

# **The Building Block Method**

Component-Based  
Architectural Design  
for  
Large Software-Intensive  
Product Families

Jürgen K. Müller

The work described in this thesis has been carried out at  
Philips Research Laboratories, Eindhoven.

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Müller, Jürgen Karl

The Building Block Method / Jürgen Karl Müller. - Amsterdam: Universiteit van  
Amsterdam, Faculteit der Natuurwetenschappen, Wiskunde en Informatica  
Proefschrift Universiteit van Amsterdam. - Met samenvatting in het Nederlands

ISBN 90-74445-58-6

Keywords: Software Engineering / Software Components / Software Architecture /  
Product Families / Component-Based Design / Aspect Design / Incremental Software  
Development / Software-Intensive Systems

© KONINKLIJKE PHILIPS ELECTRONICS NV 2003

All rights reserved. Reproduction or dissemination in whole or in part is prohibited with-  
out the prior written consent of the copyright holder.

# **The Building Block Method**

## Component-Based Architectural Design for Large Software-Intensive Product Families

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de  
Universiteit van Amsterdam  
op gezag van de Rector Magnificus  
prof.mr. P.F. van der Heijden  
ten overstaan van een door het College voor  
Promoties ingestelde commissie, in het openbaar  
te verdedigen in de Aula der Universiteit  
op vrijdag 23 mei 2003, te 10.00 uur

door

**Jürgen Karl Müller**

geboren te Schwäbisch Hall, Duitsland.

Promotiecommissie:

Promotor: Prof.drs. M. Boasson

Co-promotor: Prof.dr. D.K. Hammer

Overige leden:

Prof.dr. J. Bosch

Prof.drs. T. Bruins

Prof.dr. J van Katwijk

Prof.dr. P. Klint

Prof.dr. R.J. Meijer

Prof.dr. C.A. Szyperski

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

---

# Abstract

One of the critical issues in developing large software-intensive systems is to allow for easy evolution and extension of these systems. For this purpose, the development of a software architecture that supports evolution and extension of the software is of crucial importance. The software architecture describes the decomposition of the software into parts that should be manageable and understandable, and should localise change. To serve a broad range of customers, products are often not developed as single systems but as a family of similar products that share a common base.

This thesis presents a component-based architectural design method for large software-intensive product families. The so-called Building Block Method (BBM) focuses on the identification and design of Building Blocks. Building Blocks are software components that can be independently developed and deployed.

The BBM takes descriptions of application domain functionality, commercial product features, system qualities and technology choices as input and produces a number of architectural models and construction elements.

The identification and specification of Building Blocks are based on three design dimensions, namely object design, aspect design, and concurrency design. Object design is the transformation of application domain objects in several steps into programming language objects. Aspect design is a means for achieving system qualities. Concurrency design maps objects and aspects to computing resources.

Relations between Building Blocks are such that a system can be integrated and tested incrementally. A specific product is configured from a subset of all the Building Blocks that make up the family.

The BBM is described in the form of a core method and a method specialisation for centrally controlled distributed embedded systems. The specialised method has been applied to the development of telecommunication and medical imaging systems.

---

# Samenvatting

Eén van de belangrijke aandachtspunten bij het ontwikkelen van grote software-intensieve systemen is zorg te dragen voor gemakkelijke evolutie en uitbreiding van deze systemen. Om dit te bereiken is de ontwikkeling van een software architectuur die evolutie en uitbreiding ondersteunt van groot belang. De software architectuur beschrijft de decompositie van de software in delen die onderhoudbaar en begrijpbaar moeten zijn en die de invloed van wijzigingen tot een klein aantal van deze delen beperkt houden. Om een grote groep van klanten te kunnen bedienen worden producten vaak niet als aparte systemen ontwikkeld, maar als een familie van soortgelijke producten die een gemeenschappelijke basis delen.

Dit proefschrift presenteert een component-gebaseerde architecturale ontwerp methode voor grote software-intensieve product families. Deze zogenaamde Bouwblok Methode (BBM) richt zich op de identificatie en het ontwerp van Bouwblokken. Bouwblokken zijn software componenten die onafhankelijk ontwikkeld en ingezet kunnen worden.

De BBM gebruikt beschrijvingen van applicatie domein functionaliteit, commerciële product features, system kwaliteiten en technologie keuzes als invoer en levert architecturale modellen en constructie elementen.

De identificatie en specificatie van Bouwblokken is gebaseerd op drie ontwerpdimensies, namelijk objectontwerp, aspectontwerp en procesontwerp. Objectontwerp is de transformatie van applicatiedomein objecten in meerdere stappen in programmertaal objecten. Aspectontwerp is een middel om systeem kwaliteiten te bereiken. Procesontwerp projecteert objecten en aspecten op computationele middelen.

Relaties tussen Bouwblokken zijn zodanig dat een systeem incrementeel geïntegreerd en getest kan worden. Een specifiek product wordt geconfigureerd uit een deelverzameling van alle Bouwblokken.

De BBM wordt beschreven in de vorm van een kern methode en een methode specialisatie voor centraal-gecontroleerde gedistribueerde embedded systemen. De methode specialisatie is toegepast in de ontwikkeling van telecommunicatie- en medische beeldbewerkingsystemen.

---

# Acknowledgements

This thesis has evolved over several years. I started to work on software architecture when I was asked to add subscriber line concentration to the telecommunication switching system family tss. In the end, we extended, refactored and implemented the Building Blocks (BBs) of the equipment maintenance subsystem. I started discussions about the architecture of tss with the architecture team and the chief designers. This had a long time impact as can be seen by the fact that the tss experiences are an important source of this thesis.

There are a lot of people that accompanied me on this way. First of all I am grateful to my supervisors Maarten Boasson, my promotor of the University of Amsterdam, and Dieter Hammer, my co-promotor of the University of Eindhoven, for continuous support and advice over these years. This thesis undoubtedly benefited from Maarten Boasson's thoroughness, insightful comments, and penetrating criticism. With enormous endurance, he provided feedback on various versions of the thesis. Through his critical questions and careful listening he helped me to formulate more clearly what I wanted to say. Second, I am indebted to Dieter Hammer, who provided valuable suggestions for description of the Building Block Method and connection with other methods.

I would like to thank Lothar Baumbauer, the main architect of tss, and the tss development team, of which I would like to name especially Wolfgang Kelz, Lutz Pieger, Hubert Frosch, Helmut Merz, Georg Wenzel, and Richard Bernheine for their engineering excellence in designing and implementing the tss product family. With Lothar I discussed many issues of system architecting and the Building Block Method in particular. The Building Block research project at PKI Nuremberg was created to compare Building Blocks and object-oriented programming. This led to a very fruitful collaboration with people from Philips Research Eindhoven namely Frank van der Linden, Reinder Bril, Jan Gerben Wijnstra and Rene Krikhaar.

The first idea about what I later called *design dimensions* arose when I left the tss development to help with the development of GSM infrastructure systems. I asked myself what made the tss software structure so much better understandable, better performing and extensible than the GSM software. Naturally, there

were many reasons, but the underlying model of Building Blocks, aspects and processes was at the core. After moving to research, I started to publish about the BB experiences together with Frank van der Linden who is co-author of several of my publications.

The BAPO/CAFRCR framework of which the CAFRCR part serves as contextual model for the BBM was developed together in the Composable Architectures team with Pierre America, Hans Jonkers, Gerrit Muller, Henk Obbink, Rob van Ommering, William van der Sterren and Jan Gerben Wijnstra.

I thank Detlef Prescher for discussions about formalisations of the BBM, Karin Müller for valuable suggestions concerning layout and the use of the English language, and Frans Reijnhoudt and Rik Willems for good discussions about engineering in general. Special encouragement and support in critical times and constructive comments for the thesis I received from Clemens Szyperski and Gerrit Muller.

I thank my bosses at Philips Research Henk Obbink, Otto Voorman and Jaap van der Heijden for their support. Henk Obbink and Gerrit Muller had the vision to improve current software engineering business by consolidating and systemising best practices. I am also indebted to several of my colleagues, who have read different parts of my thesis in various stages of its evolution and provided comments: Lothar Baumbauer, Reinder Bril, Robert Dekkers, Angelo Hulshout, Auke Jilderda, Jurjen Kranenborg, Rene Krikhaar, Frank van der Linden, Gerrit Muller, Andre Postma, Joost Reuzel, Marc Stroucken, William van der Sterren, Jan Gerben Wijnstra and Rik Willems.

Last, I would also like to thank the members of the reading commission for reading and for approving my thesis: Jan Bosch (University of Groningen), Theun Bruins (University of Amsterdam), Jan van Katwijk (University of Delft), Paul Klint (University of Amsterdam), Rob Meijer (University of Amsterdam) and Clemens Szyperski (Queensland University of Technology).

---

# Table of Contents

<b>Abstract</b> .....	<b>i</b>
<b>Samenvatting</b> .....	<b>ii</b>
<b>Acknowledgements</b> .....	<b>iii</b>
<b>Table of Contents</b> .....	<b>v</b>
<b>List of Figures</b> .....	<b>xi</b>
<b>List of Tables</b> .....	<b>xv</b>
<b>1 Introduction</b> .....	<b>1</b>
1.1 Motivation .....	1
1.2 Goals .....	3
1.3 Contribution .....	4
1.4 Outline .....	5
<b>2 Context</b> .....	<b>7</b>
2.1 What is a system? .....	7
2.2 What is architecture? .....	10
2.3 What is a method? .....	11
2.4 Models and Meta-Models .....	12
2.5 About Some Terms .....	13
2.6 A Rational Architecting Model .....	14
2.6.1 Customer Business Modelling .....	15
2.6.2 Application Domain Modelling .....	16
2.6.3 Commercial Product Design .....	19
2.6.4 Architectural Design .....	22
2.6.5 Technology .....	25
2.6.6 Feedback, Navigation and Learning .....	25
2.7 Levels of Consolidation .....	26
2.8 Historical Background of the BB Method .....	27

2.9	Overview of the Concepts	29
<b>3</b>	<b>The Core Method Overview</b>	<b>31</b>
3.1	Prerequisites of the BBM	32
3.2	Main Design Tasks of the BBM	33
3.2.1	Object Design	34
3.2.2	Aspect Design	34
3.2.3	Concurrency Design	35
3.2.4	Composability Design	36
3.2.5	Deployability Design	38
3.2.6	Implementation of BBs	39
3.2.7	Design Artifacts	39
3.2.8	Stopping Criteria for Design Tasks	43
3.3	Design Dimensions	44
3.4	System-Quality-Based Design Tasks	48
3.4.1	Performance Design	48
3.4.2	Reliability Design	49
3.4.3	Security Design	50
3.4.4	Extensibility Design	50
3.4.5	Integratability and Testability Design	50
3.5	Other Design Tasks	50
3.5.1	Feature Mapping Design	51
3.5.2	Architectural Style Design	51
3.5.3	Data Structure and Algorithmic Design	51
3.5.4	Resource Usage Design	52
3.5.5	Interface Design	52
3.5.6	COTS-Based Design	52
3.6	Qualities of the BBM	53
<b>4</b>	<b>Object Design</b>	<b>55</b>
4.1	Domain-Induced Objects	55
4.1.1	Domain Object Model	56
4.1.2	System-Relevant Functionality	57
4.1.3	System Qualities	58
4.1.4	Hardware-Implemented Functionality	58
4.1.5	Hardware-Managing Objects	58
4.2	Design Objects	59
4.2.1	Classification of Functionality in Layers	59
4.2.2	Communication Objects	62
4.2.3	Interface Objects	62

4.2.4	Registry Objects . . . . .	62
4.2.5	Container Objects . . . . .	63
4.2.6	Functional Objects and Object Functions . . . . .	63
4.3	Example: Layers and Some Objects of tss . . . . .	64
4.4	Explicit Transition . . . . .	65
<b>5</b>	<b>Aspect Design . . . . .</b>	<b>67</b>
5.1	Definition of an Aspect . . . . .	67
5.2	Architectural Concern Analysis . . . . .	70
5.2.1	The Analysis Steps . . . . .	71
5.2.2	Architectural Concern List Examples . . . . .	73
5.2.3	System Qualities and Available Technology . . . . .	77
5.3	Starter Sets for Aspect Identification . . . . .	78
5.4	List of Aspects for the Product Family . . . . .	80
5.5	Designing Aspect Functionality . . . . .	81
5.6	Aspects and Building Blocks . . . . .	82
5.6.1	Aspect Completeness of Building Blocks . . . . .	83
5.7	Further Usage of Aspects . . . . .	83
5.7.1	Aspects and Reviews . . . . .	83
5.7.2	Aspects and Documentation . . . . .	84
5.7.3	Aspects and Implementation . . . . .	84
5.8	Aspects and the Whole . . . . .	84
<b>6</b>	<b>Concurrency Design . . . . .</b>	<b>87</b>
6.1	Using Address Spaces . . . . .	87
6.2	Concurrency Design . . . . .	88
6.2.1	Determining Concurrency . . . . .	89
6.2.2	Thread Interaction . . . . .	92
6.2.3	Concurrency and Aspects . . . . .	92
6.2.4	Example: Concurrency Design of tss . . . . .	92
<b>7</b>	<b>Building Block and Deployability Design . . . . .</b>	<b>95</b>
7.1	Composability Design Overview . . . . .	96
7.2	Interfaces . . . . .	99
7.2.1	Abstraction Interfaces . . . . .	99
7.2.2	Open Implementation Interfaces . . . . .	100
7.2.3	Interfaces, Components and Connectors . . . . .	101

7.2.4	Registration and Call-Back Interfaces . . . . .	102
7.2.5	Interfaces, Aspects and Concurrency . . . . .	103
7.3	Component Models . . . . .	103
7.4	Layering. . . . .	104
7.4.1	Layering Principles . . . . .	104
7.4.2	Incremental Layering . . . . .	107
7.4.3	More Facets of Layering . . . . .	108
7.4.3.1	Conceptual versus Strict Layering . . . . .	108
7.4.3.2	Opaque versus Transparent Layering . . . . .	108
7.4.3.3	Partial versus Complete Layering. . . . .	110
7.4.3.4	Communication and Layers . . . . .	110
7.4.4	Example: tss Subsystems . . . . .	111
7.5	Generic and Specific Functionality . . . . .	112
7.5.1	Generic and Specific BBs . . . . .	113
7.5.2	Generic and Specific BB Roles . . . . .	115
7.5.3	Generics and Layering . . . . .	115
7.5.4	Classification of Generic BBs . . . . .	116
7.6	Interfaces Revisited . . . . .	123
7.7	Grouping of BBs . . . . .	125
7.8	Architectural Skeleton . . . . .	126
7.9	Deployability Design . . . . .	128
<b>8</b>	<b>Family Architecture . . . . .</b>	<b>135</b>
8.1	Product Diversity and Features . . . . .	135
8.1.1	Feature Description . . . . .	136
8.1.2	Feature Relations . . . . .	136
8.1.3	Feature List . . . . .	137
8.2	Implementation Means for Diversity . . . . .	138
8.3	Product Family Architecture . . . . .	139
8.3.1	Feature Orientation . . . . .	139
8.3.2	Family Patterns . . . . .	140
8.3.3	Extensibility . . . . .	141
8.3.4	Feature Mapping . . . . .	143
8.3.5	Example: tss Feature BBs . . . . .	146
8.4	Managed Evolution . . . . .	147
<b>9</b>	<b>Comparison With Other Methods . . . . .</b>	<b>149</b>
9.1	The Architectural Meta-Model of the BBM . . . . .	149

9.2	Traditional Development Methods	150
9.2.1	Structured Design	150
9.2.2	Bare Operating Systems and Real-Time Kernels	151
9.2.3	SDL	151
9.2.4	ROOM	152
9.2.5	OMT	152
9.2.6	Object-Oriented Software Engineering	154
9.2.7	Comparison with Traditional Development Methods	155
9.3	Architectural Approaches	156
9.3.1	Architectural Styles	156
9.3.2	Soni	157
9.3.3	4+1Model	159
9.3.4	Comparison with Soni and 4+1	159
<b>10</b>	<b>Method Specialisation</b>	<b>163</b>
10.1	System Architecture	164
10.1.1	Functional Architecture	164
10.1.2	Control Architecture	165
10.2	Additional Guidelines	168
10.2.1	Object Design	169
10.2.2	Composability Design	171
10.2.3	Aspect Design	175
10.2.4	Deployability Design	177
<b>11</b>	<b>Organisational and Process Issues</b>	<b>181</b>
11.1	The Process of Architecting	181
11.2	Development Processes	182
11.2.1	Initial Stage and Steady Stage Development	182
11.2.2	Initial Stage and Steady Stage Processes	182
11.3	Building Blocks are Stable for Subsequent Phases	183
11.4	Building Blocks and the Waterfall Model	183
11.5	Documentation	184
11.6	Layered Development Processes	185
11.7	Incremental Integration and Test	186
11.8	Tool Support	187
11.9	Organisational Consequences	189
<b>12</b>	<b>Conclusion</b>	<b>191</b>
<b>Appendix A</b>	<b>The tss Product Family</b>	<b>195</b>

A.1	tss Introduction .....	195
A.2	System Architecture .....	197
A.2.1	Hardware Components .....	197
A.2.2	Redundancy .....	199
A.2.3	Software Architecture .....	201
A.3	The SW Architecture of the CC .....	201
A.3.1	Object Design .....	203
A.3.2	Functionality of the CC Subsystems .....	203
A.3.3	Aspect Design .....	204
A.3.3.1	tss SW Aspects .....	204
A.3.3.2	The System Quality Reliability in tss .....	207
A.3.3.3	The tss State Model .....	208
A.3.3.4	The tss Recovery Model .....	210
A.3.4	Concurrency Design .....	214
A.3.5	Building Block Design .....	216
A.3.5.1	The tss Component Model .....	217
A.3.6	Generics and Specifics .....	222
A.3.6.1	Abstraction Generics .....	222
A.3.6.2	System Infrastructure Generics .....	223
A.3.6.3	Resource Generics .....	224
A.3.6.4	Layer Access Generic .....	227
A.3.7	Self-Describing Components .....	228
A.3.8	EM Layer Structure .....	228
A.3.9	The Use of Heuristics Within tss .....	231
A.4	Making Products From BBs .....	242
A.4.1	Construction Set .....	242
A.4.2	Product Tailoring and Evolution .....	243
A.4.3	Product and Site Configuration .....	246
A.5	Experience Data .....	249
A.5.1	The tss Products .....	249
A.5.2	tss System Performance Data .....	251
A.5.3	Software Sizes .....	252
A.5.4	Comparing Generics and Specifics .....	253
A.5.5	Inter-Product Reuse .....	255
	<b>References .....</b>	<b>261</b>
	<b>Index .....</b>	<b>269</b>

---

## List of Figures

Figure 1:	System Theory: A System . . . . .	7
Figure 2:	System of Interest and Application Domain . . . . .	9
Figure 3:	System Functionality Origins . . . . .	9
Figure 4:	Model for Architecting . . . . .	14
Figure 5:	Application Domain Modelling . . . . .	16
Figure 6:	Feature-Centric Transition . . . . .	20
Figure 7:	Feature Matrices . . . . .	20
Figure 8:	Base Products and Features . . . . .	21
Figure 9:	BBM: Input - Output Specification . . . . .	24
Figure 10:	Dependent Functional Block Structure . . . . .	27
Figure 11:	Independent Functional Block Structure . . . . .	28
Figure 12:	Feature-Oriented Application Structure . . . . .	28
Figure 13:	Concepts of the BBM and their Main Relations . . . . .	29
Figure 14:	Prerequisites for the BBM . . . . .	32
Figure 15:	Main Design Tasks . . . . .	33
Figure 16:	Aspects and Domain-Induced Objects . . . . .	35
Figure 17:	Thread Identification . . . . .	35
Figure 18:	BB and Objects . . . . .	36
Figure 19:	Mapping of Objects, Aspects and Threads to BBs . . . . .	37
Figure 20:	Dependency Relation Between BBs . . . . .	37
Figure 21:	Identification of Deployment Sets . . . . .	39
Figure 22:	Input + Output of Design Tasks . . . . .	40
Figure 23:	Three Design Dimensions . . . . .	45
Figure 24:	Mapping of Domain Model to Software . . . . .	56
Figure 25:	Examples of Sources of Objects . . . . .	59
Figure 26:	Initial Two Layers . . . . .	60
Figure 27:	Three Layers with Basic and Advanced Applications . . . . .	61
Figure 28:	Four Layers with Operating Infrastructure . . . . .	61
Figure 29:	tss Layered Subsystems . . . . .	64
Figure 30:	Architectural Concern Analysis . . . . .	73
Figure 31:	Examples of SW Aspect Stimuli . . . . .	78
Figure 32:	Aspect Structuring of Building Blocks . . . . .	82
Figure 33:	Multi-View Approaches . . . . .	85

Figure 34:	Requires and Provides Interfaces . . . . .	100
Figure 35:	Abstraction and Open Implementation Interfaces . . . . .	101
Figure 36:	Call Back Mechanism . . . . .	102
Figure 37:	User vs. HW Technology Layering . . . . .	105
Figure 38:	Abstraction from HW . . . . .	106
Figure 39:	Generic vs. Specific . . . . .	106
Figure 40:	Partial Layering . . . . .	110
Figure 41:	Indirect Peer-to-Peer Communication . . . . .	111
Figure 42:	Generic BB and Specific BBs . . . . .	114
Figure 43:	Generic and Specifics with Interfaces . . . . .	116
Figure 44:	Abstraction Generic . . . . .	117
Figure 45:	Connectable Resource Generic and Resource Flow . . . . .	118
Figure 46:	System Infrastructure Generics . . . . .	120
Figure 47:	Layer Access Generic . . . . .	122
Figure 48:	System Structure with HW Mirroring in EM . . . . .	124
Figure 49:	BBM Interfaces . . . . .	125
Figure 50:	Architectural Skeleton . . . . .	127
Figure 51:	Basic Pattern for Diversity . . . . .	140
Figure 52:	Regular Layered Diversity . . . . .	141
Figure 53:	Feature Relation and BB Relation . . . . .	144
Figure 54:	Application Feature Implementation Relation . . . . .	145
Figure 55:	Peripheral Card Maintenance . . . . .	146
Figure 56:	Soni's Architectural Model . . . . .	158
Figure 57:	4+1 Architectural Model . . . . .	159
Figure 58:	Tree-Type Control Structure . . . . .	165
Figure 59:	Three Stage Control Communication Structuring . . . . .	166
Figure 60:	Connection Structure of a Central Controller . . . . .	168
Figure 61:	Managed Object . . . . .	169
Figure 62:	Mapping of External Objects to Internal Objects . . . . .	170
Figure 63:	The Basic Two Layers . . . . .	171
Figure 64:	Three Layers . . . . .	172
Figure 65:	Four Layers with Multi-site Resources . . . . .	172
Figure 66:	Four Layers with Basic and Advanced Applications . . . . .	173
Figure 67:	Four Layers with Operating Infrastructure . . . . .	173
Figure 68:	Control spheres of EM . . . . .	174
Figure 69:	Communication Relations of EM . . . . .	174
Figure 70:	Relations between CM,FM and PM . . . . .	177
Figure 71:	Documentation Dependencies . . . . .	185
Figure 72:	Layered Processes . . . . .	186
Figure 73:	DDD and Generators . . . . .	188
Figure 74:	The Architect's Depth of Understanding . . . . .	189

Figure 75:	Switching Systems in Context . . . . .	196
Figure 76:	tss Hardware Architecture . . . . .	198
Figure 77:	Three Peripheral Groups . . . . .	199
Figure 78:	Layered Subsystems . . . . .	201
Figure 79:	Peer-To-Peer Communication . . . . .	202
Figure 80:	Mapping of Objects to Layers . . . . .	203
Figure 81:	tss State Model . . . . .	209
Figure 82:	Recovery Phase Hierarchy . . . . .	213
Figure 83:	tss Addressing Scheme . . . . .	218
Figure 84:	Service Interface of a BB descriptor . . . . .	219
Figure 85:	Call-back Registration . . . . .	220
Figure 86:	Recovery Interface of the BB descriptor . . . . .	221
Figure 87:	Generic and Specifics with Interfaces . . . . .	222
Figure 88:	System Infrastructure Generics . . . . .	223
Figure 89:	Connectable Resource Generic and Resource Flow . . . . .	225
Figure 90:	Layer Access Generics . . . . .	227
Figure 91:	Tree-Type Control Structure . . . . .	229
Figure 92:	System Structure with HW Mirroring in EM . . . . .	230
Figure 93:	Evolving the Construction Set . . . . .	243
Figure 94:	Development Steps to Extend the Construction Set . . . . .	244
Figure 95:	Overview of the DDD . . . . .	245
Figure 96:	Process Steps for Configuring a Product Instance . . . . .	247
Figure 97:	Process Overview of Product and Site Configuration . . . . .	248
Figure 98:	Construction Sets and Projects . . . . .	256
Figure 99:	Empirical Data on the Distribution of Efforts . . . . .	260



---

## List of Tables

Table 1:	Analogy of Models . . . . .	12
Table 2:	Overview of Steps per Design Task . . . . .	41
Table 3:	Examples of Separation of Generic and Specifics . . . . .	112
Table 4:	System Design Concepts of OMT . . . . .	153
Table 5:	Comparison of Architectural Meta-Models . . . . .	161
Table 6:	Typical Functional Distribution Over The Stages . . . . .	167
Table 7:	Procedure Call Translation Scheme . . . . .	219
Table 8:	Application of Heuristics in tss . . . . .	231
Table 9:	Products of tss . . . . .	250
Table 10:	Central Controller Software Sizes . . . . .	252
Table 11:	tss Software Sizes . . . . .	252
Table 12:	PGC Generic . . . . .	253
Table 13:	PGC Specific BBs . . . . .	253
Table 14:	CASA Generic . . . . .	254
Table 15:	CASA Specific BBs . . . . .	254
Table 16:	Basic Project Data . . . . .	258
Table 17:	Number of BBs per Category . . . . .	258
Table 18:	Number of DSIs . . . . .	258
Table 19:	Degree of Reuse of Application Software . . . . .	259



---

# 1 Introduction

---

## 1.1 Motivation

The motivation for the work presented in this thesis comes from current trends in the development of large software-intensive systems. Software has become an essential technology in the development of almost all complex systems. This is exemplified by the developments in the area of electronic products where software has become the flexible counterpart to chip technology. Functionality, which undergoes evolution and extension, is more and more implemented in software.

Families of products are conceived to cover a wide functionality spectrum using a common base. Product platforms are developed, with which extended or new functionality can later be offered to the customer in form of additional features. In general, the amount of software increases in complex systems.

To cope with these trends, software systems need to be increasingly modular and need to be built from reusable components. However, splitting functionality is only one side of the coin. The increased modularity makes system integration a pivotal step in the development of a system. Architecture plays a crucial role in handling modularity and achieving required system properties.

Current software development very often results in software systems which do not exhibit the kind of modularity which facilitates integration, evolution and extension. A frequent cause is that initially required functionality is taken as the sole basis to develop the architecture. Subsequently required features come as a surprise and are implemented in an ad hoc manner. Another cause is system integration not being planned for but entered unprepared resulting in long integration times where a lot of the global consistency has to be established after the fact. Furthermore, the architecture is not updated and maintained as necessary. The resulting lack of clear structure makes evolution and extension increasingly difficult.

Therefore, development of a product and its architecture needs to be supported systematically through appropriate methods. Current software design methods do not sufficiently address these issues.

Object-oriented design methods take the notion of an object as central point. This blurs the distinction between the modelling of an application domain with building a system. The notion of an object is quite different in these two areas. Application domain modelling captures the perception of a domain and uses an intuitive notion of an object to denote things. This led to the proposal of using the less restrictive notion of a feature instead of an object (see section 2.6.2). The notion of an object in building a system is quite different. An object encapsulates state and behaviour. Furthermore, the idea of a seamless transition between application domain modelling and system building ignores the quite different intention of the two tasks (in chapter 9 a more extensive analysis of other design methods is given).

Another problem is the search for a natural modelling of the software with respect to the application domain. This implicitly assumes that the domain functionality is the major part of the system functionality. In practise, this is often not the case. Take the example of the tss system (see appendix A) where the call switching functionality amounts to 20% of the total functionality while the rest of the functionality is in response to system qualities. This percentage may be higher for other systems but system quality induced functionality will always be a considerable part of a system's functionality.

A third problem is that design methods do not take into account that for industrial products a commercial design activity takes place which determines the functionality of the products from a commercial perspective, identifying necessary and optional product features and their market priority. The priority of features of products is an important input for their timely development.

Finally, the advancement of SW component models makes the development of SW components technically feasible. However, methods are missing which yield the full power of component-based development: development of products from such components that the most likely product evolution and extensions can be done by changes to a small number of components only or by the development of a few new components.

---

## 1.2 Goals

The purpose of this work is to present a component-based architectural design method for the development of large software-intensive systems. The method, called the Building Block Method (BBM), focuses on the transition from domain-oriented decomposition to construction elements of a product family. A number of architectural models are developed to guide this transition. The name Building Block Method refers to its characteristic of supporting SW components, called Building Blocks (BB).

The BBM is designed to support the creation of product family architectures. Its main focus is identification and design of components including component frameworks and system infrastructure generics for the development of large-scale SW-intensive product families. An architectural skeleton of component frameworks and system infrastructure generics, which are stable for a whole product family, is one of the design goals of the BBM. Composing a product from pre-manufactured components is a way in which software reuse in a product family architecture can be achieved. The BBM addresses issues of configurability, in particular configuration to minimum; aspect design which complements object and concurrency design; factoring out into frameworks; extensibility; incremental integration and testing.

The BBM takes application domain functionality, commercial product features, system qualities and technology choices as input and produces a number of architectural models and construction elements. The main architectural model is an acyclic dependency graph of BBs to allow incremental integration and testing. Models for aspect designs, concurrency design and deployability design complement this model. The construction elements include the BBs, their various roles and their designs.

Architecture can only lay a good foundation on which the system is built. A good product needs more than a good architecture alone. Mistakes may be made in any of the various phases of system development: requirements, architecture, detailed design, implementation, deployment or documentation, to name the most important ones. An architectural design method needs to connect to the other phases. This connection is given in section 2.6 by the broader architecting context, which we assume for the BBM and in chapter 11 by the development process embedding.

We present the BBM in two steps:

a core method applicable to the design of large software system in general, and

a specialised method for designing centrally-controlled distributed embedded systems.

Experiences with parts of the method were gained in the development of systems in telecommunication [LM95b] and in medical imaging [Wij00]. One of the telecommunication product families, called Telecommunication Switching System (tss), from which the BBM has been bootstrapped, will also be used as an example of a concrete design made with this method. The product family is described in appendix A. Our experience comes from participation in the development of that product family. We developed design guidelines, redesigned the equipment maintenance subsystem and participated in its implementation (see section A.3.8 and section A.5.3). Since then, the BBM has been further consolidated and in parts be published. This thesis presents a systematic treatment of the method.

---

### 1.3 Contribution

This thesis makes several contributions to architectural design for large software-intensive systems. It introduces:

a component-based architectural design method for software-intensive product families,

aspect design which adds function design to object design (see chapter 5),

object design, aspect design and concurrency design as three design dimensions of architectural design (see section 3.3),

embedding of architectural design into an industry-proven rational architecting process (see section 2.6),

commercial product features to guide architectural design (see chapter 8), and

an incrementally integratable, extensible, component-based product family architecture (see section 8.3).

Variable functionality of a product family that goes beyond parameter values is encapsulated in SW components. SW components are the way to represent both common and variable parts of a product of a family. Aspect design deals with

functionality of large SW systems that is not derived directly from the application domain but is a consequence of quality requirements. It furthers conceptual integrity because aspect designs are orthogonal to object structures. The three design dimensions are key dimensions for designing functionality of large SW products. The freedom to assign functionality to SW components addresses the problem of the so-called tyranny of the dominant decomposition. This tyranny denotes the problem that the application of a certain development approach forces the same kind of decomposition upon all problems, for instance an object-oriented method will always lead to an object decomposition.

An explicit rational process for architecting relates the architectural design to other tasks which are necessary to develop SW products in industry. Commercial product features provide an important anchor for product modularity. The modularity induced by application domain objects is complemented by modularity induced by commercial features. Incremental integration based on layering of BBs is an important means to accelerate the integration and testing process of large software products.

Some of the material of this thesis has been published before<sup>1</sup>. The concept of the composition of a product of a family from components has been published in [LM95a], integration of architectural design into the development process in [Mül95]. A general overview of the BBM was published in [LM95b]. How to relate features of a product with its software structure was published in [Mül97]. Aspect-oriented design as an additional concept to object-oriented design was elaborated in [Mül99].

---

## 1.4 Outline

In chapter 2 we will describe the context for the concepts introduced in this thesis. There we will define important terms such as system, architecture and method, describe a development context in which the BBM is intended to be used, give a historical background of the BBM, and sketch the BBM concepts and their relations within the method.

- 
1. Besides the publicly accessible publications, a number of internal publications have been made. [LM95c] was the basis for the [LM95b], it contains examples which had to be left out because of space limitations. [LM95d] presents the BBM as a number of patterns. [LM97] explains the concept of virtual processors which enables budgets of processor time to be allocated to a group of processes.

With chapter 3 the main part of the thesis starts. It gives an overview of the BBM and shows how additional design tasks are related to the BBM.

The following three chapters explain these design dimensions in detail. Designing objects is dealt with in chapter 4. Chapter 5 introduces SW aspects as a functional structure orthogonal to the object structure. In chapter 6 the design of threads and processes is shown.

Chapter 7 explains the concepts that are used to define manageable and composable BBs. Furthermore, it describes how deployment sets are built. Chapter 8, then, introduces the concept of product family architecture. Its main point is that product features that are implemented in software are related to BBs.

Chapter 9 compares the BBM with other methods and approaches. The comparison will be based on the architectural meta-model of the various methods and approaches.

In chapter 10 a method specialisation for centrally controlled distributed embedded systems is presented.

Chapter 11 explains the consequences of the BBM for both the development organisation and the development process. One of the main points is the layered development process, that is, the architecture is developed in a process which guides the development process of BBs.

Chapter 12 discusses what has been achieved by the BBM and concludes the thesis.

Appendix A gives an introduction to the Telecommunication Switching System (tss) product family, which is used throughout the thesis as an example illustrating a BBM-based design.

### **Formatting**

To stress the main train of thought potentially distracting details have been formatted as insets. This paragraph is an example of an inset. While such insets may contain interesting information for the curious and patient reader, they may safely be skipped when aiming at an understanding of the *big picture*.

*Heuristic: Heuristics about the execution and application of the method are also presented as separately formatted paragraphs such as this one.*

---

## 2 Context

The successful use of a method requires an understanding of both the method and its intended application area. In this chapter we give important definitions, such as our definition of the terms technical system, architecture and method, and describe the specific system context for which the BBM has been designed. We will start with a definition of the term system.

---

### 2.1 What is a system?

A *technical system* is an assemblage of parts forming a complex or unitary whole that serves a useful purpose [Wie98b].

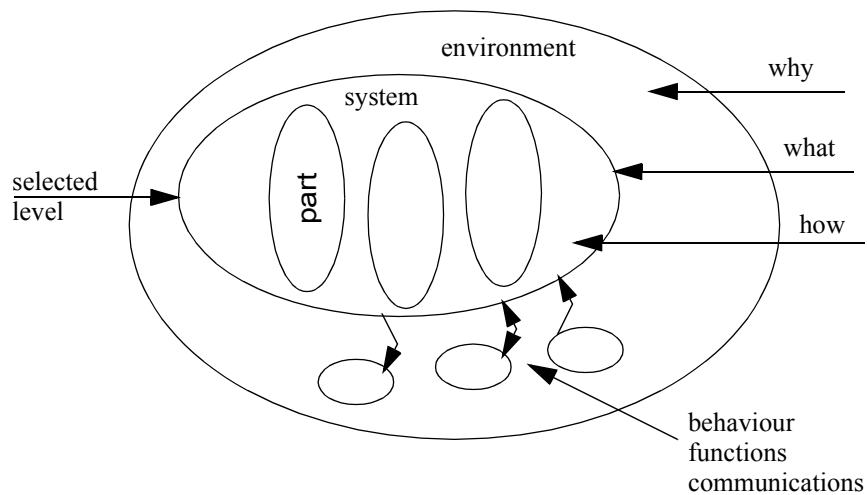


Figure 1: System Theory: A System

Since we only refer to technical systems in our context we shall use the term *system* instead.

The parts must interact in a way that ensures that the system as a whole has a useful function for one or more entities in its environment (see figure 1). Systems may be hierarchically structured, that is, a system is part of an enclosing supersystem and the parts may be subsystems themselves. It is therefore important to indicate the system of interest one is referring to.

A system delivers a service to its environment by interacting with it. Interactions can be described as functions. The functions communicate with the environment. The ordering of interactions in time is called behaviour [Wie98b]. Functions, communications and behaviour are properties of a system.

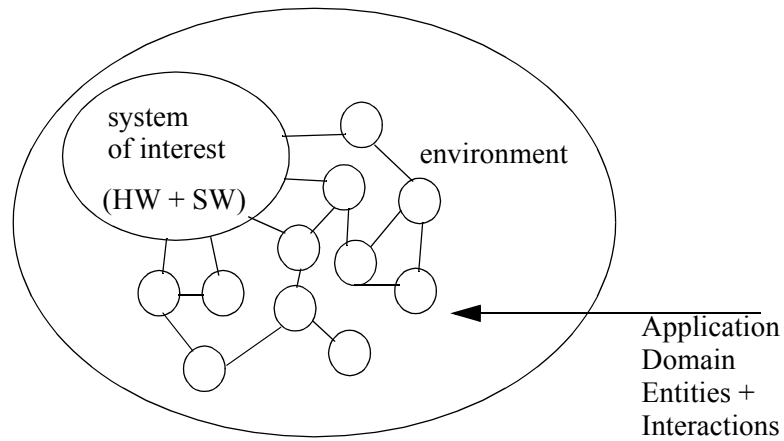
The recursive notion requires to indicate the selected level when we talk about a system (see figure 1). The enclosing system is also called the environment of a system and establishes its operational context (the *why* of the system). The functionality of the system describes the *what* of the system and its internal structure the *how* of the system.

A system is developed according to a set of requirements. Requirements are often described via externally observable properties of the system. These properties concern interactions of the system with entities in the environment. The software system handles symbols whose meanings refer to items outside the software system. The part of the world referred to by these symbols is called the application domain of the system.

Wieringa uses the term *subject domain*, that is, the symbols refer to the subject domain of the system's external interactions [Wie98a].

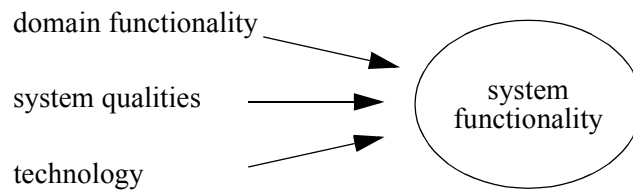
Sometimes an explicit application domain model represents the application domain. The application domain model consists of domain entities and of functions, which describe relevant interactions between domain entities and the system [Wie98a] (see figure 2).

The functionality of a system, however, is not only determined by its interactions with the environment but also by the precise characteristics of these interactions. The interactions have to exhibit certain qualities. One often speaks collectively about system qualities like reliability, performance and security, abstracting from the fact that for different interactions the quality requirements are different. Furthermore, the characteristics of the used technology may influence the system functionality; for instance, potential failures may require error handling functionality. To summarise, system functionality is induced by the



*Figure 2: System of Interest and Application Domain*

required application domain functionality, the qualities required of the system and the used technology (see figure 3).



*Figure 3: System Functionality Origins*

For the context of the BBM, the software of the system is the main focus of attention. However, it is clear that the software is part of the larger system enclosing software and hardware and of the system which form the environment (see figure 2). In the remainder of the thesis, we will simply use the term system to imply software and hardware, and we will use the term environment for the enclosing system. For the term application domain we will sometimes use simply domain.

---

## 2.2 What is architecture?

As a point of departure we take a definition given by Rechtin:

"..., the term *architecture* is widely understood and used for what it is - a top-down description of the structure of the system." ([Rec91], p.9).

The word architecture, traditionally established in the field of construction, has been used for some time in different engineering disciplines to denote the top-level structures of complex systems. In these disciplines, especially the activity of architecting, in contrast to traditional engineering, puts strong emphasis on an overall design task [RM97]. Architecting denotes the activity, which balances functional requirements, available technologies, interests of involved stakeholders and desired system qualities to create an overall architecture (see section 2.6). Rechtin and Maier [RM97] adapted the definition given by the Webster's Dictionary for architecting in the context of construction to define:

"Architecting is the art and science of designing and building systems".

Architecture, therefore, is a specific level of design. Design means the arrangement of parts or details of something so that a set of qualities is achieved ([Wei88], p9). The architecture provides a development context and builds the basis on which typical engineering tasks take place. This is the meaning of the term *top-down description* of Rechtin's definition (see above).

We distinguish three different types of architectures: system architecture, software architecture and family architecture.

The *system architecture* consists of at least the following elements:

the system structure which is broken down into hardware and software components;

the externally visible attributes of these components, such as interfaces, resource usage, and other quality characteristics;

constraints imposed on the component designs to achieve the desired properties of the system;

system standards that must be met by all components.

Additionally, the *software architecture* should be able to handle complexity

by describing views that support understanding of the SW system from different perspectives,

by providing mechanisms for an incremental SW system which is assembled from a kernel and successive additions,

by means of a modular design which lets the most likely changes be implemented locally within one or a few components.

Moreover, a *product family architecture* should be *future-proof*, which can be characterised as

covering a whole family of current and future products, enabling reuse;

providing for extensions with the most likely features a customer may want.

---

## 2.3 What is a method?

The term method is very generally defined by the Webster's Dictionary [Web13] as an

"orderly procedure or process".

However, we want to base our use of the notion of *method* on the definition given by the Esprit project ATMOSPHERE [Kro93]

"... a method is defined as consisting of: An underlying model, a language, defined steps and ordering of these steps, guidance for applying the method".

The ATMOSPHERE project considered this definition particularly useful in the context of systems engineering of software-intensive systems. The systems for which we have developed the BBM belong to this domain. The core BBM addresses large software-intensive systems, while the specialisations address a specific class of software. Methods, which aim at a very wide application area, can only be very general. The advantage of a restricted application area is that guidelines and examples can be more specific. In the context of the usability of general methods in engineering, Michael Jackson says:

"There is no place for constructive or universal methods. The methods of value are micro-methods, closely tailored to the tasks of developing particular well-understood parts of particular well-understood products" [Jac98].

