Weilkiens · Huwaldt · Mottok · Roth · Willert

Modellbasierte* Softwareentwicklung für eingebettete Systeme

verstehen und anwenden

*Ein Modell sagt mehr als 1.000 Bilder!

dpunkt.verlag

Bild: v.l.n.r. Andreas Willert, Alexander Huwaldt, Stephan Roth, Tim Weilkiens und Jürgen Mottok

Tim Weilkiens ist Vorstand und Trainer der oose Innovative Informatik eG. Seine thematischen Schwerpunkte sind die Modellierung von Systemen, Software und Unternehmen. Er ist für oose Repräsentant bei der OMG und u.a. Koentwickler des Zertifizierungsprogramms OCEB und OCEB2.

Alexander Huwaldt ist Geschäftsführer der SiSy Solutions GmbH, seit 2007 als UML Professional nach OMG-Standards zertifiziert und in der Softwareentwicklung tätig. Er ist Hochschuldozent für Mikrocontrollerprogrammierung, Software Engineering, UML, SysML und BPMN.

Prof. Dr. Jürgen Mottok lehrt Informatik an der Hochschule Regensburg. Seine Lehrgebiete sind Embedded Software Engineering, Real-Time Systems, Functional Safety und IT-Security. Er ist in Programmkomitees zahlreicher wiss. Konferenzen vertreten.

Stephan Roth ist Trainer und Berater bei der oose Innovative Informatik eG. Seine Schwerpunkte sind modellbasiertes Systems und Software Engineering, Softwaredesign und Softwarearchitektur sowie Software Craftsmanship und Clean Code Development.

Andreas Willert ist Geschäftsführer der Willert Software Tools GmbH und als Trainer, Berater und Coach für Software und Systems Engineering tätig.

Tim Weilkiens • Alexander Huwaldt • Jürgen Mottok •
Stephan Roth • Andreas Willert

Modellbasierte Softwareentwicklung für eingebettete Systeme verstehen und anwenden

dpunkt.verlag

Tim Weilkiens
Tim.Weilkiens@oose.de

Alexander Huwaldt
a.huwaldt@sisy.de

Jürgen Mottok
juergen.mottok@oth-regensburg.de

Stephan Roth
Stephan.Roth@oose.de

Andreas Willert
awillert@willert.de

Lektorat: Christa Preisendanz
Copy-Editing: Ursula Zimpfer, Herrenberg
Satz: Frank Heidt
Herstellung: Stefanie Weidner
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN:
Print 978-3-86490-524-7
PDF 978-3-96088-593-1
ePub 978-3-96088-594-8
mobi 978-3-96088-595-5

1. Auflage 2018

Wieblinger Weg 17
69123 Heidelberg

5 4 3 2 1 0

Vorwort

»Grau, teurer Freund, ist alle Theorie
und grün des Lebens goldner Baum.«

Faust – Der Tragödie erster Teil
(Johann Wolfgang von Goethe, 1749 – 1832)

Wir, die Autoren dieses Buches, sind überzeugt davon, dass die Modellierung von eingebetteten Systemen zu erfolgreichen Projekten führt und viele Probleme, die wir in Projekten beobachten, lösen kann. Gleichzeitig sehen wir, dass Modellierung häufig nicht oder ungünstig eingesetzt wird. Vor etlichen Jahren haben wir uns vorgenommen, dem entgegenzuwirken. Aus dem Impuls heraus ist das Manifest »Modeling of Embedded Systems« (*www.mdse-manifest.org*) sowie die nicht kommerzielle Konferenz »Modeling of Embedded Systems Conference« (MESCONF, *www.mesconf.de*) entstanden.

Schnell waren wir uns einig, dass wir unser Wissen und unsere Erfahrung in einem Referenzwerk zu Verfügung stellen wollen. Wir haben aus unserer langjährigen Projekterfahrung mit kleinen, mittleren und großen Projekten in diesem Buch wesentliche Aspekte der methodischen und systematischen Anwendung von Konzepten, Methoden, Techniken und Werkzeugen des Software Engineering von eingebetteten Systemen zusammengefasst. Wir wenden uns damit an Leser, die bereits über Kenntnisse in der Softwareentwicklung und Programmierung verfügen.

Der Inhalt bezieht sich speziell auf den Entwurf und die Realisierung von Mikrocontrollerlösungen und orientiert sich damit an den Bedürfnissen der Entwicklung von eingebetteten Systemen. Wir erörtern somit wichtige Aspekte der Softwareseite des Embedded Systems Engineering.

In den Inhalt dieses Buches flossen ebenfalls die Anforderungen und Erfahrungen aus unserer Lehrtätigkeit an Berufsakademien, Fachhochschulen und Universitäten ein. Gerade hier liegt ein Spannungsfeld: Die Ausbildung von Informatikern über Elektrotechniker bis zu Mechatronikern wird mehr und mehr zur

Herausforderung. Dem Informatiker wird viel Softwaretechnik in einer zunehmend von der Hardware abstrahierten Welt und faktisch keine Elektrotechnik vermittelt. Dem Elektrotechniker wird naturgemäß viel Wissen über Elektrotechnik und Elektronik mit auf den Weg gegeben, aber für die besonderen Belange der Softwareentwicklung, obwohl oft hardwarenah vermittelt, fehlt die Zeit zur notwendigen Vertiefung. Mechatroniker klagen sowieso über die ungeheure Stofffülle zwischen Mechanik, Hydraulik, Pneumatik, Elektrotechnik und Informatik.

Oft wird Software Engineering und die in diesem Diskurs angebotenen Techniken und Werkzeuge der Modellierung als nett, aber in dem Bedürfnis, endlich seine Ideen in Quellcode umzuwandeln, als hinderlich angesehen. Das »Rush to Code«-Syndrom ist weit verbreitet (und auch allzu menschlich). Gleichzeitig vertreten wir den Standpunkt, dass eine Technik nur dann von praktischem Nutzen ist, wenn die Arbeitsergebnisse des einen Schritts in geeigneter Form im nächsten Arbeitsschritt möglichst direkt weiterverwendet werden können und von geeigneten Werkzeugen unterstützt werden. Das wird in theoretischen Abhandlungen nicht erlebbar. Deshalb basieren dieses Buch und der Lernerfolg auf der praktischen Anwendung des Vorgestellten und einer konsequenten Werkzeugnutzung.

Wir haben das Buch gemeinschaftlich geschrieben und dazu viele spannende Workshops und Diskussionen durchgeführt. Obwohl es im Autorenteam unterschiedliche Schwerpunkte und Sichtweisen gibt, konnten wir mit diesem Buch die verschiedenen Puzzleteile zu einem gemeinsamen Bild zusammensetzen.

