



Web-Site zum Buch  
unter [www.b-agile.de](http://www.b-agile.de)

Mit einem Vorwort von  
Tom DeMarco

Peter Hruschka  
Chris Rupp

# Agile Softwareentwicklung für Embedded Real-Time Systems mit der UML



HANSER

## Leitfäden für agile Projekte

Effektive Wissensvermittlung:

kurz und knapp • praxis- und ergebnisorientiert • Web-powered

- Seien Sie agil, beweglich, rege, dynamisch.
- Handeln Sie stets situationsgerecht, flexibel und angemessen.
- Machen Sie soviel wie nötig, so wenig wie möglich!
- Mitdenken statt »Dienst nach Vorschrift« und Dogma.

Keines der heute verfügbaren Vorgehensmodelle passt für alle Projekte. Agiles Vorgehen passt sich an die Projekte an: es beurteilt kontinuierlich die jeweilige Projektsituation und wählt dann die geeigneten Schritte.

## Agile Softwareentwicklung für Embedded Real-Time Systems

... wendet diese Grundsätze auf die Gestaltung technischer Systeme an.

... unterstützt jeden, der an der Entwicklung von Hardware-/Software beteiligt ist, durch wertvolle und praxisnahe Tipps, Muster, Heuristiken und Erfahrungen.

... bringt die für die Praxis wichtigsten Entwicklungsaktivitäten und Ergebnisse auf den Punkt.

... deckt mit pragmatischen Modellierungshinweisen die konkreten Informationsbedürfnisse des Entwicklungsteams, der Kunden, des Produktmanagements und der Qualitätssicherung ab.

... hilft Ihnen durch praxiserprobte Vorschläge bei der Analyse und dem Entwurf Ihrer verteilten, eingebetteten Echtzeitsysteme.

## Weitere Bücher in dieser Reihe:

Christiane Gernert

Agiles Projektmanagement

ISBN 3-446-21995-1

Gernot Starke

Effektive Software-Architektur

ISBN 3-446-21998-6

Hruschka/Rupp  
**Agile Softwareentwicklung**  
**für Embedded Real-Time Systems**  
**mit der UML**



Peter Hruschka  
Chris Rupp

# **Agile Softwareentwicklung**

## **für Embedded Real-Time Systems mit der UML**

HANSER

*Dr. Peter Hruschka, Aachen*

*hruschka@b-agile.de*

*Chris Rupp, Nürnberg*

*rupp@b-agile.de*

[www.hanser.de](http://www.hanser.de)

Alle in diesem Buch enthaltenen Informationen, Verfahren und Darstellungen wurden nach bestem Wissen zusammengestellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Buch enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autor und Verlag übernehmen infolgedessen keine juristische Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieser Informationen – oder Teilen davon – entsteht.

Ebenso übernehmen Verlag und Autoren keine Gewähr dafür, daß beschriebene Verfahren usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt deshalb auch ohne besondere Kennzeichnung nicht zu der Annahme, daß solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürfen.

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Ein Titeldatensatz für diese Publikation  
ist bei Der Deutschen Bibliothek erhältlich

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren) – auch nicht für Zwecke der Unterrichtsgestaltung – reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2002 Carl Hanser Verlag München Wien

Lektorat: Margarete Metzger

Herstellung: Irene Weilhart

Umschlaggestaltung: Zentralbüro für Gestaltung, Augsburg

Datenbelichtung, Druck und Bindung: Kösel, Kempten

Printed in Germany

ISBN 3-446-21997-8

# Inhalt

---

Auf dem Weg zur „Agilität“.....	IX
Vorwort der Autoren .....	XI
Teil I: Verteilte, eingebettete Echtzeitsysteme .....	1
1 Die Vielfalt technischer Systeme .....	3
1.1 Technische Systeme – Die unsichtbare Realität .....	4
1.2 Eigenschaften von RTE-Systemen .....	5
1.3 Die Vielfalt von RTE-Systemen .....	7
1.4 Dienstqualität (Quality of Service) .....	9
1.5 Haben Sie sich wiedergefunden? .....	12
2 Der Entwicklungsprozess.....	13
2.1 Systemebene & Softwareebene.....	14
2.2 Problemstellung und Lösung .....	15
2.3 Das akkumulierte Wissen .....	15
2.4 Multitechnologiesysteme .....	16
2.5 Das RTE-Vorgehensmodell .....	17
2.6 Agile System- und Softwareentwicklung .....	19
2.7 Der Prozess und die Herausforderungen von RTE-Systemen .....	21
2.8 Wie sieht Ihr Entwicklungsprozess aus?.....	22
Teil II: Der Produkt- und Systementwicklungszyklus.....	23
3 Erste Systemanforderungen .....	25
3.1 Ziele setzen .....	26
3.2 Stakeholder finden .....	27
3.3 System-/Produktkontext abgrenzen .....	29
3.4 Systemprozesse finden.....	36
3.5 Begriffe definieren .....	45
3.6 Haben Sie eine klare Produktvision vor Augen? .....	48

<b>4</b>	<b>Systemrandbedingungen: die Fesseln für Designer .....</b>	<b>49</b>
4.1	Physikalischen Kontext abgrenzen .....	50
4.2	Festlegungen zur Hardware dokumentieren .....	51
4.3	Subsysteme skizzieren .....	52
4.4	Randbedingungen sammeln .....	54
4.5	Softwaresubsysteme auf Knoten abbilden .....	57
4.6	Kennen Sie Ihre Schranken und Freiheitsgrade? .....	58
<b>5</b>	<b>Produkt-/Systemanforderungen präzisieren .....</b>	<b>59</b>
5.1	Systemprozesse spezifizieren .....	60
5.2	Abläufe präzisieren .....	69
5.3	Verhalten modellieren .....	74
5.4	Fachliche Dinge präzisieren .....	78
5.5	Beispielhafte Abläufe diskutieren .....	83
5.6	Sind Ihre Anforderungen präzise genug für die weitere Planung? .....	84
<b>6</b>	<b>Systemarchitektur: Mutig Entscheidungen treffen.....</b>	<b>85</b>
6.1	Designen = Entscheidungen treffen .....	86
6.2	Nicht-funktionale Anforderungen systematisch präzisieren .....	87
6.3	Hardwareverteilung entscheiden .....	94
6.4	Subsysteme entscheiden .....	96
6.5	Produktarchitektur prüfen .....	100
6.6	Hatten Sie den Mut, die Systemarchitektur zu entscheiden .....	102
<b>Teil III: Der Softwareentwicklungszyklus .....</b>		<b>103</b>
<b>7</b>	<b>Erste Softwareanforderungen .....</b>	<b>105</b>
7.1	Formulieren der Softwareziele .....	106
7.2	Überarbeiten der Stakeholderliste .....	107
7.3	Abgrenzen des Softwarekontexts .....	107
7.4	Softwareprozesse analysieren .....	109
7.5	Begriffe definieren .....	111
7.6	Wissen Sie, was Ihre Software leisten muss? .....	112
<b>8</b>	<b>Software-Randbedingungen durchdringen .....</b>	<b>113</b>
8.1	Schnittstellen zur Umgebung .....	114
8.2	Nicht-funktionale Anforderungen bewerten .....	116
8.3	Kennen Sie Ihren Verhandlungsspielraum? .....	116
<b>9</b>	<b>Fachliche Softwarearchitektur .....</b>	<b>117</b>
9.1	Abläufe präzisieren .....	118
9.2	Kategorisierung von Klassen .....	120
9.3	Dokumentation des Klassenmodells .....	126
9.4	Optimieren des Klassenmodells .....	134
9.5	Frameworks und Patterns – Hilfsmittel für Architekten .....	138
9.6	Ist Ihre fachliche Architektur stabil? .....	141
<b>10</b>	<b>Technische Softwarearchitektur .....</b>	<b>143</b>
10.1	Technologiegetriebene Klassen .....	144
10.2	Subsysteme und Komponenten .....	147
10.3	Vorgehensweise im Groben .....	149

10.4 Aktive und passive Komponenten .....	153
10.5 Kriterien zur Taskbildung .....	155
10.6 Taskkommunikation .....	161
10.7 Umsetzung der Kommunikation .....	167
10.8 Hilfsmittel für Architekten.....	169
10.9 Ist Ihre Architektur zukunftssicher?.....	170
<b>Teil IV: Weitere Aktivitäten.....</b>	<b>171</b>
<b>11 Konstruktion, Test und Inbetriebnahme .....</b>	<b>173</b>
11.1 Software schreiben.....	174
11.2 Code validieren .....	177
11.3 Testen.....	178
11.4 Abnahme .....	179
11.5 Haben Sie Grund zum Feiern?.....	180
<b>12 Entwicklung großer Systeme.....</b>	<b>181</b>
12.1 Systeme und Subsysteme .....	182
12.2 Der Entwicklungsprozess großer Systeme.....	183
12.3 Die Wissensstruktur großer Systeme .....	184
12.4 Denken Sie an Großes? .....	186
<b>Teil V: Die Anhänge .....</b>	<b>187</b>
Literaturverzeichnis.....	188
Index .....	190



## **Auf dem Weg zur „Agilität“**

---

In den 90er Jahren begannen wir, unsere Vorgehensmodelle zu perfektionieren. Wir wollten nicht nur jeden einzelnen Schritt perfekt machen, sondern auch noch den Gesamtablauf genau vorhersagen können. Wir waren wie Pool-Billard-Spieler, für die das Versenken einer Kugel nicht ausreicht. Sie müssen vorher die Bahn des Stoßes präzise voraussagen und ihn dann genau so durchführen. Das ist Vorgehen in der Tradition von CMM und ISO-9000.

Einige Firmen waren erfolgreicher als andere in der Perfektionierung ihrer Vorgehensmodelle. Aber eigenartiger Weise waren die Firmen, die es am besten schafften, nicht immer die erfolgreichsten am Markt. In unseren turbulenten Zeiten wurden meist nicht die Firmen belohnt, die ihre Vorgehensweise perfektioniert haben, sondern die, die sich am schnellsten anpassen konnten. Jeden Schritt vorhersagen zu können ist wirklich nicht wichtig, wenn wir gar nicht wissen, wohin der Weg uns führt und was uns unterwegs begegnet.

Das neue Jahrhundert brachte eine Reihe neuer Vorgehensmodelle mit sich, die sich „leicht“ oder „agil“ nennen. Sie sind ein Schritt in die richtige Richtung, weil sie die sich ständig ändernden Projektrealitäten zur Maxime gemacht haben. Außerdem ist ihre Leichtigkeit – die sie hauptsächlich durch Reduzierung des Papierbergs in Projekten erreichen – eine längst überfällige wirtschaftliche Maßnahme. Wir haben viel zu viel für Dokumentation ausgegeben – eine bürokratische Last der IT-Branche. Vor allem erlauben leichte Prozesse auch schnellere Entwicklungszyklen, allerdings auf Kosten eingeschränkter Überwachbarkeit.

So gesehen opfern agile Methoden etwas an Genauigkeit und Schärfe – was zu Zeitgewinnen führt. Im Gegenzug vergrößert sich das Projektrisiko. Diese Verschiebung ist sinnvoll, wenn der Zeitgewinn wichtig ist und alle Beteiligten sich des Risikos bewusst sind. Für Embedded Real-Time Systems sind

## Vorwort

aber vielleicht Risiken, die Sie in anderen Projekten eingehen können, nicht akzeptabel. Die Abwägung, was man tut und was man lässt, muss daher mit einem ausgeprägten Sinn für Risiken getroffen werden und mit einem sicheren Gespür für die Teile von Methoden und Verfahren, die uns erfolgreich zum Projektziel führen.

Auf dieses komplexe Terrain haben sich Peter Hruschka und Chris Rupp gewagt. Sie haben einen agilen Prozess entwickelt, der die Besonderheiten der Erstellung von Embedded Real-Time Systems berücksichtigt. Passt dieser Ansatz für Ihre Projekte? Das kann ich nicht sagen, ohne Ihre Organisation besser zu kennen, die Arten von Systemen, die Sie entwickeln und die Randbedingungen, mit denen Sie fertig werden müssen. Aber ich kann Ihnen sagen, dass Peter und Chris eine Menge Erfahrung in dieser Domäne haben. Es ist also sicherlich weise, ihre Ratschläge anzuhören und dann so umzusetzen, wie Sie es brauchen.

Camden, Maine

Januar 2002

*Tom DeMarco*

„Der wahre Zweck eines Buches ist,  
den Geist hinterrücks zum eigenen Denken  
zu verleiten.“

*Marie von Ebner-Eschenbach*

## Vorwort der Autoren

---

Liebe Leserin, lieber Leser, herzlich willkommen zu unserer Reise in die Welt der Entwicklung technischer Systeme!

Gehören Sie auch zu der vergessenen Mehrheit der Menschen, die sich mit der Entwicklung technischer Systeme befasst? Beim Lesen gängiger Computerliteratur, insbesondere zum Thema Objektorientierung und Unified Modeling Language (UML), könnte man zu der Ansicht gelangen, dass die meisten Computer Windows verwenden und in Desktopgehäusen untergebracht sind. In der Realität übersteigt die Anzahl der technischen Systeme, z. B. der eingebetteten Echtzeitsysteme (RTE-Systeme), die Anzahl der deutlich sichtbareren PC-Verwandten allerdings bei weitem. Ein europäischer oder amerikanischer Haushalt hat im Normalfall einen, vielleicht sogar zwei PCs zu bieten. Dem stehen Dutzende elektronische Geräte gegenüber, die Prozessoren und Software enthalten. Vom Fernseher und Videorecorder über die Mikrowelle und Waschmaschine bis hin zu mehreren Telefonen, dem Familienwagen und vielem mehr. RTE-Systeme gibt es überall, sie begleiten unser tägliches Leben, im Haushalt, in der Öffentlichkeit und in der Industrie. RTE-Systeme steuern Züge, regeln chemische Prozesse und überwachen Kernkraftwerke. Sie sichern den Zugang zum Firmenparkgelände und zu Ihrem Arbeitsplatz.

### **Software wird bei technischen Systemen oft überbewertet**

---

Sicherlich spielt Software eine immer wichtigere Rolle in technischen Systemen. In der Automobilbranche beispielsweise bildet Software inzwischen ungefähr 30 Prozent der Wertschöpfung. Es wird Sie bestimmt nicht überraschen, dass in Ihrem Mobiltelefon eine Menge an Software steckt. Genauso wie auch in einem elektrischen Rasierapparat Software viele Steuerungsfunktionen übernimmt.

Systeme  
sind mehr  
als Software

Trotzdem bewerten vor allem Softwareentwickler ihre Rolle im Umfeld dieser technischen Systeme oft zu einseitig. Um ihr den richtigen Stellenwert zuzuweisen, erinnern wir uns an ein Zitat von Imtiaz Pirbhai<sup>1</sup>: „Wann haben Sie das letzte Mal Software auf der Straße gesehen, wie sie sich selbst ausführt, ohne dass Hardware dabei war?“

Alle Überlegungen bei der Analyse und dem Entwurf von Software für technische Systeme sollten mit Blick auf das Gesamtsystem oder Produkt getroffen werden. Wir werden Sie deshalb immer wieder auf das Gesamtsystem hinweisen, dessen Funktionalität den eigentlichen Wert darstellt.

### **Warum sollten Sie Ihre wertvolle Zeit mit diesem Buch verbringen?**

---

Weil Sie hier Antworten darauf finden, wie Sie die entscheidenden Phasen der Entwicklung eines technischen Systems meistern – Erfahrungswerte, die in der gängigen UML-Literatur nicht zu finden sind! Fundierte Regeln, Schablonen und Checklisten, die Sie direkt bei Ihrer Arbeit unterstützen.

**Leserkreis**

Wenn Sie zu den Personen gehören, die an irgendeinem Schritt in der Entwicklung technischer Produkte oder Systeme beteiligt sind, dann wurde dieses Buch für Sie geschrieben.

**Ziel des Buches**

Es ist uns wichtig, in diesem Buch nicht einfach die verfügbaren UML-Diagramme noch einmal darzustellen, sondern ihre konkrete Anwendung bei einer Produktentwicklung im technischen Umfeld zu erläutern. Da wir kein dickes, sondern ein gut handhabbares, hilfreiches Buch schreiben wollten, spendieren wir wenig Raum für Basis-Know-how, das an anderer Stelle bereits gut dokumentiert wurde, sondern verweisen auf die entsprechenden Literaturstellen.<sup>2</sup>

### **Die Struktur des Buches**

---

**Aufbau des Buches**

Damit Sie im richtigen Kapitel landen – sofern Sie nicht alles chronologisch lesen – hilft Ihnen der folgende Überblick über die fachlichen Schwerpunkte der einzelnen Kapitel bei der Navigation.

Das Buch gliedert sich in vier Teile (siehe Abbildung 0.1), welche die fortlaufend nummerierten Kapiteln zu logischen Gruppen zusammenfassen.

**Teil I:  
Unsere Domäne**

Teil I des Buches schafft den Kontext und führt in den Entwicklungsprozess von eingebetteten Echtzeitsystemen ein. In Kapitel 1 arbeiten wir heraus, was

---

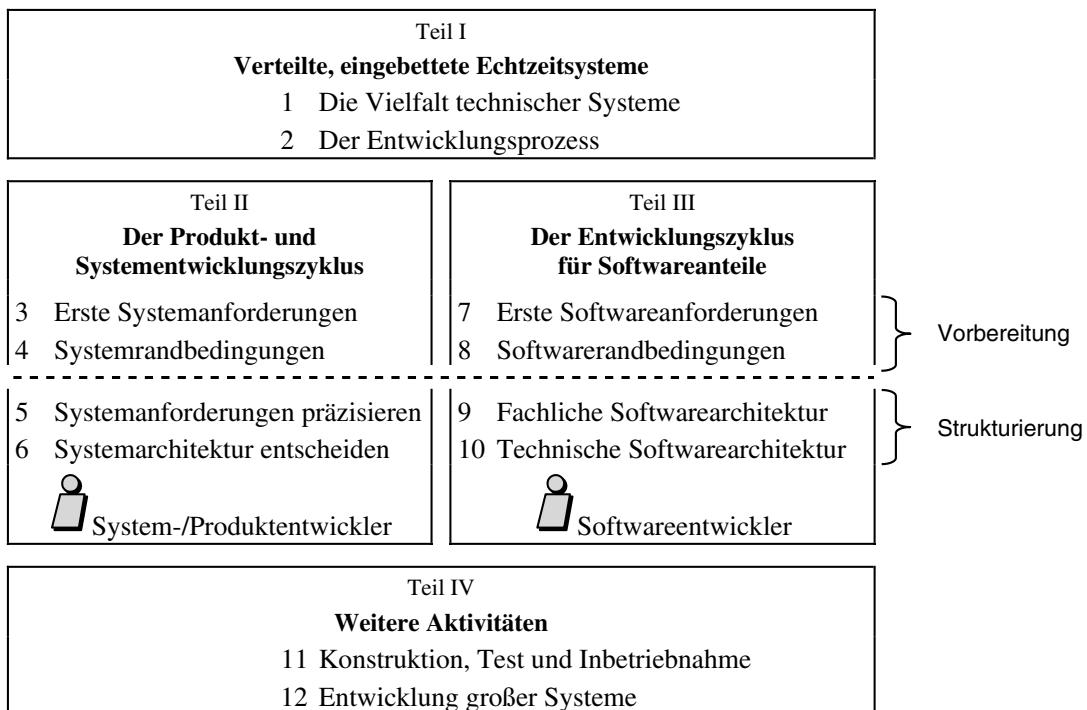
<sup>1</sup> Koautor des Klassikers [HP87]

<sup>2</sup> Insbesondere, wenn es sich um inzwischen unentgeltliche, frei zugängliche Informationen und Standards wie den Rational Unified Process [Kru01] oder den UML Notation Guide [UML1.4] handelt.

das Besondere an verteilten, eingebetteten Echtzeitsystemen ist, und leiten daraus Folgerungen für den Entwicklungsprozess ab (Kapitel 2).

Da RTE-Systeme mehr als nur Software umfassen, ist der Hauptteil des Buches in zwei große Blöcke aufgeteilt:

- ein Block für System- und Produktentwicklung aus ganzheitlicher Sicht,
- ein zweiter Block für die Spezialitäten der Softwareentwicklung im Rahmen der technischen Systementwicklung.



**Abbildung 0.1:** Gliederung des Buches

Teil II beschreibt zunächst die Aktivitäten, die als Vorbereitung für eine Produkt- oder Systementwicklung durchgeführt werden sollten. Ziel ist es, eine klare Aufgabenstellung für die Soft- und Hardwareentwicklung (Kapitel 3), sowie festgeschriebene Randbedingungen (Kapitel 4) zu erhalten.

Teil II:  
Ganzheitlicher  
Ansatz

Kapitel 5 erläutert, mit welchen Methoden und Verfahren Produkt- und Systemanforderungen präzisiert werden können. Kapitel 6 zeigt Wege zur Festlegung der Produkt- und Systemarchitektur. Dort werden die wesentlichen Entscheidungen zur Subsystembildung diskutiert und erläutert, wie man von dem Gesamtsystem zur Abgrenzung der Softwareanteile kommt.

Teil III: Softwareentwicklung	Teil III behandelt die Entwicklung der Softwareanteile. Die Kapitel 7 und 8 behandeln analog zu 3 und 4 die Vorbereitungsaktivitäten der Softwareentwicklung, welche die Softwareanforderungen und -randbedingungen festlegen. Kapitel 9 und 10 diskutieren in Analogie zu 5 und 6 die präzise Ausarbeitung der Softwareanforderungen und die Erstellung der Softwarearchitektur.
Teil IV: Zur Komplettierung	Teil IV rundet den Entwicklungsprozess ab. In Kapitel 11 finden Sie einige Hinweise auf weitere Tätigkeiten, wie Test und Inbetriebnahme bei der Entwicklung technischer Systeme, denen wir im Rahmen dieses Buches weniger Aufmerksamkeit geschenkt haben. Kapitel 12 erläutert, was Sie bei der Entwicklung sehr großer Systeme berücksichtigen müssen.
Für eilige Leser	Zu Beginn jedes Kapitels finden Sie eine Reihe von Leitfragen, die das Kapitel beantwortet. Am Ende jedes Kapitels bringen wir in einer Zusammenfassung mit wenigen Worten auf den Punkt, was im Kapitel selbst ausführlicher beschrieben ist. Checklisten sammeln dort die zentralen Fragen, die Sie sich im Projekt bei den entsprechenden Tätigkeiten stellen sollten.
Infos im Web	An einigen Stellen des Buches wird auf weiterführende Informationen auf unseren Web-Seiten verwiesen. Diesen Weg haben wir an Stellen gewählt, an denen wir Ihnen ständig aktualisierte Informationen anbieten möchten, Ihnen das Abtippen von Texten ersparen wollen oder die Detailinformationen den Fokus oder Umfang des Buches gesprengt hätten. Besuchen Sie uns auf unseren Web-Seiten <a href="http://www.b-agile.de/de/rte">www.b-agile.de/de/rte</a> und <a href="http://www.sophist.de/buch/rte">www.sophist.de/buch/rte</a> .

### **Ihre Meinung ist uns wichtig**

---

#### Ihr Feedback

Ihre Meinung zu unserem Buch ist uns sehr wichtig. Deshalb freuen wir uns auf Ihre Eindrücke und Verbesserungsvorschläge, Ihre Kritik, aber auch Ihr Lob. Treten Sie mit uns in Kontakt: [rte@b-agile.de](mailto:rte@b-agile.de)

### **Danksagungen**

---

Schreiben war unsere Tätigkeit, aber Ideen und Kommentare bekamen wir von vielen Freunden und Kollegen. Insbesondere wollen wir Christiane Gernert und Gernot Starke danken, die uns durch Ihre parallele Buchentwicklung die Gelegenheit gegeben haben, auf Ihre Ausführungen zu verweisen [Ger02], [Sta02]. Wertvolle Hinweise erhielten wir in Diskussion mit Suzanne und James Robertson, mit Tom DeMarco, Bernd Oestereich, Jutta Eckstein, Nico Josuttis, ... und unseren vielen Seminarteilnehmern und Beratungskunden. Jürgen Hahn danken wir für die zahlreichen qualitätssichernden Tage und Nächte. Spezieller Dank geht auch an unsere Lektorin Margarete Metzger.

„Das Auto hat keine Zukunft,  
ich setze aufs Pferd.“

*Wilhelm II. (1859–1941),  
deutscher Kaiser und König von Preußen*

# Teil I

## **Verteilte, eingebettete Echtzeitsysteme**

Bevor wir Ihnen im Hauptteil des Buches Vorgehensweisen, Methoden, Tipps, Tricks und Hilfsmittel zeigen, wollen wir in diesem Teil die Grundlagen dafür legen. Im ersten Kapitel analysieren wir, was verteilte, eingebettete Echtzeit-systeme (*Real-Time Embedded Systems*, kurz: *RTE-Systeme*) so anders macht.

Kapitel 1:  
Eigenschaften  
von Echtzeit-  
systemen

Wir studieren ihre charakteristischen Eigenschaften:

- Was bedeutet „Einbettung“?
- Was heißt „verteilt“?
- Was heißt „Echtzeit“?
- Welche anderen Qualitäten außer der „Echtzeittauglichkeit“ müssen derar-tige Systeme haben?

Nicht jedes RTE-System muss alle diese Eigenschaften besitzen. Anhand einer aus den Eigenschaften systematisch konstruierten Tabelle zeigen wir Ihnen die Vielfältigkeit verteilter, eingebetteter Echtzeitsysteme auf und geben Ihnen die Möglichkeit, Ihr System genau kennen zu lernen. Wenn Sie die Eigenschaften Ihres Systems oder Produkts kennen, dann können Sie Ihren Entwicklungsprozess Maß schneidern. Mit agilen Prozessen wollen wir verhindern, dass Sie viele Schritte nur deshalb durchführen, weil Ihr Vorgehensmodell es so verlangt.

## **Teil I: Verteilte, eingebettete Echtzeitsysteme**

### Kapitel 2: Der Entwicklungsprozess

Im zweiten Kapitel nutzen wir die Eigenschaften dieser Systeme, um Hinweise auf den Entwicklungsprozess zu geben. Wir unterscheiden

- den Produkt-/Systementwicklungsprozess als ganzheitlichen Ansatz und
- den Entwicklungsprozess für die Softwareanteile von eingebetteten Systemen.

Sie lernen die Verzahnung dieser beiden Prozesse kennen, die zu unterschiedlichen Arten der Herangehensweise bei einer Produkt- oder Systementwicklung führen kann. Wir diskutieren die Aktivitäten und stellen die beiden wichtigsten Ergebnisse, die wir durch die Aktivitäten anstreben, vor: Anforderungen und Architektur.

Sowohl für die System- als auch für die Softwareentwicklung besprechen wir die Ziele der beiden wichtigsten Phasen – der Vorbereitungsphase und der Strukturierungsphase – im Detail.

„Allen Fortschritt verdanken wir den Unzufriedenen. Zufriedene lieben keine Veränderung.“

*Salvatore Quasimodo (1901–1968),  
italienischer Lyriker, Kritiker und Übersetzer,  
1959 Nobelpreis für Literatur*

# 1

## Die Vielfalt technischer Systeme

### Fragen, die dieses Kapitel beantwortet:

- Was ist das Besondere an technischen Systemen?
- Was sind verteilte, eingebettete Echtzeitsysteme?
- Welche unterschiedlichen Arten technischer Systeme gibt es?
- Welche speziellen Eigenschaften haben sie?
- Welche Anforderungen an die Dienstqualität (Quality of Service) bestehen bei technischen Systemen?

Nach dem Lesen dieses Kapitels sollten Sie den Stellenwert von Software im Rahmen von technischen Systemen richtig einschätzen können. Wir wollen Sie für einen Augenblick von der Rolle des Softwareerstellers loslösen. Versetzen Sie sich in die Rolle des täglichen Nutzers vieler technischer Systeme. So gelangen Sie zu einer besseren Vorstellung davon, worauf es den meisten Nutzern bei diesen technischen Systemen wirklich ankommt. Mit diesem Wissen können Sie die Softwareentwicklung für technische Systeme gezielter vorantreiben und werden somit ein besserer Entwickler.

Ein Rollentausch  
als Gedanken-  
experiment

## 1.1 Technische Systeme – Die unsichtbare Realität

Wir sehen sie kaum! In Europa und den USA arbeiten bereits heute pro Einwohner 30 Mikrocontroller. Diese Zahl wird sich bis 2010 noch verdoppeln. Was ist das Besondere an diesen überall vorhandenen, aber kaum wahrgenommenen Systemen, die wir im Folgenden als RTE-Systeme (als Kurzform von *Real-Time & Embedded Systems*) bezeichnen? Warum können nicht einfach alle Erfahrungen und Vorgaben aus der Modellierung von kommerzieller Software unverändert in diesen Bereich übernommen werden?

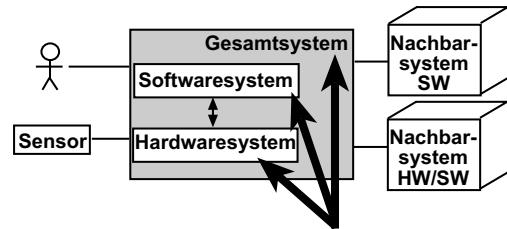
Systeme sind mehr als Software

Hinter einem RTE-System verbirgt sich neben der Software zumindest noch ein Hardwareanteil. Oftmals sind aber auch andere Technologien – z. B. mechanische, elektrische oder elektronische Komponenten – an der Verarbeitung beteiligt. Der Mensch ist zudem nicht nur Auslöser für Systemprozesse, sondern manchmal auch an deren Ausführung aktiv beteiligt.

Auf der Basis dieser Ausgangssituation stellt sich umgehend die Frage: „Von welchem System sprechen wir eigentlich?“ Abbildung 1.1 zeigt die unterschiedlichen Interpretationsmöglichkeiten.

- Die Mitarbeiter, die aus der Hardwareentwicklung kommen, deuten auf die Hardwareteile des Systems.
- Die „Softwerker“ sehen die mittels Software zu implementierenden Teile als das System an.
- Der Kunde sieht meist das System in seiner Gesamtheit. Aus seiner externen Sicht spielt es keine Rolle, welche Teile der Funktionalität mit welcher Technologie realisiert werden.

System = HW + SW + ...



**Abbildung 1.1:** Würde das richtige System sich bitte melden!

Treffen z. B. die Hardwareentwickler Entscheidungen aus ihrer Sicht, ohne die Bedürfnisse der Softwerker zu berücksichtigen, so kann es unter Umständen zu Designentscheidungen kommen, die sich für die Software als sehr ungünstig herausstellen. Bei einer rechtzeitigen Abstimmung über das gewünschte

Gesamtsystemverhalten und einer sinnvollen Aufteilung der Funktionalität auf die Technologien, könnten alle Beteiligten ein leichteres Leben führen.

Ein ganzheitlicher Ansatz

Um RTE-Systeme erfolgreich zu spezifizieren, ist daher ein ganzheitlicher Ansatz nötig, der sich nicht sofort auf die Einzelteile des Systems (z. B. Hard-

ware oder Software) konzentriert, sondern das System als Gesamtheit von außen – aus der Nutzersicht – betrachtet.

Erstens werden dadurch die Wünsche der Kunden und Käufer besser berücksichtigt, denn sie wollen ein funktionierendes Gesamtsystem. Und zweitens müssen die Entwickler die Scheuklappen ablegen. Statt einer engstirnigen Software- oder Hardwaresicht konzentriert man sich gemeinsam auf die effektivste Lösung und wählt Vor- und Nachteile der entsprechenden Technologien besser ab.

Eine Beobachtung zur Wortwahl: Falls ein RTE-System für uns räumlich gut greifbar ist und als eine abgeschlossene Einheit wahrnehmbar ist, dann neigen wir dazu, es als Produkt zu bezeichnen: eine Waschmaschine, ein Herzschrittmacher, ein Flugzeug. Falls ein verteiltes System neben Hardware und Software auch Menschen einschließt und nicht an einem Ort „greifbar“ ist, dann sprechen wir eher von Systemen: ein Zugangskontrollsystem, ein Verkehrsleitsystem. Einerseits scheint es uns weit hergeholt, einen Rasierapparat als „System“ zu bezeichnen, andererseits ist für ein Flugsicherungssystem das Wort „Produkt“ irreführend. Wir werden im Folgenden meist den Begriff „RTE-System“ verwenden und damit sowohl Produkte als auch Systeme meinen. Das Begriffspaar System/Produkt wäre korrekter, stört aber den Lesefluss zu sehr. Wo es umgangssprachlich eher angebracht ist, vor allem bei entsprechenden Beispielen, werden wir auch das Wort „Produkt“ verwenden.

System oder  
Produkt?

## 1.2 Eigenschaften von RTE-Systemen

RTE-Systeme haben einige besondere Eigenschaften, die den Entwicklungsprozess erheblich beeinflussen. Aber nicht jedes RTE-System hat alle diese Eigenschaften. Im Folgenden stellen wir Ihnen deshalb fünf wichtige Eigenschaften vor. Sie können dann für Ihr System überprüfen, in wie weit diese zutreffen. Wenn Sie damit die Charakteristik Ihres Systems erforscht haben, finden Sie im Rest des Buches immer wieder Hinweise, was Sie bei der Entwicklung besonders beachten sollen. So kommen Sie zu einem agilen Vorgehensmodell, das genau auf die Charakteristika Ihres Systems zugeschnitten ist.

Die fünf wichtigen Eigenschaften sind:

- die Einbettung in die Umgebung,
- die Verteilung,
- Zeitanforderungen,
- parallele Prozesse und
- die Datenhaltung.

Bei jeder Charakteristik betrachten wir nur zwei Extremwerte: Ist sie bei Ihrem System sehr ausgeprägt oder trifft sie eher weniger zu? Mit fünf Kriterien und je zwei Werten erhalten wir  $2^5$  Kategorien von technischen Systemen. Doch bevor wir uns mit dieser Vielfalt auseinandersetzen, ein paar Worte zu jedem Charakteristikum.

### Einbettung in die Umgebung

---

... embedded

Eingebettete Systeme überwachen oder steuern ihre Umgebung, oder tun beides gleichzeitig. Sensoren versorgen das System mit Informationen über den Zustand seiner Umwelt. Der Radsensor liefert dem Tempomat im Auto Informationen, um die nötigen Manöver einzuleiten und die Wunschgeschwindigkeit zu halten.

Viele Echtzeitsysteme verwenden nicht nur Sensoren, um ihre Umgebung zu überwachen, sondern gleichzeitig Aktuatoren oder Trigger für Nachbarsysteme, um externe Prozesse zu steuern. Der Tempomat erhält Informationen vom Bremsmodul und steuert die Motorregelung, um die vorgegebene Geschwindigkeit einzuhalten.

- Bei Systemen mit vielen Nachbarsystemen, Sensoren und Aktuatoren betrachten wir die Umgebung als **komplex**, ansonsten als **einfach**.

Auch ein System mit einer einfachen Umgebung kann wegen hochgradiger Parallelität oder vielen gleichzeitigen Benutzern noch ein komplexes RTE-System sein!

### Verteilung

---

... distributed

Ein weiteres Maß für die Komplexität unseres Systems und des Entwicklungsprozesses ist die Verteilung. Manche Systeme werden auf einem einzelnen Prozessor oder an einem Standort implementiert, andere Systeme werden auf eine Vielzahl von Prozessoren verteilt. Die kritische Stufe ist schon erreicht, wenn die Zahl eins überschritten wird. Dabei spielt es keine Rolle, ob die Prozessoren auf der selben Platine sitzen oder über einen Bus kommunizieren oder LANs oder WANs benutzen.

- Systeme werden auf einem **einzelnen** Prozessor oder Standort implementiert oder auf eine Vielzahl von Prozessoren oder Standorten **verteilt**.

### Zeitanforderungen

---

... real-time

- Wenn eine Nichteinhaltung von Zeitvorgaben durch das RTE-System in gravierenden, nicht akzeptablen Fehlern resultiert, charakterisieren wir die Zeitanforderungen als **hart**. Wenn es reicht, dass Zeiten im Wesentlichen eingehalten werden und manchmal auch geringfügig überschritten werden dürfen, dann stufen wir das Zeitverhalten als **weich** ein.

In *Hard-Real-Time-Systemen*, wie z. B. *Herzschrittmachern* oder *ABS-Systemen* sind verspätet ermittelte oder gelieferte Ergebnisse falsche und somit unbrauchbare Ergebnisse. „Hard Deadlines“ sind Zeitanforderungen, die eingehalten werden müssen, sonst drohen unkontrolliertes Systemverhalten, Systemabstürze oder Schlimmeres.

Die abgeschwächte Variante dieser Systeme nennt man *Soft-Real-Time-Systeme*. Diese sind nur an durchschnittliche Zeitvorgaben und Durchsätze gebunden (worunter ja fast alle Systeme fallen). Typische Vertreter sind Flugservierungssysteme, Navigationssysteme oder Geldautomaten. Auch hier sind verspätete Ergebnisse nicht wünschenswert, führen aber zu keinen Katastrophen. Die meisten RTE-Systeme sind eine Mischung aus Hard- und Soft-Real-Time-Systemen. Für einige wenige Teile existieren harte Deadlines. Der große Rest ist weniger zeitkritisch.

## Parallele Prozesse

RTE-Systeme reagieren auf Ereignisse aus der Umwelt. Wie wir später sehen werden, führt das zu einer sehr natürlichen Gliederung unserer Systeme in unabhängige, parallel ausführbare Prozesse. Ein Wetterdatensystem empfängt von zahlreichen Erfassungssystemen unabhängig voneinander Wettermeldungen und leitet diese an verschiedene Empfänger weiter.

Reaktion auf Ereignisse in der Systemumgebung

- Wir unterscheiden Systeme, die **wenige** oder **viele** parallele Prozesse zur Reaktion auf Ereignisse in ihrer Umgebung haben. Die Anzahl der parallelen Prozesse gibt ein natürliches Maß für die Komplexität des Systems.

## Datenhaltung

Früher spielten Datenbanken oder komplexe Datenstrukturen in eingebetteten Systemen eine untergeordnete Rolle. Mit der Verfügbarkeit von immer kleineren, leichteren und billigeren Speichermedien hat jedoch auch die Datenhaltung Einzug in RTE-Systeme gehalten und trägt zu ihrer Komplexität bei.

Klassische IT-Aufgabe auch bei RTE-Systemen: Datenverwaltung

- Muss das System auf einen Datenstamm zurückgreifen oder langfristig Daten sammeln, so besitzt es eine **signifikante**, sonst eine **nicht signifikante** Datenhaltung.

## 1.3 Die Vielfalt von RTE-Systemen

Die folgende Tabelle zeigt einige Beispiele von RTE-Systemen mit unterschiedlichen Ausprägungen der oben beschriebenen Eigenschaften.

**Tabelle 1.1:** Beispiele für RTE-Systeme und Produkte

Nr.	Einbettung in die Umgebung	Verteilung	Zeit-anforderungen	Parallel Prozesse	Datenhaltung	Beispiel
0	einfach	einzeln	weich	wenige	nicht sig.	Batterieladestation
1	einfach	einzeln	weich	wenige	signif.	Telefon
2	einfach	verteilt	weich	wenige	nicht sig.	Kasse im Supermarkt
3	einfach	verteilt	weich	wenige	signif.	Bankautomat
4	einfach	einzeln	hart	wenige	nicht sig.	Echtzeit-Trainingssimulator, Tempomat
5	einfach	einzeln	hart	wenige	signif.	File Server
6	einfach	verteilt	hart	wenige	nicht sig.	Digitales Filter mit Parallelarchitektur
7	einfach	verteilt	hart	wenige	signif.	Trainingsflugsimulator mit Landschaften und Zielen zur Auswahl
8	komplex	einzeln	weich	wenige	nicht sig.	Umweltsteuerung (Ventilation, Heizung)
9	komplex	einzeln	weich	wenige	signif.	Zugangskontrollsystem
10	komplex	verteilt	weich	wenige	nicht sig.	Meteorologische Wetterdatenerfassung
11	komplex	verteilt	weich	wenige	signif.	Wie 10 mit Datenbank
12	komplex	einzeln	hart	wenige	nicht sig.	Aufzug, Zündsystem für Motor, Defibrillator, Roboter einer Fertigungsstraße
13	komplex	einzeln	hart	wenige	signif.	Radarverfolgungssystem
14	komplex	verteilt	hart	wenige	nicht sig.	Bremskontrollsystem
15	komplex	verteilt	hart	wenige	signif.	Verteiltes Radarsystem
16	einfach	einzeln	weich	viele	nicht sig.	Gebäudezugangskontrollsystem
17	einfach	einzeln	weich	viele	signif.	Meteorologisches und aeronautisches Informationssystem
18	einfach	verteilt	weich	viele	nicht sig.	E-Mail-System
19	einfach	verteilt	weich	viele	signif.	Flugbuchungssystem
20	einfach	einzeln	hart	viele	nicht sig.	Synthesizer, Keyboard
21	einfach	einzeln	hart	viele	signif.	Börsenkontrollsystem
22	einfach	verteilt	hart	viele	nicht sig.	Multiprozessor-Synthesizer, Mischpult
23	einfach	verteilt	hart	viele	signif.	Wie 21, aber verteilt
24	komplex	einzeln	weich	viele	nicht sig.	Lokales Verkehrsüberwachungssystem
25	komplex	einzeln	weich	viele	signif.	Komponentenmanagementsystem
26	komplex	verteilt	weich	viele	nicht sig.	Verkehrsinformationssystem
27	komplex	verteilt	weich	viele	signif.	Verkehrsleitsystem
28	komplex	einzeln	hart	viele	nicht sig.	Hardware-Interfaceboardsystem

Nr.	Einbettung in die Umgebung	Verteilung	Zeit-anforderungen	Parallele Prozesse	Datenhaltung	Beispiel
29	komplex	einzel	hart	viele	signif.	Echtzeit-Ereignisaufzeichnungssystem, Service-Roboter
30	komplex	verteilt	hart	viele	nicht sig.	Fahrzeugsteuerungssystem
31	komplex	verteilt	hart	viele	signif.	Überwachungsstation einer Prozesssteuerung, Telefonnetzwerkmanagementsystem

## 1.4 Dienstqualität (Quality of Service)

Was erwarten unsere Anwender von den technischen Systemen, die wir bauen? Sicherlich die Erfüllung der Funktionalität: Ein ABS-System muss das Blockieren der Reifen beim Bremsen verhindern, ein Herzschrittmacher muss die Herzaktivität unterstützen, eine Schnittstellenkarte muss ihre Mittlerfunktion zwischen zwei Systemen erfüllen.

Aber das alleine genügt noch nicht. Wir haben – leider zu oft unausgesprochen – eine Menge zusätzlicher Erwartungen an die Qualität der Leistungserbringung. Für diese Anforderungen an die Dienstqualität (Quality of Service) existiert ein viel zitiert internationaler Standard (ISO/IEC 9126 bzw. DIN 66272), an den wir uns hier anlehnen werden.

Abbildung 1.2 zeigt eine Gliederung der Qualitätseigenschaften, die wir im Folgenden etwas genauer betrachten. Der Qualitätsrahmen soll vermeiden, dass wir nicht-funktionale Anforderungen an unser System vergessen, und gibt uns zudem eine mögliche Gliederung der Merkmale vor.

### Benutzbarkeitsanforderungen

Systeme und Produkte werden für Menschen gemacht. Achten Sie daher bei der Gestaltung einerseits auf die Bedienbarkeit, andererseits auf die leichte Erlernbarkeit und Verständlichkeit des Systems.

„Leicht bedienbar muss es sein“

Bei RTE-Systemen ist für die Benutzbarkeit nicht mehr nur die Hardware verantwortlich, welche die Schnittstelle zum Anwender des Systems in Form von Knöpfen, Hebeln oder Schaltern darstellt. Immer mehr Softwarefunktionen sind dem Benutzer z. B. beim Mobiltelefon direkt ersichtlich. Dieses Qualitätsmerkmal ist für RTE-Systeme im täglichen Leben von sehr hoher Bedeutung. Anwender von RTE-Systemen sind wesentlich weniger tolerant bezüg-

lich schlechter Bedienbarkeit, Verständlichkeit und Erlernbarkeit, als der durchschnittliche PC-Benutzer. Falls unser RTE-System auf einem Massenmarkt kommerziell vertrieben werden soll (z. B. Mobiltelefon, CD-Player), muss den Wünschen an die Benutzbarkeit bei der Erfassung der Anforderungen sehr große Aufmerksamkeit geschenkt werden.

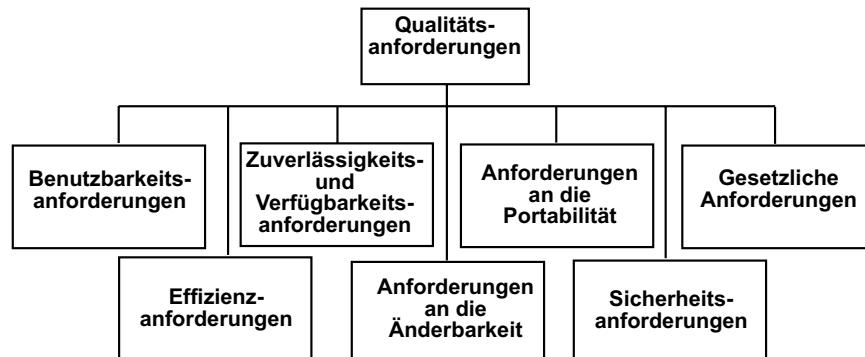


Abbildung 1.2: Kategorien von Qualitätsanforderungen

### Effizienzanforderungen

„Schnell muss es sein“

Zu den Effizienzanforderungen zählen zwei Kategorien: Zeitanforderungen und Kapazitätsanforderungen. Die Bedeutung des ersten Punktes haben wir schon unter den besonderen Eigenschaften von RTE-Systemen diskutiert. Sie sollten sehr klare Vorstellungen davon haben, wie schnell Ihr System auf externe Ereignisse reagieren muss, welche Antwortzeiten Sie erwarten und wie häufig und mit welcher Verteilung Ereignisse auf Ihr System einstürmen werden. Bezuglich der Kapazitäten sollten Sie Mengengerüste festlegen und evtl. das Verbrauchsverhalten vorgeben. Bei der Entwicklung von RTE-Systemen werden oft sehr hohe Ansprüche an möglichst geringen Ressourcenverbrauch des Systems gestellt. Insbesondere, wenn es sich um ein Produkt handelt, das in sehr hohen Stückzahlen produziert wird, ist der Bedarf an Speicher- und Prozessorleistung ein entscheidender Kostenfaktor.

### Zuverlässigkeit- und Verfügbarkeitsanforderungen

„Es muss laufen, wenn ich es brauche“

Hinter den Zuverlässigkeitssanforderungen verbergen sich Forderungen nach der Stabilität und Robustheit des Systems, Anforderungen an die Fehlertoleranz auch bei Fehlbedienungen und an die Wiederherstellbarkeit, wenn doch Fehler aufgetreten sind. Die Zuverlässigkeit wird bei RTE-Systemen selten für das Gesamtsystem, sondern häufiger für Teilfunktionalitäten eines Systems spezifiziert, da diese für den Betrieb unterschiedlich wichtig sind. Dabei ist zu

beachten, dass die Zuverlässigkeit für den Verbund von Hard- und Software betrachtet werden muss.

Die Probleme und Umwelteinflüsse (z. B. Kälte, Verschmutzung), mit denen ein RTE-System zu kämpfen hat, unterscheiden sich je nach Einsatzort des RTE-Systems stark von denen herkömmlicher PCs und damit von den Problemen, die bei der Entwicklung von kommerzieller Software berücksichtigt werden müssen.

## Anforderungen an die Änderbarkeit

Hier sollen Sie Ihre Erwartungshaltung bezüglich der Wartbarkeit des Systems formulieren. Wie leicht muss das System analysierbar sein (*Analysierbarkeit*)? Welchen Aufwand gestatten Sie für die Modifizierung (*Modifizierbarkeit*)? Welchen Grad an Stabilität erwarten Sie bzw. was darf bei Änderungen auf keinen Fall an unerwarteten Wirkungen auftreten (*Änderungsstabilität*)?

„Es muss immer auf dem Stand der Technik sein“

RTE-Systeme besitzen auch bezüglich dieser drei Ausprägungen des Qualitätsmerkmals Änderbarkeit Unterschiede zu kommerzieller Software. Bei der *Analysierbarkeit* kommt zumindest die Zusatzforderung hinzu, zwischen Hardware- und Softwarefehlern trennen zu können. Die Anforderungen an die *Modifizierbarkeit* sind meist wesentlich reichhaltiger, da die *Umwelt* eines RTE-Systems mannigfaltiger, häufigeren Änderungen unterworfen und komplexer ist. *Änderungsstabilität* ist durch die größere Wahrscheinlichkeit von Änderungen schwerer sicher zu stellen. An der sorgfältigen Erhebung der Anforderungen zur Änderbarkeit führt deshalb kein Weg vorbei.

## Anforderungen an die Portabilität

Unter Portabilität wollen wir alle Forderungen bezüglich der Übertragbarkeit in eine andere organisatorische Hardware- oder Softwareumgebung zusammenfassen. Wir müssen unsere Systeme oder Teilsysteme anpassen, installieren und austauschen können.

„Ich will es demnächst auch unter Windows XP“

Bei der Übertragbarkeit ist für RTE-Systeme vor allem das Submerkmal Anpassbarkeit wichtig, da RTE-Systeme auf unterschiedlichen Plattformen unter unterschiedlichsten Umgebungsbedingungen eingesetzt werden.

## Sicherheitsanforderungen

Bei einigen RTE-Systemen ist Sicherheit ein besonders heikles Thema. Dabei geht es um zwei Kategorien von Sicherheit: Einerseits um die Sicherheit für Leib und Leben bzw. die Vermeidung erheblicher Umweltschäden (engl. Safety). Andererseits gehören zu den Sicherheitsanforderungen auch Zugangsberechtigungsprüfungen, Integritätsprüfungen, um unbeabsichtigten Missbrauch durch legale Benutzer zu unterbinden, Forderungen nach Auditfähigkeit zur

„Es darf mich nicht gefährden“

Feststellung von Missbrauch, Integritätsnachweise nach abnormaler Beendigung von Programmen und Forderungen nach Datenintegrität bei normaler Nutzung. All diese Eigenschaften werden im Englischen unter „Security“ zusammengefasst.

### Gesetzliche Anforderungen

---

„Es muss alle Richtlinien unserer Firma einhalten“

Schließlich haben wir als letzte Kategorie gesetzliche Anforderungen unter die Qualitätsanforderungen mit aufgenommen. Dazu zählen nicht nur „staatliche“ Gesetze, sondern auch Standards, Normen und Vorschriften innerhalb von Firmen oder Organisationen. Diese Kategorie wird oft übersehen oder als selbstverständlich angesehen und daher nicht explizit durchdacht. Durch Einführung dieser Kategorie wollen wir sicherstellen, dass Sie sich bewusst Gedanken darüber machen, welchen Anforderungen Sie in diesem Bereich gerecht werden müssen.

Bevor Sie nun mit dem Entwicklungsprozess starten und sich auf die nächsten Kapitel stürzen, sollten Sie sich Gedanken über die Eigenschaften Ihres Systems machen. Ihr Entwicklungsprozess hängt im Wesentlichen von diesen signifikanten Eigenschaften Ihres Systems und von den Risiken der Systementwicklung ab. Agiles Vorgehen bedeutet ein Vorgehen zu wählen, das dem System und der Problemstellung angepasst ist.

Zur Vorbereitung der Entwicklung Ihres RTE-Systems müssen Sie sich daher mit dessen Eigenschaften auseinandersetzen. Welche der in diesem Abschnitt genannten Eigenschaften finden Sie in dem System wieder, das Sie jetzt entwickeln sollen?

Da es eine Vielzahl von unterschiedlichen RTE-Systemen gibt, können wir keine exakte Anleitung für das beste Vorgehen in genau Ihrem Fall geben. Aber wir werden in den nächsten Kapiteln näher auf die hier beschriebenen Eigenschaften eingehen und Ihnen Hinweise geben, welche Abweichungen sich ergeben können.

## 1.5 Haben Sie sich wiedergefunden?

---

In diesem Kapitel haben wir die Eigenschaften und Besonderheiten von RTE-Systemen untersucht und Unterschiede zu kommerzieller Software herausgearbeitet. Betrachten Sie Ihr Produkt oder System kritisch:

- Kennen Sie die wichtigen Eigenschaften Ihres RTE-Systems?
- Welche Eigenschaften sind für Sie besonders wichtig?
- Auf welche Qualitäten legen Sie besonderen Wert? (Wir werden Sie in Kapitel 6 bitten, dafür Prioritäten anzugeben.)

„Ohne natürliche Einsicht sind Regeln und Richtlinien wertlos.“

*Quintilian, 30–96 n. Chr.*

# 2

## **Der Entwicklungsprozess**

### **Fragen, die dieses Kapitel beantwortet:**

- Wie beeinflussen die Eigenschaften von verteilten, eingebetteten Echtzeitsystemen den Entwicklungsprozess?
- Welche Aktivitäten im Entwicklungsprozess haben besonders hohen Stellenwert?
- Wie unterscheidet sich die Entwicklung des Gesamtsystems von der Entwicklung der Softwareanteile?
- Welche Ergebnisse wollen wir produzieren?
- Wie beeinflussen verschiedene Randbedingungen die Entwicklung?

Als Alternative zu herkömmlichen Vorgehensmodellen lernen Sie in diesem Kapitel einen ergebnisorientierten Ansatz kennen. Sie brauchen keine meterdicken Vorschriften, wenn Sie Ihre Ziele kennen und viele Empfehlungen, Heuristiken und Erfahrungswerte nutzen, um sich diesen Zielen systematisch zu nähern. Wir schlagen Ihnen vor, das Wissen für erfolgreiche Projektabwicklung eher in den Köpfen und Herzen Ihrer Entwickler zu deponieren, als in Ordnern<sup>1</sup>.

b-agile:  
pragmatische  
„problemgerechte“ Vorgehensmodelle und  
viele „Best Practices“

<sup>1</sup> Nur zum Nachschlagen, falls es in den Köpfen verloren gegangen ist, schreiben wir es auch nieder. Aber eigentlich sollten diese „Best Practices“ jedem Entwickler, der sie einmal gelernt und deren Sinn verstanden hat, ständig präsent sein.

Wie im Vorwort erwähnt werden eingebettete Systeme immer softwareintensiver. Trotzdem kaufen unsere Kunden meist nicht die Software, sondern das Gesamtsystem.

2 Ebenen,  
je 2 Ergebnisse

Das hat Auswirkungen auf den Entwicklungsprozess: Wir betrachten – etwas vereinfacht – den Entwicklungsprozess auf zwei Ebenen: Auf der Systemebene und der Softwareebene. In jeder Ebene wollen wir zwei Hauptergebnisse produzieren: Anforderungen – als Niederschrift der jeweiligen Problemstellung – und Architekturen – als Dokumentation einer dazu passenden Lösung.

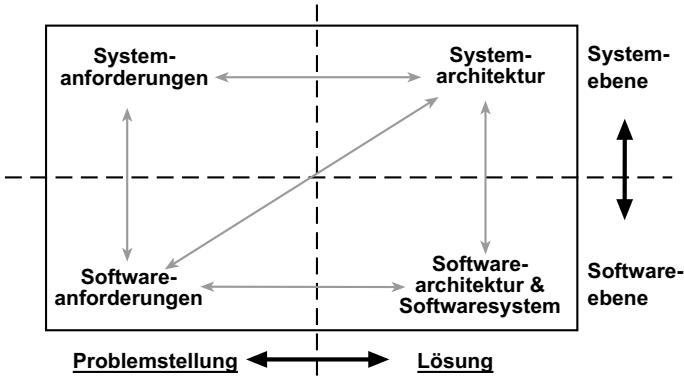


Abbildung 2.1: Die zwei Ebenen der Entwicklung (aus Softwaresicht)

## 2.1 Systemebene & Softwareebene

Systeme und  
Produkte als  
Ganzes

Die beiden Ebenen sollen uns helfen, den Umfang des jeweils betrachteten Entwicklungsgegenstandes klar im Blick zu haben: Auf der oberen Ebene betrachten wir das Produkt oder das System, das der Kunde von uns haben möchte, in seiner Gesamtheit. Bei dieser Gesamtsicht spielt es noch keine Rolle, welche Teile als Software- oder als Hardwarelösungen entwickelt werden.

Die Software-  
anteile davon

Auf der unteren Ebene betrachten wir dann nur noch die Teile der Systemarchitektur, die wir mit Hilfe von Software lösen wollen. In der Praxis stellen wir in vielen Projekten fest, dass Softwareentwickler sofort mit der unteren Ebene beginnen, auch wenn sie keinen Überblick über das Gesamtsystem haben. Wir wollen Sie als Softwareentwickler dazu bringen, dass Sie Informationen über das Gesamtsystem zumindest einfordern, um Ihre eigene Arbeit zielgerichteter und effizienter zu gestalten.

## 2.2 Problemstellung und Lösung

Die senkrechte Achse in Abbildung 2.1 trennt die Problemstellung von der Lösung. Es mag Ausnahmen geben, aber im Allgemeinen gehen wir davon aus, dass es sinnvoll ist, die Problemstellung (bis zu einem gewissen Grade) zu kennen, bevor man sich daran macht, Lösungen zu suchen. (Nur wenn Sie Monopolist sind, brauchen Sie sich um Problemstellungen nicht zu kümmern. Sie können der Welt dann Ihre Lösungen verkaufen. Die Probleme kommen dann schon.)

Sie sollten bei Ihrer Entwicklung Problemstellung und Lösung klar trennen, denn

- die Problemstellung ist meist länger stabil als die Lösungen und
- zu einer Problemstellung finden Sie normalerweise mehr als eine mögliche Lösung.

Problemstellung  
und Lösung aus  
guten Gründen  
getrennt halten

Aus diesen Gründen streben wir diese beiden Ergebnisse auf jeder Ebene der Entwicklung an und dokumentieren sie getrennt voneinander.

## 2.3 Das akkumulierte Wissen

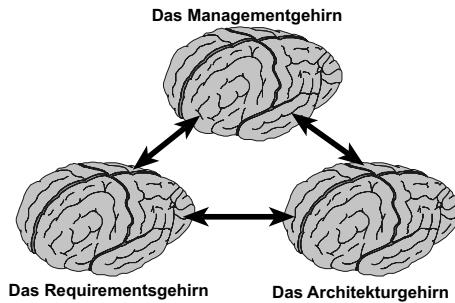
Viele Vorgehensmodelle schlagen Dutzende oder Hunderte von Ergebnistypen vor, die im Laufe der Systementwicklung erzeugt werden sollen. Agile Entwicklung konzentriert sich auf drei maßgebliche Artefakte, die konsequent weiterentwickelt werden. Um von der physikalischen Form der Ergebnisse abzulenken und nicht über Papierdokumente oder Intranetseiten oder Datenbanken sprechen zu müssen, bitten wir Sie, folgendes Gedankenexperiment mitzumachen: Stellen Sie sich einfach drei spezialisierte Gehirne vor, in denen das Wissen über die Problemstellung, das Wissen über die Lösung und das Wissen über das Management des Entwicklungsvorhabens jeweils strukturiert und hochgradig vernetzt festgehalten wird<sup>2</sup>. Selbstverständlich können wir den Inhalt dieser Gehirne jederzeit auf verschiedenste Arten zu Papier bringen, in Präsentationen vermitteln oder elektronisch übertragen.

Alles, was wir zu einem Zeitpunkt zu dem jeweiligen Thema wissen, ist im jeweiligen Gehirn gespeichert. Je nach Zustand oder Phase des Entwicklungsvorhabens wird weniger oder mehr des entsprechenden Gehirns gefüllt sein. Es gibt kein starr vorgegebenes Schema, was alles im Gehirn festgehalten werden muss. Wir betrachten die Risiken im jeweiligen Bereich und entscheiden danach, wie viel oder wie wenig wir zu einem Thema festhalten müssen.

Die vernetzten  
Entwicklungs-  
gehirne

---

<sup>2</sup> Die Idee zu dieser Metapher gab uns Dorothy Graham in einem Vortrag über „Cognitive Illusions in Development and Testing“.



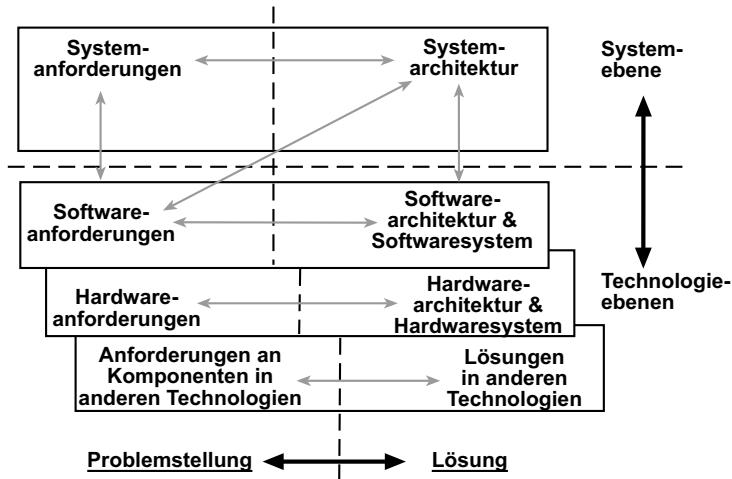
**Abbildung 2.2:** Die Ergebnisstrukturen der Systementwicklung

In diesem Buch gehen wir nur auf das Problemstellungs- oder Requirementsgehirn und das Lösungs- oder Architekturgehirn genauer ein. Über den Inhalt des Managementgehirns erfahren Sie mehr in [Ger02].

## 2.4 Multitechnologiesysteme

Nochmals für alle Softwareentwickler: RTE-Systeme sind mehr als nur Software!

Abbildung 2.1 zeigte nicht die ganze Wahrheit für RTE-Systeme. Die untere Hälfte muss man natürlich analog auch für Hardwarekomponenten und für jede andere am Projekt beteiligte Technologie ergänzen (z. B. Anforderungen und Lösungen für mechanische, elektrische oder manuelle Teilsysteme), wie die Abbildung 2.3 zeigt.



**Abbildung 2.3:** Die zwei Ebenen der Entwicklung (aus Gesamtsicht)

Wir konzentrieren uns in diesem Buch in der unteren Ebene nur auf Softwareanteile. Mehr zu Multitechnologiesystemen finden Sie im Kapitel 12 bzw. in [HHP00].

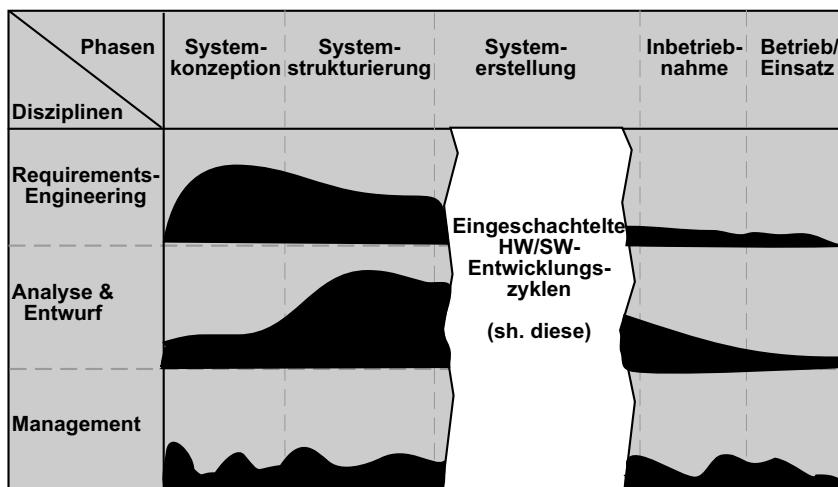
## 2.5 Das RTE-Vorgehensmodell

Der Rational Unified Process (RUP) [Kru01] ist ein in der Praxis weit verbreitetes Vorgehensmodell. Leider deckt er nur den Softwareentwicklungszyklus ab und ist daher für RTE-Systeme nicht direkt einsetzbar. Wir lehnen uns für die Softwareentwicklung an den RUP an, betonen ihn jedoch in einen Systementwicklungsprozess ein. Abbildung 2.4 zeigt die Phasen und Disziplinen dieser übergeordneten Systementwicklung von der Idee bis zum Einsatz. Sie stellt den Idealfall dar. Wir wissen, dass in der Praxis Softwareprojekte oft begonnen werden, ohne dass die Anforderungen und die Architektur des Gesamtsystems ausreichend betrachtet wurden.

System-entwicklung

Die horizontale Achse zeigt das System oder das Produkt von der ersten Idee bis zum Betrieb. Die Abbildung stellt eher logische Abhängigkeiten und nicht zwangsläufig den zeitlichen Ablauf dar. Wir konzentrieren uns in diesem Buch ausschließlich auf die kritischen ersten beiden Phasen der Entwicklung: Auf die Systemkonzeption und auf die Systemstrukturierung.

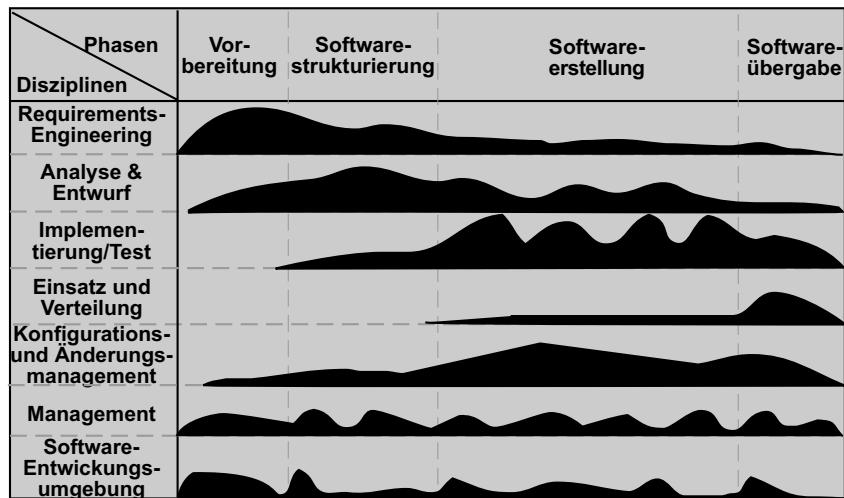
Software-entwicklungszyklus



**Abbildung 2.4:** Der Systementwicklungszyklus

Die nächste große Phase ist dann die Systemerstellung. Hier ist die parallele Entwicklung von Hardware, Software und anderen Komponenten eingebettet. Hardwareentwicklung ist nicht Thema dieses Buches. Für die Softwareentwicklung wird der hier weiß gezeichnete Bereich in Abbildung 2.5 detailliert dargestellt. In diesem Überblick entspricht der Prozess weitestgehend dem RUP. Wir haben jedoch die Phasen und Disziplinen so benannt, dass sie im Zusammenhang mit dem übergeordneten Systementwicklungszyklus besser verständlich sind.

## 2 Der Entwicklungsprozess



**Abbildung 2.5:** Der Entwicklungszyklus für Softwareanteile eines Produkts/Systems

Im Rest des Buches behalten wir die Aufteilung in die beiden Ebenen bei. Teil II befasst sich mit dem Systementwicklungsprozess für das Gesamtsystem oder Produkt. Teil III diskutiert den Entwicklungsprozess für die Softwareanteile.

