass die Ziele geordnet und priorisiert werden, was in der Entscheidungstheorie als Bildung eines Zielsystems bezeichnet wird. Die Planung ist notwendig, weil nicht alle Aspekte einer Aufgabenstellung gleichzeitig beachtet, verfolgt und umgesetzt werden können. Für den Erfolg der Entwicklung spielt die Behandlung der Risiken die größte Rolle, deshalb werden die größten Risiken zuerst bearbeitet. Daraufhin werden dann Lösungsalternativen aufgestellt. Sie werden unter Berücksichtigung der in Abschnitt **??** vorgestellten Prinzipien bewertet und ausgewählt. Solche Entwurfsentscheidungen werden in Abschnitt **??** untersucht. Frühzeitige Entscheidungen über Lösungsalternativen wie beispielsweise über eine Leitidee der Architektur müssen dazu beitragen, die wichtigsten Risiken abzudecken. Entscheidungen über Lösungsalternativen auf tieferen Verfeinerungsebenen setzen Vorgaben durch die früheren Entscheidungen um.

Für spätere Änderungen ist es wichtig, dass Ziele, Risiken und Alternativen als Bestandteil der Entwurfsdokumentation beschrieben werden.

Entwicklung dieses Elements der Architektur

Bei dieser Tätigkeit wird der sichtbare Fortschritt der Architekturentwicklung erreicht. Zusätzlich zu den bereits besprochenen sind dabei Entwurfsentscheidungen zu treffen, die die taktische Umsetzung der vorher festgelegten Strategie darstellen, und die Ziele umsetzen, die bei der Analyse ermittelt wurden. Entscheidungen können sich auf die Veränderung von Komponenten und Strukturen beziehen, auf die Anwendung von Architekturstilen, Mustern, auf den Einsatz von fertigen Komponenten oder von weiteren Arbeitsmitteln aus dem Vorrat des Entwicklers. Der Abschnitt **??** stellt einige typische Fälle dar.

Die Veränderungen an Elementen der Architektur können vielfältig sein. Bei den Elementen kann es sich z.B. um Subsysteme oder Komponenten handeln. Je nach Verfeinerungsebene der gerade bearbeiteten Iteration und der für dieses Element gültigen Ziele kommen dabei unterschiedliche Arbeitstechniken zum Einsatz. Trotz deren Unterschiede weisen diese Techniken eine Reihe von Gemeinsamkeiten auf:

- Sie tragen immer dazu bei, die in der vorherigen Phase ermittelten Ziele umzusetzen und solche Entscheidungen zu treffen, die den Rahmenbedingungen entsprechen.
- Sie halten stets die in Abschnitt **??** dargestellten Entwurfsprinzipien ein.

- Sie greifen auf eines oder mehrere der in Abschnitt **??** genannten Konzepte, Muster, Lösungen oder Standards zurück. Entscheidungskriterien für die Auswahl sind deren positive und negative Konsequenzen im Vergleich zu den jeweiligen Zielen.

Trotz aller Unterschiede dieser Techniken lassen sich sechs grundlegenden Kategorien von Veränderungen (auch als *modular operators* bezeichnet [**?**, **?**]] an den bearbeiteten Elementen unterscheiden, die in Abbildung **??** dargestellt sind.

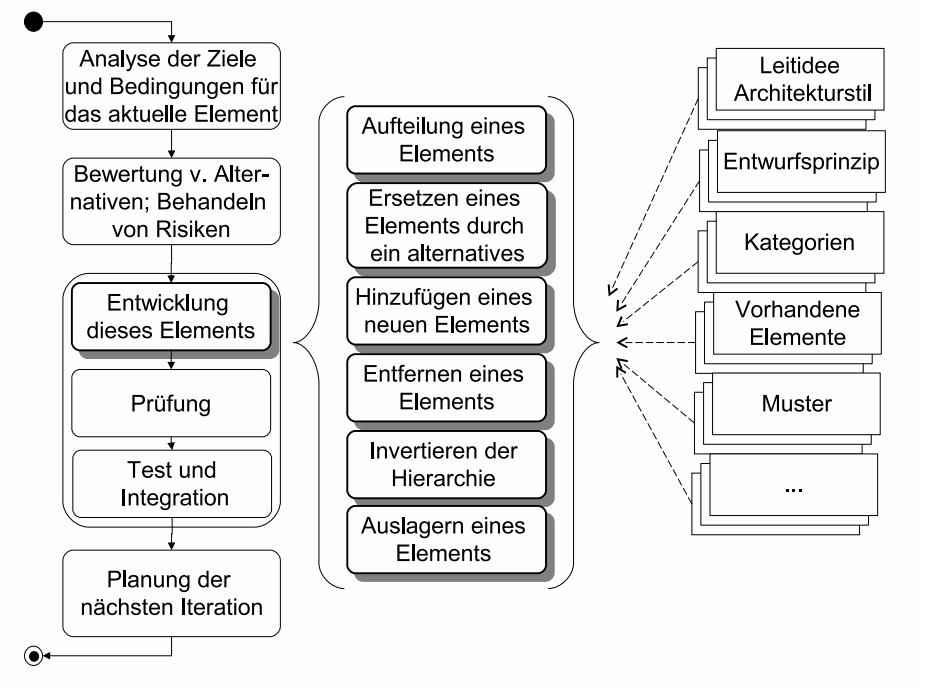

Abbildung 4.2: Schritte des eigentlichen Entwurfs, dargestellt als Teil einer Iteration

Viele dieser Kategorien bedürfen wohl keiner weiteren Erläuterung, auf die beiden untersten soll jedoch kurz eingegangen werden. Als *Invertieren der Hierarchie* wird eine Verschiebung eines Elements innerhalb einer Hierarchie nach oben bezeichnet, wobei dieses Element größere Bedeutung im betreffenden System erhält. Als *Auslagern eines Elements* wird das Extrahieren eines bisher enthaltenen Teils in ein neues Element – z.B. eine Komponente – bezeichnet, die dann z.B. in einem anderen System eingesetzt

werden kann. An die Stelle des extrahierten Teils wird eine Aktivierung dieser Komponente platziert, so dass die Funktionalität unverändert erhalten bleibt.

Prüfung, Test und Integration

Innerhalb jeder Iteration wird eine Prüfung der Entscheidungen und Veränderungen durchgeführt. Die Ziele werden zwar bereits während der Tätigkeiten der Entwicklung berücksichtigt, doch aufgrund der Vielzahl der Ziele, Bedingungen und Elemente ist nicht zu erwarten, dass sie alle erfüllt werden. Außerdem treten während der Entwicklungstätigkeiten neue Abhängigkeiten auf. Die Prüfung bezieht sich auf die

- Behandlung der Risiken,
- Erreichen der Ziele,
- Erfüllung der Entwurfsprinzipien,
- Einhalten des guten Stils – Vermeidung von Bad Smells,
- Konformität zu Vorgaben und Standards,
- Vollständigkeit,
- Widerspruchsfreiheit und
- Verständlichkeit der Ideen.

Die Prüfung kann durch einen Prototypen, ein Assessment oder andere Tätigkeiten erfolgen, wie in Kapitel **??** dargestellt. Zwei Typen von Assessments sind von Bedeutung:

- Prüfungen *durch andere Entwickler* mittels Architektur-Assessment oder Entwurfs-Review zeigen Mängel bezüglich solcher Merkmale, die für Veränderungen und Weiterentwicklung wichtig sind.
- Prüfungen *durch Abnehmer* wie nachfolgende Bearbeiter und Kunden zeigen, ob die spezifizierten Anforderungen erfüllt werden. Prüftechniken dafür sind beispielsweise der Walkthrough und die Prüfung anhand eines Prototypen.