In this thesis not all elements of a method as defined above will be given equal attention. We will pay most attention to explaining the underlying model, the so-called product part [Kro93]. The underlying model is introduced in chapter 3 and compared to underlying models of other methods in chapter 9. We do not intend to describe a language for objects, aspects and processes. Languages of other methods may be used, e.g. UML. The so-called process part [Kro93] of the BBM is given in two chapters. In chapter 3 we introduce the main design tasks of the core BBM and give an overview of the design steps of the design tasks. In chapter 11 we address issues of development process and organisation. Guidelines and examples are given throughout the text.

---

## 2.4 Models and Meta-Models

Another important point for positioning the BBM is its relation to real systems and models of these systems. The BBM is a design method for architecture. It explicitly contains a notion of what architecture consists of. This we call the architectural meta-model of the BBM. Architecture itself, however, is a model of/for a real system. To make this relation more explicit, we will compare it with the modelling of data (table 1).

Abstraction Hierarchy	Data modelling	Data modelling Example	Architectural modelling	Architectural modelling Example
Model of the Model	Meta Data Model (Data Model)	Relational Model	Meta Architectural Model	Architectural Meta Model of the BBM
Model of the Real World	Data Model (Scheme)	Tables of inhabitants register	Architectural Model	SW Architecture for Product Family
Real World	Data	Inhabitants of Eindhoven	System / Product	Products of a Family

*Table 1: Analogy of Models*

A meta-model has been established in data modelling, which has a long modelling tradition [Dit97]. Beginning with the concrete data instances, a data model describes

the facets of reality, which are relevant for the modelling purpose, e.g. a collection of tables or objects. The data meta-model describes the allowed structures for data models, e.g. the relational model or an object-oriented model. The terms in parentheses give alternative terms, which are sometimes used. The second column gives a concrete example.

Similar to the modelling of data, the SW architecture of a product family is a meta-structure for a concrete product family (table 1). A product family is dependent on its architectural model which describes a set of possible product family implementations. The architecture in itself is again constrained by the possibilities of the architectural meta-model of the BBM. The BBM has sufficient intrinsic complexity to allow the design of sufficiently flexible architectures for families of software-intensive systems (see section 9.1). Other methods and approaches, which we will discuss in chapter 9, are not sufficiently rich for creating architectures that show the required qualities.

---

## 2.5 About Some Terms

The terms we use to describe the context of software-intensive systems come from different areas such as telecommunication systems, systems engineering and software engineering. We do not intend to invent any new terms because the terms are only important for setting the context for the BBM. We will mention some of the terms' different meanings. If no further qualifications are given, it should be clear from the context which meaning is intended.

In telecommunication systems, an important distinction is made between system control and system management. Management comprises functions of a system, which are operator-oriented. System control comprises functions, which are automatic-action-oriented, for instance for recovery and graceful degradation. The background of this distinction is that high availability systems cannot rely on operators only. However, it is not common to make this distinction explicit in describing the functions configuration management (CM) and fault management (FM). The term fault control is not customary in telecommunications, while the term configuration control is used synonymously with configuration management. Therefore configuration management and fault management comprise functions of both system management and system control.

Another problem comes from software engineering, in which the term configuration management refers to functions in the software development environment. In this respect our use of configuration management could be called on-line configuration management.

Yet another area of potential confusion is the use of the terms physical and logical. In telecommunications, physical equipment or physical resources are the touchable or real or hardware things. Logical equipment usually refers to some data representation of physical equipment, while, in addition, a logical resource may also be a resource from the domain of computing like a file or a buffer.

---

## 2.6 A Rational Architecting Model

In this section we sketch a model of architecting to explain the context we assume for the use of the BBM. Several tasks are described in terms of their results. The inputs and outputs of the BBM are taken from this model.

The model for architecting (figure 4) relies on a model described in [AMO\*00]. It is a rational architecting model in the sense of Parnas and Clements [PC86], that is, the model describes not an actual process but a rational process reordered according to the logic of the cause and effect. Earlier tasks are the causes for later tasks. The model starts with the question about what the essence of a customer use of a product is. It then concentrates on the application of the product itself. The third task deals with the design from a commercial point of view. It specifies the functions and features of a product and gives its commercial rationale. The fourth task is the architectural design from a product construction point of view. This is where the BBM fits in. The last task is about technology and determines technologies used for implementation.



*Figure 4: Model for Architecting*

The five tasks can also be characterised by the questions they answer. The first two tasks, customer business modelling and application domain modelling, are about the customer and the problem domain. The first task deals with the customer's objectives, that is his business goal. The second task addresses the use of the system by the customer, that is the application. Together they answer the question why the product is needed. The last three, commercial design, architectural design and technology, are concerned with the product or the solution. The

commercial design answers the question what the product must be capable of. The last two answer the question how the product is/must be built.

Note that in traditional object-oriented development there are only two tasks, application domain modelling or analysis and system design. Design decisions are motivated directly by requirements from the application domain.

Moving to a task on the left gives the reasons why something is done, while moving to a task to the right gives the means how something is done.

We use such a wide model for the process of architecting to provide anchors for reasoning in the product development context. That does not mean that architects have a prime responsibility for all of these tasks. In fact they have prime responsibility only for architectural design. But they will be involved in the other tasks as well and have to take care that technical decisions and commercial decisions are aligned over the whole process of architecting.

We will use in the following the term *modelling* to denote that the character of a task is more concerned with faithful rendering and the term *design* to denote that the character of a task is more concerned with purposeful arrangement. But that is not black and white and all tasks have facets of both characters.

### **2.6.1 Customer Business Modelling**

Customer business modelling is about understanding what the customer really wants. Leading questions for this task are: What is essential to the business of the customer? How does he make his money?

This task results in the customer value drivers, the customer business models and the model of the markets the customer is in, that is its competitors and complementors [Mul02]. This is important to understand because it determines the value a product has for the customer. The customer value drivers are an important source of reasoning about priorities of product development. A product has to be built such that it supports the achievement of these values.

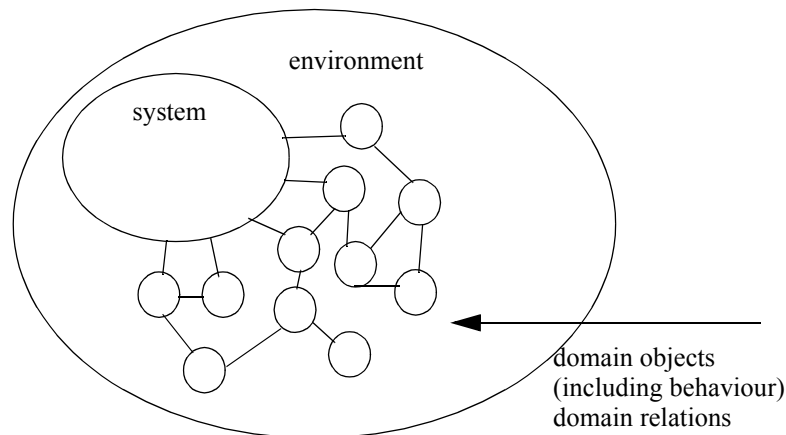
For example, a telecommunication manufacturer needs to understand consumers, which are the customers of the telecom operators. They want an easy-to-use and reliable service. A telecom operator makes money with the duration of connections implying that he wants to have a high rate of successful calls. Call facilities such as call forwarding, voice box and automatic ring back are a means for more successful calls.

In a similar way, a manufacturer of medical imaging equipment, which sells to radiology departments of hospitals, needs to understand that the quality of the images

taken from the various parts of the human body is of prime importance to a radiologist in being able to make a good diagnosis. A second value driver is the efficiency in which a radiology department can handle patients. The average time, which is needed per patient examination, is an important indicator in this respect. A third value driver is the safety of patients and operating personnel. Radiation has to be reduced to the necessary minimum and equipment handling has to be such that patients and operating personnel are safe from injuries.

### 2.6.2 Application Domain Modelling

Application domain modelling deals with capturing the application in a model, which can be used as a basis for product design. Important subtasks are the identification and scoping of the application domain itself, the identification of application stakeholders and the modelling of the application. Domain modelling models the application context of a system as shown in figure 5.



*Figure 5: Application Domain Modelling*

The stakeholders of the application are actors in the environment of the system and are modelled as domain objects. It is important to analyse what they do and how they view the system. Note that this may lead to conflicting views about the system. Conflicts are not resolved at this task, it is just important that a faithful rendering of the various views take place. The resolution of conflicting views may be done at commercial or architectural design where different strategies may be taken. Typical examples are the design of different products or different features which emphasise the one view or the other or take various compromises.

For example, when we do customer business domain modelling we look at the commercial stakeholder of the customer whoever that may be; the user itself, a customer

or a financial department of the customer organisation or other. In a similar way, other stakeholders such as product managers, marketing people, developers, testers and maintainers are part of the development organisation.

### **Domain Object Model**

The domain will usually be modelled in an domain object model, as shown in figure 5. It captures the entities of the domain as objects, determines their behaviour and their relations to other objects. The externally visible behaviour of the systems to be built is also part of the application domain model. The language used is that of the customer (or user) and the product managers. The intention of the domain object model is to describe

- what the system embraces,
- the system's environment, e.g. with which interfaces it has to comply, and
- other constraints for the system.

The domain is analysed to identify objects using heuristics.

Objects include physical entities, such as the controlled equipment and equipment interfacing with the system, as well as logical entities such as images, image sequences, telephone calls and speech announcements. In general, an approach for object identification as described in OMT [RBP\*91] can be applied. Objects are identified by extracting nouns from domain descriptions. These tentative objects are then reduced by eliminating spurious objects. [RBP\*91] describes guidelines for selecting spurious classes.

Domain object modelling has to be performed for the functionality of the entire product family. The reason why we use the term *domain object* instead of *object* is to emphasise this need for a modelling technique that extends specific system requirements to those requirements of a family or, even more, to the application domain as it exists independently of the specific system family.

Rumbaugh [Rum94] makes the distinction between domain and application explicit. He uses the concepts of a domain object and an application object. Application objects are computer aspects of the application that are nevertheless visible to the users. We come back to this difference in section 4.1.1.

The necessity to base object modelling on the functionality of the complete product family is underscored by the observation that systems often have an unstable object structure during their initial versions. This is because requirements are often oriented at a specific customer only. The stopping criterion for refinement

of the object structure is one of stability or robustness with respect to foreseeable evolution [JCJ\*92].

The domain object model is essentially a model for communication between stakeholders. Additionally, it will be taken as first domain-oriented decomposition of the system [Wie98a]. The BBM uses it as input for its object design.

### **Behavioural Modelling**

Behavioural modelling is an essential part of the application domain modelling. Behavioural modelling consists of several steps. First, all relevant domain processes in which the intended systems will participate are described. Then, the interactions of the stakeholders in the domain are described by use cases. Use cases are analysed for their elementary activities. For each activity objects and their interactions are described. Activities are, then, described as object interaction flows. State modelling in connection with attributes and relationships will be used to capture behaviour of the actors of the domain, their actions and automated actions in the system context and by the system itself. The model of the domain behaviour, consisting of domain objects, their interactions and their internal states, will be used as an input for concurrency design of the BBM.

### **Modelling Technique**

To model the application domain techniques such as object modelling, dynamic modelling and functional modelling of OMT [RBP\*91] may be used. A more recent technique is the use of the Unified Modelling Language UML [Fow97].

Another technique is role modelling as described for OOram [Ree96]. Instead of modelling objects in classes, object interaction is captured in role models. An object plays a certain role in such object interactions. Real world phenomena are described by a number of collaborating roles. Objects are, then, composed from their roles. This is one of the major advantages over traditional object modelling which is class-oriented.

Feature modelling [CE00] is yet another technique proposed for domain modelling. Based on the critique that the notion of an object implies state and behaviour, some have suggested to use more basic conceptual modelling based on perception psychology [CE00]. A further critique is that variability modelling in traditional OO cannot be done free of design. Single inheritance, multiple inheritance, parametrised inheritance, static parametrisation, dynamic parametrisation, as shown in [CE00], are used in OO modelling to express variability. The decision which of these concepts to use is already a design step and this should not be done during domain modelling. An application domain, therefore, is described using concepts as described in perception psychology. A concept comprises features and dimensions, that is, qualitative and

quantitative attributes. In [CE00], Czarnecki and Eisenecker propose hierarchical feature decomposition using several types of features such as mandatory features, alternative features, optional features and or-features. The aim is to create models of the configurability facets of concepts. This is an important modelling to use as a basis for the design of families of systems.

Feature modelling, as originally proposed in [Kan90], is widely used. For example, Griss et al. [GFA98] extended RSEB with feature modelling.

Let us take a look at a domain with maturity of domain artifacts, say cars. A lot of domain-specific objects are used to describe and compare cars. Moreover new cars are described in terms of the attributes of domain objects, e.g. the number of cylinders of the engine and its h.p., the type of gearbox, the interior design, maximum speed, and fuel economy. In a new domain, however, a description of a product uses functions of that product much more often than attributes of domain objects. The identification of objects is still a matter of system design.

As a domain matures and companies want to cover an entire application domain with their products, the focus of the domain modelling changes. While initially, the domain model described the functionality of a single product, as the domain matures, the domain model shifts towards a description of the domain itself.

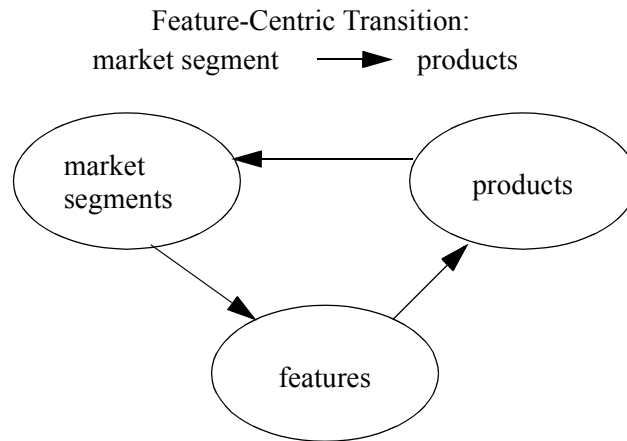
Examples of domain objects from the telecommunication switching domain include subscribers, their access types such as analog or ISDN and their account, calls and call facilities such as call forwarding, call recording and automatic ring back.

We expect that a domain object model is constructed either directly or indirectly via other models like role models and feature models. The domain object model is a first decomposition of the system's functionality. It is the basis for the design steps of the BBM. Several analyses and refactorings are applied throughout the different design steps of the BBM.

### **2.6.3 Commercial Product Design**

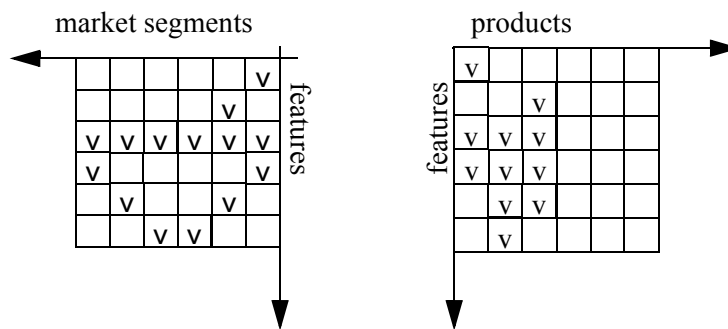
Commercial product design is about the design of products to address certain market segments. Products have to be commercially feasible before their development is started. We assume that an analysis of the market is made. Potential product features have to be assessed for their commercial value. From such com-

mercially viable features products can be designed. This is shown in figure 6.



*Figure 6: Feature-Centric Transition*

Features are used to make the transition from market segments to products. Products are realised by a cluster of features. Feature matrices are a means that are often used to describe the mappings (figure 7). A product may be used for a certain market segment if it has all required features of that market segment (figure 6)



*Figure 7: Feature Matrices*

Sometimes the notions of functions, features and options are used. Functions are the main blocks of functionality. Features extend the functions towards certain applications. Options are mainly technology-oriented variants for products. However, for simplicity we will rely only on the notion of a feature, which may also be a function or an option, since we do not further exploit the differences.

As shown in figure 8 a product is configured from features. A base product consists of a number of mandatory features. Possible extensions to the base product are determined by moving through the *railway diagram* downward. Each feature which is passed is selected.

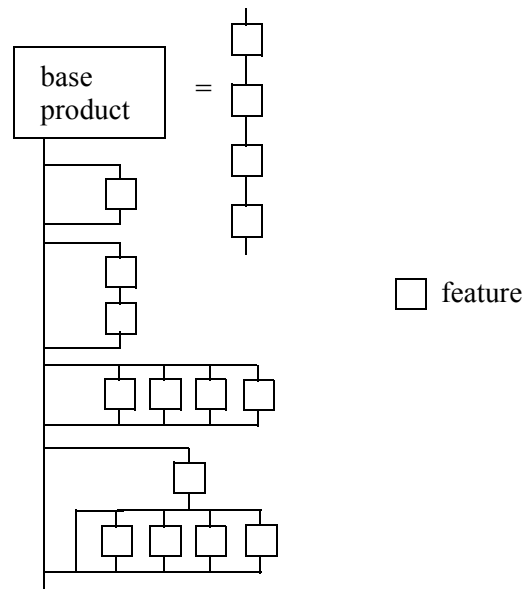


Figure 8: Base Products and Features

In initial markets often a basic product is sufficient. More mature market require in addition to basic features also differentiating features. Product families, for instance, are defined in mature markets to address a variety of customers and applications, both in parallel and as a series of products in time. Mature markets have the advantage that it is easier to predict and plan new products.

Commercial design needs input from architectural design and technology in form of technical feasibility, development effort and cost. This illustrates that the architecting tasks (figure 4) are only a rational process and do not describe a process like the waterfall model.

Requirements and functional specification are two views upon a system's functionality. Requirements describe the system from the perspective of a customer or user, while functional specification describe the system from the perspective of the developing organisation. Both rely on concepts defined during application domain modelling. Additionally to the wishes of the customers, requirements from the development organisations such as its way of doing business, its capabilities and ambitions have to be taken into account.

The most basic function of the application domain model is its use as a common terminology base. Throughout the specifications the terminology of the application domain model will be used. Furthermore, the application domain model will specify relations and behaviour of domain objects. Feature specifications can reference the application domain model and this way be more concise. The level of detail necessary for specifying features depends on the details given in the application domain model.

Also, possible conflicts of stakeholders have to be resolved. There may be conflicts between application stakeholders or between an application stakeholder and a development organisation stakeholder. Even more complex stakeholder scenarios are possible where other organisations are involved, such as buying organisations or maintenance organisations.

Related to stakeholders, quality profiles have to be defined for products.

Not only specification of functionality is important but also the reliability, performance and security of products. Quality profiles, in general, may specify run-time qualities, but also build-time qualities like extensibility, evolvability or maintainability. Furthermore, budgets of development time and resources have to be decided on. Architects will have a major input in the tasks of commercial design.

The output of the commercial design is a mapping from features to products. This mapping is an important input for the product family design of the BBM.

### **2.6.4 Architectural Design**

Architectural design is about the structuring of the actual implementation. It is the area of applicability of the BBM. In the following we will describe the input and output of the BBM. But before we go into detail of the BBM we address the split between hardware and software.

#### **Hardware - Software Split**

Up to now we have not really made a difference in the description of the architecting process between hardware and software functionality.

If we only think about developing a software product, the hardware issue is one of selecting the necessary deployment platforms. However, in the more general case we have to make design decisions about which parts are implemented in hardware and which in software. This is the issue of what is often called system architecture. Several approaches exist here. We may rely on standard hardware modules or we partially design our own HW (HW/SW co-design or sequentially ordered HW before SW). A factoring of HW and SW functionality

needs to take place. In the BBM hardware-implemented functionality is factored out. This factoring is part of the object and aspect design tasks. The design of the hardware is outside the scope of the BBM.

### **BBM Inputs**

One of the major inputs of the BBM is the application domain model. It provides a first domain-oriented decomposition of the products (see figure 9). Other inputs are the commercial design outputs in terms of features, product specifications and quality profiles. The last input are technology choices which are determined in the technology task (see section 2.6.5).

Note that stakeholders are not mentioned directly as inputs. However, they are implicitly present through domain objects and behaviour reflecting their needs, and through quality profiles.

Taking the application domain model as first input means that very important modelling has to take place prior to the use of the BBM. It also means that traditional OO development methods can be used together with the BBM (see section 9.2). They provide an object and behaviour model which is used as a problem decomposition used as input for design in the BBM. The BBM takes that input and refactors it to create construction elements and architectural models.

The question might be asked if it is useful to rely on the output of another method and if a description as a single method would not be much clearer.

An important point is that architectural design does not only directly depend on domain modelling but also on features from commercial design, which is often left out in traditional development models.

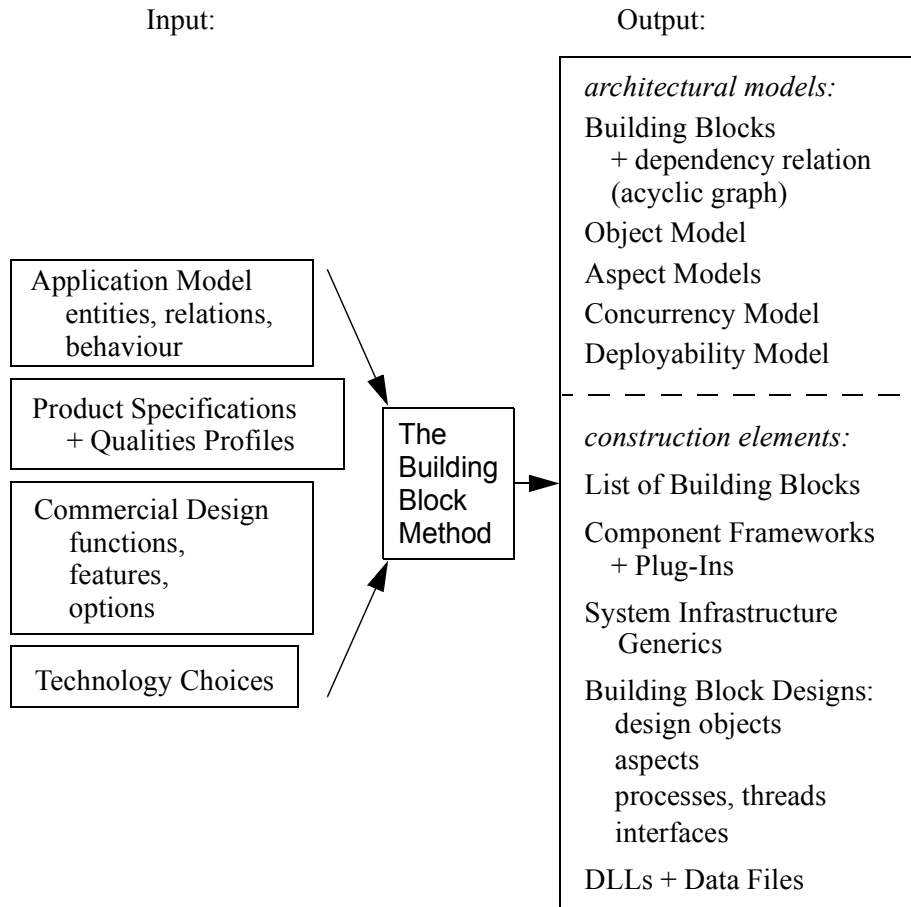
Furthermore, an analysis of the complete system functionality reveals that the domain functionality is only a small part of it. An important input for this additional functionality is derived from required quality profiles. This will get specific attention in the BBM.

An all-embracing method would be very complex and inflexible.

### **Architectural Models and Construction Elements**

The artifacts of the BBM, that is its output, (see figure 9) comprises architectural models and construction elements.

As described by Lakos [Lak96] the evolvability and extensibility of large systems depend to a high degree on the locality of the changes necessary to implement new functionality. That means the physical structure of the source code is important. In



*Figure 9: BBM: Input - Output Specification*

the design of large systems we made the same observation. This is the reason why the BBM emphasises the design of the construction elements.

The BBM leads to several architectural models. A structural view (see [P1471]) is given by the BBs and their dependency relation. An object model showing the relations between design objects. Aspects defined by the BBM for the development of a product family have their own design model. The deployability design maps the functionality to the (possibly distributed) hardware. The concurrency design gives the mapping of functionality to processes and threads within hardware units.

### 2.6.5 Technology

The technology task concerns the technical choices which are not part of the architectural design. This may be different for different products. The technology choices and technology roadmaps are an important input for the architectural design because they provide the basic means which are used for building systems.

As an example of the variation of the content of the technology task take the hardware, which may be selected as part of the technology task or developed as part of the architectural design task. Similar for other technology choices like operating systems or operating system versions. If a product is to run on Windows and Unix platforms, the specific operating system and its version is part of the technology task. If only specific real-time kernels are used they are part of the architectural design. Other technology choices concern monitors, networks, computing infrastructure, libraries and component models.

### 2.6.6 Feedback, Navigation and Learning

Having described the architecting process in five major tasks does not mean that they are executed in this order. As occasionally mentioned throughout the description of the tasks, feedback between the tasks is essential.

As Muller states in [Mul02] the actual way of working may even start with the last task, for example, in the case of technology exploration. Starting to experiment with new technology may give ideas of what type of applications are possible to build with it. Architectural consequences are explored and commercial scenarios developed. However, the rational design process remains from left to right (see figure 4).

Navigation through different architecting tasks is one of the essential means for achieving consistency within the architecting process. Major design decisions should be traceable backwards to the customer value drivers.

For example, the questions what does the customer really need? and how does a specific technical function contribute to his value creation? are important to ask at points of design decisions.

Similar is the tracing of diversity. The breadth of the architecting process allows to identify the anchors of diversity.

They include different market segments, different customers, different stakeholders or technology choices. The products should be made such that they combine maximal appeal to the customer with the lowest possible implementation diversity. For example, data parameters are preferable over code diversity.

---

## 2.7 Levels of Consolidation

An essential concept of the BBM is the concept of a component framework. Component frameworks are a way to encode domain experience in a reusable asset. Product families are built by developing new plug-ins to existing component frameworks or by refactoring existing BBs into component frameworks and plug-ins.

As we will explain below, the terms generic and specific BBs will be used for the concept of a component framework and plug-ins.

An orthogonal view is presented by Roberts and Johnson [RJ96]. They describe a pattern language for stepwise evolving reusable assets from a set of common examples to the design of a domain-specific language. Several successive levels of consolidation are shown in [RJ96]. The first consolidation step leads to white-box frameworks using inheritance; the second to so-called component libraries; the third to black-box frameworks; the fourth to the use of a visual builder and the last to the development of a domain-specific language.

We had similar experiences as those described in [RJ96]. In the development of the tss product family (see appendix A) the data definition data base was developed to generate parts of BBs which were related to system infrastructure generics. Further generation tools were discussed but not developed. In the development of the tss product family the development of new component frameworks was more urgent than easing the development of new applications by providing a domain-specific language for similar plug-ins to existing component frameworks. This, however, was a pragmatic argument for the tss development. In other areas domain-specific languages were developed successfully [DK98].

Component frameworks are chosen deliberately as the level of consolidation for the BBM. First, the BBM is a method for determining components of product families. This means that not every product development will use the BBM. In the development of initial products one will focus on achieving a product which will succeed in its market. Components are a good engineering means for independent development. However, component and their interfaces will usually not be stable initially. Refactoring is not an accident but a necessary second phase after the first principal development.

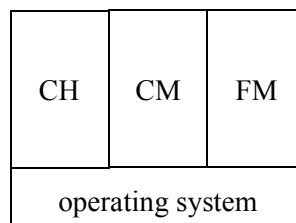
Product families may be conceived early but experience shows that components and component frameworks will need two or three redesigns to become stable [RE99].

Systems which already have stable component frameworks may be further consolidated by developing domain-specific languages. However it is important to go through the consolidation path level by level. Otherwise focus is easily shifted from making successful products to developing fancy technology. [DK98] gives advantages and disadvantages of developing domain-specific languages.

---

## 2.8 Historical Background of the BB Method

Twenty five years ago, the software of telecommunication infrastructure systems was quite commonly structured in vertical blocks. Based on the operating system, the application software consisted of the three vertical function blocks: configuration management (CM), fault management (FM) and call handling (CH) (figure 10).

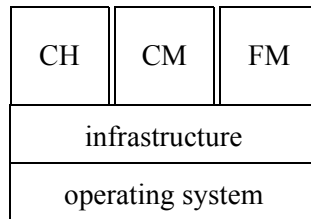


*Figure 10: Dependent Functional Block Structure*

This resulted in a strong interdependence of the functional blocks. System tests were only possible using the entire software system. It was very hard to extend the software with new functionality because of these relations. For instance, CH handled calls, CM dealt with the configuration of call-handling functions, and FM handled call failures. New call functionality had to be distributed over these three blocks.

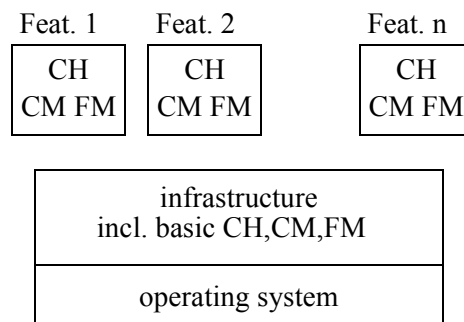
A next step was to factor out common functionality from the application and build an infrastructure layer. This also contained the system's database. Further-

more, communication between the functional blocks was restricted to proceed via services of the common infrastructure (figure 11).



*Figure 11: Independent Functional Block Structure*

A further step was to support the introduction of new features directly through modular structuring. Instead of distributing new functionality over the existing software structure, a feature is encapsulated in a module and can be introduced into and taken out of a system. The required infrastructure now comprises also basic services of CH, CM and FM (figure 12).



*Figure 12: Feature-Oriented Application Structure*

The BBM further generalises horizontal layering as a means of structuring the entire system (section 7.4). A system is developed layer after layer. CM and FM are handled as aspects (chapter 5).

Exchange of single modules on the target system was first used for fast error correction only. The BBM lifted this principle to the system level to extend/reduce the system functionality by means of product features (section 8.3). The design of the tss product family is described in appendix A.

## 2.9 Overview of the Concepts

We end this chapter on the context of the BBM with an overview of the concepts used in the BBM. The concepts, the chapters in which they are described and the main relations between them are shown in figure 13. Higher-order architectural concepts have been derived from general architecture requirements. They are expressed in terms of basic architectural concepts.

Note that for the concept *component framework* [FS97] we use the term *generic Building Block* (see section 7.5.1) for historical reasons.

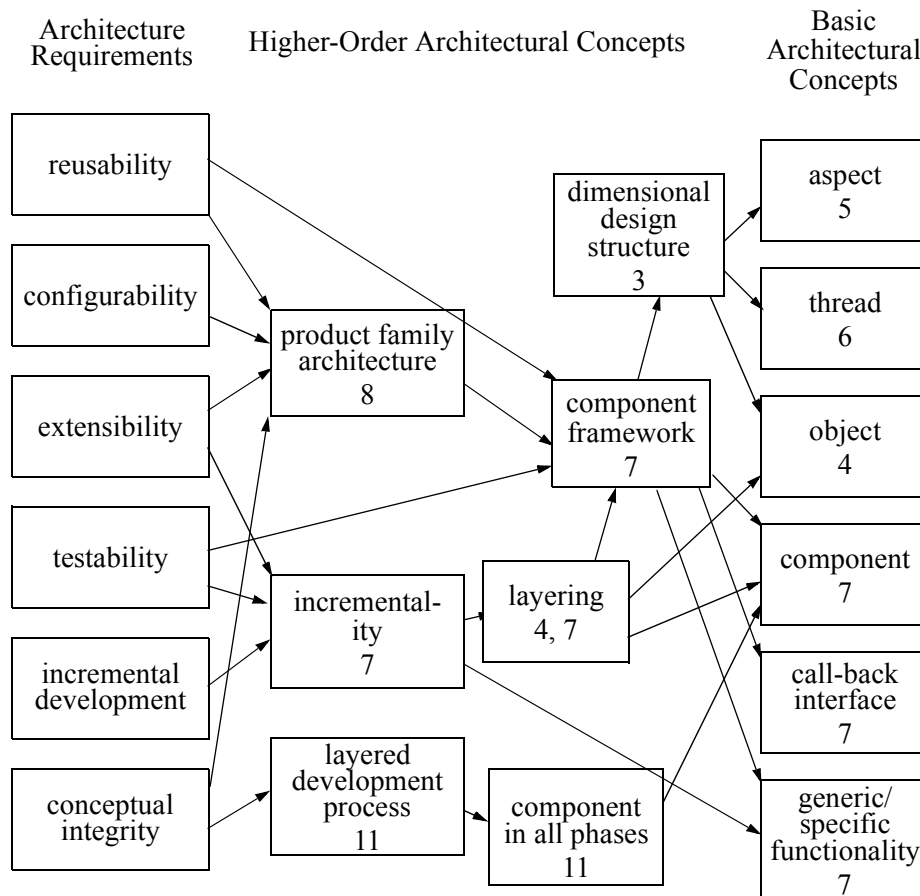


Figure 13: Concepts of the BBM and their Main Relations

The next chapter will give an overview of the BBM.



---

## 3 The Core Method Overview

The BBM is a component-based architectural design method for large software-intensive product families. Its emphasis on large software-intensive systems implies a focus on the actual construction elements of software systems. The feasibility of evolution and extension of large systems is largely determined by its deployment structures. Development effort can only be limited if changes are limited to a small number of deployment units.

Lakos [Lak96] names several problems in the development of large-scale software:

- poor encapsulation which hinders reuse and hampers testability;
- circular dependencies which leads to tight physical coupling making effective modular testing impossible;
- excessive link-time dependencies artificially increasing the deployed code;
- excessive compile-time dependencies increasing the time to not only recompile the complete system but also the time for each translation unit, for instance when a globally visible include file has to be updated; and
- global name space for variables leading to surprising name clashes.

The BBM uses software components and restricts the allowed dependencies. In general, different kinds of modularities are employed to improve the overall modularity of the software.

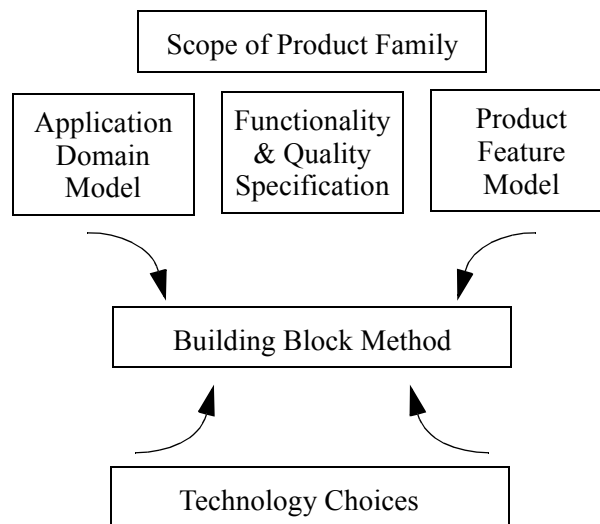
The presentation of the BBM is split up into a core method and method specialisations. An overview of the core method is described in this chapter followed by a number of chapters, which present the method in more detail. A method specialisation is described in chapter 10.

The BBM consists of five main design tasks: object design, aspect design, concurrency design, composability design and deployability design. Like the description of the architecting process (section 2.6), the BBM is described as a rational design process. We start the description of the BBM by looking at its prerequisites.

---

### 3.1 Prerequisites of the BBM

The BBM as an architectural design method for large software-intensive product families requires that certain tasks of an overall architecting process (see section 2.6) are (partially) completed. We only require partial completion because the



*Figure 14: Prerequisites for the BBM*

BBM is not part of a waterfall-like development process model. The whole architecting process described in section 2.6 is a rational process only and does not prescribe a specific execution order.

As shown in figure 14, the scope of the product family should be determined. An application model in form of domain entities, domain functions and/or domain procedures and their most important relationships should exist. The functionality of the products in the family and the required system qualities should be specified. A commercial design should have identified commercially relevant features and their dependencies. Relevant implementation technologies should be decided on.

For all these steps, relevant for an architecting process, different methods exist and can be used in combination with the BBM.

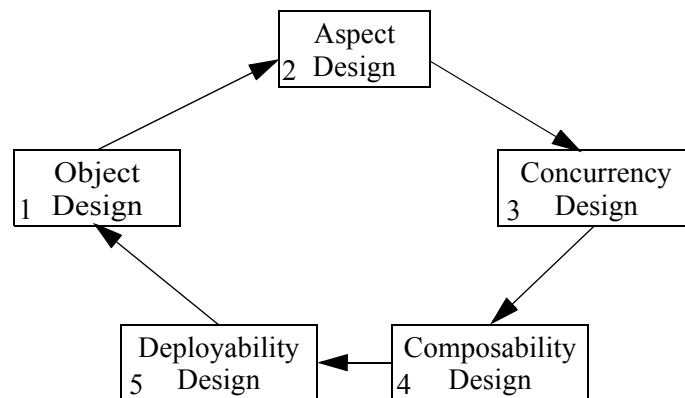
These prerequisites are used as input for the design tasks of the BBM (see figure 22). Object design, aspect design, concurrency design, composability design

and deployability design take these inputs as starting points for their modelling. We will walk through the tasks and describe what they are about.

---

## 3.2 Main Design Tasks of the BBM

The five main design tasks: object design, aspect design, concurrency design, composability design and deployability design are presented in a rational order. Objects are identified. Functionality which crosscuts objects is identified as aspect functionality. The functionality of objects and aspects is mapped to threads. Objects, aspects and threads are packed into BBs. BBs are grouped into libraries and executables.



*Figure 15: Main Design Tasks*

An initial design activity will execute the tasks in the specified order. However, since the design tasks influence each other, they may be executed in arbitrary order. Each task produces results, which are taken by the other tasks as triggers for making their own designs consistent with that of the other tasks. The process stops when the results of each task are stable.

Now we explain the major concepts of the different main design tasks.

### 3.2.1 Object Design

The concept of an object is very general. We will use objects in the context of the BBM at four different levels. Application domain objects describe entities of the application domain. Hardware domain objects describe elements of the hardware system. They are both part of the first level. On the second level we have domain-induced objects. They are a mirroring of the objects of the first level into the software design space. They are generated from inputs of the BBM. On the third level we have design objects. They are refined and refactored due to the design tasks of the BBM. On the fourth level we have implementation or programming language objects. They are mostly a mirroring of design objects. However, specific implementations may introduce new objects.

A detailed description of object design is given in chapter 4.

### 3.2.2 Aspect Design

The set of aspects is a partitioning of the complete functionality of a system. The application domain functionality is only part of a systems functionality. Other functionality is induced by quality requirements. This additional functionality is often larger than the application functionality. To construct the aspect partitioning, certain types of functionality are identified and factored out. Initially all functionality is said to be part of the operational aspect. From the operational aspect those types of functionality are factored into aspects which crosscuts domain-induced objects (see figure 16). An application domain object such as telephone call or a medical examination needs to be initialised, to be configurable, to deal with erroneous situations and to support operator interaction. These types of functionality are factored out as separate aspects. The remaining functionality will define the operational aspect.

For each of the factored-out aspects we can make designs, which apply throughout the system and give the system design a certain uniformity. Common

implementations are factored out in specific component frameworks, called system infrastructure generics.

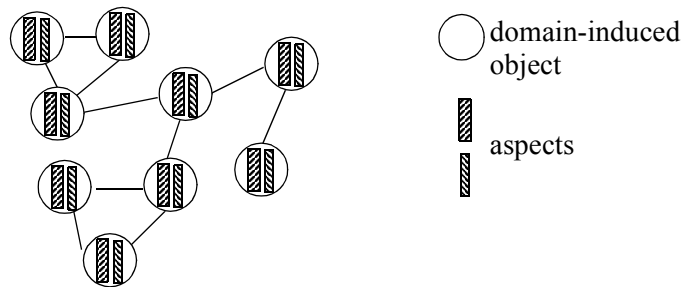


Figure 16: Aspects and Domain-Induced Objects

Aspect design takes the quality specifications as input. They are analysed for necessary additional functionality to achieve the qualities. Additionally, an *architectural concern analysis* based on checklists from prior design experience is performed to check for comprehensiveness of the specified functionality. Both analyses may result in additional aspects and objects.

A detailed description of aspect design is given in chapter 5.

### 3.2.3 Concurrency Design

Concurrency design is about mapping of functionality to processing resources. A concurrency structure is designed for the complete system and will be expressed in aspects and/or objects. Concurrency design starts with the behaviour of the application domain and results in a concurrency model consisting of threads.

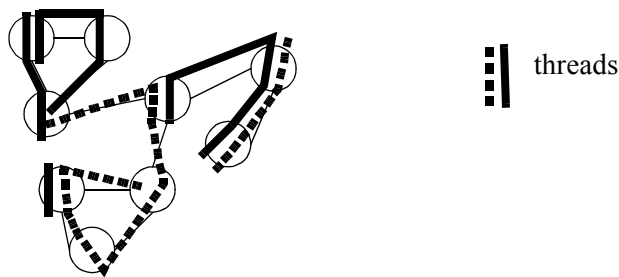


Figure 17: Thread Identification

Objects, aspects and threads are independent and span a design space of three design dimensions (see section 3.3). This means that a thread may involve one or more objects or one or more aspects without design restrictions.

A detailed description of concurrency design is given in chapter 6.

### 3.2.4 Composability Design

Composability design is about defining modularity to support the composition of products in the product family, to obtain manageable development units, to realise a simple feature mapping and to allow for incremental integration and testing.

BBs are design and deployment units, which are identified in the architectural phase [Szy98]. There are two important questions with respect to BBs: What is the content of a BB and what relations exist between BBs? The identification of BBs starts with partitioning the network of objects. A BB is initially a cluster of related domain-induced objects (see figure 18). BBs are refactored to contain

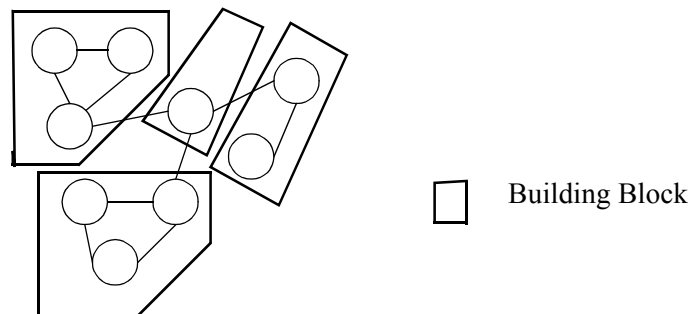


Figure 18: BB and Objects

clusters of design objects. BBs do usually not contain entire processes or aspects. The main criterion is configurability and situations are possible where an aspect or a process is itself a unit of configuration. The set of BBs covers the entire functionality in a non-overlapping way.

The BBs are technically the dominant decomposition [TOH99] of a BBM-based system (figure 19). The possibility to assign functionality along one of the three axes (object, aspect and thread) or a combination thereof provides flexibility to choose the decomposition which best supports the evolution of the product family. The *tyranny of the dominant decomposition* [TOH99] is thus avoided.