Unterschiede  
System-/  
Software-  
entwicklung

Vieles in dem Produktentwicklungszyklus und dem von uns vorgeschlagenen Entwicklungszyklus für Softwareanteile wird klarer, wenn man sich einmal die kleinen, aber maßgeblichen Unterschiede zwischen Systementwicklung und Softwareentwicklung in Tabelle 2.1 veranschaulicht.

**Tabelle 2.1:** System-/Produktentwicklung und Softwareentwicklung

System-/Produktentwicklung	Softwareentwicklung
Zeitrahmen: 1 bis n Jahre	3 – 12 Monate
Oft nur Visionen statt genauer Ziele	Relativ genaue Ziele
Budgetrahmen (ist mir bis zu ... wert)	Fixes Budget
Zeitrahmenvorgabe (in ca. 3 Jahren)	Feste Zeitvorgabe (Termin: 31.10.1!)
Eher eine Strategie für die Herangehensweise, die erst durch Erkenntnisse im Ablauf präzisiert werden kann.	Genaue Pläne
Auch iterativ machbar, aber die Iterationen erzeugen nicht unbedingt immer Software-Inkremeante, sondern evtl. auch andere brauchbare Zwischenergebnisse, die eine Risikobetrachtung für den weiteren Ablauf zulassen.	In kurze Iterationen <sup>3</sup> einteilbar, die jedes Mal ein Inkrement erzeugen

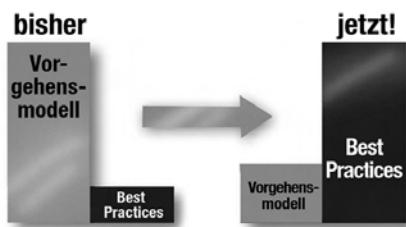
<sup>3</sup> Zu den Begriffen „Iteration“ und „Inkrement“ siehe [Oes00]

## 2.6 Agile System- und Softwareentwicklung

Viele bekannte Vorgehensmodelle geben Ihnen ein Maximum von Aktivitäten vor und legen Reihenfolgen dafür fest. Durch Anpassen („Tailoring“) können Sie Ihr persönliches Projektvorgehensmodell Maß schneidern. Dieser Anpassungsprozess erfolgt normalerweise zu Beginn eines Projekts. Danach wird der Maß geschneiderte Prozess durchgeführt.

Abbildung 2.6 zeigt das Umdenken in Richtung agile Vorgehensmodelle. Die umfangreiche Prozessvorgabe wird durch einige wenige wichtige Phasen, Disziplinen und Aktivitäten ersetzt, wie sie in den Abbildungen 2.4 und 2.5 gezeigt wurden. Statt die Phasen und Disziplinen jetzt akribisch zu detaillieren, geben wir Ihnen pro Phase und Disziplin die Ziele und nur einige, wenige essenzielle Aktivitäten vor, dazu aber eine reichhaltige Sammlung von bewährten Praktiken. Die Aufgabe jedes am Projekt Beteiligten ist es, zu jedem Zeitpunkt des ganzen Projekts immer situationsgerecht zu entscheiden, was davon am ehesten zum Ziel führt.

zielorientiert und situationsgerecht



**Abbildung 2.6:** Umdenken bei Vorgehensmodellen

Agile Prozesse verlangen von Ihnen, rege, beweglich und dynamisch zu sein. Sie sollten stets situationsgerecht, flexibel und angemessen handeln. Machen Sie nur soviel wie nötig, so wenig wie möglich. Agile Prozesse fordern Ihr Mitdenken, nicht „Dienst nach Vorschrift“ oder Einhaltung von Dogmen.

Keines der heute verfügbaren Vorgehensmodelle passt für alle Arten von Projekten und alle Anwendungsdomänen. Agiles Vorgehen wird dadurch passend, dass Sie kontinuierlich die jeweilige Projektsituation beurteilen und dann geeignete Schritte wählen.

Die Grundlage für die Beurteilung und die Entscheidungen liefern Ihnen folgende Maximen:

Maximen für agiles Handeln

- **eher offen für Änderungen als starres Festhalten an Plänen**
- **eher Menschen und Kommunikation als Prozesse und Tools**
- **eher ergebnisorientiert als prozessorientiert**
- **eher „darüber miteinander reden“ als „gegeneinander schreiben“**
- **eher Vertrauen als Kontrolle**
- **eher „Best Practices“ aus Erfahrung als verordnete Vorgaben**
- **eher Angemessenheit als Extremismus**

Sind die Maximen realistisch?

Seien wir einmal ehrlich: Für viele Organisationen ist ein Wechsel zu diesen Maximen eine undenkbare Revolution. „Wir brauchen doch Pläne!“ „Ohne Prozesse und Werkzeuge würde unsere Entwicklung zusammenbrechen.“ „Kontrolle muss sein, wir können nicht jedem bedenkenlos trauen.“

Lesen Sie die Maximen nochmals aufmerksam. Sie besagen nicht, dass Pläne schlecht sind. Sie besagen auch nicht, dass wir nichts mehr schriftlich festhalten. Sie fordern auch nicht, dass wir Kontrolle im Projekt aufgeben sollen. Sie sagen nur, dass wir im Zweifelsfall eher eine Sache vor einer anderen vorziehen würden.<sup>4</sup>

Keine Anarchie

Ein Befolgen dieser Maximen gibt den Entwicklern Freiheitsgrade, ohne Projekte in die Anarchie zu stürzen. Denn agile Vorgehensmodelle liefern Ergebnisse, nur unterscheiden sich diese für unterschiedliche Projekte in Anzahl, Tiefgang und Formalismus. Agile Entwicklung kennt auch Prozesse, Disziplinen und Aktivitäten, wie wir in den Abbildungen 2.4 und 2.5 gezeigt haben. Nur lassen diese mehr Spielraum für Alternativen und Kreativität. Agile Methoden setzen auf Verantwortung; es werden jedoch nur die Rollen im Projekt besetzt, die wirklich notwendig sind, um die Ziele zu erreichen.

Wir analysieren die Risiken in den einzelnen Disziplinen und empfehlen Aktivitäten nur dann durchzuführen, wenn sie diese Risiken minimieren. Unser Vorgehen ist eher zielorientiert als aktivitätsorientiert. Wir zeigen Ihnen oft alternative Wege zur Zielerreichung und geben Ihnen Kriterien an die Hand, welcher Weg unter welchen Randbedingungen eingeschlagen werden sollte. Diese agile Vorgehensweise baut weniger auf zeitlichen Abhängigkeiten auf, sondern nutzt Ergebnisse aus dem bisherigen Projektverlauf und neu eintreffende externe Erkenntnisse zur Gestaltung des weiteren Prozesses.

<sup>4</sup> Zugegeben: Langfristig führt die Beachtung agiler Maximen zu einer völlig neuen Entwicklungskultur. Die Maximen ändern das Wertesystem grundlegend. Lesen Sie nochmals das Vorwort von Tom DeMarco über die Erfolge unseres bisherigen Wertesystems.

## 2.7 Der Prozess und die Herausforderungen von RTE-Systemen

Die Aufteilung in Problemstellung und Lösung auf beiden Ebenen hilft Ihnen, die Herausforderungen von RTE-Systemen, die wir in Kapitel 1 aufgezeigt haben, gezielt zu bewältigen. Tabelle 2.2 zeigt daher – getrennt für Problemstellung und Lösung – für die fünf zentralen Eigenschaften von RTE-Systemen grob, wann und durch welche Entwicklungsschritte sie gemeistert werden. Für fast jeden Schritt gibt es alternative Möglichkeiten der Durchführung und Darstellung. In den genannten Kapiteln finden Sie Vorschläge für die agile, problemgerechte Behandlung der Herausforderungen.

**Tabelle 2.2:** Aktivitäten zur Bewältigung der RTE-spezifischen Herausforderungen

	Problemstellung	Lösung
<b>Einbettung</b>	Eigenes System oder Produkt klar abgrenzen, Nachbarsysteme identifizieren; Logische Schnittstellen (d. h. Ein-/Ausgaben) zu diesen Nachbarsystemen festlegen. → Kap. 3, 5, 7, 9	Physikalische Schnittstellen zu den Nachbarsystemen festlegen, d. h. Kanäle, Protokolle, Schnittstellenstandards für die Kommunikation mit Nachbarsystemen entscheiden und modellieren. → Kap. 4, 6, 8, 10
<b>Verteilung</b>	Vorgaben und Randbedingungen zur Verteilung erfassen → Kap. 4, 6	Zweistufiges Verfahren: Verteilung auf Standorte und Prozessoren festlegen → Kap. 6 Innerhalb eines Prozessors Verteilung von Funktionalität und Daten auf Softwarekomponenten entscheiden. → Kap. 6, 8, 10
<b>Zeitanforderungen</b>	Zeitanforderungen systematisch erfassen (Reaktionszeiten, Häufigkeiten von Ein-/Ausgaben.) → Kap. 6 Modellieren von externem Zeitverhalten. → Kap. 3–10	Durch Designentscheidungen: Budgetieren von Zeitkontingenten auf einzelne Komponenten. Dadurch entstehen abgeleitete Zeitanforderungen für die einzelnen Komponenten. → Kap. 6, 8, 10

<b>Parallele Prozesse</b>	Ausnutzung der natürlichen Parallelität unserer Systeme durch Modellierung der gewünschten Reaktionen auf unabhängige, externe Ereignisse. Dies erfolgt sowohl auf Systemebene (System-Use-Cases) wie auf Softwareebene (Software-Use-Cases) → Kap. 3, 5, 7, 9	Betrachtung der verfügbaren Ressourcen (Rechner, Prozesse auf Rechnern) zur Festlegung, wie weit die natürliche Parallelität beibehalten werden kann, wo weiter sequenzialisiert werden muss und wo sogar intern mehr Parallelität genutzt werden kann. Modellierung in Form des Tasking-Modells mit aktiven Objekten. → Kap. 6, 10
<b>Datenhaltung</b>	Definition aller wichtigen „Dinge“ aus der Anwendungswelt als Voraussetzung für die Datenmodellierung. Abbildung auf Entity-Klassen-Modelle, um Struktur in die Vielzahl der Begriffe zu bekommen. → Kap. 3, 5, 7, 9	Abbildung der Entity-Klassen-Modelle auf eigene Schichten oder Komponenten der Architektur, je nach Auswahl der Technologie. Einbeziehung anderer Klassen, die persistente Daten benötigen, wie z. B. Zustandsinformationen bei langlaufenden Prozessen. → Kap. 6, 8, 10

## 2.8 Wie sieht Ihr Entwicklungsprozess aus?

- Ist in Ihrem Unternehmen die Verantwortung für das Gesamtsystem und die Verantwortung für die Entwicklung der SW-Anteile klar geregelt?
- Trennen Sie zwischen Problemstellung und Lösung?
- Benutzen Sie Mustervorlagen für die Ergebnisse der einzelnen Entwicklungsaktivitäten?
- Wie weit sind die agilen Maximen in Ihrem Entwicklungsprozess bereits berücksichtigt?
- Erlauben Sie Ihren Entwicklern, die wichtigen Entwicklungsschritte mit den jeweils besten Methoden und Techniken durchzuführen und die Ergebnisse auf verschiedenen Wegen zu erreichen?

„Jedes große historische Geschehen begann als Utopie und endete als Realität.“

*Richard Nikolaus von Coudenhove-Kalergi  
(1894–1972), Schriftsteller u. Politiker,  
Begründer der Paneuropa-Bewegung*

# Teil II

## Der Produkt- und Systementwicklungszyklus

Abbildung II.1 zeigt die wichtigsten Aktivitäten und Ergebnisse der Systemkonzeption. Diese Abbildung ist Ihr roter Faden durch die Kapitel 3 und 4.

Systemkonzeption: mehr in Kapitel 3 und 4

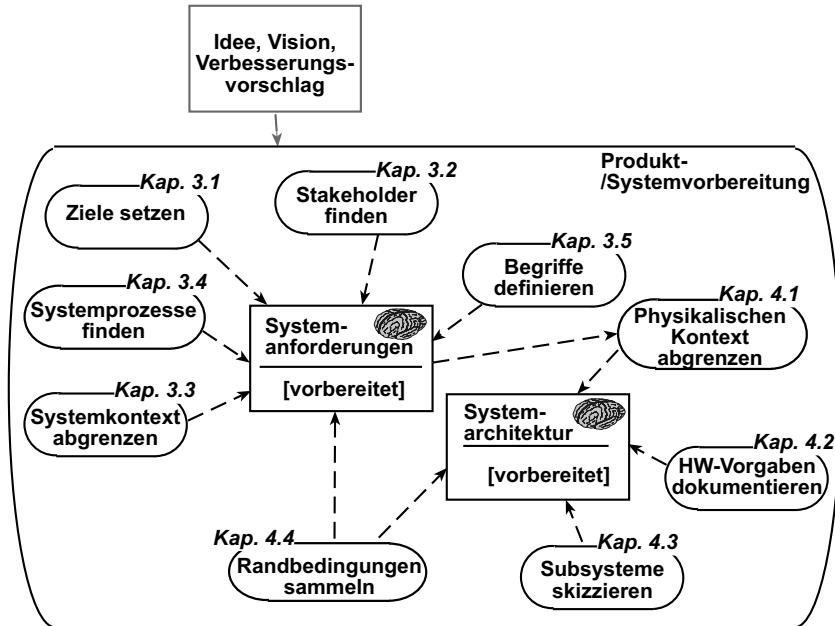


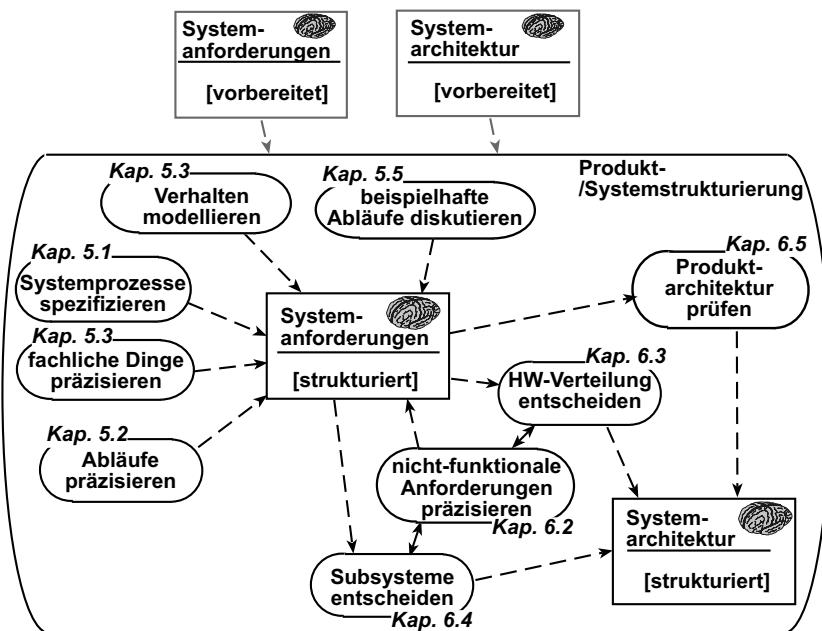
Abbildung II.1: Die Aktivitäten und Ergebnisse der Produkt-/Systemvorbereitung

## Teil II: Der Produkt- und Systementwicklungszyklus

Ziel der Konzeption ist es, die Produkt- oder Systemziele und die Randbedingungen für die Entwicklung zu kennen. Außerdem wollen wir das System so weit gliedern, dass wir die nächste Phase – die Systemstrukturierung – planen können.

Systemstrukturierung:  
mehr in Kapitel  
5 und 6

Ziel der nächsten Phase ist es, die Anforderungen an das Gesamtsystem soweit zu verstehen, dass wir uns für eine Systemarchitektur entscheiden und sie modellieren können. Abbildung II.2 zeigt die dafür notwendigen Schritte und ergänzt im Anforderungs- und im Architekturgehirn eine Menge an präziseren Spezifikationen. Diese Abbildung ist Ihr roter Faden durch die Kapitel 5 und 6.



**Abbildung II.2:** Die Aktivitäten und Ergebnisse der Produkt-/Systemstrukturierung

Selbst wenn Sie „nur“ Software für ein RTE-System entwickeln, sollten Sie den Teil II des Buches nicht überschlagen. Zumindest sollten Sie die für diese Ebene verantwortlichen Personen kennen und mit ihnen sprechen. Wenn Sie diese dazu bewegen können, auch mit UML-Modellen oder den anderen hier vorgestellten Hilfsmitteln zu arbeiten, werden Ihre Vorgaben für die Entwicklung der Softwareanteile kontinuierlich besser und die Zusammenhänge im Großen transparenter.

„Wenn dich ein Laie nicht versteht,  
so heißt das noch lange nicht,  
dass du ein Fachmann bist!“

*Hermann Frietsch, dt. Unternehmensberater*

# 3

## **Erste Systemanforderungen**

### **Fragen, die dieses Kapitel beantwortet:**

- Was muss man sehr früh für eine Produkt-/Systementwicklung klären?
- Wie kann man ein großes, komplexes Problem beherrschbar machen?
- Wie findet man die Produktziele und wie stimmt man sie mit allen relevanten Projektbeteiligten ab?

Tim Lister<sup>1</sup> sagt: „Viele Projekte scheitern, bevor sie begonnen haben“. Unsere Beratungspraxis bestätigt (leider) nur zu oft, dass er damit Recht hat. In diesem Kapitel konzentrieren wir uns auf die kurze Phase zu Beginn einer Produktentwicklung – auf wenige, aber entscheidende Stunden oder Tage. Auch wenn Sie kein anderes Kapitel in diesem Buch lesen und nur die Ideen von Kapitel 3 systematisch in die Praxis umsetzen, hat sich der Preis des Buches für Sie wahrscheinlich schon gelohnt.

Dieses Kapitel beschreibt die Vorbereitungen des Entwicklungsprozesses auf der Produkt-/Systemebene, um ein klares Problemverständnis zu erreichen. Auf dieser Basis werden im Folgenden die Anforderungen an das RTE-System präzisiert und eine Aufteilung in Hard- und Softwareteile vorgenommen.

Bevor Sie den Kontext Ihres Systems und die wesentlichen Systemprozesse analysieren können, müssen Sie sich Gedanken über die Projektziele und die am Projekt beteiligten Personen machen. Während des gesamten Projekts werden Begriffe definiert, Randbedingungen des Systems dokumentiert und die Stakeholderlisten und Begriffsdefinitionen aktualisiert. Wie Sie diese Aktivitäten vornehmen, was Sie dabei beachten müssen und wie sich die Eigenschaften Ihres RTE-Systems dabei auswirken, erläutert dieses Kapitel.

Vorbereitung

<sup>1</sup> Principal der Atlantic Systems Guild und Koautor von [DeM99]

Es gibt keine feste Reihenfolge dafür, wie die hier beschriebenen Aktivitäten abgearbeitet werden müssen. In Abbildung II.1 ist aus gutem Grund keine Reihenfolge vorgegeben. Wann Sie welche Aktivität durchführen, ist Ihnen freigestellt. Beschäftigen Sie sich zu jedem Zeitpunkt mit den Schritten, die den größten Erfolg und Projektfortschritt versprechen. Allerdings sollten Sie darauf achten, nicht mehr als 10% des gesamten Aufwands der Systementwicklung für die in den Kapiteln 3 und 4 beschriebenen Aktivitäten zu spendieren.

## 3.1 Ziele setzen

### Was ist ein Ziel?

Am Anfang einer Systementwicklung müssen die Ziele des Systems gefunden und dokumentiert werden. Unter einem Ziel wird „ein erstrebenswerter Zustand verstanden, der in der Zukunft liegt und dessen Eintritt von bestimmten Handlungen bzw. Unterlassungen abhängig ist, der also nicht automatisch eintreffe“. [GA00]

Da wir uns hier noch ganz am Anfang der Systementwicklung befinden, die einen Zeitrahmen von einigen Monaten bis zu mehreren Jahren umfassen kann, sind die Ziele teilweise noch unscharf und sollten eventuell eher als Visionen bezeichnet werden. Bezüglich der Granularität und Exaktheit unterscheiden sie sich stark von den Zielen, die wir später am Beginn der Softwareentwicklung fordern werden. Aus den dann vorliegenden Systemanforderungen und der Systemarchitektur können die Ziele für die Softwareebene abgeleitet und deutlich präziser definiert werden.

Bei der Zielsetzung für das Gesamtsystem sollte keinesfalls eine Aufteilung des Systems in Hard- und Software vorweggenommen werden. Hierfür ist erst eine fundierte Entscheidungsgrundlage zu legen.

### Warum Ziele?

Das Festlegen der Ziele für das Produkt oder Gesamtsystem hat einen starken Einfluss auf den Erfolg der Systementwicklung. Unklare oder nicht definierte Ziele beeinträchtigen vor allem die Motivation der Mitarbeiter. Häufig arbeiten Projektteams monatelang sinnlos für ein vermutetes oder selbst erfundenes Ziel. Nichtdefinierte Ziele sind zudem nicht verfolgbar und werden daher meist nicht erreicht.

Die Zielfindung ist für den Erfolg eines Projekts essenziell, weshalb keine Produkt-/Systementwicklung ohne wenigstens eine halbe Seite schriftlicher Ziele angegangen werden sollte. Dabei ist es wichtig, quantifizierbare Angaben aufzuzählen. Dies geschieht meist in natürlichsprachlicher Form. Einige Möglichkeiten, Anforderungen messbar zu machen und ihren Nutzen für das Projekt zu vergleichen, stellt der Artikel „Requirements: Made to Measure“ [Rob97] vor. Zum Prozess der Zielfindung empfehlen die meisten Vorgehensmodelle, eine Analyse der Ist-Situation durchzuführen, dann die Probleme der bestehenden Situation beziehungsweise existierende Optimierungsverfahren und Visionen herauszuarbeiten und danach den Zielzustand zu definieren.

### Wie finde ich Ziele?

So einfach stellt sich das in der Projektrealität der RTE-Systementwicklung leider nicht immer dar. Das Vorgehen, wie Sie zu Ihren Zielen gelangen, hängt stark von den gegebenen Rahmenbedingungen ab.

- Erfinden Sie ein neues Produkt? Dann ist bei der Zielfindung vor allem visionäres Denken und eine Offenheit bezüglich aller erdenklichen Lösungen gefragt. Die Ziele müssen aus der Sicht des potenziellen Kundenkreises formuliert werden und den möglichen Kundennutzen völlig lösungsneutral in den Vordergrund stellen.
- Lösen Sie gerade die 17. Generation eines Systems durch die 18. ab? Dann sollten Sie sich vor allem um mögliche Optimierungen kümmern und dürfen die bestehende Einbettung des Systems nicht außer Acht lassen. Vergessen Sie hier nicht die folgenden Fragen: Stellt die Einhaltung vorhandener Schnittstellen zu Ihren Nachbarsystemen ein Dogma dar? Oder lässt sich daran im Sinne Ihres Systems etwas optimieren? Vor allem bei RTE-Systemen mit komplexer Umgebung werden Ihnen an dieser Stelle oft von vorne herein die Schnittstellen von der Umwelt diktiert, welche die Ziele deutlich einschränken.

Mehr dazu  
in Abschnitt  
3.3 und 4.1

Ziele für die Systementwicklung bilden eine unabdingbare Basis. Für weitere Informationen zu diesem Thema lesen Sie [GA00] und [Rupp02]. Die Art, wie Ziele dokumentiert werden, reicht von lockerer Prosa bis zu stark formalisierten, musterbasierten Beschreibungen. Wichtig ist hierbei vor allem die Akzeptanz aller Projektbeteiligten.

## 3.2 Stakeholder finden

Stakeholder sind alle Personen, die von der Systementwicklung und natürlich auch vom Einsatz und Betrieb des Systems betroffen sind. Dazu gehören auch Personen, die nicht in der Entwicklung mitwirken, aber das neue System zum Beispiel nutzen, in Betrieb halten oder schulen. Wir verwenden hier den englischen Begriff Stakeholder, da die deutschen Begriffe „Systembeteiligte“ und „Systembetroffene“ entweder nicht alle Personen umfassen oder einen passiven oder negativen Beigeschmack haben<sup>2</sup>. Stakeholder sind die Informationslieferanten für Ziele, Anforderungen und Randbedingungen an unser System oder Produkt.

Was ist ein  
Stakeholder?

Dokumentieren Sie zu Beginn einer Systementwicklung alle Stakeholder, die Sie bereits kennen. Die anschließende Analyse des Systems bietet kontinuierlich mehr Aufschluss über Personen, die für die weiteren Schritte essenziell sind. Das Finden und Dokumentieren von Stakeholdern ist kein einmaliger Vorgang, sondern die Liste relevanter Stakeholder muss immer wieder aktualisiert werden.

---

<sup>2</sup> Weitere Begriffe für Stakeholder sind:  
Wissensträger, Interessenvertreter, Platzhirsch, Interessen- und Anspruchsgruppen.

### 3 Erste Systemanforderungen

Die folgende Tabelle gibt einen kurzen Überblick über potenzielle Rollen von Stakeholdern. Für Ihr System sollten Sie jede Rolle bedenken und konkrete Personen dafür suchen. Diese werden dann als Verantwortliche notiert und in geeigneter Form in die Systementwicklung integriert. Detailliertere Informationen hierzu finden Sie in [Rupp02] und [ASG02].

**Tabelle 3.1:** Rollen für Stakeholder

Rolle der Stakeholder	Beschreibung
Management	Gruppe der Sponsoren/Auftraggeber und Entscheider. Das Management muss dafür sorgen, dass das System die Unternehmensziele und -strategien unterstützt und mit der Unternehmensphilosophie konform geht.
Anwender des Systems	Sie liefern einen Großteil der fachlichen Ziele. Die Anwenderrepräsentanten benötigen viel Erfahrung in der Domäne, müssen das Vertrauen der restlichen Anwender genießen und über Weitblick für zukünftige Systeme/Produkte verfügen.
Wartungs- und Servicepersonal des Systems	Sie formulieren im Wesentlichen Ziele für die Wartung und den Service des Systems.
Schulungs- und Trainingspersonal	Für das Schulungs- und Trainingspersonal stehen Aspekte wie Bedienbarkeit, Vermittelbarkeit und Dokumentation des Systems im Vordergrund.
Käufer des Systems	Der Käufer des Systems ist nicht unbedingt mit dem Anwender identisch. Die Frage: „Wer trifft die Kaufentscheidung über das System?“ ist wesentlich, um die Gruppe der betreffenden Stakeholder zu ermitteln.
Marketing- und Vertriebsabteilung	Marketing und Vertrieb spielen häufig die Rolle des internen Repräsentanten der externen Kunden. Insbesondere bei der Produktentwicklung sind sie wichtige Ziel- und Anforderungslieferanten.
Entwickler	Sie liefern technologiespezifische Ziele, die sich meist auf den Entwicklungsprozess beziehen.
Projekt- und Produktgegner	Bereits zu Beginn der Zielfindung ist es sinnvoll, sich Gedanken über potenzielle Gegner zu machen. Jedes Ziel besitzt das Potenzial, Machtpositionen und Gewohntes in Frage zu stellen.
Sicherheitsbeauftragte	Diese Personengruppe stellt Anforderungen an das System, die aus dem absichtlichen oder unabsichtlichen Fehlverhalten anderer Stakeholder resultieren.
Personen aus anderen Kulturräumen	Sie bestimmen die Rahmenbedingungen, z. B. die Darstellung der Informationen auf der Oberfläche, Verwendung von Symbolen und Begriffen.
Gesetzgeber	Die Festlegung der rechtlichen Rahmenbedingungen wird beeinflusst durch Gesetze, Vorschriften und Verordnungen.
Standardisierungsgremien	Vorhandene oder zukünftige, externe wie interne Standards setzen Randbedingungen.
Meinungsführer und öffentliche Meinung	Meinungsführer können z. B. marktdominierende Konkurrenten sein. Problematisch wird es dann, wenn die öffentliche Meinung in den Märkten (bei Anstrengung mehrerer Zielmärkte) stark differiert.
Prüfer und Auditoren	Falls es Gruppen gibt, die das System prüfen, freigeben oder abnehmen müssen, ist es notwendig, die Ziele auf Konformität mit deren Richtlinien zu prüfen.

Die einfachste Art, Stakeholder systematisch zu erfassen, sind Tabellen.

**Tabelle 3.2:** Notationsvorschlag für Stakeholder

Rolle der Stakeholder	Beschreibung	Konkrete Vertreter	Verfügbarkeit	Wissensgebiet	Begründung
Anwender	Sind die eigentlichen Benutzer des Systems	Herr Meier Tel.: 0815 E-Mail: Meier@bl.de	Urlaub vom 20.12.01 bis 07.01.02; 20 % verfügbar	Arbeitet mit Altsystem, kennt Schwachstellen	Anwender des Systems, muss damit zukünftig arbeiten.

### Warum sind die Stakeholder so wichtig?

Die Entwicklung eines Systems hat das Ziel, die Bedürfnisse mehrerer Personen oder Gruppen zu befriedigen, wobei die Bedürfnisse und Ansprüche sehr unterschiedlich, auch gegenläufig und widersprüchlich sein können.

Da die Definition der Systemziele und -anforderungen den Erfolg der Systementwicklung stark beeinflusst und größtenteils vorherbestimmt, dürfen sie nicht nur von einer Person, sondern müssen von vielen Stakeholdern festgelegt werden. Nur dadurch können Sie alle erdenklichen Arten von Zielen und Anforderungen sammeln und abgleichen. Natürlich können nicht alle vom System betroffenen Personen an allen Diskussionen beteiligt werden. Aus diesem Grund muss für jede Gruppe von Stakeholdern ein oder mehrere Repräsentanten ausgewählt werden. In einigen Fällen ist es schwierig, den direkten Zugang zu den Personen zu gewinnen, die das System eigentlich nutzen werden; insbesondere dann, wenn es sich dabei um ein Produkt handelt, welches nicht für spezielle Nutzer, sondern für den „Markt“ erstellt wird.

Bei der Wahl der Repräsentanten sollten Sie darauf achten, dass diese ein möglichst präzises und aktuelles Verständnis der wirklichen Bedürfnisse des Marktes und der späteren Nutzer haben und dieses auch kommunizieren können. Somit eignen sich Personen, die seit Jahren den Produktbezug verloren haben und die heutige Realität nur noch aus Erzählungen kennen, genau so wenig, wie Spezialisten, die ihr Wissen leider nicht kommunizieren können.

Werden wichtige Stakeholder vergessen und zum Beispiel erst im Rahmen der Inbetriebnahme mit dem System konfrontiert, so erfahren Sie von deren Anforderungen viel zu spät. Die neu hinzukommenden Anforderungen können Sie dann nur noch aufwändig über ein Änderungsverfahren integrieren oder einfach ignorieren. Dies kostet erfahrungsgemäß mehr Zeit und Geld und führt häufig zu einem Akzeptanz- und Imageverlust.

Die Wahl der Stakeholder

Vergessene Stakeholder sind vergessene Anforderungen

## 3.3 System-/Produktkontext abgrenzen

Wir haben bereits festgestellt, dass RTE-Systeme mehr beinhalten als nur Software und meist intensiv mit ihrer Umgebung kommunizieren – teilweise sogar baulich in ihr verankert sind. Bei der Systemkonzeption muss daher ex-

plizit geklärt werden, was außerhalb und was innerhalb der Systemgrenzen liegt. Die Ergebnisse dieser Überlegung bezeichnen wir als Kontextdiagramm.

#### Kontextdiagramm

Das Kontextdiagramm wurde in der Strukturierten Analyse [DeM79] eingeführt, um das zu entwickelnde System von seiner Umgebung abzugrenzen. Dieser Schritt ist eine wichtige Voraussetzung für alle weiteren Aktivitäten. Als Entwickler müssen Sie wissen, wo die Grenzen sind, innerhalb derer Sie das System strukturieren und entwerfen dürfen.

Das Kontextdiagramm enthält folgende Elemente:

- das zu entwickelnde System als eine Black Box,
- die Nachbarsysteme, mit denen das System in irgendeiner Weise zusammenarbeitet,
- die Schnittstellen zwischen der Black Box und den Nachbarsystemen.

Die UML stellt uns zwar (viel zu) viele Diagrammartarten zur Verfügung, enthält aber kein Kontextdiagramm. Zahlreiche Autoren, die auf die Systemabgrenzung und Schnittstellenpräzisierung gegenüber der Umwelt viel Wert legen, simulieren dieses Diagramm durch UML-konforme Diagrammtypen.

#### Arten der Kontextbildung

Bei der Abgrenzung des Kontexts unterscheiden wir zwischen logischem und physikalischem Kontext. Beim logischen Kontext liegt der Fokus auf der Kommunikation mit den Nachbarsystemen, beim physikalischen Kontext auf den physikalischen Kanälen und Übertragungsmedien, die das System mit den Nachbarsystemen verbindet.

**Tabelle 3.3: Arten der Kontextbildung**

Art der Kontextbildung	Schnittstelle	UML-Diagramm	
logisch	unspezifiziert	Use-Case-Diagramm	
	Ein-/Ausgaben	Klassendiagramm	
	Nachrichten	Sequenzdiagramm	
physikalisch	Kanäle	Verteilungsdiagramm	

Welche dieser vier Diagrammartarten Sie wählen, hängt vom Typ des Systems und des von Ihnen angestrebten Schwerpunkts der Kontextabgrenzung ab. In diesem Kapitel diskutieren wir die Möglichkeiten zur logischen Kontextbildung. Die physikalische Kontextbildung mit Hilfe von Verteilungsdiagrammen finden Sie in Kapitel 4.

### 3.3.1 Identifizierung von Nachbarsystemen

Egal, welche Art der Darstellung Sie für die Kontextabgrenzung wählen: Sie müssen auf jeden Fall die Nachbarsysteme Ihres Systems identifizieren.

Das gestaltet sich dann besonders einfach, wenn Sie eine stabile Umgebung oder Integrationslandschaft vorfinden. Können Sie sicher sein, dass die Nachbarsysteme dauerhaft stabil bleiben, dann übernehmen Sie diese 1:1 als Nachbarsysteme in das Kontextdiagramm. Sofern die technischen Schnittstellen Ihres Systems zu den externen Systemen bereits feststehen und dauerhaft stabil bleiben, ist es sinnvoll, die Schnittstellen so weit herunterzubrechen, dass sie jeweils genau ein externes System bedienen.

Abgrenzung  
gegen stabile  
Umgebung

Ist jedoch damit zu rechnen, dass bei einer zukünftigen Produktgeneration das Nachbarsystem ersetzt wird oder dessen Aufgaben durch ein anderes System ausgeführt werden, sollten Sie die Systemumgebung unter fachlichen Gesichtspunkten modellieren. Die Bezeichnungen der Nachbarsysteme sollten dann die fachliche Aufgabe des Systems beschreiben und nicht den derzeitigen Systemnamen wiedergeben.

Instabile  
Systemumgebung  
logisch abgrenzen

Hier ein Beispiel zur Verdeutlichung: Ein Navigationssystem muss zur Routenberechnung den derzeitigen Standort des Fahrzeugs wissen. Dies geschieht in aller Regel durch Angabe von Längen- und Breitengrad. Die Quelle dieser Auskunft und damit das Nachbarsystem kann aber variieren, es wäre ein GPS-Modul, ein GSM-Empfänger (Mobiltelefon) oder aber auch die Ortung über eine direkte Satellitenverbindung denkbar. Um unabhängig vom konkret vorhandenen Nachbarsystem modellieren zu können, bietet sich eine fachliche Paketierung zu einem logischen Nachbarsystem „Positionsdatensystem“ an. Um welche Art von System zur Ermittlung der Positionsdaten es sich nun konkret handelt, ist für die Kontextabgrenzung irrelevant, solange die Schnittstelle (in diesem Fall die übertragenen geografischen Daten) konstant bleibt.

Eine fachliche Paketierung der Nachbarsysteme zu logischen Einheiten ist sicherlich länger stabil, als die Darstellung der real existierenden Schnittstellen Ihres Systems zur derzeitigen Umwelt. Eine Darstellung der derzeitigen Fakten liefert Ihnen aber mehr konkrete Informationen bezüglich der Umwelt, die Sie beim Einsatz Ihres Systems antreffen werden. Auch hier sollten Sie im Sinne eines agilen Vorgehens abwägen, welcher Gesichtspunkt für Sie mehr Bedeutung besitzt und für Ihre Systementwicklung mehr Klarheit erzeugt.

### 3.3.2 Kontextabgrenzung mittels Use-Case-Diagramm

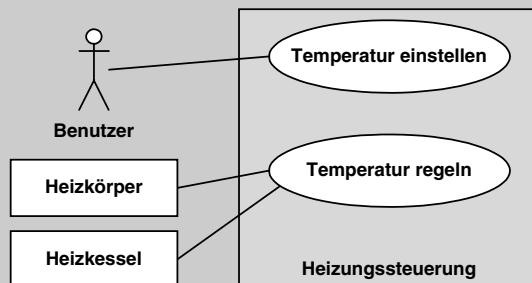
Die Kontextabgrenzung mittels Use-Case-Diagrammen ist der von vielen Vorgehensmodellen empfohlene Einstieg in die Systementwicklung.

Standard-  
vorgehen

# Use-Case-Diagramme

Ein **Use-Case-Diagramm** ist die von Ivar Jacobson vorgeschlagene Notation, um den Zusammenhang von System und Umwelt darzustellen. Ein Use-Case-Diagramm enthält:

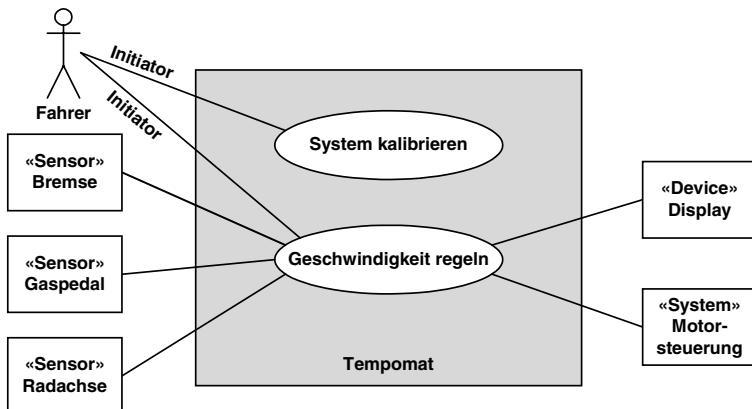
- das System in Form eines Rechtecks,
  - die Systemprozesse in Form von Ellipsen,
  - Akteure in der Systemumgebung, die Prozesse des Systems auslösen, dargestellt als Strichmännchen,
  - eventuell weitere Nachbarsysteme, dargestellt als Strichmännchen oder als Klassen,
  - Assoziationen zwischen Akteuren bzw. Nachbarsystemen und den Systemprozessen.



**Abbildung 3.1:** Use-Case-Diagramm

Im Rahmen der Kontextabgrenzung schlagen wir Ihnen folgende Konventionen für die Nutzung von Use-Case-Diagrammen vor, die teilweise über die UML-Standardempfehlungen hinausgehen:

- Benennen Sie Ihr System, um dem Gegenstand Ihrer Entwicklung Identität zu geben.
  - Zeichnen Sie nicht nur Akteure ein, d. h. nicht nur die Nachbarsysteme, die Prozesse auslösen, sondern **alle** Nachbarsysteme, mit denen Ihr System kommuniziert. Viele UML-Bücher suggerieren, nur die Akteure einzuziehen, die Prozesse auslösen. Bei der Verwendung als Kontextdiagramm müssen auch Nachbarsysteme dargestellt werden, die nur Eingaben an unser System liefern oder Ergebnisse empfangen.
  - Kennzeichnen Sie die echten Akteure als Initiator.
  - Verwenden Sie Strichmännchen nur für Menschen, die mit dem System kommunizieren.
  - Nutzen Sie das Klassensymbol für andere Arten von Nachbarsystemen. Spezifische Stereotypen stellen wir in Abschnitt 3.4.1 vor.
  - Das Einzeichnen der Systemprozesse ist eine syntaktische Notwendigkeit, auf die wir im nächsten Abschnitt eingehen. Abbildung 3.2 zeigt zunächst ein Use-Case-Diagramm zur Kontextabgrenzung.



**Abbildung 3.2:** Kontextabgrenzung mit Use-Case-Diagramm

Die Abbildung verdeutlicht die Entscheidungen über die Systemgrenze: Die Bremse ist explizit kein Bestandteil des Tempomaten, wohingegen die Tastatur, mit der der Fahrer die Kommandos eingibt, in das System hineinverlagert wurde.

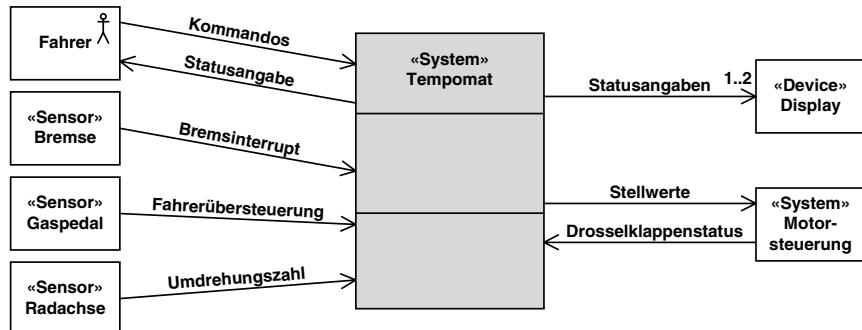
Informationen über die Ein- und Ausgaben, die zwischen dem System und seiner Umgebung ausgetauscht werden, werden im Use-Case-Diagramm normalerweise weggelassen oder formlos als Notizen an den Assoziationen notiert. Die UML bietet zwar Möglichkeiten zur genaueren Spezifikation der Assoziation zwischen Systemumgebung und System. Diese werden in Projekten aber nur selten in der erforderlichen Präzision angewandt. Grundsätzlich kann die Beziehung zwischen Use-Case und Akteur mit allen näheren Angaben einer Assoziation (zum Beispiel mit Multiplizitäten oder Rollen) versehen werden – wie im Klassendiagramm.

Use-Case-Diagramme sind bei der Modellierung des Kontexts für Sie interessant, wenn Sie zu diesem Zeitpunkt noch keine detaillierten Aussagen über die Schnittstellen treffen möchten. Wenn Sie lediglich Ihr System, die Nachbarsysteme und deren grundsätzliche Kommunikation darstellen wollen, sollten Sie auf diese Diagrammart zurückgreifen.

Use-Case-Diagramme zur Darstellung grundsätzlicher Kommunikation

### 3.3.3 Kontextabgrenzung mittels Klassendiagramm

Kontextdiagramme können auch mit Hilfe von Klassendiagrammen nachgebildet werden. Eine zentrale Klasse repräsentiert hierbei das zu erstellende System als Black Box. Wir kennzeichnen sie mit dem Stereotyp «System» oder «Produkt». Alle anderen Klassen repräsentieren die Systemumgebung, zu der unsere „Systemklasse“ eine Beziehung unterhält, und werden zum Beispiel als «System», «Device» oder «Sensor» stereotypisiert (vgl. [Gom00]).



**Abbildung 3.3:** Kontextabgrenzung mit Klassendiagramm

Nutzen Sie folgende Konventionen:

- Geben Sie der zentralen Klasse den Namen Ihres Systems.
- Achten Sie darauf, **alle** Nachbarsysteme, mit denen Ihr System kommuniziert, in das Diagramm aufzunehmen.
- Stereotypisieren Sie die Nachbarsysteme zwecks besserer Lesbarkeit.
- Verwenden Sie gerichtete Assoziationen für Ein- und Ausgaben.
- Beschriften Sie diese mit den Daten oder Ereignissen, die in das System ein- und ausfließen.
- Nutzen Sie Multiplizitäten, wenn Nachbarsysteme mehrfach auftreten. Denken Sie an ein Flugsicherungssystem, das mit vielen Flugzeugen und mehreren anderen Kontrollzentren kommuniziert.
- Falls Sie Annahmen über die Nachbarsysteme, vorgegebene Schnittstellen, Kapazitätsinformationen und ähnliches dokumentieren möchten, steht Ihnen die ganze Bandbreite der Modellierungsmöglichkeiten in Klassenmodellen zur Verfügung. Sie können Operationen und Attribute eintragen, Interfaces definieren, mit Tagged Values und Constraints arbeiten oder informelle Schnittstellenspezifikationen bei der Klasse hinterlegen.

Klassendiagramm zur Fokussierung auf Ein-/Ausgaben

Verwenden Sie diese Form der Darstellung, wenn Ihr Fokus auf der frühzeitigen Festlegung der logischen Ein-/Ausgaben liegt<sup>3</sup>.

#### 3.3.4 Kontextabgrenzung mittels Sequenzdiagramm

Auch Sequenzdiagramme können zur Abgrenzung des Kontexts eingesetzt werden. Dies lohnt sich vor allem bei Systemen, die wenig unterschiedliche, aber bereits fest definierte Kommunikationsabläufe mit ihrer Umwelt besitzen. Beachten Sie aber, dass Sequenzdiagramme Szenariencharakter haben: Jedes Sequenzdiagramm stellt immer nur eine mögliche Reihenfolge von Interaktionen dar.

<sup>3</sup> ... oder vielleicht nur, weil es dem guten alten Kontextdiagramm aus SA verblüffend ähnlich sieht.

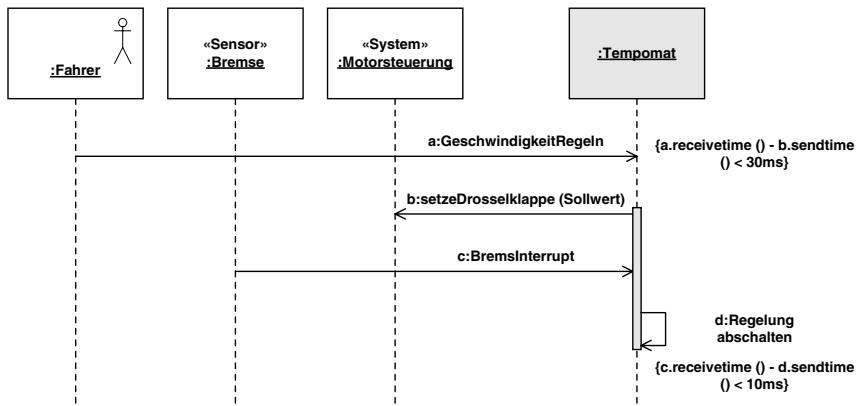
nen dar. Treten sehr unterschiedliche Interaktionsfolgen des abzugrenzenden Systems mit seiner Umwelt auf, so müssten viele Sequenzdiagramme erstellt werden. Dies ist insbesondere bei reaktiven Systemen oder bei RTE-Systemen mit vielen parallelen Prozessen der Fall. Hier empfiehlt sich, das Sequenzdiagramm nur zur detaillierten Diskussion weniger Szenarien kombiniert mit einer anderen Kontextdiagrammtechnik zu verwenden.

In dieser Form des Kontextdiagramms werden das zu erstellende System und alle seine Nachbarsysteme als Objekte auf den Lebenslinien notiert, die dann Nachrichten austauschen. Dabei kann die Reaktion unseres Systems auf externe Ereignisse, die Ein- und Ausgaben (als Parameter), sowie zeitliche Zusammenhänge spezifiziert werden. Aus diesem Grund eignet sich diese Diagrammtechnik sehr gut für RTE-Systeme mit signifikanten Zeitanforderungen. Bei der ersten Erstellung eines Kontextdiagramms werden Sie zeitliche Anforderungen eventuell noch nicht interessieren. Zu einem späteren Zeitpunkt kann das Diagramm dann allerdings auf einfache Weise um diese Anforderungen ergänzt werden.

Systeme  
statt Objekte

Nutzen Sie folgende Konventionen:

- Geben Sie dem zentralen Objekt den Namen Ihres Systems.
- Achten Sie darauf, dass **alle** Nachbarsysteme, mit denen Ihr System bei diesem Szenario Nachrichten austauscht, eingezeichnet werden.
- Stereotypisieren Sie die Nachbarsysteme zwecks besserer Lesbarkeit.
- Beschriften Sie die Nachrichten, die Ihr System erhält oder sendet, mit Namen und eventuell Parametern.
- Zeichnen Sie insbesondere die Nachrichten, die Sie zur Spezifikation von Reaktionszeiten benötigen.



**Abbildung 3.4:** Kontextabgrenzung mit Sequenzdiagramm

Abbildung 3.4 zeigt die Einbindung von geforderten Reaktionszeiten in ein Sequenzdiagramm. Die Buchstaben an den Nachrichten (a, b, ...) dienen der leichteren Referenzierbarkeit von Nachrichten in Zeitformeln.

## 3.4 Systemprozesse finden

Die meisten Systeme, die wir heute erstellen oder weiterentwickeln, umfassen viele Systemprozesse und sind relativ umfangreich und komplex – zu komplex, um sie „am Stück“ zu begreifen. Deshalb müssen wir unser System systematisch in mundgerechte, verdaubare Bissen zerteilen. Dazu hat Ivar Jacobson mit seiner Idee der Use-Cases (Systemprozesse) einen hervorragenden Beitrag geleistet<sup>4</sup>.

#### Zerlegung des Systems

Überlegen Sie, wie Sie bisher ein großes, komplexes Problem in handhabbare Teile zerlegt haben. Wahrscheinlich haben Sie es nach funktionalen Gesichtspunkten zerlegt. Ihr Gesamtsystem soll fünf oder sieben wichtige Funktionsblöcke abdecken. Vielleicht haben Sie nach Hardwareeinheiten und Verteilungsgesichtspunkten zerlegt (zum Beispiel Vorverarbeitung der Messdaten auf der Clientseite, anschließende Verifikation der Daten auf dem Server). Oder die bisherige Aufteilung in Hardware- und Softwarebestandteile war das Zerlegungskriterium eines Alt- oder Vorgängersystems.

Die Aufteilung in Komponenten, wie zum Beispiel Hardware oder Software, sollte jedoch erst **nach** der Analyse der grundlegenden Anforderungen geschehen. Würde eine derartige Aufteilung sofort erfolgen, so bestünde die Gefahr, dass Designentscheidungen aus der Vergangenheit bereits als unumstößliche Basis für die Entwicklung eines neuen Systems angesehen werden.

Alle oben genannten Zerlegungsansätze sind durch eine Innensicht des Systems bestimmt. Sie sprechen immer über interne Teile des Systems – egal, nach welchem Kriterium sie es zerlegen.

#### Der Blick von außen auf das System

Diesem Ansatz stellt Jacobson die Außenbetrachtung des Systems gegenüber – das System bleibt (zunächst) eine *Black Box*. Use-Case-Analyse bedeutet, dass wir Akteure *außerhalb* des Systems suchen und von diesen wissen wollen, was sie von unserem System erwarten. Durch diese Herangehensweise kommt man fast wie von selbst zu einer fachlichen, logischen Zerlegung des Systems, ohne über seine interne Struktur nachdenken zu müssen.

Die Idee von Ivar Jacobson lässt sich auf drei einfache Aussagen reduzieren:

- Akteure befinden sich außerhalb des Systems. Sie fordern etwas von dem System.

<sup>4</sup> Jacobson war weder der erste, noch der einzige, der diese Idee hatte. Bei den strukturierten Methoden haben McMenamin/Palmer bereits 1984 den gleichen Weg zur Beherrschung der Komplexität vorgeschlagen. Sie hatten es als ereignisorientierte Zerlegung bezeichnet. Im Bereich „Business Reengineering“ haben vor allem Hammer/Champy mit ihrem Buch „Reengineering the Corporation“ Furore gemacht. Auch dahinter steckt die gleiche Idee des Querdenkens durch ein System (in diesem Fall eine ganze Firma), um zunächst die essenziellen Abläufe herauszufinden, bevor man daran geht, diese zu automatisieren. Aber von allen war Ivar Jacobson am erfolgreichsten in der Vermarktung der Idee. Alleine der Begriff „Use-Case-Modellierung“ gilt heute als Kernbegriff aller objektorientierter Vorgehensweisen.

- *Use-Cases* stellen die Reaktion des Systems auf Ereignisse aus der Umwelt dar.
- *Assoziationen* beschreiben die Beziehung zwischen den Akteuren außerhalb des Systems und den Use-Cases innerhalb des Systems.

Als Jacobson Use-Cases einföhrte, dachte er an Softwareprozesse. Wir übertragen diese Idee auch auf die System-/Produktbene. System-Use-Cases stellen Systemprozesse quer durch Hard- und Software dar. Akteure in der Systemumgebung initiieren derartige Systemprozesse. Diese ganzheitliche Systembetrachtung mittels System-Use-Cases auf der Produkt-/Systemebene entscheidet bei RTE-Systemen über den Erfolg.

Übertragung  
der Use-Cases  
auf das System

Einer Aufteilung des Gesamtsystems oder Produkts in Software- oder Hardwarebestandteile für die Realisierung steht nichts entgegen. Doch sollte sie auf einer bewussten Designentscheidung basieren (siehe Kapitel 6 Subsystembildung). Anschließend hindert Sie nichts daran, das neu entstandene Teilsystem (zum Beispiel konkret den Softwareanteil) wiederum mittels eines Use-Cases – dann auf der Softwareebene – darzustellen.

### 3.4.1 Was sind Akteure?

Akteure befinden sich grundsätzlich außerhalb Ihres Systems und fordern etwas von Ihrem System. Jacobson hat dafür als Symbol ein Strichmännchen vorgeschlagen, was natürlich suggeriert, dass es sich bei Akteuren um Menschen handelt. Aus Sicht von vielen kommerziellen Systemen ist das auch durchaus zutreffend. Irgendjemand sitzt vor einem Rechner und fordert diesen auf, Aufgaben zu erledigen. Diese Sicht passt für Buchhalter, die Rechnungen kontieren, für Sachbearbeiter bei einer Versicherung, die einen Vertrag erstellen, oder für Bibliothekare, die Bücher verleihen.

Suchen wir doch einmal nach Akteuren für RTE-Systeme. Wer oder – und das muss man nun auch einschließen – was löst etwas in RTE-Systemen aus? Sicherlich sind auch bei diesen Systemen manchmal Menschen die Auslöser. Wenn Sie zum Beispiel zum Mobiltelefon greifen, um zu telefonieren, dann lösen Sie die Herstellung einer Verbindung zu einem Gesprächspartner aus. Für viele RTE-Systeme stellt der Akteur Mensch jedoch eher die Ausnahme dar. Es gibt mindestens vier weitere Kategorien von Auslösern, die für RTE-Systeme von Bedeutung sind:

Es gibt mehr als  
nur Agierende

- Sensoren,
- Ein- und Ausgabegeräte (Devices),
- Nachbarsysteme und
- die Zeit.

Sensoren außerhalb unseres Systems erfassen Informationen (z. B. Temperatur, Lichteinfall) und ermitteln Ereignisse (z. B. die Überschreitung von

### 3 Erste Systemanforderungen

Schwellwerten, Siedepunkten), die für unser System von signifikanter Bedeutung sind. Die von externen Sensoren gewonnenen Messergebnisse, die an unser System übermittelt werden, initiieren innerhalb unseres Systems die Ausführung von Systemprozessen. Wenn Sie den Sensor außerhalb Ihres Systems sehen, so sollten Sie ihn im Use-Case-Diagramm als Akteur darstellen. Betrachten Sie den Sensor jedoch als Teil Ihres Hard-/Softwaresystems, so erscheint als Akteur die physikalische Größe, die der Sensor misst (Druck, Temperatur, Lichteinfall usw.).

Werden die Sensoren von einem Menschen bedient (z. B. eine Person betritt einen Raum, was über eine Lichtschranke erfasst wird, oder die Person nimmt ein Telefongespräch an, indem sie die Gesprächsaufnahmetaste ihres Mobiltelefons drückt), so sollte statt der Technologie für die Eingabeermittlung (d. h. statt des Sensors) besser der menschliche Akteur modelliert werden. Damit schaffen Sie Freiheitsgrade für den Designer, der bewusst eine Entscheidung treffen kann, wodurch das Betreten des Raumes nun wirklich erfasst wird.

#### Spezielle Nachbarn

Wie der Begriff eingebettetes System bereits andeutet, sind RTE-Systeme häufig in andere Systeme eingebettet bzw. unterhalten viele Schnittstellen zu anderen Systemen (nicht nur zu menschlichen Nutzern). Dementsprechend treten Nachbarsysteme häufig als Initiatoren wichtiger Systemprozesse auf.

Wenn diese Nachbarsysteme relativ einfach sind, bezeichnen wir sie eher als Devices. Darunter verstehen wir Eingabegeräte, wie Joysticks, Scanner oder Mäuse und auch Ausgabegeräte, die nur Daten entgegennehmen und anzeigen, aber nicht weiterverarbeiten (Displays, LEDs).

Der Wortbestandteil „Real-Time“ bringt zum Ausdruck, dass die Zeit bei RTE-Systemen eine entscheidende Rolle spielt. Systemprozesse werden häufig zu definierten Zeitpunkten oder an definierten Terminen ausgelöst. Dabei ist weder ein Nachbarsystem noch ein Mensch explizit an der Auslösung beteiligt. Die Systemreaktion ist vorab eingeplant oder terminiert. Das System „weiß“, wann es Zeit ist, diese Prozesse auszuführen. Die meisten Lehrbücher haben vergessen, die Zeit als signifikanten Akteur zu erwähnen.

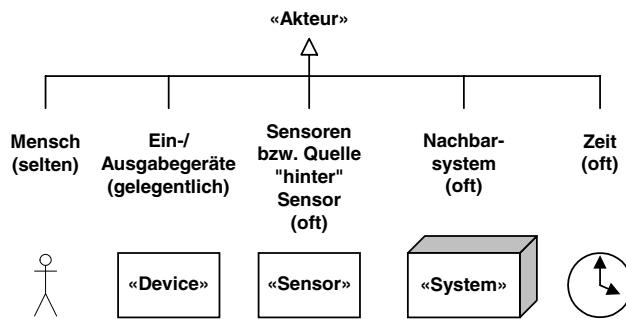


Abbildung 3.5: Eine Klassifizierung von Akteuren

Das „jacobsonsche“ Strichmännchen ist für vier der fünf aufgeführten Kategorien von Akteuren (Sensoren, Nachbarsysteme, Ein-/Ausgabegeräte und die Zeit) grafisch eher irreführend. Es fällt schwer, sich ein Nachbar-HW/SW-System als Strichmännchen vorzustellen.

Ergänzung der Standard UML-Darstellung

Die UML bietet jedoch über den Mechanismus der *Stereotypisierung* ein geeignetes Mittel, um dieses Problem methodisch in den Griff zu bekommen. Nutzen Sie nach Möglichkeit grafische UML-Stereotypen, um mit diesen Ihre Use-Case-Diagramme lesbarer zu gestalten<sup>5</sup>. Abbildung 3.6 zeigt einen Vorschlag dafür. Falls Sie keine grafischen Stereotypen einfügen möchten, oder das von Ihnen verwendete Tool das nicht zulässt, müssen Sie sich mit der verbalen Definition des Stereotyps mittels der Doppelpitzklammern begnügen. Wir versichern Ihnen, der Mensch gewöhnt sich an fast alles – auch daran, dass unter dem Strichmännchen oder dem Klassensymbol dann «Zeit», «Nachbarsystem» oder «Sensor» steht.

## UML-Stereotypen

Ein Stereotyp ist ein Erweiterungsmechanismus der UML, der die semantische Erweiterung beliebiger Modellelemente erlaubt. Mit Hilfe von Stereotypen können spezifische Eigenschaften von Modellelementen in Diagrammen festgelegt werden, ohne ihre Struktur (Eigenschaften und Beziehungen) dabei zu verändern. Zusätzlich kann ein Stereotyp weitere Bedingungen oder Eigenschaften enthalten. Die UML definiert einige vorgegebene Stereotypen, sie kann aber auch um domänen-, projekt- oder methodenspezifische Stereotypen erweitert und dem jeweiligen Bedarf angepasst werden.



**Abbildung 3.6:** Stereotyp

Stereotypen werden in Diagrammen durch Begriffe in doppelten spitzen Klammern (guillemets) oder grafische Symbole oder durch eine Kombination aus beidem notiert.

Mit Stereotypen können in den Diagrammen bestimmte Arten von Klassen, Akteuren oder anderen Modellelementen gekennzeichnet werden. Stereotypen können benutzt werden, wenn eine Menge von Elementen bei der Modellierung die gleiche Eigenschaft besitzt und diese besonders hervorgehoben werden soll. Durch die Nutzung von grafischen Symbolen für die Darstellung von Stereotypen werden Diagramme im Allgemeinen lesbarer.

<sup>5</sup> Wir sind normalerweise keine Freunde der allzu freizügigen Nutzung von neuen grafischen Stereotypen, da dadurch der Standard der UML wieder verwässert wird. Aufgrund fehlender Standardisierung werden in verschiedenen Projekten für dieselben Sachverhalte unterschiedliche Symbole genutzt, was leicht zur Verwirrung führen kann. Für fehlende Konzepte sollte man jedoch zu diesem erlaubten Mittel greifen.

Wenn Sie sich also auf die Suche nach Akteuren für Ihre Systeme machen, dann suchen Sie bitte nach allen fünf Kategorien. Prüfen Sie auch jeden Eintrag der Stakeholderliste, ob der eine oder andere Stakeholder Systemprozesse initiiert. Erfahrungsgemäß werden Sie bei vielen RTE-Systemen eine Menge von Nachbarsystemen, Sensoren oder Zeitauslösern finden. Menschen als Akteure befinden sich meist in der Minderheit.

#### 3.4.2 Wie viele Systemprozesse hat mein System?

In den Lehrbüchern zur UML finden sich immer Fallstudien mit 5 bis 15 Systemprozessen. Wie viele Systemprozesse sollten Sie aber in Ihren echten Anwendungen finden? Die einzige richtige Antwort lautet: So viele, wie es unabhängige, von Akteuren ausgelöste Prozesse gibt. Aber wie viele sind das? Weitere Fragen, die in diesem Zusammenhang immer wieder auftreten, sind:

- Soll man von übergeordneten und verfeinerten, zerlegten Systemprozessen sprechen?
- Kann man grobe und detaillierte Systemprozesse unterscheiden?
- Wie weit soll man «include», «extend» und «generalize» zur Strukturierung von Systemprozessen verwenden?

Bevor wir diese Punkte diskutieren, machen wir noch eine für viele sicherlich überraschende Feststellung:

Häufig nur ein Systemprozess!

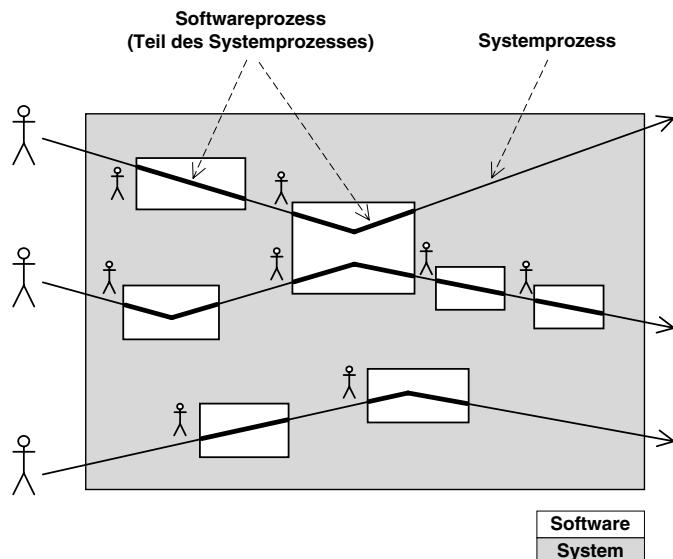
Bei bestimmten Arten von RTE-Systemen findet man nur einen einzigen echten Systemprozess! Wenn Sie ein System modellieren, das sich hauptsächlich mit Regelung beschäftigt, dann ist der Regelungsprozess häufig der einzige interessante. Seien Sie nicht enttäuscht!<sup>6</sup> Auch wenn das zugehörige Use-Case-Diagramm dann sehr einfach aussieht, sollten Sie nicht gewaltsam versuchen, daraus viele Systemprozesse zu machen. Diese Regelungssysteme haben zwar häufig noch zusätzlich einen oder wenige administrative Systemprozesse – z. B. das Voreinstellen von Parametern für die Regelung oder die Initialisierung des Systems – aber eben nur einen wichtigen primären Systemprozess.

Natürlich kann ein technisches System auch aus mehreren Systemprozessen bestehen. Häufig findet man bei Projekten von 5 bis 50 Personenjahren ca. 20 bis 30 wichtige Systemprozesse, manchmal, aber selten auch noch mehr.

<sup>6</sup> Bei vielen Regelungssystemen hören wir als Berater von unseren Kunden: Das kann doch nicht alles sein! Wir haben im UML-Kurs so viel über Use-Cases gehört. Irgendwie muss doch da noch mehr kommen. Können wir denn einen komplexen Prozess nicht mit feineren Use-Cases zerlegen? Nein, nein, und nochmals nein. Es ist so einfach. Wenn ein System nur einen Prozess hat, dann sollten Sie diesen nicht mit weiteren Use-Cases zerlegen. Es gibt viele andere UML-Diagrammtypen (wie Aktivitätsdiagramme, Zustandsmodelle, Klassendiagramme), um die Feinheiten dieses Prozesses zu verstehen und zu modellieren.

Falls Sie versucht sind, Dutzende (oder sogar Hunderte) von scheinbaren Systemprozessen hinzuschreiben, dann treten Sie einen Schritt zurück: Vergrößern Sie den Kontext Ihrer Systembetrachtung und überlegen Sie dann, welchen Effekt die Nachbarsysteme und Akteure wirklich von Ihrem System erwarten (vgl. Abb. 3.7).

Vorsicht!  
Keine funktionale  
Zerlegung



**Abbildung 3.7:** Vergrößerung des Kontexts schafft häufig Klarheit

### Beispiel: Tempomat

Wie viele Systemprozesse findet man für unser Beispiel „Tempomat im Fahrzeug“? Ein offensichtlicher Kandidat als Akteur ist der Fahrer, der z. B. über die *On*-Taste das System aktiviert. Natürlich soll der Tempomat reagieren, wenn der Fahrer zwischenzeitlich die Bremse betätigt oder mit dem Gaspedal die Regelung übersteuert. Manche Systeme bieten auch die Möglichkeit, mit einer Taste *Accel* zu beschleunigen, statt mit dem Fuß Gas zu geben. Und man muss das System manchmal kalibrieren können, um es auf unterschiedliche Reifengrößen oder den Zustand der Reifen anzupassen.

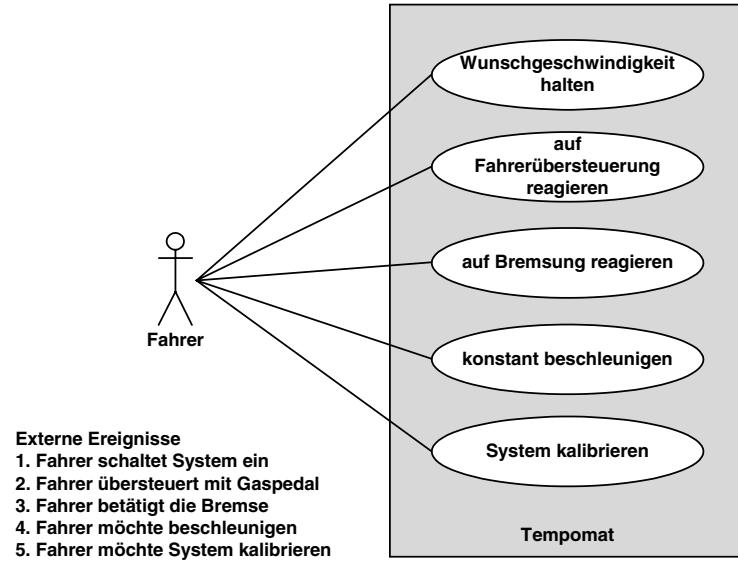
Eine erste Idee wäre es zu überlegen, welche unabhängigen Ereignisse vom Fahrer als Akteur zu internen Systemprozessen führen sollen. Oftmals kommt man dann rasch zu einem Modell wie in Abbildung 3.8.