Die Ergebnisse der Prüfung tragen zu den Zielen der nächsten Iteration bei, zum Beispiel durch Forderungen nach Überarbeitung und Veränderung eines vorhandenen Elements oder nach Ergänzung eines noch fehlenden Elements.

Aufgrund der komplexen Zusammenhänge liefert eine Prüfung der Ergebnisse der Architekturentwicklung zu wenige Informationen für eine *Bewertung des entstehenden Systems*. Deshalb muss eine Bewertung der Zielerreichung am realen Objekt erfolgen, nachdem die betreffenden Teile der Lösung realisiert und zusammengesetzt wurden. Frühzeitige Prüfungen konzentrieren sich auf besonders kritische Teile, da während der Architekturentwicklung viele Teile einer Lösung noch nicht realisiert sind.

Planung der nächsten Iteration

Teillösungen einer Iteration schaffen häufig neben einem Beitrag zum Ergebnis neue Fragestellungen, die in den folgenden Iterationen zu behandeln sind. Diese Fragestellungen tragen zur Erweiterung des Entscheidungsproblems bei. Die Planung muss die Ziele und Anforderungen für die nächste Iteration vorgeben. Bezogen auf das Entscheidungsproblem wird bei der Planung der für die nächste Iteration wesentliche Ausschnitt festgelegt. Für den Erfolg eines iterativ bearbeiteten Projekts sind genaue Vorgaben und Prüfungen des Fortschritts der einzelnen Iterationen besonders wichtig. Kunde und Management erhalten Informationen über den Stand der Entwicklung und die Erfüllung der gestellten Entwicklungsziele der Iterationen.

Der Umfang von Iterationen ist wesentlich für die Effektivität des Prozesses, denn die Anzahl der Iterationen hat großen Einfluss auf den Managementaufwand, und bei kleinen Iterationen ist die erforderliche Anzahl höher. Andererseits haben kleine und damit kurze Iterationen Vorteile für die Minimierung von Risiken und die Vermeidung von Fehlern durch frühzeitige Rückmeldung der Ergebnisse.

4.3 Entwurfsprinzipien

Bei den in Abschnitt **??** vorgestellten Entwurfstätigkeiten hat ein Entwickler bei Entscheidungen viele Ziele gleichzeitig zu berücksichtigen, bei denen er überdies noch die Risiken und Rahmenbedingungen beachten soll. Die hohe Zahl von Abhängigkeiten zwischen den Elementen trägt ebenfalls zur Komplexität der Entwurfstätigkeiten bei. In diesem Abschnitt werden die fundamentalen Prinzipien vorgestellt, die bei der Entwicklung einer Architektur helfen, die Komplexität zu beherrschen. Obwohl hier getrennt voneinander dargestellt, hängen die meisten eng zusammen.

Die Prinzipien hängen mit erfolgreichen Strategien der Problemlösung für komplexe Aufgabenstellungen zusammen. Sie basieren auf den kognitiven Fähigkeiten des Menschen, indem sie beispielsweise seine Fähigkeiten zu Assoziationen nutzen und sein beschränktes Kurzzeitgedächtnis entlasten. Die Prinzipien wirken, indem sie die Information verdichten und damit ihre Menge zu reduzieren. Sie sind zu einem großen Teil identisch mit den Grundprinzipien der Objektorientierten Entwicklung, die wiederum aus den Prinzipien der Modularisierung hervorgegangen sind. Sie eignen sich für unterschiedliche Grade an Abstraktion. Im Gegensatz zu den meisten im Rest dieses Kapitels genannten Entwicklungstätigkeiten, die zwar eng verbunden, aber sequentiell ausgeführt werden, sind die Entwurfsprinzipien bei der Architekturentwicklung *gleichzeitig* zu beachten.

4.3.1 Abstraktion

Die Bildung von Abstraktionen bewirkt, dass bei analytischen und konstruktiven Tätigkeiten Unwichtiges weggelassen und Gemeinsamkeiten zusammengefasst werden. Entscheidend für die Vereinfachungen ist der Standpunkt des Betrachters. Die Bildung von Abstraktionen stellt *das* wesentliche Prinzip dar, mit dem bei der Modellbildung Informationen auf die aktuell wesentlichen Aspekte reduziert werden.

In komplexen Aufgabenstellungen tritt eine Vielzahl von Abstraktionen auf. Zur besseren Beherrschung der Komplexität werden diese Abstraktionen deshalb gegliedert. Eine Gliederung nach Zusammengehörigkeit wird durch das Prinzip *Modularisierung* angestrebt, eine Gliederung in Ebenen durch *Hierarchische Dekomposition*, und eine Gliederung nach Eigenschaften und Aspekten durch *Separation of Concerns*. Diese Prinzipien wirken in den meisten praktischen Fällen bei der Gliederung von Abstraktionen eng zusammen.

4.3.2 Modularisierung

Das Prinzip der Modularisierung hilft ebenfalls dabei, die Komplexität von Entwicklungsaufgaben durch Zerlegung besser beherrschbar zu machen. Dazu werden logisch zusammengehörende Abstraktionen zu kleineren Einheiten zusammengefasst, wodurch sich die Anzahl der gleichzeitig zu bearbeitenden Elemente verringert. Das Prinzip *Divide et Impera* bezieht sich auf Elemente der Architektur, hier vorrangig auf Module.

Dieses Prinzip wird angewendet, indem Zusammengehörendes zu Modulen zusammengefasst und eng gekoppelt wird. *Hohe Kohäsion* wird dabei als wichtigstes Ziel der Modulbildung verfolgt. Module werden so aufgeteilt, dass sie untereinander möglichst wenig Abhängigkeiten aufweisen – *lose Kopplung* stellt ein ebenso wichtiges Ziel dar. Durch hohe Kohäsion wird erreicht, dass sich die Schnittstelle eines Moduls einfach gestalten lässt.

Eine geringe Kopplung zwischen den Modulen führt zu einer Verringerung der Komplexität bei der Bearbeitung eines Systems. Die Reduzierung der Abhängigkeiten vereinfacht Änderungen, beispielsweise durch Austausch und Kombination von Modulen. Eine Arbeitsteilung zwischen Entwicklern durch Parallelisierung von Aufgaben wird durch die reduzierten Abhängigkeiten zwischen Elementen ebenfalls vereinfacht. Die in Kapitel **??** dargestellte Bildung von Software-Kategorien unterstützt das Prinzip durch die Entkopplung von Modulen, gleichzeitig mit dem Prinzip der Separation of Concerns.

Für die Verringerung der Komplexität ist die Festlegung der richtigen Größe der Module – meist als *Granularität* bezeichnet – von großer Bedeutung. Ist die Modulgrösse zu gering, entstehen zu viele Module, zwischen

denen dann zu viele Abhängigkeiten bestehen. Sind die Module zu groß, ist ihre innere Struktur zu komplex und sie werden schwer handhabbar. In diesem Fall sind ihre Schnittstellen außerdem häufig komplex. Die optimale Granularität hängt stark von der Variabilität der Aufgabenstellung ab. Sich ändernde Teile einer Aufgabe sollten in getrennten Modulen untergebracht werden, wodurch kleinere Module entstehen. Die Variabilitätsanalyse (siehe Abschnitt **??**) hat in solchen Fällen starke Auswirkungen auf die Modulbildung. Die Granularität hängt aber auch vom Anwendungsgebiet ab, z.B. weil zwar manche Teilaufgaben einen großen Umfang haben und damit große Module erfordern. Ist die Variabilität solcher Module gering, wirkt sich ihre Größe nicht nachteilig aus.