Wir sehen es als einen kontinuierlichen Prozess, die Methoden, Techniken und Werkzeuge immer weiterzuentwickeln, zu optimieren und an die sich stetig ändernde Welt anzupassen. Dieses Buch ist allerdings naturbedingt statisch. Zukünftige Auflagen werden diese Entwicklung widerspiegeln und Sie können dem Prozess beispielsweise auf der MESCONF oder anderen Veranstaltungen der Community folgen.

»Wir lernen, was wir tun.«
John Dewey

Mai 2018

Tim Weilkiens
Alexander Huwaldt
Jürgen Mottok
Stephan Roth
Andreas Willert

Inhaltsübersicht

1	Einleitung	1
2	Basiswissen	13
3	Modellbasierte Softwareprozesse und Toollandschaften	43
4	Modellbasiertes Requirements Engineering	61
5	Modellbasierte Architekturbeschreibung	83
6	Modellbasiertes Softwaredesign	103
7	Modellbasiertes Testen	145
8	Integration von Werkzeugen	171
9	Modellbasierte funktionale Sicherheit	187
10	Metamodellierung	221
11	Einführung eines modellbasierten Ansatzes in einer Organisation	235
12	Lebenslanges Lernen	249
13	Fazit	277

Anhang

A	Ausblick: Skizze eines Reifegradmodells für MDSE	281
B	Kurzreferenz UML und SysML	295
C	Glossar	335
D	Literaturverzeichnis	355
	Stichwortverzeichnis	363

Inhaltsverzeichnis

1 Einleitung 1

1.1 Warum gerade jetzt dieses Buch? 1

1.2 Wie sollte man dieses Buch lesen? 3

1.3 Was ist an eingebetteten Systemen so besonders? 4

1.4 Wie sieht das typische Zielsystem aus? 5

1.5 Fallbeispiele 7

1.5.1 Die Anlagensteuerung 7
1.5.2 Die Baumaschine 8
1.5.3 Das Energie-Monitoring-System 9
1.5.4 Das Beispielsystem SimLine 9

1.6 Das Manifest 10

2 Basiswissen 13

2.1 Was sind eingebettete Systeme? 13

2.2 Software Engineering für eingebettete Systeme 16

2.3 Der Qualitätsbegriff 19

2.3.1 Externe Qualität 19
2.3.2 Interne Qualität 19
2.3.3 Qualitätskriterien nach ISO/IEC 25010 20

2.4 Einführung in Model-Driven Software Engineering 23

2.5 Komplexität 24

2.6 Unser Gehirn als Engpass 26

2.7 Vorgehensweisen und Techniken, um einer steigenden Komplexität zu begegnen 27

2.8 Komplexen Systemen lässt sich nicht mit simplen Methoden begegnen 28

2.9 Verstehbarkeit und Redundanz 29
2.10 Was ist ein Modell? 33
2.11 Modelle helfen, die Zukunft vorwegzunehmen? 36
2.12 Modelle helfen zu abstrahieren? 38
2.13 Resümee: Nutzen von MDSE? 40

3 Modellbasierte Softwareprozesse und Toollandschaften 43
3.1 Klassifikation von Modellierungswerkzeugen 43
3.1.1 Modellierungswerkzeuge der Kategorie A (universelle Malwerkzeuge) 47
3.1.2 Modellierungswerkzeuge der Kategorie B (spezialisierte Malwerkzeuge) 47
3.1.3 Modellierungswerkzeuge der Kategorie C (einfache Modellierungswerkzeuge mit Modelldatenbank) . . 47
3.1.4 Modellierungswerkzeuge der Kategorie D (vollständiger Zyklus einer Embedded-Software-Engineering-Werkzeugkette) 48
3.2 Vorgehensmodelle 48
3.2.1 SYSMOD Zickzack trifft auf IBM Rational Harmony 49
3.2.2 Spezifikation oder Lösung? 50
3.2.3 Wiederverwendung 52
3.3 Best Practice für kleine Teams 53

4 Modellbasiertes Requirements Engineering 61
4.1 Requirements Engineering 61
4.2 Anforderungen in der Modellierung 62
4.3 Anforderungen und Architektur im Zickzack 64
4.4 Szenario: Modellbasierte Spezifikation mit UML erstellen 66
4.4.1 Überblick über Methode 66
4.4.2 Problemanalyse 66
4.4.3 Basisarchitektur 68
4.4.4 Systemidee und Systemziele 69
4.4.5 Stakeholder und Anforderungen 70
4.4.6 Systemkontext 71
4.4.7 Anwendungsfälle und Aktivitäten 72
4.4.8 Fachwissen 75
4.4.9 Szenarien 76
4.5 Mehr Modellierung: Ausführbare Spezifikation 77
4.6 Weniger Modellierung: Diagramme für Anforderungsdokumente . . . 80

4.7 Neuentwicklung versus Weiterentwicklung 80
4.7.1 Basisarchitektur 81
4.7.2 Anwendungsfälle 81
4.7.3 Szenarien 81

5 Modellbasierte Architekturbeschreibung 83

5.1 Architektur – Was ist das eigentlich? 83
5.2 Die technische Softwarearchitektur 85
5.3 Architekturmuster und deren Bedeutung 88
5.4 Das Laufzeitsystem als Basismuster in eingebetteten Applikationen . 90
5.5 Referenzmuster für eine Laufzeitarchitektur 94
5.5.1 Sensorik, Einlesen, Filtern, Bewerten 96
5.5.2 Transformation der Sensorik in Aktivitäten (Verarbeitung) 97
5.5.3 Ausgabe der Daten und Ansteuerung der Aktoren 97
5.6 Fachliche Architektur 97
5.7 Architektur-Assessment 98

6 Modellbasiertes Softwaredesign 103

6.1 Gesichtspunkte der fachlichen Architektur und des Designs 103
6.2 Hierarchische Dekomposition 105
6.3 Diskretisierungs- und Laufzeiteffekte im Design 115
6.4 Softwaredesignprinzipien 118
6.4.1 Was ist ein Prinzip? 118
6.4.2 Grundlegende Designprinzipien 119
6.4.3 Designprinzipien in der Modellierung 123
6.5 Hardwareabstraktion 123
6.5.1 Ausgangssituation 124
6.5.2 Evolution der Mikrocontrollerprogrammierung in C 124
6.5.3 Die klassische Vorgehensweise mit der UML 127
6.5.4 Die graue Theorie 130
6.5.5 Lösungsansätze für die Modellierung 133
6.5.6 Lösungsansätze für die Codegenerierung 136