Aber: Der Systemprozess „auf Bremsung reagieren“ kann doch nicht wirklich ganz unabhängig von außen ausgelöst werden. Dieser Prozess ist nur möglich, wenn die Regelung aktiv ist. Das System muss sich also zu diesem Zeitpunkt bei der Abarbeitung des Systemprozesses „Wunschgeschwindigkeit halten“ (oder „konstant beschleunigen“) befinden. Das lässt sich auf zweierlei Art korrigieren: Ohne das Diagramm zu verändern, könnten wir bei dem Systempro-

Zusammenspiel  
von System-  
prozessen

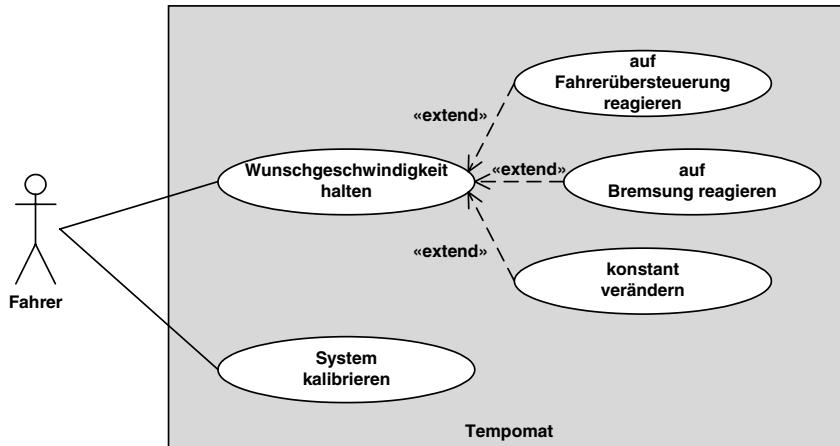
### 3 Erste Systemanforderungen

zess „auf Bremsung reagieren“ in der Systemprozess-Beschreibung als Voraussetzung hinschreiben: „Wird nur ausgeführt, wenn der Tempomat derzeit aktiv ist.“



**Abbildung 3.8:** Ein erster Versuch, Systemprozesse zu finden

In der Systemprozess-Beschreibung können beliebige Anforderungen in den Vor- oder Nachbedingungen, sowie in den essenziellen Schritten abgelegt werden. Mehr über die Systemprozess-Beschreibung finden Sie in Kapitel 5. Alternativ könnten wir das Diagramm so umzeichnen, dass diese Abhängigkeit deutlich wird. Wir modellieren einige Prozesse als „Erweiterungen“ (vgl. Abbildung 3.9).



**Abbildung 3.9:** Ein zweiter Versuch

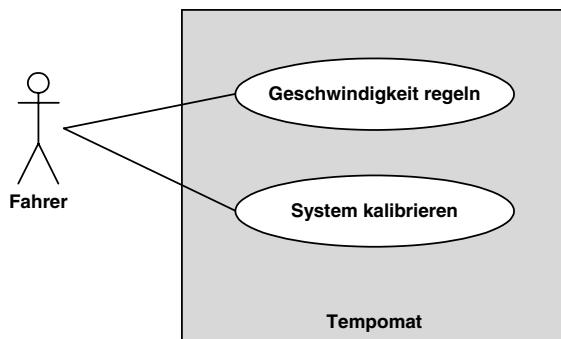
Wenn wir einmal beiseite lassen, dass «extend» in diesem Beispiel nicht ganz konform zur UML benutzt wird, dann sieht dieses Modell schon ganz gut aus. Eine Beziehung «extend» wird nach UML zur Auslagerung von Spezialfällen benutzt, in diesem Fall kennzeichnet sie getrenntes Verhalten, das den Basis-Systemprozess an einem bestimmten Punkt, dem sogenannten Extension-Point, unterbricht. In unserem Beispiel ist dieser Punkt jedoch nicht vorhersehbar, sondern asynchron (immer dann, wenn der Fahrer das Gas- oder Bremspedal betätigt).

In der Projektrealität wird «extend» oft für dieses Verhalten benutzt, was wir aber keinesfalls empfehlen!

Betrachten wir die Spezifikation des Basis-Systemprozesses. Darin werden nur die Schritte explizit beschrieben, die zur Einhaltung der Wunschgeschwindigkeit notwendig sind. In dem Ausnahmeteil dieser Spezifikation kommen die *Extension-Points* zum Einsatz, um den Basisprozess unterbrechen zu können. Bei den Erweiterungsprozessen sind die Detailabläufe für das Bremsen und Beschleunigen angesiedelt.

Aus Systemprozess-Sicht ist die Sache jedoch viel einfacher. Der Tempomat ist ein Regelungssystem. Wie wir oben festgestellt haben, gibt es dafür oft nur einen relevanten Systemprozess (und vielleicht einen oder wenige administrative Prozesse). Unsere Empfehlung für die Modellierung sehen Sie daher in Abbildung 3.10. Die beiden dort dargestellten Systemprozesse reichen zur Beschreibung des Sachverhalts völlig aus.

In der Beschreibung des Regelungsprozesses könnte man jetzt den Normalfall „Wunschgeschwindigkeit halten“ und alle Sonderfälle (Unterbrechung durch Bremsen, durch Fahrerübersteuerung etc.) leicht verständlich als umgangssprachlichen Text abhandeln.



**Abbildung 3.10:** Ein dritter Versuch

## Entscheidungshilfen

Keine der folgenden Entscheidungshilfen ist ein Allheilmittel zum Finden von Systemprozessen und zum Bestimmen, wo ein Prozess anfängt und wo er auf-

Kein  
übertriebener  
UML-Gebrauch!

### 3 Erste Systemanforderungen

hört. Zusammengenommen helfen sie aber dabei, möglichst leicht verständliche Modelle zu erzeugen.

Blickwinkel auf nächste Systemebene vergrößern

Den *ersten Trick* haben wir bereits angedeutet. Betrachten Sie das nächstgrößere System – also das System, in das Ihr System eingebettet ist. Wenn Sie für dieses größere System die Akteure suchen und deren wahre Absicht als Maßstab nehmen, dann werden häufig aus Prozessen, die vorher zersplittet waren, jetzt zusammenhängende Systemprozesse. Die vorher zersplittenen Prozesse gehören im größeren System zur Reaktion auf **ein** externes Ereignis.

Wie verhält sich das Nachbarsystem?

Ein *zweiter Trick* ist die Klassifizierung des Verhaltens Ihrer Nachbarsysteme. Suzanne und James Robertson unterscheiden drei Arten von Nachbarsystemen (vgl. auch [Rob98] und [Rob99]):

- Mit *aktiven* Nachbarsystemen sind die vier Arten von Akteuren gemeint, wie wir sie oben besprochen haben.
- *Kooperierende* Nachbarsysteme werden im Zuge der Abarbeitung eines Systemprozesses eventuell um Auskunft gebeten. Sie arbeiten aber mit unserem System zusammen und verhalten sich kooperativ, d. h. sie sind auch an der Ergebniserreichung interessiert. Deshalb betrachten wir Antworten, die von kooperierenden Nachbarsystemen kommen, nicht als neue, unabhängige Ereignisse und daher auch nicht als Auslöser für neue Systemprozesse. Der begonnene Prozess wird (nach einer kürzeren oder längeren Unterbrechung) bis zur Erreichung des ursprünglichen Ziels fortgesetzt. Falls Sie also auf ein kooperierendes Nachbarsystem stoßen, verringern Sie die Anzahl der Systemprozesse in Ihrem Use-Case-Diagramm um jene, die lediglich die Antworten der kooperierenden Nachbarsysteme auswerten. Diese werden mit den ursprünglichen Prozessen zusammengeführt. Wenn Sie dann den Ablauf des Systemprozesses mit Aktivitätsdiagrammen (siehe Kapitel 5) präzisieren, treten in diesen Diagrammen an der Unterbrechungsstelle Wartezustände auf, in denen Ihr System so lange verharrt, bis das kooperierende Nachbarsystem Ihnen eine Antwort liefert.
- *Autonome* Nachbarsysteme arbeiten völlig eigenständig, ohne auf die Ziele in Ihrem System Rücksicht zu nehmen. Wenn Sie daher in einem Ihrer Systemprozesse einen Wunsch an ein autonomes Nachbarsystem äußern, ist dieser Prozess für Sie zu Ende. Sollte das autonome Nachbarsystem so gütig sein, Ihnen eine Antwort zukommen zu lassen, dann betrachten wir diese als Auslöser für einen neuen Systemprozess.

Beachten Sie bitte, dass ein kooperatives Nachbarsystem auch ein aktives sein kann, das heißt einen Systemprozess auslöst und sich dann gleichzeitig während dessen Abarbeitung sehr kooperativ verhält. (In diesem Sinne tritt der Fahrer bei der Betätigung der *On*-Taste als aktives Nachbarsystem auf, beim Bremsen dann als kooperatives.)

Zustände modellieren

Ein *dritter Trick* ist das frühzeitige Einbeziehen von Zustandsmodellen. Häufig sind Aktivitäten im System stark durch Zustände miteinander verbunden.

Statt diese Aktivitäten als einzelne Systemprozesse zu betrachten und jeweils mit Vorbedingungen zu versehen, ist es oft hilfreich, das Zustandsmodell relativ früh im Projekt zu zeichnen. Viele der ursprünglich separat betrachteten Prozesse werden dann zu Aktivitäten an den Übergängen zwischen den Zuständen. Sie werden somit in einen besser verständlichen Zusammenhang gebracht als durch das «extend»-Konstrukt in den Use-Case-Diagrammen. Versuchen Sie sich doch an dem Zustandsmodell für das Geschwindigkeitsregelungssystem und Sie werden sehen, wie wir zu dem Diagramm in Abbildung 3.10 gekommen sind!

## 3.5 Begriffe definieren

Um ein einheitliches Verständnis unter allen Stakeholdern zu erzeugen, sollten Sie die wichtigsten Begriffe definieren, die in Verbindung mit dem zu entwickelnden Produkt/System verwendet werden. Für Namen, Bezeichnungen, Tätigkeiten und andere bei der Beschreibung eines Systems wichtige Begriffe sollte eine kurze umgangssprachliche Definition festgehalten werden. Auch hier sollten Sie keine überflüssige Zeit und Energie für Perfektionismus vergeuden. Ihr Ziel sollte ein homogener Wissensstand bezüglich der wichtigen Kernbegriffe über alle Stakeholder hinweg sein. Der gemeinsame Wissensschatz wird im Projektverlauf kontinuierlich anwachsen und sollte dann auch im Glossar festgehalten werden.

Gemeinsame  
Wissensbasis

Beim Erstellen des Glossars hat es auch Sinn, zu definierende Begriffe noch ohne ihre Definitionen schon einmal festzuhalten. Diese noch undefinierten, noch unklaren Begriffe können später wieder entfernt werden, wenn sie sich als nicht relevant herausgestellt haben. Alle relevanten Begriffe sollten im Lauf des Projekts mit Definitionen versehen werden, die bei allen Projektbeteiligten ein möglichst konkretes Verständnis über den definierten Sachverhalt erzeugen.

Undefinierte  
Begriffe im  
Glossar

Aus den festgelegten Begriffen können sich im weiteren Verlauf des Projekts Klassen, Attribute, Operationen, Zustände oder Ereignisse entwickeln, da sie wichtige Begriffe und Einheiten aus der Domäne des Systems darstellen und daher eine wohldefinierte Ausgangsbasis, z. B. für die Entwicklung eines Klassenmodells, bilden. Die Entwicklung von Klassenmodellen, und sei es zur Klärung der Begriffe, ist zu diesem Zeitpunkt allerdings noch nicht unbedingt notwendig.

20–50 zentrale  
Begriffe genügen

In dieser frühen Phase der System-/Produktentwicklung ist es völlig ausreichend, die 20 bis 50 wichtigsten Begriffe aus der Fachterminologie des Systems zu definieren. Dabei müssen Sie damit rechnen, dass bei größerer Komplexität des Systems mehr Begriffe auftreten. Insbesondere die Komplexität der Umgebung hat einen großen Einfluss auf die Zahl der zu definierenden Begriffe, da durch die Begriffsdefinitionen eine Basis für die Zusammenarbeit mit der Umgebung gelegt wird.

Zur Erhebung der Definitionen versammeln Sie am besten die wichtigsten Stakeholder und erarbeiten die Definitionen der Begriffe, die für wichtig erachtet werden, gemeinsam. Die beteiligten Personen bringen bei dieser Sitzung ihre Vorstellungen von der Welt des Systems in die gemeinsame Definition der Begriffe mit ein.

#### Begriffe aus der Sicht Ihres Systems

Für die Begriffe, die Sie definieren, sollten Sie ihre Verantwortung in Bezug auf das System und die zugehörige Domäne beachten und keine abstrakten generischen für alle erdenklichen Systeme gültigen Begriffsdefinitionen anstreben. Die Begriffe, die Sie hier definieren, sollten für Klarheit in Ihrem Projekt-sprachgebrauch sorgen. Der Aspekt einer potenziellen Wiederverwendbarkeit in späteren Projekten spielt hier noch keine Rolle.

- Definieren Sie Begriffe *immer* im Zusammenhang mit *Ihrem* System.

#### Regeln zur Begriffsdefinition

---

#### Gute Definitionen schreiben

Einige weitere Regeln können Ihnen helfen, eindeutige und „gute“ Definitionen zu schreiben:

- Formulieren Sie Definitionen im Aktiv. Passiven Sätzen fehlt häufig das Subjekt. (Der Interrupt wird erzeugt. Von wem?)
- Benutzen Sie für Definitionen einfache, direkte Vollverben und keine Hilfsverben wie „machen, haben, können“. Ein Vollverb bringt den zu beschreibenden Prozess auf den Punkt und redet nicht darum herum.
- Vermeiden Sie Nebensätze und redundante Ausdrücke. Nebensätze können Sachverhalte leichter missverständlich machen. Redundante Ausdrücke sind überflüssige Informationen, die zugunsten von leicht verständlichen kurzen Definitionen vermieden werden sollten.

#### Begriffsdefinitionen sind keine Anforderungen!

Begriffsdefinitionen sind keine Anforderungen! Bitte versuchen Sie nicht im Rahmen eines Glossars das gesamte Systemverhalten zu beschreiben. Dazu verwenden wir Modelle und natürlichsprachliche Anforderungen.

#### Umfang der Begriffe

---

Bei den zu definierenden Begriffen kann es sich sowohl um Substantive handeln, die mit dem System in Verbindung stehende Subjekte bezeichnen, als auch um Verben oder Adjektive, die Handlungen oder Eigenschaften der Subjekte beschreiben.

Ein Substantiv dient nicht nur dem Benennen eines individuellen Gegenstands, sondern bringt Ordnung in die Menge der individuellen Gegenstände. Gerade auf diese Substantive werden wir bei der Modellierung der zentralen, fachlichen Klassen zurückgreifen.

Sie können Ihre Definitionen ohne jegliche Strukturvorgaben verfassen. Sofern Sie allerdings Vorgaben dafür verwenden möchten, wie Sie ein Substantiv definieren, so empfehlen wir Ihnen die folgenden. Die Definition eines Substantivs sollte die Aspekte des Seins, der Merkmale und des Verhaltens abdecken.

Sie können dazu bei der Definition von Substantiven ein Muster benutzen, das drei eigenständige Sätze enthält:

- **Sein:** Ein *Substantiv* ist ein ... (*Klassifikation*).
- **Merkmal:** Ein *Substantiv* zeichnet sich aus durch ... (*Eigenschaft oder Fähigkeit des Subjekts*).
- **Verhalten:** Ein *Substantiv* ... (*Aktion des Subjekts*).

Zum Beispiel wird ein Autofahrer durch drei getrennte Sätze definiert:

- Ein *Autofahrer* ist eine *natürliche* Person.
- Ein *Autofahrer* hat *Zugriff* auf die Bedienelemente des Tempomaten.
- Ein *Autofahrer* *bedient* den Tempomaten.

Bei der Definition von Begriffen müssen auch die Verben und Adjektive berücksichtigt werden, die in Zusammenhang mit den Subjekten stehen können. Ein Adjektiv wird immer zusammen mit einem zugehörigen Substantiv definiert, dessen Eigenschaft es beschreibt.

- Ein *stillstehendes* Rad ist ein Rad, dessen Drehgeschwindigkeit 0,15 Grad/s unterschreitet.

Ein Verb bezeichnet einen Prozess, den ein Subjekt durchläuft, oder eine Aktion, die ein Subjekt ausführen kann. Die Definition von Verben wird häufig vergessen, da die Stakeholder davon ausgehen, dass den anderen Stakeholdern die Bedeutung von Verben im Zusammenhang mit Subjekten klar ist. Unseren Erfahrungen nach lohnt es sich, vor allem die Verben zu definieren.

Zum Beispiel sollte für einen Tempomaten das Verb „regeln“ definiert werden:

- *Regeln* ist der Prozess, kontinuierlich den Ist-Zustand zu überprüfen und bei Abweichungen vom Soll Gegenmaßnahmen einzuleiten.

Durch die Nutzung von natürlicher Sprache können bei der Definition von Begriffen Ungenauigkeiten auftreten, welche die Verbindlichkeit der Definitionen negativ beeinflussen. Vor allem sind hier die sprachlichen Defekte der Tilgung, Generalisierung und Verzerrung zu nennen, die bei Definitionen zu Problemen führen können. Ausführliche Betrachtungen zu diesen Defekten finden Sie in [Rupp02].

### **Checkliste für Begriffsdefinitionen**

---

Um gute Qualität der Definitionen sicherzustellen, sollten Sie für Ihre Definitionen einige Fragen überprüfen.

- Haben Sie alle wichtigen fachlichen Substantive, Verben und Adjektive definiert?
- Sind alle in den Definitionen benutzten Begriffe (Substantive, Verben, Adjektive) bekannt oder definiert?
- Haben Sie in Begriffsdefinitionen keine Anforderungen beschrieben?
- Sind die Definitionen so konkret verfasst, dass der Leser einen klaren Eindruck des Sachverhalts erhält?

## **3.6 Haben Sie eine klare Produktvision vor Augen?**

---

In diesem Kapitel haben wir über die Systemvorbereitung aus Sicht der Anforderungsaktivitäten diskutiert. Wenn Sie diese Schritte durchgeführt haben, sollten Sie eine klare Zielvorstellung über das System oder Produkt haben, das Sie entwickeln wollen.

Betrachten Sie Ihr bisher erreichtes Ergebnis kritisch:

- Ist Ihnen der Umfang (Scope) Ihres Systems klar?
- Wissen Sie, wer wie viel und was zu der Entwicklung beitragen kann?
- Haben Sie mit den wichtigsten beteiligten Personen Einigung über die Ziele der Systementwicklung herbeigeführt?
- Haben Sie sich mit den Stakeholdern auf gemeinsame Begriffe für das RTE-System geeinigt?
- Ist es Ihnen gelungen, Ihr Gesamtsystem in relevante Prozesse aus Sicht der Akteure in der Systemumgebung zu zerlegen?

„Wer seine Schranken kennt, der ist der Freie;  
wer sich frei wähnt, ist seines Wahnes Knecht.“

*Franz Grillparzer (1791–1872),  
Wiener Hofkonzipist und Burgtheaterdichter*

# 4

## **Systemrandbedingungen: die Fesseln für Designer**

---

### **Fragen, die dieses Kapitel beantwortet:**

- Was haben andere für mich entschieden?
- Welche Freiheitsgrade wurden mir entzogen?
- Welche Fesseln wurden mir für die Entwicklung angelegt?
- Wie modellierte ich die physikalische Umgebung meines Systems?
- Wie modellierte ich die Vergangenheit (d. h. die Altlasten, existierende Teilsysteme) und alles, was bereits entschieden ist?
- Warum sollte man über Hardware und Software getrennt nachdenken?

Fast niemand beginnt eine Produktentwicklung auf der grünen Wiese. Ziel dieses Kapitels ist es, Ihnen pragmatische Tipps zu geben, wie Sie schon zu diesem frühen Zeitpunkt im Projekt einerseits Randbedingungen und andererseits Designentscheidungen, die bereits getroffen wurden, modellieren oder wenigstens erfassen. Abbildung II.1 hat diese wesentlichen Aktivitäten zur Vorbereitung der Systemarchitektur schon gezeigt.

## 4.1 Physikalischen Kontext abgrenzen

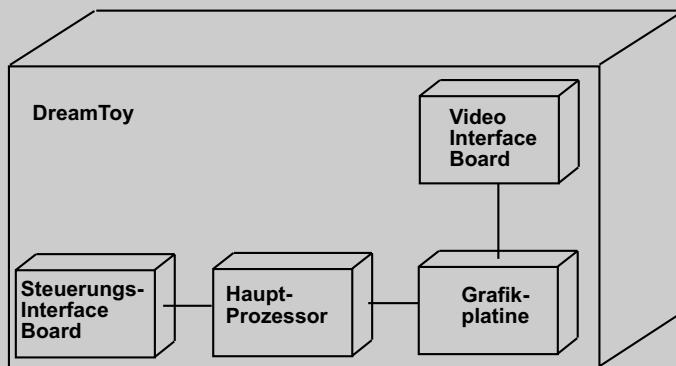
In Kapitel 3 haben wir alle Arten von Nachbarsystemen und die logischen Ein- und Ausgaben zu unserem System betrachtet. Jetzt legen wir die Verbindungskanäle zwischen allen Arten von Nachbarsystemen und unserem System fest – die Medien, über die logische Ein- und Ausgaben mit der Systemumgebung ausgetauscht werden. Zur Notation nutzen wir das Verteilungsdiagramm der UML.

### Verteilungsdiagramme (Deployment Diagrams)

Das UML-Ausdrucksmitte für geografische Verteilung oder für Verteilung auf verschiedenen Prozessoren ist das Verteilungsdiagramm. Bei Grady Booch hießen diese Diagramme früher „Prozessordiagramme“ und enthielten Rechner und Geräte, sowie deren Zusammenhänge. Dieser Name war für viele einleuchtender als der neue Name „Deployment Diagram“.

Verteilungsdiagramme enthalten im Wesentlichen zwei Elemente:

- Knoten zur Darstellung von geografischen Orten, Rechnern oder Teilen davon (dargestellt als Quader) und
- physikalische Verbindungen zwischen den Knoten, z. B. Bussysteme, Kabel, Netzwerke (dargestellt als Linien)



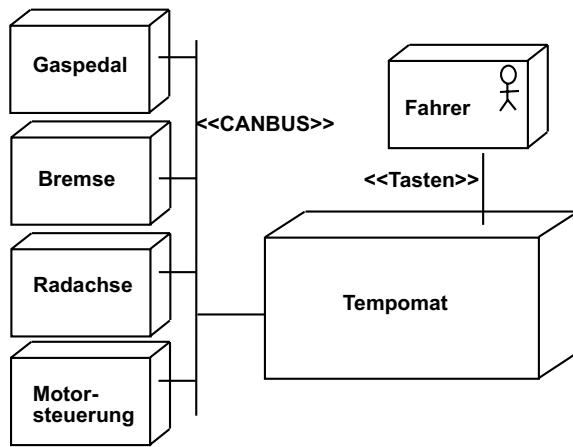
**Abbildung 4.1:** Beispiel eines Verteilungsdiagramms

Mehr Details  
zur Notation:  
[Boo99]

Den Knoten werden dann (grafisch oder in der Beschreibung des Knotens) Softwarekomponenten zugeordnet, die auf diesem Knoten ablaufen.

Bei der physikalischen Kontextabgrenzung modellieren wir das Gesamtsystem als einen zentralen Knoten und ordnen die Nachbarsysteme als weitere Knoten rundherum an. Dann zeichnen wir die Kanäle ein, die unser System zu den Nachbarsystemen unterhält.

Nehmen wir an, unser Auftraggeber für den Tempomat hat von uns gefordert, die Schnittstelle zum Fahrer über Tasten zu lösen, während alle anderen Sensoren und Nachbarsysteme am Canbus hängen sollen. Abbildung 4.2 zeigt das entsprechende Verteilungsdiagramm<sup>1</sup>.



**Abbildung 4.2:** Der physikalische Kontext als Verteilungsdiagramm

Sollten Sie bei der einen oder anderen Verbindung noch nicht wissen oder bewusst noch nicht festlegen wollen, über welchen Kanal die Kommunikation erfolgt, dann zeigen Sie mit einer Verbindungsleitung nur die Tatsache an, dass ein Kanal benötigt wird und lassen die Beschriftung weg. Noch deutlicher wird es, wenn Sie fehlendes Wissen bewusst mit einem Fragezeichen markieren, solange Sie die Antworten noch nicht haben.

## 4.2 Festlegungen zur Hardware dokumentieren

Wenn man schon sehr früh Vorgaben über die Hardware bekommt oder die geografische Verteilung kennt, sollte man dies sofort in Form von Verteilungsdiagrammen skizzieren. Bilder sind meist leichter verständlich als Text und drängen uns eher dazu, sie zu vervollständigen. Sie zeigen deutlich die Notwendigkeit, die Fragen an die Stakeholder zu stellen, die uns noch nicht beantwortet wurden. Sie sollten also beginnen, die Hardwarearchitektur in Form eines Verteilungsdiagramms zu zeichnen.

---

<sup>1</sup> Wir haben uns mit dem Einzeichnen des Canbus einen kleinen Freiheitsgrad bei der Nutzung der UML eingeräumt. Im Web finden Sie die notationell „korrekte“ Fassung, die allerdings aufwändiger und wesentlich weniger einleuchtend ist.

Mut zur Lücke!

Gehen Sie bei der Erfassung der Hardwarerandbedingungen jedoch noch nicht zu weit und halten Sie nur das fest, was der Auftraggeber schon verlangt hat. Scheuen Sie sich nicht, isoliert stehende Knoten zu zeichnen, wenn Sie über die Verbindung noch keine Aussagen haben. Das Bild dient nur dazu, das bisherige Wissen festzuhalten.

Für unser Beispiel gehen wir davon aus, dass bisher nur eine Vorgabe gemacht wurde: Die Display-Ansteuerung des Tempomaten soll auf einem eigenen Prozessor implementiert werden. Abbildung 4.3 zeigt das rudimentäre Verteilungsdiagramm, wobei der Rest des Systems durch einen als Wolke stilisierten Knoten dargestellt wurde. Die Annahme ist, dass der Display-Prozessor nur für die Ausgaben an den Fahrer verantwortlich ist und einzlig und allein mit dem oder den anderen Prozessor(en) des Tempomaten kommuniziert, nicht aber mit anderen Teilen der Systemumgebung.

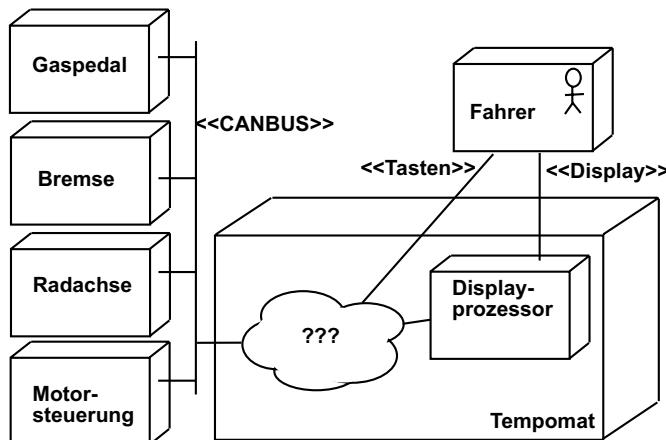


Abbildung 4.3: Festhalten von Hardwarevorgaben

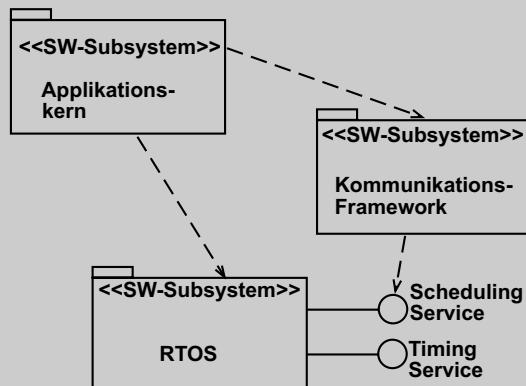
Beim Modellieren dieser Vorgabe stellen wir vielleicht auch fest, dass wir bei der Betrachtung des physikalischen Kontexts die Display-Schnittstelle zum Fahrer übersehen haben und korrigieren dies entsprechend. So wird die Hardwarearchitektur in dem Maße präzisiert, wie wir dazulernen.

### 4.3 Subsysteme skizzieren

Ähnlich wie bei Hardwarevorgaben verhalten wir uns auch bei Softwarevorgaben. Wenn Sie erfahren, dass Teile der Software gekauft werden sollen oder wieder zu verwenden sind, dann beginnen Sie damit, dieses Wissen in eine erste rudimentäre Version der Softwarearchitektur einfließen zu lassen. Die einfachste Möglichkeit, welche die UML bietet, ist, sie zunächst mit Paketen darzustellen.

## Pakete (Packages)

Die UML stellt uns ein allgemeines Ausdrucksmittel zur Verfügung, um (fast) beliebige Dinge zu größeren Einheiten zusammenzufassen und diese dann unter einem Namen ansprechen zu können. Damit lassen sich sehr gut Überblicksbilder gestalten, deren Feinheiten vielleicht erst später ausdiskutiert werden. Genauso lassen sich große „Black Boxes“ darstellen, deren Innenleben uns nicht interessiert oder uns unbekannt ist. Für Pakete wird das Ordnersymbol verwendet (siehe Abbildung 4.4). Die Abhängigkeit zwischen Paketen wird mit gestrichelten Pfeilen dargestellt. Lesen Sie diesen Pfeil zunächst als „Paket A hängt von Paket B ab“ oder „wenn Paket B nicht existiert, dann hat Paket A Schwierigkeiten“.



**Abbildung 4.4:** Pakete zur Darstellung von Softwaresubsystemen

Für Pakete mit starkem logischen Zusammenhang, die man von außen als Black Box betrachten kann, hat die UML den Stereotyp <<subsystem>> vordefiniert, den wir im Folgenden für festliegende Softwaresubsysteme auch verwenden werden. Mit der Lollipop-Notation kann man benannte Schnittstellen in diese Diagramme einfügen. Abbildung 4.4 zeigt, dass das Kommunikations-Framework explizit die Scheduling-Service-Schnittstelle des Betriebssystems RTOS nutzt. Für den Applikationskern ist vorläufig nur die generelle Nutzung des RTOS eingezeichnet.

Präzisere Notationsdiskussion:  
[Rum99]

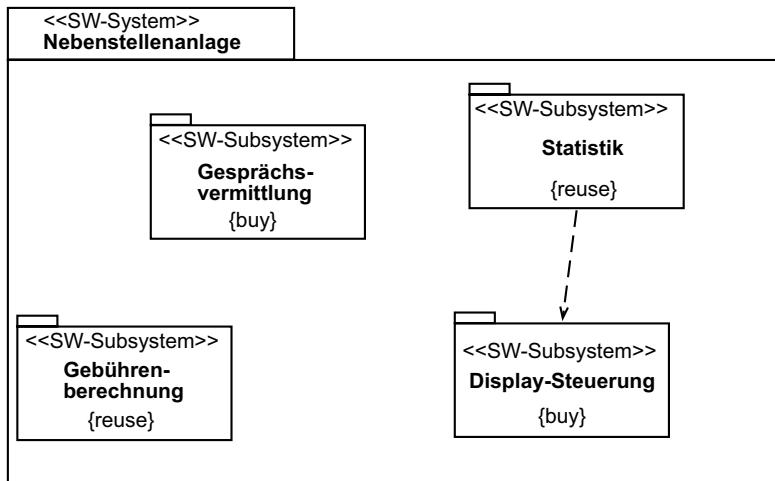
Wenn Sie also in den ersten Gesprächen davon erfahren, dass bestimmte Subsysteme eingesetzt werden sollen, dann zeichnen Sie einfach für jedes Subsystem ein Paket. Lassen Sie sich noch keine grauen Haare wachsen, wenn das Bild unvollständig ist oder wenn Sie das Gefühl haben, die Subsysteme und ihre Abhängigkeiten noch nicht genau verstanden zu haben. Wenn wir mehr über die funktionalen Anforderungen wissen, werden wir gezielte Designentscheidungen zur Softwarearchitektur treffen und dieses Modell vervollständigen. Wir werden auch hinter jedes Grafiksymbol im Laufe der Zeit genauere Beschreibungen legen, die Schnittstellen präzisieren und prüfen, wie sinnvoll die gezeichneten Abhängigkeiten sind. Momentan – bei der Systemvorberei-

Mut zur Lücke!

tung – geht es uns nur um das Festhalten bereits getroffener Entscheidungen. Auch wenn Sie Gründe hören, warum dieses Betriebssystem verwendet werden soll oder warum das Kommunikations-Framework aus dem letzten Projekt wieder verwendet werden soll, dann hinterlegen Sie diese sofort bei der Zeichnung. (Das vollständige Beschreibungsmuster für Subsysteme stellen wir Ihnen in Kapitel 6 vor.)

Mehr zu Tagged Values: [Rum99]

Abbildung 4.5 zeigt ein Beispiel für ein sehr frühes Stadium einer Softwarearchitektur. Wir gehen davon aus, dass der Kunde von uns gefordert hat, zwei Pakete aus dem letzten Projekt weiterzuentwickeln und zwei Teilsysteme zu kaufen. Oftmals ist es hilfreich, in den Modellen spezielle Eigenschaften der Pakete sichtbar zu machen. Dies können Sie z. B. durch farbliche Unterscheidung der Pakete erreichen oder aber – wie in der folgenden Abbildung – durch die Tagged Values der UML, die wir hier zu den Werten {buy} und {reuse} verkürzt haben.



**Abbildung 4.5:** Eine erste Skizze einer Softwarearchitektur

## 4.4 Randbedingungen sammeln

Wenn Sie mit Ihren Stakeholdern über die Ziele und die funktionalen Anforderungen sprechen, werden Sie immer wieder mit nicht-funktionalen Anforderungen und Randbedingungen konfrontiert werden. Diese fallen in eine von vier Kategorien:

- Qualitätsanforderungen für das System,
- Randbedingungen für den Entwurf,
- Anforderungen an den Prozess,
- Management-Randbedingungen.

Betrachten wir einige Beispiele:

- Eine Forderung nach einem bestimmten Betriebssystem: „Das System muss auf der Basis von Windows ME entwickelt werden.“
- Eine Forderung nach der Verfügbarkeit: „Die Radar erfassung darf nicht länger als zwei Minuten ausfallen.“
- Eine physikalische Anforderung: „Das Gerät darf nicht mehr als 350 Gramm wiegen und muss in ein Gehäuse von 2 x 8 x 5 cm passen.“
- Eine Vorschrift über das Vorgehensmodell: „Für die Entwicklung ist das Vorgehensmodell „Ariadne“<sup>2</sup> strikt einzuhalten.“
- Eine Forderung nach zugekaufter Standard-Software: „Die Software muss die Treiber der Firma Aviplus verwenden, die wir zukaufen werden.“
- Eine Ressourcenbeschränkung: „Die Entwicklung der GSM-Schnittstelle muss bis 28.2. abgeschlossen sein.“
- Eine Migrationsforderung: „Die Daten des Vorläuferprodukts müssen in das neue System überführbar sein.“
- Eine Schnittstellenanforderung: „Unser Auswertesystem muss mit dem System CrissCross in der Zentrale kompatibel sein.“

Allen diesen Randbedingungen gemeinsam ist, dass sie dem Designer Freiheitsgrade entziehen. Der Architekt kann frei über die Technologien zur Realisierung entscheiden, es sei denn, jemand hat bestimmte Technologien gefordert. Der Projektleiter kann das Vorgehen bei der Entwicklung festlegen, es sei denn, jemand hat das Vorgehensmodell für die ganze Firma festgelegt. Ihre Herausforderung ist es, die funktionalen Anforderungen zu erfüllen unter Einhaltung all der Randbedingungen, die Ihnen vorgegeben wurden.

Bei der Systemvorbereitung kommt es noch nicht darauf an, alle Kategorien von Randbedingungen vollständig zu erfragen und zu dokumentieren. Sie sollen nur ein offenes Ohr für solche Punkte haben. Wenn Ihnen derartige Randbedingungen über den Weg laufen, dann sollten Sie diese sofort festhalten. Erst in der nächsten Phase bei der Systemstrukturierung suchen wir gezielt nach allen. In Kapitel 6 stellen wir Ihnen ein Gliederungsschema vor, mit dem Sie Ihren Stakeholdern systematisch alle Arten von Randbedingungen entlocken können.

Offensichtliches  
Festhalten

## **Aufschreiben oder modellieren?**

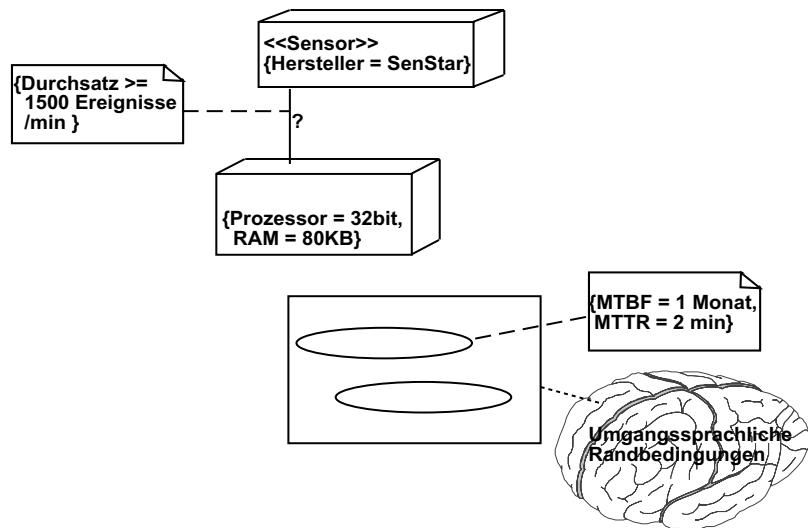
Wie sollen wir die Randbedingungen festhalten, die uns sehr früh „über den Weg laufen“? Es gibt einige Vorgehensmodelle, die eine schriftliche, umgangssprachliche Erfassung erzwingen. Wir wollen Ihnen einen möglichst einfachen Weg aufzeigen, der die Nachvollziehbarkeit erleichtert und Doppel-

---

<sup>2</sup> Das mit dem Sicherheitsfaden, damit man jederzeit zurückfindet. ☺

arbeit vermeidet. Einige Randbedingungen können wir direkt Modellen zuordnen, einige andere müssen wir vorläufig einfach umgangssprachlich erfassen.

Betrachten wir auch hier einige Beispiele, die teilweise in Abbildung 4.6 eingezeichnet wurden. Alle Forderungen nach Einbettung des Systems sowie logische und physikalische Schnittstellenforderungen sollten Sie sofort in Kontextdiagramme umsetzen. Wenn z. B. in der Kommunikation mit dem Nachbarsystem ein Durchsatz von 1500 Signalen pro Minute gefordert wird, dann können Sie im Verteilungsdiagramm einen Kanal zu diesem Nachbarsystem einzeichnen und mit dieser Forderung hinterlegen, auch wenn Sie über die Technologie noch nicht entschieden haben. Im gleichen Diagramm können Sie z. B. geforderte Prozessortypen oder Speichergrößen durch Tagged Values anmerken.



**Abbildung 4.6:** Zuordnung von Randbedingungen zu Modellen

Auch das Use-Case-Diagramm ist ein guter Aufhänger für einige nicht-funktionale Anforderungen. Fordert man von einem Prozess eine bestimmte Verfügbarkeit, so können Sie diese dem Prozess direkt als Anmerkung anhängen (oder natürlich in der Beschreibung des Systemprozesses hinterlegen – wie in Kapitel 3 erläutert). In unserem Beispiel wurde die Verfügbarkeitsforderung gleich als „mean time between failure“ und „mean time to repair“ geschrieben.

Alle Qualitätsanforderungen und Randbedingungen, für die Sie keinen offensichtlichen Aufhängepunkt in den Modellen finden, sollten Sie als Sammlung dem Gesamtsystem zuordnen. In der obigen Abbildung ist dies durch das Requirementsgehirn angedeutet, das an den Rahmen im Use-Case-Diagramm angehängt wurde. Die logische Gliederung für diesen Teil des Gehirns finden Sie in Kapitel 6.

## 4.5 Softwaresubsysteme auf Knoten abbilden

Betrachten wir einmal kurz, was wir schon über unser RTE-System wissen. Hoffentlich kennen wir die Ziele und die wichtigsten Prozesse, die unser System ausführen soll. Vielleicht haben wir auch eine Vorstellung von der Hardware oder der geografischen Verteilung unseres Systems. Eine Tabelle hat sich in vielen Projekten als sehr hilfreich für die weitere Vorgehensweise erwiesen. Versuchen wir, die Softwaresubsysteme auf die schon gefundenen Prozessoren (d. h. die Knoten im Verteilungsdiagramm) tabellarisch abzubilden, wie dies in Abbildung 4.7 beispielhaft dargestellt ist.

	Video-Interface Board	Steuerungs-Interface Board	Hauptprozessor	Grafikplatine
Bewegungs-interpolation				X
Spielelogik			X	X
...		X	X	X
...		X		

Abbildung 4.7: Abbildung Softwaresubsysteme auf Hardwareknoten

Diese Matrix erfüllt mehrere Zwecke:

- Für das Management ist sie ein Überblick über die Hardware-/Softwarekomplexität des Systems und kann als Basis für die Cluster-Analyse und Iterationsplanung herangezogen werden.
- Für den weiteren Analyse- und Designprozess gibt sie einerseits wertvolle Hilfe, worauf man bei der Zerlegung der Systemprozesse in Aktivitäten aufpassen soll. Andererseits hilft sie, die Zweckangaben für die Knoten leichter zu fassen, wenn man weiß, welche Aufgaben sie insgesamt zu erfüllen haben.
- Der Qualitätssicherung hilft die Matrix, weil Sie auf einen Blick die Namen der Knoten und die Namen der Softwaresubsysteme vergleichen können. In hardwareorientierten Firmen passiert es oftmals, dass die Softwarepakete strikt nach der schon bekannten Hardwarestruktur geschnitten wer-

den. Wenn daher alle Softwarepakete Namen wie „Hauptprozessor-Software, Grafikplatinen-Software, Videoboard-Software, ...“ tragen, dann erkennen Sie rasch, dass aus logischer Sicht der Software noch keine guten Überlegungen angestellt wurden. Bei der Systemvorbereitung können Sie diesem Trend noch entgegentreten und die Softwaresubsysteme eher aufgabenbezogen schneiden.

### 4.6 Kennen Sie Ihre Schranken und Freiheitsgrade?

In diesem Kapitel haben wir über die Systemvorbereitung aus Sicht der Architektur diskutiert. Sie sollten die offensichtlichen Fesseln für den Entwurf modelliert oder umgangssprachlich erfasst haben. Betrachten Sie Ihr bisher erreichtes Ergebnis kritisch:

- Haben Sie sich intensiv mit der physikalischen Einbettung auseinander gesetzt und Vorgaben und Forderungen in ein physikalisches Kontextdiagramm umgesetzt?
- Gibt es aus Hardware-, Organisations- oder Softwaresicht offensichtliche Subsysteme oder Komponenten? Wenn ja, haben Sie dieses Wissen auf dem verfügbaren Stand in Modelle gegossen?
- Haben Sie faktisch festliegende Entscheidungen bezüglich geografischer Verteilung oder Hardwareaufteilungen als Überblicksdiagramme transparent gemacht?
- Haben Sie eine grobe Abbildung der Systemprozesse auf diese Standorte oder Hardwareeinheiten vorgenommen?
- Sind andere, bereits bekannte Randbedingungen schriftlich festgehalten und – wo auch immer möglich – den besten Stellen in den bisherigen Modellskizzen zugeordnet?

„Es ist gar nicht so einfach, selbst den einfachsten Gedanken mit Präzision und in knapper Form zu übermitteln.“

*Cyril Northcote Parkinson (1909–1993),  
brit. Historiker und Publizist*

# 5

## **Produkt-/Systemanforderungen präzisieren**

---

### **Fragen, die dieses Kapitel beantwortet:**

- Wie lernt man mehr über die Systemprozesse?
- Wie detailliert sollte man die Systemprozesse beschreiben?
- Was tun, wenn mir keiner die gewünschten Abläufe mitteilen kann?
- Wie geht man bei stark zustandsabhängigen Prozessen vor?
- Wie ordnet man die Begriffe, vor allem, wenn es viele werden?

Gutes Requirements Engineering in agilen Projekten heißt nicht, alle Anforderungen bis ins letzte Detail festzunageln, bevor mit dem Design begonnen wird. Sie sollten nur so viel Wissen erheben und dokumentieren, dass Sie im nächsten Schritt Subsysteme bilden können, die dann eingekauft, beauftragt oder selbst weiter spezifiziert und entwickelt werden. In diesem Kapitel bieten wir Ihnen viele praktische Hilfestellungen, wie Sie funktionale Anforderungen erfassen und modellieren und nicht-funktionale Anforderungen erfassen und den Modellelementen zuordnen können. Agilität bedeutet jeweils die Maßnahmen zu ergreifen, die den meisten Erfolg versprechen und die Risiken der Systementwicklung am effektivsten einschränken. Wie weit Sie die einzelnen Maßnahmen vollständig durchlaufen, entscheiden Sie anhand der realen Projektsituation selbst.

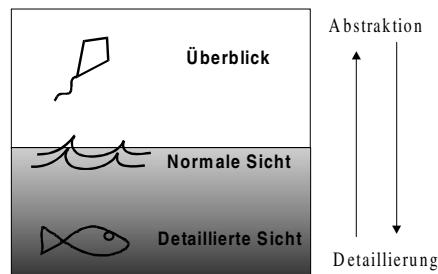
Gerade beim Erheben und Dokumentieren von Wissen werden Sie in Ihrer Arbeitsweise aber auch von den Stakeholdern „getrieben“. D. h. Sie sammeln alle Informationen, die Ihnen geliefert werden und relevant erscheinen. Sie modellieren parallel Informationen unterschiedlicher Art und Detaillierung mit

den jeweils geeigneten UML-Diagrammen. Agiles Vorgehen durchbricht den starren Ablauf, z. B. erst das Use-Case-Diagramm, dann die Use-Case-Beschreibung, dann das Sequenzdiagramm fertig zu stellen. In einem Gespräch mit Stakeholdern ergibt sich meist ein Wissenszuwachs, der sowohl Informationen zur statischen Struktur, zu den zentralen Begriffen, zu den Abläufen, zum Zeitverhalten und zu einzelnen Prozessschritten enthält. Agiles Modellieren von Anforderungen bedeutet, die relevanten Informationen in geeigneter Notation ablegen, ständig alle Informationen in dem Requirementsgehirn anreichern und ändern zu können. Auch wenn wir im Buch gezwungen sind, die einzelnen Aktivitäten nacheinander darzustellen, werden sie in Ihrem Projekt vermutlich parallel ablaufen. Abhängig von den Eigenschaften Ihres Systems können einzelne Maßnahmen keine Minimierung Ihres Projektrisikos bewirken; dann sollten Sie diese Schritte nicht durchführen (siehe Abbildung II.2).

## 5.1 Systemprozesse spezifizieren

Spezifizieren Sie nun die Systemprozesse, die Sie für Ihr RTE-System gefunden haben (siehe Kapitel 3). Die einfachste Art dies zu tun, ist eine umgangssprachliche Beschreibung.

Gerade umgangssprachliche Beschreibungen können Sie je nach Zielpublikum auf sehr unterschiedlichen Detaillierungsstufen verfassen. Um das jeweilige Niveau kenntlich zu machen, schlagen wir hier in Anlehnung an [Coc01a] die Verwendung der in Abbildung 5.1 dargestellten grafischen Symbole vor.



**Abbildung 5.1:**  
Abstraktionsstufen einer Beschreibung

Die beiden Abstraktionsstufen *Überblick* (Drache ) sowie *normale Sicht* (Welle ) sind sinnvolle Beschreibungsvarianten. Managern reicht meist bereits die Drachenebene, um das zu erfahren, was sie wissen müssen. Auch für Entwickler ist die Drachenebene ein sinnvoller Zwischenschritt, bevor man sich mit weiteren Details beschäftigt. Die Drachenebene sollte als Zusammenfassung eines Systemprozesses auch langfristig gepflegt werden. Sie enthält zwar nicht alles, aber doch einen vernünftigen Überblick über den Prozess. Die Welle stellt das richtige Maß an Detaillierung für die Systementwickler dar. Auch in dieser Beschreibung wird man nicht jedes Detail über den Systemprozess erfahren. Vor allem werden wir technologische Aspekte auch aus dieser Beschreibung möglichst heraushalten.

Travel light

Die beiden Abstraktionsebenen sollten allerdings keine zwei separaten Dokumente sein, die Sie parallel pflegen und konsistent halten – dies wäre ein Ver-

60

stoß gegen das Prinzip „travel light“<sup>1</sup>. Die Ebenen können als zwei unterschiedlich detaillierte Sichten auf ein Dokument realisiert werden. Oder es sind Stationen eines Dokumentes auf dem Weg vom Überblick zum normalen Abstraktionsniveau.

Die Unterwassersicht (Fisch  ) ist pragmatisch, oft technologiebehaftet und zu detailliert. Sie würde Entscheidungen vorwegnehmen und Details notieren, die in einer Use-Case-Beschreibung unangebracht sind. Wir betrachten die Unterwassersicht für Systemprozess-Beschreibungen als nicht mehr sinnvoll.

Um Ihren Systemprozess näher zu spezifizieren, empfehlen wir Ihnen die Erstellung von Use-Case-Beschreibungen für jeden Systemprozess. Wir sprechen daher im weiteren Verlauf von Systemprozess-Beschreibungen. Kern ist eine natürlichsprachliche Darstellung der Prozessschritte, die das System durchläuft. Für eine sehr grobe Systembeschreibung (Abstraktionsgrad Drache) genügen:

**Tabelle 5.1:** Formular für Systemprozess-Beschreibungen auf Drachenniveau

Name	ein aussagekräftiger Name des Systemprozesses (häufig in der Form Substantiv + Verb: z. B. Geschwindigkeit regeln), der auch im Diagramm verwendet wurde	Systemprozess-Beschreibung
Akteur	der initierende Akteur (wie in Abschnitt 3.3 beschrieben)	
auslösendes Ereignis	der Trigger für den Prozess	
Kurzbeschreibung	ein kurzer Text, der nicht mehr als zwei bis vier Sätze umfassen sollte	



Diese Überblicksbeschreibung eines Systemprozesses werden Sie immer weiter mit Wissen anreichern. Hier empfehlen wir ein Beschreibungsmuster zur weiteren Vervollständigung in Ihrem Requirementsgehirn.

Use-Case-Beschreibungen sind zudem ein guter Aufhänger für nicht-funktionale Anforderungen. Insbesondere Erwartungen an die Verfügbarkeit und das Zeitverhalten des Systemprozesses werden von den Stakeholdern häufig sehr früh formuliert. Nehmen Sie diese Punkte mit in die Systemprozess-Beschreibung auf, wenn sie angesprochen werden.

Nichtfunktionale Anforderungen

## Use-Case-Beschreibung

Eine Use-Case-Beschreibung ist eine umgangssprachliche Beschreibung eines Prozesses. Bei diesem Beschreibungstext kann es sich um wenige Notizen oder um eine ausführliche, strukturierte Darstellung handeln. Neben den einzelnen essenziellen Systemreaktionen, die den Standardablauf des Use-Cases beschreiben, können weitere Informationen, die zu diesem Zeitpunkt bereits von den Stakeholdern formuliert werden, aufgenommen werden.

<sup>1</sup> travel light ist ein wichtiger Grundsatz beim agilen Vorgehen: Versuchen Sie, nur so viel an Dokumentation zu verfassen, wie nötig ist, aber so wenig wie möglich. Nehmen Sie keinen unnötigen Papierballast auf Ihrer „Reise“ zum fertigen System in Kauf!

## 5 Anforderungen präzisieren

Das folgende Beschreibungsmuster hilft Ihnen, die Informationen im Requirementsgehirn strukturiert zu dokumentieren.

**Tabelle 5.2:** Beschreibungsmuster für eine Use-Case-Beschreibung

Name	Geschwindigkeit regeln	
Akteur	Fahrer	
Auslösendes Ereignis	Fahrer wählt Zielgeschwindigkeit aus	
Kurzbeschreibung	Der Tempomat regelt die durch den Fahrer eingestellte Geschwindigkeit durch Signale an die Motorsteuerung. Er kann dabei die Geschwindigkeit bis zu einer Zielgeschwindigkeit reduzieren oder erhöhen oder eine Wunschgeschwindigkeit halten.	
Vorbedingungen	Geschwindigkeit > 50 km/h, Tempomat eingeschaltet	
Essentielle Schritte	Intention der Systemumgebung	Reaktion des Systems
	Fahrer startet Motor	Tempomat ist bereit
	Fahrer will Geschwindigkeit halten	Tempomat hält die Geschwindigkeit
	Fahrer will beschleunigen	Tempomat beschleunigt Fahrzeug
	Fahrer will Tempo reduzieren	Tempomat verringert Geschwindigkeit
	Timer löst Anzeige von Ziel- und Istgeschwindigkeit aus	Tempomat meldet Ziel- und Istgeschwindigkeit
	Fahrer übersteuert Tempomaten (durch Kickdown, Vollbremsung, Geschwindigkeit < 50km/h)	Regelung beenden
	Fahrer stellt Motor ab	Tempomat ausschalten
Ausnahmefälle	Defektbedingte Einschränkung der Funktionalität des Tempomaten.	
Nachbedingung	Tempomat ist ausgeschaltet.	
Zeitverhalten	Maximal 32 ms von der Eingabe der Zielgeschwindigkeit bis zur Aktualisierung der Anzeige inklusive der Ermittlung der benötigten Maßnahmen und Ansteuerung der betroffenen Nachbarsysteme.	
Verfügbarkeit	Maximal ein Systemausfall innerhalb 1000 Betriebsstunden.	
Fragen, Kommentare	Ist es möglich, die Werkseinstellungen des Tempomaten auf eigene Wünsche anzupassen (langsameres Beschleunigen im Bereich über 120 km/h, Maximalgeschwindigkeit festlegen)?	

In den Feldern *Name* und *Akteur* können neben den bereits im Use-Case-Diagramm verwendeten Namen auch Synonyme stehen, falls es noch keine Einigung über einen einzigen Namen gibt oder mehrere Begriffe gebräuchlich sind.

Beim *Auslösenden Ereignis* wird die Frage beantwortet, wodurch der Use-Case initiiert wird.

Bei der *Kurzbeschreibung* handelt es sich um einen kurzen Prosatext, der in zwei bis vier Sätzen den Kern des Prozesses beschreibt.

Im Abschnitt *Vorbedingungen* werden die Voraussetzungen geklärt, die erfüllt sein müssen, damit der Use-Case sinnvoll ausgeführt werden kann. Leser werden kurz und bündig auf wichtige Bedingungen aufmerksam gemacht, die vorliegen müssen und können den Use-Case damit einfacher in den passenden

Kontext einordnen. Sie sollten die Vorbedingungen auch notieren, wenn sie beim Ablauf des Prozesses noch einmal explizit geprüft werden.

Die *Essenziellen Schritte* des Prozesses geben die Standardreaktion des Systems auf Wünsche und Forderungen der Systemumgebung wieder. Bereits bekannte Fehler- und Ausnahmesituationen werden im Abschnitt *Ausnahmefälle* festgehalten. Investieren Sie hier nicht zu viel Zeit, deuten Sie die möglichen Ausnahmefälle lediglich an. Sie werden später detailliert betrachtet und mittels geeigneter Diagramme dokumentiert.

Die *Nachbedingungen* beschreiben den, nach der Ausführung des Standardablaufs vorliegenden Zustand des Prozesses und mögliche Ausgaben.

Die Abschnitte *Zeitverhalten* und *Verfügbarkeit* geben Ihnen Raum, die bereits geäußerten nicht-funktionalen Anforderungen, wie schnell das System reagieren soll und wie häufig es ausfallen darf, zu notieren. Sollten Ihre Stakeholder noch weitere nicht-funktionale Anforderungen – z. B. an die Bedienbarkeit oder Wartbarkeit des Systems – formulieren, dann erweitern Sie Ihr Use-Case-Beschreibungsmuster einfach um die entsprechenden Zeilen.

Das letzte Feld *Fragen, Kommentare* ist ein temporäres Hilfsmittel für den Zeitraum, in dem Sie Ihr Requirementsgehirn füllen. Es erlaubt, Anmerkungen und Fragen zu hinterlegen, die im Gespräch mit den Stakeholdern geklärt werden müssen. Bearbeiten Sie diese Fragen spätestens dann, wenn die dort dokumentierten Unklarheiten Sie bei den nächsten Entwicklungsschritten behindern.

Die gesamte Use-Case-Beschreibung sollte allerdings nie länger als zwei Seiten sein. Sie stellt nicht das komplette Requirementswissen zu diesem Prozess dar. Dafür sind Use-Cases nicht gedacht. Sie sind nur ein erster Schritt auf dem Weg zu fundiertem Wissen über Ihr System<sup>2</sup>. Für die Dokumentation detaillierterer Informationen stellt die UML Ihnen eine Vielzahl geeigneter Notationen und Modelle zu Verfügung. In Kombination mit diesen Modellen können Sie dann detailliertes Wissen auch in Prosa dokumentieren.

Nicht länger als zwei Seiten

Die essenziellen Schritte, die bei der Ausführung des Systemprozesses durchlaufen werden, liefern wichtige Informationen über das Verhalten des Systems. Wie Sie sie am effektivsten erheben, notieren, auf die fachliche Essenz bringen, Ausnahme- oder Fehlerfälle oder technologische Besonderheiten behandeln, beantworten die folgenden Abschnitte.

## 5.1.1 Beschreibungen auf die Essenz bringen

Die zur Erstellung einer Systemprozess-Beschreibung relevanten Informationen erhalten Sie in aller Regel von den Stakeholdern, die etwas von diesem

<sup>2</sup> Der Versuch, das gesamte Wissen über Ihr System in Form von Use-Cases präzise zu formulieren, gleicht einem Rückschritt in die Steinzeit der Systemanalyse. Prosatexte sind kein Ersatz für eine etwas formalere Notation, sondern eine geeignete Ergänzung.

## 5 Anforderungen präzisieren

Prozess verstehen, ihn in der Praxis ausführen oder nutzen. Dabei stellen wir meist die folgenden Phänomene fest:

- Stakeholder reden oft in konkreten, technologiebehafteten Beispielen.
- Stakeholder nennen weniger fachliche Anforderungen, sondern vielmehr Lösungsdetails.

Auf Sie kommt nun die Aufgabe zu, die Äußerungen der Stakeholder abstrahiert und auf die fachliche Essenz beschränkt in die Systemprozess-Beschreibung aufzunehmen. In welcher Form Sie das tun – ob tabellarisch, stichpunktartig oder formal – bleibt Ihnen überlassen.

Zur Essenz gelangen

- Essentiell heißt lösungsneutral, d. h. frei von technologischen Entscheidungen und auch frei von organisatorischen Festlegungen.
- Essenzbildung heißt auch, einschränkende Reihenfolgen beseitigen, die nicht notwendig sind. Erhalten Sie möglichst viel der Parallelität, die Ihnen die reale Welt bietet. Sie geben damit Ihren Designern die Freiheit, die Parallelität zu erhalten oder die Prozesse in die Reihenfolge zu bringen, die für Ihr Systemdesign optimal ist.
- Essenz geht davon aus, dass im „Inneren“ des Systems perfekte Technologie verfügbar ist, die nie Fehler macht, keine Zeit braucht, immer genug Ressourcen hat und daher nie an ihre Grenzen gerät. Deshalb modellieren wir hier keine Fehlerfälle und keine internen Prüfungen auf technologische Grenzen.

Anhand des Systemprozesses „Gespräch führen“ eines Mobiltelefons verdeutlichen wir das Vorgehen.

**Tabelle 5.3:** Vergleich pragmatische/essenzielle Systemprozess-Beschreibung

Pragmatischer, technologiebehafteter Systemprozess	Essenzieller Systemprozess	
		
<p>Der Anrufer drückt den Knopf „Adressbuch“ auf seinem Mobiltelefon.</p> <p>Das Mobiltelefon zeigt dann den ersten Eintrag des Adressbuches an.</p> <p>Danach scrollt der Anrufer mit den Pfeiltasten zu einem gewünschten Eintrag und wählt diesen mit der Bestätigungstaste aus.</p> <p>Am Display erscheinen dann Name und Rufnummer des gewählten Eintrags (wegen der Bildschirmgröße auf 45 Zeichen beschränkt)</p> <p>Das Mobiltelefon ermittelt dann die zu dem Eintrag gehörende Nummer.</p> <p>Nach Betätigung der „Wähle-Taste“ baut es eine Verbindung zur Vermittlungsstelle auf und überträgt die Nummer.</p>	<p>Intention der Systemumgebung = <i>Akteur + Event</i></p> <p>Anrufer teilt die Rufnummer mit</p> <p>Anrufer initiiert den Wählvorgang</p>	<p>Essenzieller Schritt = <i>Reaktion des Systems</i></p> <p>1. Rufnummer entgegennehmen</p> <p>2. Verbindung mit Teilnehmer oder dessen Mailbox herstellen</p>

Der Provider baut die Verbindung auf und meldet den erfolgreichen Aufbau dem Mobiltelefon zurück, sofern die übertragene Nummer gültig ist. Das Mobiltelefon zeigt die „Verbindungsmeldung“ an. Evtl. wurde zur Mailbox umgeleitet.	Vermittlungsstelle bestätigt den Verbindungsauftbau	3. Verbindung halten	
Wenn der Anrufer das Gespräch beenden will betätigt er die „Auflegen-Taste“. Das Mobiltelefon zeigt dann die Meldung „Verbindung getrennt“ an.	Anrufer beendet Anruf	4. Verbindung trennen	

Die linke Spalte der Tabelle 5.3 zeigt einen **pragmatischen** Systemprozess, wie er von einem Stakeholder formuliert sein könnte. In der Beschreibung sind Annahmen über Reihenfolgen, technologische Entscheidungen und Einschränkungen eines existierenden Mobiltelefons genannt.

Rechts hingegen ist der **essenzielle** Systemprozess dargestellt, so wie Sie ihn in die Systemprozess-Beschreibung aufnehmen sollten. Die rechte Spalte zeigt die essenziellen Schritte, die vom System durchgeführt werden. Geben Sie zu jedem Schritt sowohl den Auslöser des Schrittes als auch das Ereignis an, welches das System veranlasst, den Schritt durchzuführen.

Die Essenzbildung reduziert die Komplexität von Vorgängen *deutlich*. So ist es nicht erstaunlich, dass oft unterschiedliche pragmatische Wege – meist von unterschiedlichen Stakeholdern formuliert – auf den gleichen essenziellen Ablauf abgebildet werden können. Seien Sie also nicht enttäuscht, wenn vielleicht fünf oder sechs Stakeholderinterviews „nur“ zu *einem* Ablauf führen.

Nur durch die Essenzbildung ermitteln Sie sicher die wahre Intention des Auftraggebers. Geben Sie sich nicht mit einer Beschreibung des bereits existierenden Systems zufrieden. Durch die Reduktion auf das Wesentliche haben Sie die Chance, eine **neue** pragmatische Lösung zu finden, losgelöst von bekanntem und althergebrachtem Vorgehen. Man nennt dies gemeinhin auch technologische Innovation ☺.

## 5.1.2 Technologie in Systemprozessen

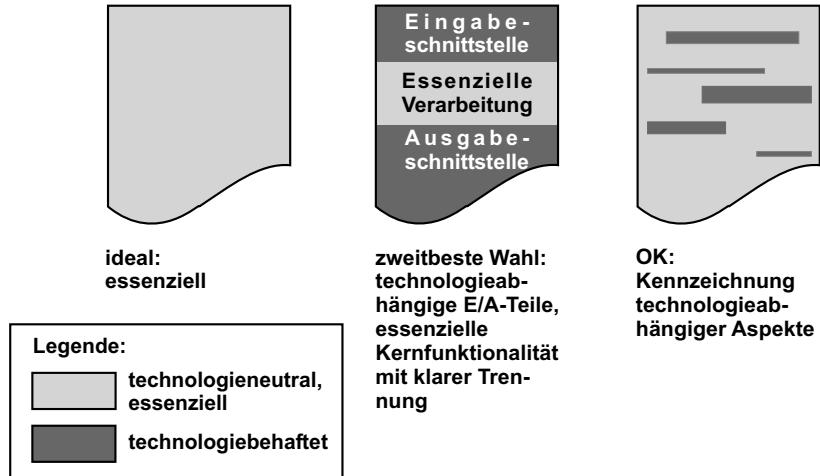
Wie wir im letzten Abschnitt erläutert haben, hat die Essenzbildung unter anderem das Ziel, technologiebehaftete Beschreibungen aufzudecken. Technologieneutrale Beschreibungen sind langlebiger und robuster. Technologische Anforderungen ändern sich im Allgemeinen viel schneller und häufiger als die Essenz der geforderten Funktionalität. Sie haben hier die einmalige Chance, möglicherweise falsch getroffene Organisations- und Technologieentscheidungen von Altsystemen bei einer Neuentwicklung zu korrigieren.

Allerdings kommen bei RTE-Systemen technologische Aspekte immer sehr früh ins Spiel. Stakeholder nehmen die Realität eben nicht getrennt nach fachlichem Kern und technologischer Realisierung wahr. Wir wollen deshalb technologielastige Informationen auch nicht verbannen, sondern zielgerichtet da-

Zielgerichteter Umgang mit technologischen Aspekten

## 5 Anforderungen präzisieren

mit umgehen. Ist beispielsweise Ihre Systemumgebung äußerst komplex, und müssen Sie viele Sensoren und Nachbarsysteme berücksichtigen, so sind dafür vermutlich bereits technologische Eigenschaften festgelegt. Sofern diese unumstößlich sind, sollten Sie sie auch notieren, aber als solche kennzeichnen. Abbildung 5.2 zeigt mögliche, bewertete Varianten, die bei Systemprozess-Beschreibungen in diesem Kontext auftreten können.



**Abbildung 5.2:** Essentielle und technologiebehaftete Anteile an einem Systemprozess

Enthält Ihre Beschreibung nur technologische Festlegungen bezüglich der Eingaben und Ausgaben, so lassen sich diese Informationen gut separiert darstellen. Bei einem Tempomaten könnten zum Beispiel von vornherein die Signale, die von den Nachbarsystemen gesendet werden, festgelegt sein. Notieren Sie alle für den Use-Case relevanten Signale mit den bereits bekannten technologischen Details am Anfang der Use-Case-Beschreibung und versehen Sie jedes eingehende Signal mit einem Namen, der den logischen Inhalt beschreibt. Für die weitere Beschreibung verwenden Sie dann nur noch diesen technologienutralen Namen.