4.3.3 Kapselung

Kapselung hilft dabei, implizite und damit schwer erkennbare Abhängigkeiten zu reduzieren; der Entwickler kann sich auf die sichtbaren Abhängigkeiten konzentrieren. Dies wird durch Abgrenzung des Verantwortungsbereichs eines Moduls erreicht. Eine gut geeignete Methode zum Identifizieren und Gestalten solcher Module ist die Class-Responsibility-Card- (CRC-)Methode [**?**].

Die englische Bezeichnung *Information Hiding* bezieht sich ebenfalls auf das Prinzip Kapselung.

Ursprünglich zielte dieses Prinzip in Verbindung mit der Modularisierung auf die Kapselung von Daten und deren Zusammenhängen, wie bereits von Parnas [**?**] zur Entkopplung von Modulen vorgeschlagen. Die objektorientierten Vorgehensweisen erweitern die Kapselung noch auf das Verhalten, wodurch mehrfach einsetzbare und stabile Komponenten geschaffen werden sollen.

Während des Entwurfs von Elementen spielt das Prinzip der Kapselung eine wichtige Rolle, insbesondere während der Tätigkeiten der Aufteilung eines Elements, des Hinzufügens eines neuen Elements sowie des Auslagern eines Elements (Abbildung **??**). Dabei werden Informationen über die geforderte Variabilität einbezogen, um zu analysieren, welche Details von außen nicht benötigt werden. Das Ziel der Kapselung besteht darin, so viele Details wie möglich zu verstecken. Modularisierung und Kapselung wirken dabei in engem Zusammenhang: während Modularisierung vor allem das Äußere einer Komponente und ihre Beziehungen zur Umwelt betrachtet, konzentriert die Kapselung die Betrachtung auf das Innere.

Das Prinzip der Kapselung und des Information Hiding steht allerdings oft im Konflikt zur Anforderung der Testbarkeit: Für eine effiziente Durchführung von Tests muss der innere Zustand eines Moduls beobachtet und möglichst beeinflusst werden können.

Noch schwerer wiegt der Konflikt mit dem Ziel der entkoppelten Behandlung von Ausnahmen. Reaktionen auf Ausnahmesituationen erfordern einen Überblick über die Situation im Umfeld. Informationen müssen dazu zugänglich sein. Eine ausschließlich lokale Behandlung von Ausnahmen innerhalb eines Moduls ist selten sinnvoll; die Forderung nach Vermeidung von Ausnahmesituationen ist unrealistisch. Ein Konzept zur Ausnahmebehandlung innerhalb einer Architektur erfordert grundlegende Entscheidungen über die für die Behandlung von Ausnahmen verantwortlichen Elemente, über das »Hochreichen« von Fehlern aus tieferen Schichten und darüber, welche Resultate einer Komponente über Argumente kommuniziert werden und welche über Ausnahmen. Für ein Beispiel zur Gestaltung der Ausnahmebehandlung sei auf die Methode Quasar [**?**] verwiesen.

4.3.4 Hierarchische Dekomposition

Das Prinzip der Hierarchischen Dekomposition trägt zur Beherrschung der Komplexität bei, indem die geschaffenen Abstraktionen in Form einer Rangfolge gegliedert werden. Abstraktionen des gleichen Rangs werden jeweils zu einer *Abstraktionsebene* zusammengefasst.

Zwei Arten der Gliederung können anhand der bestimmenden Beziehungen unterschieden werden, die Aggregation mit *Teil-von*-Beziehungen sowie die Generalisierung mit *ist-ein*-Beziehungen. In beiden Arten von Hierarchien werden Elemente umso weiter oben eingeordnet, je höher ihr Abstraktionsgrad ist.

Hierarchien der *Aggregation* werden bei der Zerlegung von Systemen angewandt, sie spielen deshalb für Software-Architekturen vor allem auf oberen Verfeinerungsebenen eine wichtige Rolle. Aufgaben werden bei der Entwicklung eines Systems in Teilaufgaben zerlegt, denen dann Subsysteme, Komponenten und Teile des Systems zugeordnet werden. Die Beziehungen in einer solchen Hierarchie geben Wechselwirkungen und Abhängigkeiten an. Hierarchien dieses Typs sind darüber hinaus häufig bei der organisatorischen Gliederung sozialer Systeme anzutreffen.

Hierarchien der *Generalisierung* und Spezialisierung klassifizieren Elemente nach ihren Eigenschaften, indem die allgemeineren Eigenschaften in der Hierarchie oben angeordnet werden. Nachgeordnete Elemente übernehmen alle Eigenschaften der übergeordneten, abstrakteren Elemente, und fügen weitere Eigenschaften im Sinne einer Spezialisierung hinzu. Ein allgemeineres Element kann bei der Verwendung immer durch ein spezialisiertes Element ersetzt werden.

Solche Hierarchien helfen in der Architekturentwicklung bei der Unterscheidung von Wichtigem und Unwichtigem, von gemeinsamen und spezifischen Eigenschaften. Eine solche Unterscheidung bereitet die Bildung wiederverwendbarer Komponenten vor, so dass Gemeinsames mehrfach ge-

nutzt werden kann. Auch Arbeitsteilung und Spezialisierung wird durch Verallgemeinerung gefördert. Werden Elementgrenzen horizontal durch solche Hierarchien gelegt, entstehen Schnittstellen, die Veränderungen durch einen einfachen Austausch von Elementen ermöglichen. Auf diesem Prinzip basiert der Architekturstil *Layers*, der zum Arbeitsmittel-Vorrat des Entwicklers gehört (siehe Abschnitt **??**).

Beide Arten der Hierarchien stehen bei der Architekturentwicklung in enger Wechselwirkung mit anderen Prinzipien. Üblicherweise werden nach einer Trennung in Software-Kategorien (Kapitel **??**) die Elemente jeder Kategorie schrittweise verfeinert. Bei der Festlegung von Schnittstellen muss das Prinzip Modularisierung beachtet werden, um Variabilität zu erreichen.

4.3.5 Separation of Concerns

Dieses Prinzip, das sich in etwa mit Trennung von Zuständigkeiten übersetzen lässt, zielt ebenfalls auf eine Gleiderung von Abstraktionen. Das Prinzip wirkt sich auf die Vereinfachung von Änderungen aus, indem für die Lösung *einer* Aufgabe genau *ein* Element eines Systems zuständig gemacht wird. Ändert sich die Aufgabe, ist ein einfacher Austausch des betroffenen Elements möglich, ohne umfangreiche Auswirkungen auf andere Bereiche des Systems. Abhängigkeiten werden dadurch außerdem leichter erkennbar und damit steuerbar.

Dieses Prinzip muss bei der Implementierung manchmal verletzt werden, weil aus Effizienzgründen oder wegen Beschränkungen der Programmiersprache mehrere Aspekte im Code einer Komponente zu implementieren sind – das Resultat wird im Englischen als *Tangled Code* bezeichnet. In der Architektur sollte – von gut begründeten Ausnahmen abgesehen – eine strikte Trennung eingehalten werden.

Bei der Trennung von Zuständigkeiten werden im wesentlichen zwei Wege verfolgt, die Zerlegung nach Software-Kategorien und die Trennung zwecks Arbeitsteilung und Spezialisierung der Tätigkeitsprofile. Die Zerlegung nach Software-Kategorien hat vor allem die Steuerung von Abhängigkeiten und damit geringen Aufwand bei späteren Änderungen zum Ziel. Sie wird in Kapitel **??** behandelt.