7 Modellbasiertes Testen 145

7.1 Warum eigentlich testen? 145
7.2 Nicht nur sicherstellen, dass es funktioniert 146
7.2.1 Ein angstfreies Refactoring ermöglichen 146
7.2.2 Besseres Softwaredesign 146
7.2.3 Ausführbare Dokumentation 147
7.2.4 Tests helfen, Entwicklungskosten zu sparen 147

7.3 Die Testpyramide 148
7.4 Test-Driven Development (TDD) 151
7.4.1 Viel älter als vermutet: Test First! 151
7.4.2 TDD: Red – Green – Refactor 153
7.5 Model-Based Testing (MBT) 154
7.6 UML Testing Profile (UTP) 155
7.7 Ein praktisches Beispiel 156
7.8 Werkzeuge, die dieses Vorgehen unterstützen 163

8 Integration von Werkzeugen 171
8.1 Anforderungen an Schnittstellen zu Werkzeugen unterschiedlicher Disziplinen 172
8.1.1 Digital Twin 172
8.1.2 Traceability aus Safety-Gesichtspunkten 173
8.1.3 Projekt- und Workload-Management 173
8.2 Synchronisation der Daten zwischen Repositories 175
8.3 Zentrales Repository 176
8.4 Ein Werkzeug für alles 178
8.5 OSLC – Open Services for Lifecycle Collaboration 179
8.6 XMI – Austausch von Daten innerhalb der Fachdomäne MDSE . . . 181
8.7 ReqIF zum Austausch von Anforderungen 181
8.8 FMI (Functional Mock-up Interface) 182
8.9 SysML Extension for Physical Interaction and Signal Flow Simulation (SysPhS) 183
8.10 AUTOSAR 184
8.11 Stand der Praxis 184

9 Modellbasierte funktionale Sicherheit 187
9.1 Funktionale Sicherheit 187
9.2 Entwurfsmuster der funktionalen Sicherheit 190
9.2.1 Zufällige und systematische Fehler 191
9.2.2 Symmetrische und diversitäre Redundanz 193
9.2.3 Architekturmuster 195
9.2.4 Monitor-Actuator Pattern (MAP) 198
9.2.5 Homogenous Redundancy Pattern (HRP) 200
9.2.6 Triple Modular Redundancy Pattern (TMR) 202
9.2.7 SIL-Empfehlung für die Safety and Reliability Design Patterns 206

9.3 Vom Modell zum sicheren Quellcode: Coding-Standards für die sichere Programmierung 207
9.3.1 Normativer Hintergrund 207
9.3.2 MISRA-C und MISRA-C++ 208
9.3.3 Prüfwerkzeuge 209
9.3.4 Codegenerierung 209
9.4 Das Safety and Reliability Profile der UML 210
9.5 Praktische Anwendung der Modellierung im Kontext sicherheitskritischer Systeme 211
9.5.1 Beweis der Korrektheit der Transformation 212
9.5.2 Verifikation der Qualität des Werkzeugs 213
9.5.3 Zertifiziertes Framework 213
9.6 Vorteile der modellgetriebenen Entwicklung im Safety-Kontext . . . 214
9.6.1 Traceability 214
9.6.2 Semiformale Spezifikation 215
9.6.3 Normen empfehlen den Einsatz von formalen und/oder semiformalen Methoden und Notationen 216
9.6.4 Automatisierte Transformationen 216
9.6.5 Modellierungsrichtlinien (Guidelines) 217
9.6.6 Dokumentation 218
9.7 Modellgetriebene Entwicklung mit der UML 218

10 Metamodellierung 221
10.1 Modell und Metamodell 221
10.2 UML-Metamodelle 223
10.3 EAST-ADL 225
10.4 AADL 230
10.5 Vergleich EAST-ADL und AADL 232

11 Einführung eines modellbasierten Ansatzes in einer Organisation 235
11.1 Ausgangslage 236
11.2 Vorgehen 237
11.2.1 Problem analysieren 237
11.2.2 Idee und Ziele des Vorgehens 238
11.2.3 Stakeholder und Anforderungen 239
11.2.4 Methodikkontext 240
11.2.5 Anwendungsfälle 240
11.2.6 Fachwissenmodell 242
11.2.7 Verifikation und Validierung 242
11.3 Auswahl der Modellierungssprachen 243

11.4 Auswahl der Modellierungswerkzeuge 244
11.5 Typische Fehler 245
11.5.1 Schnellstart 245
11.5.2 Übergewicht 245
11.5.3 Einsame Insel 246
11.5.4 Elfenbeinturm 246
11.5.5 Aus der Lernkurve fliegen 246

12 Lebenslanges Lernen 249
12.1 Lernen – die Sicht des Konstruktivismus 249
12.2 Kompetenzen – der Blick auf die modellbasierte Softwareentwicklung 251
12.3 Agilität – Lernen mit Methoden und Praktiken 254
12.4 Psychologische Grundlagen von Fehlern 261
12.4.1 Denkfallen als Fehlerursache 261
12.5 Software Craftsmanship – ein Beispiel 264
12.6 Modellierungskultur – ein Kodex 266
12.6.1 Manifest – Modeling of Embedded Systems 266
12.6.2 Big Picture – der Blick auf den Kodex einer Modellierungskultur 266
12.6.3 Moderation – die konstruktive Kommunikation 267
12.6.4 Reflexion – Lernen durch Reflektieren 268
12.6.5 Selbstverpflichtung – selbstgesteuertes Lernen als Entwickler und als Team 269
12.6.6 Teamradar – ein Feedbackbarometer 270
12.6.7 Experten als Teamcoach – Agenten der Veränderung 272
12.6.8 Funktionale Sicherheit – ein Beitrag der normativen Sicherheitskultur 272

13 Fazit 277

Anhang

A Ausblick: Skizze eines Reifegradmodells für MDSE 281
A.1 Hintergrund und Motivation 281
A.2 Die Skizze als ein Start – Diskussionsforum 283
A.3 Ausgangslage Manifest – Ziele kompakt 284
A.4 Die Reifegrade – der Weg in die Modellierungskultur 285
A.5 Evaluation und Fragenkatalog 292