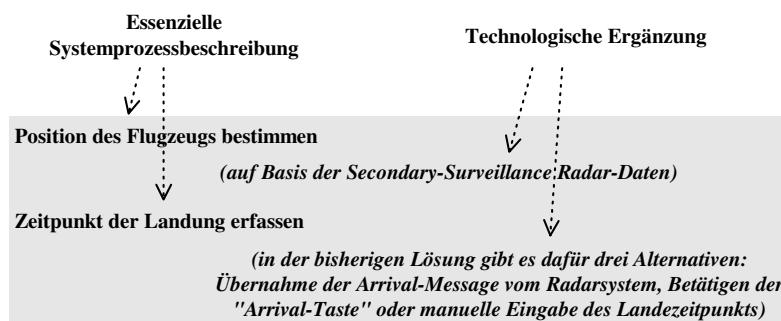
Ist hingegen der gesamte Systemprozess mit erwähnenswerten technologischen Entscheidungen durchwoven, lassen sich die spezifischen Stellen nicht räumlich abtrennen. In diesem Fall empfehlen wir Ihnen zu einem typografischen Mittel wie der Kursivschreibung, der Darstellung in Klammern, Einrückungen in der Formatierung oder Ähnlichem zu greifen.

Tabelle 5.4 stellt beispielhaft eine technologiebehaftete und eine essenzielle Systemprozess-Beschreibung für ein Anflugsystem der Flugsicherung gegenüber.

**Tabelle 5.4:** Vergleich technologiebehaftete/essenzielle Systemprozess-Beschreibung

Technologiebehaftete Systemprozess-Beschreibung	Essenzielle Systemprozess-Beschreibung
Secondary-Surveillance Radardaten des anfliegenden Flugzeuges auswerten.	Position des Flugzeugs bestimmen.
„Actual time of Arrival“ im Flugplan erfassen. Dazu „Arrival-Message“ des Radarsystems annehmen oder „ATA“ manuell mittels Drückens der „Arrival“-Taste zu Landezeitpunkt erfassen, oder Landezeit manuell über Tastatur eingeben.	Zeitpunkt der Landung erfassen.

Ist in dem Beispiel durch den verantwortlichen Stakeholder bereits entschieden, dass die Position des Flugzeugs definitiv mittels eines Secondary-Surveillance-Radar-Verfahrens ermittelt wird, so kann diese Information jederzeit zur essenziellen Systemprozess-Beschreibung hinzugefügt werden; sie sollte allerdings als technologiehaltige Information gekennzeichnet werden (in Abbildung 5.3 durch Klammern und Kursivschrift). Bleibt hingegen offen, wie – auf welche technische Art und Weise – der Landezeitpunkt erfasst wird, so fällt dies bei der essenziellen Beschreibung gewollt weg oder wird bewusst als „bisherige Lösung“ gekennzeichnet.



**Abbildung 5.3:** Essenzielle Systemprozess-Beschreibung mit technologischen Einschüben

Achten Sie bei Ihrem Vorgehen aber darauf, dass Sie die gewünschte Technologiefreiheit nicht um jeden Preis erzwingen. Essenzbildung ist immer eine Verallgemeinerung. Oftmals führt eine zu starke Verallgemeinerung bzw. Abstraktion zu Beschreibungen, die für den Leser unverständlich sind und damit die Kommunikation erschweren.

Wenn technologische Entscheidungen bereits feststehen, dann können Sie diese auch pragmatisch in die Beschreibung einfließen lassen, anstatt künstlich neue technologieneutrale Begriffe zu erfinden. Der Preis dieses pragmatischen Vorgehens ist eine schlechtere Wiederverwendbarkeit bei einem potenziellen Technologiewechsel, der Gewinn eine höhere Akzeptanz bei Stakeholdern, da die Begriffe in deren Sprachwelt verbleiben.

Essenz mit Augenmaß

### **5.1.3 Prüfen der Systemprozesszerlegung**

Reflektieren der Systemzerlegung	Beim Beschreiben der essenziellen Schritte eines Systemprozesses verifizieren Sie gleichzeitig die von Ihnen früher gefällte Entscheidung bezüglich der Aufteilung Ihres Systems in Systemprozesse (System-Use-Cases). Stoßen Sie bei der Beschreibung auf Schwierigkeiten, sollten Sie Ihre Systemzerlegung noch einmal überdenken. Für ein agiles Vorgehen ist es typisch, ausgeführte Schritte zu hinterfragen und bei Bedarf wieder zu verwerfen, wenn Ihr Wissen bezüglich des Aspektes inzwischen detaillierter geworden ist oder sich verändert hat. Die Diagramme im Requirementsgehirn sind nicht unantastbar und befinden sich nicht in einem eingefrorenen Zustand, vielmehr werden sie dem aktuellen Wissensstand angepasst und kontinuierlich verbessert.
Embrace Change	Sie sollten auch nicht zwanghaft auf die Konsistenz der Diagramme pochen. Oftmals verhindert die Befürchtung, dass eine Änderung viel organisatorischen Aufwand nach sich zieht, ein flexibles Reagieren auf Wissenszuwachs und Änderungen. Die Maxime „Embrace Change“ sollte immer Vorrang vor einer perfekten, konsistenten Dokumentation haben. Zu bestimmten Zeitpunkten werden Sie natürlich zur Statusbestimmung im Projekt und zur Qualitätssicherung der Konsistenz aller bisher gesammelten Informationen ihr wieder das Gewicht geben, das sie verdient. Sie werden dann geeignete Schritte einleiten, um Widersprüche zu beseitigen und um die Dokumentationslücken aufzufüllen, die für Ihren Projekterfolg relevant sind.
Zu wenig essenzielle Schritte	Sollten Sie für einen Systemprozess keine einzelnen essenziellen Schritte finden, so könnte Ihr Systemprozess bereits atomar und damit zu fein zerlegt sein. Sie sollten auf die Ebene des Use-Case-Diagramms zurückgehen (siehe Kapitel 3) und auf einem höheren Abstraktionsniveau zerlegen. Eventuell handelt es sich aber auch um eine rein technische Aufgabe, die dieser Systemprozess erfüllt, die in keine weiteren essenzielle Schritte zerfällt. An dieser Stelle ist unserer Meinung nach eine einfache Daumenregel hilfreich: Wenn Sie für einen Systemprozess weniger als drei essenzielle Schritte finden, dann sollten Sie die Granularität und die Schritte des Systemprozesses prüfen. Möglicherweise haben Sie den Systemprozess schon zu fein gewählt, und er beschreibt keinen selbstständigen Ablauf mehr. Oder Sie haben nicht alle essenziellen Schritte des Ablaufs gefunden. Wenn parallel dazu Ihr Use-Case-Diagramm sehr viele Systemprozesse umfasst, dann sollten Sie sie neu überdenken! Trotzdem kann es Fälle geben, bei denen Sie alles richtig gemacht haben und dieser Systemprozess eben nur wenige Schritte umfasst.
Bedingungsketten	Sollten Sie feststellen, dass einige Systemprozesse durch Nachbedingungen/Vorbedingungen miteinander verknüpft sind und dadurch vielleicht zeitlich nacheinander ablaufen, dann kann dies ein Zeichen für eine zu feine Zerlegung ihres Systems sein. Eventuell ist dies nur ein Systemprozess. Das heißt für Sie: zurück zu Kapitel 3 und nochmals die Zerlegung überprüfen.

## 5.1.4 Klassifizieren der Systemprozesse

Bei der Formulierung der essenziellen Schritte eines Systemprozesses kann es Ihnen passieren, dass Sie keine zeitlichen Abläufe finden. Gerade im RTE-Umfeld gibt es Prozesse, deren Schritte von externen Ereignissen abhängen. Weiterhin gibt es Aktivitäten, die ständig aktiv sind, ohne ein bestimmtes auslösendes Ereignis zu haben. Denken Sie beispielsweise an Scheduling- oder Timersysteme.

Reihenfolge der essenziellen Schritte?

Wir unterscheiden daher in diesem Kontext grob

- ablauforientierte und
- reaktive Systemprozesse.

**Ablauforientierte Systemprozesse** erkennen Sie daran, das ihre essenziellen Schritte in zeitlich logische Abfolgen gebracht werden können. Häufig erkennen Sie das bereits bei den natürlichsprachlichen Formulierungen ihrer Stakeholder. Begriffe wie „*anschließend, danach, nach Beendigung, im Folgenden...*“ in der Systemprozess-Beschreibung lassen auf zeitliche Folgen schließen. Können Sie die Beschreibung als kleine Geschichte (Story) erzählen, liegt meist ein Ablauf vor.

Ablauforientierte Prozesse

**Reaktive Systemprozesse** werden hingegen *immer wieder* durch Ereignisse beeinflusst. Das können Benutzereingaben, Signale von Nachbarsystemen, Geräten, Timern oder anderen Systemprozessen sein. Die Prozesse befinden sich in einem spezifischen Zustand und reagieren dementsprechend. Sie durchlaufen keine fest vorherbestimmte Aktionskette. Charakteristische Beschreibungsmerkmale in Stakeholderbeschreibungen sind hierfür „*ein Ereignis trifft ein, im Zustand ..., reagiert auf ..., wenn das passiert, dann ..., verweilt bis, ...*“.

Reaktive Prozesse

Warum diese Unterscheidung? Weil durch sie Ihr weiteres Vorgehen bestimmt wird! Bei ablauforientierten Systemprozessen empfehlen wir Ihnen, die zeitlichen Abfolgen der Systemprozess-Beschreibung durch ein **Aktivitätsdiagramm** zu präzisieren (siehe Abschnitt 5.2). Liegt hingegen ein reaktiver Prozess vor, eignen sich **Zustandsautomaten**, da mit Ihnen zustandsabhängiges Verhalten und asynchrone bzw. benutzergetriebene Reaktionen am besten modelliert werden können (siehe Abschnitt 5.3).

## 5.2 Abläufe präzisieren

Betrachten wir zunächst die Präzisierung von Systemprozess-Beschreibungen mittels Aktivitätsdiagrammen. Wir empfehlen sie, wenn Ihre essenziellen Schritte einen logischen, zeitlichen Ablauf ergeben.

## Aktivitätsdiagramme

Aktivitätsdiagramme sind das UML-Ausdrucksmittel für Ablaufdiagramme (oder Programmablaufpläne). Sie zeigen die Aktivitäten eines Prozesses und den Steuerfluss zwischen diesen. Der Beginn eines Ablaufs ist mit einem schwarzen Punkt markiert; das Ende mit einem „Bull’s Eye“. Verzweigungen werden mit kleinen Rauten dargestellt, an deren Ausgang zwei oder mehrere Bedingungen (in eckigen Klammern) angegeben werden können. Der Ablauf in einem Aktivitätsdiagramm kann auch parallel auszuführende Aktivitäten enthalten. Zur Aufspaltung des Steuerflusses und zur Synchronisation wird ein Balken verwendet.

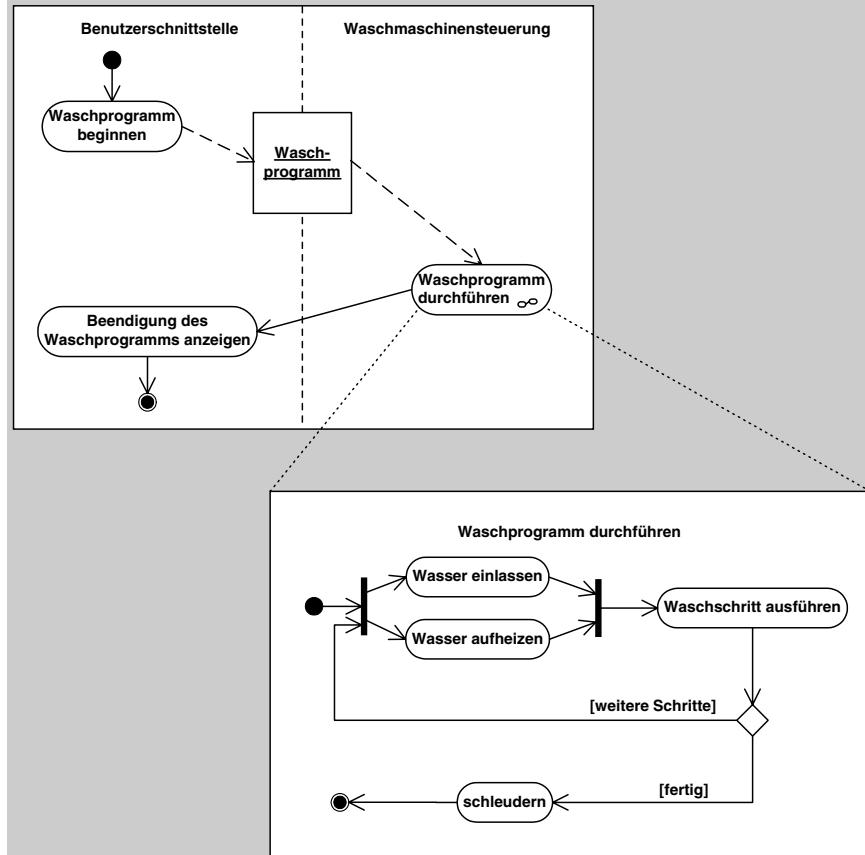


Abbildung 5.4: Basiselemente eines Aktivitätsdiagramms

Komplexe Aktivitäten können zerlegt werden. Die Teilaktivitäten werden wiederum in einem (verfeinerten) Aktivitätsdiagramm dargestellt. Zur Kennzeichnung der Verfeinerung kann ein stilisiertes Aktivitätsdiagramm ( $\infty$ ) als Stereotyp in das Symbol der komplexen Aktivität gezeichnet werden.

Zudem bietet Ihnen das Aktivitätsdiagramm die Möglichkeit, verschiedene Verantwortlichkeitsbereiche (swim lanes) zu definieren. Sie werden mittels senkrechter, gestrichelter Linie zur Trennung der Bereiche dargestellt. Jeder dieser Bereiche wird mit dem Namen des Verantwortlichen überschrieben, also demjenigen Objekt, Stakeholder oder Teilsystem, das die Aktivität ausführt.

Auch Objekte (dargestellt als Kästchen) können in Aktivitätsdiagramme eingezeichnet werden. Sie werden mit gestrichelten Linien mit den Aktivitäten verbunden, die sie lesen oder schreiben. Ein Objektfluss zwischen zwei Aktivitäten impliziert den Steuerfluss.

Besitzt Ihr System viele parallele Teilprozesse oder Verzweigungen, dann eignet sich ein Aktivitätsdiagramm, um Struktur in diesen Vorgang zu bringen. Müssen Sie mehrere Ausnahmefälle im essenziellen Ablauf berücksichtigen? Auch in diesem Fall verhilft Ihnen ein Aktivitätsdiagramm zu einer übersichtlichen Darstellung dieser Sachverhalte. Dagegen arten Fallunterscheidung in natürlicher Sprache häufig in komplizierte, schwer lesbare Gliederungsstrukturen und Verweise aus.

Wenn der von Ihnen bearbeitete Systemprozess allerdings sehr einfach ist, oder die Zusammenhänge zwischen den Aktivitäten Ihres Systems sich mit wenigen Zeilen Text beschreiben lassen und Sie den Überblick nicht verlieren, dann können Sie getrost auf die Modellierung verzichten<sup>3</sup>. Agiles Vorgehen heißt auch, auf Modelle und Diagramme zu verzichten, wenn der Aufwand den Nutzen übersteigt.

## Vorgehen

---

Wie Sie von der Use-Case-Beschreibung nun zum Aktivitätsdiagramm kommen, hängt stark von der Qualität und Präzision Ihrer vorliegenden Use-Case-Beschreibung ab. Sollte Ihre Use-Case-Beschreibung eher ein unstrukturierter Prosatext sein, müssen Sie beim Erstellen des Aktivitätsdiagramms die folgenden Fragen klären:

- Was genau sind die Aktivitäten meines Prozesses?
- Welche zeitlichen oder logischen Abhängigkeiten gibt es zwischen den einzelnen Aktivitäten?
- Welche Ereignisse starten oder beenden eine Aktivität?
- Wie ist der normale Ablauf zwischen den Aktivitäten?
- Welche Ausnahmefälle gibt es in meinem Standardablauf?

Haben Sie bereits in der Use-Case-Beschreibung den essenziellen Systemprozess sehr klar herausgearbeitet und dokumentiert, geht es bei der Erstellung des Aktivitätsdiagramms vor allem erst einmal darum, das Wissen grafisch

---

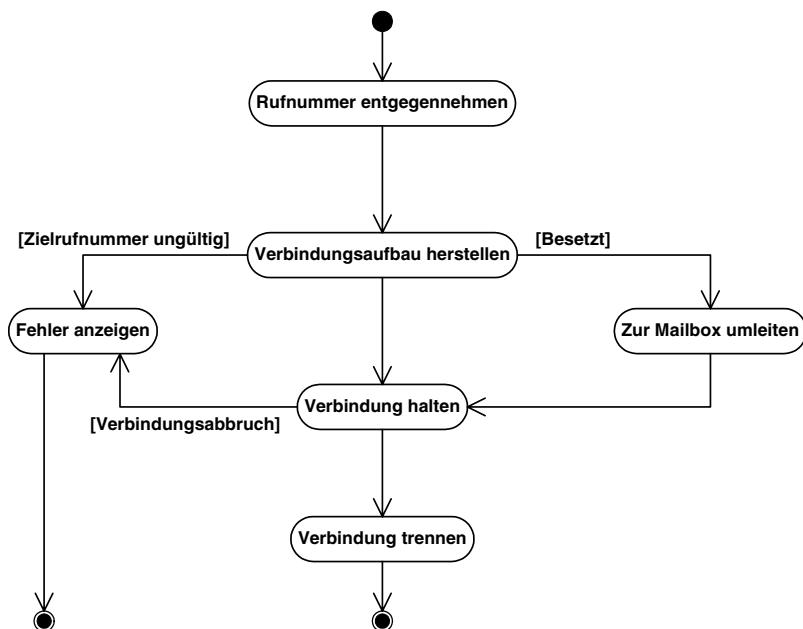
<sup>3</sup> Das Zusammenspiel zwischen Use-Case-Diagrammen und Aktivitätsdiagrammen in *kommerziellen Systemen* wird in [Oes00] ausführlich behandelt.

## 5 Anforderungen präzisieren

darzustellen. Danach werden Sie auf dieser Basis weiter analysieren, Wissen über Ausnahmefälle hinzufügen, den Ablauf verifizieren und Details erheben.

- Stellen Sie jeden essenziellen Schritt der Systemprozess-Beschreibung als eine Aktivität dar.
- Überführen Sie die zeitlichen und logischen Zusammenhänge in Steuerflüsse zwischen den Aktivitäten, so dass das Diagramm den identifizierten Ablauf nachbildet.
- Alleinstehende Aktivitäten, die nicht in einer definierten Reihenfolge enthalten sind (zum Beispiel Aktivitäten im Fehlerfall), müssen eventuell von *vielen* anderen Aktivitäten erreichbar sein und erhalten daher in der Regel eine große Menge von ein- und ausgehenden Steuerflüssen.
- Wenn Ihnen das Diagramm zu komplex wird, führen Sie abstraktere Aktivitäten ein und modellieren Sie diese als eigenes Aktivitätsdiagramm.

Abbildung 5.5 zeigt ein Beispieldiagramm zu dem Systemprozess von Tabelle 5.3. Zusätzlich sind noch einige fachliche Ergänzungen hinzugekommen.



**Abbildung 5.5:** Aktivitätsdiagramm „Gespräch führen“

Der Vollständigkeitstest

Steht das Grobgerüst unseres Aktivitätsdiagramms, kann im nächsten Schritt mit dem Überprüfen der einzelnen Aktivitäten auf mögliche Ausnahmen und Varianten begonnen werden. Hierzu sollte jede Aktivität und jeder Steuerfluss hinterfragt und Ausnahmen sowie mögliche Reaktionen auf fachliche Fehlerfälle in das Diagramm eingetragen werden. Hinterfragen Sie jede Aktivität im Diagramm.

- Kann bei dieser Aktivität etwas schief gehen? Gibt es einen Steuerfluss zu einer Folgeaktivität, die den fachlichen Fehlerfall weiterbehandelt (z. B. Fehlermeldung anzeigen)?
- Gibt es bei dieser Aktivität mehr als ein mögliches Ergebnis (z. B. Positiv- und Negativfall)? Hängt die gewählte Folgeaktivität von diesem Ergebnis ab? Sind alle möglichen Folgeaktivitäten auch im Diagramm enthalten?

Tragen Sie die gewonnenen Erkenntnisse in das Aktivitätsdiagramm ein.

Aktivitäten, die nur einen abgehenden Steuerfluss besitzen, neigen dazu, unvollständig zu sein. Möglicherweise haben Sie einen Steuerfluss vergessen oder die Aktivität kann mit ihrer Folgeaktivität zusammengefasst werden. Menschen nehmen Aktivitäten im Allgemeinen nur dann als getrennt wahr, wenn sie nicht zwingend aufeinander folgen, sondern Alternativen bestehen. Eine Aufteilung von Aktivitäten, die derartig stark verbunden sind und immer in genau der gleichen Reihenfolge auftreten, ist nur dann sinnvoll oder sogar zwingend, wenn sie durch Randbedingungen erzwungen wird. Hier spielen die in Kapitel 4 beschriebenen Entscheidungen und Randbedingungen eine wichtige Rolle: Sollte bereits jetzt klar sein, dass einzelne Prozessschritte in Subsystemen landen, die auf unterschiedlichen Knoten ablaufen werden, so sollten sie auch im Aktivitätsdiagramm bereits als getrennte Aktivitäten gezeichnet werden. Zudem ist es sinnvoll, Aktivitäten unterschiedlicher Verantwortlichkeitsbereiche oder Aktivitäten, die auf verschiedenen Daten arbeiten, zu trennen.

Aktivitäten trennen oder zusammen lassen

Bei diesem geschilderten Vorgehen bewegen wir uns bewusst von der essenziellen Abstraktionsebene wieder hin zu einer pragmatischen Ebene. Aktivitätsdiagramme können im Vergleich zu Systemprozess-Beschreibungen der Lösung einen Schritt näher und vollständiger sein. Auf spezifische technische Details sollten Sie aber auch hier möglichst noch verzichten. Modellieren Sie nur fachliche Ausnahmen. Situationen, die auf technischen Fehlern beruhen, sollten Sie nicht in das Diagramm aufnehmen.

Bewusst mehr Pragmatismus!

Nach den vielen Ergänzungen kann es sein, dass Sie ein extrem komplexes Aktivitätsdiagramm erhalten. Dies kann ein Indiz für eine falsche Aufteilung der Systemprozesse sein. Prüfen Sie nach, ob das Aktivitätsdiagramm aus mehreren, unabhängigen Prozessen besteht, die eigentlich gar keinen engen Zusammenhang im Sinne eines Use-Cases besitzen. Falls ja, dann überprüfen Sie Ihre Zerlegung des Systems erneut (siehe Kapitel 3). Sollten alle Aktivitäten zu einem Use-Case gehören und das Aktivitätsdiagramm immer noch komplex sein, dann sollten Sie einige Aktivitäten zu größeren Aktivitäten zusammenfassen und geschachtelte Aktivitätsdiagramme verwenden. Gelingt Ihnen die Zusammenfassung nicht, so liegen vielleicht Zustandsabhängigkeiten vor. Dann sollten Sie besser Zustandsdiagramme verwenden, die wir im folgenden Abschnitt behandeln.

## 5.3 Verhalten modellieren

Wenn Sie in der Systemprozess-Beschreibung keinen zeitlichen Ablauf finden und die Reihenfolge der Aktivitäten nicht vorhersagbar ist, sondern wenn externe und interne Ereignisse oder Zeitereignisse den Ablauf bestimmen, dann empfehlen wir, das Verhalten in einem Zustandsdiagramm zu modellieren.

### Zustandsdiagramme

Ein Zustandsdiagramm beschreibt das Systemverhalten durch Zustände, in denen das System eine bestimmte Zeit verweilt (dargestellt durch Rechtecke mit abgerundeten Ecken), und Ereignisse, durch die Zustandsübergänge ausgelöst werden (dargestellt durch Pfeile zwischen den Zuständen). Ein Ereignis kann dabei aus der Systemumgebung, von einem anderen internen Teilsystem oder von einem Timer ausgelöst werden. Auch die Aktivitäten in dem Zustandsdiagramm selbst können z. B. ihre Beendigung als Ereignis signalisieren.

Zustandsübergänge können zusätzlich durch Bedingungen (dargestellt durch logische Ausdrücke in eckigen Klammern) überwacht sein. Der Übergang findet statt, wenn das Ereignis eintritt UND die Bedingung wahr ist.

Nicht unterbrechbare, kurze Aktionen können direkt an den Übergängen angegeben werden; länger laufende, unterbrechbare Aktivitäten werden in den Zustand geschrieben (hinter dem Schlüsselwort „do“). Aktionen werden grundsätzlich komplett ausgeführt, bevor weitere Ereignisse überprüft werden. Aktivitäten können sich entweder nach einiger Zeit selbst beenden (wobei der Automat in dem Zustand bleibt) oder durch jedes Ereignis, das aus dem Zustand herausführt, abgebrochen werden.

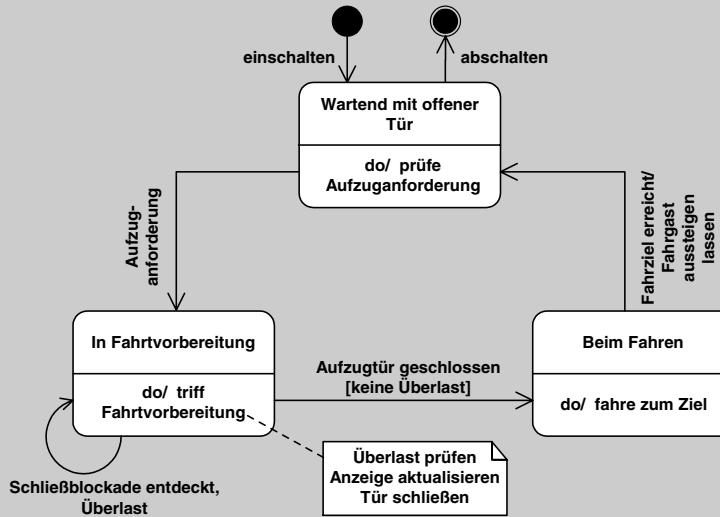


Abbildung 5.6: Basiselemente eines Zustandsdiagramms

Drei Ideen von David Harel [Har98]<sup>4</sup>, die das Modellieren von Systemverhalten noch erleichtern können, wurden in die UML aufgenommen. Erstens können Zustände geschachtelt werden: damit lassen sich Ereignisse, die auf ganze Teilsysteme wirken, bzw. Prioritäten von Ereignissen gut ausdrücken. Zweitens können Zustände in parallele Teilzustände verfeinert werden. Dadurch können Abhängigkeiten von Teilautomaten leicht verständlich dargestellt werden. Und drittens kann ein History-Zustand eingezeichnet werden (als „H“ in einem kleinen Kreis). Ein Übergang zu diesem History-Zustand bedeutet, dass ein geschachteltes Zustandsmodell wieder in den Zustand gelangen soll, in dem es vor dem Verlassen war.

Zustandsdiagramme helfen Ihnen, das Verhalten Ihres Systems besser zu verstehen. RTE-Systeme besitzen in vielen Fällen interne Zustände, von denen die Reaktion auf Ereignisse abhängt.

Zeichnen Sie Zustandsautomaten allerdings nur dann, wenn sie zur Erklärung des Systemprozesses dienen. Ein Automat mit nur zwei Zuständen und zwei trivialen Übergängen lohnt oft das Papier nicht. Eine Zustandsvariable mit zwei Werten ist hier vermutlich effizienter und genauso verständlich. Es gibt keinen Zwang, zu jedem reaktiven Systemprozess ein Zustandsdiagramm zu erstellen. Unnötigen Modellierungsaufwand zu vermeiden ist eine der wichtigsten Fähigkeiten, die Sie im Rahmen eines agilen Vorgehens benötigen.

Zustands-  
automaten mit  
Augenmaß

Auch wenn Sie einen Zustandsautomaten zur Klärung zeichnen, sollten Sie sich nicht den Zwang auferlegen, ihn perfekt auszumodellieren. Meist reicht es, jedem Zustand einen aussagekräftigen Namen zu geben und die Ereignisse für die Übergänge klar zu spezifizieren. Auf eine vollständige Beschreibung von Bedingungen und Aktionen können Sie häufig verzichten, sofern Sie Zustandsautomaten nicht zur Codegenerierung verwenden.

## Vorgehen

Wie Sie von der Use-Case-Beschreibung nun zum Zustandsdiagramm kommen, hängt stark von der Qualität und Präzision Ihrer vorliegenden Use-Case-Beschreibung ab. Sollte Ihre Use-Case-Beschreibung noch ein unstrukturierter Prosatext sein, so müssen Sie beim Erstellen des Zustandsdiagramms die folgenden Fragen klären:

- In welchen Zuständen verharrt mein System längere Zeit?
- Welche zeitlichen oder logischen Abhängigkeiten gibt es zwischen den einzelnen Zuständen?
- Welche Ereignisse bringen mich in einen Zustand hinein oder heraus?
- Wie ist der normale Übergang zwischen zwei Zuständen?
- Welche Ausnahmefälle gibt es, die mich in einen anderen Folgezustand versetzen?

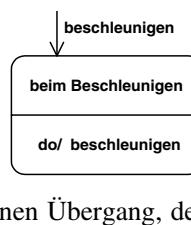
<sup>4</sup> Dies ist das Buch, das wir Ihnen empfehlen, wenn Sie intensiv mit Zustandsmodellen arbeiten.

## 5 Anforderungen präzisieren

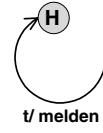
Haben Sie bereits in der Use-Case-Beschreibung die essenziellen Systemschritte sehr klar herausgearbeitet und dokumentiert, geht es bei der Erstellung des Zustandsdiagramms vor allem erst einmal darum, das Wissen in ein Zustandsdiagramm zu übernehmen. Danach werden Sie auf dieser Basis weiter analysieren und Wissen über weitere mögliche Ereignisse und Bedingungen hinzufügen.

Wenn die essenziellen Schritte und die Ereignisse (Events) in der Use-Case-Beschreibung bereits vorliegen, geht es nun um das Überführen ins Zustandsdiagramm.

- Unterscheiden Sie für jedes Ereignis in der Systemprozess-Beschreibung, ob der essenzielle Schritt unterbrechbar oder nicht unterbrechbar ist. Die unterbrechbaren werden zu Aktivitäten, die nicht unterbrechbaren zu Aktionen. Nach dieser Entscheidung konstruieren Sie das Zustandsdiagramm folgendermaßen:

- Für Aktivitäten erstellen Sie einen Zustand. Die Aktivität wird hinter dem Schlüsselwort `do` in den Zustand eingefügt. Finden Sie einen sinnvollen Namen für den Zustand. Wenn Ihnen nichts Besseres einfällt, wandeln Sie den Aktivitätsnamen durch den Zusatz „Beim“ in einen Zustandsnamen um (z. B. beim Landen, beim Beschleunigen). Zeichnen Sie einen Übergang, der in den Zustand führt, und beschriften Sie ihn mit dem Ereignis aus Ihrer Systemprozess-Beschreibung.
- Für Aktionen zeichnen Sie einen Übergang und beschriften ihn mit dem Ereignis. Direkt dahinter ergänzen Sie die Aktion.
- Untersuchen Sie freie Enden von Übergängen, ob diese potenziell Start- und Endzustände sind, und kennzeichnen Sie diese entsprechend.
- Finden Sie für Übergänge, die noch mit keinem Zustand verbunden sind, entsprechende Ausgangs- und Endzustände. Bevor Sie neue Zustände erfinden, sollten Sie die vorhandenen überprüfen. Scheuen Sie sich nicht das Ereignis auch an viele Zustände zu hängen, falls es dafür relevant ist.
- Wenn ein Ereignis aus vielen Zuständen heraus- oder hineinführt, sollten Sie einen höheren Zustand in Erwägung ziehen, um die Komplexität des Diagramms zu reduzieren. Benennen Sie diesen höheren Zustand sinnvoll.

- Falls ein Ereignis den Zustand offensichtlich nicht ändert, können Sie es als Übergang auf sich selbst an allen betroffenen Zuständen einzeichnen. Abkürzend gilt dies natürlich auch für höhere Zustände. In diesem Fall sollten Sie History verwenden, um das innere Zustandsverhalten nicht zu verändern. Dies trifft sehr oft auf Zeitereignisse zu.
- Reichern Sie die Übergänge nun mit den bereits geforderten nicht-funktionalen Gesichtspunkten wie dem Zeitverhalten an.



Überprüfen Sie Ihr Zustandsmodell nach folgenden Regeln auf Vollständigkeit:

- Besitzt jeder Zustand mindestens einen ein- und ausgehenden Übergang? (Wenn nicht: Ist das ein Start- oder Endzustand?)
- Ist Ihr Zustandsautomat insgesamt zusammenhängend? (Wenn nicht: Haben Sie vielleicht Ereignisse vergessen oder besteht Ihr Systemprozess eigentlich aus zwei zusammenhangslosen Prozessen?)
- Reagiert das System in jedem Zustand richtig auf alle gefundenen Ereignisse? Wahrscheinlich haben Sie die offensichtlichen Übergänge bereits modelliert. Oft übersieht man, dass bereits gefundene Ereignisse auch in anderen Zuständen Wirkung zeigen<sup>5</sup>.

Auf der Basis einer gut strukturierten Systemprozess-Beschreibung erhalten Sie mit diesem Vorgehen meist einen sehr guten ersten Wurf eines Zustandsautomaten.

Haben Sie die grundlegenden Zustände und Übergänge aus der Systemprozess-Beschreibung gefunden, ist eine zu den Aktivitätsdiagrammen analoge Vorgehensweise ratsam. Suchen Sie Spezialitäten, Fehlersituationen oder bringen Sie wieder ein bisschen Pragmatik in den Automaten. Gerade die Anreicherung mit nicht-funktionalen Anforderungen in Kombination mit der Vollständigkeitsprüfung macht manche Lücken offenbar. Abbildung 5.7 zeigt einen Zustandsautomaten zu dem in Tabelle 5.2 beschriebenen Systemprozess „Geschwindigkeit regeln“ unseres Tempomatenbeispiels.

---

<sup>5</sup> Diese Überprüfung geht am leichtesten, wenn Sie Ihr Zustandsmodell als Matrix mit den Zuständen und Ereignissen als Zeilen und Spalten darstellen. Sie können dann systematisch Zelle für Zelle überprüfen.

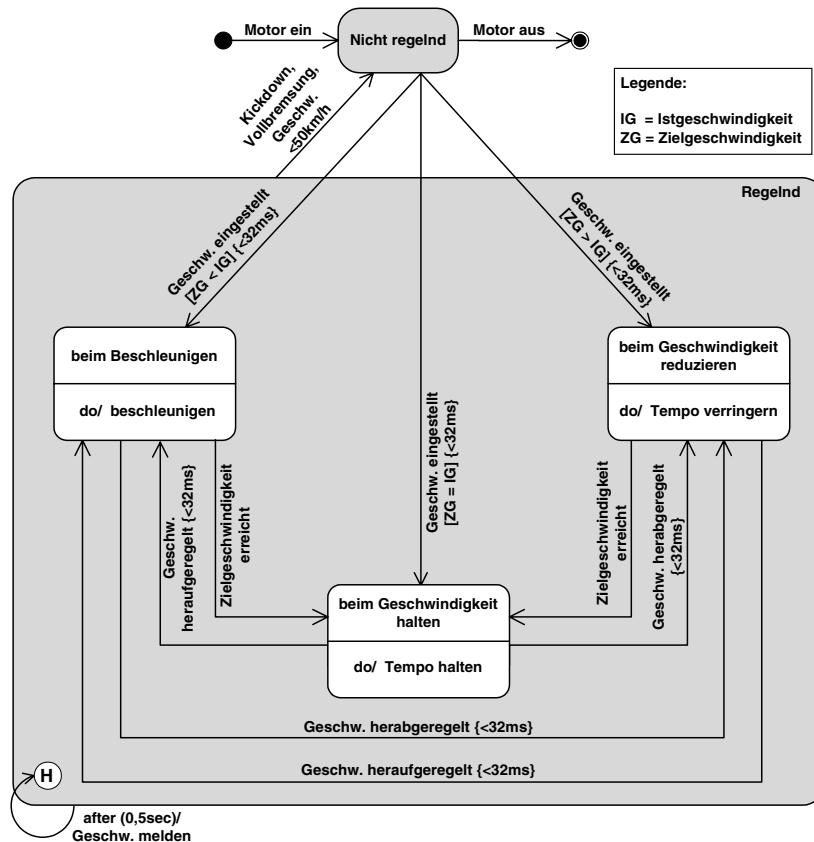


Abbildung 5.7: Zustandsdiagramm des Systemprozess „Geschwindigkeit regeln“

## 5.4 Fachliche Dinge präzisieren

Sie haben nun einige Möglichkeiten kennengelernt, die Systemprozesse näher zu spezifizieren. Wenden wir uns nun den Dingen zu, mit denen diese Prozesse arbeiten. In Kapitel 3 hatten wir Sie bereits gebeten, die wichtigsten Dinge und Begriffe zu definieren. Was tut man aber, wenn es zu viele werden?

In diesem Abschnitt kümmern wir uns um die Strukturierung unserer Begriffe. Wir nutzen dazu ein Klassendiagramm, in dem wir die wichtigsten Dinge unserer Systeme und ihre Zusammenhänge aufzeigen. Ein Klassendiagramm bietet einen Strukturrahmen für Ihre Begriffe und Definitionen, mit dessen Hilfe sie beherrschbar und weiterentwickelbar werden. Stellen Sie an diesen ersten Entwurf eines Klassendiagramms noch keinen Anspruch auf Vollständigkeit.

Es ist eine erste Sammlung von Klassen, die als Gerüst dienen, um kontinuierlich mehr Wissen zu sammeln.

Zunächst konzentrieren wir uns vor allem auf Entity-Klassen. Alle anderen Arten von Klassen stellen wir Ihnen später im Kapitel 9 vor.

Um nicht in eine „Analyse-Paralyse“ zu geraten, empfehlen wir Ihnen die verfügbare Zeit für die Erstellung eines ersten Wurfes des Klassendiagramms einzuschränken. Gönnen Sie sich nicht mehr als ein paar Stunden oder bei großen Systemen einige Tage, um den ersten Klassendiagrammwurf zu erstellen. Wenden Sie sich dann dem dynamischen Systemverhalten zu. Dabei werden Sie immer wieder relevante Informationen finden, die Sie kontinuierlich in das Klassendiagramm integrieren sollten.

„Analyse-Paralyse“

## Klassendiagramme

Ein Klassendiagramm ist eine grafische Darstellung der statischen Struktur eines Systems. Klassen werden in Form von Kästchen dargestellt. Jede Klasse kann die folgenden Informationen enthalten:

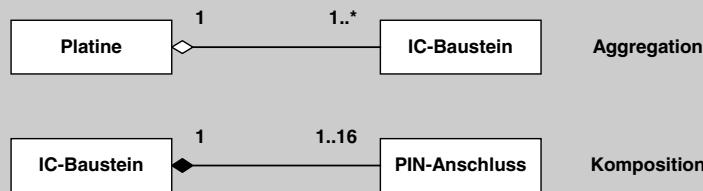
«Stereotyp»
Paket::Klasse {Merkmal}
Attribut
Operation()

- einen Namen und evtl. Zusatzangaben wie z. B. Stereotypen oder Paketname im obersten Feld,
- Attribute und evtl. Initialwerte und Zusicherungen im mittleren Feld und
- Operationen evtl. mit Ein- und Ausgabeparametern im unteren Feld.

**Abbildung 5.8:** Aufbau einer Klasse nach UML

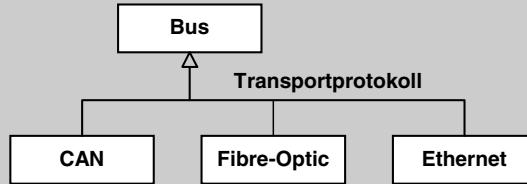
Die Zusammenhänge zwischen Klassen werden durch Beziehungen modelliert. Drei Beziehungsarten sind dabei besonders interessant.

- Die „normale“ Assoziation wird durch eine einfache Verbindungsline dar gestellt. Um Beziehungen zwischen Klassen weiter zu präzisieren, können Multiplizitäten eingeführt werden. Sie geben an, mit wie vielen Objekten der assoziierten Klasse ein Objekt verbunden sein kann.
- Eine Teile-Ganze-Beziehung, ausgedrückt durch eine Raute beim „Ganzen“, kennzeichnet ein Objekt, das sich aus mehreren Objekten zusammensetzt.



**Abbildung 5.9:** Aggregation und Komposition im Vergleich

- Die Generalisierung/Spezialisierung, ausgedrückt durch eine hohle Pfeilspitze bei der Generalisierung. Erbende Klassen übernehmen Operationen, Attribute, Beziehungen und Verhalten der vererbenden Klasse.



**Abbildung 5.10:** Vererbungsbeziehung

Wir haben uns hier bewusst kurz gefasst. Die vielen Details und Feinheiten zu Klassenmodellen werden ausführlich in fast allen UML-Büchern behandelt.

In den ersten Iterationen Ihres Klassendiagramms werden Sie vor allem die Namen von Klassen erfassen und Beziehungen einzeichnen. Bevor Sie das dynamische Verhalten des Systems nicht genauer untersucht haben, ist es nicht sehr sinnvoll, länger über Operationen nachzudenken. Tragen Sie die bereits bekannten Multiplizitäten an die Beziehungen an. Ersparen Sie sich beim Einzeichnen der Beziehungen jedoch Streitgespräche, ob es sich um eine Aggregation, eine Komposition<sup>6</sup> oder eine normale Assoziation handelt, oder ob eine andere Art der Modellierung eine bessere Wiederverwendbarkeit sichern würde. All diese Detaildiskussionen sind hier verfrüht.

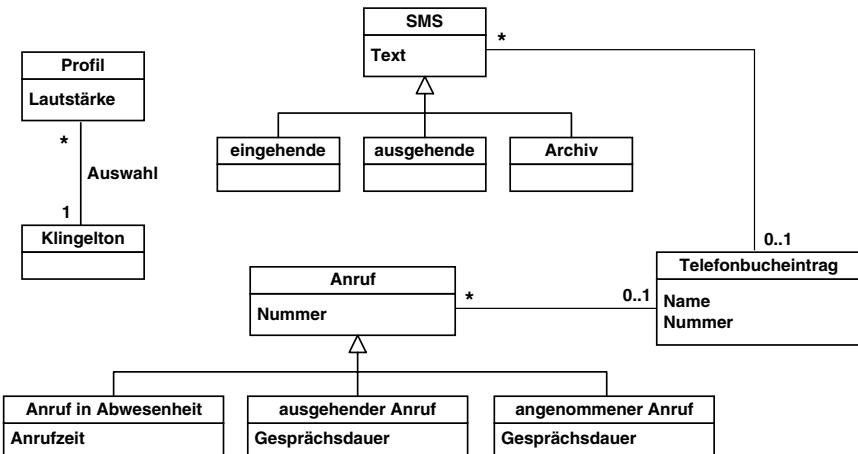
Wenn Sie am Ende über ein Klassendiagramm bestehend aus gut benannten Klassen, einigen Attributen und den bereits bekannten Beziehungen verfügen, haben Sie gute Arbeit geleistet.

Bei RTE-Systemen werden Sie im Gegensatz zu kommerziellen Produkten manchmal nur wenige Entity-Klassen finden. Betrachten Sie z. B. den schon häufiger erwähnten Tempomaten: die einzigen beiden Daten, für die er sich wirklich interessiert, sind die Istgeschwindigkeit und die Zielgeschwindigkeit. Erstere erhält er aus seiner Umgebung ständig aktuell zugeliefert. Es bleibt also nur noch ein Attribut (Zielgeschwindigkeit) übrig, das sich der Tempomat merkt. Es lohnt sich zu diesem Zeitpunkt nicht, eine Klasse mit diesem einen Attribut zu zeichnen. Dieses System ist eher regelungsintensiv – deshalb lernen Sie mit Systemprozessen und Zustandsautomaten mehr als mit Entity-Klassen. Ersparen Sie sich hier im Moment den Aufwand für die Erstellung eines Klassenmodells.

---

<sup>6</sup> Das Thema „Aggregation, Komposition oder normale Beziehung“ wird in vielen Use-groups in Hunderten von Einträgen diskutiert. Sofern eine Beziehung nicht eindeutig einer dieser Beziehungsarten zuzuordnen ist, empfehlen wir einfach eine normale Beziehung zu verwenden. Verzetteln Sie sich nicht in Detaildiskussionen, die in den Bereich der Designentscheidungen gehören.

Nehmen wir als Gegenbeispiel jedoch unser Mobiltelefon. Es soll sich eine Vielzahl von Daten merken. Deshalb lohnt sich eine erste Skizze eines Klassenmodells. Das Mobiltelefon kennt eine Menge von Namen und Nummern, die Sie üblicherweise anrufen, es speichert die letzten SMS, die Sie erhalten oder verschickt haben, es kennt die letzten ausgegangenen Anrufe und die zuletzt angenommenen oder in Abwesenheit erfolgten Anrufe. Außerdem kennt es viele Klingeltöne (bei Jugendlichen sogar noch mehr!), zahlreiche andere Geräteparameter, sowie viele weitere Informationen.



**Abbildung 5.11:** Ausschnitt eines Klassendiagramms zu einem Telefonbucheintrag

Wenn wir alleine dieses Wissen zu Papier bringen wollen (ohne uns über die Funktionen des Mobiltelefons Gedanken zu machen), dann bietet uns ein Klassenmodell eine gute Strukturierungsmöglichkeit. Abb. 5.11 zeigt eine erste Skizze für Klassen, erste Attribute und Beziehungen unseres Mobiltelefons.

Bei der Erstellung des Klassendiagramms können Sie auf alle bereits durchgeführten Schritte zurückgreifen, um geeignete Informationen zu identifizieren.

■ Einige Tipps zur Erstellung eines Entity-Klassendiagramms:

- Die von Ihnen gesammelten Definitionen umfassen die wesentlichen Fachbegriffe des Fachgebietes und sollten daher auch die Dinge umfassen, die jetzt als Klassen modelliert werden. Untersuchen Sie Ihre Definitionen auf Klassenkandidaten.
- Sammeln Sie die Attribute, die Ihnen im Gespräch oder in Dokumenten (wie Benutzerhandbücher der Vorgängersysteme) über den Weg laufen. Legen Sie keinen Wert auf Vollständigkeit. Nehmen Sie nur die offensichtlichen.
- Ordnen Sie diese Attribute sinnvollen größeren Einheiten zu, den Entity-Klassen. Versuchen Sie, für diese aussagekräftige Namen zu finden.

Tipps für die  
Erstellung eines  
Entity-  
Klassenmodells

## 5 Anforderungen präzisieren

- Auch aus den erstellten Systemprozess-Beschreibungen lassen sich Attribute und Klassenkandidaten ableiten. Nutzen Sie das bereits dokumentierte Wissen.
- Haben Sie ein Aktivitätsdiagramm erstellt? Dann überlegen Sie sich für jede Aktivität, welche Informationen von dieser Aktivität erzeugt oder genutzt werden. Auch das sind potenzielle Attribute und Entity-Klassen.
- Nutzen Sie Teile-Ganze-Beziehungen, wenn Sie Phrasen wie „besteht aus“ oder „ist Teil von“ hören.
- Nutzen Sie Generalisierung/Spezialisierung, wenn diese sich anbieten, d.h. wann immer Stakeholder Kategorien von Dingen erwähnen, die einige gleiche Eigenschaften haben und einige unterschiedliche (wie z. B. Anrufe im Abwesenheit und angenommene Anrufe). Wir werden diese Entscheidungen später noch kritisch überprüfen.
- Überlegen Sie sich Multiplizitäten für die Beziehungen, bei denen Sie schon genau wissen, ob Sie eins oder mehr davon wollen. Verwenden Sie bewusst Fragezeichen anstelle der Multiplizitäten, wenn Sie die Anzahl der verbundenen Objekte noch nicht genau kennen oder sie nicht erfragt haben.

Beschreibungs-  
muster für  
Klassen

Auch für Klassen haben wir einen Vorschlag für ein Beschreibungsmuster, das später noch um weitere Informationen ergänzt wird. Notieren Sie anfangs zu jeder Klasse:

**Tabelle 5.5:** Beschreibungsmuster für eine Klasse

Name	ein aussagekräftiger Name und eventuell verwendete Synonyme
Definition	was stellt die Klasse dar und warum ist sie im Kontext Ihres Systems wichtig
Zusatzinformationen	Details, welche die Stakeholder zu dieser Klasse äußern. Diese Informationen werden im Lauf der Modellierung weiterverarbeitet und landen dann z. B. als Attribute, Operationen oder Beziehungen dieser Klasse im Modell
Attribute	sofern sie Ihnen über den Weg laufen
Fragen/Bemerkungen	Notiz für folgende Gespräche

Technologie in  
Klassenbeschrei-  
bungen

Beachten Sie bitte, dass Sie die Klassen essenziell beschreiben. Sollten Sie hier dennoch bereits Informationen hinterlegen wollen, die nicht essenziell sind, dann kennzeichnen Sie diese – genauso wie bei der Use-Case-Beschreibung – zum Beispiel mit typografischen Mitteln. Auch hier gilt: Technologie nicht krampfhaft vermeiden, sondern zielgerichtet damit umgehen.

„Analyse-  
Paralyse“

Noch einmal zur Erinnerung: Verrennen Sie sich beim ersten Entwurf eines Klassendiagramms nicht in einen Perfektions- oder Vollständigkeitsanspruch, investieren Sie nur wenige Stunden oder wenige Tage.

## 5.5 Beispielhafte Abläufe diskutieren

Bei der Erstellung von Systemprozess-Beschreibungen stoßen Sie häufig auf das Problem, dass die Stakeholder nicht direkt die fachlich essenziellen Schritte liefern. Insbesondere, wenn Ihr RTE-System viele parallele Prozesse besitzt, fällt es manchen Stakeholdern schwer, sich die Funktionsweise des Systems vorzustellen und den entscheidenden Prozess zu beschreiben. Hier helfen Ihnen Szenarien weiter.

Mein Stakeholder denkt nicht essenziell!

### Szenarien

Ein Szenario ist eine *spezifische* Abfolge von Ereignissen und Aktivitäten. Es beschreibt beispielhaft *genau einen möglichen Ablauf* eines Prozesses. Wegen ihrer Einfachheit und Nähe zur Realität sind Szenarien ein geeignetes Mittel für die Kommunikation mit den Stakeholdern. Durch die klare Fokussierung auf einen realen Ablauf eines Prozesses erkennt der Stakeholder einen Teil seiner fachlichen Realität wieder. Besitzt ein Prozess Verzweigungspunkte, so wird in einem Szenario meist nur ein möglicher Weg dargestellt.

Wie viele Szenarien brauche ich?

Den Sinn einzelner Szenarien bezweifelt kaum jemand. Die Streitfrage ist meist, wie viele und welche Szenarien herangezogen werden sollten. Hier hilft Ihnen bei einem agilen Vorgehen eine risikogetriebene Abschätzung. Welche Prozesse sind Ihnen noch unklar? Setzen Sie genau bei diesen unklaren Prozessen Szenarien ein und versuchen Sie damit, so viel Klarheit zu gewinnen, wie Sie für Ihre weiteren Schritte benötigen. Bei welchen Prozessen haben Sie Bedenken, einzelne Situationen nicht wirklich durchdacht zu haben? Versuchen Sie an derartigen Stellen ruhig auch einmal, für einen Systemprozess mehrere mögliche Szenarien zu erstellen, um die notwendige Sicherheit zu erlangen.

Szenarien können auf unterschiedlichste Art ermittelt und festgehalten werden. Da sie kein Selbstzweck sind, sollten Sie immer die einfachste Art wählen und möglichst wenig Aufwand treiben. So können Sie z. B. die derzeitigen Abläufe einfach beobachten, mittels Videokamera oder Schnittstellenmonitoren mitschneiden und anschließend auswerten. Zudem können Sie im Gespräch mit den Stakeholdern zukünftige Szenarien konstruieren und textuell, als Skizze oder mit UML-Darstellungsmitteln notieren. Die UML sieht für Szenarien Sequenz- und Kollaborationsdiagramme vor. Wir halten diese auf Systemebene für weniger geeignet und empfehlen sie erst für Softwareszenarien (siehe Kapitel 9 und 10).

## **5.6 Sind Ihre Anforderungen präzise genug für die weitere Planung?**

---

In diesem Kapitel haben wir über die Systemstrukturierung aus Sicht der Anforderungsaktivitäten diskutiert. Betrachten Sie Ihr bisher erreichtes Ergebnis kritisch:

- Haben Sie die Systemprozesse hinreichend detailliert verstanden, so dass Sie erste Designentscheidungen treffen können?
- Konnten Sie die Systemprozesse auf die fachliche Essenz reduzieren und Technologieabhängigkeiten deutlich kennzeichnen?
- Ist es Ihnen gelungen, Abläufe mit Aktivitätsdiagrammen, Zustandsdiagrammen oder anhand von Szenarien mit Ihren Stakeholdern abzustimmen?
- Haben Sie die Abhängigkeiten zwischen Systemprozessen überprüft (evtl. unter Zuhilfenahme von Zustandsmodellen)?
- Haben Sie die zentralen Begriffe, die in den Abläufen auftreten, durch Einbindung in fachliche Klassenmodelle geordnet?

„Maximen beim Handeln sind notwendig, um der Schwäche des Augenblicks Widerstand leisten zu können.“

*Arthur Schopenhauer (1788–1860),  
deutscher Philosoph*

# 6

## **Systemarchitektur: Mutig Entscheidungen treffen**

---

### **Fragen, die dieses Kapitel beantwortet:**

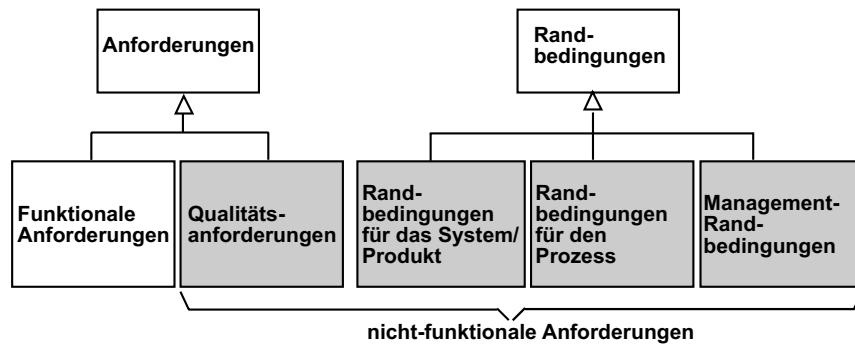
- Welche Entscheidungen muss man als Systemarchitekt treffen?
- Woher bekommt man die Entscheidungsgrundlagen?
- Wie dokumentiert und überprüft man die Systemarchitektur?
- Woraus besteht die Systemarchitektur? Welche Modelle braucht man?
- Warum sollte man zwischen Hardwarestruktur und logischen Subsystem unterscheiden?
- Warum sind nicht-funktionale Anforderungen so wichtig für die Entscheidungen über die Systemarchitektur?

„Architecture is a framework for change“. So brachte Tom DeMarco die Herausforderung für Systemarchitekten auf den Punkt. Sie lernen in diesem Kapitel, wie man Architekturziele explizit macht, damit Sie zwischen dem stabilen „Rahmen“ und den Teilen, die bewusst flexibel und ausbaubar gestaltet werden sollen, unterscheiden können.

## 6.1 Designen = Entscheidungen treffen

Architekten treffen mutige Entscheidungen

Architekturen entstehen durch mutige Architekten, die auf Basis der Anforderungen und unter Berücksichtigung von Randbedingungen Lösungsalternativen abwägen und dann Entscheidungen treffen. Diese Entscheidungen führen dazu, dass manche Ziele besser, manche schlechter erreicht werden. Dies betrifft vor allem langfristige Qualitätsziele wie Wartbarkeit, Stabilität bei Änderungen, Anpassbarkeit und Portierbarkeit. Eine Entscheidung für eine Alternative ist gleichzeitig immer eine Entscheidung gegen andere Möglichkeiten. Architekten und Designer müssen ständig die Konsequenzen ihrer Entscheidungen vorhersehen können und dann abwägen. Dazu sollten sie ihre Designziele kennen und wissen, welches davon wie wichtig ist.



**Abbildung 6.1:** Anforderungen und Randbedingungen

Vorgaben und Freiheitsgrade

Abbildung 6.1 zeigt fünf Kategorien von Anforderungen und Randbedingungen im Überblick. Zu jeder Kategorie werden Ihre Stakeholder Forderungen stellen. Hoffentlich betreffen diese eher deren wirkliche Wünsche und nicht allzu sehr Randbedingungen, damit dem Designer genügend Freiheitsgrade bleiben. Was auch immer die Stakeholder gefordert haben: Es wird zu Fesseln für den Architekten. Der Architekt kann frei über die Technologien zur Realisierung entscheiden, es sei denn, jemand hat bestimmte Technologien gefordert. Der Projektleiter kann das Vorgehen bei der Entwicklung festlegen, es sei denn, jemand hat das Vorgehensmodell für die ganze Firma festgelegt. Sicherlich bleibt in all den Bereichen noch genügend Entscheidungsspielraum für alles, was nicht explizit vorgegeben wurde. Das sind die Freiheitsgrade für den Architekten.

Bei RTE-Systemen bestimmen einige der nicht-funktionalen Anforderungen (wie Sicherheit, Verfügbarkeit oder verlangte Standards) die Architektur stärker als die Funktionalität. Dem Designer bleiben manchmal nur wenige Alter-

nativen. Das systematische Berücksichtigen dieser Vorgaben ist deshalb für Entwicklungsprojekte von RTE-Systemen oft ein erfolgsentscheidender Faktor, ihre Vernachlässigung dagegen der Auslöser von Projektkatastrophen.

## 6.2 Nicht-funktionale Anforderungen systematisch präzisieren

In Kapitel 5 haben wir Ihnen gezeigt, wie man funktionale Anforderungen präzisiert. Die Systemprozesse wurden mit Hilfe von essenziellen Beschreibungen, sowie Aktivitäts- und Zustandsmodellen, analysiert. Unsere Arbeitsmittel haben wir in Form von Klassenmodellen strukturiert.

Nun müssen wir noch alle nicht-funktionalen Anforderungen systematisch aufsammeln, hinterfragen und dokumentieren. Wir wollen sie auch möglichst sofort unseren Modellen dort zuordnen, wo man sie am ehesten braucht und weiter berücksichtigen muss. Im Folgenden betrachten wir zuerst die Qualitätsanforderungen genauer und danach die Randbedingungen.

### 6.2.1 Qualitätsanforderungen

Die Kategorien von Qualitätsanforderungen haben wir schon in Abschnitt 1.4 erläutert. Hier finden Sie Hinweise und Beispiele, wie man sie formuliert. Wir konzentrieren uns auf einige wesentliche Kategorien und beschreiben den Inhalt (Was wird hier gefordert?), die Motivation (Warum brauchen wir diese Kategorie?), konkrete Beispiele aus verschiedenen Anwendungen und Abnahmekriterien (Wie prüfen wir die Erfüllung dieser Anforderung?).

#### Zeitanforderungen

In der Kategorie Zeitanforderungen werden hauptsächlich Antwortzeiten für bestimmte Aufgaben festgelegt. Bei RTE-Systemen mit harten Zeitanforderungen müssen manche Prozesse (z. B. das Auslösen des Airbags in Ihrem Auto) unbedingt innerhalb festgelegter Zeitspannen abgeschlossen sein, sonst kann dies zu katastrophalen Fehlern führen. Auch bei Systemen mit weichen Zeitanforderungen sind die Zeitangaben wichtig (z. B. bei Automaten zum Aufladen von Wertkarten in der Kantine, wo zwar bei Zeitüberschreitung keine Katastrophe eintritt, aber unerwünschte, lange Warteschlangen entstehen).

Wie formuliert man Zeit-anforderungen?

So präzise sollten Sie Zeitanforderungen formulieren:

Beispiele

„Der Airbag muss innerhalb von 10ms nach einem Aufprall des Fahrzeuges, mit mehr als 30km/h, voll aufgeblasen sein.“

„Der Automat darf für das Aufladen der Wertkarte nach erfolgter Geldeingabe nicht mehr als 3 Sekunden brauchen, bis die Karte wieder vollständig freigegeben ist und der Automat für den nächsten Vorgang zur Verfügung steht.“

### Abnahmekriterien

Und so sollten Sie zum Beispiel die Anforderungen überprüfen:

„Von 10 Tests des Airbags unter verschiedenen Umweltbedingungen müssen alle in weniger als 10ms zum vollständigen Aufblasen geführt haben.“

„Der Automat muss innerhalb der ersten zwei Betriebswochen bei beliebiger Stückelung der Geldeingabe in 95 Prozent aller Fälle die korrekt gefüllte Wertkarte in weniger als 3 Sekunden freigeben.“

### Modellzuordnung

Zeitanforderungen können unseren Modellen an vielen Stellen zugeordnet werden. Betrifft die Zeitanforderung einen kompletten Systemprozess, können Sie die Reaktionszeiten entweder in der Systemprozess-Beschreibung unterbringen oder in einem Szenario zu diesem Prozess. Betrifft die Zeitanforderung eher eine Teilaufgabe, dann steht Ihnen die Aktivitätsbeschreibung zur Verfügung (falls die Teilaufgabe als Aktivität modelliert wurde) oder die Subsystembeschreibung, der die Teilaufgabe verantwortlich zugeordnet wurde. Im weiteren Verlauf können Sie diese Zeitbudgets durch Analyse- oder Designentscheidungen auf kleinere Einheiten verteilen.

## Zuverlässigkeit und Verfügbarkeit

---

Diese Kategorie hält Forderungen nach der Zuverlässigkeit für Ihr Gesamtsystem bzw. für Teile davon fest. Dies geschieht normalerweise durch die Angabe der erlaubten Zeit zwischen Fehlern (MTBF – mean time between failure), der erlaubten Fehlerrate oder der Maximalzeit, die nach Fehlern für die Wiederherstellung der Verfügbarkeit benötigt werden darf (MTTR – mean time to repair). Die Anforderungen beziehen sich meist auf das gesamte Produkt aus Hardware und Software (und anderen Technologien wie Mechanik, Elektronik, ...). Zuverlässigkeit der Software alleine reicht oft nicht aus. Innerhalb des Gesamtsystems werden besonders harte Zuverlässigungssangaben jedoch oft nur für ausgewählte kritische Funktionalitäten gefordert.

### Beispiele

„Das Abschaltsystem des Kernreaktors muss während der Reaktorbetriebszeit ständig verfügbar sein.“

„Die Nachrichtenvermittlungszentrale darf in 1000 Betriebsstunden maximal 2 Stunden ausfallen.“

### Modellierung

Es gibt mehrere natürliche Anknüpfungspunkte für Zuverlässigkeitssanforderungen: Entweder das Gesamtsystem oder den betroffenen Knoten (mit aller darauf laufenden Software) oder das betroffene Subsystem. Verwenden Sie die jeweiligen Beschreibungsmuster (siehe Abschnitte 6.3 und 6.4). Der Designer muss daraus dann bei der Gestaltung der Architektur – Schritt für Schritt nachvollziehbar – Forderungen für Teilsysteme ableiten, um am Ende beim

Integrationstest durch die Messergebnisse für einzelne Komponenten die Erfüllung der Gesamtforderung belegen zu können.

Verfügbarkeitsforderungen werden im Design oft durch Redundanz von Hardware- und Softwarekomponenten gelöst. Das führt zu neuen Systemprozessen (z. B. Umschalten auf Stand-by-Komponente), die wiederum modelliert werden müssen. Stellen Sie sicher, dass diese zusätzliche Funktionalität oder Hardware, die nur wegen dieser Verfügbarkeitsanforderungen hinzukamen, als solche gekennzeichnet werden. Dadurch finden Sie bei Änderungen der nichtfunktionalen Anforderungen die entsprechenden Teile Ihres Systems wieder. Bei Nichtkennzeichnung besteht die Gefahr, dass die Komponenten langsam zur „Folklore“ im System werden: Keiner weiß mehr, warum sie da sind, aber keiner wagt es, solche Teile zu entfernen.

## **Sicherheit für Leib und Leben (Safety)**

In dieser Kategorie müssen Sie das Risiko eines möglichen Schadens für Menschen, Güter und Umwelt bewerten. In einem sicheren System sind diese Risiken vertretbar klein.

„Die Gefahr, dass der Roboterarm Bedienungspersonal verletzt, soll ausgeschlossen werden.“ „Das Abschaltsystem des Kernreaktors muss so konzipiert werden, dass es auch bei Hardware-/Softwarefehlern den Reaktor in einen sicheren Zustand versetzt.“

Beispiele

Überprüfen Sie die Erfüllung der Sicherheitsanforderungen z. B. mit folgenden Tests:

Abnahmekriterien

„Der Roboter darf einen Mitarbeiter, der sich bei bewegendem Roboterarm in den Bereich begibt, nicht verletzen, unabhängig von der Geschwindigkeit mit der sich der Mitarbeiter bewegt. „Der Roboter darf einen Mitarbeiter, der in dem Bereich des Roboters unbeweglich steht, auch bei Aufnahme der Robotertätigkeit nicht verletzen.“

„Das Abschaltsystem muss bei Netzausfall mindestens solange seine Funktionalität zur Verfügung stellen, bis ein sicherer Zustand erreicht ist.“ „Das Abschaltsystem muss bei Ausfall des Betriebssystems solange seine Funktionalität zur Verfügung stellen, bis ein sicherer Zustand erreicht ist.“

In dem Bereich Sicherheit gibt es zahlreiche Standards und Normen, die für derartige Systeme befolgt werden müssen. Einen guten Überblick gibt [Dou99]. Wenn Sie sicherheitskritische Systeme erstellen, dann gibt es in Ihrem Unternehmen bestimmt Spezialisten, die mit diesen Normen vertraut sind. Nehmen Sie diese in Ihre Stakeholderliste auf und nutzen Sie deren Fachwissen beim Erfassen von Sicherheitsanforderungen.

Welche Auswirkungen haben Sicherheitsforderungen auf die Modelle? Mit hoher Wahrscheinlichkeit erhalten Sie in Ihrem Klassenmodell Schadens- oder

Modellierung

Fehlerklassen (als Klassen, als Attribute, als Stereotypen für Subsysteme, ...). Mit diesen Informationen werden Sie auch neue Prozesse und Aktivitäten formulieren, die Schadensklassen überwachen und bei Schadenseintritt Aktionen auslösen. Bauen Sie auch Sicherheitszustände in Ihre Zustandsautomaten ein und ergänzen Sie die bisher gefundenen Standardabläufe um sicherheitsrelevante Aktivitäten. Wichtig ist auch hier, wie im vorigen Abschnitt bereits erwähnt, dass Sie alle durch Sicherheitsanforderungen entstandenen Modellelemente in den Beschreibungen kennzeichnen, damit Sie diese Funktionalität wieder bis zur ursprünglichen Anforderung zurückverfolgen können und bei der Weiterentwicklung des Systems noch wissen, woher diese Daten und Funktionen kommen.

Die hier erwähnten Kategorien von Qualitätsanforderungen sind nur Beispiele. Weitere Informationen zum Umgang mit nicht-funktionalen Anforderungen finden Sie in [Volere] und [Rupp02].