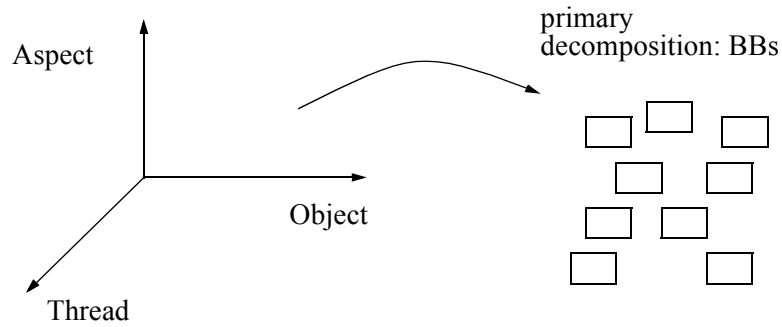


Figure 19: Mapping of Objects, Aspects and Threads to BBs

**Component Frameworks**

An important point in the design of a product family is the separation of generic and specific functionality. Generic functionality is implemented once and is used by other BBs. Component frameworks, called generic BBs in the BBM, and plug-ins, called specific BBs, are designed to encapsulate generic and specific functionality.

**Incremental Layering**

Relations between BBs are derived from the relations between the encapsulated clusters of objects, from relations created by splitting up aspects and threads and from inter-aspect and inter-thread relations. The BBM restricts these relations by requiring that the dependency graph forms a partial ordering of all the BBs. Thus, additional design may be necessary to conform to this restriction In a

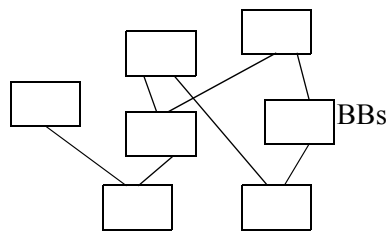


Figure 20: Dependency Relation Between BBs

graphical representation we mostly use lines without arrowheads. BBs being located higher in a figure depend on BBs located lower (see figure 20).

BBs are designed such that they can be integrated and tested layer by layer. Such layering of BBs is called *incremental layering*. Layering is used on two levels. Coarse layers are derived from the layers introduced during object design. These layers are refined such that the coarse layers are also internally layered. Specific kinds of generic BBs are used to allow exchange of BBs in lower coarse layers. Incremental layering and *plugability of BBs in lower layers* are key concepts of the BBM.

### **Architectural Skeleton**

Taken together the generic BBs form an architectural skeleton on different layers. The *architectural skeleton* is the basis for the product family architecture.

### **Features and Product Family Architecture**

Commercial product features are another input of composability design. The dependency structure of features should be reflected in the BB dependency structure. As commercial features describe a product commercially the product should be buildable from BBs which implement these features. This is called feature orientation of the product family architecture. The design of an architectural skeleton is a means to create a *feature-oriented product family architecture*.

A detailed description of composability design is given in chapter 7 and chapter 8.

### **3.2.5 Deployability Design**

Deployability design is about possible deployment scenarios of the products. The input for deployment scenarios may come from requirements for geographic distribution or from technology assessment requiring a certain HW partitioning. Geographic distribution, if required, will often come directly from the customer. This input may lead to a refactoring of objects, threads and BBs.

The deployment model defines the allocation and/or allocatability of deployment sets to hardware instances. Deployment sets consist of BBs or clusters of BBs.

A detailed description of deployability design is given in chapter 6.

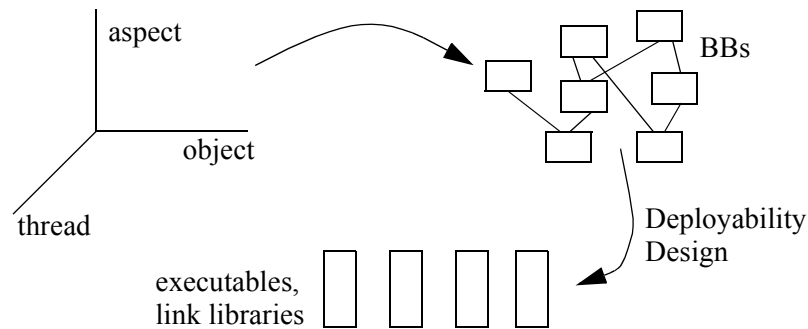


Figure 21: Identification of Deployment Sets

### 3.2.6 Implementation of BBs

BBs are also units of implementation. The functionality of BBs is defined during object and aspect design. However, object design, aspect design and thread design are about design and not directly about implementation.

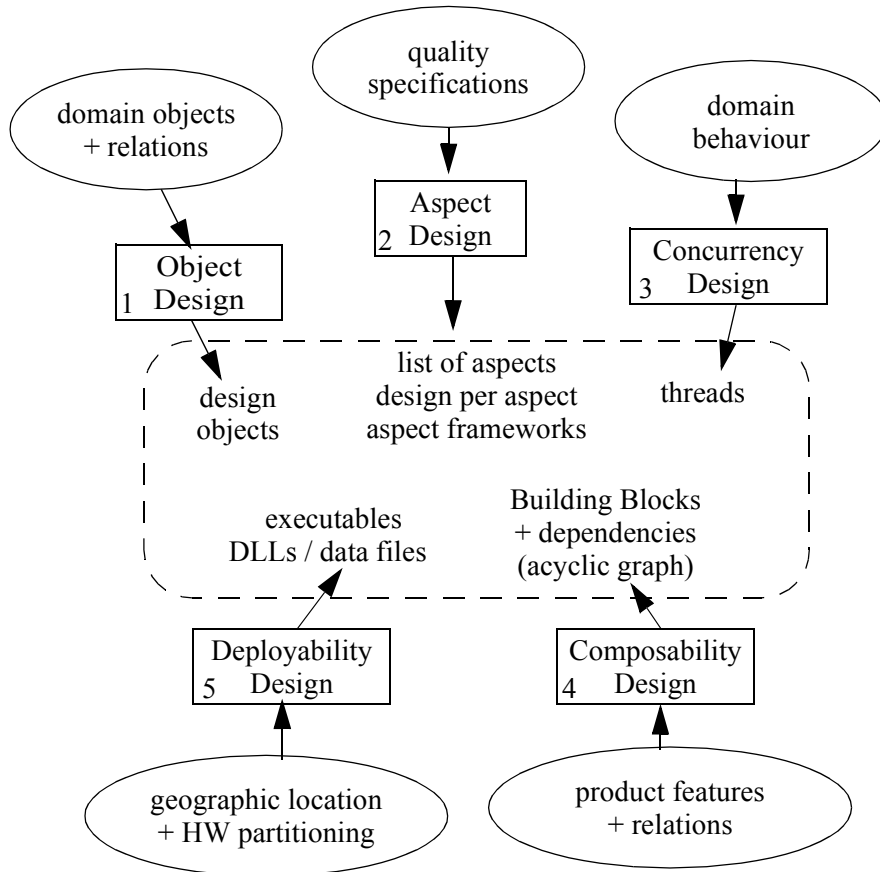
Implementing a BB in an OO language means that everything is implemented in objects. One may use design objects directly during implementation as implementation objects. This means that aspect functionality is implemented by aspect-specific object (or class) methods. However it is also possible to introduce a fourth level of implementation objects which implement the functionality of an aspect. A design object, then, is represented by a set of implementation objects.

There is a similar relation between aspects and threads at the level of implementation. Both, aspects and threads, are not visible explicitly by programming language constructs. Aspect functionality is implemented by methods or objects as explained above. Our notion of thread is sometimes referred to as reach of a thread and consists of all methods and objects, which execute under its control.

In an OO language, an interface will either be implemented as an abstract class or via the interface construct.

### 3.2.7 Design Artifacts

The results of the main design tasks are accumulated as design fragments (see figure 22). Each new execution of a design task may update some design frag-



*Figure 22: Input + Output of Design Tasks*

ments or create new ones. Other design tasks are triggered by updated and new fragments to make their own fragments consistent with the updated ones.

The results of the main design tasks are a set of architectural models and a list of construction elements. The architectural models are an object model, the list of aspects and their designs, the concurrency model, the BBs and their dependency relation, and the deployability model (see figure 22). The construction elements are the list of BBs and their designs, executables, DLLs and data files.

In table 2 we give an overview of the steps which have to be executed per main design task. The steps are described and detailed in subsequent chapters: the steps of object design in chapter 4, the steps of aspect design in chapter 5, the

Design Task	Steps
object design	creating an initial object model
	adapting the object model to required functionality
	factoring out HW-implemented functionality
	modelling HW resources in SW
	refactoring domain-induced objects into layers of design objects
	creating design objects for communication, interfacing, registration, containers and aspects
aspect design	initially taking the complete functionality as one aspect
	analysing domain-induced objects for crosscutting functionality
	performing an architectural concern analysis to find additional aspects
	using starter sets of potential aspects to support the aspect identification
	standardising the list of aspects for the product family
	determining the functionality per aspect
	making a global aspect design
	factoring out common implementation parts in system infrastructure generics.
	defining rules and guidelines per aspect.
concurrency design	starting with behaviour of domain objects
	determining independent external sources
	prioritising aspect functionality
	if necessary, encapsulating specific objects in a thread
	refining the logical threads into physical threads
	determining interfacing between threads

*Table 2: Overview of Steps per Design Task*

Design Task	Steps
com- posabil- ity design	clustering objects into BBs
	identifying variation points of functionality which belongs to different features
	factoring out common functionality in separate BBs.
	identifying interfaces of BBs
	designing component frameworks and plug-ins
	identifying system infrastructure generics
	defining layered subsystems of BBs
	designing for incremental integratability and testing
	doing detailed design of the BBs
deploy- ability design	determining fault containment units
	determining possible deployment scenarios
	packaging BBs to deployment sets
	generating data files

*Table 2: Overview of Steps per Design Task*

steps of concurrency design in chapter 6, the steps of composability design and the steps of deployability design in chapter 7.

Besides the main design tasks, the BBM does not exclude other design tasks (see section 3.4 and section 3.5). The results of other design tasks are inputs for one or more of the main design tasks. In general, design models should be built for all relevant concerns to guide the development of the system. Preferably, design models should be quantitative, for instance by using design budgets for critical resources.

Further results are rules and guidelines, which have to be applied throughout the whole system design. Rules and guidelines together provide a set of internal system standards, which are essential for achieving and maintaining conceptual integrity [Bro75]. Rules and guidelines complement the design of individual components.

Many of the rules and guidelines will be related to aspect designs. Coding standards and resource usage are other examples. An example of the tss system is that search operations in lists were not allowed because the performance of the operation varies with the length of the list. A tss system with a high load would need more time. Instead, designs have to be used where elements can be selected from the heads or the tails of a fixed set of parallel lists.

Catalysis gives three classes of system standards: horizontal or infrastructure standards defined by the infrastructure to be used by all application, vertical standards which apply to all systems in an application domain, and connector standards which are to be used for intercomponent communication [DW99].

The execution structure of the BBM design tasks (figure 22) confirms to the blackboard style (see section 9.3.1). Such a general model raises the question when to stop the BBM design tasks.

### **3.2.8 Stopping Criteria for Design Tasks**

An important question is about a stopping criteria for the various design tasks. This is especially important since we do not give a fixed order in which the design tasks have to be executed such that after the last task the design would be finished.

It is important to realise that architectural design is part of an overall development process. The time given to architectural design has to be decided in that context. An important internal criterion for stopping is when structures become stable and implementable.

When one traverses from one design task to the next, changes may be required for the structures designed in the first design task. Several cycles through the tasks are often necessary because technical systems are often at the edge of technical possibility and the applicability of prior designs is limited. Design experience and early feedback are important in dealing with this situation. Short development cycles after which various kind of users can give feedback are favourable.

Another point is the experience reported by several framework designers that framework interfaces need two to three redesigns to become stable ([RE99], [BGK\*99], and also tss design experience). Feedback, again, is essential.

---

### 3.3 Design Dimensions

The idea of structuring of domain-induced objects, aspects and threads in multiple dimensions is introduced to support the freedom of system design. If design concepts, which address different facets of the same item, can be separated so that there are no mutual restrictions, the concepts are orthogonal. We can then talk about design dimensions. Every dimension can hence be designed independently by projecting each item in the design space to one dimension. The BBM identifies three specific design dimensions.

Note that this discussion is on the design level and not on the implementation level.

The first point is to keep object structuring independent from the use of execution units. Domain-induced objects result from object design (see section 3.2.1), which has as input the domain object model. Threads determine the use of processing resources for independent, cooperating and/or sequential actions.

The designer should be free to design threads without consequences for the design of objects. They constitute two orthogonal dimensions, i.e. a method of an object may be driven by one or more threads and a thread may drive methods from different objects.

Modules were separated from processes in [HFC76] and [Cla85] already.

The second point is to construct aspects orthogonal to domain-induced objects. Those global functions to which potentially all objects contribute are handled as *SW aspects*. Each object method in the system is part of one object and part of one aspect.

The BBM combines these two ideas. This leads to three design dimensions since threads and aspects are also independent, that is, an aspect may be driven by different threads and a thread may drive different aspects.

#### A Mathematical Formulation

From a mathematical perspective, the design independence is described by the three design dimensions: object dimension, aspect dimension and thread dimension (figure 23) forming a design space  $D$ . The design space  $D$  is a discrete space. The values of the first dimension are object classes of the set  $O$  of object classes. The values of the second dimension are aspects of the set  $A$  of aspects. The values of the third are thread types of the set  $T$  of thread types. Formally:

$$D = O \times A \times T = \{ \langle o, a, t \rangle \mid o \in O, a \in A, t \in T \}$$

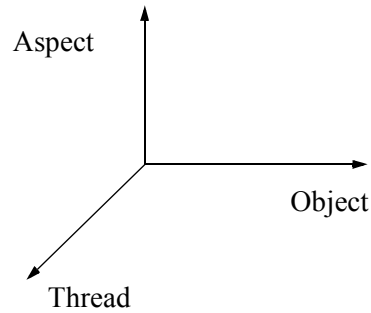


Figure 23: Three Design Dimensions

We shall use object methods as basic terms in our discussion of the design space. An object method is part of exactly one object class and of one aspect but may be driven by several thread types. Furthermore, several object methods may be part of the same object class and the same aspect and the same thread type simultaneously. This means that the points in the design space represent sets of object methods.

Formally, let  $OM$  be the set of object methods. For all elements  $(o \in O, a \in A, t \in T)$  of  $D$  we define the following projections:

$$\mathcal{O}M(o) = \{ f \in OM \mid f \text{ is a method of the object class } o \}$$

and

$$\mathcal{O}M(a) = \{ f \in OM \mid f \text{ belongs to aspect } a \}$$

and

$$\mathcal{O}M(t) = \{ f \in OM \mid f \text{ runs under control of a thread of thread type } t \}$$

The object dimension covers the decomposition of the system into object classes, thus:

$$OM = \bigcup_{o \in O} OM(o) \quad o_i \cap o_j = \emptyset \quad \forall i, j, i \neq j$$

by construction, that is:

$$OM = \sum_{o \in O} OM(o)$$

(The  $\Sigma$  sign stands for a disjoint union of sets.)

The aspect dimension partitions the functionality into specific views, such as recovery, configuration management, fault handling, etc. Thus:

$$OM = \bigcup_{a \in A} OM(a) \quad a_i \cap a_j = \emptyset \quad \forall i, j, i \neq j$$

by construction, that is:

$$OM = \sum_{a \in A} OM(a)$$

The thread dimension describes overlapping subsets of system functionality which are driven by threads. Thus

$$OM = \bigcup_{t \in T} OM(t)$$

Then, we link points  $\langle o, a, t \rangle$  of the design space  $D$  with sets of object methods by defining:

$$\langle o, a, t \rangle = OM(o) \cap OM(a) \cap OM(t)$$

We now see that object methods are basic terms: a point in the design space represents a set of object methods.

Furthermore, we can characterise object methods in terms of the design space. Let  $f \in OM$ :

With

$$OM = \sum_{o \in O} OM(o)$$

we obtain a unique  $o_f \in O$  with:

$$f \in OM(o_f)$$

Analogously, we obtain a unique  $a_f \in A$  with:

$$f \in OM(a_f)$$

However, with

$$OM = \bigcup_{t \in T} OM(t)$$

there may be more than one  $t \in T$  with  $f \in OM(t)$ .

With  $T_f = \{ t \in T | f \in OM(t) \}$  we obtain:

$$f \in \bigcap_{t \in T_f} OM(t)$$

Summarizing, we obtain:

$$f \in OM(o_f) \cap OM(a_f) \cap OM(t) \quad \forall t \in T_f$$

In terms of the design space:

$$f \in \langle o_f, a_f, t \rangle \quad \forall t \in T_f$$

This means that an object method  $f$  is spread over the straight line parallel to the thread axis with object axis value  $O_f$  and aspect axis value  $a_f$ .

The independence of the design dimensions is an important methodological starting point. This independence is complemented by design patterns which describe relations between specific objects, aspects and threads. They describe experiences of good design for a specific design context.

### System Evolution and Design Dimensions

A system, which evolves in the three dimensions simultaneously, is very complex and hard to manage. It would be good if evolution could be restricted to a single dimension. The following tentative considerations argue that such a situation exists for the most common changes for central controller software. However, worst-case changes affecting all dimensions are also possible.

Changes common to, for instance, the central controller software (see section A.3) occur with the introduction of new controlled equipment, and new services for the environment. This will most likely result in new objects and less likely in new aspects. The extensions of a system are obtained by extending objects and/or introducing new objects. Changes are local since only a few components are affected.

The two design dimensions, object and aspect, make use of object-oriented and functional modelling. Aspects are seen as a alternative form of modelling. Aspects are functions which crosscut all or most objects. The list of aspects should be standardised for an entire product family to achieve stable software structures. Adding a new aspect or extending the functionality of an existing aspect affects all the related objects. Because BBs are, usually, a cluster of objects, adding a new aspect induces changes in most or all of the BBs. Locality of change does not exist when an aspect is added.

As explained in chapter 6, concurrency design starts by looking for execution independence of objects. In the case of most systems we assume that a situation is aimed for in which the structure of threads is standardised for an entire family, either

in specific threads or in rules which guide the creation of a concurrency model. Without such a structure, to understand a large evolving system is very difficult.

For those cases where the above considerations concerning stable aspects and threads hold, we say that the system evolves in the object dimension only. In such cases the change effort will be minimal because a new feature will be implemented in a few new and/or updated BBs.

An example (chapter 5) which does not follow the above-mentioned evolution is a product family where one product uses a general login procedure for the whole of the system and another product, functionally equivalent, uses specific login and access capabilities per object. The second product could be derived from the first one by adding an extra *access control* aspect. Implementing this aspect as part of the BBs would lead to a second set of adapted BBs. A better way to implement this is to create a specific plug-in per BB. However, neither are local changes any more.

---

## 3.4 System-Quality-Based Design Tasks

System qualities are important input for architectural design. They can induce specific functionality, cause the selection of specific mechanisms or influence the way in which functionality is implemented.

The main design tasks of the BBM are not grouped according to system qualities but the BBM design tasks have to take system qualities into account. However, the way in which the BBM deals with system design supports these design tasks. In the following we describe the relation of the design for various qualities with the main design tasks of the BBM.

### 3.4.1 Performance Design

Performance design is mainly done in the concurrency design. The use of separate threads for functionality with different priorities supports the design for timeliness. Functionality must be carefully factored such that time-critical paths are minimal.

But also other design tasks may be influenced by performance design. Examples are the selection of data structures or the trade-off between communication via data messages vs. the use of shared data.

In the tss systems, for example, transparent layers (see section 7.4.3.2) are used to avoid calling overhead and an in-memory database increases database update speed.

It must not be forgotten that performance design is also a hardware design issue. The hardware capabilities should at least permit a software solution, which meets the performance requirements. If hard realtime requirements are of paramount importance for the application, it is good design practice to factor out hard realtime functionality and assign it to a separate processor. Such an approach relieves application programmers from programming soft and hard real-time SW for the same processor.

If necessary, specific performance design methods must be used to complement the BBM. For example, rate monotonic analysis may be used to find an initial thread structure to meet deadlines.

### 3.4.2 Reliability Design

The topic of reliability design is best introduced with a quote from Birman [Bir96]: "Through decades of experience, it has become clear that software reliability is a process, not a property. One can talk about design practices that reduce errors, protocols that reconfigure systems to exclude faulty components, testing and quality-assurance methods that lead to increased confidence in the correctness of software, and basic design techniques that tend to limit the impact of failures and prevent them from propagating."

In such a setting reliability design with the BBM can be done via a number of different design concepts.

Specific aspects like persistency and recovery handling are means to improve the reliability of an application in the presence of HW failure. Persistency allows to keep system state over system crashes. Recovery handling consists of actions to recover from failures.

The aspect exception handling handles SW errors. The aim is to bring the system into a state which is presumed to be without error.

[Ren97] describes a pattern language for exception handling compatible with the BBM.

Transactions can be used to guarantee consistency of data updates.

The tss system, for example, used a database to explicitly administer persistent state information. The reliability design of the tss system is described in section A.3.3.2.

Note that the use of HW redundancy is very important for the design of high-availability systems. HW failures are handled by redundant HW, for an example see section A.2.2.

### 3.4.3 Security Design

Security is about preventing unauthorised users from making use of the system. Security design is done by choosing appropriate mechanisms for the design of access points and communication channels.

Examples are, sandboxing which provides a secure execution environment for foreign applications; encryption of stored and communicated data hinders unauthorised reading or change; capabilities are a way to structure various forms of user rights; and logging provides a history of events in a system.

Security can be handled as an aspect to structure overall access right handling.

### 3.4.4 Extensibility Design

Extensibility design is supported in the BBM task composability design through the design of various generic BBs. New features and applications can take advantage of a semantically rich infrastructure. New BBs can register themselves to these generics on the deployed systems. Necessary resources may be allocated via respective resource handling generics. The design for feature extension supports extensibility in general (see section 8.4).

### 3.4.5 Integrability and Testability Design

For large systems, integration, testing and the necessary rework contribute considerably to both the development time and the effort for product updates and new product features. It is essential that throughout the design of large products measures are taken to support integration and testing. The BBM supports integrability through BBs, component frameworks, incremental system integration and dynamic loading of deployment units in deployable systems (see chapter 7).

---

## 3.5 Other Design Tasks

Besides the quality-based design issues a number of other design issues are important. We give a short list of those issues. They present another perspective on the main design tasks of the BBM.

### 3.5.1 Feature Mapping Design

Product features are a major input for composability design. BBs are carefully factored to allow localisation of code, which implements the features. It is clear that product features are not completely isolated from the rest of the product. Therefore, feature relations, with the dependency relation being the most important one, are used. BB dependencies, which mirror feature dependencies, do not hinder flexible composition of products (see section 8.1).

### 3.5.2 Architectural Style Design

Architectural styles are an important means for designing the overall structure of a system. The most prominent architectural style of the BBM is layering. Objects are refactored according to layers in object design (see section 4.2) and BBs are refactored according to layers in composability design (see section 7.4). The BBM uses incremental layers for BBs to allow for incremental integration and testing.

Other architectural styles such as pipes and filters, and blackboards can be used in the main design tasks of the BBM. Both pipes and filters, and blackboards are architectural styles which describe the communication behaviour of objects. They are used during object design and, in the case where parallel execution is involved, during concurrency design.

The usage of architectural styles as single-view architectural approaches is discussed in section 9.3.

### 3.5.3 Data Structure and Algorithmic Design

Data structure and algorithmic design is mainly done during object design and aspect design. During object design data structures and algorithms are chosen for objects. During aspect design data structures and algorithms are chosen which support a complete aspect.

Composability design may lead to a refactoring of objects to place data structures and/or algorithms into generic and/or specific BBs.

System infrastructure generics may refactor objects to support certain aspects by providing generic data structures and/or algorithms.

If the existence of variation points leads to separation of data and operations a refactoring of objects for generic and specific BBs is necessary.

Furthermore, various kinds of interfaces require a design of data structures, for instance, interfacing between processes and threads may lead to shared data, buffers or queues.

### 3.5.4 Resource Usage Design

If resources are in a significant manner constrained, an explicit design of the usage of resources is appropriate. This leads to rules and guidelines about resource usage. It may also lead to the design of generic BBs which administer resources explicitly. System infrastructure generics are usually the place for administering pools of memory, I/O channels, file handles and thread classes and priorities. The same may hold for domain-specific resources where coordinated usage is supported by resource pools. Concurrency design deals with the factoring of code to enable an appropriate allocation of processor time.

### 3.5.5 Interface Design

Interface design is distributed over several other design tasks. Composability design deals with interfaces between BBs (see section 7.2 and section 7.6) and specifically between generic and specific BBs (see section 7.5). Concurrency design deals with interfacing between threads and processes (see section 6.2.2). Object design deals with external interfaces and for managed objects with distribution interfaces (see section 10.2.1).

### 3.5.6 COTS-Based Design

Commercial-of-the-shelf (COTS) packages can reduce the own development effort of an organisation. Examples of general COTS packages are operating systems and middleware packages for (graphical) user interfaces and communication. Various application domains are supported by commercial packages as well. COTS-based design is handled by composability design. Source libraries are included in BBs. Binary packages are dealt with as BBs.

An important question is the interfacing with these packages. Design strategies can be either to directly use the provided interfaces or to hide those interfaces behind some abstraction. There are several circumstances when extra implementation effort is required:

Binary packages require bidirectional linkage with a using package. A adaptation BB is necessary to achieve uni-directional coupling by providing a binding interface. The cluster of the binary packages together with the adaptation BB fulfils the requirements of a *normal* BB.

Binary packages may have resource allocation and usage strategies, which are not compatible with the rest of the system. A separate BB is necessary to shield this from the rest of the system.

Concurrency design may also be hampered if some packages are not thread-safe. A separate BB may handle application threads instead.

Products should have the flexibility to work with alternative packages possibly from different suppliers, which may have different interface abstractions. This may lead to specific interface BBs.

In general, the value of COTS packages depends not only on their provided functionality but also on the overhead and cost, which their use imposes on the system and the developing organisation.

---

### 3.6 Qualities of the BBM

We finish this overview chapter by taking a look at the qualities of the BBM. As noted in the beginning of the chapter, the focus on the actual construction elements is vital for the design of large systems. With the BBM, we develop both, global architectural models and construction elements.

Understandability of the architecture comes from these global models and the roles of layers and various types of BBs.

Factoring out of functionality into various kinds of generic BBs like component frameworks and system infrastructure generics leads to a certain leanness of the code [Wir95]. It has to be noted that generic BBs are not easily understood. But this is compensated by several advantages; first, generic BBs have explicit interfaces in contrast to OO frameworks [Szy98], second, the scope of the validity of system infrastructure generics is the complete system and third, component frameworks stand for domain-specific generic solutions. Furthermore, compared to monolithic frameworks, generic BBs are relatively small and there are many of them so that changes often can be kept local.

Leanness of products is supported by the fact that products are configured from the minimal set of necessary BBs (configuration to minimum, see section 8.3.4). The ease of building new products depends on the appropriateness of the architectural skeleton and frameworks for the new application. This is achieved by relying on the input from the application domain modelling.

Brooks defined conceptual integrity [Bro75] to be: "design conceived by a single mind". We refine his definition of *conceptual integrity* to be the suitability and orthogonality of a set of chosen design concepts for a certain system (class) as perceived by an expert designer. The BBM supports the achievement of conceptual integrity by focusing on the selection of design concepts from multiple perspectives. The explicit use of multiple perspectives provides a source for consistency across perspectives.

---

## 4 Object Design

Object design is about the design in the object dimension. Software objects, as defined by object-oriented modelling, encapsulate data and operations on these data. Objects are connected to other objects through relations.

Objects are used in the context of the BBM at four different levels. *Application domain objects* describe entities of the application domain. *Hardware domain objects* describe elements of the hardware system. They are both part of the first level. On the second level we have *domain-induced objects*. They are a replication of the objects of the first level into the software design space. They are generated from inputs of the BBM. On the third level we have *design objects*. They are refined and refactored due to the design tasks of the BBM. On the fourth level we have *implementation* or *programming language objects*. They are mostly a mirroring of design objects. However, specific implementations may use objects, which are more fine-grained.

Object design uses the application domain model defined by application domain modelling (see section 2.6.2) as input. The application domain model (first level) is used to derive an internal software model. There are several sources for objects from outside of the software, which will be described in the first section. The other design tasks, aspect design, concurrency design, composability design and deployability design, also lead to a refinement of the object structure (from second to third level). The second section describes those design objects.

---

### 4.1 Domain-Induced Objects

There are several sources for identification of objects outside of the software. The most obvious one is the application domain consisting of the application itself and the operational context of the application. Others are induced by sys-



class is explicitly calling the superclass for functionality it needs, while the superclass is delegating calls explicitly to its subclass. Object composition relations are easier to handle between different BBs because of explicit interfacing. A further reason for transforming inheritance relations is that inheritance at the programming language level is often only a compile-time concept, whereas BBs are deployment units [Szy98]. We shall not discuss inheritance any further (see for instance [RBP\*91] or [Szy98]).

#### 4.1.2 System-Relevant Functionality

The internal software model contains only the functionality, which is relevant for the system to be built. Remember that the application domain model may contain more than is actually required by the systems to be built. Therefore the required functionality has to be selected from the application domain model.

*Heuristic 2: Remove objects, attributes and relations which do not describe required system functionality.*

The issue is the relation between the domain model and the precise requirements for the systems to be built. For instance, the system may only do a certain kind of processing whereas the domain model is wider in scope. For instance, the system may have a control perspective or a recording perspective with respect to their real world counterparts

*Heuristic 3: Adapt the functionality of domain-induced objects to the required perspective of the system.*

This adaptation may take only parts of objects and leave other objects completely outside of the system.

Sometimes application domain modelling does already adapt its model to be an internal software model. That means that the domain model reflects the actual requirements. Such a domain model loses some of its power since it will not be stable under changing requirements. The reason why domain modelling as a separate activity was introduced was to achieve exactly such a stability by being independent from requirements for a specific system.

During aspect design we will execute an architectural concern analysis (see section 5.2) which may lead to the identification of functionality not identified in the domain model.

### 4.1.3 System Qualities

Besides the functionality defined in the application domain model, additional functionality may be necessary to achieve the required system qualities. During aspect design an *architectural concern analysis* and an analysis of quality specifications (see section 5.2) will be performed. This analysis may result in extended system functionality and lead to additional objects.

Examples of objects induced by system qualities are a database to save persistent values to survive system crashes and an encryption package to achieve security of communicated data. High availability may be supported by an administration of loadable modules which allows to upgrade a system and to fall-back in case of failure.

### 4.1.4 Hardware-Implemented Functionality

The next step is to factor out functionality which is implemented in HW.

An example is the processing of signals which may be completely implemented in HW.

The integration of different types of hardware into the system may lead to a distributed HW architecture. Software then has to be distributed over different HW instances (see section 3.2.5).

### 4.1.5 Hardware-Managing Objects

Specific processing HW has to be handled by software. Specific HW-managing objects are introduced to manage the HW functionality. Flexible HW boards will also need a specific object to manage the type and state of the board.

*Heuristic 4: Create one object per replaceable HW unit.*

On a standard computer platform this is already done for standard devices in the operating system. Also communication interfaces and channels are usually handled by the operating system. If there is no operating system or it handles only part of HW, the remaining HW resource handling has to be modelled in new objects.

In figure 25 external sources for domain-induced objects are shown.

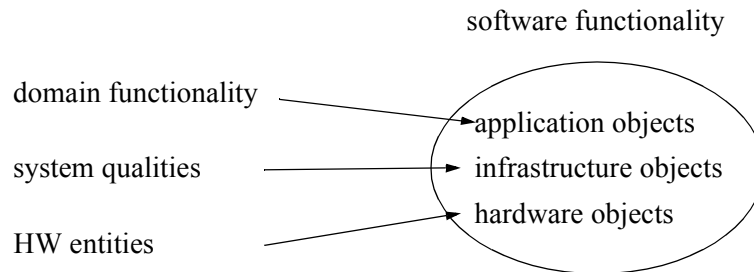


Figure 25: Examples of Sources of Objects

---

## 4.2 Design Objects

Domain-induced objects are refactored into design objects by the BBM design tasks. First we describe the refactoring to place design objects in layers (section 4.2.1). There are a number of cases for refactoring the object structure as a result of aspect design, concurrency design, composability design or deployability design. We shall describe examples of refactoring below.

More examples of refactoring are described in [DMN\*97], where for each axis of variability one additional object is recommended to hide the variations.

### 4.2.1 Classification of Functionality in Layers

Layering is very common in modelling functionality of large software-intensive systems. Layering need not be inherent in the functionality but is a way of introducing structure. The purpose of layering is to achieve separation of concerns and management of complexity. Having a layer for a certain kind of abstraction guides the identification of similar abstractions throughout the design.

Layers are specifically introduced to achieve portability and ease of evolution. Interface abstractions between two layers are chosen such that certain functionality can be executed on different hardware or operating systems platforms.

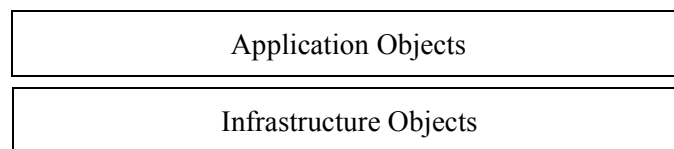
Layering is extensively used in the design of electronic systems. The main reason is to separate the concerns of hardware handling and of application functionality. Hardware technology and application domain functionality have different evolution speed. The functionality realised by the hardware is part of the application domain and

evolves with the application domain. The hardware technology changes faster than the application domain functionality. Hardware handling software abstracts from hardware specifics to abstract concepts on which the application functionality is based. The selection of functionality, its partitioning and its implementation technology change over time. The abstract nature of software makes the coupling of application functionality and solution technology a loose one.

Layers should be decided on from an engineering perspective rather than purely on the basis of the logical nature of the used abstractions, that is, a layer is a means to deal with a large amount of functionality. This means that the precise number and nature of the layers are not fixed, but are subject to system evolution, i.e. extending the functionality may necessitate the introduction of additional layers to handle complexity.

In the following we describe a possible rationale for the introduction of several coarse layers of objects.

Two initial classes of software are obtained by factoring out supporting functionality from application functionality. Objects are put in layers such that a layer may only use a lower layer but not vice versa. The supporting layer is called infrastructure and will be refined later. Two layers are shown in figure 26. An example is libraries for communication between application objects.



*Figure 26: Initial Two Layers*

*Heuristic 5: Refactor domain-induced objects to objects of an application layer and an infrastructure layer.*

On the basis of these two layers, which contain objects representing the application functionality and support functionality, additional layers may be appropriate.

A further layer may be appropriate if application functionality extends significantly. An application-specific platform encapsulates basic application abstractions. Various advanced applications make use of this platform. (figure 27).

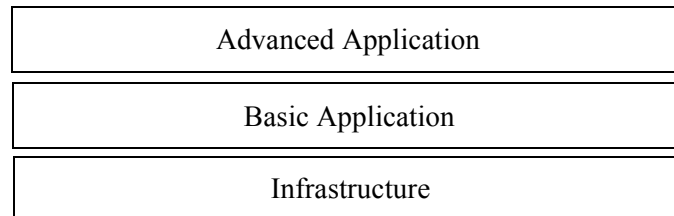


Figure 27: Three Layers with Basic and Advanced Applications

*Heuristic 6: Refactor large collections of application objects to objects of a basic application layer and an advanced application layer.*

Infrastructure functionality such as the operating system and other infrastructure services which should be used by all the application objects may be grouped in again a different layer, i.e. the lowest layer (figure 28) In such a case the application infrastructure is often called middleware.

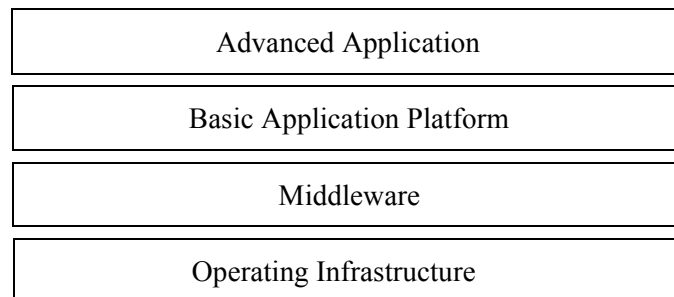


Figure 28: Four Layers with Operating Infrastructure

*Heuristic 7: Design objects which will be implemented by an operating system layer independent from an additional middleware layer.*

Layering refactors domain-induced objects. The design objects are arranged within the layers and similar abstractions are used per layer. Basic application objects are separated from advanced application objects and HW managing objects are separated from application objects.

In tss a basic call object deals with the originator of the call, the dialled number, the destination indicator and the call state. Advanced call objects are call objects containing various features such as follow-me and automatic ring-back on busy.

### 4.2.2 Communication Objects

To communicate between threads and processes communication objects are introduced. There are three types.

*Heuristic 8: Design messages which are sent between threads and processes in separate objects.*

*Heuristic 9: Design objects which hold message objects such as mailboxes, buffers, queues as separate objects.*

*Heuristic 10: Design protocol implementations as objects.*

### 4.2.3 Interface Objects

Designing BBs to be independent of each other leads to extra interface objects.

*Heuristic 11: Group interfaces of several domain-induced objects to one interface abstraction.*

*Heuristic 12: Limit the visibility of attributes and operations of domain-induced objects behind interface objects.*

### 4.2.4 Registry Objects

In designing for configurability in a product family, domain objects may represent different alternative or parallel features. Domain objects representing the variable functionality must be able to register themselves to some registry object to achieve configurability of BBs.

*Heuristic 13: Model registration functionality as a separate design object.*

If the variation is in an algorithm to be configured, the strategy pattern [GHJ\*94] may be used for the implementation.

Alternatively, a design object containing the common part of the variation may be extended to function as a registry. Then, variation objects must register themselves to it. However, a separate registry object is preferable because it can be used for all registrations.

### 4.2.5 Container Objects

Handling all or many instances of a class in a similar way can

- either be implemented as part of the class functionality,
- or it can be done via container objects such as lists, queue, etc.

The advantage of container objects is that the uniform functions are implemented by the container. The container then holds objects that need not be uniform. This allows for easier evolution.

*Heuristic 14: Use container objects for explicitly handling instances of a class in lists and queues.*

### 4.2.6 Functional Objects and Object Functions

The design for an aspect may be expressed in terms of aspect-specific objects. How do aspect-specific objects then relate to design objects? There are two ways of dealing with aspect-specific objects:

- aspect-specific objects may be transformed into common attributes and methods of all related design objects, or
- aspect-specific objects are new design objects.

Which alternative should be used depends on the amount of functionality which is modelled. The first alternative may be used for aspects with small functionality, while the second gives more structure for large aspects having separate data structures. The second possibility is also more natural with respect to object-oriented programming (see chapter 5).

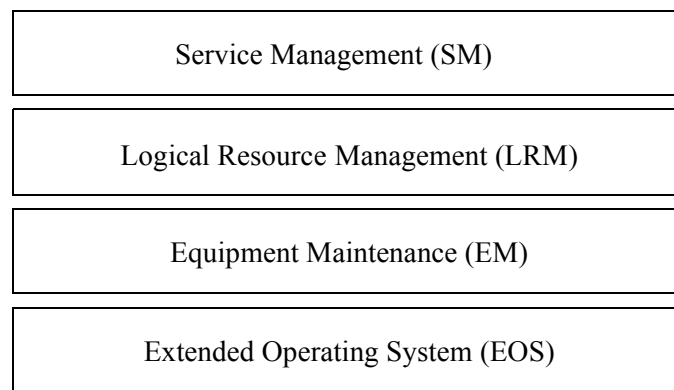
*Heuristic 15: Use separate objects to model aspects with large amount of functionality.*

A further question is whether data can be separated per aspect or whether data is global to all aspects. Experience indicates that a domain object may be reduced to a minimal design object which represents its identity and state. This minimal object is accessed from all aspects. All other data is modelled in design objects belonging to one or more aspects. The domain objects are then represented by a number of design objects. The above mentioned reduced domain objects consisting of identity and state only can be viewed as a separate *data aspect*. This will not be elaborated any further here because the practical consequences are limited.

---

### 4.3 Example: Layers and Some Objects of tss

The central controller SW of the tss system (see section A.3) is based on layers similar to those that have been developed over the years for most telecommunication infrastructure systems (for example [KBP\*95]). These coarse layers are major chunks of functionality and are therefore called layered subsystems. Typical layers are the extended operating system, equipment maintenance, logical resource management and service management. Figure 29 shows the four tss layered subsystems. We



*Figure 29: tss Layered Subsystems*

describe typical objects within the layers.

The service management subsystem (SM) comprises all the services of the application. Its main purpose is the provision of the system's intended functionality, that is, call signalling (heuristic 10) and call facilities.

The logical resource management (LRM) subsystem manages the data resources for the higher-layer subsystem. The LRM subsystem deals with data for signalling, lines (e.g., analog/digital subscriber line, basic and primary access, trunk lines) and facility data (e.g. call forwarding, *follow me*).

The separation into SM and LRM results from an application of heuristic 5. Internally, SM is further separated according to heuristic 6. The data-oriented objects of LRM and the service-oriented objects of SM result from the application of heuristic 1, heuristic 2 and heuristic 3.

The equipment maintenance (EM) subsystem consists of the control layer for the peripheral hardware and its interconnection structure, as controlled by the central controller. It deals with aspects of e.g. recovery and fault management of controlled equipment, and data distribution to the controlled equipment. All the instances of equipment are domain objects (heuristic 4). An abstract 64 Kilobit/s channel abstrac-

tion serves as a registry and interface object for different line types of LRM (heuristic 13).

The extended operating system (EOS) (heuristic 7) comprises, for instance, process handling, timer services, exception handling, data base, recovery mechanisms, administration of BB executables, file handling, memory management.

For more information about the SW architecture of tss see section A.3.

---

## 4.4 Explicit Transition

Objects are a useful concept for the execution of different tasks. Application domain modelling often uses objects to describe key domain concepts and their relations. Object-oriented design makes use of objects. Object-oriented analysis also uses objects, often in a mixture of application domain modelling and high-level design. And finally, the implementation is done via object-oriented programming.

In spite of the fact that in all the tasks of the architecting model (see section 2.6) the concept of an object is used, these tasks have a quite different character. The result is therefore that the semantics of an object is different in these tasks. For instance, in application domain modelling an object describes an entity of the application domain such as a system, or an instrument, or thing or a human being. It is not interpreted as a computational entity having state and operations such as in the case of object-oriented programming.

The object design task of the BBM makes the transition between objects in these different tasks explicit. Application domain objects are identified outside of the BBM and used as input for architectural design. Domain-induced objects are a replication of application and technology domain objects in the software design space. Aspects are identified as orthogonal functionalities to domain-induced objects. The core of the object design tasks consists of the transformation of domain-induced objects into design objects. Implementation objects are a refinement of design objects.