B Kurzreferenz UML und SysML 295
B.1 Eine kurze Geschichte der UML 296
B.2 Aufbau und Architektur der UML 297
B.3 Anwendungsfalldiagramm 299
B.3.1 Akteur 299
B.3.2 Anwendungsfall 300
B.3.3 Anwendungsfallbeziehungen 300
B.4 Aktivitätsdiagramm 300
B.4.1 Aktivität und Aktivitätsparameter 301
B.4.2 Aktion und Pin 302
B.4.3 Kontroll- und Objektfluss 303
B.4.4 Start- und Endknoten 304
B.4.5 Entscheidung und Zusammenführung 304
B.4.6 Splitting und Synchronisation 304
B.5 Klassendiagramm 305
B.5.1 Klasse und Objekt 305
B.5.2 Attribut 306
B.5.3 Operation 306
B.5.4 Assoziation 307
B.5.5 Komposition 308
B.5.6 Generalisierung 308
B.5.7 Signal 308
B.5.8 Datentyp 309
B.5.9 Templates 309
B.6 Kompositionsstrukturdiagramm 310
B.6.1 Konnektor 311
B.6.2 Port 312
B.7 Sequenzdiagramm 314
B.7.1 Interaktion 314
B.7.2 Lebenslinie 315
B.7.3 Nachricht 316
B.7.4 Kombiniertes Fragment 316
B.7.5 Zeitliche Zusicherungen 317
B.8 Zustandsdiagramm 317
B.8.1 Zustandsautomat 318
B.8.2 Zustand 319
B.8.3 Transition 319
B.8.4 Start- und Endzustand 320
B.8.5 Pseudozustand 321

B.9 Paketdiagramm 322
B.9.1 Paket und Modell 322
B.9.2 Pakete importieren 323
B.9.3 Modellbibliothek 323
B.10 Querschnittselemente 323
B.10.1 Kommentar 323
B.10.2 Zusicherung 324
B.10.3 Trace-Beziehung 324
B.11 Profil 324
B.11.1 SysML 326
B.11.2 MARTE 329
B.11.3 UML Testing Profile (UTP) 331
B.11.4 MDESE-Profil (basierend auf SYSMOD-Profil) 332

C Glossar 335

D Literaturverzeichnis 355

Stichwortverzeichnis 363

1 Einleitung

»Aus kleinem Anfang entspringen alle Dinge.«

(Marcus Tullius Cicero, 106 – 43 v. Chr., römischer Redner und Staatsmann)

1.1 Warum gerade jetzt dieses Buch?

Die Beherrschung von Komplexität ist wohl die größte Engineering-Herausforderung des 21. Jahrhunderts. Software wird der Rohstoff sein, aus dem zukunftsfähige, smarte Systeme erschaffen werden. Software ist schon jetzt ein zentraler Bestandteil unserer Infrastruktur. Täglich kommen wir Menschen – mehr oder weniger bewusst – mit vielen Systemen in Kontakt, in denen Software zentrale Aufgaben übernimmt: Sei es der Fahrstuhl, die Klimaanlage, der Fahrkartenautomat oder das Auto.

Mit dem Internet der Dinge (*Internet of Things*, IoT) und der vierten industriellen Revolution, sprich: der Zukunftsvision »Industrie 4.0« der deutschen Bundesregierung, wird der Anteil und somit auch der Einfluss von Software auf alle Lebensbereiche des Menschen noch weiter zunehmen.

So wie die Metallverarbeitung eine Schlüsseltechnologie in der industriellen Revolution war, wird Software und Systems Engineering die Schlüsseltechnologie des Informationszeitalters sein.

Im Kontext von Embedded Systems werden sich obige Anforderungen besonders auswirken und modellgetriebene Entwicklung wird hier eine zentrale Rolle im Software und Systems Engineering für eingebettete Systeme einnehmen.

Eingebettete Systeme unterliegen wie jedes IT-System ständigen Innovationen. Moore's Law lässt grüßen. Es gibt jedoch innerhalb der Genesis solcher Systeme immer wieder Umwälzungen, die der scheinbar stetigen Entwicklung sprunghaften Charakter verleihen. Für den Betrachter scheint sich innerhalb kurzer Zeit alles zu ändern. Neue Programmiersprachen, mit denen die Entwickler konfrontiert werden, sind nur die Spitze des Eisbergs. Schaut man aus der heuti-

gen Perspektive auf das Themengebiet Software Engineering zurück (siehe Abb. 1–1), kann die Entwicklung auf drei Umbrüche verdichtet werden.

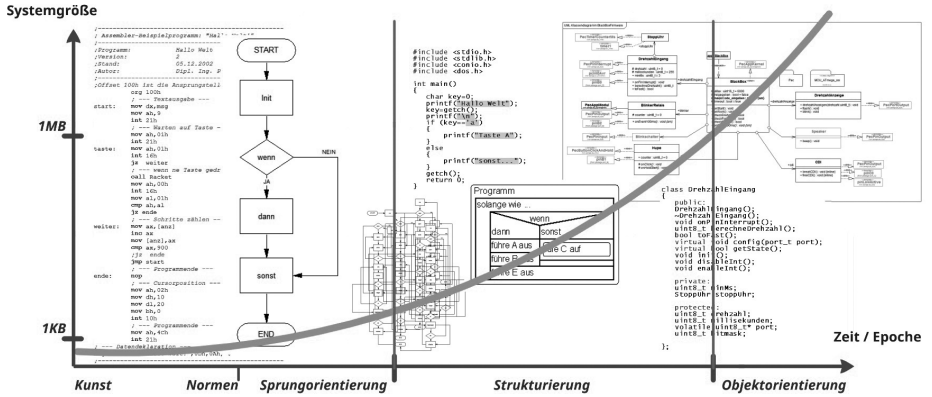

Abb. 1–1 *Entwicklung von Software-Engineering-Paradigmen*

- **1950er- bis 1960er-Jahre:** Initialzustand, maschinennahe Programmierung, dominierende Programmiersprache ist Assembler und es gibt erste Hochsprachen, Programmierparadigma ist die Sprunganweisung, Visualisierung erfolgt mit Flussdiagrammen.
- **1970er-Jahre:** Paradigmenwechsel zur Strukturierung (Funktionsorientierung), dominierende Programmiersprachen verzichten weitestgehend auf die Sprunganweisung, z.B. Pascal und C, Modularisierung, Funktionsbausteine sind zustandslos, Software Engineering erlebt Blütezeit und wird von vielfältigen strukturierten Darstellungstechniken wie strukturierter Analyse (SA), strukturiertem Design (SD), Nassi-Schneiderman-Diagrammen usw. dominiert, definierte Softwareprozesse orientieren sich am Wasserfallmodell.
- **Ab 1990er-Jahre:** Paradigmenwechsel zur Objektorientierung, dominierende Programmiersprachen sind z.B. C++ und Java, Softwarebausteine besitzen Zustände, Software Engineering erlebt einen Vereinheitlichungsprozess der objektorientierten Darstellungstechniken zur Unified Modeling Language (UML), definierte Softwareprozesse orientieren sich am Spiralmodell und werden agil.

Dieser stark verdichtete historische Abriss zeigt, dass der zunehmende Komplexitätsgrad an ausgewählten Punkten zu Paradigmenwechseln geführt hat. Es stellt sich die Frage, ob sich das auf eingebettete Systeme übertragen lässt?