### 6.2.2 Randbedingungen

Tabelle 6.1 katalogisiert die Randbedingungen. Nehmen Sie diese Katalogisierung als Hilfestellung, damit Sie an alle Punkte einmal denken und mit Ihren Stakeholdern abklären, ob es in diesem Umfeld Forderungen und damit Fesseln für den Entwurf gibt.

**Tabelle 6.1:** Kategorien von Randbedingungen für die Systementwicklung

1. Randbedingungen für das System/Produkt	
	1.1 Vorgaben für Einbettung und Verteilung
	1.2 Vorgeschrriebene Technologien
	1.3 Physikalische Anforderungen
	1.4 Umweltanforderungen
2. Randbedingungen für den Prozess	
	2.1 Anforderungen an das Vorgehensmodell
	2.2 Anforderungen an Inbetriebnahme und Migration
	2.3 Anforderungen bezüglich Systemsupport
3. Management-Randbedingungen	
	Vorgaben über Zeit, Budget, Personal, ... (mehr dazu in [Ger02])

Textuelle Anforderungen

Die meisten Anforderungen und Randbedingungen werden in Textform beschrieben, einige können sofort in Modelle umgesetzt werden. Für die Textform schlagen wir nach [Rob99]/[Rupp02] folgendes Muster als Minimum vor. Sie können gerne Beschreibungsfelder, die Ihnen für einzelne Kategorien von Randbedingungen notwendig erscheinen, ergänzen.

**Tabelle 6.2:** Muster zur Beschreibung von Anforderungen und Randbedingungen

<b>Beschreibung:</b>	Umgangssprachliche Formulierung der Anforderung
<b>Quelle:</b>	Wer hat die Anforderung gestellt? (für Rückfragen, ...)
<b>Begründung:</b>	Warum wurde die Anforderung gestellt?
<b>Abhängigkeit:</b>	Verweise auf andere Anforderungen, aus denen diese Anforderung abgeleitet wurde oder mit der sie in Wechselwirkung steht.

## Randbedingungen für das System/Produkt

Zu den Vorgaben für die Einbettung und Verteilung zählen vorgeschriebene Nachbarsysteme und Nachbarapplikationen, vorgegebene Schnittstellen zur Systemumgebung, aber auch Vorgaben zur geografischen Verteilung Ihres Systems oder Forderungen nach Einsatz von bestimmten Prozessoren für bestimmte Aufgaben.

Vorgaben für Einbettung und Verteilung

Das Feld der möglichen Technologievorgaben ist unheimlich weit. Sie müssen z. B. bestimmte Hardware, Geräte oder Netzwerke einsetzen. Oder man legt Ihnen nahe, existierende Teilsysteme wieder zu verwenden oder „COTS“-Produkte (Commercial off the shelf) einzusetzen. Die Basissoftware wie Betriebssysteme, Datenhaltungssysteme, Libraries oder Middlewareprodukte werden häufig als verbindliche Teile der Lösung vorgegeben. Auch die Forderung nach Einhaltung bestimmter interner Schnittstellen fällt in diesen Bereich.

Technologie-vorgaben

Viele dieser Randbedingungen können direkt modelliert werden; einige andere werden textuell beschrieben und an Modelle angebunden. Vorgegebene Nachbarsysteme werden im Kontextdiagramm dargestellt. Vorgegebene COTS und Libraries fließen direkt in die Subsystembildung ein. Forderungen nach bestimmter Hardware oder nach Netzwerken lassen sich im Verteilungsdiagramm ausdrücken oder hinterlegen. Vorgegebene Middleware kann entweder textuell erfasst und dem Gesamtsystem angehängt werden oder unmittelbar als Subsystem in die Architektur eingezeichnet werden. Im letzteren Fall bitte in der Beschreibung das Subsystem als Forderung kennzeichnen, damit es nicht als frei änderbare Designentscheidung betrachtet wird.

Physikalische Anforderungen

Zu den physikalischen Anforderungen gehören alle Forderungen nach Größe, Gewicht und Form des Produkts, Vorgaben für die maximale Stromaufnahme, aber auch Forderungen aus der Produktionsabteilung, die Sie beim Design berücksichtigen müssen.

Betrachten wir als Beispiel die Forderung nach einer maximalen Höhe, Breite und Länge des Produkts. Das sieht zunächst wie eine Einschränkung nur für den Hardwareentwurf aus und wird daher in der Beschreibung zum Verteilungsdiagramm erfasst. Die Forderung kann aber Folgewirkung für Prozessoren und Speicher haben, die innerhalb des beschränkten Raums Platz haben müssen. Dies hat wieder Auswirkungen auf die Software, die nun mit Spei-

cherplatzrandbedingungen leben muss (die eigentlich aus einer Größenbeschränkung kommen). Genau dafür sind im Muster für die Anforderungsbeschreibung in Tabelle 6.2 die Abhängigkeitsangaben vorgesehen.

### Umweltanforderungen

In die Kategorie Umweltanforderungen zählen wir Ansprüche an die Hitzebeständigkeit, mechanische Anforderungen (wie Stoßfestigkeit, Vibrationsempfindlichkeit), elektrische Anforderungen (wie das Verhalten bei Über- oder Unterspannung, bei kurzen Stromausfällen), Operabilitätsforderungen in Feuchtigkeit (wie Regen- oder Spritzwasserbeständigkeit), bei Verschmutzungen und unter Strahlungseinfluss.

Als Beispiel könnte gefordert werden: „Der Pulsmesser soll unter den folgenden Umgebungsbedingungen einsetzbar sein:

- Betriebstemperatur zwischen –5 bis +40 Grad Celsius
- Lagertemperatur zwischen –35 bis +60 Grad Celsius
- Relative Luftfeuchtigkeit 20–80%
- Höhe über Meeresspiegel: bis 3500 Meter“

Diese Forderung hat hauptsächlich Auswirkungen auf den Hardware- bzw. mechanischen Entwurf. Indirekt können Umweltanforderungen auch Auswirkungen auf die Software haben. Bedenken Sie, dass z. B. beim Starten des Autos Unterspannungen entstehen, die auf ein laufendes Telefongespräch oder die Radiofunktionalität störend wirken und so gut wie möglich kompensiert werden müssen.

## Randbedingungen für den Prozess

---

### Traditionen und Vorschriften hinterfragen

Diese Kategorie von Randbedingungen bezieht sich nicht auf das RTE-System selbst, sondern auf den Erstellungsprozess des Systems. Oft sind diese Vorgaben außerhalb Ihres Projekts festgelegt. Es ist Firmentradition oder Vorschrift und man ist versucht, darüber gar nicht nachzudenken. Ein Merkmal agiler Prozesse ist es jedoch, solche Randbedingungen bewusst zu hinterfragen oder wenigstens die vorhandenen Annahmen explizit zu machen.

Sie können sich sicherlich vorstellen, dass Sie zu anderen Lösungen kommen, wenn man Ihnen ein anderes Vorgehensmodell vorschreibt. Sie entwerfen Ihr Produkt vielleicht ganz anders, wenn Sie rechtzeitig daran denken, wie und in welchen Teilen es später ausgeliefert und in Betrieb genommen werden muss. Auch die rechtzeitige Absprache mit Personen, die das System hinterher betreuen müssen (Hotline, Wartungsteams) ändert vielleicht Ihren Lösungsansatz oder Ihre Herangehensweise.

Betrachten wir als Beispiel die Automobilindustrie. Sie legt (wie auch andere) Musterstände für Produktentwicklungen fest, zu denen vorgegebene Ergebnisse geliefert werden müssen. Derartige Randbedingungen für das Vorgehen bestimmen die Art, den Zeitpunkt der Verwendung und den Detaillierungsgrad für alle Modelle, die wir in diesem Buch vorstellen.

Mehr Details und Beispiele zu den Kategorien von Randbedingungen finden Sie auf unserer Web-Page.

### 6.2.3 Prioritäten für den Entwurf vorgeben

Nicht-funktionale Anforderungen verlangen teilweise Dinge, die nicht gleichzeitig optimiert werden können oder die nach widersprüchlichen Zielen streben. So stehen Sicherheitsanforderungen im Zielkonflikt mit Verfügbarkeitswünschen. Ein sicheres System wird öfter abschalten wollen, während das Verfügbarkeitsziel verlangt, die Funktionalität möglichst dauernd zur Verfügung zu haben. Auch Flexibilität zieht den Entwurf in eine andere Richtung als bestmögliches Antwortzeitverhalten. Aus Flexibilitätsgründen ist der Architekt geneigt, Zwischenschichten einzuziehen, wodurch die Performance eventuell negativ beeinflusst wird. Flexibilität verträgt sich auch nicht mit der Robustheit von Systemen: Wenn man vieles leicht ändern kann (und will), dann wird das System vielleicht etwas weniger robust. Und fast jede Forderung von Stakeholdern hat Einfluss auf Managementziele wie Fertigstellungstermine und Kosten.

Zielkonflikte

Ein Architekt kann nicht allen Herren gleich gut dienen. Um Abwägungen treffen zu können und Kompromisse für die Lösung zu finden, braucht man für den Entwurf Prioritätsvorgaben. Wir schlagen Ihnen deshalb vor, die System-/Produktziele explizit mit einer „Hitparade“ der Top5 oder Top10 Architekturziele zu ergänzen.

Hitparade der Architekturziele

Bei einem Produkt für den Massenmarkt wie einem Mikrowellenherd könnte die Liste so beginnen:

**Tabelle 6.3:** Prioritätsvorgaben für den Architekten

1. Sicherheit (Elektromagnetische Verträglichkeit)
2. Einfachheit und Robustheit der Bedienung
3. Wiederverwendung: Aus Kostengründen sollen möglichst viele vorhandene Bau-teile, Komponenten und Algorithmen eingesetzt werden.
4. ...

Diese Liste erspart Ihnen keinesfalls die genaue Beschreibung aller Kategorien von nicht-funktionalen Anforderungen. Der Designer lernt daraus aber, dass er im Zweifelsfall eher das Gerät in einen sicheren Zustand bringen soll, als einen Vorgang bei Fehlern weiterzuführen. Die Liste drückt auch aus, dass Ersparnisse durch Wiederverwendung die Bedienbarkeit des Gerätes nicht schlechter machen sollen. Wenn also Zielkonflikte auftreten, so konsultieren die Entwickler diese Prioritätsvorgaben.

Das Schreiben dieser Liste ist ein sehr kurzer Vorgang. Die Prioritäten knallhart festzulegen ist etwas schwieriger, da die meisten Stakeholder alles haben wollen und zwar 100%ig. Es ist aber für ein Projekt sehr förderlich, diese Dis-

„Alles 100%ig“ gibt's nicht!

kussion frühzeitig zu führen – und die Ergebnisse festzuhalten! Selbstverständlich dürfen Stakeholder ihre Meinung auch danach noch ändern, aber das könnte zu größeren Architekturänderungen führen.

### 6.3 Hardwareverteilung entscheiden

Wie kommen Sie dazu, einen oder mehrere Prozessoren für Ihre Lösung einzusetzen? Woran sollten Sie bei dieser Entscheidung denken? Wie wir eben beschrieben haben, ist bestimmte Hardware vielleicht als Randbedingung gefordert. Damit müssen Sie nichts mehr entscheiden. Wenn Sie aber Freiheitsgrade haben, müssen Sie jetzt über eine mögliche Hardwarestruktur nachdenken. Die Entscheidung für bestimmte Prozessoren und Kanäle ist nicht immer durch die Logik oder die Funktionalität des Systems begründbar. Im Folgenden zeigen wir Ihnen viele andere Gründe für die Auswahl einer Prozessorstruktur.

- Verfügbarkeit von Prozessoren: „Wir haben die Prozessoren schon eingekauft/installiert/in Betrieb und die Anwendung kann sie nutzen.“
- Gute Erfahrungen: „Mit dem Alpha-Prozessor sind wir auf der sicheren Seite. Er hat sich im letzten Projekt gut bewährt.“
- Leistung wird gebraucht: „Aus Performance-, Sicherheits- oder anderen Gründen wird mindestens ein 16-Bit-Prozessor mit 8MB Flash benötigt.“
- Räumliche/geografische Verteilung wird benötigt: „Wir brauchen an folgenden Standorten lokale Rechenleistung: ...“ (Dabei dominiert die Forderung nach lokaler Verarbeitung zunächst die Aufteilung im Verteilungsdiagramm, nicht die Leistungsmerkmale der Prozessoren).
- Standards: Bei bestimmten Kritikalitätsstufen der Funktionalität fordern Standards den Einsatz getrennter Prozessoren.
- Getrennte Verkaufbarkeit: Die Vermarktung des Produkts in Form von Teilprodukten kann die Hardwareaufteilung beeinflussen. Die getrennte Vermarktung ist jedoch heute kein zwingender Grund mehr für verteilte Hardware. Eventuell ist es aus Kostengründen billiger, einen Rechner mit dem Gesamtprodukt zu liefern und Teile der Funktionalität einfach softwaremäßig erst nach Kauf frei zu schalten.
- Marktgründe: Die Hardwarestruktur wird manchmal auch durch die Konstellation kooperierender Unternehmen vorgegeben, die gemeinsam ein Produkt entwickeln und aus politischen Gründen, Haftungsgründen oder wegen ihres technischen Know-hows lieber auf separaten Prozessoren entwickeln.

Nicht-funktionale Anforderungen als Treiber

Oft sind auch nicht-funktionale Anforderungen die treibenden Kräfte für die Hardwareverteilung. Die Sicherheit entscheidet die Verteilung, wenn ein Backup-Server aus Feuerschutzgründen oder Versicherungsgründen in einem

anderen Gebäude untergebracht werden muss. Zuverlässigkeit und Verfügbarkeit können zu einer redundanten Auslegung der Knoten führen, die dann im Hot Stand-by-Betrieb für das nötige Maß an Ausfallsicherheit sorgen. Auch Montageanforderungen können die Hardwarestruktur beeinflussen: Die Standardgröße von Racks führt evtl. zum Einsatz von mehreren Prozessoren in unterschiedlichen Slots.

## Knoten- und Verbindungsspezifikationen

Ist die Hardwareverteilung entschieden, sollten Sie nicht nur ein Verteilungsdiagramm zeichnen, sondern jeden Knoten und jede Verbindung beschreiben. Die Diagramme enthalten nur den Überblick. Die wirklich interessanten Fakten stehen in den Beschreibungen hinter den Diagrammen. Folgende Tabelle 6.4 gibt ein Muster für die Beschreibung von Knoten vor:

Nicht nur  
zeichnen,  
beschreiben!

**Tabelle 6.4:** Muster für Knotenbeschreibungen

<b>Name:</b>	Bezeichnung des Knotens
<b>Verantwortung:</b>	Kurzer Text zur Rolle des Knotens im Gesamtsystem.
<b>Funktionale Anforderungen:</b>	Prozesse, Daten oder Subsysteme benennen, die diesem Knoten zugeordnet wurden. (Dieser Teil kann entfallen, wenn Sie die Tabelle aus Abschnitt 4.5 systematisch pflegen.)
<b>Designbegründung:</b>	Warum wurde dieser Knoten ausgewählt? Welche Alternativen hat man verworfen? Bei längeren Abwägungen wird hier nur auf ein separates Dokument verwiesen.
<b>Designrechtfertigung:</b>	Warum glauben Sie, dass dieser Knoten das leistet, was seine Beschreibung behauptet? Wie werden durch diesen Knoten die Anforderungen erfüllt?
<b>Nicht-funktionale Anforderungen:</b>	Alle Qualitätsanforderungen und Randbedingungen, die diesem Knoten zugeordnet wurden.

Nehmen wir als Beispiel die Grafikplatine aus Abbildung 4.1. Ihre Spezifikation könnte folgendermaßen aussehen:

Beispiel

**Tabelle 6.5:** Beispiel für Knotenbeschreibungen

<b>Name:</b>	Grafikplatine
<b>Verantwortung:</b>	Übernimmt alle Funktionen zur 3D-Simulation und Ansteuerung des Video-Interface-Boards.
<b>Funktionale Anforderungen:</b>	Subsystem Simulation, Subsystem VIB-Interface
<b>Designbegründung:</b>	Die Simulationsvorgänge sind so zeitintensiv, dass dafür ein eigener Prozessor gewählt wurde. Sie mussten vom Hauptprozessor ausgelagert werden, damit dieser die Benutzerinteraktionen zeitgerecht ausführen kann.

<b>Designrechtfertigung:</b>	Unsere Tests haben ergeben, dass diese Grafikplatine derzeit am Markt das ausgewogenste Preis-/Leistungsverhältnis aufweist (sh. Dok. TN-2001/10b).
<b>Nicht-funktionale Anforderungen:</b>	1.2.17, 1.2.21, 1.3.9

Beschreibungs-  
muster  
Verbindungs-  
spezifikation

Ein ähnliches Muster schlagen wir auch für Verbindungsspezifikationen vor:

**Tabelle 6.6:** Muster für Verbindungsspezifikationen

<b>Name:</b>	Bezeichnung der Verbindung
<b>Charakteristika:</b>	Kenngrößen und Einschränkungen. (Meist durch Verweis auf Industriestandards oder eigene Standarddokumente.)
<b>Funktionale Anforderungen:</b>	Zuordnung der Informationen, die über diese Verbindung transportiert werden.
<b>Designbegründung:</b>	Warum wurde diese Verbindung ausgewählt? Welche Alternativen gab es?
<b>Designrechtfertigung:</b>	Warum glauben Sie, dass diese Verbindung das leistet, was seine Beschreibung behauptet? Wie werden durch diese Verbindung die Anforderungen erfüllt?
<b>Nicht-funktionale Anforderungen:</b>	Alle Qualitätsanforderungen und Randbedingungen, die dieser Verbindung zugeordnet wurden.

Als Designrechtfertigung würden Sie hier z. B. Durchsatzforderungen einführen, oder redundante Auslegung der Leitungen aus Sicherheitsgründen oder einfach Standardkonformität.

Diese Beschreibungsmuster sollen Ihnen helfen, wichtige Aspekte der Hardwarestruktur explizit zu machen. Nutzen Sie die Muster im agilen Sinne so weit, wie sie Ihre Projektarbeit beschleunigen. Ein Ausfüllen, ohne das Gehirn eingeschaltet zu haben, bringt Sie dem Ziel nicht näher.

## 6.4 Subsysteme entscheiden

Im letzten Abschnitt haben wir unser System oder Produkt aus Sicht der Hardwareverteilung betrachtet. Ein weiterer Ansatzpunkt zur Strukturbildung ist die systematische Suche nach fachlichen, logischen Subsystemen. Durch viele Methoden der letzten Jahrzehnte wurden wir darauf getrimmt, zuerst unsere Anforderungen systematisch erforscht und modelliert zu haben, bevor wir ernsthaft über Subsystembildung nachdenken. Machen Sie sich frei von diesem Gedanken. Wir haben schon in Abschnitt 4.3 aufgezeigt, dass Sie aus manchen Anforderungen von Stakeholdern direkt zu ersten Skizzen von Subsystemen kommen können. Setzen Sie diesen Gedankengang als Designer jetzt bewusst fort – mit oder ohne vorhandene Requirements. In den Kapiteln 9 und 10 zeigen wir Ihnen später den systematischen Bottom-up-Weg zur Architektur. In diesem Kapitel gehen wir das Problem eher top-down an. Die folgende

Liste enthält einige Denkanstöße, wie Sie aus nicht-funktionalen Anforderungen zu Subsystemen kommen.

- Sicherheitsanforderungen (Security) können Sie dazu bringen, ein eigenes Subsystem für diese Aspekte zu entwerfen, das vielleicht strengerer Regeln unterworfen wird, als die „normale“ Funktionalität.
- Zuverlässigkeit und Verfügbarkeit können, wie bei Hardware, Auslöser für bewusste Redundanz sein.
- Portabilitätsforderungen werden Sie vielleicht zu einer Schichtenarchitektur führen mit separaten Schnittstellen für alle Aspekte, die eventuell ausgetauscht werden könnten.
- Effizienzforderungen bezüglich der Systemressourcen werden die Abwägung zwischen persistenten Daten und Berechnungen beeinflussen. Wenn Sie z. B. mit weniger Speicher auskommen müssen, so bedeutet das für die Architektur vielleicht, mehr Subsysteme für Berechnungen als für die Datenhaltung zu schaffen. Dadurch enthalten Sie andere Arten von Teilung Ihrer Gesamtaufgabe.
- Effizienzanforderungen bezüglich Antwortzeiten sollten Sie auf Systemebene noch größtenteils zurückstellen. Wir werden uns beim Taskdesign im Kapitel 10 darüber explizit Gedanken machen. Lassen Sie sich bei der Subsystemfindung eher von anderen Faktoren leiten.
- Generell sollten Sie die Funktionalität so in Subsysteme aufteilen, dass getrennte Funktionalität potenziell auf unterschiedlichen Knoten laufen kann.

Für jedes Subsystem, das Sie finden, sollte eines klar ausdrückbar sein: Die Verantwortung, die es im Rahmen der Gesamtarchitektur übernimmt. Aus Sicht der Verantwortungsverteilung sollten Sie Subsysteme nicht nur aufgrund von nicht-funktionalen Anforderungen bilden, sondern auch die generellen Kategorien zu Hilfe nehmen, die wir Ihnen im Folgenden vorstellen.

Nicht-funktionale Anforderungen als Treiber für Subsystembildung

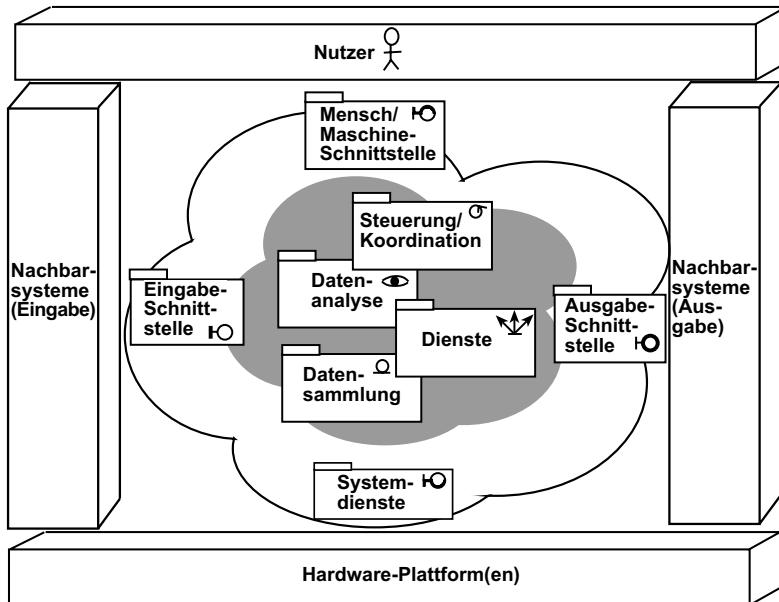
Klare Verantwortung der Subsysteme

## Vorschläge für Subsystem-Kategorien

Abbildung 6.2 zeigt innerhalb der Systemumgebung ein logisches Architekturmuster mit zwei Bereichen. In der Mitte sehen Sie (grau unterlegt) einige Kategorien von essenziellen, fachlichen, anwendungsbezogenen Subsystemen. Diese sind (im weißen Bereich) von vier technologieabhängigen Kategorien von Subsystemen umgeben. Letztere kapseln die Essenz vor den Einflüssen von Technologie und Umwelt.

Die beiden Arten von Kategorien unterscheiden sich nicht nur durch ihre unterschiedlichen Verantwortungsbereiche, sondern auch in Bezug auf ihre Robustheit, Stabilität, und Wiederverwendbarkeit. Deshalb schlagen wir Ihnen vor, diese Aspekte in getrennten Paketen zu modellieren – wenn keine wichtigeren Gründe andere Paketierungen erzwingen.

Vier essenzielle Subsystem-kategorien	<p>Die vier essenziellen Kategorien sind:</p> <ul style="list-style-type: none"><li>■ Steuerung und Koordination (Stereotyp: ): Derartige Subsysteme verantworten die Verteilung der Arbeit im Gesamtsystem und sorgen für die reibungslose Abarbeitung von Systemprozessen. Sie delegieren Arbeit an andere Subsysteme und überwachen Fristen, Termine, Ergebnisrückmeldungen. Sie sind oft zustandsgetrieben. Sie können hierarchisch aufgebaut sein: Gesamtsteuerung, Teilsystemsteuerung, Steuerung eines Einzelprozesses. Ein System muss keine Gesamtsteuerung haben, wenn es verteilt und asynchron auf externe Ereignisse reagieren kann.</li><li>■ Datensammlung (Stereotyp: ): Sie verantworten die Verwaltung und Speicherung wesentlicher Datenstrukturen des Systems, stellen den Zugriff darauf zur Verfügung, schützen diese Datenstrukturen vor unberechtigtem Zugriff und sorgen für inhaltliche Konsistenz.</li><li>■ Datenanalyse (Stereotyp: ): Sie verantworten die übergeordnete Auswertung, Verdichtung und Konolidierung von Daten, leiten querschnittliche Ergebnisse aus vielen Datensammlungen ab und stellen anwendungsbezogen gepackte Daten quer über Datensammlungen hinweg zur Verfügung.</li><li>■ Dienste (Stereotyp: ): Sie paketieren Dienstleistungen für andere, stellen querschnittlich gebrauchte Funktionen zur Verfügung und beauftragen im Normalfall keine anderen Subsysteme.</li></ul>
Vier technologie-abhängige Subsystem-kategorien	<p>Zu den vier technologieabhängigen Kategorien zählen:</p> <ul style="list-style-type: none"><li>■ Mensch-/Maschine-Schnittstelle (Stereotyp: ): Sie kapseln die Kommunikation des Systems mit menschlichen Nutzern.</li><li>■ Eingabeschnittstellen (Stereotyp: ): Sie kapseln alle Eingabegeräte, Sensoren und die Protokolle, die mit Eingabe-Nachbarsystemen vereinbart sind.</li><li>■ Ausgabeschnittstellen (Stereotyp: ): Sie kapseln alle Ausgabegeräte, Aktuatoren und die Schnittstellen zu den Ausgabe-Nachbarsystemen.</li><li>■ Systemdienste (Stereotyp: ): Sie abstrahieren von Basisdiensten der Hardware bzw. des Betriebssystems und stellen z. B. Basis-Kommunikationsdienste, Timing Services, Scheduling Services, ... zur Verfügung. Diese Subsysteme werden oft nicht selbst entwickelt, sondern gekauft oder wieder verwendet. Sie sollten als Schnittstellenlieferanten trotzdem im Architekturmödell modelliert werden.</li></ul>



**Abbildung 6.2:** Logisches Architekturmuster mit Kategorien von Subsystemen

Meist gelingt es bei RTE-Systemen nicht, die Subsysteme in den oben genannten Kategorien sortenrein zu halten. Trotzdem sind die Kategorien ein erster Ansatzpunkt für typische, kapselbare, getrennte Verantwortungsbereiche. Diskutieren Sie nicht stundenlang, ob ein Subsystem in die eine oder in die andere Kategorie gehört. Nehmen Sie die Kategorien einfach als Denkhilfe, um Aspekte, die aus Gründen des Information Hidings getrennt werden sollten, auch nach Möglichkeit zu trennen. Eine ausführliche Erläuterung der unterschiedlichen Eigenschaften dieser Subsysteme, sowie Hinweise, wie man sie findet und modelliert, finden Sie für die essenziellen Kategorien in Kapitel 9 und für die technologiebehafteten in Kapitel 10.

## Subsysteme beschreiben

Wenn Sie sich für ein Subsystem entschieden haben, beschreiben Sie es nach dem Muster in der folgenden Tabelle.

Muster für  
Subsystem-  
spezifikation

**Tabelle 6.7:** Muster für Subsystemspezifikationen

Name:	Bezeichnung für das Subsystem
Verantwortung:	Die Hauptverantwortung, die dieses Subsystem im Rahmen der Gesamtarchitektur wahrnimmt. (Eine Formulierung als Verantwortung statt als Funktionalität schärft Ihre Sinne dafür, dass jeder kleine und große Teil der Architektur tatsächlich für etwas gut ist. Wenn ein Subsystem für nichts verantwortlich ist, kann es auch entfallen.)

<b>Designbegründung:</b>	Nach welchen Kriterien wurde das Subsystem gebildet? (Füllen Sie diesen Punkt besonders sorgfältig aus, wenn die Kriterien nicht funktional waren!) Welche Alternativen hat man verworfen? Bei längeren Abwägungen wird hier nur auf ein separates Dokument verwiesen.
<b>Designrechtfertigung:</b>	Warum glauben Sie, dass dieses Subsystem das leistet, was seine Beschreibung verspricht? Wie werden durch dieses Subsystem die Anforderungen erfüllt?
<b>Nicht-funktionale Anforderungen:</b>	Alle Qualitätsanforderungen und Randbedingungen, die diesem Subsystem zugeordnet wurden.

Die in diesem Abschnitt vorgestellten Hinweise führen eher aus prinzipiellen Überlegungen zu Vorschlägen für Subsysteme: Aus Erfahrung, durch Randbedingungen und Qualitätsanforderungen oder durch Nachdenken über typische Kategorien. In Kapitel 9 und 10 werden wir dem einen Bottom-up-Weg gegenüberstellen, der ausgehend von den funktionalen Anforderungen versucht, das gleiche Ziel zu erreichen: Eine verantwortungsbewusste Aufteilung in Subsysteme.

Wir empfehlen Ihnen, in der Praxis beide Ansätze zu mischen und sich in der Mitte zu treffen. So können Sie einerseits frühzeitig mit Architekturüberlegungen beginnen, was gerade bei den technologielastigen RTE-Systemen unbedingt notwendig ist, andererseits jedoch systematisch aus funktionalen Anforderungen nur die Teile ableiten, die wirklich gebraucht werden. Der Top-down-Weg stellt eher langfristige Überlegungen bezüglich Wartbarkeit, Wiederverwendbarkeit, Kostenbewusstsein, Portierbarkeit und Standards in den Mittelpunkt der Überlegungen; der Bottom-up-Weg sorgt für die Erfüllung der konkreten Benutzerwünsche.

## 6.5 Produktarchitektur prüfen

Dynamisches Zusammenspiel der Komponenten prüfen

Eine Architekturentwicklung auf Papier mit Diagrammen und Beschreibungen geht nur dann gut, wenn Sie die gleiche Architektur schon mehrfach in der Vergangenheit eingesetzt haben. Die Diagramme drücken nur die statischen Aspekte der Architektur aus. Das dynamische Zusammenspiel der Komponenten ist anhand von Verteilungs- und Subsystemdiagrammen schwer zu überprüfen. Wenn Sie also neue Ideen für Hardware- und Softwarekomponenten in Ihrem Entwurf verwendet haben, sollten Sie die Architektur mit verschiedenen Mitteln absichern.

Dazu zählen Labormuster und Prototypen für verschiedene Aspekte der Architektur, z. B. zum Prüfen von Durchsatz oder Stabilität von Komponenten, zur Verifikation von Schnittstellen oder zur Absicherung bei gekauften oder fremd vergebenen Teilsystemen.

Das Mindeste, was Sie tun sollten, ist eine Schreibtischsimulation von typischen Abläufen in der Architektur. Die UML stellt dafür Szenarien in Form von Sequenz- oder Kollaborationsdiagrammen zur Verfügung.

## Kollaborationsdiagramme

Kollaborationsdiagramme stellen die Zusammenarbeit einer Menge von Objekten dar. Dazu zeichnet man die Objekte mit ihren Verbindungen (als Instanziierung eines Klassendiagramms). Objekte kommunizieren über Nachrichten. Dies wird in Form von kleinen Pfeilen entlang der Verbindungen ausgedrückt, die mit dem Namen der Nachricht beschriftet sind. Volle Pfeilspitzen kennzeichnen synchrone Nachrichten, einfache Pfeilspitzen stehen für asynchrone Kommunikation. Um die Reihenfolge des Ablaufs zu kennzeichnen, werden die Nachrichten von 1 bis n nummeriert. Geschachtelte Nummern (z. B. 3.1.3) deuten an, dass die Nachrichtenaufrufe geschachtelt sind. Ein Stern (\*) hinter der Nummer weist auf eine Iteration hin.

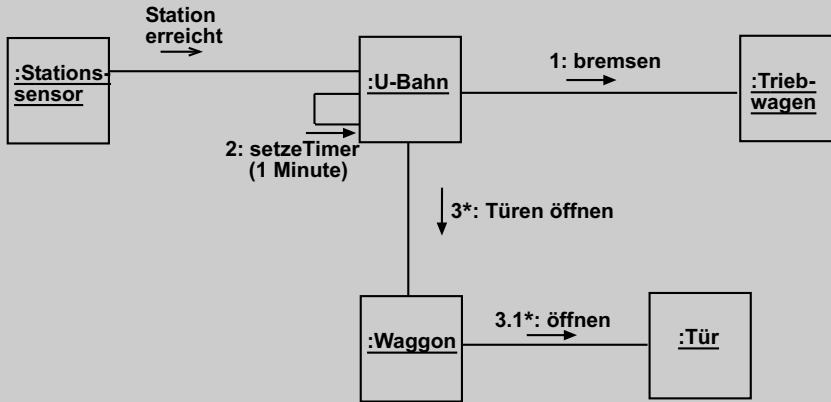


Abbildung 6.3: Beispiel für ein Kollaborationsdiagramm

Die auslösende Nachricht erhält keine Nummer, da man sonst alle weiteren Nachrichten mit 1.1, 1.2, ... kennzeichnen müsste und nie zu Nummer 2 käme.

Kollaborationsdiagramme und Sequenzdiagramme sind nur zwei Darstellungsmittel für den gleichen Zweck. Bei Sequenzdiagrammen denkt man eher an die Reihenfolge der Operationen. Bei Kollaborationsdiagrammen steht eher das Objekt mit seinen ein- und ausgehenden Nachrichten im Mittelpunkt des Denkprozesses. Die meisten Tools können ein Diagramm in die jeweilige andere Darstellung umwandeln und bieten Ihnen somit beide Gesichtspunkte an.

Nutzen Sie Kollaborationsdiagramme nicht nur auf der Ebene von einfachen Objekten. Erstellen Sie derartige Modelle mit Stellvertreterobjekten für ganze Hardwareknoten, um das Zusammenspiel zwischen Standorten oder zwischen Clients und Servern zu simulieren. Zeichnen Sie Kollaborationsdiagramme auch mit Stellvertreterobjekten für ganze Subsysteme zur Überprüfung des Zusammenspiels großer Black Boxes. Aber übertreiben Sie nicht mit diesen Diagrammarten, wenn andere Arten der Architekturprüfung vielversprechender, Kosten sparer und für die Projektbeteiligten akzeptabler sind.

## **6.6 Hatten Sie den Mut, die Systemarchitektur zu entscheiden?**

---

In diesem Kapitel haben wir über die Systemstrukturierung aus Sicht der Architektur diskutiert. Betrachten Sie Ihr bisher erreichtes Ergebnis kritisch:

- Haben Sie Ihre Architekturziele explizit gemacht, damit Sie Ihre Designentscheidungen danach treffen können?
- Ist es Ihnen gelungen, aus Anforderungen und Randbedingungen zu einer stabilen, allseits anerkannten Systemarchitektur zu kommen?
- Haben Sie dabei die Tradeoffs ausreichend dokumentiert, so dass die gleichen Diskussionen nicht bei der nächsten Version des Systems wieder in Frage gestellt werden?
- Haben Sie die Tragfähigkeit der Systemarchitektur (Zusammenspiel der Teile, Machbarkeit, Erfüllung Ihrer Architekturziele) ausreichend überprüft?
- Ist die Verantwortung der einzelnen Subsysteme so beschrieben, dass sie im Weiteren getrennt voneinander entwickelt oder extern beauftragt werden können?

„Die Idee ist da, in dir eingeschlossen.

Du mußt nur den überzähligen Stein entfernen.“

*Michelangelo (1475–1564)*

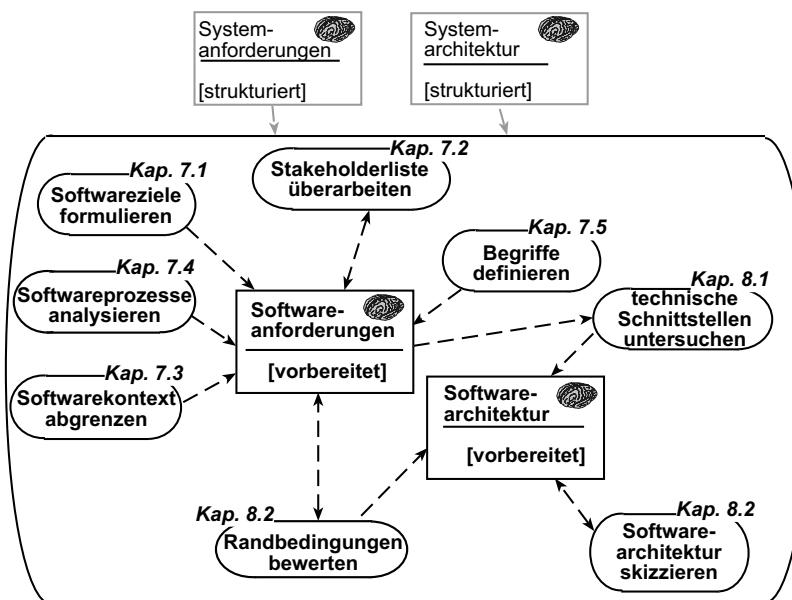
# Teil III

## Der Softwareentwicklungszyklus

Teil III führt uns in die Welt der Softwareentwicklung für RTE-Systeme. Wir betrachten schwerpunktmäßig die beiden entscheidenden Phasen der Softwarevorbereitung und der Softwarestrukturierung (vgl. Abschnitt 2.5).

Vorbereitung der Softwareanteile:  
mehr in Kapitel 7 und 8

Abbildung III.1 zeigt die wichtigsten Aktivitäten der Softwarevorbereitung. Diese Abbildung ist Ihr roter Faden durch die Kapitel 7 und 8.



**Abbildung III.1:** Die Aktivitäten und Ergebnisse der Softwarevorbereitung

### Teil III: Software-Entwicklungszyklus

Ziel der Vorbereitung ist es, die Anforderungen und Randbedingungen für Ihre Software aus den Systemvorgaben abzuleiten. Außerdem wollen wir die Softwareprozesse so weit gliedern, dass wir die nächste Phase – die Softwarestrukturierung – planen können.

Software-  
strukturierung:  
mehr in Kapitel 9  
und 10

Ziel der nächsten Phase ist es, aus allen Anforderungen und Randbedingungen eine fachliche und technische Softwarearchitektur zu entwickeln, die die Vorteile der agilen Softwareentwicklung glaubhaft demonstriert. Abbildung III.2 enthält die dafür notwendigen Schritte und Ergebnisse und ist Ihr roter Faden durch die Kapitel 9 und 10.

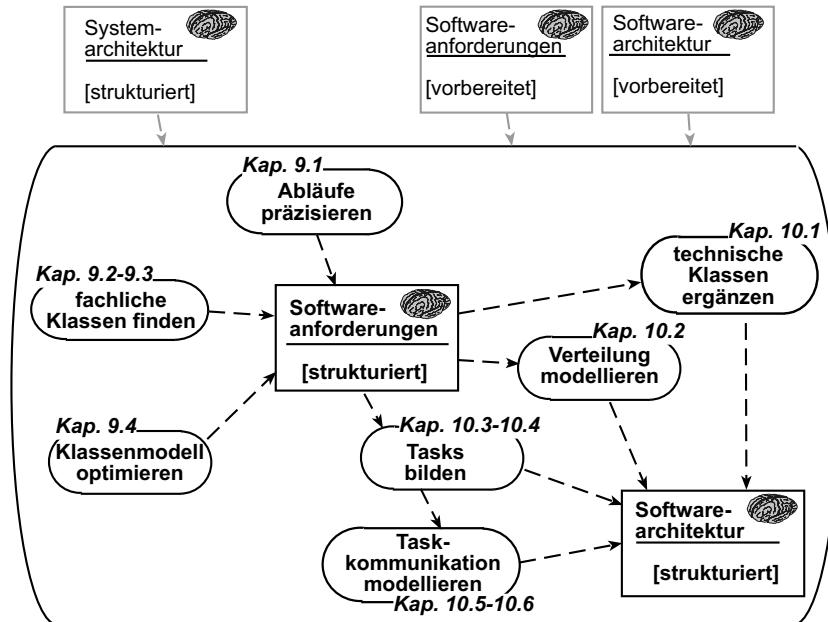


Abbildung III.2: Die Aktivitäten und Ergebnisse der Softwarestrukturierung

Die Softwareentwicklung wird Ihnen leichter von der Hand gehen, wenn die Modelle auf Systemebene gut vorbereitet wurden. Fehlt Ihnen diese Voraussetzung, so müssen Sie zurück zu den Aktivitäten, die wir in Teil II beschrieben haben.

„Aufklärung ist die Maxime, selber zu denken.“

*Immanuel Kant (1724–1804),  
deutscher Philosoph*

# 7

## **Erste Softwareanforderungen**

---

### **Fragen, die dieses Kapitel beantwortet:**

- Wie beginne ich mein Softwareprojekt?
- Was sind Softwareziele?
- Gibt es besondere Stakeholder für die Software?
- Wie unterscheidet sich der Softwarekontext vom Hardwarekontext?
- Software- & System-Use-Cases: Wo liegen die Unterschiede, wo die Gemeinsamkeiten?

In diesem Kapitel beschäftigen wir uns mit den ersten softwarespezifischen Aktivitäten. Sie werden Ihnen – sofern Sie Kapitel 3 studiert haben – alle bekannt vorkommen. Es sind die gleichen Tätigkeiten, die wir auf der Systemebene diskutiert und vorgeschlagen haben. Um Ihnen Wiederholungen zu ersparen, beschreiben wir im Folgenden nur aus der Softwaresicht begründete Besonderheiten und Unterschiede. Obwohl die Analogie da ist, sollten Sie agil vorgehen, d. h. die Schritte von der Systemebene nicht blind auf die Softwareebene übertragen. Fragen Sie sich stets, ob die Aufgaben und vor allem die Ergebnisse für Ihr Softwareprojekt relevant sind und Ihnen weiterhelfen.

Wenn Sie die in Kapitel 3 bis 6 vorgeschlagenen Aktivitäten für Ihr System sorgfältig durchgeführt haben, haben Sie es jetzt bedeutend einfacher. Sollten Sie diese Aktivitäten eher stiefmütterlich behandelt haben, müssen Sie jetzt mehr Arbeit leisten und die im Folgenden vorgestellten Aktivitäten präziser und vollständiger durchführen.

Ungünstige  
Hardware-/  
Software-  
aufteilung  
rückgängig  
machen!

Bei einem agilen Vorgehen können Erkenntnisse, die im Lauf der Softwareentwicklung gewonnen werden, auch bereits getroffene Hardware-/Softwareentscheidungen beeinflussen. Beispielsweise könnten Sie bei einer detaillierten Analyse eines Softwareprozesses feststellen, dass die geforderten Zeitanforderungen aufgrund der nun bekannten Komplexität kaum eingehalten werden können. Diese Softwarefunktionen sollten dann doch auf die Hardware verschoben werden. Verhandeln Sie umgehend mit dem Produkt-/Systemverantwortlichen, falls sich getroffene Entscheidungen bezüglich der Aufteilung in Hardware und Software als ungünstig erweisen. Eine Neuaufteilung erspart Ihnen viel Ärger und ein schwer wartbares Softwaredesign<sup>1</sup>.

### 7.1 Formulieren der Softwareziele

Softwareziele  
sind präziser als  
Systemziele

Für Ihr Softwareprojekt gibt es spezielle Ziele, die sich zum einen von den Gesamtsystemzielen ableiten, zum anderen aber auch neu ergeben. Während Sie am Anfang Ihrer Entwicklung mit dem Blick auf das Gesamtsystem oftmals nur visionäre und vage Ziele formulieren konnten, sind Sie nun in der Lage wesentlich detailliertere und prüfbarere Ziele für die Softwareentwicklung zu verfassen. Woran liegt das?

Zum einen ist die Gesamtbetrachtung wesentlich schwieriger: Sie beginnen mit weniger Wissen über die konkreten Stakeholderwünsche, haben meist nur ungefähre Vorgaben von Projekt- und Zeitrahmen, können sich auf weniger Erfahrungswerte stützen und sind mit Ihrer Projektmannschaft noch nicht vertraut. Zum anderen fehlen Ihnen wesentliche Grundlagen zum Schätzen, nämlich die groben Anforderungen und die ersten Architekturentscheidungen.

Die meisten dieser Punkte treffen zu Beginn des Softwareprojekts nicht mehr zu – vorausgesetzt Sie haben die Schritte von Kapitel 3 bis 6 umgesetzt. Jetzt sind Ihnen Umfang, Aufgaben und Randbedingungen der zu entwickelnden Software im Wesentlichen klar. Sie kennen Ihr Team und können zahlreiche Erkenntnisse aus den Aktivitäten auf der Systemebene einfließen lassen. Vorteilhaft ist dabei auch, dass sich Softwareprojekte in aller Regel in einem zeitlich und finanziell engeren Rahmen bewegen als eine System- oder Produktentwicklung. Auf dieser Basis lässt sich gut schätzen und erreichbare Ziele formulieren.

Wenn Sie Ziele für Ihr Softwareprojekt angeben, achten Sie auf Konsistenz und Widerspruchsfreiheit zu den Gesamtsystemzielen und möglichen Hardwarezielen. Durch die Aufteilung von Hard- und Software ergeben sich mitunter neue Ziele, die zunächst irrelevant waren. So müssen Sie nun zum Beispiel bei rasch wechselnder Hardware während der Projektlaufzeit Ziele für die Portabilität Ihrer Software vorgeben. Vielleicht ergeben sich auch andere neue Qualitätsansprüche an die Software.

<sup>1</sup> Und verschiebt die Probleme hoffentlich auf die Hardwareseite ☺.

## 7.2 Überarbeiten der Stakeholderliste

Zur Beginn der Softwareentwicklung sollten Sie die Stakeholderliste, die bei der Betrachtung des Gesamtsystems aufgebaut wurde, zur Hand nehmen und aus Softwaresicht nochmals überdenken.

Dabei müssen Sie die Hardwareentwickler als neue Stakeholder für den Softwareanteil des RTE-Systems hinzunehmen, da sie Anforderungen an die Schnittstellen der Software liefern. Weitere potenzielle neue Stakeholder sind Implementierungsspezialisten mit Fachwissen für spezielle Bestandteile der Software. Denken Sie dabei an Spezialisten für Speichermedien, wenn es sich um Systeme mit signifikanter Datenhaltung handelt, oder Betriebssystemexperten, die Antworten zum Tasking oder zur Interprozesskommunikation liefern können.

Es gibt neue Stakeholder!

Neben den neuen Stakeholdern gibt es auch zahlreiche, die weniger relevant werden. Nämlich jene, deren Interessen sich nur auf den Hardwareteil des RTE-Systems beziehen. Zum Beispiel müssen Sie sich keine Gedanken mehr um die Entsorgung des Produkts oder Systems machen, da Software sich in der Regel umweltschonend entsorgen lässt. Auch Normen oder Gesetze, die ausschließlich die Hardwareanteile betreffen, können Sie getrost von der Liste nehmen. Aber Vorsicht, mitunter liegen versteckte Abhängigkeiten vor. So können beispielsweise Richtlinien zur Elektromagnetischen Verträglichkeit (EMV), z. B. zur Abschirmung gegen Strahlung und Elektrosmog, die ausschließlich die Hardware betreffen, sich aber trotzdem auf die Software auswirken. Durch einen uneffizienten Algorithmus in der Software könnte z. B. der Prozessor ständig an seiner Leistungsgrenze arbeiten und dementsprechend abstrahlen. In dieser Situation müssten Sie wieder auf die für die Hardware zuständigen Stakeholder zurückkommen.

## 7.3 Abgrenzen des Softwarekontexts

Um den Rahmen Ihrer Softwareentwicklung abzustecken, müssen Sie den Kontext der Software basierend auf den Ergebnissen von Kapitel 6 näher spezifizieren. Ausgehend von der Aufteilung des RTE-Systems in Hard- und Software empfehlen wir, die Nachbarn des Softwareanteils und ihre logische Kommunikation mit der Software in Kontextdiagrammen zu dokumentieren. Unter Umständen wurden bereits Vorgaben für wesentliche Komponenten des Softwaresystems formuliert und zum Beispiel das Betriebssystem, Treiberschnittstellen oder die Technologie der Datenhaltung festgelegt. Diese Randbedingungen sollten Sie nun berücksichtigen.

Konzentration auf die Software

### 7.3.1 Welche Nachbarn hat meine Software?

Alles, was ich nicht programmieren muss!

Die zu erstellende Software wird im Software-Kontextdiagramm als Black Box betrachtet. Die Nachbarn sind die Komponenten, mit denen Ihre Software kommuniziert oder auf denen sie läuft. Das heißt, die Hard- und Software, die Sie **nicht** im Rahmen des Softwareprojekts selbst entwickeln, die aber direkt mit Ihrer Software in Verbindung steht. Das können Zukaufkomponenten, Hardwaretreiber, Bibliotheken oder weitere existierende Softwarekomponenten sein. Bei RTE-Systemen mit parallelen Prozessen ist wahrscheinlich die Systemaufrufsschnittstelle des unterlagerten Betriebssystems als Nachbar zu betrachten.

Kommuniziert Ihre Software allerdings direkt mit der Hardware, so ist die Hardware im Rahmen der Kontextabgrenzung als Nachbar der Software zu modellieren. Typische Fälle sind hier Softwareentwicklungsprojekte, die Treiber oder Assemblerprogramme selbst entwickeln und die dann den Prozessor bzw. Speicher aus Optimierungsgründen direkt ansprechen.

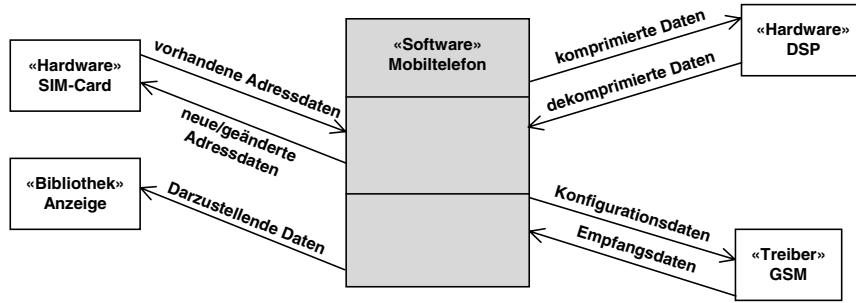
Sie sollten den Kontextrahmen möglichst eng halten. Ein Argument dafür sind die anfallenden Softwaretests. Alle Software, die Sie entwickeln, muss irgendwann einmal getestet werden. Die Testfallgrenze ist dabei mit der Softwarekontextgrenze identisch. Anbindungen, Treiber und Bibliotheken sollen von denen getestet werden, die sie entwickeln. Nicht von Ihnen als Nutzer der Module und Dienstleistungen. Erst im Rahmen der Systemintegration testen Sie dann das Zusammenspiel zwischen der von Ihnen entwickelten Software, den Zukaufkomponenten und der entwickelten Hardware.

Keine menschlichen Akteure

Im Gegensatz zum Systemkontext treten hier menschliche Akteure nicht auf, da die Software nie direkt mit ihnen kommuniziert. Allenfalls müssen Sie die Geräte der Mensch-Maschine-Schnittstelle berücksichtigen, sofern das Betriebssystem Ihnen keine geeigneten Treiber zur Verfügung stellt.

### 7.3.2 Darstellen des Kontexts

In Abschnitt 3.3 haben wir Ihnen Use-Case-, Klassen- und Sequenzdiagramme als mögliche Notationsformen für die System-Kontextdiagramme vorgeschlagen. Diese können Sie analog für das Softwareumfeld einsetzen – mit allen Detaillierungsmöglichkeiten, Vor- und Nachteilen. Abbildung 7.1 zeigt dies exemplarisch in Form eines Klassen-Kontextdiagramms für ein Mobiltelefon.



**Abbildung 7.1:** Software-Kontextabgrenzung mittels Klassendiagramm

Sie können den Detaillierungsgrad in den Diagrammen erhöhen, da Sie Ihren Blickwinkel auf die Software konzentrieren und damit nicht mehr der Komplexität des Gesamtsystems gegenüberstehen.

## 7.4 Softwareprozesse analysieren

Nun ist es an der Zeit, sich detaillierter mit der Analyse der Softwareprozesse als Teil der bereits identifizierten Systemprozesse zu beschäftigen. Grundsätzlich kann zur Ermittlung der Softwareprozesse das gleiche Vorgehen angewandt werden, das in Abschnitt 3.4 für das Auffinden der Systemprozesse beschrieben wurde. Allerdings hat sich der Detaillierungsgrad, unter dem wir das RTE-System betrachten, erhöht.

### 7.4.1 Softwareprozesse in Systemprozessen

Die bereits gefundenen Systemprozesse einfach als Softwareprozesse zu übernehmen, wäre sicherlich nahe liegend. Aber die Systemprozesse sind für den aktuellen Stand der Entwicklung des RTE-Systems vielleicht noch zu grob, als dass man sie unreflektiert weiterverwenden könnte. Außerdem beziehen sich viele der Systemprozesse auf die Hardware des Systems, die für die Softwareprozesse völlig uninteressant sind. Systemprozesse, die vollständig in Hardware ablaufen, können daher getrost weggelassen werden, da sie keinen Einfluss auf die hier analysierten Softwareprozesse haben.

Oft werden nur kleine Bestandteile von Systemprozessen zu Softwareprozessen. Meist haben diese dann auch andere Akteure als die zugrunde liegenden Systemprozesse. Nehmen wir beispielsweise das Ausgeben einer CD bei einem CD-Player. Nach der Registrierung des Tastendrucks auf die Eject-Taste wird lediglich ein Interrupt ausgelöst. Der Rest des Prozesses hat keinen Bezug mehr zur Software. Der Interrupt steuert einen Motor an, der dann das

Extrakt des System-  
prozesses

Softwareprozess  
als Extrakt des System-  
prozesses

CD-Laufwerk öffnet. Sind die Tasten eines CD-Players allerdings mehrfach belegt, so ist der Softwareprozess komplexer. Die Software wertet hier z. B. aus, wie oft eine Taste gedrückt wurde, in welchem Menü sich das System befand und auf welcher Ebene.

Zur Ermittlung der Softwareprozesse aus den Systemprozessen müssen Sie versuchen, die Beteiligung der Software in den Systemprozessen zu erkennen, und alle Schritte der Systemprozesse aufdecken, an denen Software beteiligt ist. Dies machen Sie am besten anhand der vorliegenden Systemprozess-Beschreibung. Auch die Szenarien, die mit diesem Systemprozess zu tun haben, sollten Sie auf der Basis der auf der Systemebene erstellten Prosaszenarien und Sequenzdiagramme nach dem Einfluss von Software durchsuchen. Aus den nun inhaltlich zusammenhängenden Schritten der Systemprozess-Beschreibung und Szenarien (ohne Unterbrechungen durch Hardware, Mechanik oder Elektronik) erstellen Sie die neuen Softwareprozesse. Tragen Sie diese zunächst in ein Use-Case-Diagramm ein.

### Use-Case-Beziehungen

Die UML sieht drei Beschreibungsmittel vor, um Use-Cases zu strukturieren und Abhängigkeiten zwischen Use-Cases auszudrücken: include-, extend- und Generalisierungsbeziehungen.

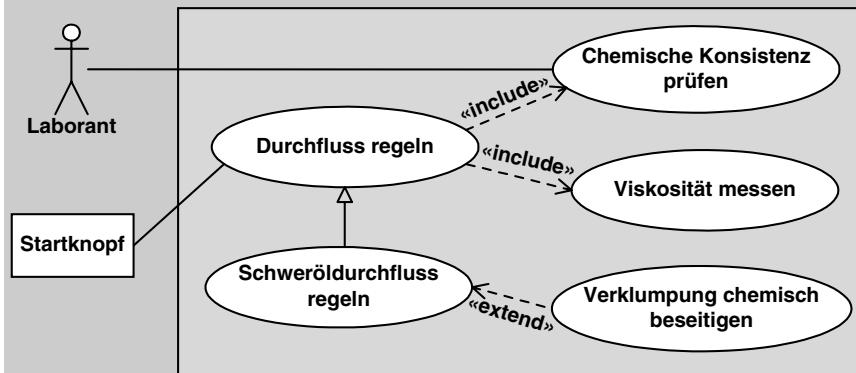


Abbildung 7.2: Varianten von Use-Case-Beziehungen

Eine *include*-Beziehung vom Use-Case „Durchfluss regeln“ zum Use-Case „Viskosität messen“ besagt, dass der Use-Case „Viskosität messen“ in den Use-Case „Durchfluss regeln“ **immer eingefügt wird**.

Eine *extend*-Beziehung von Use-Case „Schweröldurchfluss regeln“ nach Use-Case „Verklumpung chemisch beseitigen“ besagt, dass der Use-Case „Verklumpung chemisch beseitigen“ in den Use-Case „Schweröldurchfluss regeln“ **eingefügt werden kann**. Dazu wird im Rahmen des Use-Cases „Schweröldurchfluss regeln“ am sogenannten Extension-Point eine Bedingung überprüft,

die angibt, ob der Use-Case „Verklumpung chemisch beseitigen“ durchlaufen wird oder nicht. Tritt die Bedingung nicht ein, wird der Use-Case „Schweröldurchfluss regeln“ direkt weiter ausgeführt.

Include- und extend-Beziehungen werden mit einem gestrichelten, gerichteten Pfeil notiert, der mit dem Schlüsselwort «include» bzw. «extend» beschriftet wird.

Die dritte Form von Beziehungen unter Anwendungsfällen ist die Generalisierung. Im Fall einer Generalisierung erben die spezialisierten Anwendungsfälle (hier „Schweröldurchfluss regeln“) die Kommunikationsbeziehungen und das gesamte Verhalten des Basis-Use-Cases (hier „Durchfluss regeln“), welches ergänzt und überschrieben werden kann. Dargestellt werden Generalisierungen äquivalent zum Klassendiagramm mit einem Pfeil, der vom erbenden auf den vererbenden Anwendungsfall gerichtet und dessen Spitze nicht ausgefüllt ist.

Die UML lässt auch eine Generalisierungsbeziehung zwischen Akteuren zu.

Wahrscheinlich haben Sie im Rahmen der Systembetrachtung so viel Wissen angesammelt, dass Sie die Softwareprozesse sehr präzise beschreiben und Abhängigkeiten zwischen den Prozessen herausarbeiten können. Die UML unterstützt Sie dabei mit zusätzlichen Notationselementen, die in Use-Case-Diagrammen eingesetzt werden dürfen.

Da gerade bei diesen Aktivitäten sehr viele Stakeholder aus dem technischen, insbesondere aus dem informationstheoretischen Umfeld kommen, ist es durchaus legitim, etwas speziellere UML-Elemente – vor denen vielleicht Ihr Anwender die Flucht ergreifen würde – zu benutzen. Verwenden Sie

- «include» für wiederverwendbare Teilsysteme,
- «extend» für selten auftretende Sonderfälle und fachliche Ausnahmen,
- «generalize» für sehr abstrakte Softwarebestandteile, wie z. B. Frameworks mit unterschiedlichen Clients.

Bitte investieren Sie hier nicht viel Zeit für Diskussionen, ob das Use-Case-Diagramm mittels extend-, include- oder Generalisierungsbeziehungen optimiert werden kann. Dies sind Designentscheidungen, die Sie gerne integrieren können, wenn sie offensichtlich sind, die sie sonst aber auch später treffen können.

Maßvolle  
Anwendung der  
UML

## 7.5 Begriffe definieren

Bei der Entwicklung des Softwareanteils des RTE-Systems tauchen zusätzliche Begriffe auf, die Sie ebenfalls dokumentieren sollten.

Da wir bereits ein erstes Klassendiagramm besitzen, können Sie die Begriffsdefinitionen nun in dieses Modell integrieren. So haben Sie eine Stelle in Ihrem Requirementsgehirn, an der die Informationen zentral, leicht zugänglich

Integration der  
Begriffe ins  
Klassendiagramm

und konsistent aufbewahrt werden. Aus den Modellen können Sie (z. B. mit Hilfe guter Modellierungswerkzeuge) tagesaktuell beliebige Zusammenstellungen von Begriffsdefinitionen extrahieren und so für unterschiedliche Stakeholder angepasste Glossare generieren.

## **7.6 Wissen Sie, was Ihre Software leisten muss?**

---

In diesem Kapitel haben wir die Softwarevorbereitung aus Sicht der Anforderungsaktivitäten diskutiert. Betrachten Sie Ihr bisher erreichtes Ergebnis kritisch:

- Kennen Sie das Zusammenspiel Ihrer Software mit der Hardware und den fertigen Softwarekomponenten?
- Mit welchen Softwarezielen sehen Sie sich konfrontiert?
- Haben Sie die zusätzlichen Stakeholder identifiziert, die die Anforderungen für Ihre Software liefern?
- Kennen Sie die Grenzen (Schnittstellen) Ihrer Software?
- Haben Sie Schwierigkeiten mit den Vorgaben für die Software mit den Systemverantwortlichen abgeklärt und verhandelt?

„Wer nicht hart genug ist, dem Leben seine Bedingungen aufzuprägen, der muß die Bedingungen hinnehmen, die es ihm bietet.“

*George Eliot (1819–1880), englische Erzählerin  
(Pseudonym für Mary Ann Evans)*

# 8

## Software-Randbedingungen durchdringen

---

### **Fragen, die dieses Kapitel beantwortet:**

- Wie setzt man Systemvorgaben in Software-Randbedingungen um?
- Was hilft bei zu vielen Randbedingungen?
- Wie modelliert man die physikalische Umgebung der Software?

Als Softwareentwickler für RTE-Systeme haben Sie meist ungleich mehr Randbedingungen als Ihre Kollegen in Banken und Versicherungen. Diese frühzeitig gut zu durchdringen ist ein entscheidender Schritt auf dem Weg zum Projekterfolg.

Betrachten wir Ihre Ausgangssituation als Softwaredesigner, die in Abbildung 8.1 dargestellt ist. Idealerweise sollte das Requirementsgehirn Ihres Projektes bereits neben den Zielen für die Software auch den logischen Kontext beinhalten. Alle wesentlichen Nachbarsysteme sollten identifiziert sein, und Sie sollten die Informationen, die Ihre Software erhält und liefern muss, kennen. Außerdem ist Ihre Software hoffentlich bereits als Komponente in die gesamte Systemlandschaft eingeordnet.

Ziele der nächsten Schritte sind nun, diese Vorgaben und Randbedingungen gut zu verstehen, die Schnittstellen zu den Nachbarsystemen vor Augen zu haben und zu wissen, mit wem Sie noch verhandeln können (oder müssen) und wo starre Vorgaben einzuhalten sind. Außerdem werden Sie offensichtliche Strukturvorgaben sofort in Modellskizzen für die Softwarearchitektur umsetzen.

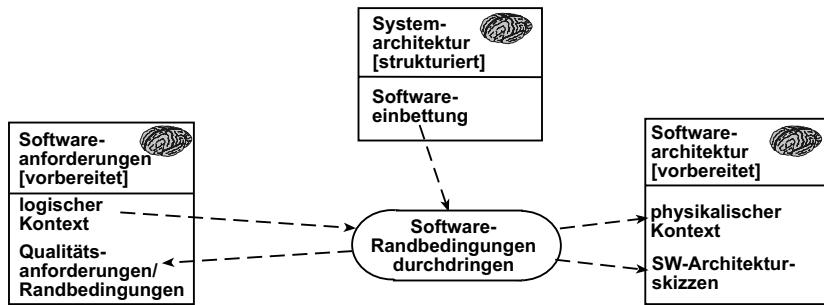


Abbildung 8.1: Architekturvorbereitung im Kontext

## 8.1 Schnittstellen zur Umgebung

Wie schon auf der Systemebene sollten Sie auch für Ihre Software die Technologie an den Schnittstellen zur Umgebung in Form eines physikalischen Kontextdiagramms modellieren und die Kanäle spezifizieren. Klären Sie ausgehend von jeder Eingabe und Ausgabe im logischen Kontext detailliert ab, über welches physikalische Medium die Software die Eingaben bekommt und die Ausgaben verteilt.

Ihre Zielsetzung lautete zum Beispiel, die Software für einen Analysecomputer zu erstellen, der Wasserproben auf Verunreinigungen durch Schadstoffe untersucht<sup>1</sup>. Die Software wird durch Kommandos des Operators gesteuert. Sie erhält Spektralanalysedaten zur Auswertung von einem Spektrometer. Die Proben werden über aufgeklebten Barcode identifiziert. Die Hauptaufgaben Ihrer Software sind die Aufbereitung und Verdichtung der Spektraldaten und die Weiterleitung an ein Auswertungssystem sowie die Weitergabe an die zentrale Forschungsstelle.

Eine einfache Tabelle hilft Ihnen, die Informationen systematisch zu hinterfragen. Tragen Sie für jedes Nachbarsystem alle Ein- und Ausgaben ein und entscheiden Sie für jede einzelne über die zu verwendende Technologie. In unserem Beispiel gehen wir von folgenden Entscheidungen aus: Wir verwenden zur Kommunikation mit dem Operator eine Standardtastatur und einen Bildschirm. Interessanter ist die Diskussion der Schnittstelle zum Spektrometer. Der Hersteller könnte mehrere Schnittstellen anbieten. Wir entscheiden uns in der ersten Version für die Infrarotübertragung. Diese werden wir softwaremäßig natürlich kapseln, um bei Technologieänderungen flexibel zu sein. Aber wir müssen schließlich wissen, welches Schnittstellenmodul wir jetzt konkret entwickeln müssen. Der Barcodeleser lässt sich problemlos an den internen

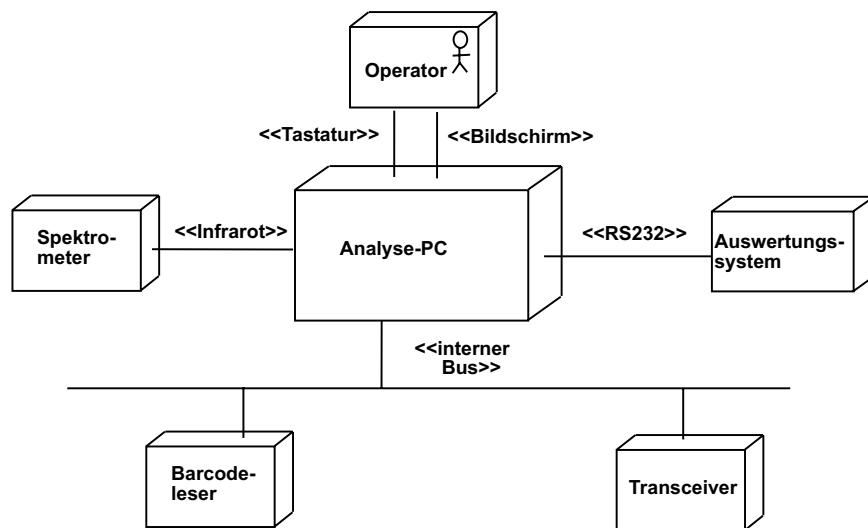
<sup>1</sup> Eine ausführliche Problembeschreibung des Gesamtsystems und einen Lösungsansatz mit der PSARE-Methode finden Sie in [HHP00].