Beim zweiten Weg werden die Parallelisierung von Aufgaben und die Aufteilung gemäß Qualifikationsprofilen ermöglicht, indem der Arbeitsteilung und Spezialisierung der Entwickler als Kriterium der Zerlegung angewendet wird. Durch Trennung von Aspekten wie Datenspeicherung und Anwendungslogik in separate Elemente entstehen weitgehend entkoppelte Aufgaben. Ein auf Datenbank-Anbindung spezialisierter Entwickler bekommt eine für ihn geeignete Aufgabe, bei der er sich wenig um Aspekte der Anwendungsdomäne z.B. der Materialwirtschaft kümmern muss. Durch die

damit erreichte Aufgaben- und Arbeitsteilung wird vor allem das Projektmanagement unterstützt.

4.3.6 Einheitlichkeit

Dieses Prinzip unterstützt ebenfalls die Beherrschung der Komplexität, indem es Strukturen, Schemata, Muster, Vorgehensweisen und Entwurfsentscheidungen durchgehend und einheitlich anwendet. Bereits getroffene Entscheidungen werden beibehalten und Konzepte gestärkt, statt sie durch punktuelle Ausnahmen auszuhöhlen. Speziallösungen sollen so vermieden und die Vielfalt der angewandten Konzepte verringert werden. Speziallösungen und «Tricks«, die zwar funktionieren, jedoch schwer zu verstehen sind, werden verhindert. Speziallösungen sind nur dann zulässig, wenn ihre Unverzichtbarkeit nachgewiesen wird. Bloße Eignung als Problemlösung stellt keine Rechtfertigung für eine Speziallösungen dar.

Entwickler können durch Anwendung dieses Prinzips Entwurfsentscheidungen und Lösungsideen leichter verstehen: Durch Verringerung der Vielgestaltigkeit der Lösung bewirkt es eine Reduzierung der Komplexität der Lösung. Der Nutzen dieses Prinzips steht in engem Zusammenhang mit Spezialisierung udn realer Arbeitsteilung der Entwickler. Er wird vor allem bei der Kommunikation von Ideen sichtbar. Beim Fehlen realer Aufgabenteilung – etwa wenn Architektur und Implementierung von ein und derselben Person entwickelt werden – ist dieses Prinzip nur sehr schwer durchsetzbar. Existiert eine reale Spezialisierung, wird auch die Entwicklung von einheitlichen, durchgängigen Lösungen gefördert, weil Spezialwissen und ein Lösungsvorrat an Architekturen herausgebildet werden kann.

Bei der Umsetzung dieses Prinzips sind häufig Kompromisse notwendig. In der Praxis der inkrementell-iterativen Entwicklung von Architekturen kommt es fast zwangsläufig zu Stilbrüchen innerhalb von Lösungen. Bei Änderungen einer Architektur, zum Beispiel auf einen neuen Architekturstil, ist aus Zeit-, Risiko- und Budget-Gründen selten eine komplette Überarbeitung des ganzen Systems möglich. Für welche Teile der Aufwand für ein Reengineering zur Umstellung auf den neuen Architekturstil geleistet werden sollte, muss anhand einer Risikoanalyse und einer Kosten-Nutzen-Analyse entschieden werden.

Bei der Implementierung einer Architektur kommt es zu Verletzungen dieses Prinzips, wenn Entwickler wegen besonderen Anforderungen, Optimierungsbestrebungen, Unkenntnis der Prinzipien oder auch nur wegen fehlendem Problembewusstsein Speziallösungen schaffen. Information, Motivation und Kontrolle müssen dann die Akzeptanz durchgängiger Lösungen unterstützen. Ein aufeinander abgestimmtes Zusammenwirken von Architektur-Reviews unter Beteiligung der betroffenen Entwickler, von klarer Information über Entwurfsentscheidungen und Architekturstile sowie von

flächendeckenden Code-Inspektionen hat sich hierbei als erfolgreich erwiesen. Eine «diktatorische« Durchsetzung der Architektur oder extrem detaillierte Vorgaben für Entwickler stellen im allgemeinen keine geeignete Lösung dar. Der Bedarf nach Speziallösungen sollte in die Verbesserung der Architektur einfließen, um deren Reifegrad zu erhöhen.

Der Ausdruck Konzeptionelle Integrität [**?**] bezeichnet ebenfalls dieses Prinzip, betont aber den Aspekt der Widerspruchsfreiheit etwas stärker.

4.4 Entwicklungstätigkeiten

Als Inhalt der dritten Phase einer Iteration «Entwicklung dieses Elements« wurden in Abbildung **??** (Seite **??**) Kategorien von Veränderungen genannt, die teilweise den Arbeitsmittel-Vorrat des Entwicklers nutzen. Zur Konkretisierung werden hier typische Aktivitäten etwas genauer beschrieben, die auf den meisten Verfeinerungsebenen und für Systeme der meisten Domänen auftreten.

Aufstellen einer Leitidee

Bei einer Leitidee handelt es sich um einen Architekturstil, der die Gesamtarchitektur eines Systems oder eines wesentlichen Systemelements bestimmt. Dieser Architekturstil sollte sich auch für alle untergeordneten Elemente eignen, wie vom Prinzip Einheitlichkeit (siehe Abschnitt **??**) gefordert. Diese Tätigkeit tritt folglich meist bei frühen Iterationen auf, wenn Elemente der oberen Verfeinerungsebenen bearbeitet werden. Die Menge der Entscheidungsalternativen wird durch die Architekturstile (siehe Abschnitt **??**) gebildet, von denen einige unter Berücksichtigung der Ziele und Umweltbedingungen bereits ausgeschlossen werden können. Zu berücksichtigen sind z.B. die eingesetzten Software-Produkte wie Middleware und Werkzeuge, die Hardware-Plattform, Unternehmensstandards bezüglich IT-Betrieb und nicht zuletzt die Kenntnisse des Projekt-Teams. Das Treffen der Entscheidung erfolgt nach dem in Abschnitt **??** beschriebenen Vorgehen; die Alternativen und die Entscheidung werden sofort dokumentiert. Der gewählte Architekturstil wird dann im weiteren Verlauf der betreffenden Iteration weiter ausgearbeitet – z.B. zu einem Komponentenmodell – und geprüft. Sollte sich zeigen, dass dieser die Ziele nicht erfüllen kann, muss die Entscheidung revidiert werden.

Für jedes im Verlauf der Verfeinerung zu schaffende Element ist mit der Leitidee die Entscheidung über den Architekturstil bereits getroffen. Sollten spezielle Bedingungen für eines der untergeordneten Elemente wirklich einen abweichenden Architekturstil fordern, so ist der Entscheidungsprozess identisch zum gerade angegebenen. Die Menge möglicher Alternativen

ist in diesem Fall durch die früheren Entscheidungen und durch übergeordnete Elemente stärker eingeschränkt.

Bilden eines funktionalen Elements

Das Bilden von Architekturelementen wie Subsystemen, Komponenten oder Module aufgrund von funktionalen Anforderung stellt eine der grundlegenden Entwurfstätigkeiten dar. Der Entwickler definiert ein solches Element, indem er eine Abstraktion bildet, die die betreffende funktionale Anforderung erfüllt, wie z.B. die Abarbeitung eines Workflows, das Anlegen oder das Konvertieren eines Datensatzes oder die Suche eines Elements in einer Liste.