Schaut man sich die dominierende Programmiersprache für eingebettete Systeme an, so stellen wir fest, dass sich C seit geraumer Zeit durchgesetzt hat. Nur noch wenige ausgewählte Aufgabenstellungen werden in Assembler realisiert.

Offensichtlich folgte man hier dem Paradigmenwechsel zur Strukturierung. Es lässt sich sogar eine Kenngröße ausmachen, an der man diesen Wechsel fest-

machen kann. Es ist die Programmgröße für eingebettete Systeme, die Größe des Programmspeichers. Ab 1 bis 16 Kilobyte Programmgröße wird es zunehmend schwieriger, die Aufgabenstellung in Assembler zu lösen. C wurde, obwohl hungriger nach Ressourcen, die adäquatere Programmiersprache für eingebettete Systeme.

Oberhalb der 16 Kilobyte wird nicht mehr ernsthaft darüber diskutiert, das System in Maschinensprache zu programmieren. Für den Paradigmenwechsel zur Objektorientierung lässt sich ebenfalls diese Kennzahl heranziehen. Dieser wurde vollzogen, als die typische Programmgröße deutlich die 1-Megabyte-Grenze überschritten hatte. Spätestens ab 4 Megabyte Hauptspeicher waren alle Diskussionen, ob PC-Programme in C programmiert werden sollen, erledigt. Objektorientierte Programmiersprachen wie C++ und Java haben sich innerhalb kurzer Zeit, zwischen 1990 und 1995, durchgesetzt. Heute verfügen selbst kleine Mikrocontroller wie die ARM-Cortex-M-Familie über mehr als 1 Megabyte Programmspeicher.

1.2 Wie sollte man dieses Buch lesen?

Das Buch spannt einen weiten Bogen über das Fachgebiet der Entwicklung eingebetteter Systeme. Die Modellierung ist die Klammer, die die einzelnen Aspekte zusammenhält. Um das Fachgebiet als Ganzes zu verstehen, sollte das Buch kursorisch gelesen werden. Dabei können Details einzelner Aspekte auch übersprungen werden. Die Abschnitte sind in sich so weit abgeschlossen, dass eine zwingende Reihenfolge für die Bearbeitung nicht vorliegt. Für ausgewählte Leser sind einzelne Abschnitte von besonderem Interesse. Diese sollten intensiver studiert werden. Dafür sind im Anhang wichtige Erklärungen zu finden. Für die praktische Anwendung des Gelernten bietet die Webseite zum Buch unter *www.mdese.de* Werkzeuge, Beispiele und Tutorials an.

Studenten lernen das Fachgebiet Stück für Stück im Überblick kennen und werden für wichtige Aspekte sensibilisiert, deren Inhalte im Studium zu vertiefen sind. Zusammenhänge einzelner Disziplinen werden verstanden und die Möglichkeiten modellgetriebener Technologien können abgeschätzt werden. Dieses Buch ist auch als Nachschlagewerk geeignet.

Entscheider vertiefen ihr Beurteilungsvermögen und werden in die Lage versetzt, moderne modellgetriebene Technologien zu analysieren sowie qualifiziert zu bewerten.

Projektleiter erhalten wichtige Impulse zur Einführung von Modellierungstechniken. Das Verständnis der vorgestellten Technologien ist die Voraussetzung für deren Anwendung. In Kombination mit gesammelter Projekterfahrung gelingt es, für zukünftige Projekte eine neue Synthese mit modellgetriebenen Technologien zu erarbeiten.

Softwareentwickler verstehen den gesamten Prozess der modellgetriebenen Entwicklung und werden in die Lage versetzt, die für sie relevanten Technologien zu beurteilen und anzuwenden. Besonders modellgetriebene Realisierung und Test werden im Detail verstanden.

Hardwareentwickler lernen die Arbeitstechniken der Softwareentwickler kennen und sind in der Lage, mit diesen eine gemeinsame Sprache zu finden.

1.3 Was ist an eingebetteten Systemen so besonders?

Eingebettete Systeme haben im Vergleich zu konventionellen Computern, wie unseren Desktop-PCs oder unseren Notebooks, geringe Ressourcen an Speicher und Rechengeschwindigkeit. Es gibt vielfältige Hardwarearchitekturen von 4 bis 64 Bit. Die verfügbaren Systeme bieten Single- und Multicore sowie sehr unterschiedliche Softwarearchitekturen von Bare-Metal-Programmierung über Real-Time Operation Systems bis Multitask/Multiuser-Betriebssysteme an.

Abb. 1–2 *Programmieradapter und Zielsystem*

Der sinnliche Zugang wird für den Neueinsteiger dadurch erschwert, dass diese eingebetteten Digitalrechner als solche meist nicht zu erkennen, also quasi »unsichtbar« sind. Sie verfügen oft weder über gebräuchliche Eingabegeräte wie Maus und Tastatur noch über grafische Displays. Ein Taster und wenige LEDs bilden in vielen Fällen die einzige Mensch-Maschine-Schnittstelle.

Daher erfordert es auch ein spezielles Equipment für die Programmierung. Der Einsteiger muss sich ein Programmiergerät und wenn möglich Debugger-Hardware für das Zielsystem zulegen.

Zusätzlich sind eine spezielle Entwicklungsumgebung und Compiler nötig, die die gewünschte Hardware auch unterstützen. Die verfügbare Literatur ist entweder proprietär auf die Hardware- und Softwarearchitektur sowie die Entwicklungsumgebung des Chipherstellers fokussiert oder so allgemein gehalten, dass die konkrete Anwendung des Gelernten nur schwer möglich ist.

Von der breit angewendeten Softwaretechnologie im Mikrorechnerbereich ist der Biotop der eingebetteten Systeme eher abgeschnitten.

```
int main() {
  DDRB &= ~(1 << PB0);
  DDRB |= (1 << PB1);

  PORTB |= (1 << PB1);  //PB1 High
  PORTB &= ~(1 << PB1); //PB1 Low
}
```

```
public class HelloWorld
{
      public static void main (String[] args)
      {
            // Ausgabe Hello World!
            System.out.println("Hello World!");
      }
}
```

Abb. 1–3 *Hallo Welt, zwei Welten*

So viel zu den Schwierigkeiten bei der Annäherung an eingebettete Systeme. Es gibt auch Erfreuliches zu berichten: Die Komplexität der Hardware eines eingebetteten Systems ist im Vergleich zu den großen Verwandten PCs, Notebooks und Co. oft noch bis auf die Registerebene durch- und überschaubar. Die gegebene Hardware ist in vielen Fällen bis ins Detail durch den Entwickler selbst determiniert. Es bestehen zahlreiche Alternativen zu einer Architektur, die damit eine entsprechende Vielfalt von Lösungsmöglichkeiten konkreter Aufgabenstellungen eröffnen. Die erlangte Kompetenz ist nicht so einfach nachzubilden und kann dadurch für längere Zeit einen Marktvorsprung darstellen.