### Heuristics Overview

*Heuristic 1: Use application domain objects and relations to generate an initial object model of the product family by mirroring them in the software system.*

- Heuristic 2: Remove objects, attributes and relations which do not describe required system functionality.*
- Heuristic 3: Adapt the functionality of domain-induced objects to the required perspective of the system.*
- Heuristic 4: Create one object per replaceable HW unit.*
- Heuristic 5: Refactor domain-induced objects to objects of an application layer and an infrastructure layer.*
- Heuristic 6: Refactor large collections of application objects to objects of a basic application layer and an advanced application layer.*
- Heuristic 7: Design objects which will be implemented by an operating system layer independent from an additional middleware layer.*
- Heuristic 8: Design messages which are sent between threads and processes in separate objects.*
- Heuristic 9: Design objects which hold message objects such as mailboxes, buffers, queues as separate objects.*
- Heuristic 10: Design protocol implementations as objects.*
- Heuristic 11: Group interfaces of several domain-induced objects to one interface abstraction.*
- Heuristic 12: Limit the visibility of attributes and operations of domain-induced objects behind interface objects.*
- Heuristic 13: Model registration functionality as a separate design object.*
- Heuristic 14: Use container objects for explicitly handling instances of a class in lists and queues.*
- Heuristic 15: Use separate objects to model aspects with large amount of functionality.*

---

## 5 Aspect Design

In this chapter we describe the design task aspect design. In the first section the notion of an aspect as used by the BBM is defined. The second section describes the architectural concern analysis, which is performed to identify aspects. The third section describes an approach for deriving starter sets for aspect identification. The fourth section deals with the list of aspects in relation to the product family. The fifth section describes the design of aspect functionality. The sixth section takes a look at aspects in connection with BBs. The seventh section shows how aspects support architectural design in general. The last section places aspects in the context of multi-view design.

---

### 5.1 Definition of an Aspect

The motivation for the introduction of aspects is twofold. First, reasoning over and design of large systems is eased by explicitly identifying functionality which results from quality attributes and technology, and complements the application domain functionality. Second, upgrading of a system is eased by using unique designs for these different kinds of functionality. Application BBs which comply to these design are easy to integrate.

Aspects are a non-hierarchical, complete, functional decomposition of software functionality. To construct this decomposition, certain types of functionality are identified and factored into aspects. Initially all functionality is said to be part of the operational aspect. From the operational aspect those types of functionality are factored into aspects which crosscuts domain-induced objects. A software *aspect*, except for the operational aspect, is a certain type of functionality cutting across objects.

The list of aspects should be anchored in the application domain and the operational context of the product family and is defined for the entire family.

To identify software aspects we first look at the high-level functions of the system to be built. Relating these functions to the identified domain-induced objects will result in one of the following cases:

a function has relations to and/or defines functionality of a few objects only, for instance billing in telecom switch;

a function will be used by almost all the objects, for instance logging;

a function defines a type of functionality which is part of almost all the objects, for instance error handling.

In the first case the function will be handled as (part of) some domain-induced object. In the second case the function will also be handled as an object, but it will form part of the system infrastructure. In the third case the function cuts across objects and can be factored out from domain functionality to be an aspect. Which of the above mentioned cases applies to a specific function depends on the requirements for that function.

For example, consider the function *access control*. If access control is to be done only whenever a user attempts to enter a system and, once access has been granted, the user is free to use all functionalities, access control can be localised as an *access control object* which implements the required functionality. On the other hand, if access control should be more sophisticated and depend on user profiles and user groups that have certain rights at certain times, the functionality logically belongs to the application objects. A design may use access control lists and a state model enabling each object to decide whether access is to be granted. Access control could, then, be defined as an aspect of all the domain objects. An implementation could be split into a generic access control object which implements common functionality, and the domain objects which have to implement their specific access control functionality. The generic component would form part of the system infrastructure (see section 7.5.4). This example shows how the second and third cases defined above can be related.

### **BBM Aspects and Aspect-Oriented Programming**

A related definition of an aspect is given by Kiczales et al. [KLM\*97]. Independent of architectural discussions, limitations of object-oriented programming have been recognised. Examples are described where object-oriented modelling is too limited and leads to very complex code. An additional structuring is looked for, which leads to a natural design structure for those complex examples. Kiczales et al. call their approach aspect-oriented programming (AOP). They define an aspect to be functionality which crosscuts objects. An AOP-based program consists of a module containing the aspect source text per aspect. These source texts are automatically integrated by an aspect weaver into the normal object-oriented code. Source texts are not pol-

luted by code from other aspects and are claimed to be easier to maintain. However, automatic weaving of aspects relies on rules to be incorporated in the aspect code such as "before and/or after execution of command x do the following aspect statements". This anchoring of aspect code in the normal code introduces potential dependencies between different aspects. Problem-oriented (sub-)languages are designed in AOP for each aspect. This approach makes aspects very problem-specific. Since the concern of Kiczales et al. is programming and development of next generation programming languages, it could be said that their approach is a bottom-up approach while the BBM method is a top-down approach from the system point of view.

In contrast to AOP, aspects of the BBM are a complete partitioning of functionality. As described above, identification of aspects looks for crosscutting functionality to be factored out from the operational aspect. Analysis for different types of functionality is very important for large systems. A lot of functionality in large systems is not actually concerned with the application itself but with providing support for the application and with achieving the quality requirements for the overall system. In the tss system, for instance, only 20% of the code is application functionality. The rest is functionality like recovery functions to initialise and to bring the system in an operational state, database handling functions for persistent state information and error handling functions to detect error, isolate them and support a recovery of (possibly degraded) functionality.

Furthermore, aspects in the BBM are dealt with primarily on the design level and are not units of configuration like in AOP [CE00]. AOP and the BBM are complementary in this respect. Aspects as used in the BBM are standardised for the complete product family. Additional functionality often comes with new objects (see section 3.2.4). Introduction of a new aspect is a non-local change. In the BBM, common aspect implementations are factored out in system infrastructure generics (see section 7.5.4). For implementing BBM aspects see section 5.7.3.

### Steps of the Aspect Design Task

We give now a list of steps done during aspect design:

Initially taking the complete functionality as one aspect.

*Heuristic 16: Take the complete functionality as the first aspect called operational aspect.*

Analysing domain-induced objects for crosscutting functionality

*Heuristic 17: Look for common behaviour of domain-induced objects. Allocate similar cross-cutting behaviour to one aspect.*

Performing an architectural concern analysis to find additional aspects (section 5.2).

Using starter sets [CE00] of potential aspects to support the aspect identification:

Previous design experience is reused to identify aspects. This often shortens the aspect analysis. Several sources for the creation of starter sets will be given in section 5.3.

Standardising the list of aspects for the product family:

This may lead to fewer aspects because some may be only relevant for a single or a few products. (The identified product-specific crosscutting functionality may be supported by the design of a generic BB during composability design.)(section 5.4)

Determining the functionality per aspect:

After the identification of aspects the precise functionality has to be determined. This may involve more functionality than originally analysed. For instance, error handling may consist of error localisation, notification, recovery and logging (section 5.5).

Making a global aspect design:

A unique design of an aspect increases conceptual integrity. The number of design concepts will be easier to limit with this global scope. This may lead to specific models like initialisation model and configuration management model.

Factoring out common implementation parts in system infrastructure generics.

(This is part of composability design.)

Defining rules and guidelines per aspect.

The following section describe the concepts and the design steps in detail.

---

## 5.2 Architectural Concern Analysis

System design is a multi-disciplinary task with many stakeholders. Besides having required application functionality, a system must provide this functionality

with the required qualities. In addition, various stakeholders have different concerns about the system, its operational environment and its development context. The architecture has to address these qualities and concerns which often lead to additional functionality.

High reliability may lead to specific recovery and error handling. High uptime may lead to on-the-fly HW and SW corrections, updates and extensions and portability may lead to abstraction layers.

The BBM uses an *architectural concern* analysis to identify additional functionality and in particular to identify new software aspects. However, functionality which does not lead to a new aspect may lead to new objects or to an extension of existing ones.

The architectural concern analysis consists of several steps and is presented in section 5.2.1. In section 5.2.2 we take examples of architectural concerns and analyse them for system aspects, the first step in the architectural concern analysis. In section 5.2.3 we describe how system qualities and available technology can induce software aspects.

### 5.2.1 The Analysis Steps

We use an architectural concern analysis to identify new aspects. The outcome, however, may also lead to new objects or extend existing aspects and objects. The architectural concern analysis takes examples of lists of architectural concerns which are compiled in various contexts to consolidate design experience. These lists are analysed to identify new software aspects for the product family which we design.

The analysis consists of five steps and is shown in figure 30. The analysis is presented now step by step.

In the next section we will present four lists of architectural concerns and analyse them for software aspects of a hypothetical system. The list of architectural concerns are taken from checklist or standards and represent design experience of certain domains.

The first step is to analyse the architectural concerns whether they influence the system implementation or only the context of system implementation. We call the architectural concerns which influence the system implementation *system aspects*.

Examples of architectural concerns which influence the context of implementation are cost structure and testing strategy, while performance and security are system aspects.

The second step is to analyse system aspects whether they directly specify system functionality or only put constraints on how system functionality is to be implemented. System aspects which specify functionality may be application specific or result from the operational environment.

Examples for system aspects which come from the operational environment are field service, field test and user guidance.

System aspects which only constrain the implementation of system functionality fall in two categories. Either they constrain the implementation from an outside perspective and are called system qualities or they constrain the implementation from an inside perspective and need to be expressed by design guidelines. System qualities can indirectly induce functionality (see section 5.2.3).

The third step is to analyse system aspects specifying functionality whether they specify functionality to be implemented in software or in hardware.

The fourth step is to analyse system aspects specifying software functionality whether this functionality cross-cuts domain-induced objects or is a domain-induced object itself.

The fifth step analyses functionality which cross-cuts domain-induced objects whether it is product specific functionality or stable for the complete product family (see section 5.4).

The architectural concern analysis leads to several possible outcomes (the leaves of the tree in figure 30). An architectural concern

- influences the context of system implementation, that is, the development organisation has to take it into account; these are out of the scope of the BBM; additional methods and strategies have to be employed; or
- constrains the system implementation from the outside, i.e. a system quality, or from the inside, i.e. design or implementation guidelines; system qualities are analysed in section 5.2.3; or
- specifies hardware functionality; these are out of the scope of the BBM; or
- specifies a domain-induced object; this is an input for refinement of object design or

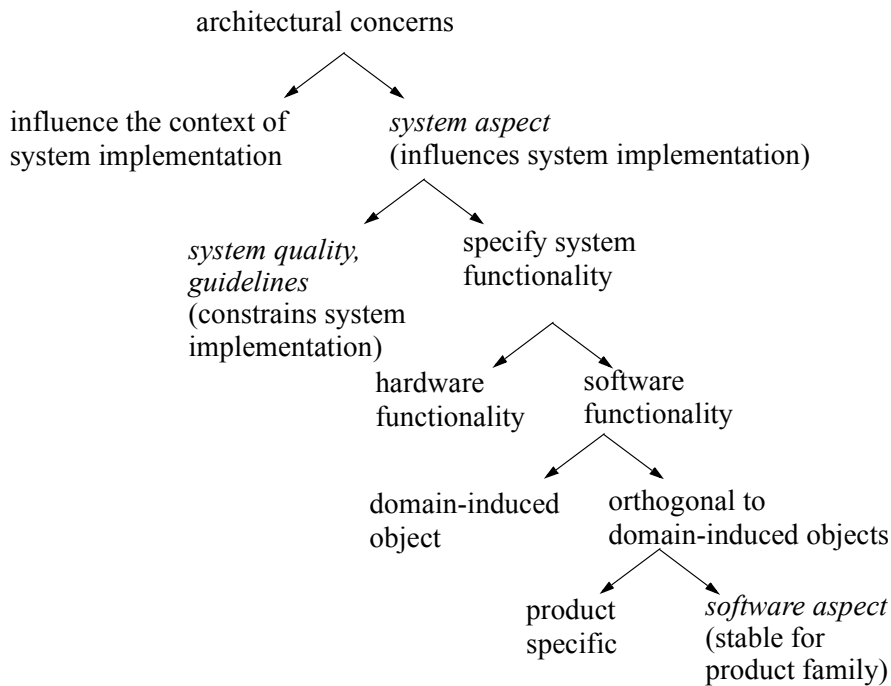


Figure 30: Architectural Concern Analysis

- specifies aspect functionality which is specific for a certain product; this is an input for refinement of object design and possibly composability design to analyse if the functionality can be supported by a component framework; or
- identifies a software aspect.

Note that a specific architectural concern does not always lead to the definition of an aspect.

### 5.2.2 Architectural Concern List Examples

In the following subsections four different collections of architectural concerns will be presented. These concerns are taken from different contexts and we use them to demonstrate a high-level architectural concern analysis. They can be used as a starting point for the identification of new software aspects for a particular product family. The first example is the popular distinction in functional and non-functional requirements. It is too context-dependent to be useful for deriving additional functionality. The second example is a list of quality attributes collected by people from the Software Engineering Institute. The third example is a list of system management functional areas taken from a telecommunication

standard. The fourth example is a list architectural concerns collected from design experience in medical imaging systems.

These lists are not part of the BBM but are examples from design experience which we can take as input for analysing functionality of a specific product family for new aspects.

*Heuristic 18: Use lists of architectural concerns from design of similar systems for analysing the required functionality for the identification of aspects.*

### **Functional and Non-Functional Requirements**

A distinction is often made between functional and non-functional requirements. A well-designed system is required to exhibit many more properties, besides its functional characteristics. Depending on the system to be built, system properties such as performance, safety, technology choices, testability, reuse, portability, use of standards, etc. may be among the customer requirements. In general, a customer may choose not to specify those requirements at all, to specify them only partially or to specify them fully. However, implicit system properties which are expected to be present in all systems of a certain class in a specific market segment have to be added to those explicitly specified. Furthermore, additional requirements may come from a development organisation for achieving internal benefits such as consistency with a product policy. Therefore, the distinction between functional and non-functional requirements is too context-dependent to be useful for direct use in finding software aspects. System aspects (see section 5.2.1) cover both functional and non-functional properties.

### **Quality Attributes**

Quality attributes constitute an important view on a system. Bass et al. [BCK98] give a classification in four classes. They distinguish between business qualities, quality attributes discernable at run time, quality attributes not discernable at run time and intrinsic architecture qualities.

The following business qualities are mentioned: time to market, cost, projected lifetime of the system, targeted market, roll-out schedule and extensive use of legacy systems

Quality attributes discernable at runtime are, for instance, performance, security, availability, functionality and usability.

Quality attributes not discernable at runtime are, for instance, modifiability, portability, reusability, integrability and testability.

Finally, intrinsic architecture qualities are conceptual integrity, correctness and completeness and buildability.

These qualities are useful for guiding the process of architecting a system. The architectural concern analysis leads to the following results. Business qualities influence the context of system implementation only. Intrinsic architecture qualities are system aspects and constrain the system implementation through their guidance for internal architectural characteristics. Quality attributes, both those discernable at runtime and those not discernable at runtime, constrain the system implementation from an outside perspective and are system qualities. They will be further analysed in section 5.2.3.

### **Operator-Oriented System Functionality**

In the field of telecommunication infrastructure systems, tasks and procedures of operators have been classified. Classification groups tasks and procedures so that different types of operators can be assigned to each class.

A traditional classification comprises operation, maintenance and administration tasks. It is often abbreviated as OMA.

FCAPS is the classification of the OSI system management functional areas (SMFAs) [X700]. The functions are divided into fault management, configuration management, accounting management, performance management and security management.

These operator-oriented function classifications influence system implementation directly and are system aspects. In the tss product family fault management, configuration management and performance management were software aspects. Accounting management and security management lead to domain-induced objects. Accounting management was implemented as a billing application and security management was implemented as a login and operator rights management application.

### **Checklist of Architectural Concerns**

G. Muller made a checklist of architectural concerns [Mul98] for designing medical imaging systems. His checklist is relevant for designing other systems as well. Muller pointed out that system architects have to take all these concerns into consideration, i.e. know the specific requirements, communicate with the various stakeholders, assess the relative importance of the individual requirements, etc.

The architectural concerns mentioned in the list represent a wide variety of mostly technical views. They relate to the system (to be built), its development environment and its use environments. We mention the list because of its breadth of technical issues. It helps to design a system without bias to certain technical issues. We have separated the list into items that represent system aspects and those which are broader architectural concerns. The following system aspects are listed:

- application requirements,
- functional behaviour,

functional chain specifications (print, store, etc.),  
information model: world standardisation, company standardisation, department  
standardisation, application specific,  
image quality,  
performance, throughput, response time,  
typical load,  
resource usage (CPU, memory, disk, network, etc.),  
module design, process design, function allocation (method, file, component,  
package),  
selection and use of mechanisms,  
installation, configuration, customisation, etc.,  
configuration management (technical and commercial),  
safety, hazard analysis,  
security,  
interfacing to other applications,  
factory and field testability.

Architectural concerns that are broader than system aspects are:

test strategy, harnesses, suites, regression,  
re-use consequences, provisions, development process impact, organisational  
impact, business impact,  
interoperability, other connected systems, selected partners, other vendors,  
verification,  
assessment of strong and weak aspects, road map for all views,  
technology choices (software, hardware, computer, dedicated digital, make/buy),  
system engineering (cables, cabinets, environment, etc.),  
cost structure (material, production, initial, maintenance, installation),  
logistics, purchasing (long lead items, vulnerability, second sourcing).

A complete architectural concern analysis for a medical imaging workstation would describe most of its design decisions.

These lists are used as a starting point for the identification of software aspects. The results of the architectural concern analysis depends on the product family for which the analysis is done.

### 5.2.3 System Qualities and Available Technology

Similar to the discussion above about the architectural concern analysis, software aspects can also be induced by system qualities and technology. As mentioned in section 3.4 there are no straightforward design methods for system qualities.

The characteristics of available technology play a crucial factor in designing a system having certain qualities. These characteristics determine if the simple use of a certain technology is sufficient or if a careful design using that technology is necessary. Design to achieve these qualities often leads to additional functionality, the selection of specific design mechanisms and an implementation masking the shortcomings of underlying technologies.

For instance high reliability may be realised through automatic recovery, diagnostics and error handling functionality. Early products of the tss family contained an error correction unit for memory access because memory technology was unreliable.

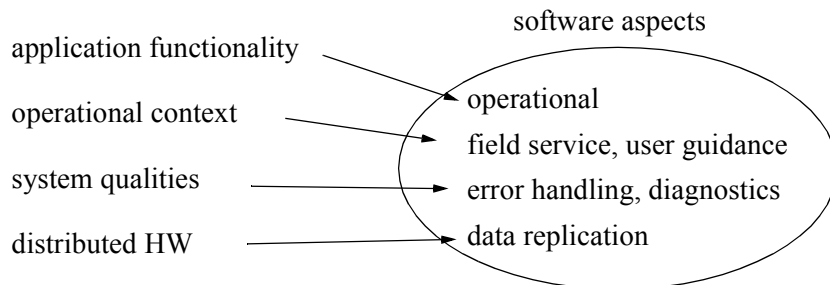
Supporting functionality is new system functionality and will be analysed according to the presented scheme (figure 30). The analysis identifies software functionality either as domain-induced objects or crosscutting to objects, and furthermore determines whether the functionality is a stable software aspect for the product family.

#### Summary

The functionality of software aspects is induced by the application functionality itself, by functionality for the operational context, by system qualities and by technology.

Examples of aspects are the operational aspect induced by the application functionality, a field service aspect and a user guidance aspect induced by the operational con-

text, an error handling aspect and a diagnostics aspect induced by system qualities, and a data replication aspect induced by distributed HW (see figure 31).



*Figure 31: Examples of SW Aspect Stimuli*

The relation between system aspects and software aspects can be summarised as follows. A system aspect either

- has no relation to software aspects, e.g. it
  - constrains a system implementation (system quality, guidelines), or
  - is (partially) handled outside SW, e.g. by HW, or
  - is realised by a functional unit, e.g. domain object, BB, subsystem, or
- is handled by a specific SW aspect, or
- is subsumed under another SW aspect, or
- is distributed between several other SW aspects and/or functional units.

Describing how system aspects are dealt with in a specific system and which software aspects are used, is a way to consolidate design experience (see section A.3.3.2 for an example).

---

### 5.3 Starter Sets for Aspect Identification

Additionally, SW aspects can be identified by using starter sets [CE00]. Starter sets are lists of aspects from other products. They represent design experience from other projects. The use of such experience may shorten the analysis for aspects described in the previous section. Starter sets of SW aspects and examples of specific aspect designs can serve as consolidated design experience.

*Heuristic 19: Use lists of aspects from other systems as starter sets for aspect identification.*

In the rest of the section we will give examples of such aspect lists which can be used as starter sets. Examples of specific aspect designs are described in section A.3.3.3 and section A.3.3.4.

### **Example: tss SW Aspects**

We shall give the list of SW aspects of the tss system [Bau95] as an example. These aspects are derived from the requirements and are a consequence of the system architecture (see section 10.1.1 and appendix A):

- system management interfacing
- recovery
- configuration control
- data replication
- test handling
- error handling
- diagnostics
- performance observation
- debugging
- overload control
- operational

The aspects are described in more detail in section A.3.3.1. The tss software aspects make reference to the *system management functional areas* (see section 5.2.2). Note that the areas *accounting management* and *security management* are not aspects in tss. The reason is that accounting management and security management are realised as objects. Accounting management is implemented as a set of BBs in the logical resource management layer (see appendix A) and security management is implemented generically via login procedures and user profiles in the operation and maintenance terminals (section A.2.1).

### **Example: Aspects of the Intentional Programming System**

Another example is the list of aspects of the intentional programming system as described in [CE00]. An intention is a set of programming language features. The intentional programming system allows to extend programming languages with extensions which are close to the semantics of the application domain, that is, the intentions of the application specialist. The intentional programming system allows the user to define intentions and provides generic support for certain kinds of functionality. These kinds of support functions are orthogonal to intentions and are SW aspects. They are:

- editing, that is a set of language features may have their own way of editing, e.g. with graphical or textual support;

display, that is a set of language features may have graphical, textual or mixed way of displaying;

translation, that is a set of language features may have their own way of translating it into the internal syntax graph representation. Similar

debugging,

code optimisations,

profiling,

testing, and

error reporting may be supported by each intention specifically.

The intentional programming system provides interfaces per aspect which may be used by the functions of an intention.

---

## 5.4 List of Aspects for the Product Family

Aspects are identified by analysing the functionality of a system. If not already done from the beginning, the results of this analysis have to be placed in the perspective of the product family. Some of the aspects may be important in some products only.

*Heuristic 20: Select only those aspects which are relevant for the complete product family as SW aspects.*

This may lead to fewer aspects because some may be only relevant for a single or a few products.

*Heuristic 21: Support identified product-specific crosscutting functionality through the design of a generic BB during composability design.*

The operational aspect has a specific character. It contains all functionality not factored out into other aspects. For example, domain functionality such as the handling of calls in a telecommunication switching system or the taking of images in a medical imaging system may be part of the operational aspect.

---

## 5.5 Designing Aspect Functionality

A major part of the aspect design is the design of the aspect functionality. The functionality of each aspect has to be defined. This may involve further refinement of functionality. For instance, error handling may consist of error localisation, notification, recovery and logging.

*Heuristic 22: Limit the number of different design concepts per aspect to increase conceptual integrity.*

A unique design for a complete aspect makes the aspect easy to understand. This leads to specific models like an initialisation model and a configuration management model. However, design trade-offs have to be made to achieve all relevant system qualities.

An example of a design of an aspect from tss is the recovery aspect. It consists of a common recovery model where all BBs can register 9 different types of initialisation methods. Each of the 6 types of recovery executes a subset of these methods in a pre-defined order. The design is given in more detail in section A.3.3.4. As a further example the tss configuration control model is described in section A.3.3.3.

[Ren97] describes a design for exception handling similar to the one used for the tss system. Eight patterns addressing different facets of the design form a pattern language.

Sometimes it is more appropriate to have a small number of specific designs instead of a single one. For instance, error handling for HW faults will be different from handling of communication failure. Areas of aspect functionality with similar characteristic should be identified and uniform design concepts should be used within each area.

*Heuristic 23: Weigh the smaller number of aspects with potentially different designs against a larger number of small aspects with a unique design.*

The goal for the introduction of aspects is a better overall design of the system functionality.

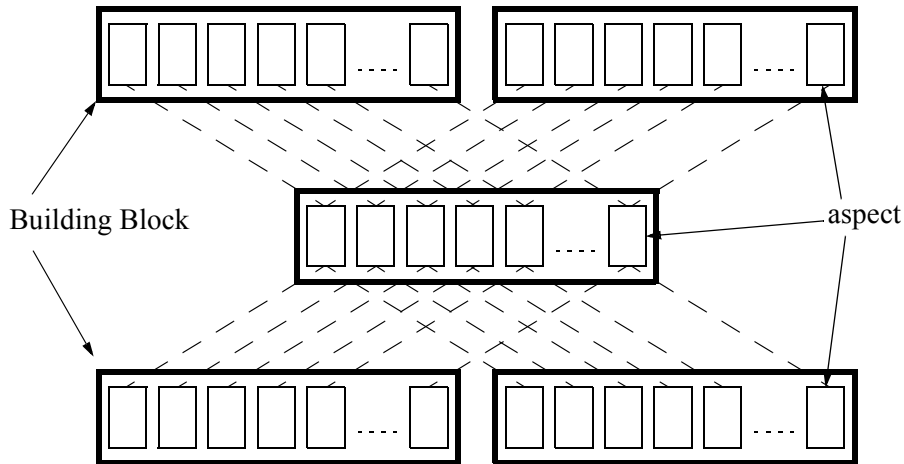
Common implementations of elements of aspect functionality is put into system infrastructure generics (SIG) (see section 7.5.4).

For instance, the implementation of the recovery aspect of tss is completely factored out into a SIG. The implementation of the error handling aspect of tss is only partially

factored out. The guideline for the error handling aspect states that failures should be analysed and faults be handled as local as possible. Appropriate recovery actions should be taken as soon as possible. Reporting should be done only for original faults. This functionality could not be factored out entirely because it involved too much local knowledge about the possible failures and faults. Only the reporting of faults was implemented by a SIG.

---

## 5.6 Aspects and Building Blocks



*Figure 32: Aspect Structuring of Building Blocks*

As we argued in section 3.3, we assume that most of the systems designed evolve mainly in the object dimension. Most of the BBs contain one or more complete domain-induced objects. Aspects can be used as a standard structuring for each BB.

*Heuristic 24: Introduce a standard structuring for BBs by letting all aspects be present in each BB, even if some of the aspects are empty in a particular BB.*

Figure 32 shows five BBs together with a standardised list of aspects within each BB. The dashed lines connect identical aspects.

Note that some of the aspects such as debugging may require functionality to be present in all the BBs. Functionality of other aspects such as error handling may be empty in BBs where no errors occur, for instance those which do not handle hardware or external interfaces.

Product families designed with the BBM are primarily decomposed into BBs. Since BBs often contain clusters of objects, aspects constitute a second-order design-time decomposition within the BBs. Introducing a new aspect into a product family will involve an almost maximum change effort because all domain-induced objects and their BBs are affected.

### **5.6.1 Aspect Completeness of Building Blocks**

Ideally, application features are modelled so that they can be added to an installed system without changes (see section 8.3.1). The goal of the BBM is to implement systems from pluggable BBs only. To achieve this property BBs have to be independent, that is, the insertion of a BB into a system must not necessitate changes to be made elsewhere in the system. Independence, however, is always relative to a given infrastructure. Aspects combined with a well-designed infrastructure are a means for achieving independence of BBs. We introduce the term aspect-complete for such an independence and define that a (set of) BB(s) is aspect-complete if it is responsible for allocating all of its required resources itself, and implements all aspect functionalities [Mül97]. Such a BB is a self-describing component (see section 7.8) because it contains descriptions for all its needs from the rest of the system.

---

## **5.7 Further Usage of Aspects**

Besides for designing functionality, aspects can also be used for structuring of reviews, documentation and implementation.

### **5.7.1 Aspects and Reviews**

The list of aspects can be used by the architects to check the functional completeness of the identified BBs. Functionality has to be specified for each of the aspects, such as the initialisation actions of BBs, the faults to handle and the configuration data.

*Heuristic 25: Use the list of aspects for checking completeness during review sessions. Structure large review team by allocating aspects to specific reviewers.*

### **5.7.2 Aspects and Documentation**

In the BBM the notion of a BB is pervasive, that is, it is an entity of specification, design, implementation and deployment (see section 11.5). Each BB has its own documents.

*Heuristic 26: Make a separate chapter per aspect in the BB documents.*

Furthermore, the list of aspects may be used for completeness checking in review sessions of BB documentation (see above).

### **5.7.3 Aspects and Implementation**

In the implementation each object method is characterised by a triple <object, process, aspect> in the design space, i.e. each method is part of an object, is driven by a process and is part of an aspect (see section 3.3). Making aspects standard structures of a BB (see section 5.6) leads to a uniform modularity. Files, programming language modules or naming conventions are means of implementing this modularity.

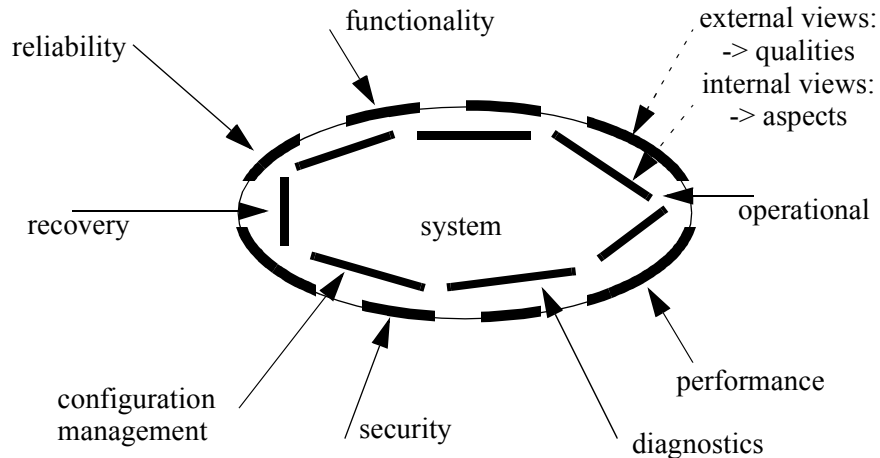
*Heuristic 27: Structure the implementation of a BB according to aspects.*

---

## **5.8 Aspects and the Whole**

Aspects are a means for structuring functionality. Whereas stakeholder concerns and system qualities are multiple external views of a system's functionality, aspects are types of functionality from multiple internal views. Aspects, stakeholder concerns and system qualities, are ways of capturing the whole of a system. A stakeholder will usually have a single-view approach to a system. Things which are not visible at the first levels of the single-view decomposition are details, with respect to that view, at lower levels of the decomposition hierarchy. The problem with this approach is that important system characteristics are described at different levels. This makes a system hard to understand.

A multi-view approach addresses the whole from different angles. In a multi-view approach, the important system issues are not hidden by a dominating view



*Figure 33: Multi-View Approaches*

but are addressed by their own view. Different views complement each other. In the end the whole can be tackled more easily than with a single-view approach. A single-view approach is like addressing all construction problems with a Swiss Army knife, and this requires a lot of effort.

Object-orientation in itself is such a single-view approach. Every application concept is an object or coupled to an object. The concurrency design and the different aspects as introduced in this thesis complement object-orientation to a multi-view approach.

### Heuristics Overview

*Heuristic 16: Take the complete functionality as the first aspect called operational aspect.*

*Heuristic 17: Look for common behaviour of domain-induced objects. Allocate similar cross-cutting behaviour to one aspect.*

*Heuristic 18: Use lists of architectural concerns from design of similar systems for analysing the required functionality for the identification of aspects.*

*Heuristic 19: Use lists of aspects from other systems as starter sets for aspect identification.*

*Heuristic 20: Select only those aspects which are relevant for the complete product family as SW aspects.*

- Heuristic 21: Support identified product-specific crosscutting functionality through the design of a generic BB during composability design.*
- Heuristic 22: Limit the number of different design concepts per aspect to increase conceptual integrity.*
- Heuristic 23: Weigh the smaller number of aspects with potentially different designs against a larger number of small aspects with a unique design.*
- Heuristic 24: Introduce a standard structuring for BBs by letting all aspects be present in each BB, even if some of the aspects are empty in a particular BB.*
- Heuristic 25: Use the list of aspects for checking completeness during review sessions. Structure large review team by allocating aspects to specific reviewers.*
- Heuristic 26: Make a separate chapter per aspect in the BB documents.*
- Heuristic 27: Structure the implementation of a BB according to aspects.*

---

## 6 Concurrency Design

Concurrency design is about the mapping of functionality to processing resources like threads and processes. The concept of an address space is important for concurrency design and deployability design (see section 7.9).

This chapter consists of two sections. In the first section we describe the usage of address spaces and the consequences for objects, aspects and threads. The second section describes the concurrency design focusing on an internal concurrency model.

---

### 6.1 Using Address Spaces

Both, units of deployment and units of execution make use of address spaces. Address spaces do not actually represent execution structures. Rather, they represent boundaries to execution structures. Address spaces can be a consequence of hardware boundaries or may be instantiated by software.

Symmetric multi-processing uses a single address space with several processors. Threads are executed on the processors according to allocation strategies. Identification of threads is done in the same way as with single processor address spaces.

Address spaces are used for two purposes.

*Heuristic 28: Use address spaces as failure containment units. Recovery from failure is realised within an address space.*

*Heuristic 29: Use address spaces to design for deployability. The freedom to relocate functionality to different processors depends on the absence of common data between address spaces.*

For the discussion of the BBM, we shall assume that both the object and the thread dimensions are partitioned by address spaces. In the following we shall describe the relations between the three dimensions and address spaces.

Object dimension: Objects may be internally distributed over address spaces. The concept of a managed object (section 10.2.1) does this via an asymmetrical distribution, that is, the proxy object forms part of the controlling equipment, while the real object is controlled by it. A symmetrical distribution over different address spaces which splits an object into multiple peer subobjects is also possible. We will not discuss this concept here and assume that if the concept of distributed objects is needed, it will be modelled explicitly as a collection of objects.

Thread dimension: Threads are seen as logical threads of execution which cross address space and even hardware boundaries. This view may be helpful in the early stages of architecting. Later, these logical threads are split into a number of physical threads, limited to using a single address space. The physical threads are connected via inter-address space communication. In the BBM, concurrency design maps functionality clustered into logical threads to physical threads of the execution environment.

Aspect dimension: Aspects constitute a classification of system functionality which cuts across objects and they are not affected by address space boundaries.

---

## 6.2 Concurrency Design

Concurrency design is about the mapping of functionality designed during object design and aspect design to execution structures of the computing platform. **Logical threads** are used to describe parallel execution of functionality described in domain-induced objects and aspects. Physical threads are a refinement of logical threads according to address spaces. We use the term **physical thread**, or thread for short, to consist of all methods and objects, which execute under its control. This is sometimes called the reach of a thread.

The use of term *process* in the context of execution structures is confusing. The term *physical thread* means an independent unit of processor cycles allocation having its own stack. Some real-time kernels use the term *process* in this way, others use the term *task*, which again is used for minimal independent execution units of application functionality. In the context of workstation operating systems, Solaris and Windows NT use the term *process* either as a pure handle for resources and for an address space or as additionally having also a thread. Throughout this thesis we will omit the term *process* to denote execution structures and rely on the terms *thread* or *address space* depending on which facet of the term *process* we refer to.

Rules on the architectural level must be given to guide local design. Because of the complexity of the relation of threads to objects, aspects and BBs, a consistent global picture of all threads is essential to good system design. Such a global picture supports the understanding of local interface descriptions. The usage of threads is not considered local implementation detail.

*Heuristic 30: Consider the use of a thread on the architectural level.*

*Heuristic 31: An overview of all threads should be given in a global concurrency design.*

Besides the identification of threads we will also take a look at the identification of address spaces.

### **6.2.1 Determining Concurrency**

The design starts with initial steps which are later refined.

#### **Starting with behaviour of domain objects**

The basic approach for structuring functionality into threads is to look for intrinsic concurrency in the application domain. Behavioural modelling of the application domain is described via domain objects, their interaction and their internal states (see section 2.6.2).

*Heuristic 32: Mirror independent behaviour of application domain objects by separate logical threads.*

#### **Determining independent external sources**

Independent external interaction sources are users or external devices modelled as domain objects. External concurrency is the first source for determining internal concurrency. A system's internal concurrency structure should resemble the concurrency structure of its external environment (see above). This mirroring makes the concurrency structure easier to understand. A concurrency structure of a system with more concurrency than that of the application domain needs to introduce extra synchronisation within the system. A concurrency structure with less concurrency needs to explicitly switch between external sources. Two types of external concurrency are important.

Application concurrency is a consequence of actors in the application domain. These actors may be users or other equipment interfacing with the system.

*Heuristic 33: Use a separate thread to handle an external connection or external messages.*

*Heuristic 34: Cluster all functionality which is activated via object interaction by the external connection or messages into the thread.*

*Heuristic 35: Use a separate thread for the interaction of a user with the system.*

This is the user role task type of [Gom93]

Hardware equipment directly connected to a system is important for the concurrency structure of the software handling these connections. Take for instance a system controller having connections to controlled hardware equipment, or instances of the management systems. All equipment usually runs in parallel.

*Heuristic 36: Represent the receiving direction of an external channel or bus by its own thread.*

*Heuristic 37: Message sending over an external channel is done on the budget of the sending thread*

These heuristics are mentioned by Gomaa as I/O task structuring criteria [Gom93].

*Heuristic 38: Refine the design of a separate thread per bus to have thread instances per connected equipment instance to the bus.*

### **Prioritising aspect functionality**

Several heuristics are used to determine priorities of aspect functionality.

*Heuristic 39: Let internal consistency have priority over external reaction.*

Internal consistency of the system is vital for its correct functioning.

*Heuristic 40: Give operational tasks priority over background tasks.*

Certain types of logging may be skipped under heavy system load while other types may be more important than operational tasks. For instance performance logging may have a lower priority than operational tasks. Error logging may have a higher priority than operational tasks. Design decisions have to be compatible with the expectations of the stakeholders.

*Heuristic 41: Use separate thread per different priority.*

An example is the separation of fault handling actions from other actions in controller SW. If a fault message is received from some equipment, the states of the managed objects of the equipment and of all the dependable equipment have to be adjusted. To limit the effects of the failure, this update action has priority over other actions.

### **If necessary, encapsulating specific objects in a thread**

Internal concurrency may additionally be necessary to deal with different priorities of actions to be performed. This may concern functionality of specific objects. For instance, an emergency call may have priority over other calls.

*Heuristic 42: Use a separate thread per cluster of objects with given priority.*

This is often a mixture of Kruchten's outside-in approach [Kru95] and Gomaa's task priority criteria [Gom93].

### **Refining the logical threads into physical threads**

Physical threads are confined to a single address space.

*Heuristic 43: Split logical threads up into physical threads per address space.*

Inter-address-space communication connects physical threads.

### **Determining interfacing between threads**

Buffers or queues of messages or shared data may be used. The use of messages in the same address space may result in unnecessary copying of data (see also section 6.2.2).

Thread identification is also treated in Kruchten [Kru95] and Gomaa [Gom93]. The main difference is that in the BBM the design of the thread structure is an independent dimension besides the object and the aspect dimension.

Gomaa uses the term "object or function" when referring to the content of a thread. The BBM puts these concepts in different design dimensions. In one situation objects are used as concurrent units while in another situation aspects may be concurrent units.

### 6.2.2 Thread Interaction

Interaction between threads is always located within a BB. An object method calls a method of another BB in the same thread and data is transferred to the other BB. Particular BBs such as message handlers managing different kind of buffers or a socket manager may be involved. Call-backs may be used to actively deliver data to a BB (see section 7.2.4).

### 6.2.3 Concurrency and Aspects

Sometimes the suggestion is made that concurrency is just another software aspect. Why is that not the case?

Our definition states that an aspect is a type of system functionality. Examples are initialisation and fault management. The domain functionality is captured by the operational aspect. The list of aspects is a partitioning of a system's functionality. All the system functionality needs to be driven by threads. The concurrency design is different from aspects, it is the mapping of aspect and object functionality to the available processor time in form of threads. The unit of functionality allocated to a thread may be one or more objects or one or more aspects or part of them (see section 6.2.1).

In the tss system the aspect functionality of two of the four layers (equipment management and logical resource management) is handled by a set of shared threads (see section 6.2.4).

### 6.2.4 Example: Concurrency Design of tss

The concurrency design of the entire application functionality (Equipment Maintenance, Logical Resource Management, Service Management; see section 4.3) is based on one address space and comprises six thread types.

tss service management performs call processing. A singleton thread type receives the call-initiating messages from the peripheral cards. A call thread is started to handle a new call. The call thread type has instances for the maximum number of parallel calls allowed in a system, e.g. several thousand (heuristic 32).

tss equipment management and logical resource management perform control processing in two thread types. They go along aspects and cross many objects. A fault handler covers the operational, recovery and fault management aspects (heuristic 39), while a configuration handler covers configura-

tion control and data replication. A separate thread instance is used for each equipment instance at the peripheral bus (heuristic 32).

A further thread type covers the system management interfacing aspect of all three layers of the application functionality. It is instantiated per operator (heuristic 32, heuristic 35).

The entire incoming and outgoing communication is handled via one central BB which handles the bus connection. The incoming direction is covered by a singleton thread type (heuristic 36), while for the outgoing direction functions are provided which run under the budget of the sending thread (heuristic 37). The incoming messages are distributed from this single thread to the threads representing equipment instances (heuristic 38).

A more extensive description of the tss concurrency design can be found in section A.5.4.

### Heuristics Overview

*Heuristic 28: Use address spaces as failure containment units. Recovery from failure is realised within an address space.*

*Heuristic 29: Use address spaces to design for deployability. The freedom to relocate functionality to different processors depends on the absence of common data between address spaces.*

*Heuristic 30: Consider the use of a thread on the architectural level.*

*Heuristic 31: An overview of all threads should be given in a global concurrency design.*

*Heuristic 32: Mirror independent behaviour of application domain objects by separate logical threads.*

*Heuristic 33: Use a separate thread to handle an external connection or external messages.*

*Heuristic 34: Cluster all functionality which is activated via object interaction by the external connection or messages into the thread.*

*Heuristic 35: Use a separate thread for the interaction of a user with the system.*

*Heuristic 36: Represent the receiving direction of an external channel or bus by its own thread.*

*Heuristic 37: Message sending over an external channel is done on the budget of the sending thread*

*Heuristic 38: Refine the design of a separate thread per bus to have thread instances per connected equipment instance to the bus.*

*Heuristic 39: Let internal consistency have priority over external reaction.*

*Heuristic 40: Give operational tasks priority over background tasks.*

*Heuristic 41: Use separate thread per different priority.*

*Heuristic 42: Use a separate thread per cluster of objects with given priority.*

*Heuristic 43: Split logical threads up into physical threads per address space.*

---

## 7 Building Block and Deployability Design

*Building Blocks* are software components. The definition of the term *software component* follows that of Szyperski [Szy98]:

"Software components are executable units of independent production, acquisition, and deployment that interact to form a functioning system."