Für dieses Element legt der Entwickler die zu erfüllenden Aufgaben fest. Dazu ordnet er diesem Element einige der aus der darüber liegenden Verfeinerungsebene stammenden funktionalen Anforderungen zu. Die nichtfunktionalen Anforderungen werden in einer folgenden Aktivität berücksichtigt. Ausgehend von dieser Zuordnung legt er dann Dienste und Leistungen fest, die dieses Element erbringen soll. Im nächsten Schritt entwickelt der Entwickler die Schnittstelle des Elements, zu dem neben den angebotenen Diensten auch die vom Element benutzten Dienste sowie für Test und Ausnahmebehandlung zusätzlich erforderliche Dienste gehören. Außerdem ist die Variabilität der Einsatzbedingungen des Elements zu beachten. Die Entscheidungen und die entwickelten Lösungen werden umgehend dokumentiert. Während der Tätigkeit prüft der Entwickler, ob alle Ziele und Anforderungen für dieses Element erfüllt sind.

Diese Tätigkeit wird auf auf allen Ebenen der Verfeinerung durchgeführt. Dabei stehen vor allem die Prinzipien der Abstraktion, der Modularisierung, der Kapselung und der Separation of Concerns im Vordergrund. Außerdem ist auf die Trennung der Software-Kategorien (siehe Kapitel **??**) zu achten. Bei der Einbindung existierender Teile in die Lösung wird außerdem das Prinzip der Hierarchischen Dekomposition angewandt. Während der Bildung von Elementen der Lösung ist auch für eine Schaffung von Arbeitsteilung und Spezialisierung zu sorgen, wodurch die Grundlagen für ein erfolgreiches Projektmanagement während der Implementierung dieser Elemente geschafften werden.

Diese Tätigkeit entspricht dem ersten Schritt «Umsetzung funktionaler Anforderungen« der Methode von Bosch [**?**]. Die beiden weiteren Schritte dieser Methode werden in den folgenden Abschnitten vorgestellt.

Transformation qualitativer Anforderungen in funktionale Elemente

Nachdem alle funktionalen Anforderungen zugeordnet wurden, muss als nächstes die Erfüllung der nicht-funktionalen, also der qualitativen Anfor-

derungen erfolgen. Häufige Anforderungen beziehen sich auf Zeitverhalten, Verfügbarkeit, Robustheit und Portabilität.

Bei der Untersuchung der qualitativen Ziele und Vorgaben spielen häufig die Mängel darüber liegender Verfeinerungsebenen oder Iterationen für die Entscheidung eine Rolle. Der Entwickler stellt mögliche technische Lösungen auf, die mit funktionalen Mitteln die qualitativen Anforderungen erfüllen. Beispielsweise kann zur Lösung von Zeitproblemen ein lokaler Zwischenspeicher auf einem Client eingesetzt werden, um die Häufigkeit der Übertragung von Stammdaten in einem verteilten System zu verringern. Zur Verbesserung der Benutzbarkeit kann z.B. die Einführung persönlicher Einstellungen für jeden Nutzer einen Beitrag leisten.

Diese Aktivität stellt nach dem Vorgehen von Bosch [**?**] den zweiten von drei Schritten dar. In der Praxis sind die Schritte allerdings nicht immer so ideal zu trennen.

Die Vorschläge für geeignete Alternativen können wieder aus dem Arbeitsmittel-Vorrat stammen, wie z.B. Architekturstile und –muster. Dabei wird das in Abschnitt **??** beschriebene Vorgehen der Auswahl durchgeführt. Bei der Schaffung neuer funktionaler Elemente muss die Trennung der verschiedenen Kategorien von Software beachtet werden (siehe Kapitel **??**).

Implementierung der verbleibenden qualitativen Anforderungen

Eine Reihe qualitativer Anforderungen kann nicht durch Hinzufügen *eines* zusätzlichen funktionalen Elements erfüllt werden, sondern erfordert Änderungen an einer Reihe von Elementen. Dies ist beispielsweise häufig für Anforderungen bezüglich Zeitverhalten, Robustheit und Wartbarkeit der Fall. Dann muss die Implementierung der betreffenden Elemente überarbeitet werden. Architekturentwicklung und Implementierung sind in einem solchen Fall eng gekoppelt. In der Praxis führt diese Kopplung häufig dazu, dass Architekturaspekte zu wenig beachtet werden. Ein zunehmender Verlust an Architekturqualität (im Englischen als Architectural Decay bezeichnet) ist eine häufige Folge. Das Prinzip Separation of Concerns muss deshalb besonders beachtet werden.

In der bereits im vorherigen Abschnitt genannten Methode von Bosch ist diese Aktivität als dritter Schritt enthalten. Auch bei dieser Aktivität können als Problemlösung oft Elemente aus dem Arbeitsmittel-Vorrat ausgewählt werden, wie Architektur- und Entwurfsmuster.

Angenommen es muss das Zeitverhalten einer Methode verbessert werden, die Datenelemente in einer geschachtelten Objektstruktur sucht, und funktionale Lösungen führen nicht zum geforderten Zeitverhalten, so wird eine Optimierung des zugrunde liegenden Suchalgorithmus vorgenommen. Bleibt diese Optimierung auf die betreffende Methode beschränkt, wird das Prinzip Separation of Concerns nicht verletzt.

Risiko-Behandlung

Die Behandlung von Risiken stellt außer für die Architekturentwicklung auch für das Projektmanagement ein wichtiges Ziel dar, weil Risiken unbekannte Aufwände zur Problemlösung erfordern. Behandlung des einem Architekturelement zugeordneten Risikos heißt hier, dass dieses Risiko durch Einfügen zusätzlicher funktionaler Elemente, durch Ändern eines funktionalen Elements oder dessen Veränderung behoben wird.

Besteht das Risiko zum Beispiel in der unbekannten oder mangelhaften Zuverlässigkeit und Robustheit einer extern entwickelten Komponente, so muss diese möglicherweise durch eine selbst entwickelte Komponente ersetzt werden.

Risiko-Weitergabe

Neben der direkten Behandlung von Risiken in einem Element besteht auch die Möglichkeit der Weitergabe und Aufteilung an weitere, durch Verfeinerung entstandene Elemente – häufig als Risiko-Vererbung bezeichnet. Im Zuge der Verfeinerung werden alle qualitativen Anforderungen, Ziele und Risiken auf die neuen Elemente aufgeteilt. Dabei reicht es aus, dass alle verfeinerten Elemente *in der Gesamtheit* die gleichen Risiken zugeordnet bekommen wie das *eine* vorher betrachtete Element. Dazu wäre es ausreichend, wenn mindestens eines der verfeinerten Elemente das volle Risiko übernimmt[1].

Ist zum Beispiel das Risiko einer gestörten Datenübertragung für ein verteiltes Informationssystem zu behandeln, so kann es vom Gesamtsystem an die für die Übertragung genutzte Middleware weitergeben werden, die es durch ein Übertragungsprotokoll mit Sicherung, Quittierung und Wiederholung behandelt.

Verfeinern und Beschreiben der Lösung

Vom Umfang und der Bedeutung – verglichen mit der Gesamtheit der Tätigkeiten bei der Entwicklung einer Software-Architektur – stellt die Verfeinerung und Beschreibung der erstellten Architektur einen der wesentlichsten Beiträge zum Ergebnis dar. Die Bedeutung dieser Aktivität wird erst bei realer Arbeitsteilung und bei der Notwendigkeit der Prüfung einer Architektur als Zwischenergebnis deutlich – in vielen anderen Situationen ist die Tätigkeit der so genannten «Dokumentation« zu Unrecht wenig geachtet.