Bus anschließen. Treiber dafür sind im Produkt enthalten. Die Analyseergebnisse werden in zwei Formen weitergegeben: Über eine RS232-Schnittstelle kommunizieren wir mit dem zugekauften Auswertungssystem und an die Forschungsstelle liefern wir, wie verlangt, per Funk. Dazu muss unsere Software einen Transceiver versorgen. Dies haben wir im letzten Projekt schon gemacht und kennen daher das Protokoll, das wir über den internen Bus absetzen müssen.

**Tabelle 8.1:** logische Ein-/Ausgabe und die Medien

Nachbarsystem	Ein-/Ausgabe	Medium
Operator	Kommandos	Tastatur
	Hinweise und Meldungen	Bildschirm
Spektrometer	Spektraldata	Infrarot
Barcodeleser	Barcode der Proben	Interner Bus
Auswertesystem		RS232
Zentrale	Analyseergebnisse	Per Funk. Der Transceiver hängt an dem internen Bus des PC

Wenn Sie alle Technologieschnittstellen vollständig aufgesammelt bzw. entschieden haben, können Sie sie zum besseren Überblick in ein Verteilungsdiagramm umsetzen. Abbildung 8.2 zeigt die oben getroffenen Annahmen.



**Abbildung 8.2:** Physikalischer Kontext eines Analysecomputers

Vergessen Sie nicht, zu dem Diagramm Knoten- und Verbindungsspezifikationen nach den Mustern im Kapitel 6 anzulegen.

## **8.2 Nicht-funktionale Anforderungen bewerten**

Wenn Sie als Softwarearchitekt eine gut spezifizierte Gesamtsystemarchitektur zur Verfügung gestellt bekommen, sind Ihrer Softwarekomponente hoffentlich bereits alle relevanten Qualitätsanforderungen und Randbedingungen zugeordnet worden. Wenn nicht, dann führen Sie bitte die Schritte zur Präzisierung der nicht-funktionalen Anforderungen aus, die wir in Abschnitt 6.2 ausführlich beschrieben haben.

Mit Blick auf Ihre Software sollten Sie die einzelnen Kategorien von nicht-funktionalen Anforderungen nochmals kritisch hinterfragen:

- Prüfen Sie Softwaretechnologievorgaben, z. B. bezüglich vorgeschriebener Programmiersprachen und Betriebssysteme, Einsatz von Bibliotheken und Standardkomponenten etc. Wie zwingend sind die Gründe für diese Randbedingungen wirklich? Wer hat es verlangt? Können Sie mit diesen Stakeholdern verhandeln, wenn Sie effizientere Lösungswege vorschlagen?
- Modellieren Sie Vorgaben über Prozessoren und einzusetzende Komponenten direkt als Verteilungs- bzw. als Paketdiagramm, wie in Kapitel 4 erläutert. Diese Modellskizzen bilden den Ausgangspunkt für die Weiterentwicklung der fachlichen und technischen Architektur.
- Prüfen Sie, ob das Vorgehen für das Gesamtsystem auch für das Softwarteilsystem zwingend einzuhalten ist. Viele Branchen legen für die Produktentwicklung langjährig erprobte Vorgehensweisen fest, sind aber durchaus bereit, für die Softwareteile moderne, agile Verfahren zuzulassen.

Hitparade  
überprüfen

Wenn Sie alle nicht-funktionalen Anforderungen überprüft und ergänzt haben, sollten Sie den Prioritätsvorgaben für die Architektur (vgl. Abschnitt 6.2.3) nochmals Ihre Aufmerksamkeit schenken. Für die Software gelten unter Umständen andere Prioritäten als für das Gesamtsystem. Schreiben Sie die Softwarearchitekturenprioritäten explizit auf und stellen Sie sicher, dass diese mit den Prioritäten auf Systemebene harmonisiert sind.

## **8.3 Kennen Sie Ihren Verhandlungsspielraum?**

In diesem Kapitel haben wir Sie auf die rechtzeitige Fokussierung auf Technologie und Randbedingungen hingewiesen. Prüfen Sie Ihr bisheriges Ergebnis:

- Haben Sie Ihren Verhandlungsspielraum erkundet und ausgeschöpft?
- Kennen Sie die Hitparade der Architekturenprioritäten für Ihre Software?
- Kennen Sie die technologische Einbettung Ihres Softwaresystems?

„Um die Dinge ganz zu kennen,  
muß man um ihre Einzelheiten wissen.“

*François VI. Duc de La Rochefoucauld  
(1613–1680), französischer Offizier,  
Diplomat und Schriftsteller*

# 9

## Fachliche Softwarearchitektur

---

### Fragen, die dieses Kapitel beantwortet:

- Softwareprozesse gefunden! Und nun?
- Nach so viel Vorarbeit – wie komme ich jetzt zu meinen Klassen?
- Wie muss ich meine fachliche Architektur dokumentieren?
- Welche Wege gibt es zur fachlichen Architektur?
- Wie lassen sich das Verhalten und die Abläufe meiner Software präzise und angemessen modellieren?

Objektorientierte Software besteht aus Klassen, durch deren Interaktion Ihre Problemstellung gelöst wird. In den Kapiteln 5 und 7 haben wir erläutert, wie Sie Entity-Klassen und Softwareprozesse finden und beschreiben. Bei der Erstellung der fachlichen Softwarearchitektur müssen Sie jetzt die Brücke zu einem integrierten Klassenmodell schlagen.

Dieses Kapitel beschreibt die wesentlichen Schritte zur Umsetzung der Anforderungen in die fachliche Architektur. Jetzt integrieren Sie Entity-Klassen und Prozesse systematisch in ein einheitliches Klassenmodell und präzisieren so mit Ihre Vorstellung von der zu entwickelnden Software. Sie dokumentieren damit das angesammelte Wissen konsistent im Requirementsgehirn und schaffen die Basis für die Implementierung (oder Generierung) Ihres Softwaresystems.

Die meisten Aktivitäten, die wir hier durchlaufen, kennen Sie bereits aus Kapitel 5 und Kapitel 7. Jetzt haben wir natürlich nicht mehr unsere Systemprozesse im Fokus, sondern den Softwareanteil des Systems und dessen Prozesse. Führen Sie daher folgende Schritte durch:

- Beschreiben Sie Ihre Softwareprozesse nach dem Muster von Kapitel 7 mit essenziellen Softwareprozess-Beschreibungen. Sollten Sie dabei bereits Aussagen zur Technologie erhalten, nehmen Sie diese ebenfalls mit auf, kennzeichnen sie aber.
- Modellieren Sie die Abläufe der Softwareprozesse in Form von Aktivitätsdiagrammen oder Zustandsautomaten (je nach Typ des Prozesses).
- Entwerfen Sie ein erstes Klassendiagramm für Ihre Software, indem Sie Entity-Klassen sammeln.

Um die Tätigkeiten dieses Kapitels durchführen zu können, müssen Sie diese Vorarbeiten geleistet haben. Die Ergebnisse nutzen wir für die Bildung des integrierten Software-Klassenmodells.

## 9.1 Abläufe präzisieren

Hier zeigen wir Ihnen noch ein paar Ergänzungen zum Umgang mit Aktivitätsdiagrammen auf der Softwareprozessebene und zwar die intensive Nutzung der Aktivitätsdiagramme in ihrer erweiterten Form als Objektflussdiagramm.

Objektfluss  
modellieren

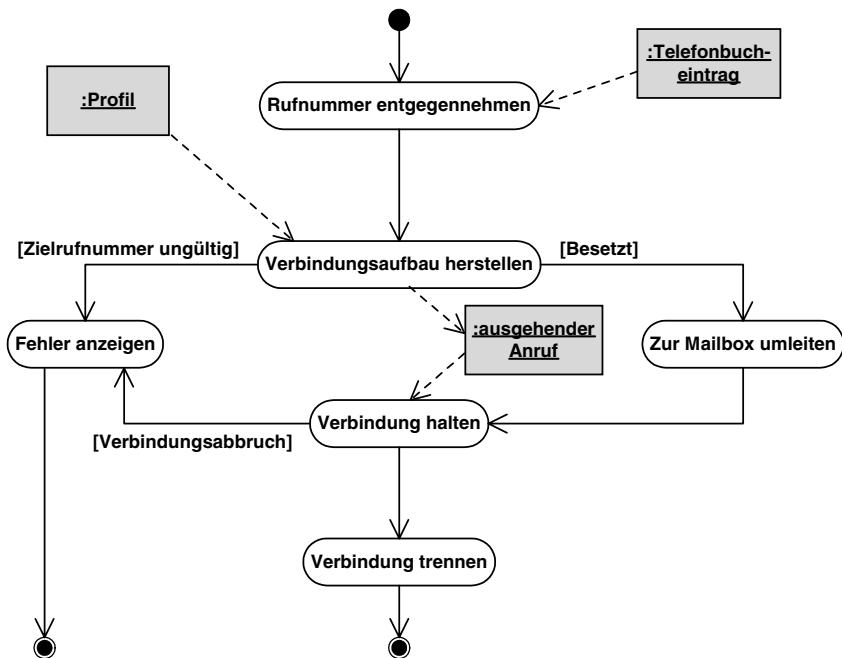
Im Vergleich zum einfachen Aktivitätsdiagramm müssen Sie beim Objektflussdiagramm zusätzlich die Objekte analysieren, die während des Ablaufs eine Rolle spielen.

### Vorgehen

Die ersten Schritte bei der Erstellung des Objektflussdiagramms gestalten Sie, wie die auf Systemprozessebene. Sie müssen, wie in Abschnitt 5.2 beschrieben, die notwendigen Aktivitäten und die zugehörigen Steuerflüsse identifizieren und modellieren – diesmal auf Ihr Softwaresystem bezogen. Da zu diesem Zeitpunkt im Projekt bereits Lösungen beschlossen wurden und Sie aller Wahrscheinlichkeit nach sehr viel über den Softwareprozess und seine Randbedingungen erfahren haben, empfehlen wir Ihnen, dieses pragmatische Wissen in das Diagramm einfließen zu lassen – viel stärker noch, als dies beim Aktivitätsdiagramm auf Systemprozessebene schon geschehen ist. Haben Sie also Mut, bereits gefallte Entscheidungen auf dieser Ebene in Ihre Modelle einfließen zu lassen, auch wenn sie vielleicht nach der reinen *Lehre der Analyse* dort nicht hingehören.

Zur Ausführung einer Aktivität benötigen Sie Daten oder erzeugen Informationen, die der Nachwelt erhalten bleiben sollen. Diese sollten in Form von Entity-Objekten in das Diagramm übernommen werden. Um diese „Wissensspeicher“ zu modellieren, ist es hilfreich,

- alle Transitionen zu untersuchen, und jeweils zu fragen, welche Daten hier benötigt, verändert oder erzeugt werden,
- das erstellte Glossar nach Entity-Klassen zu durchforsten,
- oder die Objekte mit signifikanter Datenhaltung aus den bereits vorliegenden Entity-Klassendiagrammen zu übernehmen.



**Abbildung 9.1:** Beispiel für ein Objektflussdiagramm

Sie müssen nicht bei jeder Aktivität alle Ein- und Ausgaben modellieren. Nutzen Sie das Objektflussdiagramm eher dazu, durch die Analyse der Aktivitäten neue Entity-Klassen zu entdecken oder Attribute zu bestehenden Klassen zu ergänzen. Nutzen Sie auch für Objektflussdiagramme die Möglichkeit der hierarchischen Zerlegung, wenn es der Übersichtlichkeit und Komplexitätsreduzierung hilft.

Sollten zwei Aktivitäten durch ein Entity-Objekt verbunden sein, so zwingt Sie die UML, den zwischen den beiden Aktivitäten vorhandenen Steuerfluss zu entfernen. Damit sind sie zwar UML-konform, der dargestellte Sachverhalt entspricht aber evtl. nicht mehr Ihrer fachlichen Realität, in der eine Aktivität

Entity-Objekte finden

zwar Informationen erzeugt, die von einer nachfolgenden Aktivität weiterverarbeitet werden, die Steuerung aber trotzdem nicht über den Datenfluss erfolgt. Ignorieren Sie diese Schwachstelle der UML und verwenden Sie Steuer- und Datenfluss, wie es der fachlichen Realität entspricht.

## 9.2 Kategorisierung von Klassen

Vier Arten  
fachlicher  
Klassen

Bisher haben wir uns vor allem mit Entity-Klassen beschäftigt. Dies sind aber nicht die einzigen Klassen einer fachlichen Architektur. Wir stellen Ihnen im Folgenden drei weitere Arten von Klassen vor und geben Ihnen für alle vier Arten Empfehlungen, wie Sie sie finden, welche Eigenschaften sie haben und wie Sie sie modellieren sollten. Wir unterscheiden

- Entity-Klassen (Stereotyp: ),
- Sichten-Klassen (Stereotyp: ),
- Steuerungs-Klassen (Stereotyp:  ) und
- Service-Klassen (Stereotyp:  ),

die wir jeweils durch Stereotypisierung kenntlich machen. Für jede Kategorie können wir typische Aufgabenbereiche, Attribute und Operationen nennen.

Pro Kategorie liefern wir Aussagen über die typische Lebensdauer von Objekten, die Komplexität ihres Verhaltens und typische Beziehungen, die Objekte dieser Klassen eingehen. Diese Kategorisierung ist nicht zwingend, sie hilft Ihnen aber,

- ein besseres Verständnis Ihrer Facharchitektur zu erlangen, da Sie sie nach der Fachlogik strukturieren können,
- den Klassen typische Eigenschaften zuzuordnen und diese zu überprüfen,
- Modellierungsziele auf Klassenebene präzise anzugeben,
- die Komplexität der Klassen zu reduzieren und damit ihre Wartbarkeit zu vereinfachen,
- die Basis für eine mögliche Subsystembildung zu schaffen und
- den Umbau des Systems zu erleichtern, da sich bestimmte Klassenarten besonders leicht wiederverwenden lassen.

### 9.2.1 Entity-Klassen

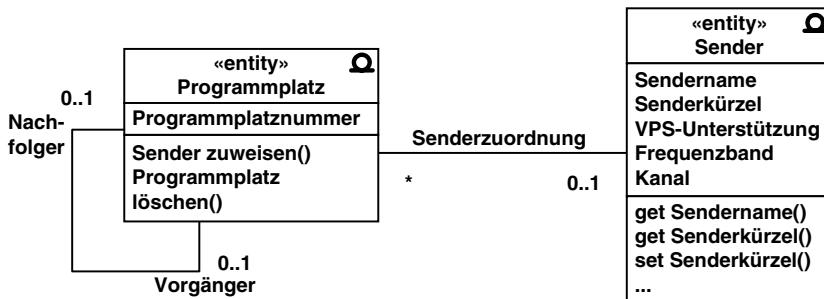
Entity-Klassen  
halten die Daten

Entity-Klassen kennen Sie bereits aus Kapitel 5. Sie kapseln die Daten Ihrer Software und die Zugriffe auf die Daten. Charakteristisch für diese Klassenart ist ihre hohe Zahl an Attributen, auf die viele primitive Operationen (get- und set-Operationen) zugreifen. Entity-Klassen enthalten nicht viele komplexe

Operationen, die – wenn vorhanden – meist Aufgaben der Datenverwaltung, wie z. B. die interne Konsistenzsicherung oder periodische Prüfungen, übernehmen.

Entity-Objekte sind sehr langlebig. Sie werden dauerhaft (persistent) auf einem Hintergrundspeicher angelegt, aus Geschwindigkeitsgründen dynamisch auf dem Heap. Ihre Lebenszeit überdauert meist die der Softwareprozesse, von denen sie verwendet werden.

Falls Entity-Klassen Zustandsautomaten enthalten, beschreiben diese meist die Lebensgeschichte der Objekte dieser Klasse von der Erstellung bis zur Zerstörung. Man bezeichnet sie deshalb auch als Entity-Life-History oder Object-Life-Cycle. Entity-Klassen initiieren selbst keine Nachrichten, sie reagieren eher, wenn ihre Operationen gebraucht werden.



**Abbildung 9.2:** Beispiele für Entity-Klassen in einem Videorecorder

## Wie finde ich Entity-Klassen?

Entity-Klassen finden Sie zum Beispiel, indem Sie Ihre in Form von Aktivitäts-, Sequenz- und Zustandsdiagrammen modellierten Softwareprozesse systematisch daraufhin untersuchen, welche Daten für diese Prozesse benötigt werden. Folgende Fragen helfen Ihnen dabei:

Informationsbedarf in Abläufen finden

- Gibt es Aktivitäten oder Aktionen in einem Zustands- oder Aktivitätsdiagramm, bei denen die benötigten Daten noch nicht identifiziert wurden?
- Werden Informationen benötigt, damit diese Aktivität oder Aktion arbeiten kann?
- Erzeugt eine Aktivität oder Aktion Daten von fachlicher Relevanz, die auch nach ihrer Beendigung erhalten bleiben sollen?
- Gibt es in Ihrem Aktivitätsdiagramm eine Aktivität, die nicht durch einen Steuerfluss aktiviert wird, sondern genau dann starten kann, wenn bestimmte Daten vorliegen?

Falls Sie eine der gestellten Fragen mit Ja beantworten, haben Sie Daten gefunden, die Sie in Form von Entity-Klassen ablegen sollten. Aber nicht immer ergibt sich aus einer benötigten Datenmenge sofort eine Entity-Klasse. Oft-

mals müssen Sie dieses Datenbündel bis auf Attributebene zerlegen und die einzelnen Attribute Entity-Klassen zuordnen.

Durchforsten Sie zudem Ihr Begriffslexikon, sofern Sie es noch nicht in ein Klassendiagramm überführt haben. Hinter den zentralen Begriffen verbergen sich häufig Entity-Klassen. In Kapitel 5 ist ein geeignetes Vorgehen dafür beschrieben.

## 9.2.2 Sichten-Klassen

Sichten-Klassen verdichten, sammeln und konvertieren Daten

Sichten-Klassen verdichten, sammeln und konvertieren Daten. Sie besitzen dazu kaum eigene Attribute, sondern arbeiten mit den Attributen anderer Klassen, vor allem der Entity-Klassen. In der UML kennzeichnet man solche Attribute als abgeleitet oder ableitbar (mit/vor dem Attributnamen). Die Operationen einer Sichten-Klasse bewältigen Aufgaben, die eng mit dem Erstellen oder Verwenden der Sichten verbunden sind, nämlich mit dem Verteilen oder Zusammenstellen von Daten (auf bzw. von den einzelnen Entity-Klassen, Subsystemen, Schnittstellen zur Umgebung, ...).

Die Sichtenbildung ist meist nur zu bestimmten Zeitpunkten während eines Softwareprozesses notwendig. Die zugehörigen Objekte sind daher eher kurzlebig und werden selten dauerhaft gespeichert. Die Klassen enthalten nur dann Zustandsmodelle, wenn sie zur Datenaufbereitung notwendig sind. Als Beispiel können Sie sich ein über ein Netzwerk verschicktes Datenpaket vorstellen. Dessen Datenfelder sind in einer vom Netzwerkprotokoll vorgegebenen Reihenfolge angeordnet. Sichten-Klassen übernehmen dabei die Aufgabe, den Inhalt der Datenfelder zu analysieren und auf unterschiedliche Entity-Klassen zu verteilen. Zusätzlich überprüfen sie die Einhaltung des Protokolls mittels eines Zustandsautomaten, dessen Zustände so angeordnet sind, dass sie die Reihenfolge der Abarbeitung der Datenfelder wiedergeben. Bei fehlenden oder falsch angeordneten Feldern geht er in einen Fehlerzustand über.

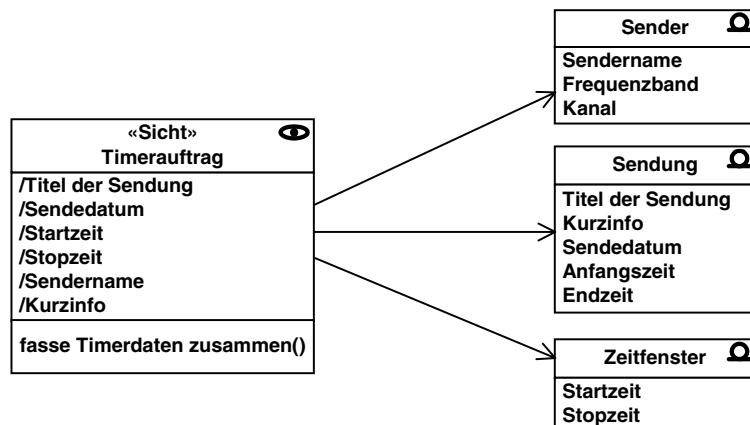


Abbildung 9.3: Beispiel einer Sichten-Klasse des Videorecorders

## Wie finde ich Sichten-Klassen?

Überlegen Sie sich, welche Sichten auf Ihre Entity-Klassen Sie benötigen. Auch hier wieder das gleiche Vorgehen: Ein Ja auf eine der folgenden Fragen ist ein Indikator, dass eine Sichten-Klasse Ihr System an dieser Stelle optimieren könnte.

- Brauchen Sie gefilterte oder verdichtete Informationen über mehrere Objekte hinweg für Suchvorgänge oder für die Präsentation dieser Informationen?
- Unterscheidet sich die fachliche Gruppierung der eingegebenen Informationen von der, wie Sie die Daten intern in Ihren Entity-Objekten verwalten?
- Interessieren Sie verdichtete Informationen über verschiedene Entity-Objekte zur Optimierung der Steuerung Ihres Systems?
- Benötigen Sie für die Kommunikation zwischen Komponenten oder mit der Umwelt komprimierte, aufbereitete Daten?
- Werden in Ihrem System Daten verdichtet, analysiert, effizienter gemacht, zusammengestellt, komprimiert, konvertiert?

Ein-/Ausgabe-sichten finden

Leidet Ihr System an knappen Speicherressourcen? Evtl. genügt es, nur die verdichteten Daten aufzubewahren und die ursprünglichen Entity-Objekte zu vernichten. In diesem Fall sollten Sie diese Sicht dann eher als Entity-Klasse kennzeichnen.

### 9.2.3 Steuerungs-Klassen

Steuerungs-Klassen steuern und koordinieren übergreifende Abläufe von Softwareprozessen durch Delegation. Sie haben wenig eigene Attribute, die dann meist nur einen (Prozess-)Status anzeigen oder den Ablaufzeitpunkt einer Operation festhalten. Die wenigen, komplexen Operationen dieser Klassen aktivieren die Methoden der gesteuerten Klassen.

Steuerungs-Klassen koordinieren Abläufe

Die Objekte der Steuerungs-Klassen existieren in etwa so lange wie der zugehörige Softwareprozess, den sie steuern. Bei RTE-Systemen sind sie daher sehr kurzlebig und meist transient. Zu Steuerungs-Klassen existiert meist nur ein einziges Objekt (Singleton).

Sie besitzen manchmal Zustandsautomaten, die globalere Abläufe beschreiben und oft Nachrichten an andere Objekte als Aktionen absetzen oder Ereignisse von anderen Objekten mitgeteilt bekommen.



Abbildung 9.4: Beispiel einer Steuerungs-Klasse eines Videorecorders

### Wie finde ich Steuerungs-Klassen?

---

Koordinationsbedarf aufdecken

Überlegen Sie sich für Aktivitäten und Aktionen Ihrer Aktivitätsdiagramme, Sequenzdiagramme und Zustandsdiagramme, ob Sie diese als Operationen Entity-Klassen zuordnen können, oder ob es sinnvoll ist, daraus eine eigenständige Steuerungs-Klasse zu kreieren. Ein Ja auf eine der folgenden Fragen ist ein Indikator, dass eine Steuerungs-Klasse Ihr System an dieser Stelle optimieren könnte.

- Gibt es einen Softwareprozess, bei dem mindestens drei Entity-Objekte am Gelingen aktiv beteiligt sind? Aktiv beteiligt bedeutet, dass von allen Objekten Informationen zusammengetragen und weiterverarbeitet werden müssen. Keines der beteiligten Objekte ist dabei prädestiniert, diese Aufgabe zu steuern.
- Gibt es parallele Aktivitäten in Ihrem Aktivitätsdiagramm, die synchronisiert werden müssen? Ist auch hier wieder keines der Entity-Objekte geeignet, alle in den Synchronisationsbalken eingehenden Ereignisse zu empfangen, auszuwerten und dann das ausgehende Ereignis zu senden?
- Gibt es Zustände in Ihren Zustandsdiagrammen, die sich keinem der gefundenen Entity-Objekte zuordnen lassen? Handelt es sich dabei eher um einen Zustand, der einem Steuerungs-Objekt zugeordnet werden sollte, das mehrere Entity-Objekte unterstützt und kontrolliert?
- Müssten Sie die im Zustandsdiagramm für einen Prozess gefundenen Zustände mindestens drei verschiedenen Objekten zuordnen? Sind somit mindestens drei Objekte am Gelingen dieses Softwareprozesses aktiv beteiligt?
- Müssen Sie parallele Zustandsautomaten innerhalb eines Systemprozesses wieder synchronisieren? Ist keines der Entity-Objekte, das Zustände des Automaten innehat, geeignet, alle für die Synchronisation benötigten Ereignisse zu überwachen, auszuwerten und dann das ausgehende Ereignis zu senden?
- Gibt es in einem Ihrer Sequenzdiagramme ein Entity-Objekt, das für die erfolgreiche Ausführung einer Operation sehr viele andere Objekte ansprechen und damit auch kennen muss? Diese Kommunikation können Sie durch eine Steuerungs-Klasse entkoppeln. Dies entspricht dem Mediator-Pattern.

### 9.2.4 Service-Klassen

---

Service-Klassen kapseln Dienstleistungen

Service-Klassen kapseln Dienstleistungen, die von unterschiedlichen Softwareprozessen genutzt werden. Sie reduzieren den Aufwand für Pflege und Implementierung, da sie gemeinsame Funktionalität an einer Stelle zentralisieren. Service-Klassen besitzen je nach Situation zahlreiche komplexe Operationen, die mehr oder minder aufwändige Algorithmen ausführen.

Die Lebensdauer der Service-Objekte hängt von ihren Dienstleistungen ab. Da sie prozessübergreifend zur Verfügung stehen, können die Objekte unabhängig von einem einzelnen Softwareprozess existieren. In der Regel sind sie aber transient und arbeiten auf den persistenten Daten der Entity-Klassen, die sie sich beschaffen oder per Parameter übergeben bekommen.

Modellieren Sie Zustandsautomaten für Ihre Service-Klassen, wenn die Serviceleistungen nicht uneingeschränkt angeboten werden oder ihr Verhalten komplex ist.

Zeitfenster	
Sendeauskunft(Kanal, Datum)	
Videotextseite abrufen(Kanal, Seitennummer)	

**Abbildung 9.5:** Beispiel für eine Service-Klasse des Videorecoders

## Wie finde ich Service-Klassen?

Wiederkehrende Dienste finden Sie, indem Sie nach häufiger auftretenden Operationen suchen, die dann als ein allgemeiner Service in einer Service-Klasse angeboten werden können. Sollten Sie bereits Service-Klasse in Ihrem Modell besitzen, so sollten Sie zuerst versuchen, neue Services den bereits bestehenden Klassen zuzuordnen. Hierbei müssen Sie evtl. bestehende Klassen reorganisieren und generalisieren.

Nach Wieder-verwendbarem suchen

- Werfen Sie einen Blick auf Ihr Use-Case-Diagramm. Gibt es dort einen Include-Use-Case? Könnte die Dienstleistung dieses Prozesses so allgemein sein, dass man ihn als allgemein in Ihrem System verfügbaren Service zur Verfügung stellen sollte?
- Nehmen Sie die erstellten Softwareprozess-Beschreibungen noch einmal zur Hand. Gibt es dort essenzielle Schritte, die in mehreren Softwareprozessen auftauchen?
- Finden Sie in den Ihnen vorliegenden Zustandsautomaten gleiche Aktivitäten oder Aktionen in mehreren Zuständen (auch über Zustandsautomaten unterschiedlicher Softwareprozesse oder Objekte hinweg)?
- Prüfen Sie die Operationen der unterschiedlichen Objekte Ihrer Sequenzdiagramme. Gibt es hier mehrfach auftretende Operationen bei Objekten unterschiedlicher Klassen?
- Lassen sich aus Ihrem Entity-Klassenmodell Services von allgemeinem Interesse herausfaktorisieren?

Eine positive Antwort auf eine der oben gestellten Fragen weist auf eine Operation hin, die in einer Service-Klasse landen sollte.

## **9.3 Dokumentation des Klassenmodells**

---

### **9.3.1 Erstellen des Klassendiagramms**

---

Ein Modell oder viele Modelle?

Wie sollten Sie nun die gefundenen Entity-Klassen, Sichten-Klassen, Steuerungs-Klassen und Service-Klassen modellieren? Sie können sich vermutlich gut vorstellen, dass die Anzahl der Klassen im Vergleich zu den bisher bereits notierten Entity-Klassen stark ansteigt. Deshalb müssen Sie zu Hilfsmitteln greifen, um die Übersicht behalten zu können. Viele Tools stellen Ihnen Filtermechanismen zur Verfügung, mit denen Sie Klassen nach bestimmten Merkmalen (z. B. dem Stereotyp) ein- oder ausblenden können. Damit können Sie sich beliebige Sichten generieren.

Für jeden Stereotyp von Klassen gibt es spezifische Modellierungsregeln, die wir im Folgenden erläutern.

#### **Modellierung der Entity-Klassen**

---

Fette Entity-Klassen modellieren

Gute Klassenmodelle haben „fette“ Entity-Klassen, die so viele komplexe Operationen enthalten, wie es fachlich sinnvoll ist. Versuchen Sie, in den Klassen ausreichend fachliche Verantwortung zu konzentrieren. Dazu gehört auch die Unterbringung von komplexeren Operationen, die Sie nicht gleich in eine andere Klassenart, z. B. eine Steuerungsklasse, auslagern sollten.

Wundern Sie sich nicht, wenn Sie für Ihr System eher wenige Entity-Klassen finden. Sie sind in RTE-Systemen eher selten. Die wenigen vorkommenden Exemplare sind dafür aber meist umfangreich. Ein separates Diagramm für Entity-Klassen lohnt sich für fast jedes Projekt, weil es Ihnen einen Überblick über alle langlebigen Datenstrukturen gibt. Modellieren Sie Entity-Klassen immer und aktualisieren Sie sie bei Änderungen.

#### **Modellierung von Sichten-Klassen**

---

Sichten-Klassen mit ihren Datenklassen modellieren

In RTE-Systemen treten Sichten-Klassen häufig an Stellen auf, an denen aus Technologieaspekten oder begrenzter Übertragungskapazität Daten gebündelt oder komprimiert werden müssen. Die Sichten-Klassen selbst sind den Klassen, von denen sie ihre Daten erhalten, meist nicht bekannt. Sichten-Klassen haben somit meist nur gerichtete Beziehungen zu den Entity-Klassen. Verwechseln Sie Sichten-Klassen nicht mit Präsentations-Klassen (wie z. B. CWindow aus der MFC oder JFrame aus der Swing-Bibliothek), die rein zur

Repräsentation an der Oberfläche für eine bestimmte Darstellungstechnologie dienen.

Sollten Sie in Ihrem System viele Sichten benötigen, so überfrachten Sie das Klassenmodell schnell. Zeichnen Sie für Sichten-Klassen immer separate Diagramme – nur mit der jeweiligen Sicht und den beteiligten Datenhaltungsklassen (oder nutzen Sie die Filterfunktionen Ihres Tools).

## Modellierung von Steuerungs-Klassen

Steuerungs-Klassen sind typische Boss-Klassen. Hassan Gomaa [Gom00] vergleicht sie auch mit Dirigenten eines Orchesters. Modellieren sie magere Steuerungs-Klassen, die die Arbeit in großen Teilen zum Beispiel an fette Entity-Klassen delegieren. Haben Sie wirklich komplexe Operationen, die Sie keiner anderen Klasse eindeutig zuordnen können, dann und nur dann sind diese bei Steuerungs-Klassen gut aufgehoben.

Sie können Steuerungs-Klassen auch in einer hierarchischen Struktur zur Aufgabenverteilung anordnen, um komplexe Aufgaben zu verteilen. Nehmen Sie aber keine Designoptimierungen, wie z. B. das Mediator-Pattern (vgl. [Gam95]) als alleinigen Grund für die Erzeugung einer Steuerungs-Klasse.

Vermeiden Sie bei RTE-Systemen mit Multiprozessorverteilung Hierarchien von Steuerungs-Klassen. Den Gewinn an Flexibilität und Beherrschbarkeit bezahlen Sie meist mit erhöhtem Kommunikationsaufwand und schlechteren Durchsatzwerten.

Zeichnen Sie auch die Steuerungs-Klassen (analog zu den Sichten-Klassen) mit gerichteten Assoziationen in ein separates Klassendiagramm (oder stellen Sie die Filterfunktionen Ihres Tools entsprechend ein).

## Modellierung von Service-Klassen

Service-Klassen bieten Ihnen mehr Flexibilität durch die gemeinsame Nutzung von Dienstleistungen und eine höhere Wiederverwendbarkeit dieser Dienste. Dieser Optimierungseffekt ergibt sich oftmals nicht auf den ersten Blick, sondern kann durch Strukturierung und Optimierung des bestehenden Klassenmodells erreicht werden. Service-Klassen werden über gerichtete Beziehungen von ihren Klienten benutzt und benutzen ihrerseits Entity-Klassen, falls der Service sich seine Daten von dort beschafft.

Magere  
Steuerungs-  
Klassen  
modellieren,  
die delegieren

Besitzt Ihr RTE-System viele parallele Prozesse? Dann sollten Sie unter Berücksichtigung von konkurrierenden Zugriffen auf Daten und des Synchronisationsaufwandes Service-Klassen modellieren. Wenn Sie diese lokal instanzieren, erhöhen Sie dadurch Durchsatz und Verarbeitungsgeschwindigkeit.

**Tabelle 9.1:** Fachliche Klassenarten in der Übersicht

Entity-Klassen	Sichten-Klassen
<p><b>Zweck:</b> Datenspeicher, die Zugriffs- und Verwaltungsdienstleistungen für die Daten anbieten.</p>	<p><b>Zweck:</b> Klassen, die Daten sammeln, verdichten, auswerten und auf andere Klassen verteilen.</p>
<p><b>Charakteristik:</b></p> <ul style="list-style-type: none"> <li>• viele Attribute</li> <li>• wenige komplexe, viele primitive Operationen (getter/setter)</li> <li>• sehr einfache Zustandsmodelle</li> <li>• langlebig, meist persistent</li> </ul>	<p><b>Charakteristik:</b></p> <ul style="list-style-type: none"> <li>• fast nur abgeleitete bzw. ableitbare Attribute</li> <li>• wenige, komplexe Operationen, die Daten zusammenstellen bzw. verteilen</li> <li>• Zustandsmodelle im Zusammenhang mit der Datenbearbeitung</li> <li>• kurzlebig, meist transient</li> </ul>
<p><b>Beziehungen:</b> bidirektional zu anderen Entity-Klassen</p>	<p><b>Beziehungen:</b> gerichtete Beziehungen zu den Datenquellen oder -senken der originären Daten</p>
<p><b>Modellierungsziel:</b></p> <ul style="list-style-type: none"> <li>• „fette“ Klassen</li> <li>• Machtzentration durch Übernahme von Verwaltungsaufgaben durch Operationen der Entity-Klassen</li> <li>• als eigenständiges Klassendiagramm modellieren und aktualisieren</li> </ul>	<p><b>Modellierungsziel:</b></p> <ul style="list-style-type: none"> <li>• von Entity-Klassen separieren</li> <li>• einzeichnen in separate Diagramme nur mit den nötigsten beteiligten Klassen</li> </ul>
Steuerungs-Klassen	Service-Klassen
<p><b>Zweck:</b> Koordination von komplexen Abläufen durch Delegation.</p>	<p><b>Zweck:</b> Kapselung von gemeinsam genutzten Dienstleistungen.</p>
<p><b>Charakteristik:</b></p> <ul style="list-style-type: none"> <li>• wenige und dann meist Zustandsattribute</li> <li>• wenige, komplexe Operationen, die die Operationen anderer Klassen nutzen</li> <li>• Zustandsmodelle, die größere zusammenhängende Abläufe beschreiben</li> <li>• kurze bis mittlere Lebensdauer</li> </ul>	<p><b>Charakteristik:</b></p> <ul style="list-style-type: none"> <li>• nur kurzlebige Attribute, oder Konstante</li> <li>• viele, komplexe Operationen, die hauptsächlich Algorithmen ausführen</li> <li>• u. U. komplexe Zustandsmodelle</li> <li>• meist kurzlebig</li> </ul>
<p><b>Beziehungen:</b> gerichtete Beziehungen zu allen gesteuerten Klassen</p>	<p><b>Beziehungen:</b> gerichtete Beziehungen zu den Klassen der im Service benutzten Daten</p>
<p><b>Modellierungsziel:</b></p> <ul style="list-style-type: none"> <li>• „magere“ Klassen</li> <li>• nur komplexe Operationen</li> <li>• Das Leben als Chef ist leichter, wenn man delegiert!</li> <li>• in Ausschnittsdiagramme einzeichnen</li> </ul>	<p><b>Modellierungsziel:</b></p> <ul style="list-style-type: none"> <li>• Flexibilität durch Parametrisierung</li> <li>• Kapselung von wiederverwendbaren Einheiten zur Reduktion von Pflege- und Implementierungsaufwand</li> <li>• Modellierung im Rahmen von „Factoring“</li> </ul>

## 9.3.2 Beschreibung der Klassendiagramm-Elemente

Das fachliche Klassenmodell ist die Basis für die technische Architektur und die Programmierung Ihres Softwaresystems und wird somit die zentralen grauen Zellen in Ihrem Requirementsgehirn belegen. Berücksichtigen Sie die Fertigkeiten Ihrer Programmierer und technischen Architekten bei der Frage, wie viel Dokumentation Sie erstellen müssen. Wir geben Ihnen dazu in den nächsten vier Abschnitten Beschreibungsmuster für Klassen, Beziehungen, Attribute und Operationen an die Hand. Die meisten CASE-Tools besitzen ähnliche Felder. Prüfen Sie, ob alle relevanten Aspekte bei Ihren Tools schon vorhanden sind bzw. ergänzen Sie diese.

Wenn Sie aus Ihrem Modell automatisch Code generieren wollen, müssen Sie in aller Regel die Beschreibungsmuster vollständig ausfüllen.

Aktualisierungen zu den Mustern und weitere Beispiele finden Sie unter [www.sophist.de/buch/rte](http://www.sophist.de/buch/rte) und [www.b-agile.de/de/rte](http://www.b-agile.de/de/rte).

**Tabelle 9.2:** Beschreibungsmuster für eine Klasse

Name	ein aussagekräftiger Name und eventuell verwendete Synonyme bzw. Abkürzungen des Namens	Beschreibungsmuster: Klasse
Merkmale, Eigenschaftswerte	frei definierbare Merkmale ( <i>tagged values</i> ) oder Eigenschaftswerte ( <i>properties</i> ) der Klassen, insbesondere: <ul style="list-style-type: none"> <li>• Autor</li> <li>• Instanziierbarkeit (abstrakt?)</li> <li>• Ableitbarkeit</li> <li>• Persistenz (persistent/transient)</li> </ul>	
Definition	was stellt die Klasse dar und warum ist sie im Kontext Ihrer Software wichtig	
Zusatzinformationen	Details, welche die Stakeholder zu dieser Klasse äußern. Diese Informationen werden im Lauf der Modellierung weiterverarbeitet und landen dann z. B. als Attribute, Operationen oder Beziehungen dieser Klasse im Modell	
Regeln für Konstruktion und Destruktion	Regeln, die beim Anlegen bzw. Vernichten von Objekten dieser Klasse zu erfüllen sind	
Sichtbarkeit	die Sichtbarkeit der Klasse	
Interfaces	die Schnittstellen, die die Klasse implementiert	
Mengengerüst	die Anzahl der Objekte, die maximal gleichzeitig existieren	
Fortschritt der Implementierung	Wie viel Prozent der Klasse wurden bereits in Code umgesetzt? Geschätzter Zeitaufwand für die noch ausstehende Codierung inkl. der zugehörigen Attribute, Operationen und Realisierung der Beziehungen der Klasse.	
Fragen/Bemerkungen	Notiz für folgende Gespräche	

Bei der Klassenbeschreibung können Sie auch Steuerungsanweisungen für den Code-Generator hinterlegen, z. B. welche Operationsrumpfe automatisch generiert werden sollen. In der Regel sind dies der Konstruktor bzw. Copy-Konstruktor, der Destruktor und, abhängig von der Programmiersprache, bestimmte Operatoren (z. B. in C++ „<<“, „[ ]“, „->“, ...).

**Tabelle 9.3:** Ein ausgefülltes Beschreibungsmuster einer Klasse

Name	Programmplatz
Merkmale, Eigenschaftswerte	{Autor = Mr. X.} {instanziierbar} {ableitbar} {persistent}
Definition	Die Klasse repräsentiert einen Speicherplatz, in dem die vom Bediener festgelegten Einstellung zu einem Sender (Nr., Kanal, Sendername, ...) festgehalten werden.
Zusatzinformationen	–
Sichtbarkeit	Öffentlich
Interfaces	keine
Mengengerüst	Maximal 99
Fragen/Bemerkungen	–

**Tabelle 9.4:** Beschreibungsmuster für eine Beziehung

Beschreibungs- muster: Beziehung	Name	ein aussagekräftiger Name und eventuell verwendete Synonyme bzw. Abkürzungen des Namens.
	Merkmale, Eigenschafts-werte	frei definierbare Merkmale ( <i>tagged values</i> ) oder Eigen-schaftswerte ( <i>properties</i> ) der Klassen, insbesondere: <ul style="list-style-type: none"> <li>• Persistenz (persistent / transient)</li> </ul>
	Definition	<i>was</i> stellt die Beziehung dar und <i>warum</i> ist sie für die beteiligten Klassen sinnvoll
	Zusatzinformationen	Details, welche die Stakeholder zu dieser Beziehung äußern. Diese Informationen werden im Lauf der Modellierung weiterverarbeitet und landen dann z. B. als Attribute, Operationen oder Beziehungen in einer Klasse im Modell (Auflösen einer attributierten Beziehung)
	Regeln für Konstruktion und Destruktion	Regeln, die bei Anlegen bzw. Vernichten der Beziehung eingehalten werden müssen
	Multiplizitäten	Anzahl der an der Beziehung beteiligten Objekte
	Ableitungsinformationen (bei abgeleiteten Beziehungen)	Ableitungsregeln für die Beziehung
	Mengengerüst	die Anzahl der Beziehungen, die maximal gleichzeitig instanziert sind
	Fragen/Bemerkungen	Notiz für folgende Gespräche

**Tabelle 9.5:** Ausgefülltes Beschreibungsmuster für eine Beziehung

Name	wickelt ab
Merkmale, Eigenschaftswerte	{transient}
Definition	Die Beziehung hält fest, welcher Timerauftrag aktuell durch die Aufnahmeauftragsabwicklung abgewickelt wird.
Zusatzinformationen	–
Regeln für Konstruktion und Destruktion	Aufbau zu Beginn der Aufnahme, Abbau nach Beendigung der Aufnahme
Multiplizitäten	Die Aufnahmeauftragsabwicklung wickelt genau einen Timerauftrag ab.
Ableitungsinformationen (bei abgeleiteten Beziehungen)	nicht abgeleitet
Mengengerüst	1 Instanz, da gleichzeitig nur 1 Timerauftrag abgewickelt werden kann.
Fragen/Bemerkungen	Sollen die abgearbeiteten Aufträge ermittelt werden können?

**Tabelle 9.6:** Beschreibungsmuster eines Attributs

Name	ein aussagekräftiger Name und eventuell verwendete Synonyme bzw. Abkürzungen des Namens	Beschreibungs-muster: Attribut
Definition	was stellt das Attribut dar und warum ist es im Kontext der Klasse wichtig	
Zusatzinformationen	Details, welche die Stakeholder zu diesem Attribut äußern. Diese Informationen werden im Lauf der Modellierung weiterverarbeitet und können eventuell dazu führen, dass das Attribut eine eigenständige Klasse wird.	
Merkmale, Eigenschaftswerte	frei definierbare Merkmale ( <i>tagged values</i> ) oder Eigenschaftswerte ( <i>properties</i> ) des Attributs, insbesondere: <ul style="list-style-type: none"> <li>• Veränderbarkeit {invariant}</li> <li>• Persistenz {persistent} / {transient}</li> </ul>	
Datentyp	(programmiersprachenunabhängiger) Datentyp des Attributs	
Wertebereich	Werte (inkl. Einheit und Genauigkeit), die das Attribut annehmen kann	
Initialwert	Vorbelegungswert beim Erzeugen des Attributs	
Struktur des Attributs (bei komplexen Attributen)	Anzahl bei Attributfeldern oder die Strukturelemente bei zusammengesetzten Attributen	
Ableitungsinformationen (bei abgeleiteten Attributen)	Quelle für originäre Attribute oder die Ableitungsregeln	
Fragen/Bemerkungen	Notiz für folgende Gespräche	

Nutzen Sie die Möglichkeit vieler Softwarewerkzeuge, die Zugriffsfunktionen (get und set) automatisch zu generieren. Gerade bei Entity-Klassen erspart Ihnen dies sehr viel Schreib- und Modellierungsarbeit. Falls möglich, sollten Sie diese Funktionen durch Sichten aus Ihrem Modell ausblenden. Sie verlieren dabei nichts an fachlichem Inhalt und Ihre Diagramme gewinnen an Übersichtlichkeit.

**Tabelle 9.7:** Ein ausgefülltes Beschreibungsmuster eines Attributs

Name	VPS Signalcode; Video Programm System Signalcode
Definition	Der VPS Signalcode ist eine von Sendern ausgestrahlte digitale Information zu einer Sendung. Er enthält codiert den Kanal, das Sendedatum und die Startzeit der Sendung. Der VPS Signalcode ermöglicht es, timergesteuerte Aufnahmen auch bei Sendeausfällen oder allgemeinen Programmänderungen korrekt abzuwickeln.
Zusatzinformationen	–
Merkmale, Eigenschaftswerte	{veränderbar}, {persistent}
Datentyp	long (32bit)
Wertebereich	0x0000 0000 – 0xFFFF FFFF (Bedingung: logische Datumskombinationen)
Initialwert	0x 0000 0000
Struktur des Attributs (bei komplexen Attributen)	2 bit reserviert (LSB), 5 bit Tag, 4 bit Monat, 5 bit Stunde, 6 bit Minute, 4 bit Land, 6 bit Programmquelle.
Ableitungsinformationen (bei abgeleiteten Attributen)	–
Sichtbarkeit	Privat
Fragen/Bemerkungen	Gibt es Tabellen für die Länder- und Programmquellen-codierung?

Das nachfolgende Beschreibungsmuster für Operationen gilt auch für Aktivitäten und Aktionen, da aus diesen früher oder später Operationen einer Klasse werden müssen.

**Tabelle 9.8:** Beschreibungsmuster für eine Operation

Beschreibungs-muster:  
Operation

Name	ein aussagekräftiger Name und eventuell verwendete Synonyme
Definition	was macht die Operation und warum ist sie im Kontext der Klasse wichtig
Zusatzinformationen	Details, welche die Stakeholder zu dieser Operation äußern. Diese Informationen werden im Lauf der Modellierung weiterverarbeitet und landen dann z. B. als Attribute, Operationen oder Beziehungen dieser Klasse im Modell

Merkmale, Eigenschaftswerte	frei definierbare Merkmale ( <i>tagged values</i> ) oder Eigenschaftswerte ( <i>properties</i> ) der Operation, insbesondere: <ul style="list-style-type: none"> <li>• Überschreibbarkeit</li> <li>• Instanziierbarkeit, z. B. {abstrakt}</li> </ul>
Eingaben	Werte, die der Operation übergeben werden
Ausgaben	Werte, die die Operation zurückgibt
Exceptions	Ausnahmen, die von der Operation generiert werden können
Ablauf	Beschreibung der Abläufe oder des Algorithmus, den die Operation durchläuft
Dauer	Maximale Ausführungsduer der Operation
Vor- und Nachbedingungen	Garantierte Bedingungen beim Eintritt in die Operation und nach dem Abschluss der Operation
Fragen/Bemerkungen	Notiz für folgende Gespräche

**Tabelle 9.9:** Ausgefülltes Beschreibungsmuster für eine Operation

Name	Timersendung aufnehmen
Definition	Die Operation steuert eine timerprogrammierte Aufnahme einer Sendung.
Zusatzinformationen	–
Merkmale, Eigenschaftswerte	<ul style="list-style-type: none"> <li>• überschreibbar,</li> <li>• nicht abstrakt,</li> <li>• läuft als eigener Thread</li> </ul>
Eingaben	Keine
Ausgaben	Statusmeldung: <ul style="list-style-type: none"> <li>• OK = Aufnahme korrekt abgewickelt</li> <li>• NOTIMER = kein Timerauftrag vorhanden</li> <li>• HW_ERROR = hardwarebedingter Fehler</li> <li>• NOVPS_ERROR = fehlendes VPS-Signal</li> <li>• ERROR = sonstiger Fehler</li> </ul>
Exceptions	Keine
Ablauf	Normalablauf: <ul style="list-style-type: none"> <li>• Timerauftrag ermitteln (Aufnahmestatus = Timer ermittelt)</li> <li>• Sender einstellen (Sender eingestellt)</li> <li>• Hardware zur Aufnahme veranlassen (Aufnahme gestartet)</li> <li>• Hardwareaufnahme nach Ende der Sendung anhalten (Aufnahme beendet)</li> <li>• Timereintrag löschen (Timereintrag gelöscht)</li> <li>• Status „OK“ zurückliefern (zu Aufnahme bereit)</li> </ul>
Dauer	mindestens solange die Aufnahme andauert
Aus- und Eingangsbedingung	Eingangsbedingung: Aufnahmestatus = „zu Aufnahme bereit“
Fragen/Bemerkungen	–

## 9.4 Optimieren des Klassenmodells

Die bisherigen Aktivitäten dieses Kapitels haben uns zu einer Sammlung von kategorisierten Klassen geführt, die durch ein Geflecht aus Assoziationen verbunden sind. Ihr fachliches Klassenmodell ist damit inhaltlich komplett. Um aber eine flexible, überschaubare und redundanzfreie Facharchitektur zu erhalten, sollten Sie anschließend Ihre Modelle folgendermaßen überarbeiten:

- Reduzieren Sie die Komplexität des Gesamtmodells, indem Sie Klassen nach den in Kapitel 6 vorgestellten Kategorien gruppieren.
- Suchen Sie Gemeinsamkeiten oder Spezialitäten in Ihren Klassenmodellen, um diese effizient zu modellieren.
- Nutzen Sie Polymorphie aus.
- Bilden Sie fachliche Interfaces.

### 9.4.1 Vererbung und Polymorphie nutzen

Während der Erstellung Ihres Klassendiagramms werden Sie bereits die eine oder andere Generalisierung eingezeichnet haben, wenn Ihre Stakeholder schon Oberbegriffe verwendet haben. Trotzdem sollten Sie nach einem ersten Wurf des Klassendiagramms nochmals im Rahmen einer Refaktorisierung über weitere mögliche Generalisierungen nachdenken. Dabei können Sie grundsätzlich zwei Strategien verfolgen: Verallgemeinern und Zusammenfassen oder Spezialisieren und Ausgliedern.

Verallgemeinern und Zusammenfassen

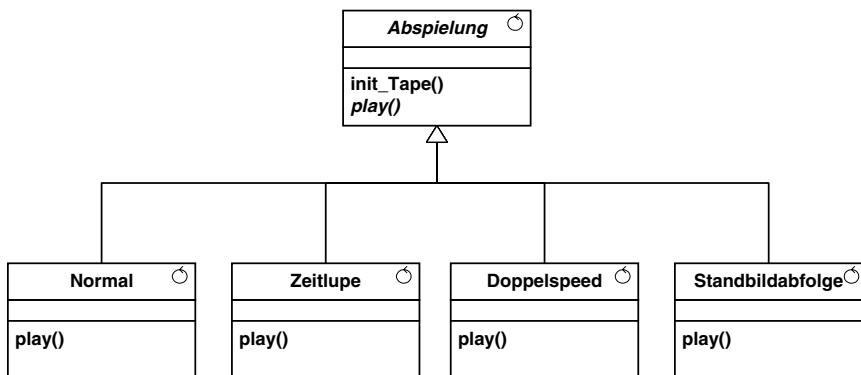
Beim Verallgemeinern und Zusammenfassen prüfen Sie systematisch, ob mehrere bestehende Klassen ähnliche Informationen verwalten oder ähnliche Dienstleistungen anbieten. Erstellen Sie für die gemeinsamen Aspekte eine Klasse, in der Sie die Gemeinsamkeiten sammeln. Behalten Sie die speziellen Daten und Fähigkeiten in den spezialisierten Klassen. Achten Sie darauf, dass Sie nur fachlich zusammenhängende Klassen generalisieren.

Spezialisieren und Ausgliedern

Beim Spezialisieren und Ausgliedern untersuchen Sie Klassen daraufhin, ob sie Operationen und Daten besitzen, die abhängig von der speziellen Ausprägung des Objekts unterschiedlich verwendet werden. Lagern Sie diese Aspekte in getrennte, spezialisierte Klassen aus.

Sowohl bei der Verallgemeinerung als auch bei der Spezialisierung sollten Sie darauf achten, dass Klassen entstehen, die eine hinreichende Zahl an Attributen und Operationen besitzen. Eine Klasse mit weniger als zwei Attributen oder Operationen ist sicherlich unnötig und verschlechtert nur das Laufzeitverhalten des Softwaresystems.

Bei einer Generalisierung erbt die spezialisierte Klasse die Datenstrukturen, das Verhalten (also den Zustandsautomaten), die Beziehungen und die Abhängigkeiten der generalisierten Klasse. Sie darf das Verhalten und die Operationen zwar überdefinieren (= Polymorphie) und diese damit spezifischer anpassen, aber nicht einfach weglassen. Oberklassen dürfen nie mehr Informationen beinhalten, als ihre Unterklassen.



**Abbildung 9.6:** Beispiel für Polymorphie

Überprüfen Sie nach der Neustrukturierung die anderen Beziehungsarten der Klassen. Womöglich können Sie Assoziationen, die zwischen Unterklassen bestehen, auflösen und an die Oberklassen anhängen.

Beziehungs-generalisierung

Es existieren jedoch auch Alternativen zur Spezialisierung, z. B. Delegation, generische Programmierung (z. B. mittels parametrisierbarer Klassen, Templates in C++ und Java) und generisches Design (z. B. über CASE-Tool-Makros und Skripte). Nutzen Sie diese Möglichkeiten, wenn Ihre Implementierungssprache keine Vererbung unterstützt.

Der Generalisierungsmechanismus hat – neben allen Vorteilen, die er bietet – auch einige Nachteile. Er hebt z. B. das Kapselungsprinzip auf. Eine Änderung an einer Klasse im Vererbungsbaum zieht automatisch – häufig auch unbeabsichtigte – Änderungen an allen darunter hängenden, spezialisierten Klassen nach sich. Komplexe Vererbungshierarchien erschweren zudem die Lesbarkeit Ihrer Modelle. Beschränken Sie sich bei RTE-Systemen auf maximal zwei Ebenen und sorgen Sie bei der Modellierung früh für die Stabilität der obersten Ebene.

Gefahren der Generalisierung

Einige Programmiersprachen unterstützen die Generalisierung auf unterschiedliche Basisklassen (Mehrfachvererbung). Wir empfehlen Ihnen, dieses Konzept nur dann anzuwenden, wenn Sie vollkommen disjunkte Basisklassen haben und sich die Vererbung nicht durch andere oben genannte Mechanismen

Mehrfach-vererbung

ersetzen lässt. Verwenden Sie diesen Mechanismus nie, wenn Ihre Sprache dies nicht unterstützt, sie sich weit oben in einem Vererbungsbaum befinden oder Ihr RTE-System harte Echtzeit-Anforderungen hat. Letzteres liegt am zeitaufwändigen Adressierungsmechanismus der Mehrfachvererbung.

### Generalisierung und harte Zeitanforderungen

Grundsätzlich lässt sich dieses Problem allgemein auf die Generalisierung übertragen. Die Realisierung der Polymorphie mit dynamischer (später) Bindung erhöht den internen Aufwand der Laufzeitumgebung. Das Pflegen der Zuordnungstabellen und die aufwändige Suche der korrekten Implementierung verhindern ein deterministisches Laufzeitverhalten. Haben Sie ein RTE-System mit harten Zeitanforderungen oder knappen Speicherressourcen? Dann gehen Sie mit Vererbung am besten sparsam um! In Sprachen wie C++ oder C# haben Sie zumindest die Möglichkeit, die polymorphen Operationen zu umgehen, in Java ist dies nicht möglich.

## 9.4.2 Interfaces bilden

Interfaces bringen durch die Einführung eines zusätzlichen Abstraktionsniveaus mehr Flexibilität in Ihre Architektur.

### Interface

Interfaces sind die nach außen bekannten Dienstleistungsschnittstellen unterschiedlicher Architekturelemente, z. B. einer Klasse, einer Komponenten oder eines Subsystems. Interfaces beschreiben abstrakte Operationen (auch wenn dies manche Programmiersprachen nicht fordern) und können daher als degenierte, abstrakte Klassen angesehen werden.

Neben der Komplexitätsreduzierung bieten Interfaces die Möglichkeit, einem Architekturelement spezielles Verhalten aufzuerlegen, da alle Bausteine, die ein Interface implementieren, dies vollständig realisieren müssen. Ein Baustein kann gleichzeitig mehrere Interfaces anbieten.

In der UML werden Interfaces durch das Symbol des entsprechenden Architekturelements mit dem Stereotyp <<Interface>> kenntlich gemacht. Abkürzend dafür kann auch das Lollipop-Symbol (○—) eingesetzt werden.

Bausteine, die ein Interface implementieren, sind mit einer gestrichelten Generalisierungsbeziehung mit dem Interface verbunden. Bausteine, die das Interface nutzen, sind mit einem gestrichelten Pfeil mit dem Interface verbunden (dependency).

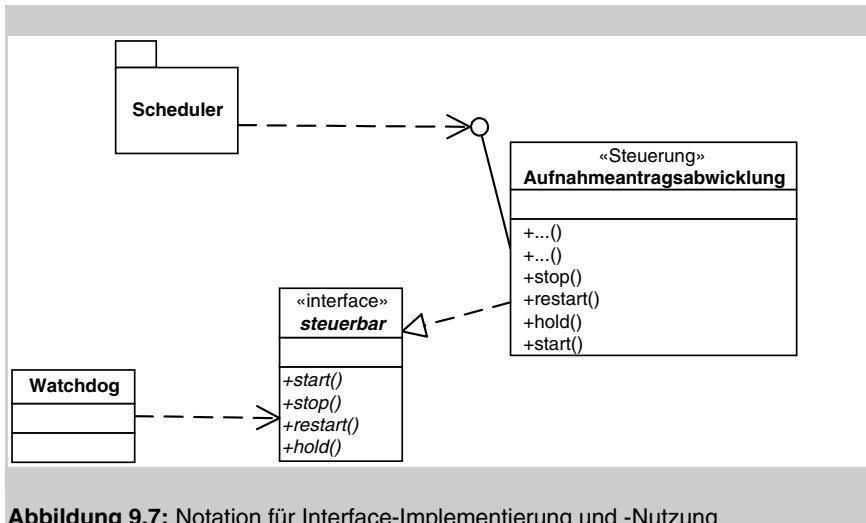


Abbildung 9.7: Notation für Interface-Implementierung und -Nutzung

Interfaces sind ein beliebtes Mittel in Klassenbibliotheken (wie MFC oder Java Swing). Es ist aber durchaus sinnvoll, Interfaces auch in die fachliche Architektur einzubringen. Interfaces bieten Ihnen die Möglichkeit, erste Informationen im Requirementsgehirn bereits zu berücksichtigen. Bei einer iterativen Entwicklung gibt es oft Situationen, bei denen Sie noch keine konkreten Anforderungen vorliegen haben. Sie wissen zwar, dass eine bestimmte Funktionalität oder Dienstleistung durch Ihr System realisiert werden muss – Ihnen fehlen aber noch konkrete Umsetzungsvorgaben. An dieser Stelle können Sie Interfaces modellieren, die bereits die Schnittstellen für die weitere Modellierung oder sogar Programmierung vorgeben. Hat sich Ihr Requirementsgehirn ausreichend gefüllt, können Sie sukzessive die „Interface-Implementierer“ nachziehen.

Eine weitere Abstraktions-ebene

Unter Umständen haben Sie in der Analyse aufgrund eines szenariogetriebenen Vorgehens **nur eine** mögliche Funktionalität einer Gruppe von ähnlichen Funktionen gefunden. Sie wissen aber, dass noch weitere Elemente der Gruppe, z. B. in Form von Klassen, hinzukommen. Auch dann sollten Sie das ähnliche Verhalten durch Interfaces herausarbeiten.

Modellieren Sie Interfaces immer dann, wenn Sie gemeinsames, abstraktes Verhalten kapseln wollen. Im Unterschied zur Generalisierungsbeziehung müssen die Klassen, die ein Interface implementieren, keine fachlichen Zusammenhänge besitzen. Häufig macht das Interface nur einen minimalen Teil der gesamten Operationen der Klasse aus.

Abbildung 9.8 zeigt Ihnen ein Beispiel für die Bündelung gemeinsamer fachlicher Dienstleistungen unterschiedlicher Softwaresysteme in einem Interface.

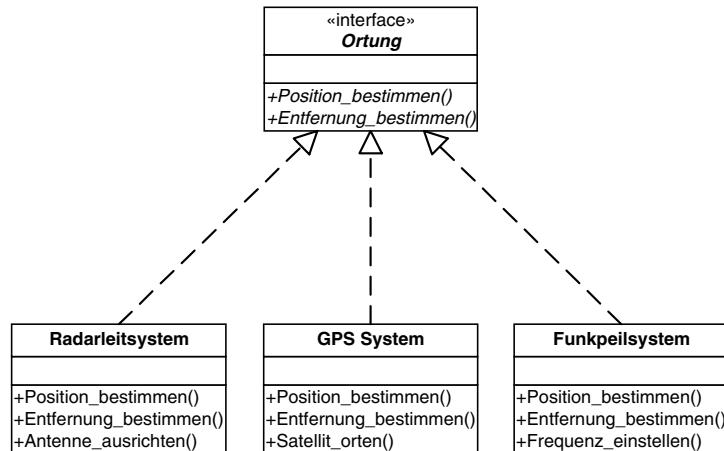


Abbildung 9.8: Beispiel einer fachlichen Interfacebildung

## 9.5 Frameworks und Patterns – Hilfsmittel für Architekten

Als wichtige Hilfsmittel bei der Top-down-Modellierung der Facharchitektur stehen Ihnen

- fachliche Frameworks und
  - Architektur-Patterns
- zur Verfügung.

### Frameworks

Ein Framework ist eine vordefinierte und vorimplementierte Lösung eines bestimmten Problems. Frameworks sind keine Fertigprodukte, sondern Anwendungsrahmen. Bei der Verwendung eines Frameworks müssen Sie die abstrakten Basisklassen konkretisieren und gemäß den spezifischen Anforderungen Ihrer Aufgabenstellung erweitern. Frameworks geben neben Code-Rümpfen auch ausprogrammierte Klassen vor.

Neben vielen technischen Frameworks drängen in den letzten Jahren auch fachliche Frameworks auf den Markt, insbesondere in Bereichen, die branchenübergreifend interessant sind. Häufig kann auch das Architekturgerüst eines vorhandenen Altsystems als Vorlage für ein Framework dienen. Das müssen Sie dann allerdings selbst erstellen (unter Aufwand-/Nutzenabwägung).

## Pattern

Patterns stellen wiederverwendbare, anpassbare Lösungen für spezielle Probleme in einem gewissen Kontext dar. Jedes Pattern hat einen Namen, beschreibt eine Problemstellung und stellt eine Lösung in Form kooperierender Klassen vor.

Wenn Sie Patterns verwenden, setzen Sie den Patternnamen als gestrichelte Ellipse in Ihr Diagramm. Verbinden Sie die Ellipse durch gestrichelte Pfeile mit den Klassen, die die entsprechenden Rollen im Pattern spielen. Die Rollennamen platzieren Sie an den Pfeilen.

**Tabelle 9.10:** Auszug aus dem Observer-Patterns (vgl. [Gam95])

Name	Observer (dt. Beobachter)
Problem	Um Inkonsistenzen in einem System zu vermeiden, müssen Objekte über Zustandsänderungen anderer Objekte informiert werden.
Lösung	Abhängige Objekte (Observer) registrieren sich bei einem Objekt (Subject) und werden bei Zustandsänderungen des Subjects von diesem benachrichtigt.
<pre> classDiagram     class Timer {         attach (process)         detach (process)         notify ()     }     class Process {         * update ()     }     class Scheduling {         update ()     }     class Idle {         update ()     }      Timer "1" -- "*" Process :      Timer "1" --&gt; Subject /      Process --&gt; Observer-Pattern /      Process --&gt; Scheduling /      Process --&gt; Idle /   </pre>	
<p><b>Abbildung 9.9:</b> Beispiel für ein Pattern</p>	
Folge	(+) Subject und Observer sind lose gekoppelt. (-) Änderungen können zu Kaskaden von Aktualisierungen führen.

Je nach Aktivität des Softwareentwicklungs-Prozesses werden Patterns aus unterschiedlichen Kategorien eingesetzt:

- **Requirements Patterns** [Rupp02], [Fow99] zur Erstellung von Anforderungsdokumenten mit integrierten Analysemethoden,
  - **Architektur-Patterns** [POSA98], [POSA00], zur Strukturierung, Gliederung und Stabilisierung der grundlegenden, fachlichen Softwarearchitektur.
  - **Design Patterns** [Gam95], zum **Bau** der technischen Softwarearchitektur und ihrer Komponenten.
  - **Idiome** zur **Optimierung** und **Gestaltung** von Codesequenzen.

Patterns sind ein hervorragendes Kommunikationsmittel für den Wissenstransfer zwischen Modellierer und Leser des Modells. Mit einem Schlagwort können umfangreiche Konzepte transferiert werden.

Haben Sie ein Pattern gefunden, das Ihr Modellierungsproblem darzustellen scheint, so sind noch einige Fragen zu klären:

- Erfüllt Ihr Problem alle Randbedingungen, die für den Einsatz dieses Patterns wichtig sind?
- Fallen die Nachteile, die durch das Pattern resultieren, für Sie nicht ins Gewicht?
- Gibt es Zusammenhänge mit anderen Patterns, die Sie für den Einsatz mit beachten sollen?
- Gibt es ein verwandtes Pattern, das Ihr Problem noch besser löst?

### Architektur-Patterns für das RTE-Umfeld

Der Einsatz von Architektur-Patterns bei der Modellierung der fachlichen Architektur von RTE-Systemen hilft Ihnen bei der Erfüllung von speziellen Eigenschaften Ihres RTE-Systems (siehe Kapitel 1). Ausgehend von diesen Eigenschaften schlagen wir Ihnen geeignete Patterns vor. Diese sollten Sie bei Ihrer Patternsuche vorrangig auf Tauglichkeit prüfen.