The BBM uses the notion of a product family (chapter 8) to cover the market aspect of components. This chapter explains BBs from a technical point of view.

BBs are design and deployment units. Identification of BBs usually goes along the object dimension, that is, a BB is a cluster of objects. However, this is no restriction. A BB can follow the other dimensions as well, or even encapsulate arbitrary parts of the three design dimensions. The main criteria are configurability and incremental integration, as will be outlined in this chapter.

The chapter starts with an overview of the composability design task. Then follows an explanation of the most important ingredient of a BB, its interfaces. The interfaces will be discussed only summarily, but this brief treatment will suffice for our purpose. A more elaborate treatment can be found in [Szy98]. BBs, like other software components, are based on a common component model. The component model defines standard properties for components. Component models will be introduced in the third section, after which layering of BBs will be explained. The roles of BBs, generic or specific, will be dealt with in the following section. A further section will take up the issue of interfaces again and relate it to layering and genericity. Hierarchical components in the BBM will be described then. A section follows which introduces the concept of an architectural skeleton as a way of organising inter-BB relations.

The last section of this chapter is about deployability design. Deployability design deals with possible deployment scenarios of products to the hardware environment. BBs are the minimal deployable units. Deployability design

describes rules for refactoring and grouping of BBs to adapt them to certain deployment scenarios.

If not described differently, we depict BBs by boxes and their directed dependency relation by lines between boxes where BBs located above depend on BBs located below.

---

## 7.1 Composability Design Overview

Composability design is about defining modularity to support the composition of products in the product family, to obtain manageable development units, to realise a simple feature mapping and to allow for incremental integration and testing. This overview covers the application of concepts explained in this and the next chapter.

Composability design consists of the following design steps:

### Clustering objects into BBs

There are several criteria for clustering objects into BBs. They are:

*Heuristic 44: Cluster objects into BBs such that coupling of objects across BB borders is low and cohesion of objects within a BB is high.*

*Heuristic 45: Cluster objects into a BB which represent a feature.*

*Heuristic 46: Cluster objects into different BBs which belong to independently evolvable parts.*

*Heuristic 47: Cluster objects into BBs such that a BB can be used as a work allocation units for 1 or 2 persons.*

### Identifying variation points of functionality which belongs to different features

Start with variation points identified in the application domain model.

Select those variations which can be solved by data. Configuration parameters may be supplied by users or via configuration files and passed to the appropriate places.

For variation points requiring code, analyse if the variation point is inside a BB or at an interface of a BB. If the variation point is at the border of a BB, the desired situation, in which variation is modelled by alternative BBs, is already reached.

*Heuristic 48: If the variation point lies inside a BB, refactor the BB such that the variation point lies at the border of a BB.*

Refactoring of BBs affects the object structure and may require refactoring of objects in object design.

### **Factoring out of common functionality in separate BBs.**

*Heuristic 49: Factor out functionality which is present in several BBs in a separate BB.*

This may also lead to a refactoring of objects.

### **Identifying interfaces of BBs:**

Create abstractions at the interfaces which are implementation-independent and efficiently executable. Often there is a tension between these two requirements.

*Heuristic 50: Take as main criterion stability under evolution, that is, an interface should be such that it can serve for those implementations and those usages which are likely to happen.*

### **Designing component frameworks**

A component framework (generic BB) is designed when a BB contains only the generic parts of an implementation. The various specific parts (plug-ins) are located in specific BBs. The specific BBs are extensions of the generic BB. A generic BB can usually not be used without its extensions. The coupling between generic BB and specific BB is usually tight, that is, changes of the implementation of the generic BB will affect the specific BBs. The interface of the generic BB to other BBs should be stable.

*Heuristic 51: Factor generic implementation parts which are used by several specific parts into a generic BB.*

There are examples of BBs which are generic w.r.t. several kinds of extensions. This leads to the introduction of the concept of a generic role (see section 7.5.2).

The introduction of a generic BB will usually lead to a refactoring of objects. But cases where one or more aspects of certain objects are factored out are also possible.

### **Identifying system infrastructure generics**

Generic functionality which is to be extended by almost all BBs is put in system infrastructure generics (SIG). A SIG is a special case of a generic BB.

*Heuristic 52: Take the implementation of common aspect functionality as a candidate for a SIG.*

Examples are the initialisation model of the system or the handling of user reports. SIGs are mostly part of the operating system or middleware packages.

### **Defining layered subsystems of BBs:**

Start with the layers defined during object design. Place a BB in the lowest possible layer of a layered subsystem (recursive layering) according to the dependency relation with other BBs.

### **Designing for incremental integratability and testing:**

Only uni-directional relations between BBs are allowed. Transform mutual dependence into a uni-directional one. The criterion for refactoring is again expected stability under evolution.

*Heuristic 53: Resolve mutual dependence between BB A and BB B in the follow way:  
if A is expected to be more stable than B, then make B depend on A; and vice versa  
if the communication between A and B is expected to be the most stable part, factor the communication out into a new BB and let both, A and B, depend on it.*

Give each BB sufficient functionality for useful tests. This means for a generic BB that it should have useful behaviour (null-behaviour) without any specific BB connected to it. Specific BBs contain extensions of the generic BB.

### **Doing detailed design of the BB**

Elaborate the identified interfaces and design the internals of BBs. This may mean refinement of objects, design of data structures and algorithms as part of the object design task and the aspect design task.

The extensive refactoring and factoring into generics leads to a compactness of the code (see section A.5.4) and reduces the size of the system. The detailed description of composability design is split into two chapters. This chapter describes the design of BB, layering and generics. Chapter 8 describes the design of a product family architecture. We will now look at the concepts in detail and start with interfaces of a BB.

---

## 7.2 Interfaces

First we describe abstraction interfaces and open implementation interfaces. Then we discuss the notion of a connector. Registration and call-back interfaces are introduced next. They are important concepts for achieving uni-directional dependencies. The last section discusses the relation between interfaces, and aspects and threads. In section 7.6 after the discussion of layering and generic BBs we will come back to the topic of interfaces.

### 7.2.1 Abstraction Interfaces

BBs have a *provides* interface, a *requires* interface and a body which is not visible but only accessible via the provides interface.

***Provides interfaces*** define functionality of a BB that can be accessed by other BBs. On the syntactic level, the *provides* interface consists of a (structured) list of method signatures and a description of all their data types. This may be extended with any semantic information which a particular context requires. For example, pre- and post-conditions may be added if necessary. A very common practice is to describe an interface protocol that describes a suggested or required ordering of method calls. However, we do not require such a description because not every context needs it. Provides interfaces should be carefully designed to be independent of implementation detail and consist of abstractions of the functionality which do not prescribe a certain implementation.

The ***requires interface*** (see figure 34) describes provides interfaces of other BBs that are required for the implementation of a BB.

Note that these interfaces include those explicitly identified during specification of an application as well as additional ones required for the implementation, such as those to infrastructure services.

Requires interfaces consist of a set of references to BBs, a list of those method signatures which are actually used by the BB and the description of all the used data types. The requires interface makes the dependencies of a BB explicit. To be

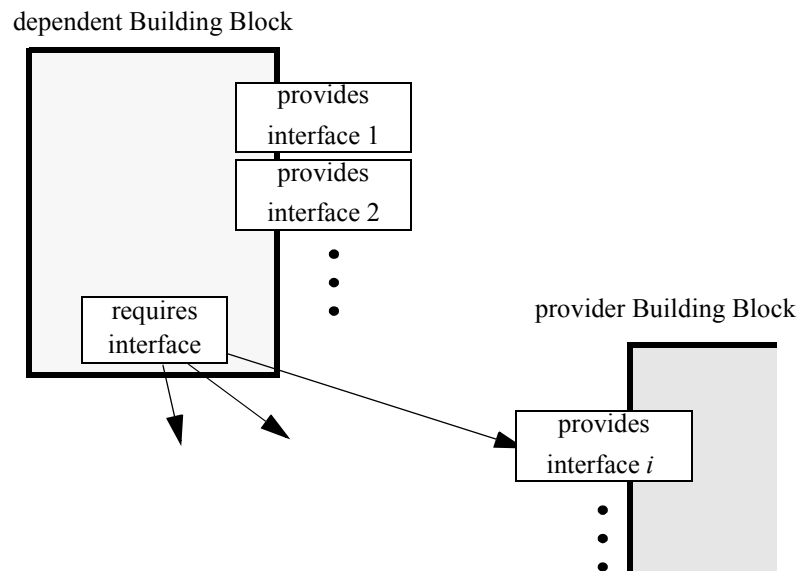


Figure 34: Requires and Provides Interfaces

able to use provides interfaces of other BBs, a BB has to reference (import) these interfaces.

*Heuristic 54: In the case of embedded systems, use importing of interfaces at compile time if needed for performance reasons. Otherwise use dynamic exploration of interfaces for more flexibility.*

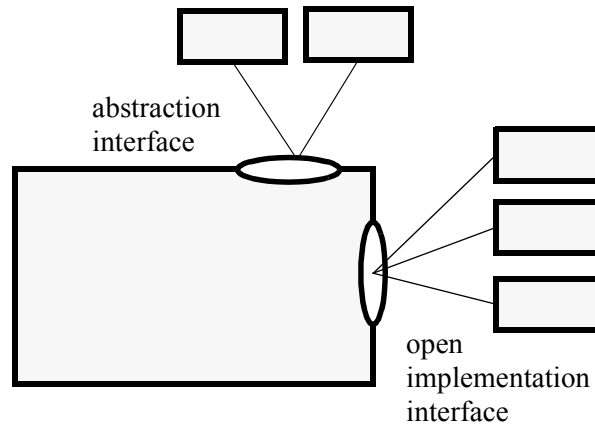
A BB is dependent on the BBs from which it imports interfaces.

For a discussion of the issue of object-oriented programming and stable interfaces we refer to [Szy98]. In particular, this work explains the problems involved in using implementation inheritance across component boundaries (see the fragile base class problems) and discusses alternatives. The BBM relies on object composition instead and avoids these problems, because BB interaction is realised via explicit interfaces.

## 7.2.2 Open Implementation Interfaces

Unlike abstraction interfaces, open implementation interfaces, as introduced by Kiczales [Kic96], do not establish an abstraction for information hiding [Par72],

but are connection points to other BBs which provide implementation alternatives (figure 35). The BBs, which extend a component via its open implementation interface, are extensions of that component. Open implementation interfaces are not intended to remain unchanged if the implementation of the component which offers them changes. Both, abstraction interfaces and open implementa-



*Figure 35: Abstraction and Open Implementation Interfaces*

tion interfaces are provides interfaces. In depicting BBs, we do usually not show the difference between abstraction and open implementation interfaces as done by [Kic96] shown in figure 35. The emphasis is only on the fact that there is a dependency relation between BBs.

Abstraction interfaces and open implementation interfaces may be compared to upper and lower interfaces, respectively, as described for Catalysis [DW99]. However, abstraction interfaces and open implementation interfaces are both provides interfaces. Catalysis, in contrast, uses lower interfaces both for plug-ins and for access to an underlying virtual machine. Plug-ins, however, are optional extensions and should rely on provides interfaces, while access to a virtual machine is part of the requires interface.

### 7.2.3 Interfaces, Components and Connectors

Some authors recommend modelling a system on the basis of components and connectors [SG96].

Interfaces are collections of methods at the programming language level. An interface is the most trivial connector. It connects two components, one of which implements the interface as a provides interface while the other implements it as a requires interface. More complex connectors are buses, pipes, blackboards, remote method call packages, etc. In fact, a connector can be any abstract data type or other

package which connects two or more components. In the BBM, all but the most simple connectors are implemented as components themselves.

The notion of a connector is useful in top-down modelling. Application entities can be connected abstractly. The precise properties of a connector can be determined later. Product variations may require different connectors. The BBM does not use connectors as a primary concept because the concept of a generic BB can be used instead. Connectors can be modelled as special classes of generic BB (for an example see section 7.5.4).

### 7.2.4 Registration and Call-Back Interfaces

Call-back interfaces offer a mechanism for making components minimally dependent on the context in which they execute. Besides the services part of the provides interface, a BB provides a call-back interface through which it can call methods of the calling BB. Instead of adapting a service-providing BB to many different interfaces of using BBs, the service-providing BB sets a standard which has to be met by using BBs. The service providing BB itself thus remains independent of the using BBs while adapting its behaviour.

The call-back mechanism is shown in figure 36. BB A defines a service interface, a register interface and a call-back interface. BB B, interested in the service of A, has to register methods, which conform to the call-back interface, with BB A. A *call-back* occurs when A calls the registered methods of B.

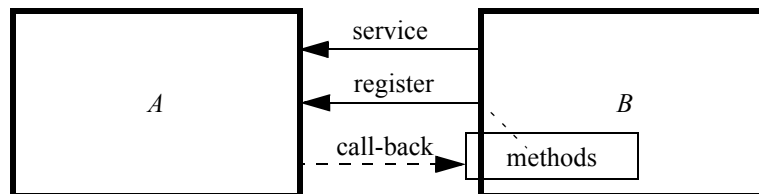


Figure 36: Call Back Mechanism

However, BB B may have to meet restrictions in the implementation of the call-back methods.

Examples are certain interfaces which may not be called during the call-back from A, and execution time limitations of the call-back (for a broader discussion of these restrictions, see [Cla85]).

The BBM requires that BBs have syntactically only unidirectional relations. Mutual dependencies are not allowed. This establishes a partial ordering of BBs.

The partial ordering of BBs is a necessary condition for integrating a system incrementally. If a design requires bidirectional communication, call-back interfaces have to be used to establish a bidirectional communication. Below, we shall give guidelines for design with unidirectional relations.

### 7.2.5 Interfaces, Aspects and Concurrency

Besides their functional characteristics, interfaces must describe additional information.

*Heuristic 55: Structure interfaces according to aspects.*

They enable the connection of aspect functionality which is distributed over BBs. The interface description of the BB should describe the aspect to which an interface belongs.

Similarly, the restrictions imposed by a concurrency design have to be described with an interface, that is, the assumptions about threading have to be made explicit in the interface description. Examples are information about re-entrance of interfaces, timing constraints and resource usage.

Besides the information which is described per interface of a BB, the overall descriptions of aspect designs and the concurrency design support the understanding of the role of the interfaces.

---

## 7.3 Component Models

Another facet of BBs is the underlying component model. The component model defines how a component is accessed. In particular, calling conventions for the use of the interfaces are defined by the component model.

Different component models are currently proposed. The most prominent examples are the Component Object Model (COM) of Microsoft, JavaBeans of Sun and CORBA of the OMG. Other component models have been developed for specific applications. To achieve wide applicability, these component models have to demonstrate their usefulness for different application domains, interoperability with existing (legacy) software and bridging to other component models. We shall neither explain nor compare these component models. Szyperski [Szy98] gives a good introduction to and comparison of COM, JavaBeans and CORBA.

The BBM just assumes that a component model is used. Nevertheless, as an example, we describe the component model of the tss system in appendix A.3.5.1. The tss component model is a dedicated component model developed in the mid-eighties to allow flexible design, implementation, loading [Fra97] and testing of BBs. The tss component model shows that components can be implemented with little overhead. This is an important consideration for embedded systems.

---

## 7.4 Layering

In section 4.2.1 layering of objects was introduced for separating functionality with different evolution characteristics. Layering for BBs extends and refines the notion of layering of objects. BBs make use of interface abstractions of other BBs. The abstraction serves as an infrastructure for the using BBs. Each BB provides a part of an infrastructure for one or more using BBs. The BBM requires the infrastructure to be independent of the supported BBs. To achieve this independence, mutual relations are transformed in unidirectional relations (see also section 7.4.2).

The BBM uses a *layer* as a clustering of functionality which has unidirectional syntactical relations only, that is, a layer depends on lower layers, and higher layers may depend on it. Parnas identifies layering, besides information hiding, as a desirable property of a system structure [Par72].

*Heuristic 56: Use layering for BBs on two levels. Subsystems, which are collections of BBs, are layered. These layers are based on the classification of layers of domain objects done during object design.*

*Heuristic 57: Individual BBs within subsystems are also layered in relation to other BBs.*

Layering of subsystems will be discussed in this section. Section 7.5 on generic and specific functionality describes layering of individual BBs.

### 7.4.1 Layering Principles

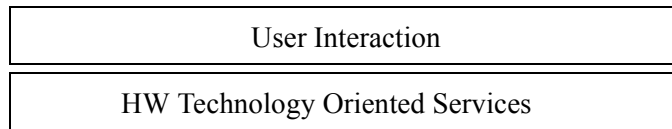
Structuring of functionality in an infrastructure and its using applications can be based on several principles: layers can have a different scope of visibility, layers can be used conceptually or even in implementation, and layers can provide dif-

ferent degrees of completeness. The following subsections introduce the different kinds of layering.

### Application versus Technology Layering

*Heuristic 58: A common principle for the layering of software is to separate hardware-technology-oriented functionality from application-oriented functionality.*

Hardware-technology-oriented functionality is seen as the base on which the application-oriented functionality is implemented (section 4.2.1). This approach is often taken in interactive systems. If the user interaction needs to be regularly adapted or extended, this layering approach (figure 37) provides flexibility since changing the application on the basis of a stable infrastructure restricts the impact of those changes. A variant of this principle is the separation of base technology from applica-



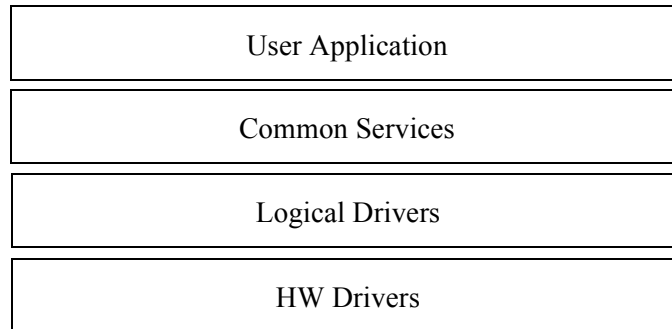
*Figure 37: User vs. HW Technology Layering*

tion functionality. [BGK\*99] and [RE99] use several layers to bridge from base technology to application-specific frameworks.

### Abstraction from HW

A related layering principle is one in which layers abstract from the concrete HW [Dij68]. Each layer provides abstractions for the higher layers. Layers are also called virtual machines. In figure 38 the lowest layer consists of the HW drivers. Logical drivers abstract HW specific attributes and constitute the second layer. Common

services are a collection of services used by many applications. The user applications, finally, constitute the highest layer.



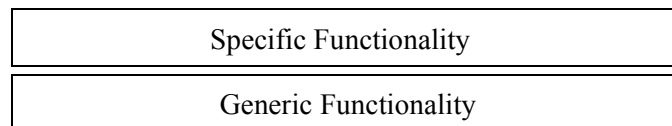
*Figure 38: Abstraction from HW*

*Heuristic 59: Construct layers as virtual machines for higher layers.*

### Generic versus Specific Functionality

*Heuristic 60: Another way of introducing layers is to distinguish between generic and specific functionality.*

Generic functionality is the infrastructure on which specific functionality is built (figure 39). This principle is used when generic middleware is separated from applica-



*Figure 39: Generic vs. Specific*

tion programs. More generally, interface abstractions can be encapsulated in a BB. The BB which actually provides the functionality of the interface BB and the BB which uses these interface abstractions to access the functionality are both located in a higher layer. The BB which contains the interface abstractions is more generic than the other ones and therefore resides in a lower layer.

### Example: tss Layering Principles

Within the tss system two principles for layering have been combined, notably abstraction from hardware and generic vs. specific functionality.

The rule that all relations are uni-directional creates some tension between the application of both principles. Generic services are often hardware-independent.

According to the *abstraction from hardware* principle, these services are located in higher layers, whereas according to the generic vs. specific functionality, these services are located in lower layers. For example, an interface between hardware drivers and the remainder of the system is more generic than the specific cases of the hardware drivers. Therefore, the BB containing the interface has to be lower than the specific hardware drivers. However, drivers are closer to the hardware itself. No general rule exists to reconcile these two principles for the actual layering of system functionality. A decision has to be taken for each function separately.

The tss central controller layers (subsystems) (see section 7.4.4) are based on the abstraction from hardware principle. But the system infrastructure generics (see below) are also placed in the lowest layer. Within each layer there is a micro-layering which separates generic from specific functionality (see section 7.5.3).

### 7.4.2 Incremental Layering

A layer is said to be *incremental* if the dependency relation of this layer satisfies the following criteria:

1. The import graph is acyclic, that is, the dependencies are unidirectional.
2. The semantics of layer  $L$  may depend only on the semantics of layers from which  $L$  imports.

The second criterion especially states that a BB must have well-defined meaning even if call-back methods have not (yet) been registered. This means the BB may neither depend on the presence of registered call-back methods nor on the semantics of these methods.

An incremental layer builds a platform for BBs of the higher layers, that is, it implements a virtual machine. The semantic independence of higher layers requires functional completeness of an incremental layer and the layers below it. We call this functional completeness of a layered subsystem its *platform property*.

Layering is already started during object design. The classification of functionality in layers is a step for getting meaningful functionality per layer from the application domain.

This platform property requires careful design. Techniques for defining a fine-grained platform for a single BB will be described in section 7.5.

The main reason for incremental layering is complexity management. A system with incremental layers can be integrated and tested layer by layer. [Dij68] and [HFC76] mentioned incremental testability as one of the key advantages of incremental layers.

This partial (in)dependence of layers is a compromise between total independence, which is not possible, and total dependence, which is not desirable.

A system is integrated according to its incremental layer structure. They are compiled and linked incrementally. If the target system is equipped with an incremental loader, the layers can be loaded incrementally. Each increment may be developed and tested as soon as enough information on the underlying system is available. System construction does not have to wait until all the increments are present. A small (core) system may be built from initial BBs. The system may be extended repeatedly with new increments until the entire system is ready. Incremental layering is a key concept of the BBM.

### **7.4.3 More Facets of Layering**

Besides the layering principles and the incremental nature of layers, other facets of layering are important. The following subsections discuss several alternative strategies for layering.

#### **7.4.3.1 Conceptual versus Strict Layering**

Conceptual layering is a design technique which models functionality in such a way that a particular layer is conceptually independent of other layers.

Strict layering is an implementation technique in which no compile or link time dependencies exist from a layer to a higher layer. This technique makes it possible to build each layer (or rather each increment) independently of the higher layers. Furthermore, a system can be loaded increment by increment.

Strict layering implies conceptual layering, but the reverse does not hold.

An example of conceptual layering only is a middleware layer which is independent of its using applications but has hardwired knowledge of these applications to be able to activate them. In the implementation such a middleware layer needs to be updated for new applications.

The BBM relies on strict layering of BBs. Strict layering is a precondition both for incrementally adding new BBs to the deployed system and for removing BBs from it.

#### **7.4.3.2 Opaque versus Transparent Layering**

There are two alternatives to the visibility of layers:

opaque layering: each layer hides the layers below it. A layer may only use functionality provided by the layer directly below it.

transparent layering: layers do not hide the lower layers. A layer may use all lower-level layers.

Other terms for opaque and transparent layering are strict layering (different from our usage of the term) and non-strict layering, respectively, proposed by [Szy98], or closed and open architecture, respectively, proposed by [RBP\*91].

Opaque layering requires each layer either to provide an additional abstraction for all the lower-layer services needed in higher layers or to add dummy interfaces. This eases the development of a particular layer, because a layer uses only interfaces of one layer below. Furthermore, restructuring of BBs in a lower layer is shielded from visibility of a higher layer by an opaque layer. On the other hand, updating functionality becomes more complex. Functionality from the lower layers used by a higher layer has to be presented to the higher layer in one form or another. Updates and successive testing are therefore required even if no functionality within a layer itself has been changed.

Transparent layering assumes that each layer provides only its own services. Only if an additional abstraction or an additional service is needed, an interface is introduced in a layer. In the case of transparent layering the implementors of a layer must know in which layer they will find the necessary interfaces. No maintenance effort is necessary in intermediate layers for making new or changed services of the lower layers available.

Opaque layering focuses on the using layer. The usage relations are defined to be exactly those of one layer with the layer immediately below it. Transparent layering focuses on the providing layer. The changes necessary after an update are restricted to the providing layer where a service is located and the layers of its direct users. The users may be from multiple higher layers. No effort in intermediate layers is necessary.

The BBM permits both techniques.

*Heuristic 61: The usage of transparent layers is favourable to the usage of opaque ones.*

*Heuristic 62: Opacity is used for layers that function as facades, such as abstraction layers for hardware, operating system or middleware.*

The facade pattern is described in ([GHJ\*94])

### 7.4.3.3 Partial versus Complete Layering

Sometimes systems are designed with partial layering only. For example, functionality that belongs to an application domain may be layered while functionality that provides computing resources is outside this layering. An example is given in figure 40.

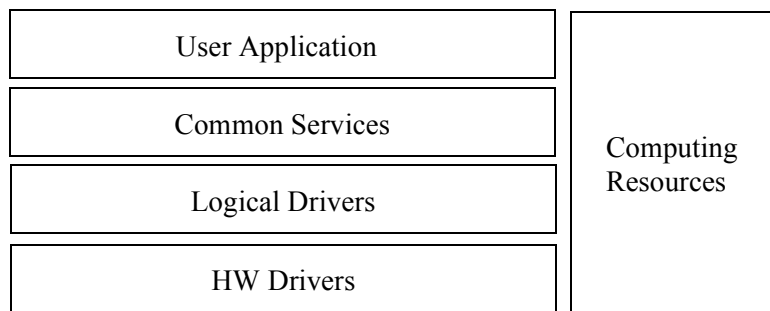


Figure 40: Partial Layering

All layers may make use of the computing resources and the computing resources may or may not access any of the layered functionality. The computing resources are a kind of general entity which must always be present and needs to be adapted to entities present in the layers.

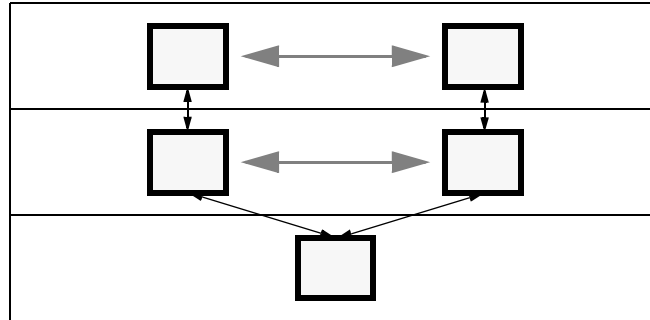
The BBM does not allow partial layering. Instead, if, in the example, partial layering were to be restricted so that only layered functionality would access the computing resources and, if necessary, register call-backs, it would be equivalent to a complete layering in which the computing resources are the lowest layer and are visible to all the other layers. In the last situation there would be no bidirectional dependencies.

### 7.4.3.4 Communication and Layers

*Heuristic 63: Use layers to structure communication in a system.*

Suppose a functional element in the system communicates with another one in the same layer. Communication happens, then, between elements of the same conceptual level. However, if the functional element may only have relations to functional elements of lower layers, the communication may not be implemented directly. Unidirectional dependencies require that such a communication relation be established indirectly. The lower layers are used for the implementation of the

peer-to-peer communication (figure 41). They transport messages from one peer to the other and notify the receiving peer.



*Figure 41: Indirect Peer-to-Peer Communication*

This model of peer-to-peer communication introduces semantic relations but no direct syntactic relations between the communicating parties. The ISO-OSI communication model [DZ83] made this communication structuring popular for computer-to-computer communication. Layered subsystems in a distributed system usually also communicate in this way. In BBM-based systems peer-to-peer communication will always be implemented via indirect communication.

For reasons of performance, an optimised version of layered peer-to-peer communication based on transparent layering may be introduced. Layered application entities make direct use of a lower-layer communication facility without intermediate packaging. Available knowledge about the sending environment, the communication channel and the receiving environment is used to minimise intermediate processing. Required functionality of intermediate layers may be implemented by the application layers taking advantage of the application knowledge. The communication within the tss system is based on such optimised peer-to-peer communication (section A.2.3).

#### **7.4.4 Example: tss Subsystems**

tss layered subsystems have been described in section 4.3. They comprise the extended operating system, equipment maintenance, logical resource management and service management. The subsystems have incremental semantics. The layering is strict, transparent and complete. The operating system provides a facade interface (opaque layering, heuristic 62).

---

## 7.5 Generic and Specific Functionality

A very important way of structuring software is by separating generic from specific functionality. Functionality of similar applications is analysed for diversity. This analysis is applied to domain objects, domain functions, algorithms, etc. [CHW98]. The common part is factored out and captured as an abstract concept. The common part is called *generic*, the diverse parts are called *specific*.

Separation of functionality into generic functionality and specific functionality is also present in object-oriented modelling and in design patterns [GHJ\*94] (see table 3).

Patterns	Generic	Specifics
OO modelling	supertype	subtypes
strategy pattern	data structure	algorithms
container	algorithm	data structures
template method pattern	skeleton algorithm	fill-in steps
bridge pattern	interface	implementations
observer pattern	event source	event listeners

Table 3: Examples of Separation of Generic and Specifics

The observer pattern which decouples event source from event listeners is an extreme form of generic and specific. The coupling between the generic part and potentially many specific listeners is reduced to events. Essentially, two independent entities are coupled via some events. Common to all examples is that there may be several specifics that are related to one generic.

The specific part is an extension of the generic part. Replacing functionality of the generic part by the specific is not intended. Functional units are built by combining a generic part and a specific part. Achieving stable generic parts for an application domain eases evolution.

The separation of generic and specific functionality is relevant for areas other than software too. Examples are system requirements, domain modelling, hardware, and development and customer documentation. Using the same separation in related areas makes correlating them easier. Domain terminology, for instance, may be extended with those generic concepts.

Note that there may be degrees of genericity. Certain pieces of functions are common to all members of the family, whereas others are common to only a few members of the family. In the latter case we still have generic functionality, but on a smaller scale.

The embodiment of the generic functionality represents a major part of the know-how of an application domain, cf. the domain kernel of Gomaa [Gom95].

*Heuristic 64: Separate common functionality from specific functionality.*

Analysis of functionality in terms of generic and specific functionality is the key concept in architecting a product family [Par76]. Frequently used terms are *commonality analysis* [CHW98] and *diversity analysis*. It is common experience that diversity analysis is more effective for identifying generic parts than commonality analysis [KMM96].

*Heuristic 65: Look for the diverse parts in similar functionality for different features.*

### 7.5.1 Generic and Specific BBs

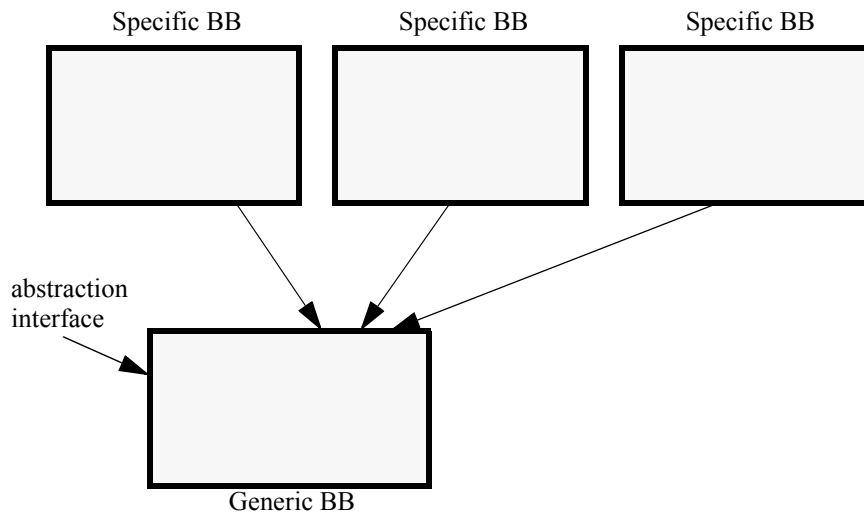
Generic and specific functionality is captured in different BBs. A generic BB contains generic functionality and a specific BB contains specific functionality. A *specific BB* extends the functionality of a generic BB. A *generic BB* is a component framework [Szy98] to which specific BBs, also called plug-ins, are added to extend its functionality. Figure 42 shows the basic inter-BB pattern generated by the BBM.

A generic BB defines standards for its specific BBs similar to a superclass setting standards for its subclasses. However, the specific BB extends the generic BB conservatively, whereas subclasses are often also allowed to overwrite parts of the superclass.

[FS97] call those frameworks black-box frameworks in contrast to object-oriented frameworks, which are called white-box frameworks.

More specifically, the generic BB defines interfaces for its specific BBs. If the interface between the generic BB and the specific BB is based on a domain object, the interface is stable because of its anchoring in the application domain. Usually, however, the interfaces between generic and specific BB depend on the implementation of the generic BB.

Implementation-dependent interfaces are called open implementation interfaces [Kic96] (see section 7.2.2). Call-back methods, also called hook methods in [FS97],



*Figure 42: Generic BB and Specific BBs*

decouple the generic behaviour of an application domain, encoded by the generic BB, from the specific instantiations needed for a particular application, as encoded by the specific BB.

General application processing steps are executed by the generic BB. The invocation of methods of specific BBs customises and extends this generic behaviour to one for the specific application domain. Inversion of control [FS97] means that the generic BB (rather than each specific BB) has control and gives it to specific BBs to do certain actions.

*Heuristic 66: Use inversion of control for designing the functionality of a generic BBs.*

For instance, external event handling is effected in the generic BB, which decides which methods of a specific BB to call in response to those events.

Modelling a generic service requires careful analysis. To become stable, a generic BB usually undergoes several redesigns [Sch97].

*Heuristic 67: A generic BB is stable if new specific functionalities may be based on the generic BB without changing it.*

Of course, generic BBs may be adapted and/or added during the lifetime of the product family. A generic BB embodies the similarities of certain functions. Therefore, changes in a good generic BB are rarely necessary. However, changes

in a generic BB almost always have consequences for all the corresponding specific BBs.

Each specific BB uses the functionality provided by the generic in its own way. Contrary to superclasses, a BB hides its internal structure. The specific BBs do not have direct access to the internal details of the generics. This means that the specific BBs may use only the interfaces provided by the generic to access the generic functionality.

### 7.5.2 Generic and Specific BB Roles

In practical situations a BB will often be both, generic and specific. It is therefore necessary to use the term generic and specific in relation to some functionality. The terms generic role and specific role of a BB are used to denote that a BB contains generic or specific functionality. A BB can have several generic and / or specific roles.

Most BBs have multiple specific roles. This is similar to a class structure with multiple inheritance in object-oriented modelling. The difference with multiple inheritance is that generic BBs have explicit interfaces and the generic-specific relation is not recursive.

Note that we use the term generic BB without reference to its particular role if it is clear from the context what that role is. Thus, a BB which has at least one generic role may be called a generic BB for short.

### 7.5.3 Generics and Layering

The BBM uses layering to structure overall functionality (section 7.4). But layering is also used to structure inter-BB relations. Since a specific BB is an extension of a generic one, the specific BB is dependent on the generic BB. The specific BB imports an interface of the generic BB.

Besides syntactical independence of its specific BBs, a generic BB is required to be semantically independent too. Layers have an incremental character (section 7.4.2), but so does a generic BB with respect to its specifics. Semantic independence means that a generic BB without registered call-backs to specific functions has to perform basic functions without leading to any error (section 11.7) to enable incremental testing.

A generic BB is a kind of infrastructure on top of which specific BBs are located. The generic BB is located in a layer below the specific BBs.

This may seem counter-intuitive from the point of view of object-oriented modelling in which super classes are usually drawn above subclasses. In contrast, the BBM uses the notion of an extensible platform upon which further BBs are constructed (see figure 43).

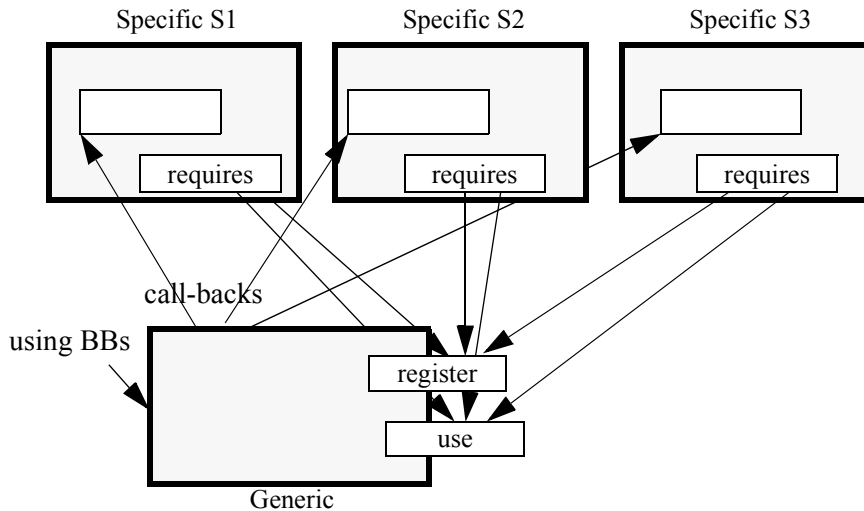


Figure 43: Generic and Specifics with Interfaces

For instance an operating system can to a great extent be modelled to consist of generic services which are used by almost all the application.

Specific BBs register call-backs with their generic BBs (section 7.2.4). In fact, BBs with generic roles are the only BBs that provide a registration interface. The BBs with specific roles are the only BBs that use the registration interface and have a call-back interface. In figure 43 a generic BB is depicted with its specifics, together with the registration and use interfaces of the generic. The arrows indicate method invocations.

#### 7.5.4 Classification of Generic BBs

Generic BBs can be classified according to patterns. The idea is that the very general concept of separation between generic and specific functionality leads to a number of domain-specific types of generics. We describe four different types as part of the core BBM. The list is not exhaustive but can be extended with new ones by specialised versions of the BBM.

### Abstraction Generic

An abstraction generic is the basic form of a generic (see figure 44).

*Heuristic 68: Use an abstraction generic to implement an abstract concept which is to be extended by specific BBs.*

The distribution of functionality may range from a very thin generic to a very thick one. A thin generic means that the generic implements almost exclusively an interface and the rest of the functionality resides in the specifics. A thick generic means that the specifics add only minor variant data or functions.

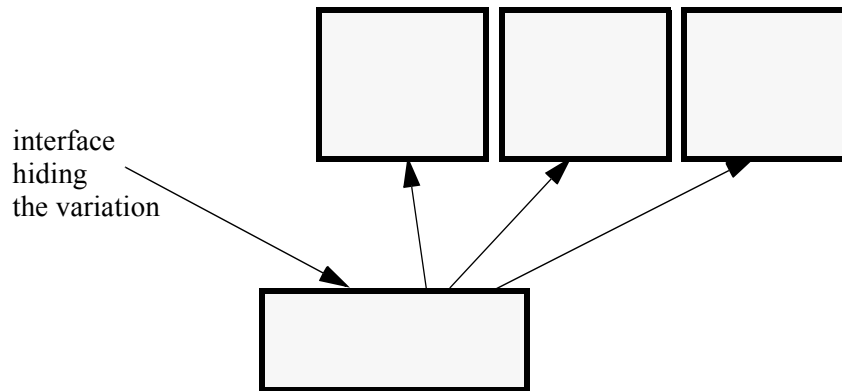


Figure 44: Abstraction Generic

An example of a thick generic is one which implements a template method pattern [GHJ\*94] in which the factor-out steps are small.

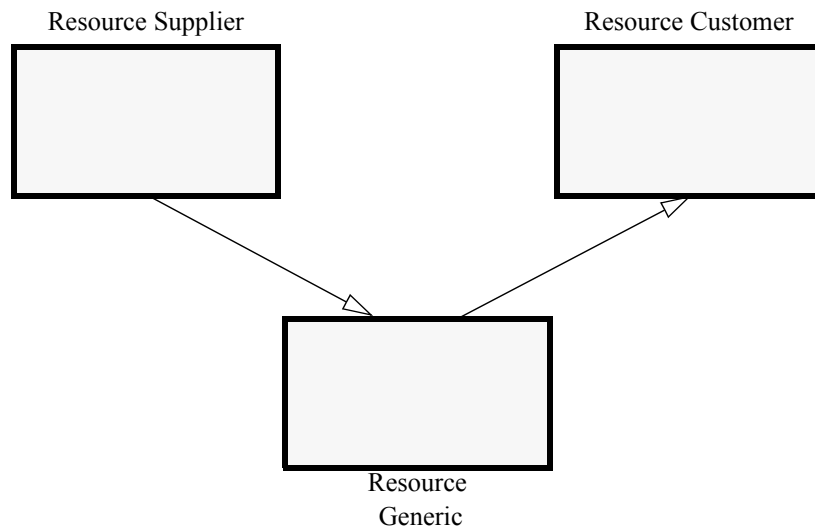
An example of a thin generic is a generic which only forwards calls to its specifics. Ethos [Szy92a] uses *directory objects* to access basic services in an operating system which should be extensible. A file system, for instance, is not directly accessed to create a new file but the access goes via a directory object. A file system can be updated in a running system without its clients noticing. A somewhat thicker generic would be a file handling generic which provides the interface to the client, does general file administration but has the handling of different file types, such as sequential or indexed files, factored out in specifics.

All variations between the extremely thin and the extremely thick ones are possible. Examples given by Kiczales [Kic96] such as an application-specific paging strategy that can be coupled to a generic memory handler would probably rank between the extremes.

### Connectable Resource Generic

Connectable resource generics are defined to manage connectable resources which are supplied via HW boards. Changing the configuration of those boards also changes the number of connectable resources.

*Heuristic 69: Use a connectable resource generic to manage connectable resources which are supplied by HW boards.*



*Figure 45: Connectable Resource Generic and Resource Flow*

Examples of tss are boards which provide a number of connectors for other boards, or timeslots to be used by some application. These connectors or timeslots are resources supporting system configurability. The generic has the task to administer these resources.

Figure 45 shows a connectable resource generic with one supplier and one customer. Note that the arrows in the figure show the resource provisioning from supplier to customer and not the dependency relation. The dependency relation is from the specifics, that is the resource supplier and the resource customer, to the generic.