Bei der Verfeinerung werden alle die Informationen zur Beschreibung hinzugefügt, die für die Verständlichkeit, für die Realisierung, für die Prü-

[1] Diese Regel der Risiko-Vererbung findet sich in verschiedenen Vorgaben zum Projektmanagement wieder, wie zum Beispiel im V-Modell [**?**], wo sie als »Vererbung von Kritikalitäten« bezeichnet wird.

fung sowie für spätere Änderungen benötigt werden. Dazu gehören Beschreibungen für Alternativen, Entscheidungen, für die Struktur der Lösung sowie für weitere notwendige Sichten. Insbesondere Informationen zu angewendeten Prinzipien sowie zum Verhalten sind für das Verständnis durch andere wichtig.

Eine Architekturbeschreibung muss vom Umfang her zwei gegenläufige Ziele erfüllen. Einerseits soll sie möglichst umfangreich sein, um viele Aspekte darzustellen und genaue Informationen zu liefern. Andererseits soll sie einen geringen Umfang haben, um einfach erstellbar, prüfbar und veränderbar zu sein. Kompromisse sind insbesondere bei Entscheidungen zu benötigten Sichten [**?**], zum Detaillierungsgrad und zum Grad an Redundanz notwendig.

Die Prüfung dieser Beschreibungen erfolgt nicht nur hinsichtlich Konsistenz und Vollständigkeit, sondern auch auf Verständlichkeit für ihre Nutzer, d.h. andere Entwickler. Für weitere Ausführungen zur Prüfung von Architekturdokumenten sei auf Kapitel **??** hingewiesen; weitere Ausführungen zur Beschreibung von Architekturen sind in Abschnitt **??** und im Anhang **??** enthalten.

4.5 Entwurfsentscheidungen

Entwurfsentscheidungen zur Architekturentwicklung sind sehr komplex: viele Ziele werden gleichzeitig verfolgt, die sich teilweise widersprechen und teilweise vage sind. Die Konsequenzen von Entscheidungen sind oft schwer abschätzbar. Zu einem ingenieurmäßigen Vorgehen gehört, dass der Entwickler den sich daraus ergebenden Entscheidungsspielraum erfasst, geeignete Alternativen aufstellt, diese bewertet und daraufhin eine rational begründete Entscheidung trifft.

In anderen Disziplinen wie z.B. dem Maschinenbau ist die Menge möglicher Alternativen durch Material- und Werkzeugeigenschaften sowie Technologie viel stärker eingeschränkt als bei der Software-Entwicklung. Als Alternativen für Architekturentscheidungen ist nahezu alles vom Menschen überhaupt vorstellbare auch möglich.

Häufig werden Entscheidungen unbewusst getroffen. Lösungen werden gestaltet, »... weil man das so macht.«, wie eine typische Auskunft auf Nachfragen lautet. Manchmal ist Entwicklern nicht einmal die Tatsache der Entscheidungsfindung klar, sie führen keine Alternativenbewertung und keine Dokumentation durch. Doch gerade dokumentierte Entscheidungen vereinfachen spätere Änderungen, weil frühere Alternativen wiederholt geprüft werden können.

Die Mittel der Entscheidungstheorie [**?**] bieten eine gute Basis, rationale und damit gut fundierte Entscheidungen zu treffen. Die Entscheidungsfin-

dung ist innerhalb der Iterationen in den Phasen »Bewertung von Alternativen und Behandlung der Risiken» sowie »Entwicklung dieses Elements» enthalten (siehe Abbildung **??** auf Seite **??**). Abbildung **??** zeigt ebenfalls eine einzelne Iteration, hebt aber die Entscheidung hervor. Auf letztere beziehen sich auch die folgenden Abschnitte.

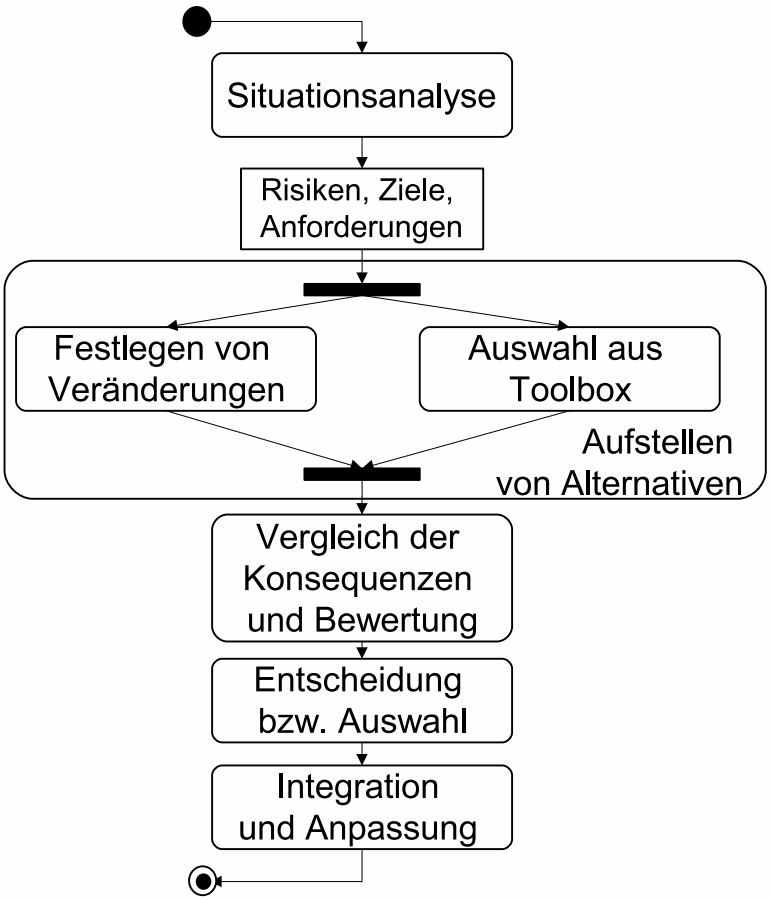

Abbildung 4.3: Aufstellen und Bewerten von Alternativen im Entscheidungsprozess

Aufstellen von Alternativen

In den meisten Fällen stößt der Entwickler bereits bei der Analyse der Situation auf geeignete Lösungsalternativen, die zu seinem Erfahrungsvor-

rat an Problemlösungs-Wissen gehören. Weitere Alternativen stehen im Arbeitsmittel-Vorrat (siehe Abschnitt **??**) zur Verfügung, z.B. Guidelines zur Problemlösung, Architekturmuster, Architekturstile oder vorhandene wiederverwendbare Module.

Bei komplexen Aufgabenstellungen kann auch die Methode *Goal-Question-Metric* GQM [**?**] [**?**] zur strukturierten Suche nach Lösungsalternativen angewendet werden. Diese Methode dient der strukturierten Problemanalyse. Die Problembeschreibung `Goal` wird anhand von Fragestellungen `Question` präzisiert, um geeignete Maßnahmen `Metric` festzulegen. Im häufigsten Anwendungsgebiet dieser Methode, dem Software-Qualitätsmanagement, wird bei `Metric` eine Qualitäts-relevante Maßnahme festgelegt. Hier wird statt dessen eine Architektur-relevante Maßnahme bestimmt. Hier ein Beispiel:

Goal-Question-Metric an einem einfachen Beispiel `Goal`: Änderungshäufigkeit von Klassen hinter einer Schnittstelle verringern
`Question`: Wie können diese Klassen von Änderungen unbeeinflusst gehalten werden?
`Metric`: Kapselung der Klassen
`Question`: Welche Maßnahmen der Kapselung sind für die Klassen hinter dieser Schnittstelle geeignet
`Metric`: Anwendung des Entwurfsmusters Fassade [**?**]

Insbesondere bei einer Bearbeitung im Team wird durch das Stellen geeigneter Fragen die Fokussierung auf das Wesentliche erleichtert und die Aufstellung von Varianten für Entwurfsentscheidungen vorangetrieben. Dabei wird ein inkrementelles Verfeinern der Fragestellung vorgenommen.