1.4 Wie sieht das typische Zielsystem aus?

Dieses Buch fokussiert auf Systeme, die aus softwaretechnologischer Sicht an einem Kipppunkt zu verorten sind. Diese Systeme sind nicht mehr klein, aber auch noch nicht wirklich groß. Sie sind so leistungsfähig und komplex, dass die klassische Entwicklung mit einem Zeileneditor in einer strukturierten Sprache wie C an ihre Grenzen stößt. Es ist sinnvoll, die anstehenden Herausforderungen mit einer modellgetriebenen Entwicklung und in einer höheren Programmiersprache wie z. B. dem objektorientierten C++ zu bewältigen.

Als kleine eingebettete Systeme bezeichnen wir in diesem Kontext Mikrocontroller, meist mit 8-Bit-Verarbeitungsbreite, mit wenigen Kilobyte Programmspeicher und vielleicht maximal 30-MHz-Taktfrequenz.

Abb. 1–4 *8-Bit-Controller im Vergleich zu einer 1-Cent-Münze*

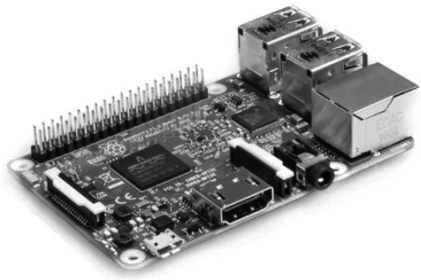

Abb. 1–5 *32 Bit ARM Cortex A8 Singleboard Computer*

Als große eingebettete Systeme bezeichnen wir 32- bis 64-Bit-Multicore-Architekturen, oft mit grafischem Display, einem entsprechenden Betriebssystem, mehreren Gigabyte Speicher und Taktraten im Gigahertz-Bereich. Letztere sind vielleicht noch eingebettete Systeme, aber keine Mikrocontroller mehr, sondern gehören schon zur Klasse der Mikrorechner.

Die kleinen Systeme sind mit den klassischen Mitteln, also hardwarenah in Assembler oder C, beherrschbar. Für die großen Systeme stehen etablierte Betriebssysteme und standardisierte Plattformen zur Abstraktion von der komplexen Hardware zur Verfügung. Zwischen diesen öffnet sich jedoch zurzeit ein großes Feld von Mikrocontrollern, die zum einen zu komplex sind für die althergebrachte Vorgehensweise und zum anderen zu diversifiziert für standardisierte Betriebssysteme.

Da die Übergänge nicht scharf abgrenzbar sind, soll eine Bandbreite von Systemen angesprochen werden, für die eine modellgetriebene Entwicklung, wie sie in diesem Buch beschrieben wird, praktikabel und kommerziell sinnvoll ist (siehe Abb. 1–6). Die untere Grenze bilden leistungsfähige 8- und 16-Bit-Architekturen mit 16 bis 32 KByte Programmspeicher und 1 bis 30 MHz Taktfrequenz. Als typisches Zielsystem sind 32-Bit-Single-Core-Architekturen mit 32 KByte bis 4 MByte Programmspeicher und bis zu 300 MHz Taktfrequenz anzusehen. Die Obergrenze sollten wir bei 32-Bit-Multicore-Systemen mit vielleicht 1 GByte Speicher ziehen.

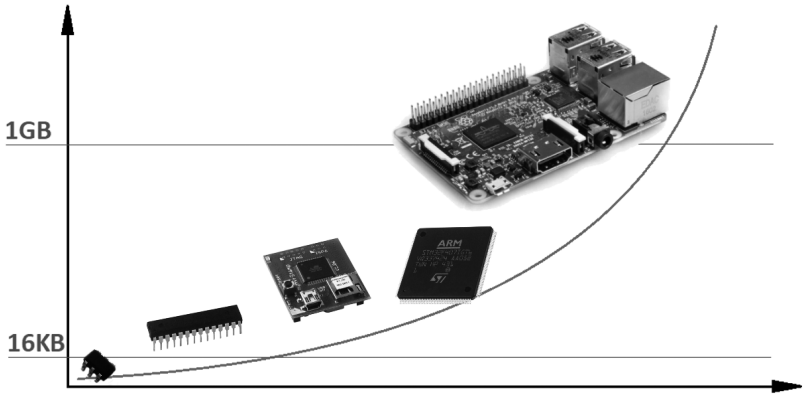

Abb. 1–6 *Bandbreite der Zielsysteme*

In dieser Bandbreite können die hier vorgestellten modellgetriebenen Technologien wertvolle Beiträge zur Qualitätsverbesserung und zum betriebswirtschaftlichen Erfolg leisten. Die zur Vertiefung der Buchinhalte angesprochenen und auf der Webseite verfügbaren Beispiele sind für ein Referenzsystem auf der Basis eines 32 Bit ARM Cortex M4 von Infineon konzipiert. Mit 1 MByte Programmspeicher und 120 MHz Taktfrequenz liegt dieses System genau an der kritischen Schwelle, die den Einsatz modellgetriebener Technologien besonders wirksam werden lässt.

1.5 Fallbeispiele

Die hier kurz charakterisierten Fallbeispiele beziehen sich auf Projekte, die von den Autoren durchgeführt oder begleitet wurden.

1.5.1 Die Anlagensteuerung

Das Projekt »Anlagensteuerung« ist bei einem mittelständischen Unternehmen mit ca. 100 Mitarbeitern zu verorten. Es werden Lüftungsanlagen in Kleinserien und nach Kundenspezifikation gefertigt. Die Entwicklungsabteilung der Firma besteht aus zwei Elektrotechnikingenieuren.

Für die Steuerung der Anlagen haben die Ingenieure im Laufe der Zeit eine kleine, modulare Hardwareplattform entwickelt. Die Ausstattung der Hardware bietet den Grundbedarf an Sensorik, Aktorik und einfachen Benutzerschnittstellen, die für jede Anlage benötigt werden. Zusätzlich kann die Steuerung mit weiteren Sensoren und Aktoren, grafischem Display, Netzwerkanschluss usw. ausgestattet werden.

Vor der Einführung modellgetriebener Technologien bestand das Hauptproblem darin, dass neue oder kundenspezifische Anlagenvarianten von der mechani-

schen und elektrotechnischen Seite her innerhalb weniger Tage realisierbar waren. Die Steuerungssoftware aber, sobald die neuen Anforderungen über eine einfache Parametrisierung hinausgingen, war erst nach Wochen oder gar Monaten verfügbar. Die Auslieferung der Anlagen verzögerte sich und Ressourcen (Kapital, Material, Fertigungs- und Lagerplätze) waren unnötig gebunden.