The connectable resource generic handles the resource abstraction and has two classes of interfaces. The first class is for supplying and allocating resources, the second for communication between resource supplier and customer. The connectable resource generic has an allocation table which relates instances of sup-

pliers to instances of customers. The communication interfaces have one of the following characteristics:

- a customer instance signals some event to its supplier instance,
- a supplier instance notifies all the related customer instances (broadcast) about some event, and
- a supplier instance notifies a specific customer instance about some event on the basis of some selection criteria.

Suppliers and customers of a connectable resource generic may evolve independently. A connectable resource generic bears some resemblance to a bottleneck interface [Szy98]. A bottleneck interface couples two independently extensible abstractions and is itself not extensible. A connectable resource generic also allows independent extension of suppliers and customers via their respective specific BBs.

### **System Infrastructure Generic**

System infrastructure generics provide an operating infrastructure for application BBs. System infrastructure denotes a collection of services which is part of the operating system, standard UI functionality, or generic and domain-specific middleware.

*Heuristic 70: Design a system infrastructure generic for functionality which provides an operating infrastructure for almost all application BBs.*

System infrastructure generics (SIG) are a means for encapsulating these services. The characteristic of a SIG is that all application BBs are potential specifics of it. The operating infrastructure provides the basic framework whereas individual BBs only provide increments.

SIGs may administer system resources such as processor time and memory pools, and standardise system functionality such as the user interface, a recovery strategy or error handling.

*Heuristic 71: System Infrastructure Generics must provide interfaces for application BBs for indicating their resource requirements.*

The SIG will either have service interfaces to provide data directly or registration interfaces to register call-backs which the SIG uses to retrieve the data from the application BBs.

Generic parts of aspects (chapter 5) which can share a common implementation are factored out into SIGs. This leads to an instantiatable aspect infrastructure.

A SIG for recovery handling is such an example. A generic report handler may be used by the configuration management aspect and the fault management aspect. An internal message handler may be used by more aspects.

Since these generics offer very generic functionality, they are located low in the system hierarchy of layers, i.e. they can be seen as part of an extended operating system. Layering them low in the system is an example of the *generic-versus-specific functionality* layering principle. An application BB has a specific role for many of the SIGs. Figure 46 shows a BB together with three SIGs.

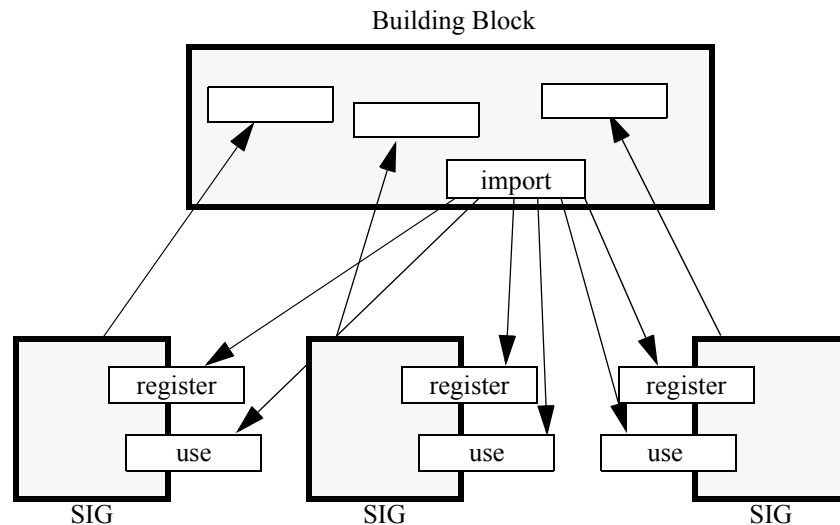


Figure 46: System Infrastructure Generics

Standardisation of SIGs may go so far that specific functionality is reduced to specific data instances. All the relevant algorithms reside in the SIG and the specific data instances reside in the specific parts. In such a case BBs do not access these data directly, but through a SIG.

An example of such a SIG is one which handles exceptions. The SIG defines standardised functionality such as the handling of severity of exceptions, writing to a log file and communication to a user interface, whereas the actual severity, output formats and reporting texts are located in the application BB.

Using a SIG to implement system functions such as exception handling is an alternative to a coded case statement with each case alternative representing a specific instance of an exception. Maintaining lists of case alternatives for an evolving system leads to frequent changes in the source code of the exception handler. This is a potential source of errors. In contrast to this a SIG is not changed. The list of different exceptions in a specific product is automatically built by the configuration of selected BBs in the product. Consider the addition of a BB to the system. No recompilation, relinking or reloading of, for instance, exception handling is necessary. Instead, the BBs define their own exceptions and make them known to exception handling. After adding the Building Block, the system will have an adapted list of exceptions. There are, hence, no separate configuration files or common include files in the system.

The use of the SIG can reduce the code of the specific BBs considerably. Furthermore, this standardisation allows automatic code generation. Specific functionality of a SIG may be reduced to parametrised data structures. The collection of specific BBs together with the SIG implement a configurable table in the running system whereby all access methods are implemented in the SIG itself. The parts that extend the table can be generated automatically.

The tss system used a specific database-based tool (section 11.8) for data generation. Because of their relevance for the whole system, data can easily be reviewed and changed without entering an implementation phase. Part of the production of a BB is the code generation for all tool-supported SIGs. An implementor of a BB may not even be aware that some of the SIG-specific data of his BB form part of the Building Block.

Often when diagrams with relations between BBs are drawn, the relations to SIGs are not shown because all the BBs are specifics of at least some of the SIGs.

### **Layer Access Generic**

Layer access generics provide access to a layer's functionality. They restrict the visibility of the structure within a layer for higher layers, i.e. references are channelled through a number of specifically designed indirections. The advantages of layer access generics are, first, that higher-layer BBs only have to know about layer access generics of the lower layers, and, second, that they provide for configurability of BBs in lower layers without any need to update BBs in higher layers.

*Heuristic 72: Design a layer access generic to restrict the visibility of the structure of a layer for higher layers.*

A specific pattern is used to achieve this purpose. In contrast to normal interfaces, which are often conceived as being on top of a BB, layer access generics are located below the BBs they provide access to. The BB register themselves with the layer access generic (see section 7.2.4). Figure 47 shows import rela-

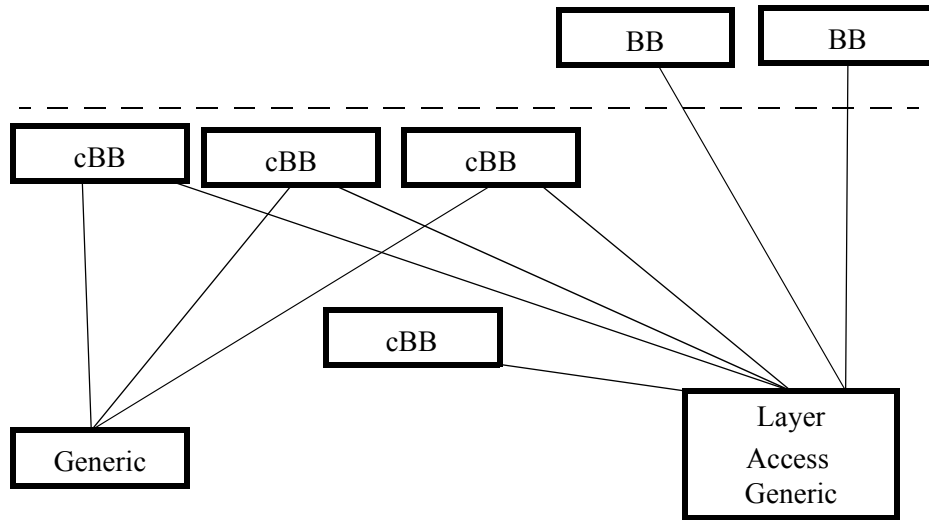


Figure 47: Layer Access Generic

tions of BBs in a lower and a higher layer. Configurable BBs (cBB) of lower layers have no provides interface. They can be replaced without affecting higher layers. Also, changes within the lower layer are less likely to propagate to higher layers. Layer access generics are brokers for layer functionality.

### Example: tss Generics

The tss system has around 50 generics. We shall mention some of them below without going into much detail.

Examples of Abstraction Generics from the OS subsystem include file handling and I/O device handling. File handling includes the operations open, close, write and read. Specifics of file handling have specialised handlers for different file types such as sequential or indexed files. I/O device handling includes the operations mount, dismount, assign and deassign. I/O device handling has separate specifics for handling different device types such as PCs, printers and disks.

Examples of abstraction generics from the equipment management subsystem include two for handling two classes of peripheral cards. Peripheral card handling recovers and supervises peripheral cards in the central controller. The protocols and data structures are standardised for all cards and they belong to the generic part. Spe-

cific parts are reduced to peripheral card-specific data structures and parameters. For each peripheral card type, there is a separate specific part.

Examples of abstraction generics from the logical resource management subsystem include generics for administering generic classes of analog and digital subscriber lines and trunk lines. Digit analysis and automatic audible responses are other abstraction generics. Digit analysis has as its input digit strings and has specifics for different destination types such as analog or digital subscribers or PABXs. Audible response generic has specifics for single and periodic announcements.

Examples of abstraction generics in the service management subsystem include the call-handling generic which implements basic call handling such as call set-up, call phase and call release, which are valid for (almost) all call types. It can be extended to different signalling handlers. Another generic handles the generic call forwarding feature. Specifics are for call forwarding on absent, on busy or on other conditions.

The equipment management subsystem contains four connectable resource generics for administering hardware (-related) resources. Two generics administer internal bus slots, the so-called time slot (TS) generic and the so-called universal peripheral slot (UPS) generic. External connections are administered by the so-called circuit generic, which is based on a 64kbit communication channel abstraction. A fourth connectable resource generic establishes pools of dynamically allocatable communication channels for a group of subscribers, so-called concentration groups. Figure 48 shows how Equipment Maintenance structuring is based on mirroring of the peripheral hardware structure to a central controller BB structure.

Three generics of equipment management also have the role of a layer access generic. The circuit generic, the TS generic and the concentration group generic function as a layer interface to the higher layers LRM and SM.

The OS contains a number of SIGs. They are persistent data handling, process handling, memory management, user interface presentation data handling, recovery handling and exception handling.

---

## 7.6 Interfaces Revisited

After the sections on layers (section 7.4) and generic BBs (section 7.5), we shall take another look at interfaces. In section 7.2 we made a distinction between provides and requires interfaces. From the discussion of incremental layering it is clear that a BB may depend only on BBs in lower layers. As a consequence a

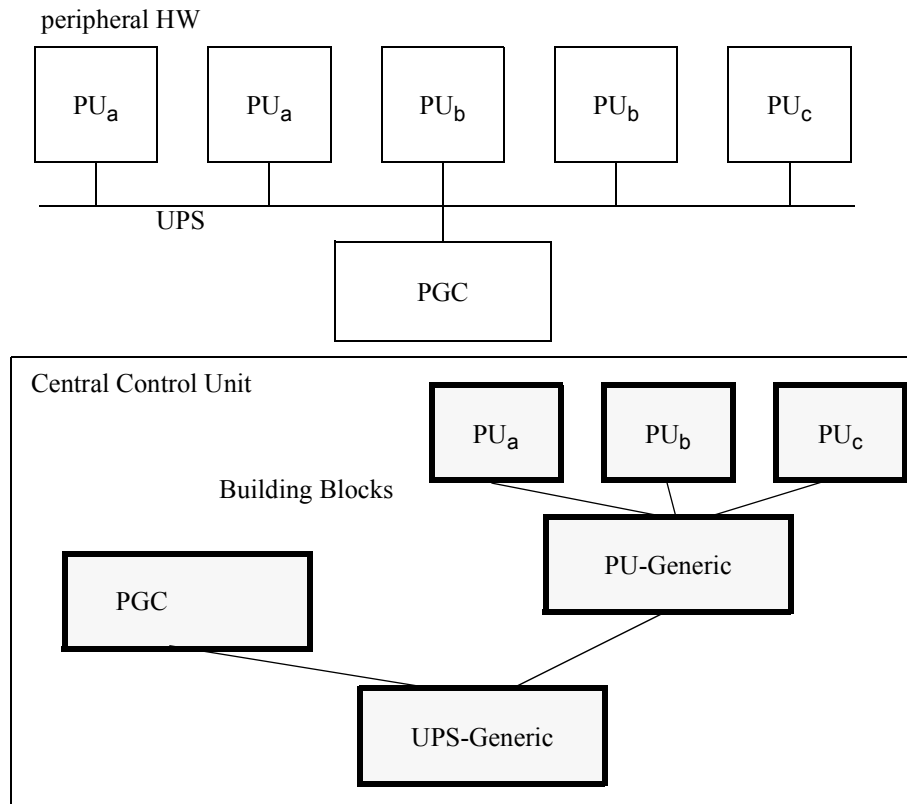


Figure 48: System Structure with HW Mirroring in EM

requires interface relates to BBs in lower layers only and a provides interface relates to higher layers only.

There are, however, two types of interfaces, call-back interfaces and so-called first-access interfaces (see section A.3.3.4), of which it is not so clear whether they are provides or requires interfaces. One may say that a BB *requires* certain processing to be done by a calling BB which knows the precise content of a data structure or that a BB *provides* a first-access interface to be used by the recovery manager. Note that in both cases the activation goes from the lower BB to the higher BB. In the first case, a using BB has registered call-back methods, whereas in the second case a standardised entry in a BB header allows the recovery manager to activate initialisation methods.

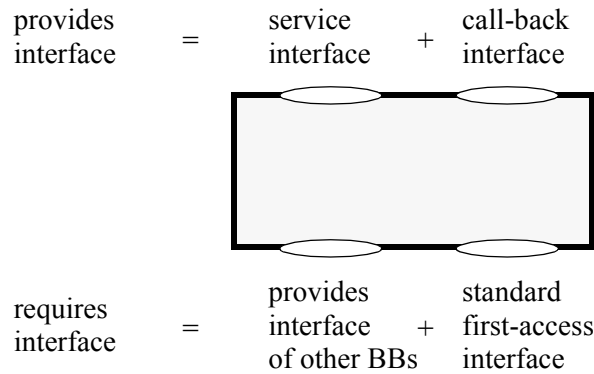


Figure 49: BBM Interfaces

However, contrary to the use of *require* and *provide* suggested above, call-back interfaces are part of the provides interface and the first-access interface is part of the requires interface, because the incrementality of a BBM-based system is the leading concern for the definition of provides and requires interfaces. Even if a BB has only limited processing capabilities in a certain situation, the BB may not depend on the presence of a call-back method. A BB always has to implement some default behaviour if no call-back method is present. Call-back interfaces are in a sense optional.

A similar argument as for call-back methods applies to the first-access methods. A BB requires the recovery manager to call the BB's initialisation methods to be initialised. Therefore, first-access methods are part of the requires interface of the BB. Figure 49 shows the provides interface at the top and the requires interface at the bottom of a BB to indicate how interfaces support incrementality.

---

## 7.7 Grouping of BBs

Hierarchical components are components which consist of components themselves. This is a useful means for abstraction. The BBM does not support hierarchical BBs. A BB is a black box, that is, its internal design is not visible, and it is deployed as one unit. Recursive deployment does not seem to be a useful concept.

However, collections of BBs can be grouped to a *white box component*, that is, a component that does not hide its internals. Examples of such groupings are layers (section 7.4), a feature set (see section 8.3), that is, a set of BBs which implements a certain feature, or *managed object* related BBs (controlling and controlled BB) (see section 10.2.1). The BBs of a feature set may belong to different subsystems.

*Heuristic 73: Apart from a BB itself, the collection of BBs of a white box component can be packaged as unit of deployment.*

Packaging of BBs into larger deployment sets is described in section 7.9. In a running system only deployment units are recognisable.

### Substitutability

Two BBs are substitutable if they have the same provides and requires interfaces and if they are fulfilling the same role in the architecture.

However, there may be cases where technical substitutability as defined above is not enough. If both BBs were planned and one of them is not just an improvement of the other, difference in features is involved (see section 8.3.4). Depending on the feature set of the product, one of them will be chosen.

*Heuristic 74: If two substitutable BBs are to be present in the same product, the BBM requires that there must also be some generic which switches between the two.*

---

## 7.8 Architectural Skeleton

An architectural skeleton is the result of structuring the software in incremental layers which themselves consist of generic and specific BBs. An **architectural skeleton** is defined as the collection of generic BBs in the different layers. The specific BBs that implement the wanted functionality constitute the "meat". The skeleton, however, provides a structured *platform* at different levels, which supports the rapid creation of products. It is an incrementally layered *component framework*. The skeleton implements the stable structure in a domain. It is a generic domain architecture. Abstracting from internal structure, an architectural skeleton is also called a product platform [ML97].

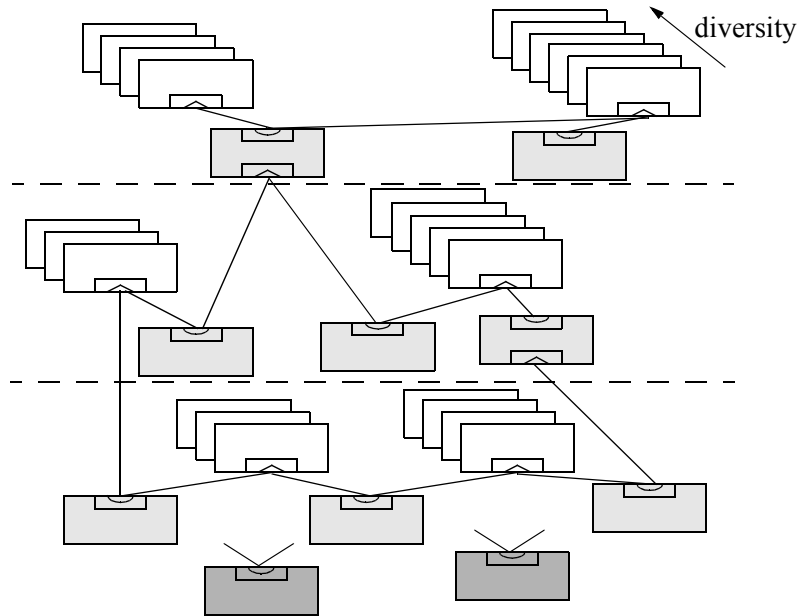


Figure 50: Architectural Skeleton

Figure 50 gives an example of an architectural skeleton together with the sets of specific BBs. The architectural skeleton consists of the shaded BBs only. They are represented in two types of generic BBs. First, there are generics in each of the three layers of the system. Secondly, there are two SIGs in the lowest layer (transparent layering). Note that the relations of the system infrastructure generic with their specifics are not indicated, because potentially all the Building Blocks are specifics of a system infrastructure generic.

Another important point in this example is that the interfaces of the subsystems (or layers) are represented by generics only. These so-called layer access generics are brokers of the layer functionality towards higher layers. They allow configurability within a layer without affecting the higher layers.

Note that the system is incremental in two ways. First, the system can be built subsystem by subsystem. Secondly, BBs can be added in each subsystem.

*Heuristic 75: Choose generic BBs in such a way that stability of architectural skeleton increases.*

Another important point is that BBM-based systems are always open. A new BB may always be added to generics using them as its relative infrastructure. Such a new BB has a specific role but may also have generic roles. Thus, a completed system can always be extended if it provides interfaces to access its functionality. This can be realised either by removing a BB which has no provides interface and substituting it by one that has a provides interface, or by using the provides interface of a BB and adding new functionality.

A self-describing component is a component that may be introduced into a system without requiring adaptations in other parts in the system.

*Heuristic 76: Make a BB is a self-describing component by letting it communicate its characteristics such as its resource requirements to the infrastructure.*

In particular, the infrastructure should not be adapted because of requirements imposed by a component. In this way the exchange or adaptation of a component does not have any direct consequences for the rest of the system. Aspect completeness (section 5.6.1) is a way to make BBs self-describing.

A system integration approach has a low integration complexity if it can be based on lists of components without requiring adaptations in these components for the sake of integration. In particular, if the infrastructure provides all facilities to make components self-describing, the complexity of resource management decreases considerably because it is separated into a requesting and an administering part. In particular, the requirements of resources are part of the components themselves. The BBM uses such an integration approach (see section 11.7).

---

## 7.9 Deployability Design

Deployability design is about possible deployment scenarios of the products. Address spaces (section 6.1) are also used within deployability design.

The input for deployment scenarios may come from the customer, from product management or from technology assessment requiring a certain HW partitioning. Geographic distribution, if required, will often come directly from the customer.

BBs are the minimal units of deployment. However, for commercial or other reasons, BBs may be packaged to larger sets. Delivery will be in statically or dynamically linked code libraries. Since the library structure is important for the flexibility of software upgrading care should be taken when BBs are packaged.

As described earlier, address spaces are boundaries for objects, BBs and threads (see section 6.1). Address spaces may also be used for fault containment and recovery (see below). Furthermore, the flexibility to move BBs across HW instances is based on the presence of BBs implementing all requires interfaces of the moved BBs at the target location.

Deployability design consists of the following steps:

#### **Determining fault containment units**

Fault containment units are chosen to confine the consequences of failures and to be able to do a recovery from the failure. Fault containment units should not cross HW boundaries. Design has to take care that a fault containment unit can always synchronise with the rest of the system.

*Heuristic 77: Select a set of objects in such a way that the set may be independently recoverable when an error occurs.*

This may require refactoring in object design.

#### **Determining possible deployment scenarios**

Deployment scenarios have to take the following factors into account:

different geographic locations,  
HW partitioning and  
different SW address spaces.

*Heuristic 78: Align BB-, thread-, fault containment unit- boundaries to HW instances*

This may lead to refactoring in the respective design task.

### **Packaging BBs to deployment sets**

BBs are deployment units. They may be packaged for commercial or practical deployment reasons into deployment sets. Product management may package BB, taking into account:

sellable units (basic product may be one unit), that is, a commercial component may be different from a technical component (visibility to the customer), and

independent evolvability.

*Heuristic 79: Package BBs to deployment sets such that independent selling and evolution remains possible.*

Deployment sets consist of executables and libraries.

White-box collections of BBs such as subsystems or feature sets (see section 7.7) are examples of useful deployment sets.

### **Generating data files**

Data files are used amongst others for initial or default configuration values, persistent data, international strings, font libraries, sounds, tones and announcements.

The creation of a tss product is described in section A.4.

### **Heuristics Summary of BB Design**

*Heuristic 44: Cluster objects into BBs such that coupling of objects across BB borders is low and cohesion of objects within a BB is high.*

*Heuristic 45: Cluster objects into a BB which represent a feature.*

*Heuristic 46: Cluster objects into different BBs which belong to independently evolvable parts.*

*Heuristic 47: Cluster objects into BBs such that a BB can be used as a work allocation units for 1 or 2 persons.*

*Heuristic 48: If the variation point lies inside a BB, refactor the BB such that the variation point lies at the border of a BB.*

- Heuristic 49: Factor out functionality which is present in several BBs in a separate BB.*
- Heuristic 50: Take as main criterion stability under evolution, that is, an interface should be such that it can serve for those implementations and those usages which are likely to happen.*
- Heuristic 51: Factor generic implementation parts which are used by several specific parts into a generic BB.*
- Heuristic 52: Take the implementation of common aspect functionality as a candidate for a SIG.*
- Heuristic 53: Resolve mutual dependence between BB A and BB B in the follow way: if A is expected to be more stable than B, then make B depend on A; and vice versa if the communication between A and B is expected to be the most stable part, factor the communication out into a new BB and let both, A and B, depend on it.*
- Heuristic 54: In the case of embedded systems, use importing of interfaces at compile time if needed for performance reasons. Otherwise use dynamic exploration of interfaces for more flexibility.*
- Heuristic 55: Structure interfaces according to aspects.*
- Heuristic 56: Use layering for BBs on two levels. Subsystems, which are collections of BBs, are layered. These layers are based on the classification of layers of domain objects done during object design.*
- Heuristic 57: Individual BBs within subsystems are also layered in relation to other BBs.*
- Heuristic 58: A common principle for the layering of software is to separate hardware-technology-oriented functionality from application-oriented functionality.*
- Heuristic 59: Construct layers as virtual machines for higher layers.*
- Heuristic 60: Another way of introducing layers is to distinguish between generic and specific functionality.*
- Heuristic 61: The usage of transparent layers is favourable to the usage of opaque ones.*

- Heuristic 62: Opacity is used for layers that function as facades, such as abstraction layers for hardware, operating system or middleware.*
- Heuristic 63: Use layers to structure communication in a system.*
- Heuristic 64: Separate common functionality from specific functionality.*
- Heuristic 65: Look for the diverse parts in similar functionality for different features.*
- Heuristic 66: Use inversion of control for designing the functionality of a generic BBs.*
- Heuristic 67: A generic BB is stable if new specific functionalities may be based on the generic BB without changing it.*
- Heuristic 68: Use an abstraction generic to implement an abstract concept which is to be extended by specific BBs.*
- Heuristic 69: Use a connectable resource generic to manage connectable resources which are supplied by HW boards.*
- Heuristic 70: Design a system infrastructure generic for functionality which provides an operating infrastructure for almost all application BBs.*
- Heuristic 71: System Infrastructure Generics must provide interfaces for application BBs for indicating their resource requirements.*
- Heuristic 72: Design a layer access generic to restrict the visibility of the structure of a layer for higher layers.*
- Heuristic 73: Apart from a BB itself, the collection of BBs of a white box component can be packaged as unit of deployment.*
- Heuristic 74: If two substitutable BBs are to be present in the same product, the BBM requires that there must also be some generic which switches between the two.*
- Heuristic 75: Choose generic BBs in such a way that stability of architectural skeleton increases.*
- Heuristic 76: Make a BB is a self-describing component by letting it communicate its characteristics such as its resource requirements to the infrastructure.*

**Heuristics Overview of Deployability Design**

*Heuristic 77: Select a set of objects in such a way that the set may be independently recoverable when an error occurs.*

*Heuristic 78: Align BB-, thread-, fault containment unit- boundaries to HW instances*

*Heuristic 79: Package BBs to deployment sets such that independent selling and evolution remains possible.*



---

## 8 Family Architecture

The goal of the BBM is to support the architecting of product families. This chapter describes how the BBM concepts are used to achieve this goal. The first section introduces commercial product features in the scope of product diversity. The second section summarises means for implementing diversity. The third section discusses the concept of a product family architecture. The last section places the concept of a product family architecture in the context of managing the evolution of products.

---

### 8.1 Product Diversity and Features

Nowadays, electronic products usually are developed as sets of related products aimed at covering a segment of a specific market. The commonality between these products is complemented by diverse parts, which makes them different.

In describing a product and its advantages to a prospective customer, one usually refers to its features. A *feature* is a characteristic capability or attribute of a product which is believed to be of importance to the customer, or alternatively important for distinguishing it from a competitive product.

A set of related products which share a common set of features is also called a product line or *product family*. The development of a product family is a strategic decision for a specific market. We shall not discuss the business and technical prerequisites which justify the development of a product family (see [MSG96], [DKO\*97] and [ML97]).

Product diversity is a term broader than what is covered by features. For example, if the only difference between two cars were to be their colour, one would not speak about different features of the cars. However, what a product precisely is, and what degree of difference is necessary to speak about a different product will depend on the market. It is not a technical question. Diversity can

exist between different products as well as between different instances of one product. We shall use the term feature to define functionality of a product and not to distinguish different instances of a product.

Note that the creation of feature descriptions and their dependencies is not part of the BBM but is done as part of the commercial product design (section 2.6.3). The BBM uses the feature descriptions and feature dependencies as input for the creation of a product family architecture.

### 8.1.1 Feature Description

A feature has been loosely defined above as a capability of a product. The terminology is borrowed from the application domain model (see section 2.6.2). The domain model describes functionality not only of one product, but of a whole range of possible products. The precise form of a feature description therefore depends on the domain model. If the domain model is complete, a feature description can be restricted to references to the domain model. Otherwise feature descriptions complement descriptions of the domain model.

### 8.1.2 Feature Relations

Features are not described in isolation. Features will usually make reference to related features. A feature relation graph describes relations between features. The relations are:

*Dependence:* One feature may be dependent on the presence of another feature. Application features are usually dependent on infrastructure features. But within an application too, features may rely on other features. A special case of dependency is when two features are mutually dependent.

*Exclusion:* The exclusion relation describes features which may not be present together in one product.

*Selection from a set:* a number of features may form a set, only a certain maximum of which is allowed to be present in a product.

Further relations to indicate business considerations may be defined to aid the development people in making the right design decisions. This is not worked out any further.

*Heuristic 80:* Use a feature relation graph to describe relations between features.

### 8.1.3 Feature List

A collection of features is called a feature list. A feature list is used as a high-level description of a product. The BBM uses this approach to define products (see section 8.3).

The feature list can only describe an entire product if it is complete. A feature list is complete if all the required features are either directly mentioned in the feature list or are implied because of the feature relation.

#### **Example: Typical tss Customer Features**

As an example we describe typical features of the tss family. Different tss family members differ in type and number of the peripheral cards, different types of subscribers, service functions and network connections, etc. Additionally, products differ in kind of signalling handlers and call facilities (call forwarding on busy, automatic call answering if absent). After system delivery a customer may ask for more concentrated subscriber cards, additional service functions, updated signalling handlers when the signalling standard was updated, and/or additional call facilities. This illustrates that a product may evolve into a different one after delivery to the customer.

#### **Product Features and Feature Modelling**

Features as used by the BBM are different from features used for feature modelling. Feature modelling [CE00] is a variant of conceptual modelling and used throughout all development phases to describe domain and system functionality. Features are used as alternatives to objects in creating a domain model. The notion of an object as consisting of state and behaviour is artificial for describing, for instance, a table. Feature modelling, in contrast, claims to make more natural descriptions by just describing necessary and optional features of a table. Feature modelling concepts are described in feature diagrams consisting of a hierarchy of features and subfeatures.

A feature as used by the BBM is an independently sellable unit of functionality, that is, a feature is a unit from a commercial point of view. It is an outcome of the commercial design activity. Features have a dependency relation with the meaning that if a particular feature is needed also features it depends on are needed. In the BBM, features are not described in hierarchies. A product is described as a set of mandatory and optional features. The mandatory set determines the distinguishing characteristic of a particular product while the optional features allow for variation. Besides the dependence between features, the implementation of the features may introduce additional dependencies between BBs which results in certain features being a must with certain products. Furthermore, late changes in the feature relations by product management may lead to an implementation which restricts product configurability beyond feature dependence.

An example of tss was the request for low-end switching systems without HW redundancy. This request could not be met without major redesigns. The problem was not so much the SW where visibility of redundant HW was easily localised but the design of the HW itself.

---

## 8.2 Implementation Means for Diversity

The BBM supports the creation of a product family on the architectural level. This means that BBs are designed so that each product can be configured from a list of BBs. Several concepts of the BBM are used to model architectural diversity. Layering of BBs is used to limit the effects of changes; for example, hardware-related changes are separated from application changes (see section 7.4.1). Generic BBs implement common functionality with variation points for specific BBs. Specific BBs connect themselves at variation points [DW99] to a framework via call-backs (see section 7.5.1). Feature orientation, to be described below, relates features of a product to BBs [MHM98]. Product variability is obtained through selection of BBs. From a product family point of view, the diversity between different products is most important.

Diversity in a product family can also be implemented in a non-architectural manner. In such a case we speak of a product family without a product family architecture.

Examples of non-architectural means for product diversity are to be found in four different areas:

Control-flow-driven diversity concepts of programming languages for changing a program's control flow such as IF or CASE statements.

Data-driven diversity such as property values, i.e. the diversity occurs via some initialised constant or dynamic value; configuration database, i.e. a systematic implementation of property values; and configuration bits, i.e. a bit list represents different functions which are usable or not.

Development-environment-supported diversity, such as conditional compilation and conditional linking.

Customer-enabled diversity is a means for customisation such as setting specific initial or default values, enabling or disabling certain functions. It is a variant of data-driven diversity.

Relying only on non-architectural diversity for implementing a product family leads to a situation where the product family becomes a monolith. Maintenance for specific products always affects all products.

The tss system uses, besides programming language constructs, a configuration database to store data such as the hardware configuration, the logical function configuration, the subscriber and network configuration, and various other parameter values. Each table of the database belongs to a BB. A BB may have zero, one or more tables. These configuration data may be different per product installation.

The tss system does not use conditional compilation because conditions are difficult to maintain and the programs are difficult to understand. Configuration bits are not used either, because the products are configured to a minimum, i.e. unnecessary functionality is omitted.

*Heuristic 81: Design a system infrastructure generic to handle data-driven diversity.*

---

## 8.3 Product Family Architecture

A product family architecture is an architecture which covers a product family through architectural diversity. A product is obtained through selection of software components.

### 8.3.1 Feature Orientation

A product family is not developed in one step. Evolution and extension are the most important development tasks after the initial development of the family. A family's architecture is one of its critical success factors. A good architecture should allow to develop and update independently those parts which are independently present in products. Those independent parts are encapsulated in different BBs [Par76]. The BBs obtained in this way are called feature BBs. That is, the identification of BBs is based on features [MHM98]. The relation between customer features and feature BBs should be simple.

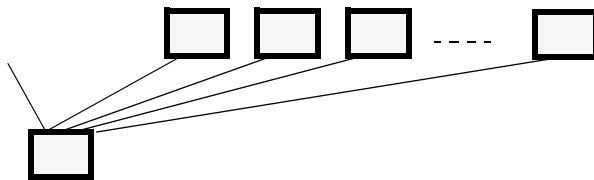
A feature that is sold to a customer does not necessarily have a direct relation with SW, however, many will require SW functionality. Only those that require SW functionality are considered here.

Feature orientation means that a system's architecture is used primarily to support the system's evolution. Evolution and extension which can be limited to one

or a few BBs are less costly to develop than those which are spread over large parts of the system. In the most simple case a change is realised by exchanging a BB, and an extension by introducing an additional BB; in [Ben97] this is called "hard component software". Such an approach requires a good understanding of the nature of a system and its evolution. Often this understanding will itself evolve. Feedback from first products improves the development of the product family.

Feature orientation requires a good, stable infrastructure and a BB granularity which matches features. Feature orientation may be difficult to realise in an initial version of a product. However, as a system and its functionality are better understood, it becomes easier to take evolution into account.

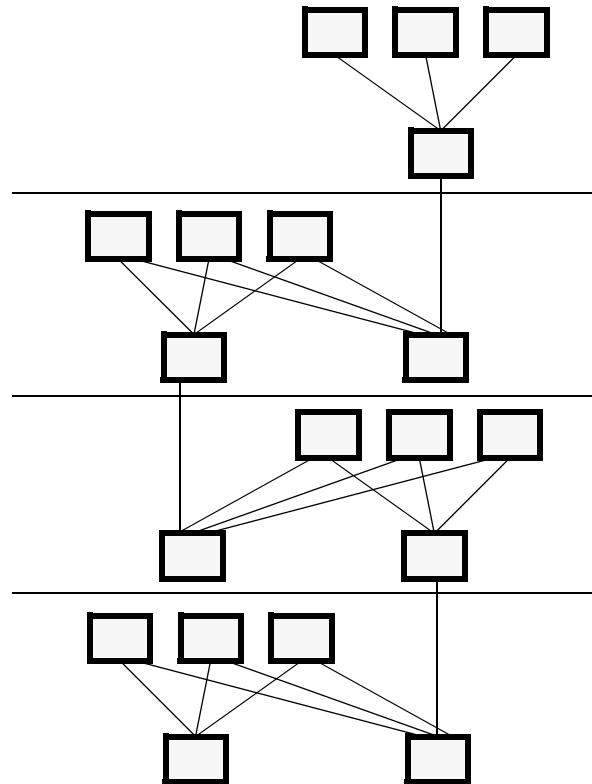
### 8.3.2 Family Patterns



*Figure 51: Basic Pattern for Diversity*

The basic pattern for diversity used to build a family architecture is a combination of a generic BB with its specific BBs (see figure 51). Specific BBs can register with the generic BB if they obey the abstractions provided by the generic BB.

The family architecture is based on the architectural skeleton introduced in section 7.8. The family architecture shown in figure 52 can be built from the



*Figure 52: Regular Layered Diversity*

basic diversity pattern and layer access generics which handle the access to the next layer above (see section 7.5.4). Layer access generics make it possible to configure BBs in lower layers, because specific BBs have no direct relations to higher layers.

### 8.3.3 Extensibility

Extensibility of systems is one of the major challenges during the development of a product family. Extensions may be conservative or non-conservative. Conservative extension does not change BBs in the system. Non-conservative extension extends a system by affecting existing parts.

New features are easiest to handle if they can be developed and incorporated through new components only. Existing systems are not affected and no new versions of components already deployed need be developed. The BBM supports this through the development of generic components in different layers of the architecture. The restriction to uni-directional dependencies clearly limits the knowledge of BBs of their environment. Higher-layer BBs can be substituted without the need to make changes in a lower-layer BB [Fra97]. The skeleton of generics (see section 7.8) is extensible in all layers. Generics are the mechanism for achieving conservative extensibility.

There are many different forms of non-conservative extension. Extended versions of existing BBs are the easiest. The most difficult extensions are those which require changes to the architectural skeleton, that is, changes have to be made in many generics. In such a case the system must be updated almost in its entirety.

*Heuristic 82: A desirable non-conservative extension is the refactoring of a BB to a generic BB.*

Forcing extensions into new BBs only, without updating existing BBs, degrades the quality of the architecture and limits the possibility of realising further extensions. In such a case parts of the system will be redundant in slightly different versions. Short-term considerations and long-term architectural conceptual integrity have to be carefully weighted, since the architecture is the basis for a long-lived product family. Compromise will eventually destroy its integrity.

Refactoring BBs into generic and specific ones is a way to remove duplicate implementations. However, a stable generic is rarely developed in one step. Feedback from implementation is necessary. If variation is only conceptually known, designing stable interfaces between a generic BB and its specific BBs is difficult.

Besides the ability to extend a system with new functionality, it is also important to be able to reduce it. It should be possible to remove functionality which is not needed or outdated from a system. The minimisation of functionality through BB configuration under the constraint that all the required features are supported is called configuration to minimum. Configuration to minimum is a desirable characteristic for a system to support the management of evolution [Par79]. It means that each product contains exactly the BBs which are necessary for it, and no more.

### 8.3.4 Feature Mapping

*Heuristic 83: A good family architecture is one whose BB structure resembles the feature structure, i.e. a good family architecture is feature-oriented.*

Indeed, a feature-oriented architecture localises foreseeable evolution and extension in small sets of BBs. Development, then, takes place as a development of increments to a stable base.

Product managers and architects are jointly responsible for a product family's features. Product managers translate customer wishes into features. Architects contribute features from a technical perspective. Any other stakeholder may also suggest features (see section 11.1).

Product features and their relations are an input for architects. To achieve feature orientation, the architects model the system so that a feature is realised by a small set of BBs. Dependent features may result in dependent BBs. Mutually dependent features can, if they are small enough, be implemented in common BBs. Features, that are not necessarily present together in a product, should be implemented in separate BBs. We define, therefore, the following relations (see figure 53) for the set  $F$  of features  $f$  and the set  $B$  of Building Blocks  $bb$ :

the feature dependency relation (FD):

$$FD := \{ (f_1, f_2) \in F \times F \mid f_1 \text{ depends on feature } f_2 \};$$

often the dependency relation contains only direct dependencies. FD can be build by the transitive closure over the direct dependency relation;

the feature exclusion relation (FE):

$$FE := \{ (f_1, f_2) \in F \times F \mid f_1 \text{ and } f_2 \text{ are not both part of the same product} \};$$

the BB dependency relation (BBD):

$$BBD := \{ (bb_1, bb_2) \in B \times B \mid bb_1 \text{ depends on } bb_2 \} \text{ and}$$

note, that the BBD is a partial order (reflexive, transitive, anti-symmetric) as defined by the BBM and, therefore, BBD is transitively closed;

the feature implementation relation (FI):

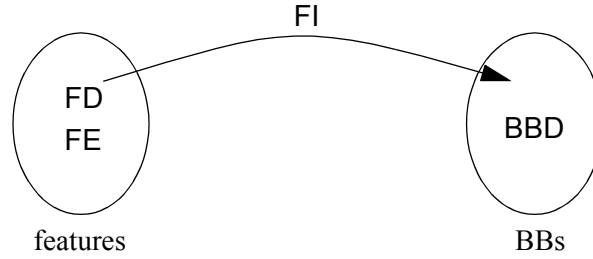


Figure 53: Feature Relation and BB Relation

$FI := \{ (f, bb) \in F \times B \mid f \text{ is (partially) implemented by } bb \}$ .

A feature  $f$  may be directly implemented by a set of BBs, that is, each of the BBs of this set partially implements  $f$ .

From the above defined relations we can construct the relation FID of independent features:

$FID := \{ (f_a, f_b) \in F \times F \mid (f_a, f_b) \notin FD \}$ ,

FID is the complement of FD

the set of feature a particular feature  $f$  depends on:

$FD(f) := \{ f_2 \in F \mid (f, f_2) \in FD \}$

the set of directly implementing feature BBs of a certain feature  $f$ :

$FBB(f) := \{ bb \in B \mid (f, bb) \in FI \}$

the set of BBs a particular BB  $bb$  depends on:

$BBD(bb) := \{ bb_2 \in B \mid (bb, bb_2) \in BBD \}$

and the set of completely implementing BBs of a certain feature  $f$ :

$CIBB(f) := \{ bb \in B \mid bb \in FBB(f) \vee$   
 $(\exists bb_1 \in FBB(f): bb \in BBD(bb_1)) \}$

Starting from one feature one can construct a set  $FD(f)$  consisting of the feature and all its dependent features. Following the implementation relation for all elements of  $FD(f)$  leads to the respective feature BBs

$FDBB(f) := \{ bb \in B \mid \exists f_2 \in FD(f) \wedge bb \in FBB(f_2) \}$ .

These BBs have a dependency relation to dependent BBs. Thus, starting from one feature one can obtain a consistent partial system that implements that feature

$$PS(f) := \{bb \in B \mid f_2 \in FD(f) \wedge bb \in CIBB(f_2)\}.$$

By using a feature list of a product one obtains a complete collection of BBs which realise this product.

Figures 52 and 53 can be combined to obtain figure 54. Four application features are shown in their mapping to specific BBs.