Hier sei noch einmal wiederholt, wie wichtig die Dokumentation aller Lösungsalternativen ist, um bei späteren Änderungen der Situation auf andere Alternativen zurückgreifen zu können oder diese weiter zu entwickeln.

Vergleich der Konsequenzen und Bewertung der Lösungsalternativen

Die Bewertung der Lösungsalternativen erfolgt durch Vergleich ihrer Konsequenzen, die sich auf die gewünschten Merkmale positiv oder negativ auswirken. Beispiele für negative Auswirkungen sind die Beeinträchtigung des Zeitverhaltens durch ein Entwurfsmuster oder die Beeinträchtigung der Testbarkeit durch einen Architekturstil.

Entscheidung

Die Entscheidung erfolgt anhand einer Gegenüberstellung der Ziele mit den positiven und negativen Konsequenzen der Alternativen. Die verschiedenen Ziele bilden ein System, in dem sie nach Standpunkt des Betrachters unterschiedliche Wichtungen erhalten. Vom unternehmerischen Standpunkt aus sind Risiken, funktionale und Zuverlässigkeits-Kriterien sowie die Vertrautheit mit einer Technologie häufig wichtiger als Fragen der Einheitlich-

keit und Verträglichkeit von Architekturmaßnahmen, die jedoch für Entwickler größere Bedeutung haben. Aus Sicht des Projektmanagements sind Risiken, Aufwände, Abhängigkeiten zwischen Arbeitspaketen und Vorhandensein von Know-How besonders wichtig. Es erfolgt eine Priorisierung der Ziele, wobei meist die Interessen des Kunden bzw. Auftraggebers höher als die der Entwickler und des Projektmanagements gewichtet werden.

Häufig fehlen wichtige Informationen zur Entscheidungsfindung und müssen erst ermittelt werden. Dies trifft insbesondere auf Konsequenzen von Alternativen zu. Aussagen zur Realisierbarkeit erfordern beispielsweise häufig eine prototypische Implementierung. Wie hoch der Aufwand für die Entscheidungsfindung getrieben wird, hängt vor allem von den jeweiligen Risiken ab.

Die Entscheidung ist ebenfalls zu dokumentieren, damit sie später nachvollzogen und gegebenenfalls revidiert werden kann. Fast jedes Vorgehensmodell enthält Vorschriften für die Dokumentation, z.B. schreibt das V-Modell [**?**] die Dokumentation von Entwurfsentscheidungen im Abschnitt `Lösungsvorschläge` des Produkts `SW-Architektur` vor.

Anpassung

Die festgelegte Alternative wird in der Phase »Entwicklung dieses Elements« der betreffenden Iteration umgesetzt. Je nach Typ der Alternative sind in den meisten Fällen noch Nacharbeiten daran erforderlich. Wiederverwendete Lösungselemente aus anderen Projekten sind meist anzupassen. Architekturmuster oder -stile sind beispielsweise noch zu konkretisieren und umzusetzen.

Weitere Arten von Entscheidungen

Scoping-Entscheidungen Beim Scoping wird darüber entschieden, ob und mit welchem Aufwand etwas entwickelt werden soll und welche der Anforderungen umgesetzt werden. Solche Entscheidungen sind bei Neuentwicklungen genauso zu treffen wie bei Weiterentwicklungen. Sie erfordern eine Konfliktlösung zwischen Zielen bezüglich Termin, Umfang, Qualität und Kosten. Projekt- und Unternehmens-strategische Ziele sowie die Marktsituation spielen eine wichtige Rolle. Der Architektur-Entwickler liefert dabei wichtige Argumente, denn er verfügt über das für langfristige Entwicklungsziele notwendige Wissen, das zum Erreichen der tatsächlichen Ziele des Unternehmers oder Auftraggebers benötigt wird. Die Ziele Beispiele sind Entscheidungen, die kommende Standards betreffen oder die sich auf die Erweiterbarkeit eines Systems auswirken. Da bei unternehmerischen Entscheidungsträgern das Verständnis für Änderungsaufwand und Architekturqualität oft fehlt, kommt der engagierten Beratung durch den Architektur-Entwickler eine große Bedeutung zu.

Für Scoping-Entscheidungen ist die Methode Quality-Function-Deployment QFD besonders geeignet [**?**]. Bei dieser Planungsmethode werden

Marktsituation, Entwicklungsaufwand und strategische Ziele gegenübergestellt. Scoping-Entscheidungen sind vorrangig auf den frühen Iterationen zu treffen.

Entscheidung zwischen Eigenentwicklung und Kauf Entscheidung zwischen Eigenentwicklung und Kauf (Make or Buy) betreffen die Wiederverwendung sowohl von eigenen als auch von fremden Entwicklungsergebnissen. Beispiele sind so genannte COTS (Commercials Off The Shelf) -Components und frühere Ergebnisse aus dem eigenen Unternehmen. Eine Eigenentwicklung erfordert Ressourcen wie Zeit, Geld und Qualifikationen, führt jedoch zu Know-How und zum Eigentum an der betreffenden Lösung. Die Nutzung eines außerhalb entwickelten Elements führt häufig zu geringeren Kosten. Der Kostenreduktion stehen jedoch eine Reihe von Projektmanagement-Risiken bei Verwendung des Fremdprodukts gegenüber: der Aufwand für die Anpassung und die Integration, die Erfüllung der Anforderungen wie Zeitverhalten und Zuverlässigkeit sowie die verfügbare Unterstützung durch den Hersteller bei Integration und späterer Wartung sind unbekannt. Manchmal müssen trotz hoher Risiken Fremdprodukte genutzt werden, weil für eine (Eigen-)Neuentwicklung die Zeit nicht ausreicht. In anderen Fällen wird trotz geringer Risiken des Fremdprodukts eine Eigenentwicklung vorgenommen, um die Rechte (und Möglichkeiten) zu einer Weiterentwicklung einer strategisch wichtigen Komponente zu besitzen.

Bei den meisten Make-or-Buy-Entscheidungen stehen unternehmerische Kriterien im Vordergrund, der Entwickler wirkt oft nur beratend mit. Solche Entscheidungen betreffen alle Verfeinerungsebenen. Häufig ist es möglich, ganze Produkte – beispielsweise COTS-Komponenten – und Plattformen einzusetzen, wie beispielsweise ein Echtzeitbetriebssystem.