Mit der Einführung modellgetriebener Technologien wurde die Entwicklungszeit der Software so weit beschleunigt, dass diese jetzt in der Regel mit der Fertigstellung der Hardware zeitgleich zur Verfügung steht. Die Wettbewerbsfähigkeit der Firma wurde mit einer deutlich verkürzten Entwicklungszeit (*Time-to-Market*) entscheidend verbessert.

1.5.2 Die Baumaschine

Ein traditionsreicher Hersteller von robusten Baumaschinen im unteren und mittleren Preissegment sah sich am Markt zunehmendem Wettbewerbsdruck ausgesetzt. Der Verkauf über den Preis konnte nur für begrenzte Zeit die Wettbewerbsfähigkeit erhalten. Als Alternative kam nur eine deutliche Verbesserung der Funktionalität der Baumaschinen infrage.

Die Herausforderung bestand jedoch darin, auf teure Zukaufkomponenten zu verzichten und trotzdem bessere Funktionalität anzubieten. Es wurde die Entscheidung getroffen, in Begleitung durch ein Beratungsunternehmen eine eigene Kompetenz zur Entwicklung von Hydrauliksteuerungen aufzubauen. Dabei wurde von Anfang an auf modellgetriebene Entwicklung gesetzt.

Innerhalb eines Jahres ist es gelungen, eine hauseigene Plattform für Steuergeräte zu entwickeln und vor allem durch modellgetriebene Tests die Konformität mit Normen und Sicherheitsanforderungen begleitend zur Hardwareentwicklung sicherzustellen.

Das Entwicklerteam konnte einen definierten und modellgetriebenen Entwicklungsprozess etablieren. Dabei herrscht ein hoher Grad an Arbeitsteilung. UML-Modelle dienen nicht nur zur Systemgenerierung, sondern auch als normierendes Element in der Kommunikation zwischen den beteiligten Entwicklerteams.

Heute wird ein wesentlicher Anteil der Wertschöpfung des Unternehmens über die Softwareentwicklung erbracht. Verifikation und Validierung neuer Entwicklungen erfolgen bereits auf Modellebene. Nicht nur die Quellcodes und automatisierte Tests, sondern auch weite Teile der Projektdokumentation werden aus den Modellen generiert.

1.5.3 Das Energie-Monitoring-System

Für die wissenschaftliche Untersuchung von besonders energiesparenden Gebäuden (Passivhäusern) wird eine Vielzahl unterschiedlicher Messwerte benötigt. Es müssen Hunderte Sensoren im und am Baukörper verbaut und vernetzt werden. Dabei kommen über verschiedene Kommunikationsmedien sehr unterschiedliche Protokolle und Datenformate zur Anwendung.

Herstellerspezifische Lösungen für moderne Zählersysteme *(Smart Meter)* erschweren die Integration. Das System besteht aus zahlreichen einzelnen Controllern zur Datenerfassung, -übertragung und -aufbereitung.

Die Komplexität der Gesamtlösung lässt sich mit folgenden Parametern umreißen: Individuelle, auf mehrere Hundert unterschiedliche Controller verteilte Softwarelösung, die mehr als zwei Millionen Messwerte am Tag erfasst, verarbeitet und an einen Server überträgt.

Die Lösung wurde innerhalb von weniger als drei Monaten durch ein kleines in der modellgetriebenen Entwicklung erfahrenes Team erstellt und automatisiert auf das Zielsystem übertragen. Späte und nachträgliche Anforderungen des Auftraggebers konnten zuverlässig und zügig implementiert werden. Das System wurde und wird in der Einsatzphase ständig an neue Anforderungen angepasst. Die Anforderungen an das System werden durch Akteure, die an der wissenschaftlichen Auswertung der Daten beteiligt sind, immer wieder erweitert.

Es hat sich herausgestellt, dass modellgetriebene Anpassungen zuverlässiger und um ein Vielfaches schneller sind als vergleichbare Lösungen.

Die Liste der Fallbeispiele lässt sich noch wesentlich länger gestalten. Eines ist allen Erfolgsbeispielen gemeinsam: Der zunächst nicht unerhebliche Aufwand für die Einführung modellgetriebener Technologien hat nicht nur zu moderaten Verbesserungen geführt, sondern zu völlig neuen Dimensionen. Die Auswirkungen sind vor allem in der Softwarequalität, der Prozessqualität, der Entwicklungsgeschwindigkeit und der Wartbarkeit der Systeme festzustellen.

1.5.4 Das Beispielsystem SimLine

Im Buch werden Anwendungsbeispiele anhand eines Referenzsystems erläutert. Dieses Referenzsystem wurde speziell für dieses Buch zusammengestellt und besitzt wichtige Merkmale, die für die genannten Fallbeispiele relevant sind. Zum Beispiel ein grafisches Display wie bei der Anlagensteuerung, ein Fahrzeugbussystem wie in der Baumaschine und Sensorik und Aktorik sowie eine Ethernet-Schnittstelle wie beim Energie-Monitoring-System.

Das Referenzsystem ist das Smart-Home-System SimLine. Es besteht aus einer Plattform, die die Kerninfrastruktur zur Verfügung stellt, um Sensoren und Aktuatoren anzuschließen, die Sensoren auszulesen und über definierbare Regeln die Aktuatoren entsprechend anzusteuern.

Zusätzlich gibt es einzelne SimLine-Sensoren und -Aktuatoren sowie spezielle SimLine-Module mit vordefinierten Regeln, beispielsweise für Einbruch- oder Sturmschutz.

Vertiefende Informationen, Beispielmodelle und Beschaffungsquellen für das SimLine-System finden Sie auf der Webseite zum Buch.

1.6 Das Manifest

Das Manifest *Modeling of Embedded Systems* wurde 2015 von mehreren Personen erarbeitet, die überzeugt sind, dass Projekte den steigenden Anforderungen an die Entwicklung eingebetteter Systeme nur gerecht werden können, wenn sie modellbasierte Methoden verwenden. Gleichzeitig sahen die Autoren, dass trotzdem die Modellierung noch sehr verbreitet eingesetzt wird.

Unter den Autoren und Unterzeichnern des Manifests waren auch Autoren dieses Buches. Das Manifest ist somit auch ein Leitfaden für dieses Buch und Sie werden die einzelnen Thesen des Manifests im gesamten Buch wiederfinden.