Patterns für  
flexible  
Einbettung

Um eine flexible **Einbettung in die Umgebung** zu gewährleisten, bieten sich gleich mehrere Patterns aus [POSA98] an. Wenn Sie hohe Portabilität anstreben, können Sie Ihre Softwarearchitektur in Schichten anordnen (*Layers-Pattern*). Diese müssen dann allerdings sehr gut entkoppelt sein, was häufig zu einer geringeren Performance führt. Eine andere Möglichkeit, Ihre Architektur flexibel zu strukturieren, ist das *Model-View-Controller-Pattern (MVC)*. Es nutzt die Vorteile der oben vorgestellten Sichten-Klassen optimal aus. Ist damit zu rechnen, dass die Nachbarn der Software dynamisch, d. h. zur Laufzeit ausgetauscht werden, empfiehlt sich unter Umständen das *Microkernel-Pattern*. Hier müssen Sie Ihre Architektur um einen essenziellen Kern entwerfen, dem dann die wechselnden Klassen als „Schale“ hinzugefügt werden. In diesem Zusammenhang ist auch das *Interceptor-Pattern* aus [POSA00] hilfreich.

Patterns für  
Verteilung

Für die Architektur von **RTE-Systeme mit verteilten Umgebungen** ist das *Broker-Pattern* aus [POSA98] interessant. Es schafft die Basis für eine möglichst lose gekoppelte Architektur, bei der es möglich ist, die einzelne Komponenten auf verschiedenste Knoten zu verteilen (Kapitel 10). Der größte Nachteil dieses Patterns gerade im RTE-Umfeld ist der Kommunikationsoverhead und der damit verbundene Zeitverlust. Setzen Sie dabei möglichst nur eine Programmiersprache ein und verzichten Sie auf eine Interface Definition Language, falls dies möglich ist.

Patterns für  
paralleles  
Processing

Bei **RTE-Systemen mit parallelen Prozessen**, die insbesondere zur Abarbeitung gleichrangiger Algorithmen oder Anfragen verwendet werden, sind die *Reactor* und *Proactor Patterns* aus [POSA98] interessant. Sie haben allerdings den entscheidenden Nachteil, dass sie hohe Anforderungen an das Betriebssystem hinsichtlich Event-Handling und Operationsverarbeitung stellen. Diese Anforderungen müssen bei der technischen Architektur berücksichtigt werden.

## 9.6 Ist Ihre fachliche Architektur stabil?

In diesem Kapitel haben wir die fachlichen Bestandteile unserer Software präzisiert und in ein fachliches Architekturmodell integriert. Betrachten Sie Ihr bisher erreichtes Ergebnis kritisch:

- Kennen Sie die fachliche Architektur der Software Ihres RTE-Systems?
- Haben Sie die fachliche Funktionalität in Ihr Klassenmodell integriert?
- Haben Sie die Klassen, Daten, Operationen, und Beziehungen mittels Beschreibungsmustern ausreichend dokumentiert?
- Nutzen Sie die Mechanismen der Generalisierung und Polymorphie im richtigen Maß?
- Haben Sie an den relevanten Stellen gemeinsame Schnittstellen mittels Interfaces abstrahiert?
- Haben Sie fachliche Standard-Probleme mit Architektur-Patterns gelöst?



„Maximen sind für den Intellekt das,  
was Gesetze für die Handlungen sind:  
Sie erleuchten nicht, aber sie leiten und führen  
und, obgleich selbst blind, sind sie Schutz.“

*Joseph Joubert (1754–1824)*  
*französischer Moralist*

# 10

## Technische Softwarearchitektur

---

### Fragen, die dieses Kapitel beantwortet:

- Wie betten wir essenzielle Aufgaben in verfügbare technologische Lösungsmöglichkeiten ein?
- Wie organisieren wir unser System in Form von parallelen Prozessen?
- Wie modelliert man die Struktur und das Verhalten der Softwarearchitektur mit UML-Ausdrucksmitteln?
- Wie modelliert man die verschiedenen Kommunikationsmöglichkeiten zwischen Rechnern und Prozessen?

In Kapitel 9 haben wir uns auf die Kernaufgabe unseres Systems oder Produkts konzentriert: die essenziellen Aufgaben und Daten. Von all den Klassen, die wir erstellen müssen, macht dies nur ca. 20–25 Prozent aus. Die anderen 80 Prozent der Klassen stellen den Zusammenhang der Kernaufgabe mit der verwendeten Technologie her.<sup>1</sup> In Abbildung 6.2 haben wir schon ein Muster für die Gesamtarchitektur vorgestellt, das die essenziellen Aufgaben inmitten eines Rings von technologieabhängigen Aufgaben zeigt. Jetzt betrachten wir den äußeren Ring genauer.

---

<sup>1</sup> Keine Angst: Sie brauchen dafür nicht vier bis fünf mal so lange, sondern auch nur etwa so lange, wie für die essenziellen Klassen. Die technologieabhängigen Klassen sind viel stärker nach Schema F aufgebaut und können daher teilweise auch generiert werden, wenn das Strickmuster einmal bekannt ist, d. h. wenn die Architekturentscheidungen gefallen sind.

## 10.1 Technologiegetriebene Klassen

Die Auftrennung in essenzielle und technologieabhängige Aufgaben hilft Ihnen, Ihr System änderungsfreundlicher und wartbarer zu gestalten. Alle in diesem Abschnitt beschriebenen Klassen hängen von Technologieentscheidungen ab. Sollten Sie diese einmal revidieren, dann sind auch nur Klassen in dem Ring betroffen, nicht jedoch die Kernaufgabe. Die Klassen im Ring verantworten also allgemein gesehen den Schutz der essenziellen Aufgaben vor technologischen Aspekten.

Überlegen Sie sich die Bereiche, in denen Technologie ins Spiel kommt:

- An der Schnittstelle zu menschlichen Nutzern des Systems
- An der Eingabeschnittstelle zu Nachbarsystemen
- An der Ausgabeschnittstelle zu Nachbarsystemen
- An der Schnittstelle zur darunterliegenden Hardware und Basissoftware

Abbildung 10.1 zeigt typische Aufgaben, die in diesen vier Bereichen wahrgenommen werden müssen.

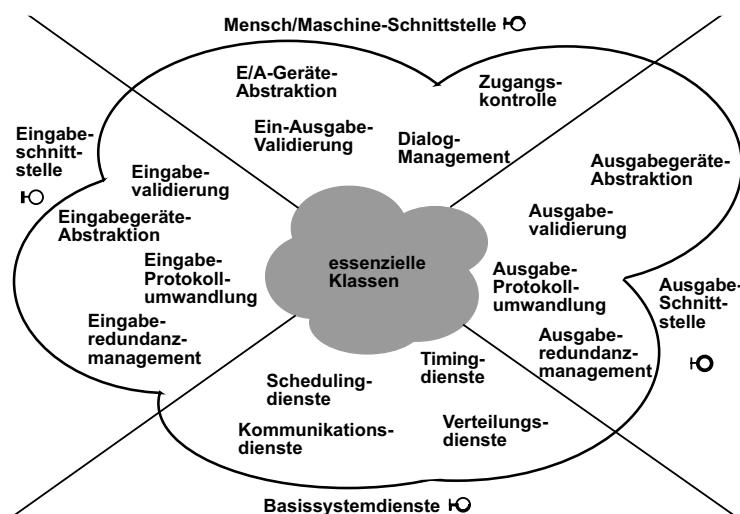


Abbildung 10.1: Technologieabhängige Klassen

### 10.1.1 Die Mensch-/Maschine-Schnittstelle

Die Mensch-/Maschine-Schnittstelle sorgt für die Umwandlung der physikalischen Eingaben von Menschen in logische Strukturen, wie sie von den essenziellen Klassen gebraucht werden, und – in der anderen Richtung – für die Aufbereitung essentieller Ergebnisse für verschiedene Ausgabemedien, die von Menschen genutzt werden. Betrachten wir als Beispiel den Bordcomputer eines Autos, dem der Fahrer Kommandos übermitteln will. Für die Essenz ist

es egal, ob diese Kommandos über eine Tastatur, über Spezialtasten oder über ein Mikrofon per Spracheingabe in das System gelangen.

Kapseln Sie in der Mensch/Maschine-Schnittstelle die Charakteristika von Ein- und Ausgabegeräten. Typische Klassen sind Tastaturtreiber, Bildschirmtreiber, sowie Interfaceklassen für Schalter, Tasten oder Lämpchen.

Aber Sie können hier nicht nur die reine Hardware verbergen. Auch Zugangskontrolle und Sicherheitsüberprüfungen sind Aufgaben für weitere Klassen in diesem Bereich: die Überprüfung von Passwörtern, das Setzen von Zugangsparametern und Nutzerprioritäten, die Überprüfung der Zugriffsberechtigung in Abhängigkeit von Systemzuständen u. ä.

Mit Klassen in diesem Bereich überprüfen Sie zudem die Syntax der Eingabe, machen Plausibilitätsprüfungen auf Wertebereiche, fangen Eingabefehler wie z. B. prellende Tasten ab und validieren Ausgaben an den Nutzer.

Hier können Sie auch die ergonomischen Aspekte der Eingabe berücksichtigen: die Formatgestaltung, die Art und Form des Dialogs mit Menschen, die Häufigkeit von Bildschirm-Updates etc.

E/A-Geräte-Abstraktion

Zugangskontrolle

Ein-/Ausgabe-Validierung

Dialog-management

## 10.1.2 Die Ein-/Ausgabe-Schnittstelle

In Abbildung 10.1 sind die Ein- und die Ausgabeschnittstellen getrennt in zwei Bereiche dargestellt. Dies soll andeuten, dass Sie meist sehr unterschiedliche Technologien auf der Eingabeseite und auf der Ausgabeseite verwenden werden. In der Praxis sollten Sie pro Technologie auch unterschiedliche Klassen schreiben. Nehmen Sie als Beispiel einen Forschungssatelliten, der seine Eingaben von Sensoren bekommt und auf der Ausgabeseite verschiedene Geräte über Aktuatoren ansteuert.

Inhaltlich erfüllen die Klassen in diesen Bereichen jedoch ähnliche Aufgaben, weshalb wir sie hier zusammen behandeln. Sie müssen die Essenz von den technologischen Aspekten der Nachbarsysteme freihalten.

Modellieren Sie in diesem Bereich Klassen, die die technischen Details des Sendens und Empfangens verbergen, u. a. die Interrupt-Behandlung von Eingangsgeräten, Handshake-Verfahren, Eingabefehlererkennung und -management sowie Plausibilitätsprüfungen von Eingangsdaten. Auch Algorithmen zum Redundanzmanagement für Ein- und Ausgaben, Protokolltreiber und Wartungsschnittstellen für Geräte können Sie in diesem Bereich unterbringen.

## 10.1.3 Die Basissystemschnittstelle

In unseren essenziellen Modellen gehen wir davon aus, dass das System sich Daten auch nach der Beendigung einer Aktivität merken kann und dass Aktivitäten Informationen untereinander austauschen können. Wir wollten auch auf Zeitereignisse reagieren, haben aber die Frage verdrängt, wer die Zeit für

uns überwacht. Wir haben auch außer Acht gelassen, dass vielleicht nicht genügend Rechenleistung vorhanden ist, um alle Aktivitäten, die wir gleichzeitig ausführen wollen, tatsächlich gleichzeitig ausführen können.

**Erweiterte Betriebssystemdienste**

In der Basisschnittstelle können Sie sich all diese Dienste zur Verfügung stellen, damit Sie sich beim Modellieren von Klassen im essentiellen Kern darüber keine Gedanken machen müssen. Zu diesen Diensten zählen Persistenzdienste, Timingdienste, Scheduling-Dienste, Kommunikationsdienste und Verteilungsdienste.

Wenn Sie Ihre Software nicht gerade auf einen nackten Rechner aufsetzen, dann stellen Ihnen Betriebssysteme – insbesondere Echtzeitbetriebssysteme – einige dieser Basisdienste zur Verfügung. Oftmals sind diese jedoch primitiv. Sie sollten lieber mit höheren Abstraktionen arbeiten. Dafür gibt es inzwischen ein weites Angebot an Frameworks und Middleware. Eine ausgezeichnete Behandlung des Themas Middleware für verteilte Echtzeitsysteme finden Sie in [POSA00].

In den folgenden Abschnitten diskutieren wir die Wünsche bezüglich Tasking und Kommunikation noch ausführlicher. Wir zeigen Ihnen, wie Sie diese Architekturaspkte mit der UML modellieren können. Die Basisdienstklassen sind dann dafür verantwortlich, diese Modelle auf Ihrer Hardware und Basissoftware zum Laufen zu bekommen.

### **10.1.4 Finden von technischen Klassen**

Die Tipps für das Auffinden von technologieabhängigen Klassen sind ziemlich einfach. Nutzen Sie Ihre Kontextdiagramme, durchforsten Sie Ihre nichtfunktionalen Anforderungen und füllen Sie systematisch das logische Architeturmuster von Abbildung 6.2 aus.

**Kontextdiagramm als Hilfe**

Legen Sie Ihr logisches und physikalischen Kontextdiagramm (vgl. Kapitel 7 und 8) nebeneinander. Darin finden Sie alle logischen Ein-/Ausgaben und alle physikalischen Kanäle. Überlegen Sie sich für jede logische Schnittstelle, auf welchem physikalischen Weg die Daten in das System oder aus dem System kommen. Notieren Sie dann die Schritte, wie man an der Eingabeschnittstelle die physikalische in die logische Form und an der Ausgabeschnittstelle die logische in die benötigte physikalische Form konvertieren kann.

Für unser Mobiltelefon aus der Abbildung 7.1 finden Sie als Schnittstellenklassen dann unter anderem

- GSM-Interface (für die Umsetzung der GSM-Treiber-API auf logische Empfangsdaten),
- DSP-Interface (mit der Aufgabe der Komprimierung und Dekomprimierung von Daten von und zu dem Digital Signal Prozessor),
- SIM-Adapter (mit der Verantwortung für die Umsetzung von Namen und Nummern in das spezifische Format der SIM-Card).

Viele der Qualitätsanforderungen und Randbedingungen geben Ihnen Hinweise auf benötigte technologiespezifische Klassen. Gehen Sie die Liste Punkt für Punkt durch und leiten Sie technologiespezifische Aufgaben daraus ab. Im Folgenden greifen wir nur einige typische Kategorien heraus.

Die nicht-funktionalen Anforderungen als Hilfe

Forderungen nach Betriebssystemen, Netzwerken, Bussystemen und Datenhaltungssystemen führen direkt zu Designentscheidungen über Basissystemdienste. Wandeln Sie die Forderung nach Windows als Betriebssystem in eine technische Komponente um, die als Schnittstelle u. a. die Eventmechanismen von Windows anbietet. Erzeugen Sie aus der Forderung nach Einsatz des CAN-BUS eine Komponente, die das Protokoll des Busses als Schnittstelle anbietet.

Überprüfen Sie Performanzanforderungen und entscheiden Sie daraufhin, welche Leistung Sie von den Basisdiensten brauchen. Ordnen Sie die Leistungsmerkmale den Beschreibungen der Komponenten zu.

Aus Sicherheitsanforderungen (Security) können Sie Klassen erstellen, die im Ein-/Ausgabebereich oder in der Mensch-/Maschine-Schnittstelle nicht erlaubte Nutzungen abfangen und Missbrauch melden.

Das logische Architekturmuster als Hilfe

Nicht zuletzt ist die Vorgabe der vier Bereiche im logischen Architekturmuster selbst bereits eine wertvolle Hilfe. Gehen Sie die Bereiche mit Hilfe von Brainstorming durch und fügen Sie alle Klassen ein, die Sie in diesem Bereich für notwendig erachten. Nicht hinterfragtes Brainstorming kann aber auch gefährlich sein. Einerseits finden Sie vielleicht aus langfristigen Designüberlegungen heraus Klassen, die aus den Anforderungen oder dem Kontext nicht ableitbar waren, aber die Architektur besonders gut erweiterbar und wartbar machen. Andererseits besteht dabei die Gefahr, dass Sie goldene Griffe in die Architektur einbauen, die niemand gefordert hat und auch nicht bezahlen will. Bewerten Sie die Ergebnisse des Brainstormings daher nochmals kritisch bezüglich Kosten und Nutzen.

## 10.2 Subsysteme und Komponenten

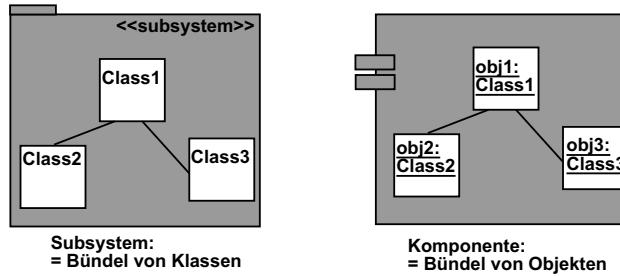
Wenn Sie alle essenziellen Klassen und auch alle Klassen aus den vier Bereichen des technologischen Rings gefunden haben, haben Sie sicherlich selbst bei kleinen Systemen eine stattliche Anzahl von Klassen. Wir brauchen größere Einheiten, damit wir den Überblick nicht verlieren. Deshalb werden wir Klassen bündeln.

Statische und dynamische Paketierung

Bündeln können wir nach zwei Gesichtspunkten: was statisch zu einer größeren Einheit zusammengehört und was zur Laufzeit eine Einheit bildet. Für die statischen, größeren Pakete (von Klassen) haben wir bisher den Begriff „Subsystem“ verwendet und bleiben auch dabei. Für die dynamischen größeren Pakete (von Objekten), die zur Laufzeit existieren, verwenden wir den Begriff

„Komponente“<sup>2</sup>. Stellen Sie sich darunter lauffähige Programme, Executables, Prozesse, Threads, aber auch DLLs, Libraries, Files, Datenbanken etc. vor. Subsysteme bilden wir eher aus fachlichen und logischen Gründen, Komponenten aus Implementierungs- und Laufzeitgründen.

Abbildung 10.2 zeigt die UML-Darstellung von Subsystemen und Komponenten.



**Abbildung 10.2:** Subsysteme und Komponenten

Die Zeichnung vereinfacht allerdings den Sachverhalt: Die Inhalte von Subsystemen und Komponenten müssen nicht direkt Klassen bzw. Objekte sein. Beide dürfen beliebig tief geschachtelt werden. Erst auf der untersten Ebene der Schachtelung finden Sie dann Klassen und Objekte.

Am besten 1:1

Idealerweise sollte die Bündelung von Klassen in Subsystemen keine andere Struktur als die Bündelung von Objekten in Komponenten ergeben. Wir empfehlen eine 1:1 Abbildung zwischen Subsystemen und Komponenten. Dies ist jedoch nicht immer machbar. Betrachten wir die anderen möglichen Zusammenhänge zwischen Subsystemen und Komponenten.

### Viele Subsysteme zu einer Komponente zusammenfassen

Dieser Fall kommt häufig vor: Sie wollen eine Menge von Subsystemen nicht zu einer größeren Einheit zusammenfassen, weil sie inhaltlich nichts miteinander zu tun haben. Trotzdem zwingen Sie technische Gründe dazu, die Gruppe von Subsystemen als eine DLL auszuliefern oder zu einer EXE zusammenzubinden.

### Ein Subsystem auf mehrere Komponenten verteilen

Das kann Ihnen passieren, wenn Sie sehr große logische Subsysteme gebildet haben. Sie haben z. B. die komplette Präsentationsebene in Ihrer Architektur

<sup>2</sup> Die UML verwendet den Begriff „Komponente“ für physikalische, austauschbare Teile des Systems (wie Source Code, Binärdateien, Executables, Skripte, Command-Files, ...). Zu einer Komponente gehört neben dem Source Code auch der ausführbare Code – das was wirklich auf den Rechnern abläuft. Wir verwenden das Wort „Komponente“ bewusst eher für diesen zweiten Aspekt, den Laufzeitaspekt, der in der UML präziser als „Instanz einer Komponente“ bezeichnet werden müsste.

als ein logisches Subsystem modelliert. Physikalisch betrachtet besteht die Präsentationsebene aber aus einigen lauffähigen Programmen und einigen Bibliotheken, die von diesen Programmen genutzt werden. In diesem Fall wollen Sie die logische Abstraktion nicht verlieren, obwohl es technisch gesehen einige separate Komponenten sind. Im Normalfall sollten Sie diese separaten Komponenten aber auch in der Struktur Ihres logischen Subsystems wiederfinden. Wenn nicht, dann sollten Sie die innere Struktur des Subsystems nochmals überprüfen und eventuell anpassen. Vielleicht haben Sie einen guten Grund für die Aufteilung übersehen – nämlich den, der dann auch die Aufteilung in getrennte Komponenten erzwingt? Falls der Schnitt mitten durch eine einzelne Klasse geht, sollten Sie Ihr Design auf jeden Fall korrigieren!

Sie können von einem logischen Subsystem auch auf eine andere Art zu mehreren Komponenten kommen, wenn Sie Objekte aus den logischen Klassen instanziieren und dann die Instanzen auf unterschiedliche Komponenten verteilen (z. B. redundante Daten, lokal gebrauchte identische Funktionalitäten).

### **Viele Subsysteme auf viele Komponenten verteilen**

Während die beiden ersten Fälle gut begründet werden können, ist uns in unserer bisherigen Praxis kein Fall vorgekommen, in dem die beiden Strukturen zwingend völlig unterschiedlich aufgebaut werden mussten. Falls Ihnen das passiert, sollten Sie Ihre Strukturen kritisch hinterfragen. Eine viele-zu-viele-Abbildung ist schwer nachvollziehbar und erschwert die Wartung und Weiterentwicklung Ihres Systems.

## **10.3 Vorgehensweise im Groben**

Designen ist ein komplexer Prozess mit vielen Entscheidungen und Abwägungen. Um diesen Prozess übersichtlich erläutern zu können, benutzen wir eine Vereinfachung. Grob gesehen kann man die Softwarearchitektur eines RTE-Systems auf vier wichtigen Abstraktionsebenen betrachten, die in Abbildung 10.3 dargestellt sind. Aus den drei Übergängen zwischen den vier Ebenen ergeben sich die drei wichtigen Tätigkeiten beim Architekturentwurf, die Sie – wie immer bei agilen Methoden – in jeder beliebigen Reihenfolge ausführen können:

- Bündeln nach Verteilungskriterien (Standort, Rechner, Platine)
- Finden von aktiven und passiven Komponenten
- Strukturbildung innerhalb einer Komponente

Wenn Sie eher bottom-up vorgehen, kennen Sie als Ergebnis der Anforderungsanalyse Ihre essenziellen Aktivitäten und Daten. Sie werden daraus dann in der dritten Ebene Prozesse bilden und danach in der vierten Ebene entscheiden, wie Sie die Prozesse auf verschiedene Knoten verteilen.

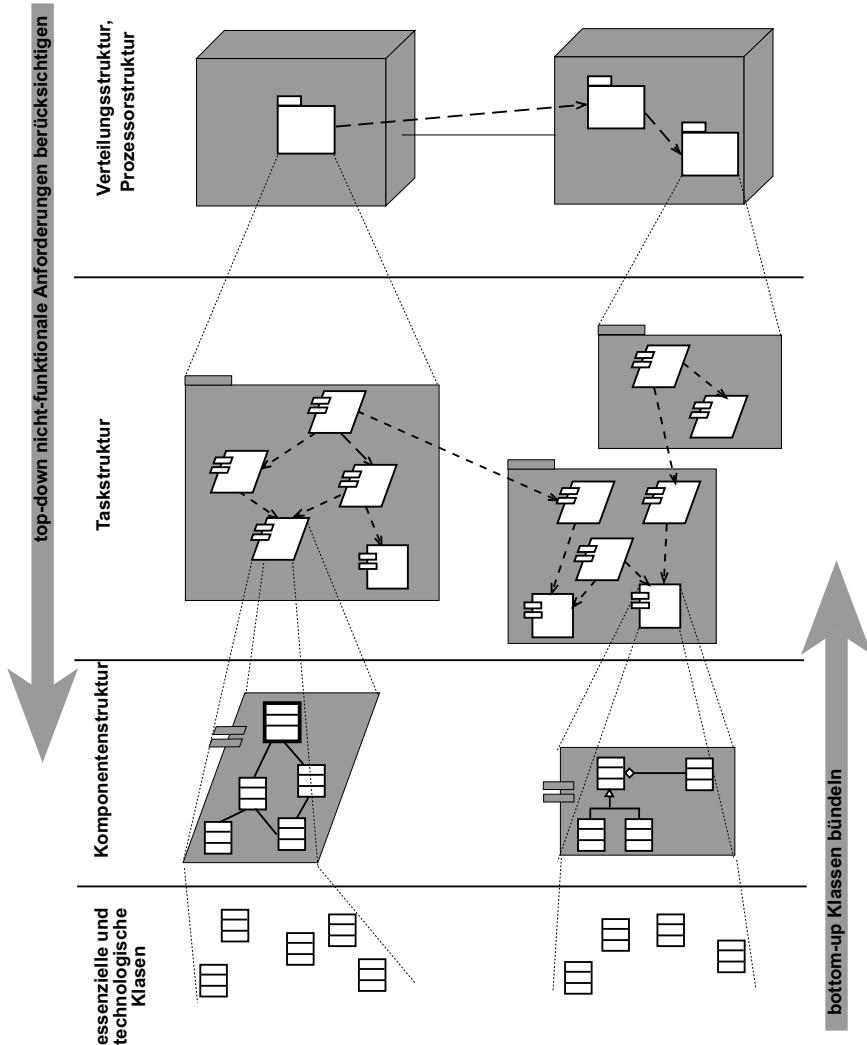


Abbildung 10.3: Abstraktionsebenen der Architektur

Die Erstellung der Verteilungsstruktur steht immer dann im Vordergrund, wenn Ihnen starke Randbedingungen bezüglich der geografischen Verteilung oder der einzusetzenden Hardware auferlegt wurden. Da Sie als Architekt in diesem Bereich dann nur wenig Freiheitsgrade haben, können Sie die Randbedingungen auch sofort in Verteilungsmodellen festhalten, wie wir in Kapitel 4 und 6 bereits erläutert haben.

Haben Sie keine derartigen Randbedingungen, dann können Sie zuerst aus Ihren logischen Systemprozessen die Taskingstruktur ableiten und dann die

Client-Server-Verteilung eher anhand von weiteren nicht-funktionalen Anforderungen (wie Effizienz bezüglich Zeit und Ressourcen, Software-Update-Verfahren, ...) bestimmen.

Lesen Sie die Abbildung 10.3 zunächst von unten nach oben: Im ersten Schritt bündeln Sie Ihre fachlichen und technologischen Klassen zu Komponenten. Sie präzisieren die Zusammenhänge zwischen den Klassen innerhalb der Komponente, indem sie statische und dynamische Abhängigkeiten festlegen.

Der Bottom-up  
Weg von  
Klassen zu  
Subsystemen

## Die interne Komponentenstruktur

Eine Komponente enthält Objekte, die normalerweise in Form eines sequenziellen Prozesses ihre Arbeit verrichten und daher leichter zu entwerfen sind als die höheren Ebenen. Die Prinzipien dafür wurden bereits in den 70er-Jahren gefunden und dokumentiert und gelten heute noch.

- Bilden Sie „Information Hiding Klassen“.
- Nutzen Sie Coupling und Cohesion als Kriterien.
- Kapseln Sie Teile, die einer hohen Änderungswahrscheinlichkeit unterworfen sind.

Einige für die Softwarearchitektur wesentlichen Prinzipien hat [Sta02] nochmals zeitgemäß unter Hinweis auf alle heute verfügbaren Technologien auf den Punkt gebracht.

Generelle Designprinzipien sind nicht der Schwerpunkt dieses Buches. Deshalb wenden wir uns der für RTE-Systeme interessantesten nächsten Abstraktionsebene zu: der Taskingstruktur. Sie fassen Komponenten zu aktiven Prozessen (Tasks) oder passiven Kommunikationsobjekten zusammen<sup>3</sup>. Dieser Tätigkeit ist der Hauptteil dieses Kapitels ab dem Abschnitt 10.3 gewidmet.

Danach müssen Sie auf der obersten Ebene die Tasks und Kommunikationsobjekte noch Platinen, Rechnern oder Standorten zuordnen.

Bei allen Zusammenfassungen müssen Sie natürlich stets Ihre nicht-funktionalen Anforderungen im Auge behalten, die Ihnen Randbedingungen für die Bündelung vorgeben. Sie müssen unter Umständen genau die Komponenten auf einem Rechner zusammenfassen, die ein Stakeholder dort haben möchte und können sich nicht um logische, fachliche Gruppierungen kümmern. Sie müssen ein logisches Bündel evtl. auch aufteilen, wenn ein Prozess oder ein Rechner die geforderte Antwortzeit nicht erbringen kann.

---

<sup>3</sup> Wir gehen davon aus, dass wir nur eine Ebene von Prozessen brauchen und innerhalb der Prozesse alles sequenziell abläuft. Wir wollen Prozesse nicht schachteln. Die UML unterscheidet über die Stereotypen «process» und «thread» schwergewichtige Prozesse mit eigenem Adressraum von leichtgewichtigen Prozessen (Threads) mit gemeinsamem Adressraum. Für viele Applikationen ist diese Unterscheidung nicht notwendig, da man entweder nur mit „echten“ Prozessen oder nur mit quasiparallelen Threads arbeitet.

Schon bei diesen ersten Zusammenfassungen von Klassen zu größeren Einheiten werden die nicht-funktionalen Anforderungen Sie evtl. mehr beeinflussen, als Ihnen lieb ist. Auch wenn fachliche Kriterien Sie bei der Bündelung in eine andere Richtung weisen würden, so erzwingen Sicherheitsüberlegungen, Speicherbegrenzungen oder Übertragbarkeitsanforderungen andere Zusammenfassungen. Deshalb ist es notwendig, bei der Softwareentwicklung für RTE-Systeme auch eine andere Herangehensweise zu berücksichtigen.

### 10.3.1 Der Top-down-Ansatz zur Architektur

Beim Top-down-Ansatz versuchen wir, die Architektur eher aus den Randbedingungen und Qualitätsanforderungen abzuleiten und grobe Entwurfsentscheidungen vorzugeben, statt sie systematisch aus den funktionalen Anforderungen aufzubauen.

Wir konstruieren die in Abbildung 10.3 gezeigten Ebenen eher von oben her. Dazu sind folgende Schritte notwendig:

1. Verteilung entscheiden
2. Tasking entscheiden
3. Innenleben ausfüllen

### 10.3.2 Verteilung modellieren

Die wesentlichen Aspekte für die Verteilung haben wir bereits in Kapitel 6 beschrieben. Beachten Sie jedoch, dass Sie Verteilungsmodelle beliebig schachteln dürfen. Abbildung 10.4 zeigt zuerst die geografische Verteilung eines Zugangskontrollsystems auf eine Zentrale und vier Eingänge mit jeweils einem eigenen PC für den Pförtner zur Überwachung sowie dem Prozessor, der das Drehkreuz bedient und Bedienungshinweise anzeigen kann. Die Zentrale selbst ist auf einer zweiten Ebene in die dort ansässigen Rechner und ihre Verbindungen aufgeschlüsselt.

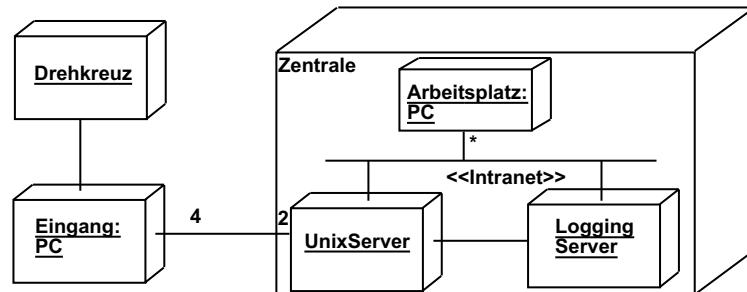


Abbildung 10.4: Geschachtelte Verteilungsmodelle

Aus Sicht der Softwarearchitektur müssen Sie die Verteilung nur soweit modellieren, wie es für die Zuordnung der Softwarekomponenten zu den Knoten notwendig ist. In obigem Beispiel gehen wir davon aus, dass wir für alle gezeichneten Knoten Software schreiben müssen.

Beschriften Sie die Verteilungsmodelle nur mit den Informationen, die für die Diskussion notwendig sind. Wenn z. B. die physikalischen Kanäle zwischen Platinen für die Softwareentwickler dadurch verborgen sind, dass ohnehin Systemdienste für die Kommunikation genutzt werden, dann müssen Sie die Kanäle auch nicht ausführlich spezifizieren. Wenn jedoch nur die Bussysteme feststehen und Sie noch Protokollhandler erstellen müssen, dann sollten Sie etwas mehr Aufwand in die Modellierung des Verteilungsmodells stecken und präzise Beschreibungen zu den Knoten und Verbindungen anlegen.

## 10.4 Aktive und passive Komponenten

Programmteile, die auf unterschiedlichen Rechnern ablaufen, sind aus physikalischen Gründen unabhängig laufende Prozesse. Die Verteilung auf verschiedene Standorte oder Prozessoren ist ein zentrales Kriterium für die Zerlegung der Gesamtaufgabe in Prozesse.

Auch Software auf einem einzelnen Rechner kann wiederum aus einigen (oder vielen) unabhängig laufenden Prozessen bestehen. Teilweise werden diese durch die asynchron ankommenden Signale aus der Systemumgebung erzwungen, teilweise durch unseren Wunsch, durch parallele Verarbeitung unsere Ressourcen besser auszunützen und schnellere Reaktionszeiten zu bekommen.

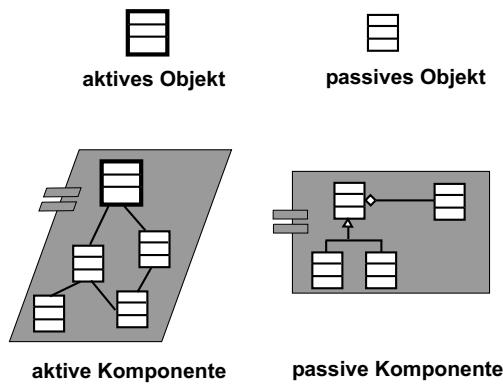
Parallele Prozesse

Philosophisch betrachtet ist jedes Objekt in unserem System ein eigenständiges Gebilde mit eigenem Leben. Jedes Objekt könnte theoretisch als selbständiger Prozess auf einem Rechner laufen. Der Ausflug in die Philosophie wird von der Realität rasch wieder beendet. Einerseits haben wir selten so viele Prozessoren wie Objekte zur Verfügung, und die Prozesskommunikation ist (zu) teuer, andererseits brauchen wir auch nicht so viele parallele Prozesse, um unsere Anforderungen zu erfüllen.

Wir werden uns daher als Designer Gedanken machen müssen, wie viele parallele Prozesse wir modellieren wollen und wie wir diese Prozesse finden.

Prozesse können auf unterschiedliche Arten miteinander kommunizieren: über synchrone und asynchrone Nachrichten, aber – wenn sie auf einem Rechner laufen – auch über gemeinsame Datenstrukturen. Im Gegensatz zu den Prozessen sind Datenstrukturen eher passiv. Sie werden von den Prozessen als Dienstleister benutzt. Wir unterscheiden deshalb in unseren Modellen passive und aktive Komponenten.

Passive Komponenten	Eine passive Komponente wartet auf Nachrichten, mit denen Operationen ihrer Objekte aktiviert werden. Sie führt diese Operation dann aus und gibt die Steuerung wieder demjenigen zurück, der die Dienstleistung angefordert hat.
Aktive Komponenten	Aktive Komponenten haben ihren eigenen Steuerfluss. Sie können Aktionen auslösen und andere Komponenten beeinflussen.
aktive und passive Komponenten	Die UML stellt uns für größere Bündel von Objekten keine Unterscheidung zur Verfügung, ob diese Bündel aktiv oder passiv sind. Sie kennt nur aktive Objekte, die durch eine dicke Umrandung von den „normalen“, passiven Objekten unterschieden werden (vgl. Abbildung 10.5).



**Abbildung 10.5:** Aktive und passive Bausteine der Architektur

Die meisten Objekte in unseren Programmiersprachen sind passive Objekte. Sie können durch Verwendung entsprechender Betriebssystemdienste aktiv gemacht werden. Sprachen wie Ada oder Java unterstützen aktive Objekte direkt.

---

<sup>4</sup> Wenn Ihr Case-Tool keine Icons für Stereotypen zulässt, so können Sie aktive Komponenten durch den Stereotypstring «aktiv» kennzeichnen. Oder versuchen Sie einmal unterschiedliche Farben für aktive und passive Komponenten! Das hilft zumindest am Bildschirm für die Unterscheidung.

## 10.5 Kriterien zur Taskbildung

Software von RTE-Systemen ist selten ein sequenziell laufendes Programm. Im Finden und Optimieren der Taskstruktur liegt eine der großen Herausforderungen für Softwarearchitekten. In diesem Abschnitt geben wir Ihnen einige Tipps, wie Sie zu Designentscheidungen bezüglich der Prozessbildung kommen. Sie können dazu viele der Vorarbeiten nutzen, die Sie schon geleistet haben. Insbesondere werten Sie die Kontextmodelle und die Use-Case-Modelle aus. Auch das logische Architekturmuster mit seinem essenziellen Kern und dem physikalischen Ring gibt Ihnen Hinweise, welche Prozesse Ihr System braucht. Anschließend sollten Sie – wenn nötig – die Taskstruktur nochmals überdenken und optimieren.

**Tabelle 10.1:** Taskarten im Überblick

<b>Durch die Systemumgebung motivierte Tasks</b>		
<b>Taskart</b>	<b>Begründung</b>	<b>Quelle</b>
Interrupt-Task	für aktive Eingabegeräte oder Nachbarsysteme	Kontext, Gerät oder System als Trigger eines Systemprozesses
Polling-Task	für passive Eingabegeräte oder Nachbarsysteme, die abgefragt werden müssen	Kontext, Gerät oder System als Trigger eines Systemprozesses
Eingabekoordinations-Task	für mehrere gleichartige aktive Eingabegeräte oder Nachbarsysteme	Logischer Kontext
Benutzereingabe-Tasks	für die Eingabe-Schnittstelle zu Menschen	Kontext, Mensch als Trigger eines Systemprozesses
Ausgabeentkopplungs-Task	zur Entkopplung der Ausgabe bei Bedarf	Systemprozess-Beschreibung
Ausgabekoordinations-Task	für mehrere Prozesse, die das gleiche Ausgabegerät oder Nachbarsystem ansprechen wollen	Logischer Kontext
Zeitgetriggerte Tasks	für Prozesse mit Zeitauslösern	Zeit als Trigger eines Systemprozesses
<b>Intern motivierte Tasks</b>		
<b>Taskart</b>	<b>Begründung</b>	<b>Quelle</b>
Steuerungs-Task	zur Koordination mehrerer anderer Prozesse	Zustandsmodelle, Aktivitätsdiagramme, ...
Periodische Tasks	zur regelmäßigen Durchführung interner Prozesse	Intern aufgesetzte zyklische Überwachung oder Berechnung
Service-Task	Zur Entkopplung wiederverwendbarer Prozesse	Interne Ereignisse
Rechenintensive Task	Zur Entkopplung zeitintensiver Prozesse	Aktivitätsbeschreibung

## 10.5.1 Arten von Tasks

Betrachten wir die einzelnen Arten von Tasks etwas näher. Wir beginnen mit den Tasks, die durch die Einbettung in die Systemumgebung motiviert sind.

Durch die System-umgebung motivierte Tasks

Durch die Kontextanalyse und die Systemprozesszerlegung mit Use-Case-Diagrammen haben Sie eine gute Vorgabe für die Taskbildung geleistet. Sie haben Ihr System so zerlegt, dass jeder asynchrone externe Trigger und jedes Zeitereignis zu einem Prozess geführt hat. Untersuchen Sie die Eigenschaften der Akteure, so führt Sie das zu folgenden Kategorien von Tasks.

### Interrupt-Tasks

Einige der Nachbarsysteme oder Geräte sind aktiv. Sie lösen von sich aus Ereignisse aus, auf die unsere Software reagieren soll. Für derartige Akteure konzipieren Sie Interrupt-Tasks (oder Interrupt-Handler). Ihre Hauptaufgabe ist es, die asynchron eingehenden Nachrichten und Signale aufzufangen. Bei wenig zeitkritischen Reaktionen auf diese Ereignisse können diese Tasks auch die vorgesehene Reaktion (z. B. das Benachrichtigen eines Operators) direkt ausführen.

Hohe Priorität

Sehr oft sind diese asynchronen Interrupt-Tasks jedoch mit sehr hoher Priorität ausgestattet, um auf schnell nacheinander eintreffende Interrupts reagieren zu können. Sie nehmen nur den Input entgegen, halten diesen in einem Zwischenspeicher fest und überlassen die zeitaufwändiger Ausführung der Reaktion (z. B. die statische Auswertung und Verdichtung) anderen Tasks.

### Polling-Tasks

Passive Nachbarsysteme abfragen

Erstellen Sie für passive Nachbarsysteme oder Geräte, die abgefragt werden müssen, Polling-Tasks. Dies ist typisch für passive Sensoren, deren Werte Sie zyklisch oder nur manchmal brauchen. Für zyklisch zu ermittelnde Werte wird ein Timer die Task regelmäßig aktivieren, worauf sie dann die Eingabedaten abholt. Die Task kann bereits den Nachrichtenverkehr optimieren, wenn sie Eingabedaten nur dann an andere Prozesse weiterleitet, wenn diese sich wirklich geändert haben. Eine Polling-Task kann auch für die nicht-zyklische, bedarfsgtriebene Abholung von Daten passiver Geräte verwendet werden, um die Applikation selbst von dieser Aufgabe zu befreien. Diese Art von Entkopplung ist jedoch auf der Ausgabeseite (s. u.) öfter zu finden.

Oft können Sie mit einer einzelnen Polling-Task eine Gruppe von passiven Sensoren der Reihe nach abfragen, wenn sich die Abfragezyklen koordinieren lassen.

## **Eingabekoordinations-Tasks**

---

Führen Sie eine Eingabekoordinations-Task ein, wenn Sie bei der Betrachtung des logischen und physikalischen Kontexts feststellen, dass die gleiche logische Eingabe von mehreren unterschiedlichen aktiven Nachbarsystemen oder Geräten kommen kann. Sie überwacht alle Lieferanten gleichartiger Informationen und liefert das Ergebnis einem anderen Prozess, der dann von der Herkunft der Informationen meist nichts mehr wissen muss.

## **Benutzereingabe-Tasks**

---

Einen Sonderfall bildet die Mensch-/Maschineschnittstelle. Üblicherweise sind Menschen wesentlich langsamer als Computer, so dass wir die Eingabe durch Menschen in einer eigenen Benutzereingabe-Task behandeln können. Für Standard-Eingabegeräte wie Tastatur oder Maus werden die Kommunikationsmechanismen typischerweise vom Betriebssystem zur Verfügung gestellt. Die Task übernimmt dann hauptsächlich Plausibilitätsprüfungen der Eingaben, das Aufsammeln einer für die Anwendung relevanten Menge von Basiseingaben, um dann logische, überprüfte Datenstrukturen an die Anwendung weiterzuleiten. Für Spezialeingabegeräte müssen Sie die Kommunikation mit dem Gerät unter Umständen selbst erstellen. Nehmen Sie als Beispiel ein ganzes Feld von Knöpfen und Schaltern, für die die Benutzereingabe-Task dann die Abbildung des physikalischen Eingangssignals auf die Bedeutung dieser Knöpfe vornimmt.

## **Ausgabeentkopplungs-Tasks**

---

Sehr oft ist die wirkliche Weiterleitung eines von der Applikation produzierten Ergebnisses an ein Gerät oder Nachbarsystem eine komplexe Aufgabe. Befreien Sie Ihre Applikation davon und delegieren Sie diese Aufgabe an eine eigenständige Ausgabeentkopplungs-Task. Damit können Sie die Ergebnisberechnung und die Ergebnisweitergabe parallel laufen lassen. Während die Ausgabeentkopplungs-Task mit der Weiterleitung eines Ergebnisses beschäftigt ist, kann die Applikation bereits weitere Berechnungen vornehmen.

## **Ausgabekoordinations-Tasks**

---

Stellen Sie sich ein Nachbarsystem vor, das sinnvollerweise immer nur einen Auftrag gleichzeitig bearbeiten kann, z. B. einen Lautsprecher oder einen Drucker. Wenn zwei oder mehrere essenzielle Prozesse im gleichen Zeitraum auf das gleiche Gerät zugreifen oder an das gleiche Nachbarsystem senden wollen, sollten Sie eine Task einführen, die die Ausgaben koordiniert oder sequenzialisiert. Für zwei oder mehrere Audioquellen, die an den Lautsprecher wollen, muss eine Ausgabekoordinations-Task entscheiden, wer wann und wie lange Zugriff hat.

## Zeitgetriggerte Tasks

---

Für alle Systemprozesse, die durch Zeitereignisse ausgelöst werden, sollten Sie genauso eigenständige Tasks einrichten, wie für jeden extern getriggerten Prozess. Die Zeiteinplanung wird von einer Steuerungs-Task (siehe unten) überwacht. Diese setzt die Timer, die zeitgetriggerte Tasks dann rechtzeitig aktivieren.

Intern motivierte Tasks	Betrachten wir nun eine Menge von Tasks, die wir nicht aus dem Kontextdiagramm und der Use-Case-Analyse ableiten können. Alle diese Tasks kommen durch Refaktorisierung zustande, d. h. durch Designentscheidungen, die bestimmte Tätigkeiten aus den Systemprozessen herauslösen, um vorhandene Ressourcen besser zu nutzen oder Zeit zu sparen. Auch zum Finden dieser Taskarten haben Sie bereits Analyseergebnisse erzeugt, die Ihnen zur Taskstrukturierung Anhaltspunkte geben.
-------------------------	---

## Steuerungs-Tasks

---

Im Zuge unserer detaillierten Analyse haben wir Aktivitäten gefunden, die am besten durch Zustandsmodelle präzisiert werden konnten. Derartige Aktivitäten können als eigenständige Steuerungs-Tasks herausgelöst werden. Ihre Verantwortung ist die Koordination des Zusammenspiels zwischen anderen Tasks.

Ein weiteres Indiz für die Einführung von eigenen Steuerungs-Tasks sind Systemprozesse, die stark durch Vor- und Nachbedingungen miteinander verflochten sind. Nützen Sie Steuerungs-Tasks, um die Vorbedingungen zu überwachen und zum geeigneten Zeitpunkt andere Systemprozesse zu beauftragen.

Neben den Interrupt-Tasks sind Steuerungs-Tasks die zweite Art von Tasks, die meist mit sehr hoher Priorität ausgestattet sind, weil sie andere Tasks oft aus dringlichen Gründen unterbrechen oder abbrechen müssen.

## Periodische Tasks

---

Lösen Sie wiederkehrende, zyklische Berechnungen aus Prozessen als periodische Tasks heraus und aktivieren Sie diese durch einen Timer. Damit ersparen Sie sich lange Prozessblockaden. Dies ist sinnvoll, wenn die Zeitperiode zwischen den Berechnungen viel länger ist als die Zeit der Berechnung selbst. Der Prozess würde dann die Zeit zwischen den Berechnungen nur warten. Der logische Auslöser für derartige periodische Tasks kommt somit nicht aus der Umwelt, sondern aus der internen Ablauflogik eines Prozesses. Prüfen Sie dazu die Systemprozessbeschreibungen sorgfältig auf Sätze wie „Danach soll im Abstand von 100 ms ...“ oder „Der Prozess soll regelmäßig (alle 3 Minuten) ...“.

## Rechenintensive Tasks

Oft können Sie rechenintensive Aufgaben als eigenständige Tasks herauslösen und mit niedriger Priorität im Hintergrund laufen lassen, wann immer Zeit dafür ist. Dies optimiert die Ausnutzung vorhandener Ressourcen. Suchen Sie in Ihren Aktivitätsdiagrammen, Zustandsmodellen und Szenarien nach Aktivitäten und Aktionen, die zwar gemacht werden müssen, die aber nicht zeitkritisch sind.

## Service-Tasks

Als letzte Kategorie betrachten wir noch den unteren Bereich des logischen Architekturmusters, die Basissystemdienste. Selbstverständlich wird man Betriebssystemerweiterungen und höhere Dienste oft als eigenständige Service-Tasks laufen lassen. Da diese im Normalfall wichtige Basismechanismen für andere Tasks zur Verfügung stellen, werden sie oft hohe Priorität haben.

### 10.5.2 Taskstruktur überdenken

Wenn Sie mit der Anzahl von Tasks, die Sie nach den oben genannten Kriterien gebildet haben, in Hinblick auf Ihre Ressourcen zurechtkommen, dann sollten Sie diese logische Aufteilung beibehalten. Falls es doch zu viele Tasks geworden sind, stellen Sie zusätzlich folgende Überlegung an:

- Gibt es zeitgetriebene Tasks, die in gleichen Intervallen oder zu gleichen Zeitpunkten ausgeführt werden müssen? Dann können Sie diese vielleicht zu einer Task zusammenführen.
- Gibt es Tasks, die aus fachlichen oder logischen Gründen ohnehin nie zeitgleich laufen können, weil sie z. B. nur in unterschiedlichen Zuständen des Systems aktiv sein können oder zeitlich nacheinander ausgeführt werden müssen<sup>5</sup>? Auch dann können Sie diese Tasks zu einer Task zusammenfassen.
- Gibt es Tasks, die nicht unbedingt asynchron laufen müssen, sondern als Unterprogramm eines Hauptprozesses ausgeführt werden können? Dann sollten Sie diese in den Hauptprozess integrieren.

In [Gom00] finden Sie eine sehr ausführliche Behandlung von Kriterien für das Bündeln und Restrukturieren von Tasks.

---

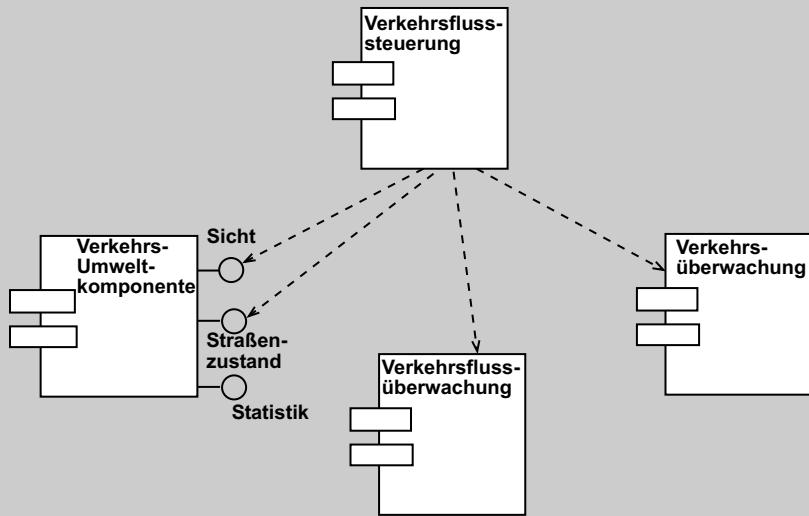
<sup>5</sup> Letzteres deutet auf einen Fehler in der Use-Case-Analyse hin. Sie hätten die Prozesse wahrscheinlich gar nicht trennen sollen. Aber gute agile Methoden sind selbstkorrigierend.

### 10.5.3 Taskabhängigkeiten modellieren

Sobald Sie Ihre wesentlichen Tasks gefunden haben, sollten Sie diese in einem separaten Taskingmodell zeichnen. Wir verwenden dazu Komponentendiagramme der UML, in denen wir die aktiven Komponenten (die Tasks) und die wichtigen passiven Komponenten (zur Taskkommunikation über gemeinsame Datenstrukturen) einzeichnen.

#### Komponentendiagramme

Ein Komponentendiagramm ist die graphische Darstellung der statischen Struktur der Architektur. Es zeigt im Wesentlichen die größeren, physikalischen Bausteine der Software (als Komponentensymbole) und ihre Abhängigkeiten (mit gestricheltem Pfeil). Zur Präzisierung können die Komponenten mit benannten Schnittstellen versehen werden (dargestellt als Lollipop-Symbol).



**Abbildung 10.6:** Beispiel eines Komponentendiagramms

Wir verwenden die Symbole für aktive und passive Komponenten aus Abschnitt 10.4 und zeichnen die Abhängigkeiten nach folgenden Regeln.

1. Passive Komponenten sind von den Tasks abhängig, die sie zur Kommunikation verwenden.
2. Tasks, die von anderen Tasks Nachrichten oder Ereignisse erhalten, sind von diesen abhängig. In anderen Worten: der Empfänger ist abhängig vom Sender. Auch bei Nachrichten mit Antworten zeichnen wir nur die Richtung der Nachricht als Abhängigkeit ein.
3. Für Tasks, die gegenseitig Nachrichten austauschen, sind beide Abhängigkeitspfeile einzulegen.

4. Wenn Sie die Nachbarsysteme und Eingabegeräte auch modellieren, sind
  - a) interne Tasks abhängig von aktiven Eingabegeräten,
  - b) passive Geräte abhängig von internen Tasks.
5. Ausgabeempfänger sind abhängig vom Sender.
6. Timer werden normalerweise nicht als eigene Komponenten modelliert.  
Daher entfallen auch Abhängigkeiten von Timern.

In Abbildung 10.7 sind die Fälle mit den Nummern der obigen Empfehlungsliste gekennzeichnet.

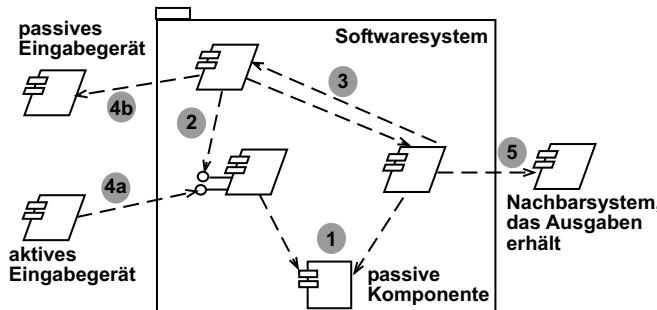


Abbildung 10.7: Tasking-Modell als Komponentendiagramm

## 10.6 Taskkommunikation

Parallel laufende Prozesse müssen hin und wieder ihre Aktivitäten synchronisieren und oft auch Nachrichten austauschen. Für diese Synchronisation und Kommunikation stehen verschiedene Möglichkeiten zur Verfügung, die in Abbildung 10.8 im Überblick dargestellt sind.

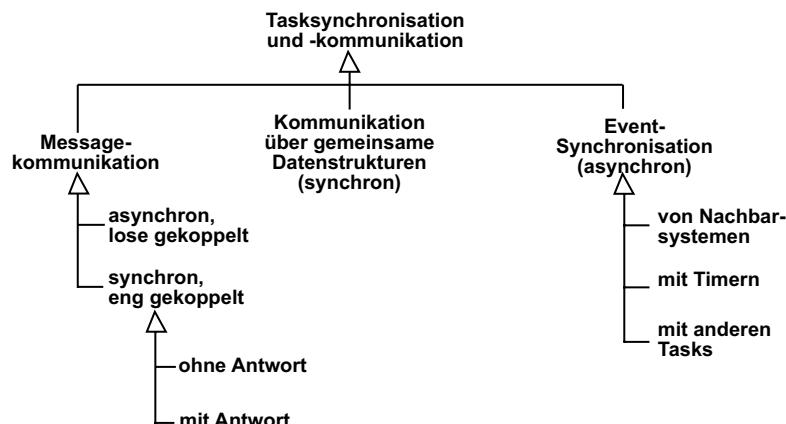


Abbildung 10.8: Synchronisations- und Kommunikationsmechanismen

Die einfachste Form ist die Ereignissynchronisation über Signale. Ein Prozess erhält ein Signal, das ihm mitteilt, dass ein Ereignis in der Systemumgebung stattgefunden hat oder von einem Timer oder einem anderen Prozess ausgelöst wurde. Der benachrichtigte Prozess kann dann in geeigneter Weise reagieren.

Wenn Sie einen Prozess neben dem Eintreten eines Ereignisses auch noch andere Informationen übermitteln lassen wollen, reichen Signale nicht aus. Dies ist immer dann der Fall, wenn ein Prozess Informationen produziert, die von einem anderen Prozess konsumiert werden sollen. Nutzen Sie dafür Prozesskommunikation in Form von Nachrichten (Messages). Sie sollten sich überlegen, wie sich die beiden Prozesse beim Nachrichtenaustausch verhalten sollen.

### Asynchrone Kommunikation

- Der Produzent schickt eine Nachricht an den Konsumenten und möchte sofort weiterarbeiten. Wir nennen solche Tasks lose gekoppelt. Verwenden Sie hierfür asynchrone Nachrichtenübermittlung.
- Der Produzent möchte seinen Ablauf mit dem des Konsumenten mehr oder weniger lang synchronisieren. Verwenden Sie synchrone Nachrichten mit folgenden zwei Spielarten:

### Synchrone Kommunikation ohne Antwort

- Synchrone Nachrichten ohne Antwort: Der Produzent schickt die Nachricht, der Konsument bestätigt das Erhalten sofort und gibt damit den Produzenten wieder frei. Dieser kann weiterarbeiten. Danach arbeitet der Konsument nach seinem Zeitplan die Nachricht ab.

### Synchrone Kommunikation mit Antwort

- Synchrone Nachrichten mit Antwort: in diesem Fall ist der Produzent nach dem Abschicken der Nachricht blockiert. Der Konsument führt die Reaktion auf die Nachricht aus und sendet danach erst die Antwort an den Produzenten zurück. Jetzt können beide Tasks wieder ihre eigenen Wege gehen. (Synchrone Kommunikation in Form von Calls ist natürlich auch die normale Art der Zusammenarbeit mit passiven Klassen: eine Operation wird aufgerufen; der Aufrufer wartet, bis die Operation fertig ausgeführt ist, bevor der Ablauf fortgesetzt wird.)

### Kommunikation über gemeinsame Datenstrukturen

- Zwei Prozesse können über eine gemeinsam bekannte passive Komponente kommunizieren, sofern sie sich auf dem gleichen Rechner befinden. Ein Prozess liefert dann die Daten bei der passiven Komponente ab, der andere Prozess holt sich diese Daten.

Die wesentlichen statischen Abhängigkeiten zwischen den Tasks haben wir schon beim Finden der Tasks modelliert. Betrachten Sie jetzt für jede Abhängigkeit die Kommunikationsbedürfnisse genau. Muss die andere Task nur wissen, dass etwas passiert ist? Dann reicht das Austauschen von Signalen. Müssen Daten übermittelt werden? Dann nutzen Sie eine der Formen der Message-Kommunikation, oder Sie entscheiden sich für gemeinsam genutzte Datenstrukturen.

Die UML bietet Ihnen mehrere Möglichkeiten, die Kommunikation zwischen Tasks grafisch darzustellen:

- Sequenzdiagramme,
- Kollaborationsdiagramme,
- kooperierende StateCharts.

Zusätzlich lässt sich das Senden und Empfangen von Signalen oder Messages natürlich auch in den Operationsbeschreibungen innerhalb von Tasks unterbringen.

## 10.6.1 Taskkommunikation mit Sequenzdiagrammen

Erstellen Sie speziell für die Taskkommunikation und Synchronisation eigene Sequenzdiagramme. Das, was innerhalb einer Task als sequenzieller Prozess abläuft, kann dann in einem separaten Diagramm dargestellt werden. Zeichnen Sie dazu die Tasks, deren Zusammenspiel Sie beispielhaft ausdrücken wollen, und alle beteiligten passiven Komponenten. Tasks werden in Sequenzdiagrammen grundsätzlich als dauernd aktiv (mit der Doppellinie) gezeichnet. Passive Komponenten sind nur aktiv, wenn sie synchrone Nachrichten erhalten und abarbeiten.

Sowohl in Sequenz- wie auch in Kollaborationsdiagrammen gilt die folgende Konvention für Pfeile:

```

asynchron ----->
synchron ----->
return (von Call) <-----

```

- Asynchrone Nachrichten erhalten eine einfache Pfeilspitze.
- Synchrone Nachrichten werden mit ausgefüllter Pfeilspitze gezeichnet und
- die Rückkehr von einem sequenziellen Aufruf als gestrichelte Linie.

In Abbildung 10.9 haben wir die verschiedenen Arten der Taskkommunikation beispielhaft dargestellt.

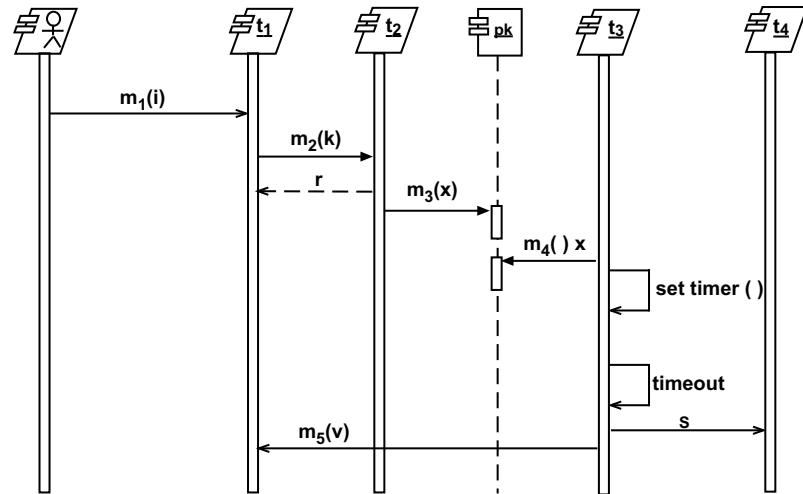


Abbildung 10.9: Taskkommunikation mit Sequenzdiagrammen

m<sub>1</sub> ist ein asynchroner Interrupt aus der Umwelt. m<sub>2</sub> kommuniziert synchron mit Task t<sub>2</sub> und wartet auf die Antwort r. Task t<sub>2</sub> und t<sub>3</sub> nutzen die passive Komponente pk für die Kommunikation miteinander. Task t<sub>3</sub> reagiert auf ein Timersignal, informiert Task t<sub>4</sub> über das Signal s und anschließend Task t<sub>1</sub> asynchron ohne Reply mit Nachricht m<sub>5</sub>.

Wenn Sie bei der Verfeinerung der Architektur jetzt genauer beschreiben wollen, wie Task t<sub>1</sub> auf den Eingang der Nachricht m<sub>1</sub> reagiert, so können Sie das Innenleben als neues Sequenzdiagramm, nun mit dem zentralen aktiven Objekt aot<sub>t1</sub> (fett umrandet) und passiven Objekten der Task darstellen. Beachten Sie, dass die gesamte Kommunikation innerhalb der Task mit synchronen Nachrichten, d. h. mit normalen Calls abläuft.

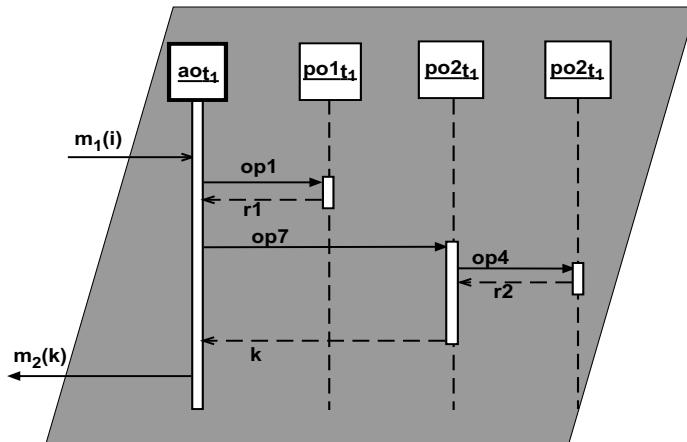
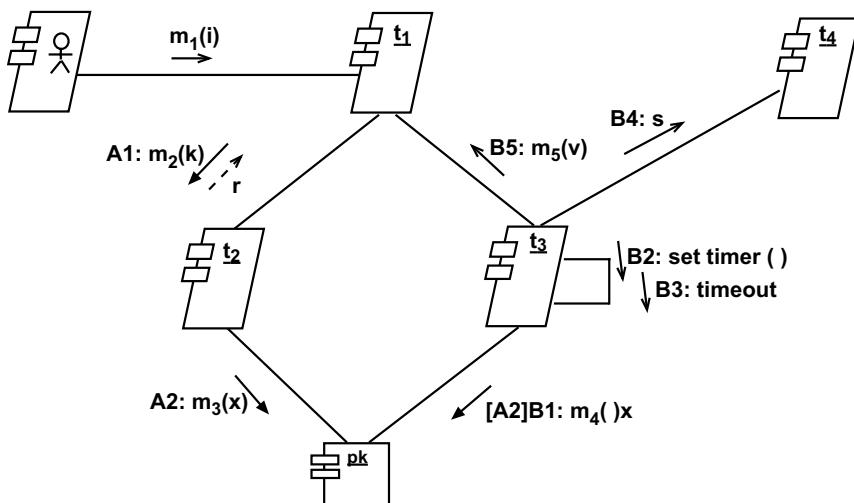


Abbildung 10.10: Das Innenleben von Task t1

## 10.6.2 Taskkommunikation mit Kollaborationsdiagrammen

Kollaborationsdiagramme sind – wie bereits erwähnt – nur eine andere syntaktische Variante der UML für beispielhafte Abläufe. Sie haben die freie Wahl zwischen Sequenz- und Kollaborationsdiagrammen, um die Kommunikation zwischen Tasks beispielhaft darzustellen.<sup>6</sup>

Abbildung 10.11 zeigt die Inhalte von Abbildung 10.9 in Form eines Kollaborationsdiagramms. Um die Reihenfolge der Nachrichten zu kennzeichnen, haben wir jetzt Nummern vor die Nachrichtennamen gestellt. Der Auslöser m1 bleibt unnummeriert. Danach sehen Sie die Nachrichten 1, 2, 3, ... Da in diesem Szenario zwei parallele Prozesse laufen, wurden die Nummern noch mit Kennbuchstaben für die Prozesse versehen: A1, A2 für den ersten Prozess, B1, B2, ... für den zweiten Prozess. Bei Schritt B1 sehen Sie noch eine Besonderheit: Der Konsumentenprozess B kann auf die passive Komponente pk erst zugreifen, wenn der Produzent seinen Schritt 2 ausgeführt hat. Deshalb ist die Wächterbedingung [A2] dem Prozessschritt B1 vorangestellt.



**Abbildung 10.11:** Taskkommunikation mit Kollaborationsdiagramm

<sup>6</sup> Viele Case-Tools nehmen Ihnen die Qual der Wahl dadurch ab, dass Sie die eine Diagrammform automatisch in die andere umwandeln können. Was auch immer Sie als erstes Szenario zeichnen – Sie können anschließend auch die andere Form sehen!

### 10.6.3 Taskkommunikation mit StateCharts

Viele der Tasks sind zustandsgetrieben. Oft werden Sie daher das Verhalten der Task mit Zustandsautomaten modellieren. Wie zeichnet man in diesem Diagramm die Kommunikation mit dem StateChart einer anderen Task ein?

Abbildung 10.12 zeigt den Fahrer und vier aktive Komponenten in der Auto-Software, die sich zur Synchronisation ihrer Leistung miteinander unterhalten müssen.