Reengineering-Entscheidung Verändern sich die Anforderungen an ein System, muss entschieden werden, in welchem Umfang im Zuge von Überarbeitungen auch die Architektur angepasst wird, so genanntes Reengineering (siehe auch Abschnitt **??**). Mit funktionalen Erweiterungen entstehen für den Kunden unmittelbar sichtbare Veränderungen, im Gegensatz zu Änderungen nichtfunktionaler Eigenschaften. Reengineering kann eingesetzt werden,um innere Qualitätsmerkmale wie Zuverlässigkeit, Wartbarkeit und Portabilität zu verbessern. Werden funktionale Erweiterungen vorgenommen, ohne die Architektur anzupassen, wirkt sich dies im Allgemeinen negativ auf die Qualitätsmerkmale der Architektur aus. Damit steigen die Risiken für langfristige Nutzbarkeit des betreffenden Systems. Der Aufwand für Reengineering-Maßnahmen steigt überproportional, wenn diese längere Zeit aufgeschoben wurden. Ursachen für eine solche Steigerung sind zum einen die sinkende Qualität der Architektur, zum anderen der große Umfang der betroffenen Systemteile. Werden notwendige Maßnahmen längere Zeit unterlassen, wird ein Reengineering oft undurchführbar.

Bei der dann notwendigen Ablösung und Neuentwicklung treten besonders hohe Kosten und Risiken auf. Sie sind häufig deutlich höher als bei einem rechtzeitigen und regelmäßigen Reengineering.

Vom unternehmerischen Standpunkt aus geht es bei einer Entscheidung über ein Reengineering auf Archtiekturebene um eine Abwägung zwischen kurz- und langfristigen Zielen, zwischen Kosten und Risiken. Der Reengineering-Aufwand stellt eine Investition in die zukünftige Nutzbarkeit eines Systems dar. Da für unternehmerische Entscheidungsträger das Gefühl für die mit mangelnder Architekturqualität verbundenen Risiken meist fehlt, muss der Entwickler seine beratende Funktion entsprechend wahrnehmen. Entscheidungen sind umso einfacher zu treffen, je besser die Mängel und Risiken verdeutlicht werden und je klarer qualitative Eigenschaften wie Zuverlässigkeit und Wartbarkeit definiert sind. Dazu sind vorher durch eine Bestandsaufnahme die benötigten Informationen über Architekturqualität und Wartbarkeit der betroffenen Elemente zu ermitteln.

Die langfristig geringsten Kosten treten erfahrungsgemäß dann auf, wenn bei jeglicher Erweiterung eines Softwaresystems um funktionale Merkmale immer eine Überarbeitung seiner Architektur erfolgt, um die qualitativen Eigenschaften zu erhalten.

4.6 Arbeitsmittel-Vorrat des Entwicklers

Bei der in den Abschnitten *Bewertung von Alternativen und Behandlung der Risiken* und *Entwicklung dieses Elements der Architektur* (Abschnitt **??**) beschriebenen Aufstellung von Lösungsalternativen greift der Entwickler auf einen Vorrat an Dingen zurück, die gewonnene Erfahrungen und frühere Arbeitsergebnisse repräsentieren. Es handelt sich um eine Sammlung von Arbeitsmitteln recht unterschiedlicher Typen, die hier nur genannt, aber nicht vorgestellt werden: Zum einen ist das Zusammentragen noch im Gange, zum anderen würde sein Umfang ein eigenes Buch füllen.

Dieser Vorrat umfasst nicht nur in der Literatur zusammengetragene Ergebnisse, sondern auch die eigenen Erfahrungen und Lösungen von anderen Entwicklern aus früheren Arbeiten. Eine explizite Dokumentation des Erfahrungsvorrats hat gegenüber der »Speicherung« als Erinnerung den Vorteil, ihn auf einfache Weise um neue Erfahrungen erweitern zu können, in dem diese in strukturierter Form hinzugefügt werden.

Alle Arbeitsmittel im Vorrat werden durch Beschreibungen ihrer positiven und negativen Eigenschaften ergänzt, anhand derer die Auswahl und Alternativenbewertung erfolgt. Sie werden situationsspezifisch eingesetzt, im Gegensatz zu den Prinzipien aus Abschnitt **??**, die in jeder Situation zu beachten sind.

Guidelines Bei Guidelines handelt es sich um Empfehlungen, die auf Erfahrungen und etablierten Methoden beruhen. Im Gegensatz zu Methoden,

Mustern und Regeln sind sie nicht hart festgelegt oder gar formal definiert. Sie ergänzen vielmehr syntaktische und semantische Regeln von Beschreibungsmitteln wie z.B. der UML sowie Vorschriften von Methoden um Erfahrungswissen im Sinne von Best Practices. Sie sind weicher formuliert als beispielsweise Muster. Bei Prüfungen liefern sie Verdachtsfälle, bei Entwicklungstätigkeiten liefern sie Hinweise. Sie sollten befolgt werden, wenn nicht spezielle Forderungen entgegenstehen.

Guidelines existieren auf unterschiedlichen Verfeinerungsebenen, von Guidelines zur Nutzung virtueller Methoden in C++ bis zu Empfehlungen der Verwaltung von nutzerspezifischen Anpassungen in GUI-Elementen. Auf allen Ebenen der Iterationen im Entwurfsprozess können deshalb geeignete Guidelines Unterstützung leisten. Sie werden vorrangig bei den Entwicklungs- sowie bei den Prüfungstätigkeiten der jeweiligen Ebene eingesetzt (siehe Abbildung **??**, S. **??**).

Einige werden in diesem Abschnitt vorgestellt. Anders als bei Mustern, sind nur recht wenige Guidelines in der Literatur beschrieben, wie z.B. [**?**]. Weitere Guidelines sind negativ als so genannte Bad Smells formuliert [**?**], und beschreiben zu vermeidende Strukturen. Gelegentlich werden Guidelines auch als Heuristiken bezeichnet, wie z.B. von Posch, Birken und Gerdom [**?**].

Die meisten Guidelines existieren im persönlichen Vorrat von erfahrenen Entwicklern und warten noch auf ihre Veröffentlichung.

Architektur-Stile können wie folgt definiert werden:

Definition (Architekturstil) *Architekturstile sind prinzipielle Lösungsstrukturen, die für ein Element einer Architektur durchgängig und mit weitgehendem Verzicht auf Ausnahmen angewandt werden.*

Sie können für Elemente auf allen Verfeinerungsebenen einer Architektur eingesetzt werden. Auf oberen Verfeinerungsebenen werden Architekturstile häufig als Leitidee der Architektur bezeichnet. Beispiele für Architekturstile sind Pipes&Filters, Layers, Object-Orientation und Implicit Invocation [**?**].

Architekturmuster Architekturmuster stellen ebenfalls Lösungen typischer Problemstellungen dar. Sie beziehen sich im Gegensatz zu Entwurfsmustern [**?**] auf mittlere Verfeinerungsebenen, oberhalb des Quellcodes. Siehe hierzu auch Kapitel **??**.

Referenzarchitekturen Architekturstile, die auf einer hohen Abstraktionsebene definiert sind und für die Architektur eines gesamten Systems oder einer Familie von Systemen einen Rahmen vorgeben, werden als Referenzarchitekturen bezeichnet. Sie stellen für die Elemente auf dieser Verfeinerungsebene Vorgaben ähnlich eines Standards bereit. Im Gegensatz zu

einer Leitidee muss eine Referenzarchitektur nicht notwendigerweise auf tieferen Verfeinerungsebenen angewandt werden. Bei der Festlegung einer Referenzarchitektur handelt es sich um eine strategische Entscheidung mit weit reichenden Auswirkungen auf das aktuelle Projekt und alle seine Nachfolger. Beispiele für herstellerspezifische Vorgaben, die Referenzarchitekturen umfassen, sind .NET und COM bzw. DCOM von Microsoft, JavaBeans und EnterpriseJavaBeans von Sun, Eclipse von IBM sowie CORBA. Siehe hierzu auch Kapitel **??**.