Das Manifest postuliert 7 Thesen:

1. ***Beteilige Stakeholder*** durch geeignete domänenspezifische Abstraktionen, Notationen und Sichten.
2. ***Strebe nach langlebigen Modellen*** durch angemessene Abstraktionen, Trennung von Aspekten und Modellierungsrichtlinien.
3. ***Mache Modelle überprüfbar, transformierbar und ausführbar*** durch semantisch klar definierte Sprachen für funktionale und nicht funktionale Aspekte.
4. ***Lerne rechtzeitig*** durch Modelle, die in einer frühen Entwicklungsphase Anforderungen und Spezifikationen überprüfen und Fehler aufzeigen.
5. ***Vermeide Redundanzen und wiederkehrende Arbeiten*** durch Automatisierung und Integration von Modellen für verschiedene Aspekte.
6. ***Mache Modelle zugänglich*** durch eine skalierbare Infrastruktur und benutzerfreundliche und leicht erlernbare Werkzeuge.
7. ***Etabliere eine Modellierungskultur*** durch Ausbildung und Harmonisierung der Modellierungsaktivitäten mit den Entwicklungsprozessen.

Das Manifest wurde erstmals 2015 auf der Konferenz MESCONF *(www.mesconf.de)* in München vorgestellt. Es ist auf einer eigenen Internetseite *www.mdse-manifest.org* publiziert und kann von jedem als Zeichen der Unterstützung unterzeichnet werden.

In Anhang A werden die Thesen des Manifests zur Entwicklung der Skizze eines Reifegradmodells für modellbasierte Softwareentwicklung herangezogen.

Legende

Am Ende der meisten Kapitel gibt es eine Zusammenfassung, die in kurzen Stichpunkten eine Art Essenz bildet. Dabei gibt es folgende Kategorien an Stichpunkten, die entsprechend dem Charakter des jeweiligen Kapitels mehr oder weniger ausgeprägt oder nicht vorhanden sind (z.B. hat Anhang B »Kurzreferenz UML und SysML« keine Zusammenfassung).

- ✓ Checklisten: Die so gekennzeichneten Aussagen sind als Checkliste zu sehen. Sie helfen Ihnen, zu überprüfen, ob Sie die essenziellen Konzepte dieses Kapitels berücksichtigt haben.
- ☞ Hinweise: Die so gekennzeichneten Aussagen weisen noch einmal auf wichtige Aussagen des Kapitels hin.
- Warnungen: Die so gekennzeichneten Aussagen weisen explizit auf Fehlentscheidungen/Fehlannahmen hin, die in der Praxis immer wieder anzutreffen sind und deren Auswirkungen erfahrungsgemäß besonders aufwands- oder kostenintensiv ausfallen.

2 Basiswissen

»Die Wahrheit und Einfachheit der Natur sind immer die letzten Grundlagen einer bedeutenden Kunst.«

(Paul Ernst, 1866 – 1933, deutscher Essayist, Novellist, Dramaturg, Versepiker)

2.1 Was sind eingebettete Systeme?

Als eingebettete Systeme werden Digitalrechner bezeichnet, die für die Steuerung, Überwachung, Regelung oder für andere Aufgaben, wie z.B. eine Signalverarbeitung, in den räumlich-technischen Kontext der zu erfüllenden Aufgabe integriert sind. Das System stellt mit dem möglichen Benutzer die Umgebung des eingebetteten Systems dar. Die Integration in das System bewirkt, dass der meist recht kleine Digitalrechner (Mikrocontroller) als solcher von außen oft nicht wahrgenommen werden kann. Für die zu erfüllende Aufgabe werden Informationen aus der Umgebung über Sensoren erfasst und verarbeitet, die dann wiederum über Aktuatoren auf das System einwirken. Die Benutzerschnittstellen solcher Systeme sind oft sehr schmal und bestehen mitunter nur aus einzelnen LEDs und wenigen Tastern. Da sich dieses Buch mit modellgetriebener Entwicklung eingebetteter Systeme beschäftigt, soll ein solches auch gleich modellhaft dargestellt werden (siehe Abb. 2–1).

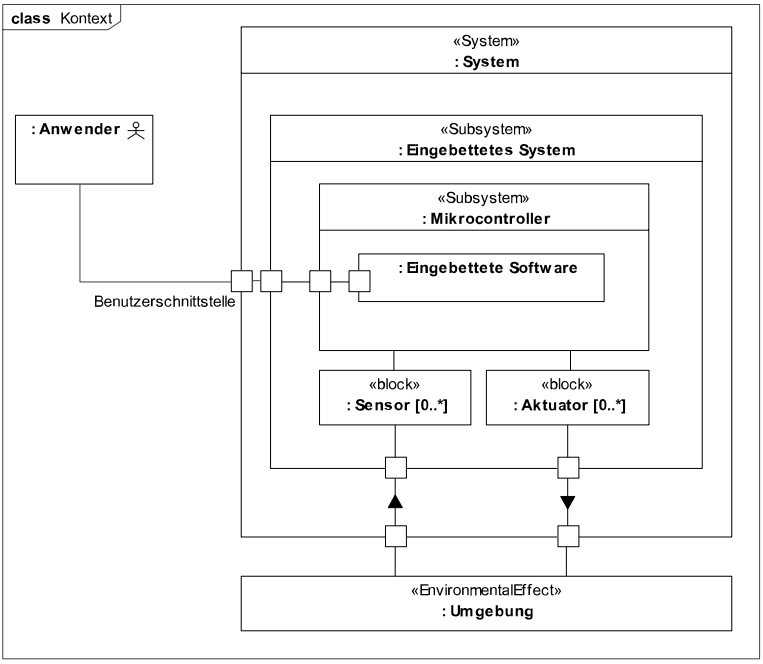

Abb. 2–1 *Ein vereinfachtes Modell eingebetteter Systeme in ihrer Umgebung*

Für ein tieferes Verständnis reicht dieses einfache Modell jedoch nicht aus. Das System selbst verfügt neben dem Mikrocontroller und den Sensoren und Aktuatoren noch über weitere physische Komponenten und steht mit der übergeordneten Umgebung in Wechselwirkung. Solche weiteren physischen Komponenten können z.B. hydraulische oder mechanische Teilsysteme sein. Die physikalischen Größen dieser Wechselwirkungen des Gesamtsystems mit seiner Umwelt sind unter anderem Temperatur, Feuchtigkeit und nicht zu vergessen die elektromagnetische Strahlung. Sensoren und Aktuatoren können nicht nur auf das System, sondern auch auf die Systemumgebung gerichtet sein. Für eingebettete Systeme ist inzwischen die Vernetzung mit anderen Systemen eine Selbstverständlichkeit.

Dieses Buch beschäftigt sich mit der Entwicklung solcher Systeme. Der Softwareentwickler hat seinen ganz eigenen Zugang zum System. Dazu nutzt er Werkzeuge und muss sich an Vorgaben halten. Fassen wir all diese Aspekte zusammen, erhalten wir das folgende Modell (siehe Abb. 2–2).