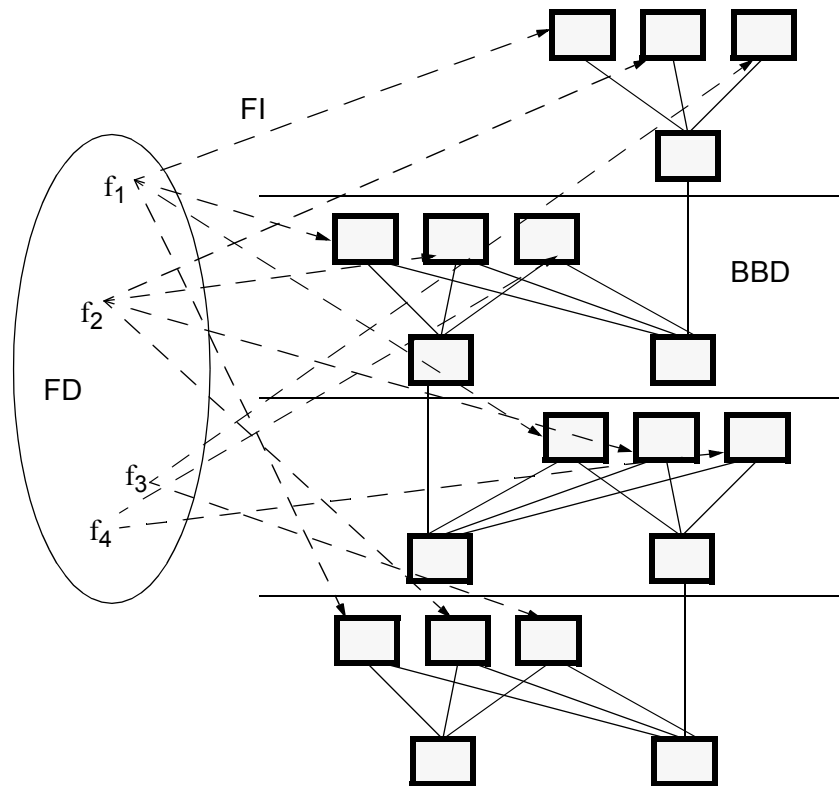


Figure 54: Application Feature Implementation Relation

The relations, described above, are used as a basis for assessing the quality of a family architecture. A good family architecture is one in which independent features can evolve independently. In such a case feature adaptation leads to adaptations in FI-related BBs only.

Excluding features are implemented by different BBs, if and only if

$$\forall (f_1, f_2) \in FE \rightarrow \{bb \mid (f_1, bb) \in FI\} \cap \{bb \mid (f_2, bb) \in FI\} = \emptyset$$

Independent features are independently configurable, if and only if

$$\forall (f_1, f_2) \in \text{FID} \rightarrow [ (\text{FBB}(f_1) \cap \text{CIBB}(f_2) = \emptyset) \wedge (\text{CIBB}(f_1) \cap \text{FBB}(f_2) = \emptyset) ]$$

In a good architecture, BB dependency relations follow the feature dependencies FD.

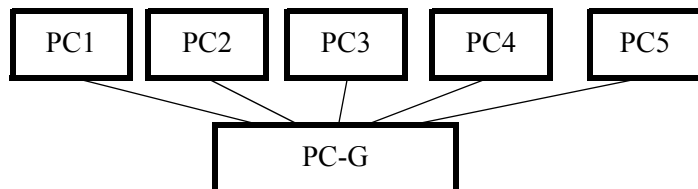
A product family architecture which is feature-oriented allows configuration of a product with minimal functionality consisting only of the functionality of its required features. We call this configuration to minimum.

### 8.3.5 Example: tss Feature BBs

We shall give some typical examples of features of the tss product family.

#### Card Maintenance BBs

BBs in the EM layer (section ) which are responsible for the maintenance and supervision of the peripheral hardware cards are structured so that each peripheral hardware card type has a corresponding maintenance and supervision BB. Changes in the PCs lead only to changes in the corresponding maintenance and supervision BBs (figure 48). Since a large part of the functionality for maintenance and supervision has been standardised for all the peripheral cards, a separate generic BB, PC-G, implements this standard functionality (Figure 55). During the design a careful analysis has been performed to determine which SW parts can be standardised and which must be specific per PC. A customer feature of a new PC type leads as a result of HW mirroring to an extra PC-specific BB in the central controller SW. Figure 55 shows the PC-G with five card specific BBs.



*Figure 55: Peripheral Card Maintenance*

The card maintenance BBs define recovery actions, error handling, configuration parameters of the peripheral cards. Peripheral cards and their maintenance BBs are self-describing (see section 7.8).

#### Line Handling

The line-handling BBs are logical resources which handle all attributes and properties of lines as defined by ITU, i.e. subscribers (analog, digital, ISDN). ITU describes

logical resources as managed objects. BBs are structured according to these object definitions.

### **Announcement Handling**

The tss system also offers automatic announcement services. Examples are automatic time announcement, operator console number announcement and automatic subscriber number announcement for directory services. The BB announcement handling in the service management layer receives ASCII strings containing the time, operator console number or the subscriber number. It has to translate the ASCII string into a number of speech fragments. A special service function card plays the announcement for a caller. Each actual announcement consists of a fixed part (“the current time is”, ”hours”, ”minutes”, “and”, “seconds”) and a flexible part (“twelve”, ”thirteen”, ”fourteen”). However, the translation will vary for each type of announcement. The subscriber number string “213719” is translated into the speech fragments “two”, “one”, “three”, “seven”, “one”, “nine”. The same string for time announcement translates to “twentyone”, “thirtyseven”, “nineteen”. Each translation has been encapsulated in one BB. Adding announcements in different languages, for example, only requires loading the corresponding speech fragments and adding a new translation BB.

### **Call Facilities**

The call facilities in the service management layer are encapsulated in a separate BB per facility. Facilities are state machines which are active during the call. That is, a facility such as *call forwarding on busy* is a type of state machine which is instantiated per call. A generic BB implements the basic call model. Facilities extend this basic model, that is, facility BBs are coupled to the generic BB. A customer requiring a new facility receives just the corresponding BB.

---

## **8.4 Managed Evolution**

The functionality of almost all products is constantly being changed. The technology used to realise those products is also changing. Evolution is a very important issue for successful product development. The reason for that is that evolution affects all areas of development. Each analysis and design task has to be considered from the perspective of evolution.

The important point when preparing for evolution is that one addresses those facets of evolution which can be handled without attempting to address the unpredictability of the future. An architecture’s future-proofness (see section

2.2) therefore is relative to those issues which are foreseeable. Through consequent usage of the available knowledge about likely changes, evolution may be dealt with in product development in a controlled manner.

Perry [Per94] characterises evolution in three dimensions: relevant domains, experience and process. He emphasises that these dimensions capture the nature of evolution much better than performed changes like corrections, improvements and enhancements.

Several of the concepts of the BBM address evolution. The three design dimensions - object, aspect and thread - separate design issues and support independent handling of them. Evolution should preferably even be limited to a single dimension, the object dimension (see section 3.3). BBs and layers support evolution through the incrementality of layers (see section 7.4.2) and generic BBs (see section 7.5.3)

The family architecture, then, is focused on features. This allows flexible definition of products as members of a product family. Moreover, since features change over time and new features are conceived, different products are defined via different feature sets, and a product may over time evolve into a different one. Conservative and non-conservative extensions (see section 8.3.3) are both necessary to keep an architecture balanced with respect to conceptual integrity and development efficiency.

The concept of a product family is very useful because of its mixture of properties. It is a business concept as well as a technical one ([MF93], [ML97]). On the business side a product family is viewed as a collection of parallel and consecutive products made out of sets of features. On the technical side a product family architecture provides a collection of BBs from which different products are configured.

### Heuristics Overview

*Heuristic 80: Use a feature relation graph to describe relations between features.*

*Heuristic 81: Design a system infrastructure generic to handle data-driven diversity.*

*Heuristic 82: A desirable non-conservative extension is the refactoring of a BB to a generic BB.*

*Heuristic 83: A good family architecture is one whose BB structure resembles the feature structure, i.e. a good family architecture is feature-oriented.*

---

## 9 Comparison With Other Methods

To compare the BBM with other design methods and approaches we take a look at their architectural meta-models. The architectural meta-model is the underlying model of architectural design methods (see section 2.3). Often, design methods have no built-in notions of architecture. However, each design method induces a specific kind of architecture. SDL, for example, induces systems that consist of asynchronously communicating state machines (see section 9.2.3). We shall call those elements of an architecture which are required or induced by a method the *architectural meta-model* (AMM) of that method.

In this chapter we shall first summarise the architectural meta-model of the BBM and then compare it with the architectural meta-model of other design methods and approaches.

---

### 9.1 The Architectural Meta-Model of the BBM

The architectural meta-model of the BBM consists of

- a domain object model,
- a product feature dependency model,
- the Building Block design dimensions, that is the object model, aspects and the concurrency model,
- the Building Block dependency model, and
- the deployment model.

Note that the first two model are created as part of the architecting context of the BBM.

The domain object model is created as part of the application domain modelling task (see section 2.6.2) which is not part of the BBM per se, but the BBM requires it as a necessary input. It consists of the domain objects, their behaviour and relations.

The product feature dependency model is the result of the commercial product design task (see section 2.6.3) and not part of the BBM per se, but the BBM requires it as input. The product feature dependency model consists of product features and their dependency relation. It is described in section 8.1.

---

## 9.2 Traditional Development Methods

In the following we shall take a look at traditional software development methods and approaches. They have been selected for their historical importance. These methods and approaches shall be examined with respect to their support for software architecture. Each of these methods has either an implicit or an explicit architectural meta-model. A more general comparison is given in [Wie98b].

### 9.2.1 Structured Design

Structured Design [YC79] brings structure to the task of designing a system by analysing system functionality in a top-down manner. It uses data flow analysis to analyse the flow of data from input to output. A data flow diagram models the functionality of a system in terms of data-transforming functions. So-called Transform and Transaction Analysis is used to map these data flows to a hierarchy of functional modules. Modules at the leaves of the hierarchy, ideally, have the task of performing simple computations, while modules in the hierarchy have the task of controlling and coordinating the flow of data from one leaf module to the next. The top of the hierarchy is called executive module. The model for this module structure is the hierarchical organisation as it is often found in human organisations. Structured Design introduces cohesion within modules and coupling between modules as measures for the quality of a design.

Constantine [Con80] sees function-oriented structuring, such as Structured Design, and object-oriented structuring as two extremes. He concludes that which of the two will be used will (usually) depend on the nature of the problem.

We describe the architectural meta-model of structured design as consisting of a hierarchy of functional modules. Structured design does not provide other

modelling means or views. Exceptional cases such as faults are not really handled in structured design. The provided architectural support is minimal.

### **9.2.2 Bare Operating Systems and Real-Time Kernels**

A simple approach is often used in the domain of real-time systems where the modelling of dynamic behaviour has always received the most attention. As a system's performance is of critical value to its function, system design focused on structuring of execution entities, their prioritising and execution time prediction. Operating systems and real-time kernels provide good support for this dynamic structure.

However, now that systems are becoming larger, reuse of existing components and system evolution support are becoming increasingly important. Moreover, the nature of these systems is unfortunately rarely such that the chosen thread and/or process structures match with reusable parts. The architectural metamodel consists of communicating processes and enclosed threads. The exclusive usage of threads and processes results in a lack of structuring means which should be overcome by additional structures

### **9.2.3 SDL**

Specification and Description Language (SDL) is a system description language based on state machines. A state machine is also called a process. State machines communicate via the exchange of asynchronous messages. State machines may be grouped into functional blocks. SDL-92 [FO94] is an object-oriented extension of SDL. It adds a distinction between types and instances, specialisation of types into subtypes, and the concept of generic types. A system instance consists of a network of connected peer block instances. These block instances may be composed either from other block instances or from process instances. Process instances build a communicating network inside the connected blocks.

The architectural metamodel consists of communicating statemachines. SDL processes serve as both structural entities and behavioural entities. This is the reason for the intuitive simplicity of SDL. However, it prevents the design of optimised structures independently for structure and behaviour. The structure becomes artificial for algorithmic and data-oriented parts of an application which are not statemachines. In such cases SDL hides the real complexities in transition procedures.

### 9.2.4 ROOM

The Real-time Object-Oriented Method [SGM\*92] is based on an SDL-style of design. It shares the identity of structural and behavioural modelling with SDL. Components are called actors and are functional blocks. An actor is both a static and a dynamic entity. It communicates with other actors via message-based communication. Actors share no memory. Actors are arranged in layers. Layers have interfaces called service access points. An actor communicates either to another actor in the same layer or to another layer via the service access points. Procedural libraries can only be used inside an actor.

The architectural metamodel consists of communicating statemachines which can be placed in layers. ROOM adds layers and provides the possibility for more advanced structuring. However, there is still limited modelling flexibility because there is no distinction between modularisation and execution entities.

### 9.2.5 OMT

The Object Modelling Technique (OMT) [RBP\*91] is probably the most used object-oriented design method. Its main intention is to closely couple problem analysis and system design. Object modelling is used as a vehicle which provides a conceptual continuum from problem analysis to implementation. The analysis phase not only analyses the requirements, but also builds an object model of the system to be built. OMT uses three models:

- the object model itself, which describes classes and associations between classes,

- the dynamic model, which describes state transitions in classes and global event flow between classes,

- the functional model, which describes data flow and functional dependencies between classes.

These models are used during the development phases. The OMT process identifies the three phases analysis, system design and object design.

The analysis phase is concerned with understanding and modelling both the application and the domain within which it operates. Analysis takes the problem statement as initial input. This input is enriched with knowledge about the operational environment and the intended usages. On the basis of these inputs an analysis model of the functionality of the system is built, consisting of the object model, the dynamic model and the functional model.

The overall architecture of the system is determined during system design. On the basis of the object model, the system is structured into subsystems, classes are grouped into concurrent tasks, and further decisions about inter-process communication, data storage, and priorities for design trade-offs are taken. The chapter on system design in [RBP\*91] describes several system design concepts (see table 4).

Architectural styles such as <i>horizontal layers</i> , <i>vertical partitions</i> and <i>pipeline-</i> and <i>star-like system topologies</i> can be used to structure subsystems.
For the overall control architecture, three control styles are given to handle externally visible events: <i>procedure-driven sequential</i> , <i>event-driven sequential</i> and <i>concurrent</i> .
Internal control (within a process) can be <i>purely procedural</i> , <i>quasi-concurrent call-back scheduling</i> or <i>concurrent threads</i> .
So-called boundary conditions give a functionality classification in <i>normal operation</i> , <i>initialisation</i> , <i>termination</i> and <i>failure</i> .
Common architectural frameworks for describing classes of systems are: <i>batch transformation</i> , <i>continuous transformation</i> , <i>interactive interface</i> , <i>dynamic simulation</i> , <i>real-time system</i> and <i>transaction manager</i> .

*Table 4: System Design Concepts of OMT*

During the object design phase, the analysis models are enriched with detail using the dynamic model and the functional model.

We describe the architectural meta-model of OMT as being based on the object model. The dynamic model and the functional model provide other views which refine facets of the object model. During the system design phase classes are grouped into subsystems and into concurrent tasks. The object design phase is again based on the object model. It uses the dynamic model and the functional model to further design the classes and their relations. The architectural concepts of the system design phase are not really integrated into the method. Instead of using the architectural concepts for designing a good system architecture, priority is given to the seamless transition from the analysis models to the object design phase. We conclude, therefore, that the architecture of an OMT-based system consists of a network of classes grouped into subsystems and tasks. Subsystems and tasks are more like annotations to the object model than architectural concepts in their own right. However, this is less of a problem for small non-real-time systems as the authors of [RBP\*91] characterise their focus (p. 198, p. 169). Large SW-intensive systems require more explicit architectural modelling for

which the concepts listed in table 4 can be used but OMT does not give any help. However, an application domain model (see section 2.6.2) may be built with OMT.

### 9.2.6 Object-Oriented Software Engineering

We shall now take a look at Object-Oriented Software Engineering (OOSE) [JCJ\*92] as a further representative of object-oriented methods. OOSE works with five different models:

- a requirements model for capturing the requirements,
- an analysis model for giving the system a robust object structure,
- a design model for adapting and refining the object structure to the implementation environment,
- an implementation model consisting of the code, and
- a testing model for verifying the system.

These models are the output of three development phases. The requirements model and the analysis model are the products of the analysis phase. The design model and the implementation model are the products of the construction phase. In the testing phase the test model is produced and the system is tested. The transition from objects in one model to objects in another model is seamless, that is, the identity of objects does not change during transitions.

OOSE defines the requirements model as consisting of a use-case model, interface descriptions and a problem domain model. The use-case model is the most important of all the models. It has a central position for building all other models. The use-case model is expressed in terms of the objects from the domain. The analysis model is structured as an implementation-environment-independent object model derived from the use cases.

The analysis model does not directly use the domain objects from the domain object model. Instead, it derives from use cases three types of objects: entity objects, interface objects and control objects. [JCJ\*92] claims that under changing requirements this object structure will be more stable than a standard object model, that is, changes will be local to hopefully a single object. Entity objects model information which is most stable. Entities are often derived from domain objects. Interface objects model information and behaviour that is dependent on the system's external interfaces. Control objects model functionality which is not captured by the other two object types. They represent the coordination between

entity objects, and between entity objects and interface objects. Often, one use case will result in one control object.

As the focus of OOSE is on analysis, the design is a refinement of the analysis. The concept of a block is introduced to capture the design of an object. There may be interface blocks, entity blocks and control blocks. The notion of a subsystem is introduced to group blocks. Subsystems may contain other subsystems recursively which at the lowest level contain blocks. However, a designer may deviate from the object structure if necessary. Also, a mapping to threads and processes may change the design model. OOSE makes a distinction between application modules, which are called blocks, and components, which are essentially infrastructure libraries. Reuse is discussed only for these library components.

In modelling real-time systems, OOSE attaches real-time requirements to use cases. Behaviour of use cases is mapped onto individual concurrent processes and threads. Threads are seen as orthogonal to objects.

We describe the architectural meta-model of OOSE as based on the three objects types. An OOSE-based system consists of a network of entity, interface and control objects. Block design groups some of the classes and *provides interfaces*. So-called components provide reusable infrastructure libraries. Threads and processes are used to design the real-time dimension. The emphasis of OOSE is on the process steps which lead to a system. OOSE uses the term architecture on a meta-level to denote the structure of its consecutive models.

The centrality of the use-case model, which is a view from outside the system, is the cause of the lack of emphasis on system internal structuring. The text mentions some exceptions where internal considerations overrule the use-case structure, but they are not really integrated in the method. The design of an architecture suffers from the focus on seamless transition of models in different development phases.

### 9.2.7 Comparison with Traditional Development Methods

As described in the previous sections, traditional development methods lack the necessary structures for developing large software-intensive structures. An analysis of object-oriented systems led to the observation of the *tyranny of the dominant decomposition* [TOH99]. Object-oriented methods imply that the world consists of objects only. However, different design needs require different modularities. The BBM, because it originates from the development of large software-intensive systems, provides a richer set of structuring means.

---

## 9.3 Architectural Approaches

We shall now take a look at three other approaches to SW architecture. The first, architectural styles, is a single-view approach. The other two have in common with the BBM that they move away from an overall transformational approach to SW development in which the architecture is an intermediate result obtained on the way to an implementation. SW architecture is represented by different views which are not ordered via temporal relations. These views are maintained and evolved independently of the component level derived from them. They remain valid system descriptions during the active development of the system.

### 9.3.1 Architectural Styles

An architectural style is the dominant structural pattern of a system. Such patterns can be used as a single-view architectural approaches. The design elements of an architectural style are often made visible even in the application domain model. These styles characterise systems in such a way that even customers are made knowledgeable about the presence of a particular style.

Several architectural styles are to be found in [SG96], [RBP\*91] and [BMR\*96]. Examples are

pipes and filters

The pipes and filters architectural style provides a structure for systems that process streams of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters.

blackboard

The blackboard architectural style decomposes a system into three types of components. Knowledge sources are components designed for a specific task. A blackboard can store data that is used to communicate between knowledge sources. A vocabulary describes the data formats which the blackboard is allowed to use. A control component coordinates the knowledge sources at the blackboard.

layers

The layer architectural style decomposes a system into a group of units in which each group works at a particular level of abstraction. A unit makes use

of services of lower layers and provides services to higher layers. Layering has a prominent role in the BBM (see section 7.4).

[JB95] mention the Linda-related [CG89] style SPLICE

The SPLICE architectural style [Boa93b] is a refinement of the blackboard architectural style and relies on a shared data space. For distributed applications, the shared data space is built on top of a SW bus. Applications are then connected to the data space. Data classes are broadcasted on the bus. Applications which are interested in a specific data class subscribe to that data class. The instances of the data class are forwarded to a receiving area of these applications. Applications can poll the area for new data or be notified on arrival. The possibility of buffering decouples update speed of data and reading speed of applications. Measurement data allows for single element buffers in which a new value overwrites the old one. This is an important point for real-time data applications. SPLICE makes processes and components identical.

Architectural styles constitute single-view architectural models [Ben97]. All views coincide in one overall view. This makes systems easy to understand but also limits the flexibility of architectural design. Their adequacy depends on the kind of system under design.

They are a first step towards explicit architectural modelling. They should be considered part of a system architect's handbook. A system architect may use them together with other architectural and design patterns, and with an architectural method (see section 3.5.2).

### 9.3.2 Soni

Through the analysis of 15 systems at Siemens, Soni et al. [SNH95] came to recognise certain structures explicitly or implicitly available in all of the systems.

Figure 56 gives an overview of the different views. A conceptual architecture

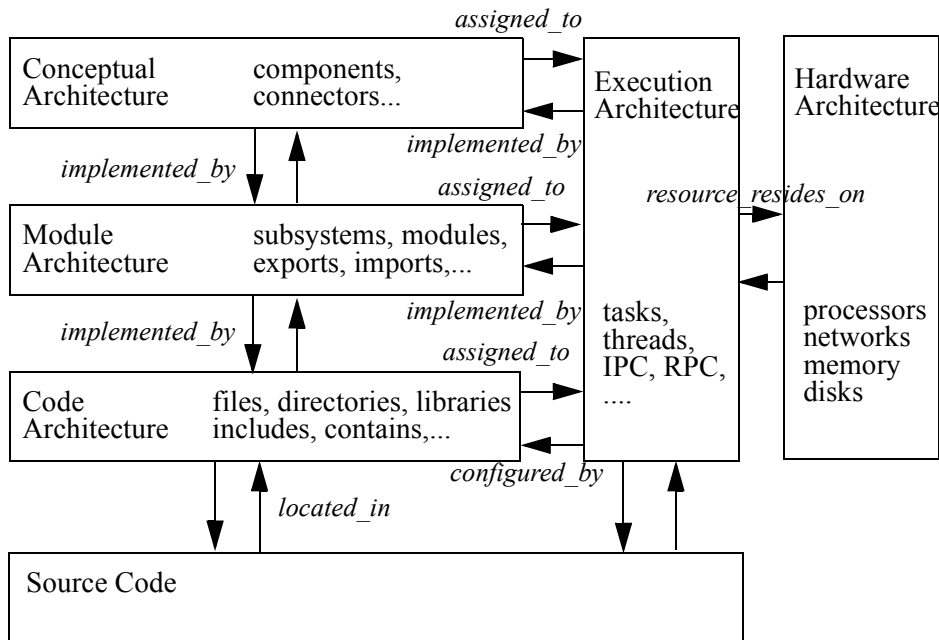


Figure 56: Soni's Architectural Model

describes the system at the highest level of abstraction. It contains a decomposition of the system into its main components and the connectors relating them. A module architecture describes the static partition of the software into modules and the dependency relation between modules. A code architecture describes files, directories and libraries which are used by the module architecture. The execution architecture describes resources of the operating system used for execution and communication, and the assignment of elements from the afore-mentioned architectures to them. Execution elements reside on a hardware architecture. The hardware architecture describes processing nodes and network connections.

The conceptual architecture is comparable with the domain model used by the BBM. However, a domain model is totally in terms of externally observable behaviour, and does not contain a high-level partition of the system. The module architecture is comparable with the object dimension and the BBs. The execution architecture is comparable with the thread dimension. The hardware architecture is comparable with the deployment model of the BBM.

### 9.3.3 4+1 Model

Kruchten [Kru95] presents a model which uses four views (figure 57) to describe a software architecture. A logical view describes the system's functionality in terms which the customers and end users can understand. The development view describes the system's development units. The process view describes the use of execution units. The physical view describes the allocation and allocatability to hardware instances. Scenarios are used as methodical means for specifying functionality within and between the view descriptions. Kruchten explicitly assigns the views to certain stakeholders.

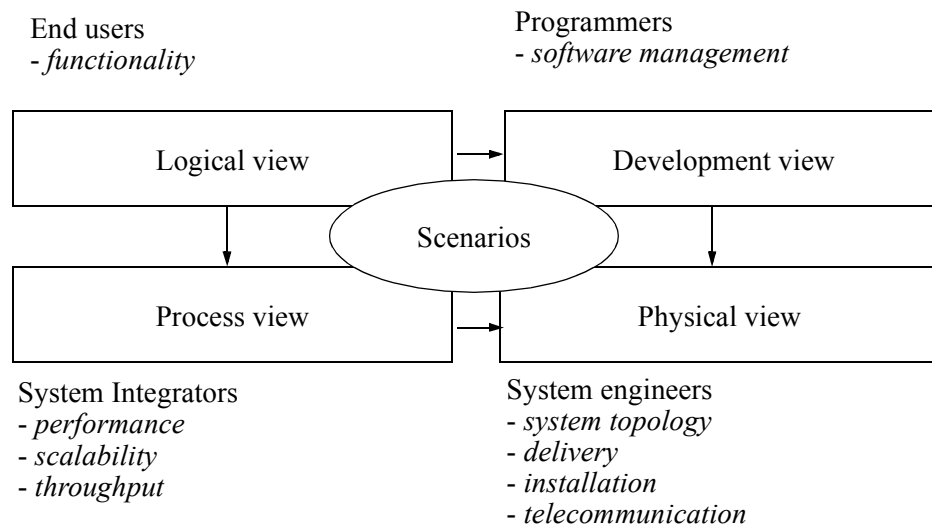


Figure 57: 4+1 Architectural Model

An example of a model derived from Kruchten's model is described in [MHM98]. It uses the four views object view, layered view, task view and scenarios.

### 9.3.4 Comparison with Soni and 4+1

In this section we shall compare the model developed by Soni [SNH95] and Kruchten [Kru95] with the BBM. The architectural models of Kruchten and Soni et al. make a distinction between object and thread dimension for their implementation structuring. Kruchten uses the terms development view and process view, while Soni et al. use module interconnection architecture and execution architecture. The examples given by Soni et al. indicate that the conceptual architecture provides a functional decomposition which is also the top-level

structure for the module interconnection architecture and the execution architecture. Kruchten's logical view provides no constraints for the further structuring in the development and the process view. The BBM by relying on the application domain model uses a similar approach to that proposed by Kruchten.

Kruchten's model is object-oriented and recognises the independence of the modelling of processing resources from development units. It has no aspect dimension, that is, there is no support for functional structuring. Soni et al. work with a functional structuring in the conceptual architecture and distinguish between development units and processes only on the next level. The functional structuring is dominant, object-oriented structuring may be used on a micro-level. Perhaps this is the case because their model is more a reverse-architecting model than an architecting model.

Neither Kruchten's nor Soni's model work with components. Kruchten's development units and Soni's modules are traditional decomposition structures. No separate modelling for flexible integration and composition is addressed. However, such modelling would be a quite natural extension to their approaches.

The code view is only explicitly present in Soni's model. The BBM assumes that the code is structured per BB. Because that is not the focus of the BBM, no additional support is provided. If necessary, a project may add a separate model to describe an independent code view.

The application domain model used by the BBM is close to the logical model developed by Kruchten [Kru95]. The BBM additionally is based on a product feature dependency model resulting from commercial product design. This is an important input because it presents the commercial perspective on the system. Evolution of a system will be via new or updated features.

Table 5 lists the models of the three compared approaches.

Comparison	Soni	Kruchten (4+1)	BBM
Logical Model	Conceptual Architecture	Logical View	Application Domain
Feature Model	-	-	Product Feature
Object Model	-	part of Development	Object
Functional Model	part of Conceptual	-	Aspect
Process Model	Execution Architecture	Process View	Thread
Development Unit Model	Module Architecture	Development View	BB Dependency
Distribution Model	Hardware Architecture	Physical View	Deployment
Code Model	Code Architecture	part of Development	part of BB

*Table 5: Comparison of Architectural Meta-Models*

We can conclude that the BBM has more comprehensive means for structuring the architecture. Especially the product feature model provides important guidance for the structuring in BBs. This is important because a product family architecture needs to support the implementation of the required feature in each of the products.



---

## 10 Method Specialisation

The core BBM can be specialised for specific kinds of systems. Additional architectural patterns and guidelines are integrated into the core method to make it suitable for these kind of systems.

An important question is how the BBM is to be applied to distributed systems such as distributed client-server systems or centrally-controlled distributed systems. In this chapter, we want to show the specialisation of the core BBM for centrally-controlled distributed embedded systems. This architectural style is quite common for large electronic products. Also the tss product family uses that style.

Other specialisations for distributed client-server systems are also possible. Technologies such as COM+ (.NET) and Enterprise JavaBeans provide a good basis for the BBM. Their support for different types of aspect functionality such as persistence and security and for SW components fit with the concepts of the BBM.

The main points of this chapter is to show how the application of the BBM to the central controller of such a system leads to specific objects, layers and aspects. First, the notion of a managed object will be introduced as an extension of the object notion presented in chapter 4. Second, at least one additional standard layer of functionality called equipment management will be introduced. Third, a number of standard aspects such as configuration management, fault management and performance observation are introduced. These specific objects, layers and aspects become part of the specialised BBM because they are relevant for all systems for all centrally-controlled distributed embedded systems.

---

## 10.1 System Architecture

In this section we describe the system architecture of many centrally-controlled distributed systems. Processing nodes of such systems are managed via control relations between the nodes. We take a look at the functional and the control architecture to provide a background for the additional guidelines given by the specialised BBM.

### 10.1.1 Functional Architecture

The system architecture of embedded systems is often partitioned along the flow of domain-specific signals and streams. Signals and streams are forwarded from processing node to processing node. The complete function of the system is realised by the coordinated activities of the processing nodes. Processing nodes are implemented by for instance hardware boards or operating system processes.

Two example architectures illustrate that situation. The first example concerns telecommunication infrastructure systems, the second medical imaging systems.

The architecture of the tss system is partitioned into I/O boards, that is, subscriber and trunk line cards and a central switching matrix (see figure 76 on page 198). Telephone calls are switched by the switching matrix from their incoming lines to outgoing lines. The central controller handles the hardware configuration, determines switching parameters and handles call facilities.

The architecture of a medical imaging system [Pro99] is partitioned along the flow of image data. An acquisition process handles the generation of imaging data by the front-end equipment, e.g. X-ray or MR. A reconstruction process transforms the raw image data into basic images. Image enhancement procedures are applied to basic images. A viewing application interactively allows enhancement and annotation of images. Images are then forwarded to diagnostic workstations, printed and/or stored on different media.

The two examples illustrate functional architectures. Object-oriented modelling, as advocated by the BBM (see chapter 4), takes place within the system's overall functional structure. Hardware entities as well as logical entities such as images and calls are naturally modelled as objects.

Aspects, as defined by the BBM (see chapter 5), are an example of functional modelling. They constitute a refinement of the overall functional structure and are used as standard substructures of BBs (see section 5.6).

A similar approach to the functional refinement is presented in [BM99] where iterations of architectural transformation are following the definition of an initial functionality-based architecture. Architecture transformations are used to adjust the initial architecture to meet quality requirements like maintainability, performance and reliability. Requirements for each of the qualities are met by transforming the architecture in such a way that the new architecture is functionally equivalent and meets the quality requirements.

### 10.1.2 Control Architecture

In centrally-controlled systems, a specific processor, the central controller (CC), runs the central control software. In the hierarchical control relation, the central controller has the role of controlling equipment, while the so-called peripheral equipment has the role of controlled equipment. If the control relation has more than two levels, so-called intermediate controllers (sometimes also called peripheral group controllers (PGC)) have both roles, controlled equipment and controlling equipment. Such an architecture is called central control architecture.

Distributed control architectures also exist but are, in general, difficult to evolve due to the built-in shared information. An interesting alternative to distributed control systems are SPLICE-based systems [Boa93b]. A data backbone forms the essential infrastructure of these systems. Problems, such as management of inter-component relations, can be avoided because of this backbone [Boa93a].

Figure 58 shows a control architecture with tree-type relations. General hierar-

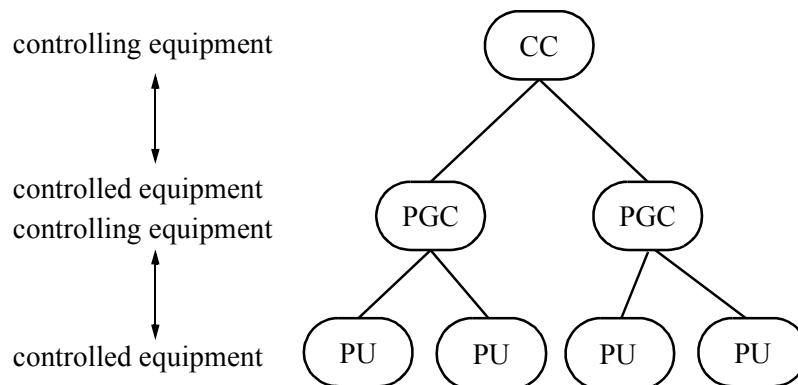


Figure 58: Tree-Type Control Structure

chical relations are also possible. The exact structure is important for modelling equipment control software (see section 10.2.2)

The advantages of a central control architecture are its low implementation complexity and easy extensibility. Knowledge about a specific functional unit is only needed in the directly cooperating units and the controlling equipment. Disadvantages are the single point of failure and the processing bottleneck. These disadvantages can be addressed in the following ways:

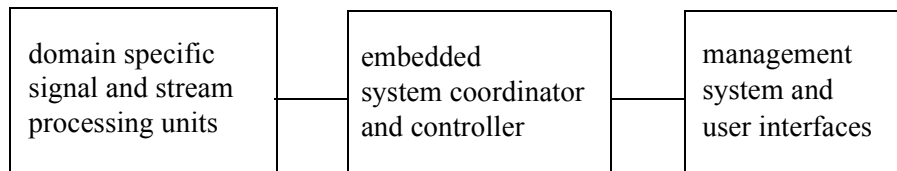
the single point of failure can be avoided by additional reliability measures such as redundancy of controllers in a cold or hot stand-by configuration (see section A.2.2), and

the processing bottleneck can be addressed by locating only functions at the central controller which have a coordinating character over the periphery. Protocols between the central controller and the peripheral processing units need to be designed so that consistency-preserving actions have priority (see section 3.2.3) and the central controller can exercise flow control.

The quality that can be achieved with a central control architecture is often sufficient.

### Management for Centrally-Controlled Systems

An extension of the central control architecture introduces connections to one (or more) management system(s). The resulting system comprises three stages, with the central controller in the middle stage (figure 59). The functionality of the



*Figure 59: Three Stage Control Communication Structuring*

three stages can be characterised as follows:

1. Domain-specific signal and stream processing may be effected in HW and/or SW. The use of specific processing HW with its usually better performance has to be weighted against the usually cheaper and more flexible general-purpose hardware. Often a mixed solution is chosen. This is the field of hardware / software co-design. Processing usually has to meet strict timing requirements. Processing units in this domain are connected, in the most general case, to form a network of streaming relations. Control relations are restricted to a hierarchy. The BBM does not address this area specifically.
2. Building a system consisting of several processing units requires that they be coordinated and kept in a consistent state. Large systems have hundreds or even thousands of those processing units. The coordination and control func-

tion is called system control or embedded system control. It has to bring, and subsequently keep, a system automatically in a consistent state, possibly support graceful degradation and transition to a fail-safe state. This is the area where the specialised BBM is aimed at.

3. The third stage comprises the functions that are responsible for the system's management. They have to support (different kinds of) operators in their daily work. Management may be local or remote, affect single systems or networks of systems, and provide different kinds of user interfaces. Timing requirements are oriented at human perception. The specialised BBM can be used here as well.

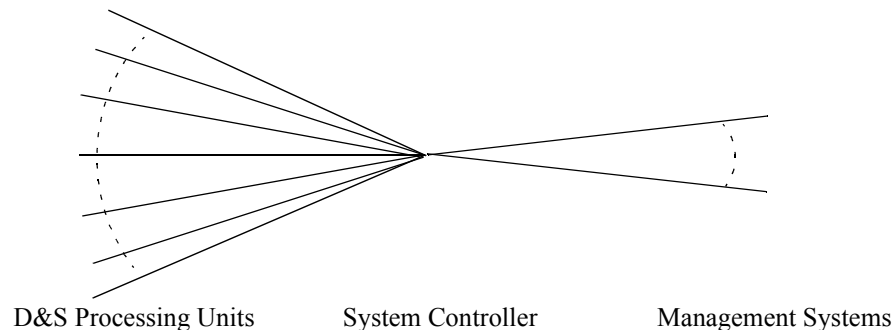
Table 6 characterises the three stages by giving typical examples.

	signal and stream processing	system control	system management
cardinality	depending on system size, different types (between 5 and 50) and instances (between 20 and 1000)	logically 1, may be duplicated for reliability reasons	usually small: < 10, specialisation leads to different types
main functions	signal and stream processing	automatic recovery (and re-configuration) & centralised logical processing	flexible operator support
configuration management	establish processing element parameters	configure equipment & functions (reference DB for system state)	manage configuration data
fault management	self supervise processing	monitor HW and SW configuration, automatic recovery from faults and failures	alarms, fault notifications, error logs
performance observation	generate appropriate data	monitor system performance	performance logs, statistics

*Table 6: Typical Functional Distribution Over The Stages*

Extreme cases are not mentioned in table 6, since they would only extend it without adding any value to it. An example of an extreme case is the requirement to re-configure in a situation of failure in such a way that the system's signal and stream processing function is not noticeably interrupted. Such a case brings hard-real time requirements to the central controller.

Another observation is that with most of these systems the greatest complexity lies in the embedded system controller because it pertains to the whole of the processing units, has to supervise them, execute centralised logical processing, receive configuration change commands from the management system and report configuration changes and errors to the management system. This is illustrated in figure 60, which represents the connection structure of the system controller. The number of connections on the left are much more than the ones on the right. Possible direct connections between the processing units have been omitted.



*Figure 60: Connection Structure of a Central Controller*

Yet another point is that the core system, consisting of the processing units and the system controller, must be able to run without the management systems. The core system is a highly reliable subset. It must run autonomously and maintain system consistency. The reasons for this are that operators cannot be forced to be "on-line", and that the management systems might be located at another site with less reliable connections.

---

## 10.2 Additional Guidelines

In the following we present additional guidelines for four of the BBM design tasks, namely object design, aspect design, composability design and deployability design. The guidelines are related to the system architecture as described in the previous section.

### 10.2.1 Object Design

Modelling the functionality of a system controller (section 10.1) is different from modelling functionality which is assumed to run on a network of distributed peer nodes. The location of an object is not transparent. Part of the functionality of the system controller deals with bringing and keeping the entire system in a consistent state. As explained in section 10.1, a system controller is a processing bottleneck of the system. System functionality has to be designed so that the consistency-maintaining functions of the controller are not impeded by this bottleneck (see section 3.2.3).

As with the core BBM, object design starts with the application domain model. Domain objects are used as an initial implementation object model (see chapter 4).

However, modelling hardware and software of the rest of the system relies on the concept of a managed object. A managed object (MO) models a real resource (RR) for the purpose of control or management [X700]. It encapsulates the underlying resource and allows its manipulation through well-defined operations (figure 61).

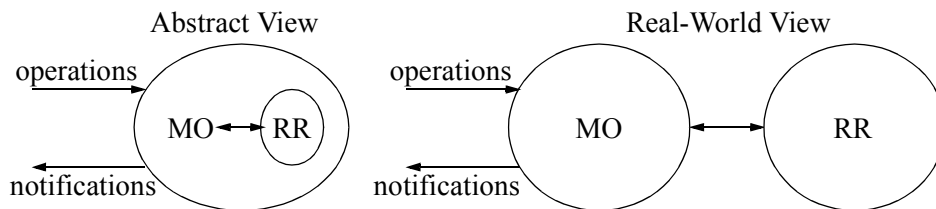


Figure 61: Managed Object

There are two types of real resources, namely *physical* and *logical* ones. A typical example of a *physical* real resource is a processing board and an example of a *logical* real resource is a software processing node. If a logical real resource is located on the system controller itself, it is collapsed and joined with its managed object. We shall base our discussion on the abstract view.

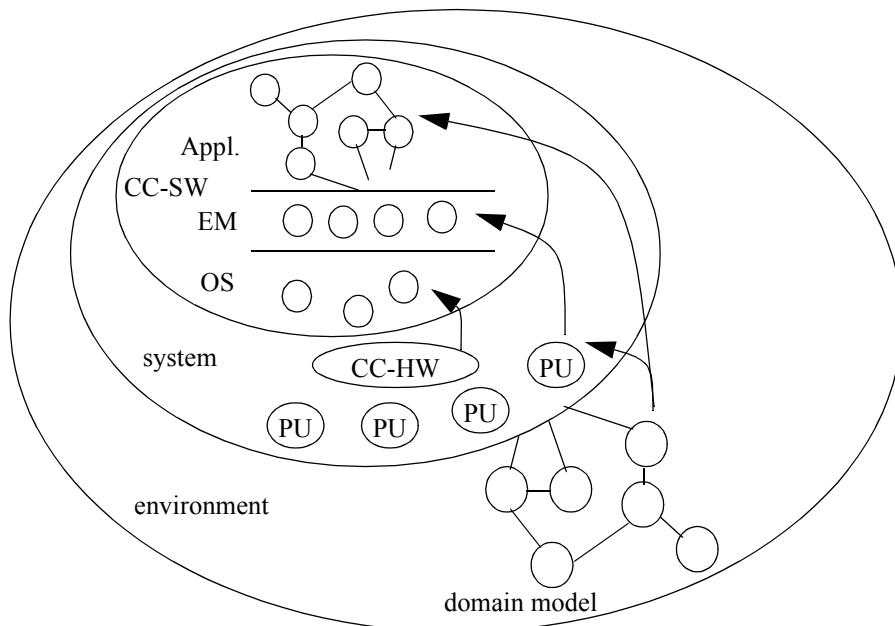
This means that we shall not discuss communication protocol issues which are assumed to be local to managed objects.

The concept of a managed object is comparable to that of a proxy object [GHJ\*94] [BMR\*96]. The proxy design pattern makes the clients of an object communicate with a representative, i.e. the proxy object, rather than with the object directly.

The mapping of domain objects to the system is more complex than the one for the core BBM. Application functionality will be distributed over peripheral units (PU) and the central controller (see section 10.2.4). Furthermore, the PUs will have control objects in the equipment maintenance layer of the CC (see section 10.2.2). In figure 62, the arrows show the mapping of domain objects into system objects and the mapping of hardware entities into the OS and EM layer of the CC.

*Heuristic 84: A managed object may consist of an object in the CC and an object in the peripheral hardware.*

*Heuristic 85: Hardware objects and hardware abstractions of the CC will often be part of the OS.*



*Figure 62: Mapping of External Objects to Internal Objects*

Determining what a hardware object is may not in all cases be trivial. Sometimes hardware entities are modular in themselves.