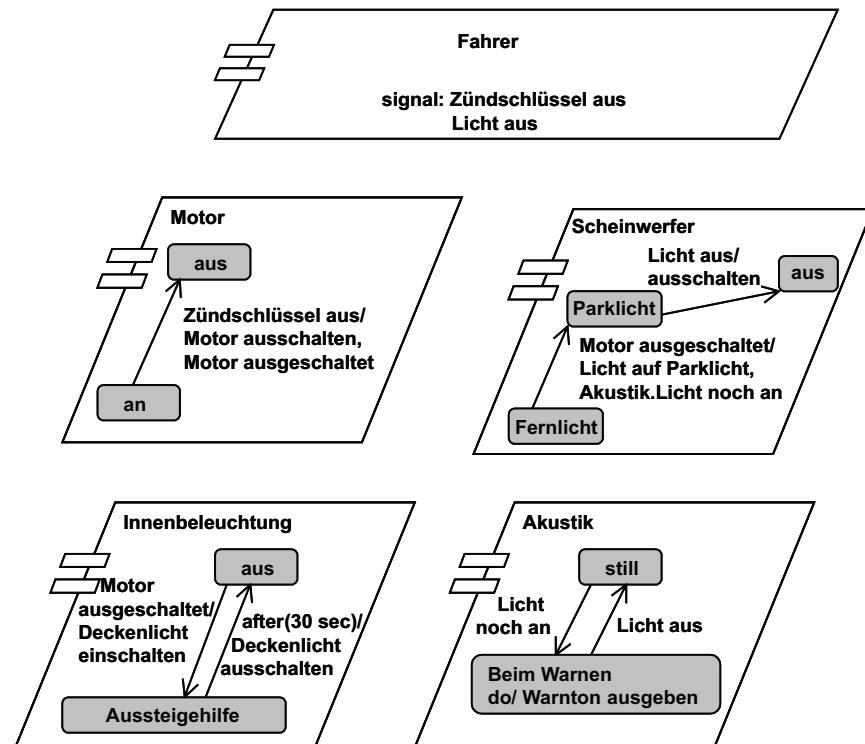


Abbildung 10.12: Kommunikationsdarstellung in StateCharts

Beginnen wir beim Motor, der auf das Ereignis „Zündschlüssel aus“ vom Fahrer eine taskeigene Aktion auslöst (Motor ausschalten) und dieses „Motor ausgeschaltet“ als Signal propagiert. Dieses Signal wird evtl. von zwei anderen Komponenten aufgegriffen, wenn sie im richtigen Zustand sind.

Wenn die Innenbeleuchtung aus ist, dann bewirkt das Signal den Wechsel in den Zustand „Aussteigehilfe“. Auch beim Scheinwerfer kommt es eventuell zu einem Zustandswechsel, wenn er noch auf „Fernlicht“ steht. Der Scheinwerfer schaltet dann zurück auf Parklicht und signalisiert der Task Akustik, dass das

Licht noch an ist. Diese würde darauf entsprechend reagieren und den Fahrer so lange akustisch warnen, bis er das Licht ausschaltet, was der Akustik-Task als externes Signal mitgeteilt wird.

In dem Beispiel wurden für die Kommunikation einfache Signale statt Messages benutzt. Diese Signale können auch Parameter bekommen, die vom Empfänger ausgewertet werden. Sie können aus Zustandsautomaten heraus auch synchrone Nachrichten verschicken. Dazu setzen Sie in den Aktionsteil einfach den Namen des Empfängers und die Operation ein, die Sie veranlassen wollen. Zum Beispiel kann der Zugriff auf ein passives Eingabegerät im Zustandsautomaten einer Applikation mit „Polling-Task.hole\_Wert“ notiert werden. Auch Multicast-Nachrichten an Gruppen von Empfängern und Broadcast-Nachrichten an das System als Black Box sind auslösbar.

## 10.7 Umsetzung der Kommunikation

Sie haben nun die unterschiedlichen Möglichkeiten der Taskkommunikation und ihre Darstellung mit den Ausdrucksmitteln der UML kennen gelernt. Wie aber lässt sich diese Kommunikation realisieren? Dafür stehen Ihnen vier Möglichkeiten zur Verfügung.

- Das Betriebssystem bietet Ihnen sicherlich den einen oder anderen Mechanismus zur Kommunikation zwischen Prozessen an (Events bei Windows, Signals bei Unix, ...). Diese Mechanismen sind aber oft sehr rudimentär.
- Einige Programmiersprachen, die parallele Prozesse direkt unterstützen (wie Ada oder Java) haben auch Kommunikationsmechanismen (wie das Ada-Rendezvous).
- Sie können einige Thread-Packages kaufen, die Ihnen als API entsprechende Aufrufe anbieten.
- Last but not least bleibt Ihnen noch die Möglichkeit, die benötigten Dienste selbst zu entwerfen und zu implementieren.

Die letzte Möglichkeit wollen wir noch etwas genauer betrachten, weil sie bei der Entwicklung von RTE-Systemen häufig genutzt wird. Gehen Sie dabei wie folgt vor:

- Lernen Sie die Möglichkeiten Ihrer Basissysteme kennen. Was bietet Ihr Betriebssystem? Welche Libraries oder Frameworks stehen Ihnen sonst zur Verfügung?
- Überlegen Sie sich, welche Arten der Kommunikation Sie brauchen. Schreiben Sie eine Liste der höheren Kommunikationsoperationen, die Sie gerne anwenden würden.

- Modellieren Sie dann in einem Basissystemdienst diese Operationen in Form von Szenarien (z. B. mit Sequenzdiagrammen) für einen Stellvertreter-Sender und einen Stellvertreter-Empfänger bis auf die Ebene der vorhandenen Basisoperationen.
- Danach können Sie unter Nutzung dieser „höheren“ Kommunikationsmechanismen Taskdiagramme zeichnen. Der Weg, wie die Kommunikation auf Ihrem Betriebssystem wirklich abgewickelt wird, ist durch Ihren Basissystemdienst dann ein für alle Mal vorgegeben.<sup>7</sup>

Betrachten wir diese Vorgehensweise am Beispiel eines Kommunikations-Patterns. Nehmen Sie an, Sie haben n Tasks, die ein Ereignis signalisieren können, und m andere Tasks, die bei Eintreten dieses Ereignisses darauf reagieren sollen. Dann bietet sich zur Entkopplung der m:n-Kommunikation an, einen Vermittler oder Agenten einzuschalten. Die Sender teilen dem Vermittler durch die Operation offer() mit, welche Signale sie liefern können. Die Empfänger teilen dem Vermittler durch die Operation request() mit, auf welche Signale sie reagieren wollen. Sender und Empfänger können sich über die Operationen cancel() und derequest() beim Vermittler auch wieder abmelden. Wenn ein Sender das Signal s an alle Interessierten weiterleiten möchte, verwendet er dazu die Operation notify(). Aufgabe des Agenten ist es, alle, die sich angemeldet haben, über das Eintreffen des Signals s zu informieren.

Modellieren Sie dieses Verhalten als Basissystemdienst unter Nutzung primitiver Signalkommunikation einmal vollständig. Abbildung 10.13 zeigt das Modell in der unteren Hälfte<sup>8</sup>. Wenn Sie das als bekannt voraussetzen, dann können Sie diese Form der Taskkommunikation auf der essenziellen Ebene einfach durch ein direktes notify() darstellen, obwohl der Mechanismus dahinter wesentlich komplexere Schritte erfordert.

Mit dieser Vorgehensweise können Sie beliebig viele komplexe, abstrakte Schichten in Ihre technische Architektur einführen und so komplizierte Protokolle modellieren. Qualitätssicherer werden die Details einmal genauestens prüfen. Danach können Entwickler die abstrakte Form der Kommunikation anwenden, ohne sich über die technologische Lösung dahinter Gedanken machen zu müssen.

---

<sup>7</sup> Mit einem flexibel anpassbaren Codegenerator in Case-Tools lässt sich dieser Weg auch automatisieren. (Vgl. dazu auch Kapitel 11)

<sup>8</sup> Um den Mechanismus vollständig zu erläutern, müssen Sie einige mögliche Szenarien zwischen Sender, Empfänger und Agent modellieren. Wir haben die Zusammenarbeit in dem Diagramm nur angedeutet.

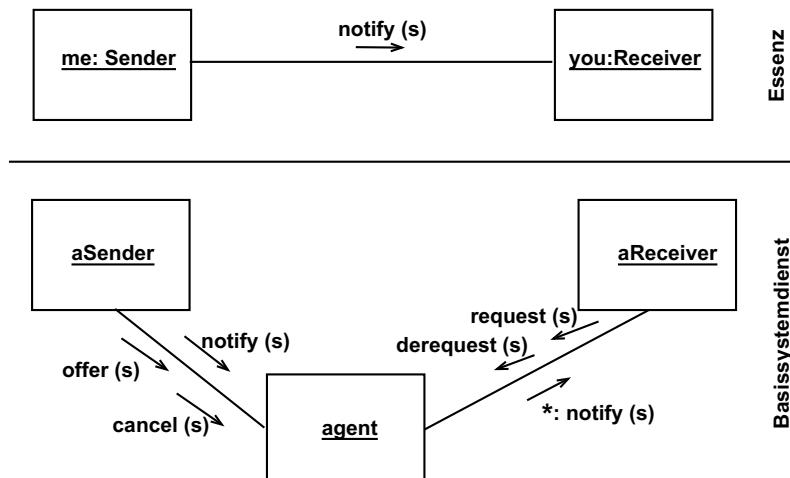


Abbildung 10.13: Notify-Mechanismus zur Optimierung einer m:n-Kommunikation

## 10.8 Hilfsmittel für Architekten

Je spezieller Ihre Randbedingungen sind, je exotischer Ihr Betriebssystem, Ihre Programmiersprache und Ihre Hardware sind, desto mehr technologische Klassen müssen Sie selbst rund um Ihren essenziellen Kern entwickeln. In den letzten Jahren entstanden aber immer mehr Hilfsmittel, die den Architekten die Arbeit erleichtern.

- Für typische, wiederkehrende Designprobleme im Kleinen finden Sie in der Patternwelt hervorragend dokumentierte und bewährte Lösungsideen. Sie sollten zumindest die Patterns von [Gam95] kennen und einsetzen.
- Für einige Bereiche werden fertige Frameworks angeboten, die Ihnen Standardkommunikationsdienste, Persistenz, Exception Handling oder Garbage Collection bereit stellen. Sehen Sie sich um, seien Sie aber noch vorsichtig. Manche dieser Frameworks sind noch nicht ausgereift, andere enthalten versteckte Ressourcenfresser, die vielleicht mit Ihren Effizienzanforderungen nicht in Einklang zu bringen sind.
- Im Bereich von RTE-Systemen entstehen derzeit ebenfalls eine Menge von Architektur- und Designpatterns. Lesen Sie die Ideen über Embedded Middleware in [POSA00] und surfen Sie regelmäßig ausgehend von [www.hillside.net/patterns](http://www.hillside.net/patterns), einem seit vielen Jahren existierenden, guten Einstiegspunkt in die Patternwelt.

## 10.9 Ist Ihre Architektur zukunftssicher?

In diesem Kapitel haben wir Strukturvorschläge und Modellierungsvorschläge für Ihre Softwarearchitektur diskutiert. Betrachten Sie Ihr Ergebnis kritisch:

- Trennt Ihre Architektur die fachlichen Klassen sauber von den technologischen Klassen (soweit Randbedingungen dies zulassen)?
- Ist Ihre Taskstruktur ein möglichst getreues Abbild der natürlichen Parallelität Ihrer Systemprozesse, gepaart mit den notwendigen Kompromissen, die Ihre Ressourcen erzwingen?
- Haben Sie alle Technologieabhängigkeiten gekapselt, so dass Sie Änderungen und Verbesserungen sehr lokal vornehmen können?
- Nutzen Sie UML-Diagramme auf jeder Abstraktionsebene – von der Verteilung über das Tasking bis zum Innenleben von Komponenten – effizient?

„Perfektion ist ein Ergebnis von kleinen Dingen,  
und Perfektion ist kein kleines Ding.“

*Michelangelo (1475–1564)*

# Teil IV

## Weitere Aktivitäten

---

Da Ihr Kunde sicherlich etwas mehr als Papier und Modelle erwartet, stehen Sie nun vor der Herausforderung Ihr System zu Implementieren, zu Testen und eine Abnahme gegen die Spezifikation durchzuführen. Kapitel 11 gibt hier einige für RTE-Systeme wichtige Hinweise.

Sollten Sie mit einem wirklich großen RTE-System konfrontiert sein, so empfiehlt Ihnen Kapitel 12 ein mehrschichtiges Vorgehen, mit dem Sie selbst umfangreichste Produkt- und Systementwicklungen managen können.



„Erst die Anwendung des Wissens, seine Realisierung, seine Durchsetzung und Weiterentwicklung bringen Resultate und Fortschritt – also:  
**Wissen ist Schlaf, Realisieren ist Macht.**“

*Reinhold Würth, 1993*

# 11

## **Konstruktion, Test und Abnahme**

---

### **Fragen, die dieses Kapitel beantwortet:**

- Wie komme ich von den Modellen zum Code?
- Wie erstelle ich aus Software- und Hardwarekomponenten mein Gesamtsystem/Produkt?
- Wie kann ich die richtige Funktionsweise der Software und des Gesamtsystems/Produkts überprüfen?

In diesem Kapitel werden Sie einen Überblick über weitere Schritte erhalten, die Sie bis zur Fertigstellung Ihres Produkts/Systems durchführen müssen. Implementierung, Test und Inbetriebnahme Ihres Systems oder Produkts sind aus mehreren Gründen allerdings nicht der Hauptfokus dieses Buches.

- Es gibt eine Menge von Büchern, die sich speziell mit diesen Themen befassen.
- Die Art der Implementierung ist sehr sprachspezifisch, im Detail auch compilerspezifisch.
- Mehr und mehr Arbeit in diesem Bereich wird von speziellen Code-Generatoren übernommen, die kontinuierlich bessere Ergebnisse erzielen.

Zudem haben Sie sich durch die vorhergehenden Aktivitäten eine gute Ausgangsbasis geschaffen. Wenn Sie es bis hierher geschafft haben, müssen Sie das System/Produkt „nur“ noch bauen.

„Test before Code“	Noch vor der Erstellung des Codes sollten Sie Ihre Testfälle erstellen. XP und viele moderne Vorgehensweisen haben das Prinzip „Test before Code“ in den letzten Jahren (wieder) populär gemacht. Wenn Sie Ihr Requirements- und Architekturgehirn mit Struktur und Verhalten füllen, sollten Sie parallel auch die dafür nötigen Testszenarien und -daten generieren. Hierbei können Ihnen z. B. die Sequenz- oder Kollaborationsdiagramme oder auch einfach textuelle Abnahmeszenarien wertvolle Dienste leisten.
Reihenfolge bei der Implementierung	Auch Implementierung, Test und Abnahme müssen nicht unbedingt in dieser Reihenfolge ablaufen. Im Rahmen einer iterativen Entwicklung werden Sie die drei Schritte auf beliebig fein aufgeteilte Teilsysteme anwenden. So können Sie zum Beispiel die Realisierung für den Teil des Systems starten, der als erstes hinreichend gut spezifiziert ist. Sukzessive, parallel zur fortschreitenden Spezifikation, können weitere Teile realisiert, getestet und kontinuierlich integriert werden.
Integrationsplan frühzeitig erstellen	Wenn Sie in Kapitel 6 festgestellt haben, dass Sie einen sinnvollen Teil Ihres Systems als eigenständiges Produkt/System liefern können, ist es sinnvoll, mit diesem Teil zu beginnen. Sie erstellen und integrieren dann zuerst die Softwarekomponenten, die für dieses Teilsystem nötig sind. Im weiteren Text werden wir zwar nur von der Erstellung des „Systems“ sprechen, damit aber trotzdem auch jedes Teilsystem meinen.  Die Implementierungsreihenfolge wird sehr stark durch die Randbedingungen der Integration beeinflusst. Einige Basiskomponenten (z. B. Betriebssystemkomponenten) sind für die Integration und den Test unabdingbar und sollten dementsprechend frühzeitig zur Verfügung stehen.  Im Gegensatz zu den Schritten, die Sie bisher in diesem Buch kennen gelernt haben, können Sie die Erstellung, den Test und die Integration nicht auslassen (außer Sie haben einen Code-Generator, der Ihnen diese Aufgabe vollständig abnimmt). Nur wenn Sie Software- oder Hardwarekomponenten zukaufen, ersparen Sie sich die Erstellung. Den Testumfang der Software sollten Sie in Abhängigkeit vom Risiko eines Fehlverhaltens festlegen. Die Integration bleibt Ihnen auf keinen Fall erspart.  Wenn Ihnen für die Erstellung des Systems nur wenige der Ergebnisse aus den bisherigen Kapiteln vorliegen, müssen Sie die fehlenden Teile an dieser Stelle – wenn auch nicht so ausführlich – nachholen. Wenn Sie das System dagegen bis in die kleinsten Details modelliert haben, werden Sie das entstandene Modell fast direkt in Code umsetzen können.

### 11.1 Software schreiben

Besonderheiten der RTE-Implementierung	Die Programmierung von RTE-Systemen weist einige Besonderheiten auf. Die augenfälligste ist, dass Sie häufig nicht auf der Zielplattform entwickeln können. Sie werden die Software wahrscheinlich auf einem PC mit Hilfe einer
--	---

Entwicklungsumgebung für Ihre Zielplattform schreiben und einen entsprechenden Cross-Compiler einsetzen. Weitere Unterschiede, die große Auswirkung auf die Codierung haben, sind z. B. eingeschränkte Ressourcen wie Prozessorleistung, Speicher, Batterieleistung oder ein Echtzeit-Betriebssystem mit eingeschränktem Befehlssatz, reduziertem API und minimierten Bibliotheken.

Vermutlich haben auch Sie die wirklich erfahrenen RTE-Implementierer nicht in rauen Mengen im Unternehmen sitzen. Leider ist auch die Literatur zu diesem Thema nicht gerade üppig<sup>1</sup>. Da die Implementierer auf sehr spezifische Hardware, wenig standardisierte Betriebssysteme und eine Vielzahl von speziellen Treibern zugreifen müssen ist es schwierig, die Tipps und Tricks dafür zu verallgemeinern. Deshalb können Neueinsteiger das Programmieren von RTE-Systemen auch nicht aus der Literatur lernen.

Um einen guten Know-how-Transfer von erfahrenen RTE-Implementierern zu Neueinsteigern zu sichern, sollten Sie Ihr Team deshalb nach dem alten Handwerksprinzip organisieren. Die „Lehrlinge“ lernen von den „Meistern“ im Projekt die Tricks, die jene aus Erfahrung kennen. Die „Lehrlinge“ bringen dafür frisches Technologie-Know-how ein und sichern damit die Zukunft. Ob Sie dieses Prinzip ähnlich wie bei eXtreme Programming dadurch lösen, dass immer zwei Personen an einem Rechner sitzen (pair programming), oder deutlich offener gestalten, müssen Sie im Rahmen Ihrer Unternehmenskultur entscheiden.

Da inzwischen auch einige RTE-Systeme Java-Code-Bestandteile enthalten, hier ein kleiner Ausflug in die Java-Welt. In der kommerziellen Welt ist im Java-Umfeld das Prinzip „Write-Once-Run-Everywhere“ verbreitet und funktioniert weitestgehend. Auch wenn man Ihnen als Entwickler eines RTE-Systems glauben machen möchte, dass dies für Ihre Umwelt ebenso zutrifft, warnen wir vor dieser Illusion. Gerade in Ihrem Bereich, wo weniger verbreitete Echtzeit-Betriebssysteme und Hardwareplattformen zum Einsatz kommen, sind die virtuellen Maschinen im Detail alles andere als ausgereift und standardkonform, sofern es sie überhaupt gibt.

Im RTE-Umfeld lautet das Prinzip eher „Write-Once-Test-Everywhere“. D. h. Sie schreiben erst einmal eine Software einheitlich für alle potenziellen Zielumgebungen. Danach müssen Sie sie aber auf all diesen Zielplattformen testen, weil der Teufel im Detail steckt. Vielleicht ist das Zeitverhalten auf der zweiten Maschine doch geringfügig anders. Oder die floating-point-Arithmetik spielt Ihnen einen Streich bei kritischen Regelungsalgorithmen. Sie müssen also auch die Funktionalität noch einmal vollständig testen, um damit nachzuweisen, dass Ihr System auf der speziellen virtuellen Maschine wirklich funktioniert. Wenn sich Änderungen auf einer speziellen Zielplattform ergeben, dann sollten Sie versuchen, Ihr Design so anzupassen, dass diese Änderungen in speziellen Komponenten gekapselt werden.

Lehrlinge und Meister

Java: „Write-Once-Test-Everywhere“

---

<sup>1</sup> Wenn Sie nach Büchern für Ihre Implementierer suchen, dann sind [Barr99] und [Sim99] gute Empfehlungen.

Code-Generierung – mit Vorsicht genießen	Unter Umständen können Sie beim Codieren auf Code-Generatoren der Werkzeuge zurückgreifen, die Sie für die Modellierung benutzt haben. Die Unterstützung reicht dabei von der Generierung des Codes für die Klassenstruktur Ihres Modells bis zur automatisierten Erzeugung von Code aus Zustandsautomaten und der Erzeugung von Testszenarien aus Sequenzdiagrammen.
Generatoren anpassen	Verwechseln Sie bei der Code-Generierung Quantität nicht mit Qualität. Häufig ist der erzeugte Code der „Standard-Code-Generatoren“, die mit den vielen Modellierungstools ausgeliefert werden, zwar sehr umfangreich, aber nicht unbedingt effizient. Fast alle Tools bieten jedoch die Möglichkeit, die Code-Generierung z. B. über Skripte oder Templates zu beeinflussen. Sie können Ihre Technologie- und Architekturentscheidungen als Basis für eine maßgeschneiderte Umsetzung in Sourcecode hinterlegen. Sofern Sie über einen längeren Zeitraum zu diesen Architekturentscheidungen stehen, wird Ihr Code-Generator für diese eine sehr effiziente Umsetzung liefern. Einmal konfiguriert, erspart Ihnen ein Code-Generator zumindest für die Teile Ihres Systems, die nicht zeit- oder sicherheitskritisch sind – und das sind meist große Teile des Systems – eine Menge Arbeit.
Modelle aktualisieren	Andererseits werden Sie dadurch gezwungen, Ihre Modelle stets aktuell zu halten. Insbesondere veraltete oder gar überflüssige Diagramme werden häufig (unbeabsichtigt) in „toten“ Code umgesetzt. Dieser Code verlängert oder verhindert sogar den Kompiliervorgang, verwirrt Ihre Tester und belastet zudem die Speicherumgebung des Laufzeitsystems.  Auch bei der Implementierung lernen Sie ständig dazu. Lassen Sie Ihre Erfahrungen bezüglich der Fachlogik in die dafür relevanten UML-Diagramme mit einfließen, sofern sie für Änderungen oder Weiterentwicklungen des Systems relevant sind. Überlegen Sie sich genau, welche Modelle für wen welchen Wert haben. Nur diese Modelle sollten Sie (im geeigneten Maße und zu geeigneten Zeitpunkten) pflegen und aktualisieren.

- Die bestehenden Sequenz- und Kollaborationsdiagramme übergeben Sie an die Tester (auch wenn die erste Version von den Entwicklern erstellt wurde). Die Tester benötigen Testszenarien für die Regressionstests. Die Sequenz- und Kollaborationsdiagramme bieten hierfür eine optimale Basis und werden somit aktuell gehalten.
- Die Zustandsautomaten sollten Sie auf jeden Fall pflegen. Insbesondere, wenn sie für die Code-Generierung verwendet werden, müssen dort kontinuierlich Änderungen und Optimierungen einfließen.
- Aus dem Klassendiagramm wird meist der Code-Rahmen generiert. Somit sind auch sie ein Kandidat für die Aktualisierung.
- Use-Case-Diagramme und -Beschreibungen, sowie Aktivitätsdiagramme, sollten Sie nur dann kontinuierlich anpassen, wenn Sie sie für die Abnahme des Systems durch die Stakeholder nutzen. Erfahrungsgemäß treten hier selten Änderungen auf, da sie sich auf einem abstrakten Niveau befinden, essenziell und technologienutral sind.

- Auch wenn Sie die Use-Cases und deren Beschreibung sowie die Aktivitätsdiagramme nicht zur Testunterstützung herangezogen haben, sollten Sie spätestens am Ende Ihrer Systementwicklung eine Aktualisierung in Erwägung ziehen. Aktuelle Use-Cases sind für Ihre Nachwelt ein gut lesbarer, informeller Einstieg für die nächste Produktversion.
- Häufig in Verbindung mit der Code-Generierung gewinnen die Komponentendiagramme bzw. Paketdiagramme an Wert. Sie legen die Schnittstellen fest, die eine potenzielle Fehlerquelle bei der Programmierung darstellen. Daran vorgenommene Änderungen müssen sofort dem Programmierteam mitgeteilt werden.
- Pakete bilden in der Programmierung logische Einheiten, die unter Umständen von unterschiedlichen Seiten genutzt werden. Gerade bei Verwendung von Code-Generatoren erweist sich ein Paketdiagramm mit eingezeichneten Abhängigkeiten der Pakete als sinnvoll. Dadurch können Kompilier- bzw. Bindereihenfolgen bestimmt werden. Auch hier gilt: Halten Sie die Paketdiagramme stets aktuell und vermeiden Sie überflüssige oder redundante Abhängigkeiten.
- Beurteilen Sie für alle anderen Diagramme immer erst, ob Sie sich oder der Nachwelt wirklich einen Gefallen erweisen, wenn Sie sie aktualisieren. Gibt es jemanden, der diese Diagramme benötigt? Steht der Nutzen in einem guten Verhältnis zum Pflegeaufwand? Sparen Sie sich Zeit und Geld für die Pflege, wenn niemand wirklich Nutzen aus den Modellen zieht. Im Gegenteil, vernichten Sie die Dokumente oder machen Sie eindeutig kenntlich, dass sie potenziell veraltet sind. Somit führen Sie keinen Leser in die Irre.

Behalten Sie zudem die Anwenderanforderungen im Auge. Auch wenn Sie bereits Teile Ihres Systems implementieren, kann Ihr Anwender neue Erkenntnisse gewinnen, seine Meinung ändern oder Prioritäten verschieben. Nur eine gute Release-Politik kann Sie davor bewahren, mit „beweglichen Zielen“ zu arbeiten.

Anforderungen  
im Auge  
behalten

## 11.2 Code validieren

Nachdem Sie den Code erstellt haben, sollten Sie ihn gegenüber der Spezifikation auf Korrektheit überprüfen (validieren). Dabei helfen Ihnen Reviews, Walkthroughs und Inspektionen. Gewissheit über die Korrektheit erhalten Sie allerdings nur mit Hilfe von mathematischen Korrektheitsbeweisen oder symbolischer Programmausführung.

Reviews,  
Walkthroughs,  
Inspektionen

Diese Techniken sollten Sie nur auf einzelne Komponenten anwenden, wenn Ihr System extrem sicherheitskritisch ist. Analysieren Sie zunächst die Kritikalität einzelner Module und investieren Sie danach gezielt den Aufwand, eine mathematisch exakt festgelegte Spezifikation der Komponente zu erstellen, um damit einen derartigen Beweis durchzuführen.

## **11.3 Testen**

Testen Sie  
iterativ

Starten Sie sofort mit dem Testen Ihrer Software, sobald relevante Teile vorliegen!

Überdenken Sie auch die bei Ihnen vielleicht üblichen Teststrategien. Die Modelle, die bisher Ihr Vorgehen strukturiert und vorangetrieben haben, stehen Ihnen nun auch für den Test zur Verfügung. Nutzen Sie diese Mittel. Sollten Sie bisher sehr stark Use-Case-getrieben gearbeitet haben, eignet sich dieser Ansatz für die anstehenden Black-Box-Tests. Wenn Sie Szenarien in Form von Sequenzdiagrammen verwendet haben, nutzen Sie auch diese für den Test.

### **11.3.1 Software-Units testen und integrieren**

Emulation der  
Zielhardware

Software für RTE-Systeme wird meist auf einer Entwicklungsumgebung erstellt, die von der Laufzeitumgebung abweicht. Da Software- und Hardwareentwicklung parallel laufen, kann es passieren, dass die Hardware noch nicht für einen Test der Software-Units zur Verfügung steht. Auch bei verfügbarer Hardware kann es dort, z. B. auf Grund minimalen Speichers, schwierig sein, automatisierte Tests durchzuführen. Auf der Softwareebene sind automatische Tests aber unbedingt zu empfehlen.

Können automatische Tests auf der Zielhardware nicht realisiert werden, sollten Sie eine Emulation der Zielhardware einsetzen. Sie treiben dadurch zwar zunächst höheren Aufwand, da Sie Treiber und Stubs erstellen müssen, sparen sich allerdings bei der Testdurchführung (vor allem bei wiederholten Tests) viel Arbeit, so dass sich der Mehraufwand lohnt.

In Ihrer Testumgebung testen Sie den Code in einem White-Box-Test gemäß eines vorgegebenen Überdeckungsmaßes (z. B. 95% Anweisungsüberdeckung). Lassen Sie alle Testfälle nach jeder Code-Änderung erneut laufen, um alle Auswirkungen einer Änderung überprüfen zu können (Regressionstests).

### **11.3.2 Komponenten testen**

Beim Komponententest führen Sie einen Black-Box-Test aus und vergleichen das tatsächliche Verhalten einer Einheit mit dem erwarteten Verhalten.

Für zugekaufte Komponenten, die Sie nicht selbst erstellt haben, sollten Sie beim Kauf den Nachweis eines solchen Tests fordern, um die Qualität Ihres Systems sicherstellen zu können.

### 11.3.3 Software integrieren

Sobald mehrere Softwarekomponenten vorliegen, die gemäß Ihres Softwarearchitektur-Modells voneinander abhängen, können Sie integrieren.

Integrieren Sie die Komponenten inkrementell (funktionsorientierte Integration oder Integration nach Verfügbarkeit). Sie erhalten somit ein sehr genaues Bild, welche der Komponenten im Zusammenspiel Probleme verursachen. Testen Sie nach jedem Integrationsschritt das entstandene Teilsystem an den Schnittstellen der Komponenten, um die Zusammenarbeit der Komponenten zu überprüfen.

Inkrementelle  
Integration

Spätestens nach der Integration aller Komponenten zum fertigen Softwaresystem führen Sie einen weiteren Black-Box-Test durch, um die Software gegen ihre Spezifikation zu testen.

### 11.3.4 System integrieren

Sobald alle Bestandteile Ihres Systems vorliegen, können Sie Hard- bzw. Software und andere Bestandteile integrieren und erhalten damit hoffentlich Ihr Ihr funktionsfähiges Produkt/System.

Wenn Sie für die Tests Ihrer Software einen Emulator benutzt haben, können Sie nach der Systemintegration zum ersten Mal die Treiberschicht Ihrer Software testen, die für die Ansteuerung der Hardware zuständig ist. Benutzen Sie ein Oszilloskop oder einen Logic- bzw. Protocol-Analyzer, um die korrekte Umsetzung der Steuersignale der Software auf der Hardware zu überprüfen.

Treiberschicht  
testen

Zuletzt sollten Sie das integrierte System gegen die Spezifikation testen. Häufig stellt das Protokollieren der Ergebnisse auf der System-/Produktebene ein Problem dar. Ebenso sind automatische Tests schwer umsetzbar. Z. B. lässt sich das Ergebnis, dass der CD-Player das CD-Fach öffnet und die CD ausschiebt, nicht so einfach automatisch überwachen, wie sich ein Softwareereignis überwachen lässt.

## 11.4 Abnahme

Sobald Sie Ihr Produkt/System fertiggestellt haben und durch die Tests von der Funktionsfähigkeit überzeugt sind, können Sie es an den Kunden ausliefern.

Nutzen Sie für die Abnahmetests mit den Stakeholdern die Use-Case-Spezifikation und die dazu erstellten Verfeinerungen, wie Aktivitätsdiagramme und Zustandsautomaten, die einen Prozess darstellen. Die meisten Stakeholder begrüßen ein derartig Use-Case-getriebenes Vorgehen bei der Abnahme, da sie dann das System in einer ähnlichen Weise abnehmen können, wie sie es später im Betrieb benutzen werden.

## **11.5 Haben Sie Grund zum Feiern?**

In diesem Kapitel haben wir die letzten Schritte zum fertigen Produkt diskutiert. Betrachten Sie Ihr bisher erreichtes Ergebnis:

- Haben Sie die Software gemäß den fachlichen, technischen und architekturbezogenen Vorgaben erstellt?
- Haben Sie das Zusammenspiel aller Softwarekomponenten getestet?
- Erfüllt Ihr System die fachlichen, technischen und architekturbezogenen Anforderungen?
- Haben Sie für Ihre Programmier- und Testumgebung schon Hilfsmittel im Einsatz?
- Wissen Sie über die Stärken und Schwächen der am Markt erhältlichen Tools Bescheid?
- Lesen Ihre Entwickler Fachliteratur zum Thema Codieren, Testen und Integrieren von Systemen?

„Wenn Sie sich verwirrt fühlen,  
fassen Sie sich ein Herz:  
Sie sind nur in Kontakt mit der Wirklichkeit.“

*Dean Rusk (1909–1994),  
amerik. Politiker, 1961–1969 Außenminister*

# 12

## **Die Entwicklung großer Systeme**

---

### **Fragen, die dieses Kapitel beantwortet:**

- Lassen sich die Prinzipien dieses Buches auch auf sehr komplexe Projekte anwenden?
- Was ändert sich, wenn Dutzende oder Hunderte von Personen über Jahre an großen Systemen arbeiten?
- Wie wendet man agile Methoden an, wenn das System bereits im Einsatz ist und nur Teilsysteme verbessert werden sollen?
- Wie organisiert man die Informationsflut bei großen Systemen und lang laufenden Produktentwicklungen?

In Kapitel 2 haben wir Ihnen einen Entwicklungsprozess für RTE-Systeme vorgestellt. Betrachten Sie nochmals die Abbildungen 2.1 und 2.3. Sie führten in das Grundkonzept unseres agilen Vorgehensmodells ein: Die Aufteilung in die Systemebene und die Technologieebene (insbesondere die Softwareebene).

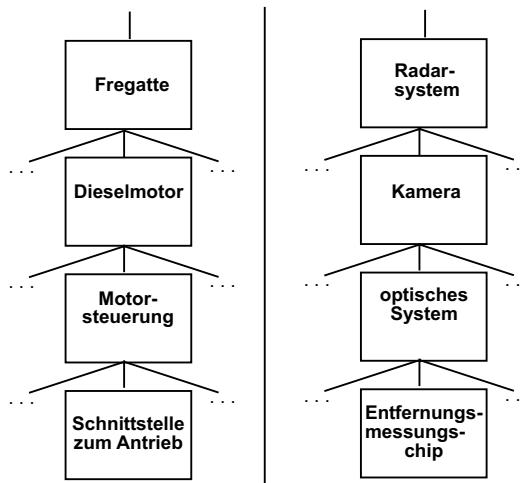
Nun, da Sie es bis hierher im Buch geschafft haben, wollen wir gestehen, dass das noch nicht die ganze Wahrheit über die Entwicklung von RTE-Systemen war. Vor allem, wenn die Systeme wirklich groß sind. Aber zunächst die gute Nachricht. Bei vielen Projekten reicht die Betrachtung dieser beiden Ebenen vollkommen aus, um erfolgreich Produkte und Systeme zu entwickeln. Die zweischichtige Betrachtung lehrt Sie als Softwareentwickler, sich nicht allzu wichtig zu nehmen. Sie schärft Ihren Blick für die wahren Kundenwünsche – für einsetzbare Produkte und Systeme, die aus Hardware, Software, und eventuell auch noch anderen Technologien bestehen.

Diesen zweistufigen Prozess anzuwenden, ist daher schon wesentlich besser, als sich sofort auf die Software zu stürzen, wie es die meisten Vorgehensmodelle im Umfeld der UML vorschlagen. Wenn Sie allerdings sehr große Systeme (z. B. einen Helikopter, ein internationales Flugsicherungssystem, die International Space Station (ISS), ...) entwickeln, müssen wir die Herangehensweise noch etwas komplizierter gestalten.

## 12.1 Systeme und Subsysteme

Auch der Entwicklung sehr großer Systeme liegt ein sehr einfacher Gedanke zugrunde: Jedes System ist Teil eines nächst größeren Systems. Versetzen Sie sich in die Lage einer jungen Entwicklungsingenieurin, deren Aufgabe es für die nächsten Monate ist, mit zwei Kollegen die Schnittstelle zur Antriebswelle neu zu entwickeln. Ihr fertiges System ist Teil einer Motorensteuerung. Diese ist Teil eines Schiffs diesels, der wiederum in eine Fregatte eingebaut ist. Auf jeder Ebene gibt es neben dem hier betrachteten System „gleichrangige“ Nachbarsysteme – Schwestern und Brüder, wenn Sie diese Ideen auf einen Stammbaum umsetzen. Abbildung 12.1 greift als zweites Beispiel ein System aus einer anderen Branche auf. Jetzt ist unsere Entwicklungsingenieurin verantwortlich für die Entwicklung eines hochpräzisen Kamerasystems. Ihr System wird aber nicht direkt verkauft, sondern als Teil eines Radarüberwachungssystems.

Drehen wir den Gedanken einmal um: So wie jedes System Teil eines nächst größeren Systems ist, so lässt sich jedes System auch in Teilsysteme zerlegen. Das Kamerasystem enthält neben einigen mechanischen Teilsystemen bestimmt auch ein optisches Teilsystem. Dieses optische Teilsystem enthält neben anderen Komponenten einen eigenen Chip zur Entfernungsmessung.



**Abbildung 12.1:** Jedes System ist Teil eines nächst größeren Systems

Wir nutzen diese einfache Beobachtung, um unseren Entwicklungsprozess zu verallgemeinern. Entscheidungen über die Struktur in dem nächst größeren System sind Randbedingungen und Vorgaben für das darin eingebettete System. Umgekehrt sind Architekturentscheidungen, die wir in dem eingebetteten System treffen, Anforderungen und Randbedingungen für seine Teilsysteme.

## 12.2 Der Entwicklungsprozess großer Systeme

Abbildung 12.2 zeigt die Verallgemeinerung von Abbildung 2.3. Oberhalb der Technologieebenen kann es beliebig viele Ebenen von Subsystemen geben. Auf jeder Ebene können wir für alle darin enthaltenen Subsysteme die Anforderungen analysieren und modellieren und Architekturentscheidungen treffen. Der Prozess, den wir in diesem Buch beschrieben haben, lässt sich also auf jeder Abstraktionsebene anwenden.

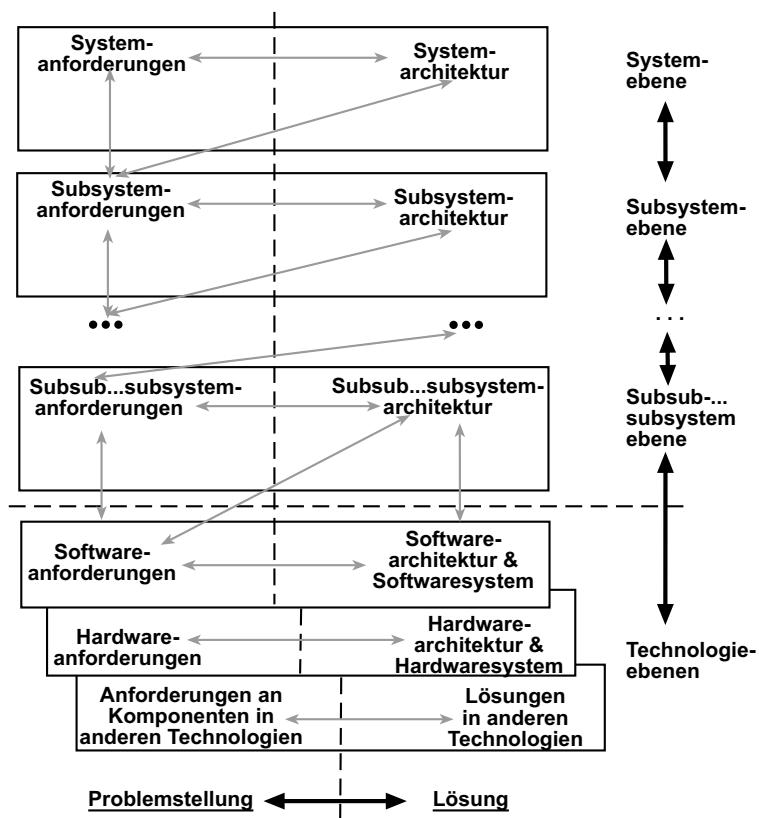


Abbildung 12.2: Ebenen von Subsystemen

Wie schon erwähnt, reichen sehr oft zwei Ebenen aus. Damit können Sie Projekte mit Mannschaften bis zu 100 Personen durchführen. Drei Ebenen sollten Sie einkalkulieren, wenn Ihr Entwicklungsteam bis zu 500 Personen umfasst. Bei langjährigen, international verteilten Großprojekten mit 1000 oder mehr Beteiligten können es auch vier bis sieben Ebenen werden.

An beliebiger Stelle beginnen

Beachten Sie in Abbildung 12.2 die Richtung der Pfeile zwischen den Ebenen. Alle Pfeile haben zwei Pfeilspitzen. Das deutet an, dass der Entwicklungsprozess im Großen keine Richtung hat. Quer durch das ganze Buch haben wir Sie immer wieder darauf hingewiesen, dass es keine verbindliche Reihenfolge für die einzelnen Entwicklungsschritte gibt. Dies gilt um so mehr im Großen. Große Systeme werden selten top-down oder bottom-up oder outside-in oder inside-out entwickelt. Sie beginnen mit einer bestimmten Problemstellung in irgendeiner Ebene in einem bestimmten Subsystem. Und die Auswirkungen Ihres Projekts reichen so weit nach rechts und links, nach oben und unten, wie die Macht, das Geld und die Zeit Ihrer Auftraggeber es erlaubt. Oder Sie können so viel beeinflussen, wie Ihr Verhandlungsgeschick mit Nachbarsystemen rechts und links bezüglich Randbedingungen und Anforderungen von „oben“ und Möglichkeiten, vorhandene Teilsysteme „unten“ zu modifizieren, hergibt.

Mit diesen Vorgaben Ihrer Auftraggeber und Ihrem Verhandlungsgeschick können Sie dann Ihre persönlichen beiden Ebenen finden, für die wir den Prozess in Teil II und III dieses Buches ausführlich beschrieben haben.

### 12.3 Die Wissensstruktur großer Systeme

Vielleicht fragen Sie sich jetzt, wie man bei großen Systemen die ungeheure Flut von Fakten, Anforderungen, Randbedingungen, Designentscheidungen und Hintergrundmaterial in den Griff bekommen kann. Die Antwort haben wir Ihnen bereits in Kapitel 2 gegeben: Füllen Sie das Requirementsgehirn und das Architekturgehirn mit all den Modellen und dazugehörigen Beschreibungen, die wir besprochen haben.

Aber füllen Sie nicht alles über große Projekte in nur zwei Gehirne, sondern leisten Sie sich je zwei Gehirne pro Subsystem! Und das auf jeder Ebene. Die obersten beiden Gehirne – das System-Requirementsgehirn und das System-Architekturgehirn müssen nicht vollständig sein. In dem Requirementsgehirn halten Sie nur so viel fest, wie Sie für globale Entscheidungen über die Gesamtarchitektur benötigen. In dem Architekturgehirn halten Sie die Entscheidungen zur Gesamtarchitektur fest (mit der Begründung, warum so entschieden wurde). Die Struktur dieser Architektur gibt den Rahmen für die erste Ebene von Subsystemen vor, für die Sie wiederum im Requirementsgehirn die Anforderungen des Subsystems präzisieren können.

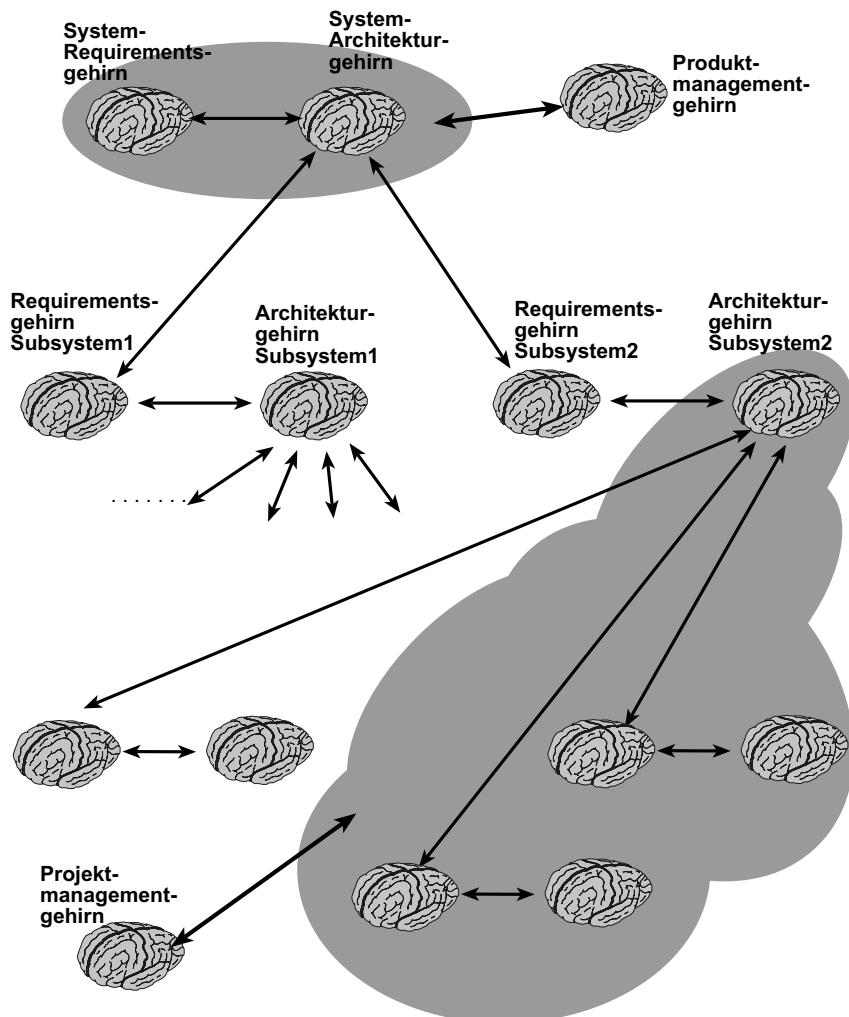


Abbildung 12.3: Produkt- und Produktwissensstruktur

Abbildung 12.3 zeigt den strukturellen Zusammenhang der Gehirne auf den verschiedenen Ebenen. Das in Kapitel 2 erwähnte dritte wichtige Gehirn – das Managementgehirn – ist hier nur zweimal angedeutet. Ganz oben sehen Sie das Gehirn des System- oder Produktmanagers. Darin sind alle Ziele, Pläne und Ressourcen-Überlegungen für das Gesamtsystem enthalten. Unten links ist angedeutet, dass Sie für ein bestimmtes Entwicklungsprojekt, dessen Umfang im Bild grau unterlegt ist, ein Projektmanagementgehirn brauchen werden. Darin werden alle managementrelevanten Daten für dieses Projekt festgehalten. Wir haben den Aspekt des Projektmanagements im Rahmen der Ent-

wicklung bewusst aus dem Buch herausgehalten. Lesen Sie dazu [Ger02]. Darin wird auch gezeigt, dass ein guter Manager die Requirements- und Architekturstrukturen in dem Bereich, für den er zuständig ist, sehr gut kennen und bei der Planung berücksichtigen sollte.

Wenn Ihnen das zu viele Ebenen, zu viele Modelle und zu viele Abhängigkeiten untereinander waren, dann trösten Sie sich damit, dass für einen Großteil aller Projekte der Zwei-Ebenen-Ansatz völlig ausreicht. Falls Sie doch mit den Tücken von Großprojekten kämpfen, finden Sie eine ausführlichere Behandlung der Requirements- und Architekturprozesse für große Systeme mit drei komplexen Fallstudien technischer Systeme in [HHP00].

### 12.4 Denken Sie an Großes?

In diesem Kapitel haben wir die Ausweitung des Vorgehensmodells auf große RTE-Systeme diskutiert. Konsistente Modelle auf allen Abstraktionsebenen mit Querverweisen zu erstellen, ist ein Fernziel, für das Sie etwas Geduld mitbringen müssen.

- Rechnen Sie mit einer Übergangszeit von zwei bis fünf Jahren, bevor Sie die Requirements- und Architekturgehirne aller Teilsysteme eines großen Systems systematisch gefüllt und miteinander verzahnt haben.
- Auch Teilmodelle bringen bereits Nutzen. Wenn Sie den Prozess für diese und jene Komponente isoliert einsetzen, so verbessern Sie die Qualität zumindest dort maßgeblich. Aber vergessen Sie nicht den Masterplan im Hintergrund!
- Der Lernprozess für die einzelnen Schritte der agilen Softwareentwicklung für RTE-Systeme ist nicht so schwierig. Bedenken Sie beim Einsatz im Großen, dass neben den Softwareentwicklern auch andere Personengruppen in den agilen Entwicklungsprozess mit eingeschlossen werden müssen. Produktmanager, Subauftragnehmer, Marketingabteilungen, Kundenrepräsentanten und Standardisierungsgremien müssen auch agil werden und Ihren Prozess mittragen. Dass dieses Ziel erreichbar ist, wurde von mutigen Pionieren bereits erfolgreich demonstriert.

„Jedes Problem erlaubt zwei Standpunkte:  
unseren eigenen und den falschen.“

*Chenning Pollock*

# Teil V

## Die Anhänge

Gratulation! Sie haben es geschafft. Nun müssen Sie Ihr gesamtes Know-how „nur“ noch in die Praxis umsetzen (oder vielleicht haben Sie das bereits parallel zum Lesen getan).

Ihre Eindrücke und Erfahrungen dabei interessieren uns brennend. Ihre Meinung zu unserem Buch ist uns sehr wichtig. Deshalb freuen wir uns auf Ihre Eindrücke und Verbesserungsvorschläge, Ihre Kritik, aber auch Ihr Lob. Treten Sie mit uns in Kontakt. Unsere gemeinsame Mailadresse [rte@b-agile.de](mailto:rte@b-agile.de) gibt Ihnen hierzu Gelegenheit

Ihr Feedback

Um das Thema RTE abzurunden, finden Sie in den folgenden Anhängen Informationen zu weiterführender Literatur. Auch zu diesen Themen finden Sie umfassendere Informationen auf unseren Webseiten: [www.b-agile.de](http://www.b-agile.de) und [www.sophist.de](http://www.sophist.de). Den Weg über eine Auslagerung ins Web haben wir an den Stellen gewählt, an denen wir Ihnen ständig aktualisierte Informationen anbieten möchten, das Abtippen von Texten ersparen wollen oder an denen die Detailinformationen den Fokus oder Umfang des Buches gesprengt hätten. Ein Besuch der beiden Webseiten lohnt sich immer!

Web-powered

Objektorientierte Modellierungstools gibt es viele am Markt. Die folgenden vier alphabetisch sortierten Verweise auf Herstellerwebseiten haben wir auf Grund deren Präsenz im RTE-Umfeld ausgewählt (mehr Infos im Web).

Tools und  
Hersteller

- Artisan Real-time Studio: [www.artisansw.com](http://www.artisansw.com)
- Rhapsody: [www.ilogix.com](http://www.ilogix.com)
- Rose RT: [www.rational.com](http://www.rational.com)
- Telelogic Tau: [www.telelogic.com](http://www.telelogic.com)

Neben den vier hier genannten Toolherstellern kämpfen unzählige weitere UML-Tools ebenfalls um Kunden auf diesem Markt, mit denen Sie die Ideen dieses Buches umsetzen können. Die Auswahl eines Tools ist immer ein sehr individueller Prozess, der von vielen bei Ihnen vorliegenden Randbedingungen abhängt und Ihre Ziele und Vorlieben berücksichtigen sollte.

## Literaturverzeichnis

- [AKZ96] M. Awad, J. Kuusela, J. Ziegler: **Object-Oriented Technology for Real-Time Systems: A practical Approach Using OMT and Fusion**, Prentice Hall, 1996
- [ASG02] www.systemsguild.com: Webseite der **Atlantic Systems Guild**, Stand 2002
- [Barr99] M. Barr: **Programming Embedded Systems in C and C++**, O'Reilly, 1999
- [Boo99] G. Booch, I. Jacobson, J. Rumbaugh: **The Unified Modeling Language User Guide**, Addison Wesley, 1999
- [Coc01a] A. Cockburn: **Writing Effective Use Cases**, Addison Wesley, 2001
- [Coc01b] A. Cockburn: **Agile Software Development**, Addison Wesley 2001
- [DeM79] T. DeMarco, P. J. Plauger: **Structured Analysis and System Specification**, Prentice Hall, 1979
- [DeM99] T. DeMarco, T. Lister: **Wien wartet auf dich**, Hanser, 1999
- [Dou99] B. P. Douglass: **Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns**, Addison Wesley, 1999
- [Fow99] M. Fowler: **Analysemuster**, Addison Wesley, 1999
- [GA00] C. Gernert, N. Ahrend: **IT-Management: System statt Chaos – Ein praxisorientiertes Vorgehensmodell**, Oldenbourg 2000
- [Gam95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, **Design Patterns – Elements of Reusable Object-Oriented Software**, Addison Wesley, 1995
- [Ger02] C. Gernert: **Agiles Projektmanagement**, Hanser, 2002
- [Gom00] H. Gomaa: **Designing Concurrent Distributed and Real-Time Applications with UML**, Addison Wesley, 2000
- [Har98] D. Harel, M. Politi: **Modeling Reactive Systems with StateCharts: The Statemate Approach**, McGrawHill, 1998
- [HHP00] D. Hatley, P. Hruschka, I. Pirbhai: **Process for System Architecture and Requirements Engineering**, Dorset House, 2000
- [HP87] D. Hatley, I. Pirbhai: **Strategies for Real-Time System Specification**, Dorset House, 1987

- [Kru01] P. Kruchten: **Der Rational Unified Process – Eine Einführung**, Addison Wesley, 2001
- [Oes00] B. Oestereich (Hrsg.), P. Hruschka, N. Josuttis, H. Kocher, H. Krasemann, M. Reinhold: **Erfolgreich mit Objektorientierung**, Oldenbourg, 2000
- [POSA98] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: **Pattern-orientierte Softwarearchitektur - Ein Pattern-System**, Addison Wesley, 1998
- [POSA00] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann: **Pattern-Oriented Software Architecture, Vol.2: Patterns for Concurrent and Networked Objects**, Wiley, 2000
- [Pyle93] I. Pyle, P. Hruschka, M. Lissandre, K. Jackson: **Real-Time Systems: Investigating Industrial Practice**, Wiley, 1993
- [Rob97] S. Robertson, J. Robertson: **Requirements: Made to Measure**, www.systemsguild.com, 1997
- [Rob98] J. Robertson: **On setting the context**, www.systemsguild.com, 1998
- [Rob99] S. Robertson, J. Robertson: **Mastering the Requirements Process**, Addison Wesley, 1999
- [ROS 99] D. Rosenberg, K. Scott: **Use Case Driven Object Modeling with the UML – A practical approach**, Addison Wesley, 1999
- [Rum99] J.Rumbaugh, I. Jacobson, G. Booch: **The Unified Modelling Language Reference Manual**, Addison Wesley, 1999
- [Rupp02] C. Rupp, SOPHIST GROUP: **Requirements Engineering**, Hanser, 2002
- [SGW94] B. Selic, G. Gullekson, P. T. Ward: **Real-Time Object-Oriented Modeling**, Wiley, 1994
- [SIM99] D. E. Simon: **An embedded Software Primer**, Addison Wesley, 1999
- [Sta02] G. Starke: **Effektive Software-Architekturen. Ein praktischer Leitfaden**, Hanser, 2002
- [UML1.4] OMG 2001 Object Management Group: **OMG Unified Modeling Language Specification**, Version 1.4, Needham: Object Management Group, Inc., 2001
- [Volere] J. Robertson, S. Robertson, **Volere Requirements SpecificationTemplate**, www.systemsguild.com

# Index

- Aggregation 79
- Agile System-/Softwareentwicklung 19
  - , Maximen 20
  - Akteure 37
  - Aktionen 74
  - Aktive Komponente 153
  - Aktives Objekt 154
  - Aktivitäten 74
  - Aktivitätsdiagramm 70
  - Anforderungen 86
    - , Änderbarkeit 11
    - , Benutzbarkeit 9
    - , Effizienz 10, 97
    - , gesetzliche 12
    - , physikalische 91
    - , Portabilität 1, 97
    - , Sicherheit 11, 89
    - , an die Software 14
    - , an das System 14
    - , Umwelt 92
    - , Verfügbarkeit 10, 88, 97
    - , Zuverlässigkeit 10, 88, 97
  - Architektur
    - , gehirn 15, 185
    - , muster 99
    - , pattern für RTE-Systeme 140
    - , Software 14
    - , System 14, 23, 24, 85 ff
  - asynchrone Kommunikation 162
  - Attribut 79, 81
    - , Beschreibungsmuster 135
  - Basissystemschnittstelle 145
  - Begriffe definieren 45, 111
  - Benutzbarkeit 9
  - Beschreibungsmuster
    - , für Beziehungen 130
    - , für Klassen 129
    - , für Komponenten 99
    - , für Attribute 131
    - , für Knoten 95
    - , für Subsysteme 99
    - , für Systemprozesse 61, 62, 64
    - , für Verbindungen 95
  - Beziehung 79
    - , arten 79, 80
  - , beschreibungsmuster 135
  - Codegeneratoren 176
  - Deploymentdiagramm
    - Verteilungsdiagramm
  - Dienstqualität 9
  - Distributed System
    - Verteiltes Systems
  - Disziplinen 17,18
  - Effizienzanforderungen 10
  - Ein-/Ausgabeschnittstelle 145
  - Eingebettetes System 6, 21
  - Entity-Klassen 80, 120,124
  - Embedded System
    - Eingebettetes System
    - RTE-System
  - Echtzeitssystem
  - Entwicklungsprozess 13
    - , Randbedingungen für 92
    - , für große Systeme 183
  - Ereignis 37, 74
  - Essenz 63
  - Extend-Beziehung 110
  - fachliche Klassen 120 ff
  - fachliche Softwarearchitektur 121 ff
  - Frameworks 142, 169
  - Generalisierung/Spezialisierung 80, 111
  - Hard Real-Time 7
  - Hardware
    - , Anforderungen 16
    - , Architektur 16
    - , Verteilung 94
  - History 75, 77
  - include-Beziehung 110
  - Interface 136
  - Interrupt-Task 156
  - Klasse 79
    - , Beschreibungsmuster 82, 130
    - , fachliche Kategorien 120

- , technologiegetriebene Kategorien 144
- Klassendiagramm 79
  - , erstellen 126
  - , zur Kontextabgrenzung 33
  - Kollaborationsdiagramm 83, 101
    - , zur Taskkommunikation 165
    - Kommunikation zwischen Tasks 161 ff
      - , technische Umsetzung 167
    - Komponente 147 ff
      - , aktive und passive 153
      - , struktur intern 151
    - Komponentendiagramm 160
    - Komposition 79
    - Knoten 50
      - , spezifikation 95
    - Kontextabgrenzung
      - , für Software 107ff
      - , für das System 29ff
      - , mit Klassendiagramm 33, 109
      - , mit Sequenzdiagramm 34
      - , mit Use-Case-Diagramm 31
      - , mit Verteilungsdiagramm 50
      - , physikalisch 50
    - Kontextdiagramm 30
  - Maximen agilen Handelns 20
  - Messages → Nachrichten
  - Mensch-/Machine-Schnittstelle 144
  - Multitechnologiesysteme 16
  - Nachbarsystem 30
    - , identifizieren 31
  - Nachrichten 161
  - nicht-funktionale Anforderungen 86 ff
    - , als Treiber für Hardware 94
    - , als Treiber für Software 116
  - Objektflußdiagramm 119
  - Operation 79
  - Package → Paket
  - Paket 53
  - parallele Prozesse 7,21
  - passive Komponente 153
  - Pattern 142-145, 169
  - Phasen 17, 18
  - physikalische Anforderungen 91
  - physikalische Verbindungen 50
  - Polling-Task 156
  - Polymorphie 138
  - Prioritäten
    - , für den Entwurf 93, 115
    - , für Tasks 155
  - Produkt 5
  - Produktentwicklung
    - Systementwicklung
  - Qualitätsanforderungen 54, 87
  - Quality of Service 9
  - Randbedingungen 86
    - , für den Prozess 92
    - , sammeln 54, 86ff
    - , Software 113ff
    - , System 49ff, 90ff
  - Rational Unified Process 17
  - reaktive Systemprozesse 69
  - Real Time 6
  - Requirementsgehirn 15, 185
  - Rollen 28
  - RTE-System 5
    - , Vorgehensmodell 17
  - RUP 17
  - Schnittstellen
    - , Ein-/Ausgabe 145
    - , Mensch-/Maschine 144
    - , zum Basissystem 145
    - , zur Umgebung 114
  - Sequenzdiagramm 83
    - , verfeinerung 164
    - , zur Kontextabgrenzung 34
    - , zur Taskkommunikation 163
  - Service-Klassen 124, 127
  - Sicherheitsanforderungen 11, 89, 97
  - Sichtenklassen 122, 126
  - Soft Real-Time 7
  - Software
    - , anforderungen 103, 105 ff
    - , architektur 14, 103
      - fachlich 117 ff
      - technisch 143 ff
    - , entwicklungszyklus 103 ff
    - , kontext 107
    - , prozesse 109
    - , randbedingungen 113 ff

- , strukturierung 18
- , vorbereitung 18
- , Ziele 105
- Stakeholder 27, 107
- StateChart → Zustandsdiagramm
- Stereotypen 39
  - , für Subsysteme 98
- Steuerungsklassen 123, 127
- Subsystem 52, 147, 182
  - , beschreibung 99
  - , entscheiden 96
  - , kategorien 97
- synchrone Kommunikation 162
- System 5
  - , anforderungen 14, 23, 24, 25, 59
  - , architektur 14, 23, 24, 85 ff
  - , entwicklungszyklus 23 ff
  - , kontext 29
  - , konzeption 17
  - , prozesse 36, 40
  - , prozess-Spezifikation 60 ff
  - , prozess-Zerlegung
  - , randbedingungen 49 ff
  - , strukturierung 17
- Szenarien 83
  
- Task
  - , abhängigkeiten 160
  - , arten 155ff
  - , bildung 155
  - , kommunikation 161 ff
    - mit Kollaborationsdiagramm 165
    - mit Sequenzdiagrammen 163
    - mit StateCharts 166
  - , struktur 159
  - , synchronisation 161
- technische Software-Architektur 143 ff
- Technologieabhängigkeiten 65
  
- , in Klassen 82
- , in Systemprozessbeschreibungen 67
- technologiegetriebene Klassen 144 ff
- Technologievorgaben 91
- Testen 178
- Thread 151
  
- Umweltanforderungen 92
- Use-Case 36
  - , Beschreibung 61
  - , Beziehungen 110
  - , für Systeme 36 ff
  - , für Software 109 ff
- Use-Case-Diagramm 32
  - , zur Kontextabgrenzung 31
  
- Verbindung (physikalisch) 50
- , spezifikation 95
- Vererbung 138
- Verfügbarkeitsanforderungen 10,88, 97
- verteiltes System 6, 21
- Verteilungsdiagramm 50, 152
  - , zur Kontextabgrenzung 51, 115
- Vorgehensmodell
  - Entwicklungsprozess
  
- Zeitanforderungen 21, 87
- , in Sequenzdiagrammen 35
- , in Subsystembeschreibungen 88
- , in Systemprozessbeschreibungen 88
- Ziele
  - , für Software 106
  - , System 26
- Zustandsdiagramm 74
  - , zur Taskkommunikation 166
- Zustandsmodelle 44
- Zuverlässigkeitssanforderungen 10, 88, 97

# *Mehr Experten Know-How aus erster Hand!*

**Offene Seminare**  
zum Buch:  
**Agile Softwareentwicklung  
für Embedded Real-Time  
Systems mit der UML**  
von Chris Rupp  
und Peter Hruschka

Informieren Sie sich  
über alle Termine unter  
[www.b-agile.de](http://www.b-agile.de) und  
[www.sophist.de](http://www.sophist.de)

## **Projektcoaching durch die Autoren für *Ihr Projekt***

- ▶ strategische Beratung zu Methoden und Tools
- ▶ Training on the Job durch Mitarbeit bei der Modellierung
- ▶ Reviews, Audits, Qualitätssicherung Ihrer Modelle

Inhouse Termine auf Anfrage

Chris Rupp  
[chris.rupp@sophist.de](mailto:chris.rupp@sophist.de)  
+49.911.409000

Peter Hruschka  
[hruschka@b-agile.de](mailto:hruschka@b-agile.de)  
+49.172.2411656



**b-agile**



**SOPHIST GROUP**  
SOPHIST GmbH SOPHIST Technologies GmbH

## Die Autoren

**Chris Rupp** engagiert sich seit mehr als 10 Jahren – immer noch mit viel Begeisterung – als Trainerin und Beraterin in Projekten internationaler Kunden. Ihr Fokus liegt auf sicherheitskritischen technischen Großprojekten, z.B. aus der Flugsicherungs-, Telekommunikations- oder Automobilbranche. Ihre Arbeitsmethodik umfasst die Bereiche Vorgehensmodelle, Objektorientierung, Requirements-Engineering und Organisationspsychologie.

»Nebenbei« ist sie Geschäftsführerin der SOPHIST GROUP.

Ihre privaten Vorlieben (zu viele für zu wenig Zeit): Menschen, Reisen, Literatur, Rotwein und die Suche nach dem Sinn des Lebens.

Sie erreichen die Autoren unter [hruschka@b-agile.de](mailto:hruschka@b-agile.de) und [rupp@b-agile.de](mailto:rupp@b-agile.de)



**Peter Hruschka** trägt als Trainer, Berater und Autor seit mehr als 25 Jahren Methoden- und Verfahrenswissen in die Praxis. Am Beginn seiner Karriere stand die Mitarbeit an der Standardisierung und Implementierung der deutschen Echtzeitsprache PEARL. Seither hat er an der Analyse und Konzeption vieler technischer Systeme in unterschiedlichsten Branchen mitgewirkt.

Das dabei gewonnne Wissen setzte er – wie seine Partner der Atlantic Systems Guild – immer wieder in Verbesserungen von Methoden und Werkzeugen um.

In seiner Freizeit reist er gern und versucht dabei, weiße Bälle in viel zu weit entfernten, viel zu kleinen Löchern zu versenken.

Software für technische Systeme unterliegt ständiger Innovation und stellt besondere Anforderungen an Entwickler. Die Aufgabe wird leichter, wenn Sie den Nutzen des Produktes für den Kunden und die Erfüllung von Quality-of-Service-Anforderungen in den Vordergrund stellen.

Dieser Praxisleitfaden verrät Ihnen, wie Sie technische Systeme mit objektorientierten Methoden effektiv und systematisch realisieren.

**Darum geht's:**

- Was sind Ihre essenziellen Aufgaben als Entwickler technischer Systeme?
- Wie hilft Ihnen die Betrachtung des Gesamtprodukts, zu besseren Software-Strukturen zu kommen?
- Wie kommen Sie von den Produktzielen über Modelle zu einsetzbaren Systemen?
- Welche Auswirkungen haben Qualitätsanforderungen und Randbedingungen auf Ihre Architektur?
- Wie nutzen Sie die UML für Systeme, die mehr als nur Software beinhalten?
- Wie modellieren Sie Hardware, Verteilung, Prozesse, Kommunikation und Synchronisation?

**Im Internet:** Hintergrundinformationen, Anwendungsbeispiele, Fallstudien, weitere Praxistipps finden Sie unter [www.b-agile.de](http://www.b-agile.de)