*Heuristic 86: Maintenance replaceable units (MRU) are good candidates for hardware managing objects.*

*Heuristic 87: Represent MRUs, which only together realise a specific function in the system, by one hardware managing object.*

In the latter case, the hardware handling object needs to be updated whenever an MRU is replaced.

The object design of the tss product family is described in section A.3.1.

### 10.2.2 Composability Design

As described for the core BBM, layering is very common in modelling functionality for electronic systems. The main reason for this is a desire to differentiate between hardware and the functionality realised by this hardware. The functionality realised by the hardware is part of the application domain and evolves with the application domain. The selection of functionality, its partitioning and its implementation technology change over time. The abstract nature of software makes the coupling of application functionality and solution technology a loose one.

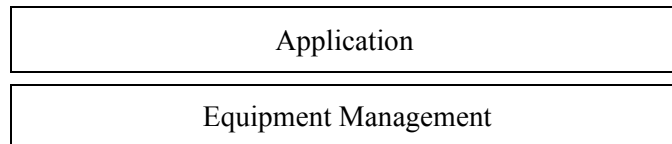


Figure 63: The Basic Two Layers

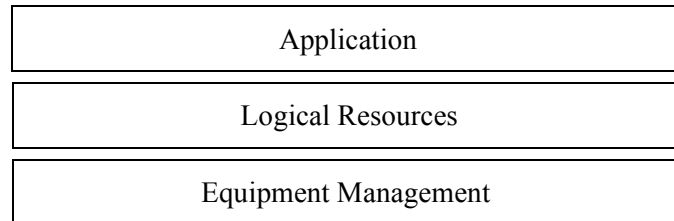
*Heuristic 88: For the specialised BBM, we will always have the two layers in the central controller, namely application and equipment management.*

The equipment management layer (see figure 63) contains MOs for the hardware entities and the application layer contains MOs implementing application functionality. The equipment management layer is sometimes also called hardware reflection layer since it mirrors the hardware.

On the basis of these two layers, which contain objects representing the application functionality and support functionality, additional layers may be appropriate. We shall give examples in which functionality is so extensive that additional layers are reasonable.

*Heuristic 89: When interface abstractions between the two layers have themselves state and behaviour create a new layer for these abstractions. They are then to be modelled as managed objects in their own right and be represented as an intermediate layer.*

Examples of such interface objects are abstract devices and logical resources. They are abstractions of the real hardware with the property that they evolve at a slower speed than the actual hardware. Figure 64 shows the logical resource layer between

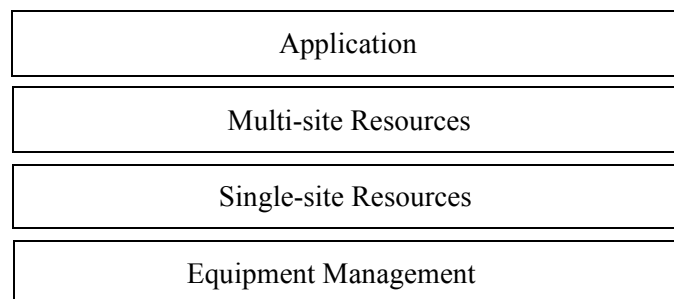


*Figure 64: Three Layers*

the two basic layers (see also [MHM98]).

*Heuristic 90: A further division may be appropriate if additional abstractions are introduced to abstract from the distribution of the controller over several sites. The application functionality then runs on top of the multi-site abstractions.*

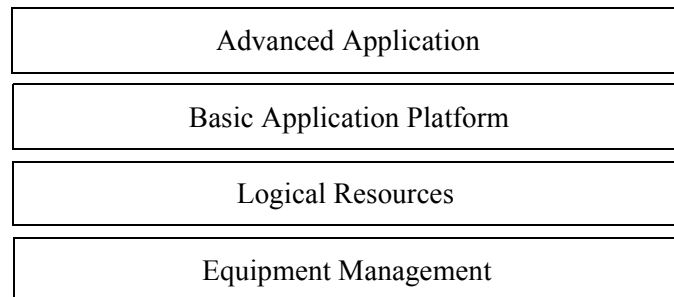
Functionality such as channel abstractions with data transmission bandwidth and redundancy handling are assigned to objects of such a layer. The logical resources layer is split in two: a lower layer for logical devices and single-site abstractions, and a higher one for multi-site abstractions (figure 65).



*Figure 65: Four Layers with Multi-site Resources*

*Heuristic 91: A different division of layers may be appropriate if application functionality extends significantly. An application-specific platform encapsulates application infrastructure abstractions. Various advanced applications may run on this platform.*

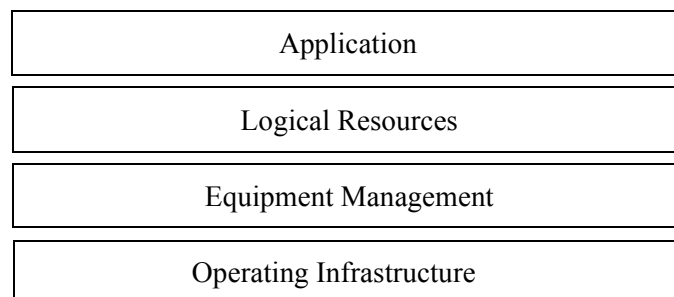
See figure 66 for these layers.



*Figure 66: Four Layers with Basic and Advanced Applications*

*Heuristic 92: Infrastructure functionality such as basic services which should be used by all the objects implemented on the system controller are modelled in the lowest layer.*

See figure 67 for these layers.



*Figure 67: Four Layers with Operating Infrastructure*

As noted earlier, layering is not inherent in the functionality but is a means of introducing structure. The purpose of layering is to achieve separation of concerns and management of complexity.

### **Equipment Management Layer**

The task of the EM layer is to bring and keep the peripheral units, that is the domain-specific signal and stream processing units, in a consistent state, to administer them and to provide an application platform for the distributed application objects.

The spheres of control of the EM layers of the controllers, e.g. CC and PGC, concern the complete subtrees (see figure 68). EM offers generic recovery serv-

ices for the controlled periphery to the application layers. Applications manage to initialise themselves using these service starting from the CC on outwards.

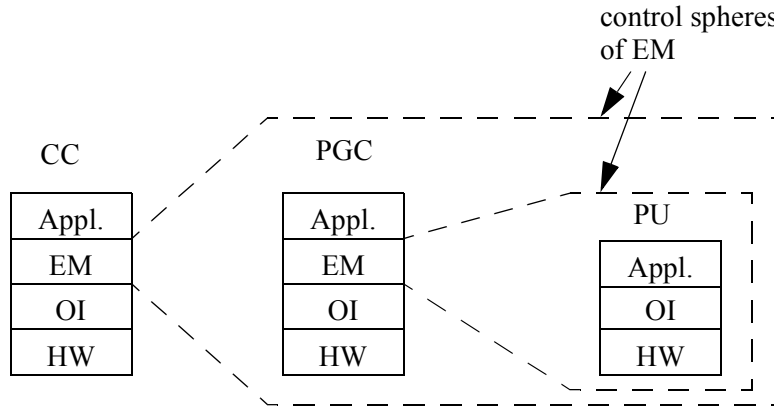


Figure 68: Control spheres of EM

*Heuristic 93: An important set of managed objects and their respective BBs concerns the handling of the PUs. The BBs in the EM layer will reflect the connection structure of the PU.*

The communication structure between the EM layers of the CC and the PUs is shown in figure 69.

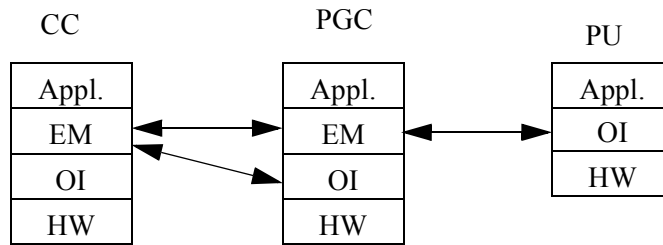


Figure 69: Communication Relations of EM

Section A.3.8 describes the EM structure of the tss product family

To summarise the discussion about layering we can say that centrally-controlled distributed systems have at least three layers:

the equipment management layer which is intrinsic to the task of a system controller,

a layer below the equipment management layer which utilises the HW functionality of the system controller itself, that is, the operating infrastructure, and

an application layer above the equipment management layer which models the application functionality.

Object modelling and layering together are shown in figure 62.

### 10.2.3 Aspect Design

Aspect design for centrally-controlled distributed embedded systems can be much more specific than for the core BBM. The reason is we can assume quite a bit of required system functionality from the system architecture. In particular the facts that the system is embedded, that it has a distributed architecture and that it has a central controller induces certain types of aspect functionality.

The identification of aspects is not different to the one of the core BBM. However, several aspects can be taken for granted, namely:

- a system management interfacing aspect,
- a recovery aspect,
- a data replication aspect,
- a configuration management (control) aspect,
- a fault management (error handling) aspect,
- a performance management aspect.

Note that this list is not a starter set to analyse for functionality but these aspects are inherent from the system architecture. They are types of functionality that are present independently of the application domain.

*Heuristic 94: The system management interfacing aspect consists of the functionality to communicate with a system management system and with the operators.*

*Heuristic 95: The recovery aspect consists of functionality for system initialisation and automatic recovery.*

An example for the design of the recovery aspect is given in section A.3.3.4.

*Heuristic 96: The data replication aspect is a consequence of the distributed architecture. It consists of functionality to replicate data within a managed object, that is, the control and management data of the control object is sent to the real resource object, and changes in the real resource object are propagated to the control object.*

The configuration management aspect, the fault management aspect and the performance management aspect are closely related (see below).

*Heuristic 97: The configuration management aspect establishes configuration parameters according to a system database or operator actions.*

An example of the design of the configuration management aspect is given in section A.3.3.3.

*Heuristic 98: The fault management aspect supervises the system configuration and takes decisions on required actions in case of failure or other abnormalities.*

As an example we describe a widely used approach to fault management. It bases fault management functions on a standardised state model for all components. Failures, faults and errors are arranged into a model of fault classes. System-wide fault management leaves handling of specific faults to the context in which the fault occurred and is based only on the fault classes. Objects are responsible for handling their faults. They choose the appropriate fault class and use the reporting support which is designed for the entire system.

A basic approach for a system is to handle only hardware failures and failures on external interfaces. Fault management functions are then restricted to those objects which deal directly with hardware or external interfaces. Objects which deal only indirectly with hardware and external interfaces handle a boolean availability state. This way specific knowledge about failures remains local with the respective objects.

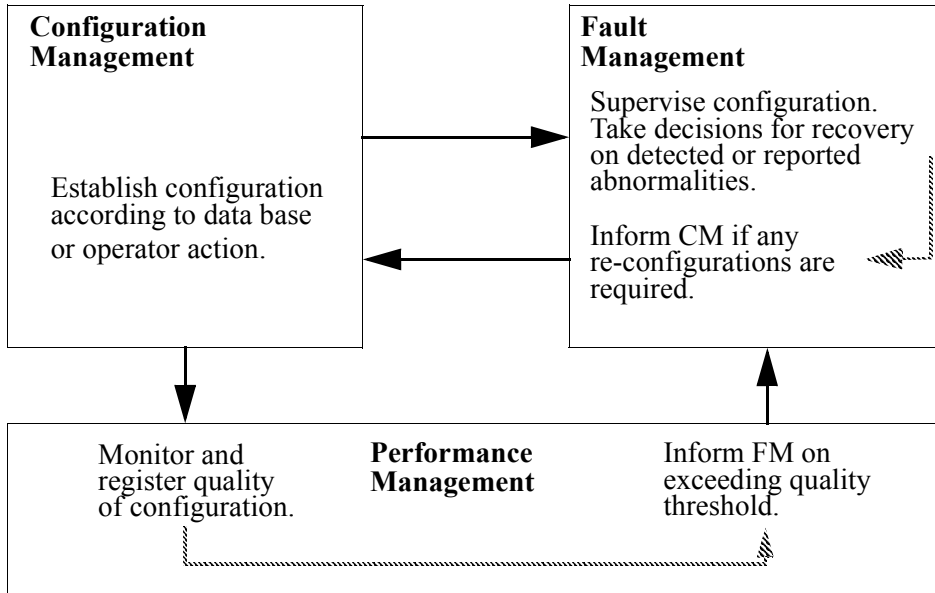
An extension to this basic approach also handles some addressing faults. The system functionality is separated into independently recoverable parts. Multiple address spaces are used to prevent unallowed access across these parts. Fault management ranges from reinitialising individual address spaces to reinitialising the entire system.

A standardised fault management model permits a generic solution for fault management functions. This approach is also chosen for the tss product family.

*Heuristic 99: The performance management aspect has the task to monitor and register the quality of the system configuration. If*

*certain quality thresholds are exceeded fault management is informed.*

The relations between the three aspects are shown in figure 70.



*Figure 70: Relations between CM,FM and PM*

The terms configuration management, fault management and performance management stem from the system management functional areas (SMFAs) of the ITU standard on system management [X700]. The standard additionally identifies the areas accounting management and security management. They may be aspects as well but they are not of such general importance because for many systems they are functional objects or not part of system functionality at all. The terms configuration management, fault management and performance management are synonyms for configuration control, error handling and performance observation, respectively.

The aspect design of the tss product family is described in section A.3.3.

#### 10.2.4 Deployability Design

The deployability design follows the description of the core BBM (see section 3.2.5). In a centrally-controlled distributed embedded system the central controller has a specific role: it controls and coordinates the operation of the peripheral units and thus of the entire system.

As a central controller is a potential processing bottleneck in the system careful allocation of functionality is necessary. PUs may be either general purpose units or special purpose units with respect to a system's functionality. Allocation of objects has to take possible resource shortage into account. Some objects may be split into managed objects or be moved to the periphery if possible.

*Heuristic 100: A quite typical design is to separate control functionality from processing functionality. Processing is allocated to the periphery, while control is allocated to the CC.*

The general strategy is to let the PU do as much of the computation intensive tasks as possible. The CC functionality is restricted to control and coordination of the overall functionality.

The tss product family does not have subscriber cards with a tone generator for the dial tone. Instead it has specific peripheral cards with tone generators. In consequence each off-hook event from a subscriber leads to building up of a call in the peripheral unit, the PGU and the CC in order to be able to switch a path from the subscriber to the tone generator located on one of the service function peripheral units. Such a design puts considerable burden on the CC during call build-up limiting the overall system performance.

After an initial allocation of system functionality component boundaries, fault containment unit boundaries, and thread and process boundaries have to be aligned with HW instances.

The BBM specialisation for centrally-controlled distributed systems uses the same main design tasks as the core BBM. Based on the characteristics of the system architecture of such systems we described additional guidelines and examples for object design, composability design, aspect design and deployability design.

### Heuristics Overview

*Heuristic 84: A managed object may consist of an object in the CC and an object in the peripheral hardware.*

*Heuristic 85: Hardware objects and hardware abstractions of the CC will often be part of the OS.*

*Heuristic 86: Maintenance replaceable units (MRU) are good candidates for hardware managing objects.*

*Heuristic 87: Represent MRUs, which only together realise a specific function in the system, by one hardware managing object.*

- Heuristic 88: For the specialised BBM, we will always have the two layers in the central controller, namely application and equipment management.*
- Heuristic 89: When interface abstractions between the two layers have themselves state and behaviour create a new layer for these abstractions. They are then to be modelled as managed objects in their own right and be represented as an intermediate layer.*
- Heuristic 90: A further division may be appropriate if additional abstractions are introduced to abstract from the distribution of the controller over several sites. The application functionality then runs on top of the multi-site abstractions.*
- Heuristic 91: A different division of layers may be appropriate if application functionality extends significantly. An application-specific platform encapsulates application infrastructure abstractions. Various advanced applications may run on this platform.*
- Heuristic 92: Infrastructure functionality such as basic services which should be used by all the objects implemented on the system controller are modelled in the lowest layer.*
- Heuristic 93: An important set of managed objects and their respective BBs concerns the handling of the PUs. The BBs in the EM layer will reflect the connection structure of the PU.*
- Heuristic 94: The system management interfacing aspect consists of the functionality to communicate with a system management system and with the operators.*
- Heuristic 95: The recovery aspect consists of functionality for system initialisation and automatic recovery.*
- Heuristic 96: The data replication aspect is a consequence of the distributed architecture. It consists of functionality to replicate data within a managed object, that is, the control and management data of the control object is sent to the real resource object, and changes in the real resource object are propagated to the control object.*
- Heuristic 97: The configuration management aspect establishes configuration parameters according to a system database or operator actions.*

*Heuristic 98: The fault management aspect supervises the system configuration and takes decisions on required actions in case of failure or other abnormalities.*

*Heuristic 99: The performance management aspect has the task to monitor and register the quality of the system configuration. If certain quality thresholds are exceeded fault management is informed.*

*Heuristic 100: A quite typical design is to separate control functionality from processing functionality. Processing is allocated to the periphery, while control is allocated to the CC.*

---

# 11 Organisational and Process Issues

The way in which an architecture is conceived has consequences for the development process and the development organisation. The BBM has so far been described as an architectural design method in the broader scope of architecting. In this chapter we shall take a look at some consequences for process and organisation. We shall not describe a complete development process or development organisation (see [Kru99b] and [JGJ97]). Instead, we shall concentrate on those parts which are specific to the BBM.

---

## 11.1 The Process of Architecting

The process of architecting has to be such that a system can be developed which fulfils its purpose. The success or failure of a system will depend on how well it is able to serve its purpose under the constraints of cost and time. It is important that architects are in close contact with business and product managers to be able to use their input early in the development life cycle. Architects have to be involved in customer business modelling, application domain modelling and commercial product design (see section 2.6) even if these design tasks are not their prime responsibility. Architects have to analyse requirements for their technical impact and decide on their feasibility. Architectural design and technology, on the other hand, are design tasks which are driven by the architects themselves.

Internally, the process is driven by risk. The architects identify issues of risk and set priorities for their mitigation. Work proceeds with the issues of the highest risk. Risk is regularly re-evaluated. Instead of working on a general level, architects may sometimes therefore be forced to perform in-depth investigations to secure major design decisions. We shall not attempt to describe such a risk-driven process in detail.

---

## 11.2 Development Processes

A business unit needs to execute processes for developing its products, for policy and planning, for managing people and technologies and for production, sales and service [AMO\*00]. We will not describe all these processes but only look at the development processes. The processes which are needed for developing a product family depend on the stage of the development. Initial stage development has to be distinguished from steady stage development.

### 11.2.1 Initial Stage and Steady Stage Development

Initial stage development of a product family is characterised by the absence of a product family architecture and of implemented BBs. In the initial stage a product family architecture and one (or a small set of) product(s) are developed. This development should deliver a basis for the product family.

*Heuristic 101: Develop a first product that can be used as a basis for the product family.*

Only meeting both goals together, a commercially and technically viable product and the product can also serve as a stepping stone for a product family, makes the development successful.

Steady stage development of a product family is characterised by refactoring and extension. The product family architecture may need to be changed to address new product features or new technologies. Existing BBs may need to be refactored and new BBs need to be added.

The product family architecture and the implementation of the products should be kept up-to-date, that is, decay because of environment changes or implementation short-cuts should be fixed through refactoring. BBs are refactored and generics consolidated. The development of new products can take advantage of the fact that a proven base of BBs can be used as starting point. The quality of the product family architecture and its implementation determines how easy the development of similar products is.

### 11.2.2 Initial Stage and Steady Stage Processes

In the initial stage there are two parallel processes and in the steady stage there will be three processes.

The initial stage comprises the product family engineering process and the product engineering process. The product family engineering process executes the process of architecting making use of the BBM. The product engineering process focuses on the specification and implementation of the specific products using the architectural design resulting from the product family engineering process to the extent that it is available.

The steady stage comprises the product family engineering process, the product engineering process and additionally the platform engineering process. The platform engineering process is responsible for the development of the architectural skeleton, that are the common BBs shared by most of the products. In the product family engineering process of the steady stage the architects determine if new features are within the tolerance of the architecture. If this is the case no architectural changes are necessary. Product engineering uses the BBs of the architectural skeleton and may develop new BBs as necessary.

The three development processes product family engineering, platform engineering and product engineering are similar to application family engineering, component engineering and application engineering of [JGJ97].

---

### **11.3 Building Blocks are Stable for Subsequent Phases**

BBs, including generic BBs, specific BBs and system infrastructure generics, are identified in the architectural design process. The architectural design assigns functionality to BBs and defines design guidelines and constraints for BBs. A BB is developed by being specified, designed, implemented, integrated and tested (see section 11.7). A BB remains a stable entity throughout this process. Even in a deployed system the BB is recognisable. Its identity, as designed in the architectural process, will remain stable throughout the system's lifetime. This allows tracing of BBs from component identification to component deployment.

---

### **11.4 Building Blocks and the Waterfall Model**

BBs can be developed in parallel. Areas of risks must then be identified and solutions proposed by the architectural design process. Therefore, the process for the development of the BBs may be simple. The simplest process is according to

the waterfall model, without iterations: specification, design and coding. This should be sufficient for the development of BBs. We moreover use it as the quality criterion for the architectural process, i.e the simple waterfall model should be sufficient to develop BBs.

*Heuristic 102: Define and detail the architecture in such a way that BBs can be developed according to a simple waterfall model.*

Technical know-how and experience are essential for achieving such an architecture.

---

## 11.5 Documentation

The architectural design is documented in the architecture document. Like any other documentation, it should present the logic for the architectural design rather than the historical process [PC86]. The architecture document contains only an architectural view of the system.

*Heuristic 103: The architecture document describes the architectural models such as the BB dependency model, aspect designs, concurrency design and deployability design. The architecture document should be structured in a way which minimises the impact of changes.*

This means that specification and design studies of certain areas have to be recorded in investigation documents, which are later to be replaced by the respective BB documentation. General rules and guidelines are part of the architecture document.

*Heuristic 104: The BB documentation consists of at least three documents: its specification document, its design document and its code document.*

Other documents which support other stakeholders may be documents for test cases, manuals or user interface descriptions. The BB documentation may be based on parts of the investigation documents.

Each of the documents mentioned above may be organised as a set of sub-documents which are independently handled by a document-management sys-

tem. This is especially important if parts of a document differ in evolution characteristics.

This leads to document dependencies as shown in figure 71. Feature specifi-

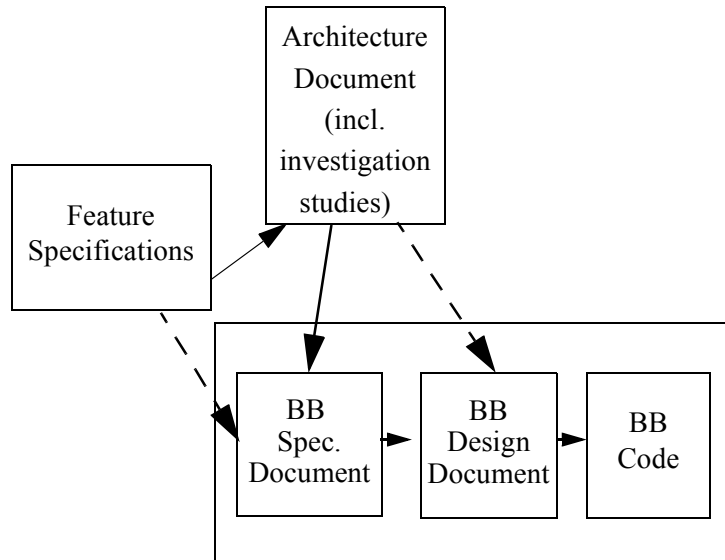


Figure 71: Documentation Dependencies

cations are the input for the architectural models. The architecture document and investigation studies are input for the BB specification and design. However, the complete specifications can not be derived from the architecture document. Additionally, the BB specification has to be based on requirements derived from feature specifications.

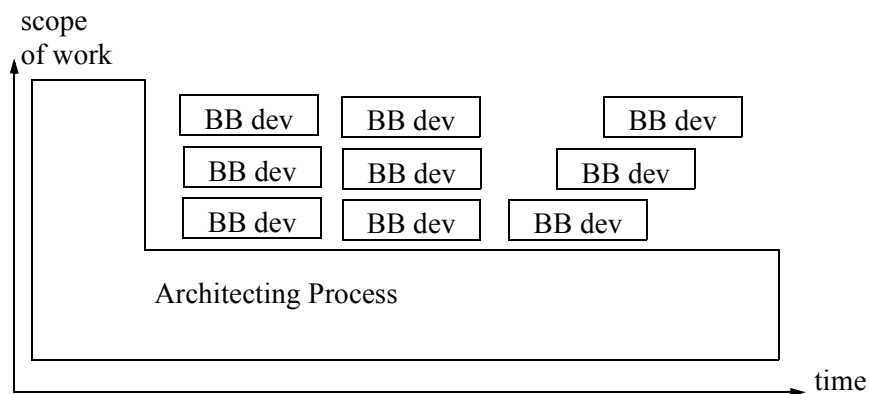
---

## 11.6 Layered Development Processes

The architecting process and the BB development process are separate processes. They are related in that a BB can be developed as soon as it has been sufficiently defined. This leads to the notion of a layered process. One process, the architectural process, is on a lower layer and the other processes, the BB development processes, are on a higher layer (figure 72). The aim is to develop BBs in a simple waterfall model without iteration (see section 11.4).

*Heuristic 105: Consider a deviation of the BB development process from the simple waterfall a quality problem of the architectural process.*

A new risk, identified during BB development, is therefore signalled to the architectural process. The solution is worked out under the control of the architectural process.



*Figure 72: Layered Processes*

The overall design process does not follow the model of phased transformations. Instead, the architectural design remains stable throughout component development.

In comparison with the spiral model [Boe87], the risk-driven initial cycles are mapped to the architectural process, while the final waterfall is taken as the BB development process. Work distribution, work parallelisation and work planning are all directed at BBs.

---

## 11.7 Incremental Integration and Test

The incremental integratability of the BBs is used for the integration and test process. Any developer will be able to integrate and test his BB on the platform of the lower BBs. The system integration is completed when the highest layer has been integrated.

Integrating and testing a BB tests the lower BBs. Achieving a stable set of BBs is easier in such an incremental manner than when all BBs are integrated at once.

*Heuristic 106: Proceed with the process of integration by extending a stable set of BBs with one or a few BBs.*

Starting with the BBs of the lower layers BBs are added step by step to the system to be integrated. The experience is that the time needed for the integration is shorter due to the incremental process.

Furthermore, deadlines can be set in steps according to the incremental structure, from lower layers to higher layers. Layered subsystems will usually be reasonable groupings to have the same deadlines.

Functional tests will be started during incremental integration. Tests with the complete system concentrate on long-term stability and stress conditions. Automatic regression tests and stress test are crucial for the success of evolving systems.

Incremental integratability is the main contribution of the BBM to the testing process.

---

## 11.8 Tool Support

Standardisation is the basis for automation and tool support. The BBM defines several concepts which lend themselves for this. Examples are:

- partial ordering of BB import relations,
- grouping of BB in layers,
- generics which support certain aspects, and
- component model attributes of BBs.

More standardisation is possible in the area of design, coding and development environment.

We give several examples of tools taken from tss. The tools are based on those standardised concepts.

### **Example: tss Architecture Support Tools**

Feature descriptions, relations to other features, organisational responsibilities and various status attributes are kept in a feature database and administered via product

strategy and product progress meetings. Anybody in the product management and development departments may read the feature descriptions and write comments on them.

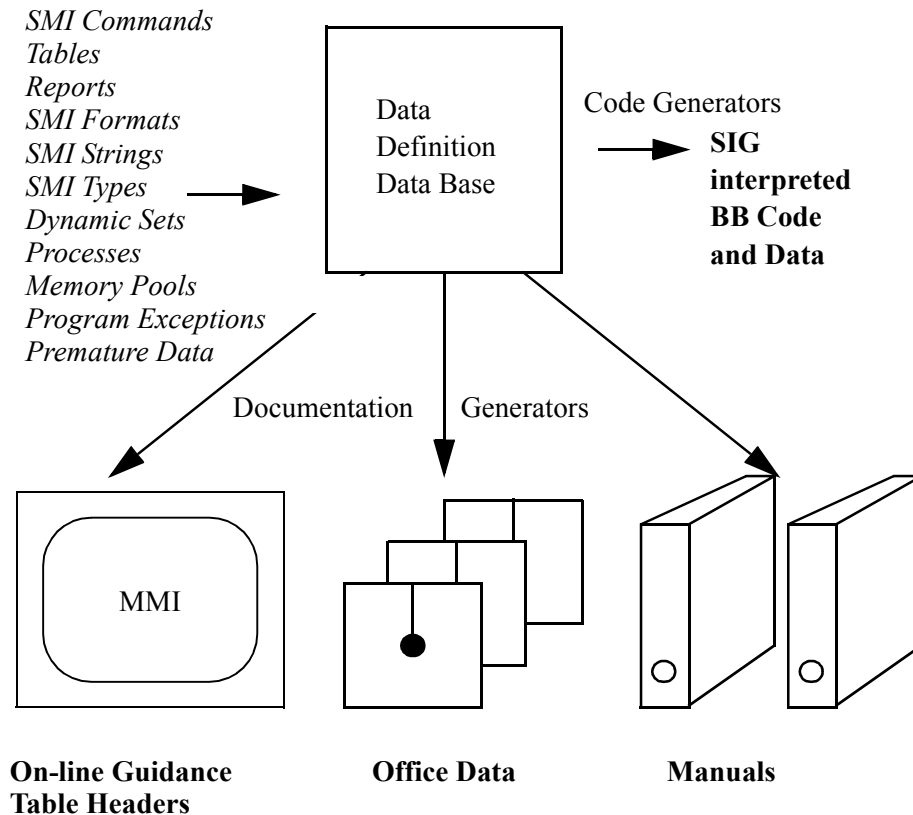


Figure 73: DDD and Generators

The dependency relation between BBs as defined by the partial ordering is managed via a specific administration tool. The allowed dependencies to other BBs are described per BB. Dependencies may not be changed without authorisation from the architectural team.

Different forms of documentation are supported through document generators of the Data Definition Database (DDD). Figure 73 gives an overview of the DDD. Besides the code generators for supporting the system management interface (SMI) code, there are also generators for office data and for manuals. All this documentation is consistent with the system implementation by virtue of *generation from a single source*. The DDD tool thus supports data consistency between different departments.

Data generation for specific system configurations is supported by further database-based generators.

---

## 11.9 Organisational Consequences

Since architecture is not only a phase but a process itself, an architectural team [Kru99a] is responsible for the creation and evolution of the architecture. BB development teams may be organised according to layered subsystems [DKO\*97].

Architects are not specialists in all areas [Mil85]. Figure 74 (adapted from [RM97]) shows an example of the depth of understanding required on the part of the architects. This can be achieved either by an architectural team or by architects together with chief designers who have the required depth of understanding to guide architectural decisions.

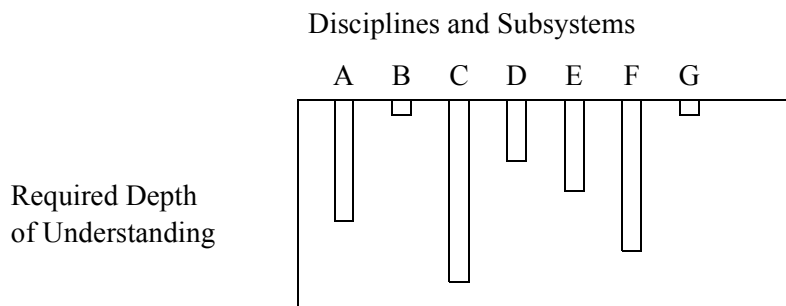


Figure 74: The Architect's Depth of Understanding

### Heuristics Overview

*Heuristic 101: Develop a first product that can be used as a basis for the product family.*

*Heuristic 102: Define and detail the architecture in such a way that BBs can be developed according to a simple waterfall model.*

*Heuristic 103: The architecture document describes the architectural models such as the BB dependency model, aspect designs, concurrency design and deployability design. The architecture*

*document should be structured in a way which minimises the impact of changes.*

*Heuristic 104: The BB documentation consists of at least three documents: its specification document, its design document and its code document.*

*Heuristic 105: Consider a deviation of the BB development process from the simple waterfall a quality problem of the architectural process.*

*Heuristic 106: Proceed with the process of integration by extending a stable set of BBs with one or a few BBs.*

---

## 12 Conclusion

Component-based development is one of the major trends in the software industry today. Many development projects base their products on component technology. Much attention is at present being given to the pros and cons of different component models. However, methods for identifying components are lacking [Szy98]. We regard our work as a contribution in the search for component methods.

The BBM is a component-based architectural design method for large software-intensive product families. It uses components to develop families of products in such a way that particular products can be configured from pre-manufactured components, that is, the BBs. The development of a family provides the context where BBs are identified and developed for multiple use in different products.

The architecting context of the BBM is a rational architecting process consisting of the tasks customer business modelling, application domain modelling, commercial product design, architectural design and technology. The tasks are described in their logical order, their execution is concurrent. The BBM supports the architectural design task of that model (see section 2.6).

Conceptual integrity is essential for the evolution of large systems. Because of the overwhelming amount of detail relevant in the development of such systems, architects get easily distracted from pursuing this integrity. The BBM presents a frame of reference for architectural design. Relevant design tasks and their underlying technical concepts are described (see chapter 3).

The BBM addresses the gap between domain functionality and system functionality by using system qualities and technology choices in the identification of objects and aspects (see chapter 5).

The core of the BBM consists of a number of design tasks, namely object design, aspect design, concurrency design, composability design and deployability design. It uses input from application domain modelling and commercial design (see chapter 3).

Object design starts with domain objects from application domain modelling and refines them to implementation objects (see chapter 4). Aspect design complements object design by identifying functionality which crosscuts objects. The uniform design of aspect functionality promotes conceptual integrity. Aspects are a second partitioning of a systems functionality (see chapter 5). Concurrency design describes the mapping of objects and aspects to computing resources (see chapter 6).

Composability design deals with the identification of BBs and their relations. BBs are construction elements having provides and requires interfaces. BBs are grouped in layers and have a partially ordered dependency relation. Generic BBs, i.e. component frameworks, are means for encapsulation and extension of common functionality. The architectural skeleton is formed by the set of layered generic BBs. These concepts are used in the design of product family architectures. Deployability design describes the mapping of BBs to different deployment scenarios (see chapter 7).

Products evolve through new and updated features. During commercial design a product family is defined as consisting of sets of overlapping features. These features are an important input of the BBM for creating a product family architecture. Relations between commercial features are taken as guidance for the design of inter-BB relations. Feature orientation means that the software architecture is connected to these evolving features. The quality of a component-based product family architecture will depend on its resemblance to the feature structure (see chapter 8).

The design of large software-intensive systems necessitates an emphasis on the actual construction elements. The use of the BBM leads to those construction elements, namely the BBs. A set of complementary global design models describes the position and role of the construction elements. They are the acyclic dependency graph of BBs, the various aspect designs, the concurrency design and the deployability design. They are light-weight and updating their documentation is eased by their relative independence.

Note that it is not possible to make statements about the ease or difficulty of updating an actual design. Likely evolution and extension are taken into account by using input from domain modelling, commercial design and technology choices.

The development process consists of an initial stage and a steady stage (see chapter 11): In the initial stage a single product (or a small set of products) are developed, which have a potential to serve as a base for a product family. In the steady stage more members of the family are developed. The architecture developed in the initial stage is taken as a base for the product family architecture. The

product family architecture is the basis for a long-lived product family. Refactoring is important to keep this base conceptually clean. The criterion for finishing the architectural design is that all BBs can be developed in a classical waterfall.

The BBM is based on a rich underlying architectural meta-model consisting of a domain object model, a product feature dependency model, the Building Block design dimensions, that is the object model, aspects and the concurrency model, the Building Block dependency model, and the deployment model. The first two are taken as inputs from its enclosing rational architecting model. The architectural meta-model provides the basis for expressing the relevant architectural elements of large software-intensive product families.

The BBM is presented as a core method describing the concepts of the method. A specialisation of the BBM for centrally-controlled distributed embedded systems describes additional guidelines (see chapter 10).

The BBM is based on experiences gained in the development of a product family of telecommunication switching systems (tss). Various examples throughout the text describe designs for parts of that system. The appendix contains a description of the architecture of tss and some experience data. The original intention in the development of tss was to develop a configurable product family that would achieve conceptual integrity. Reuse was achieved as a by-product [RF96] [FJ95]. The method has also been used in designing parts of a family of medical imaging systems [Wij00].

The goal, most important for an architecture, is to support the management of development complexity. Component-based approaches provide a way of extending a system but also of reducing the system, notably by removing superfluous components. Keeping an architecture up-to-date and effective is an essential precondition for high-quality product lifecycles.

### **The Future**

The BBM can be evolved and extended in several ways.

The embedding of the BBM in the rational architecting process can be extended to a general multi-view architecting method covering the complete range of the architecting model. Such a method would address multiple viewpoints in each of the tasks, providing logical threads between these viewpoints and give patterns for selecting design mechanisms for such threads.

The relation to Microsoft's .Net [Pla01] can be elaborated. This might be fruitful because .Net does not use a global registry like COM but uses direct

dependencies between assemblies, the name in .Net for components. A so-called manifest describes the interface of an assembly. Consequently, the BBM can be used as a design method for .Net applications.

A further point is the application of the BBM to a wider range of applications. Besides the feedback for the method it would provide more examples of specialisations of the BBM and of aspects in particular. Consolidated infrastructures for

programming environments such as the intentional programming system [CE00] (mentioned in section 5.3) and

server components such as COM+ [Pla99] and EJB [Mon00]

provide support comparable to system infrastructure generics of which each application component can make use. The BBM can be specialised in ways similar to the specialisation for centrally-controlled distributed embedded systems (see chapter 10) to provide more support for these kind of systems.

---

# Appendix A The tss Product Family

This appendix contains an introduction to the digital Telecommunication Switching System (tss), which was the main source of inspiration in designing the BBM. However, this appendix is not an introduction to telecommunications in general, nor to switching systems and telephony. Rather, it is a description of a specific telecommunication system, containing only a brief motivation of its design decisions related to the BBM.

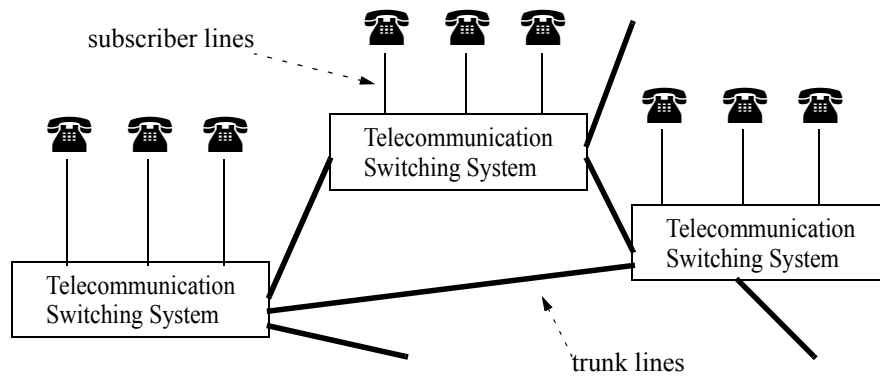
For the design of the tss product family the specialised method for centrally-controlled distributed systems (see chapter 10) has been used. However, the BBM is only used for the design of the software of the central controller. The appendix does not repeat the method descriptions given throughout the rest of the thesis but describes the usage of the architectural concepts.

---

## A.1 tss Introduction

A telecommunication switching system is part of a telecommunication network. The main connections a switching system has, are to subscribers, and to other switching systems (so-called trunks), see figure 75.

For all the connections of a switch specific communication protocols are defined. Some of the protocols are internationally standardised, while others are defined by national authorities or private parties. Many of the international standards have national adaptations. Switching systems, as part of the public telecommunication infrastructure, have to be reliable. Telecommunication is an essential service for life saving and order preserving agencies. Degradation of this service is undesirable and complete loss unacceptable. Switching systems are configured per site in functions and capacity.



*Figure 75: Switching Systems in Context*

The tss product family, developed at Philips Kommunikations Industrie (PKI) Nuremberg (now Lucent Technologies) since the middle of the eighties, is based on a platform approach for telecommunication infrastructure systems. The product family was designed for niche markets with a high variety of features. A key requirement was to achieve a reasonable price even with a low sales volume. Products from the family include public telephony exchanges, GSM radio base stations, technical operator service stations, operator-assisted directory services and combinations of the aforementioned.

The tss product family was built to meet the requirements of a cumulative downtime not exceeding one hour per year. This included repair and upgrading actions. A key design element to meet this requirement is the use of hardware redundancy and graceful degradation. On the other hand, requiring that the system loses its capability to establish new calls not longer than a few seconds and never cancelling existing calls leads to very complex and expensive designs. The tss system is not based on hot stand-by redundancy for the whole of the system and does not provide stable-call-saving. The system is designed to minimise down time and to remain never in an inconsistent state. It autonomously handles first faults (single error assumption) in critical central components such that service is resumed or continued without degradation.

The tss product family is modular, both in hardware and software. Customer visible modularity serves to replace erroneous or adapted components and to extend the system with new functionality. Each tss system contains a database for configuration data and other parameters. Changes to configuration data and

parameters are done in transactions with the associated roll-back if the transaction cannot be finished.

Data about typical sizes and performance of tss systems are given in section A.5.

---

## A.2 System Architecture

The hardware architecture is based on a distributed network of printed circuit boards, many containing microprocessors. A central processor is responsible for central control of the switching system. The software in the peripheral hardware areas (switching matrix and peripheral groups) (figure 76) handles routine tasks, of the kind that occur during call setup/cleardown and digital switching.

### A.2.1 Hardware Components

The hardware of the tss system comprises the following main components: the central processor, the switching matrix, the peripheral groups and the operation and maintenance terminals (figure 76). The central processor contains the system's main intelligence. It is a controller of the other hardware components, which act according to the directives of the central processor. It comprises two processor planes (section A.2.2) and some common units. The switching matrix actually switches telephony lines. The connections, which have to be set up or given up, are decided by the central processor at the subscriber's request. The peripheral groups consist of collections of multi-functional peripheral units (PU) controlled by a peripheral group controller (PGC). In cases where the peripheral group consists only of a single PU there is no PGC. Each PU is of one of three classes. The first class consists of trunk cards which handle connections to other switches. The second class consists of subscriber cards which handle speech and/or data connections to subscribers and private branch exchanges. The third class consists of service function cards which generate special announcements or tones, or support special services such as conference calls. To unburden the central processor, much of the processing is moved into the peripheral groups. The speed of the central processor determines the system's call capacity, i.e. the number of connections the system can setup (usually measured in busy hour call attempts (BHCA)). The operation and maintenance terminals support system management tasks.

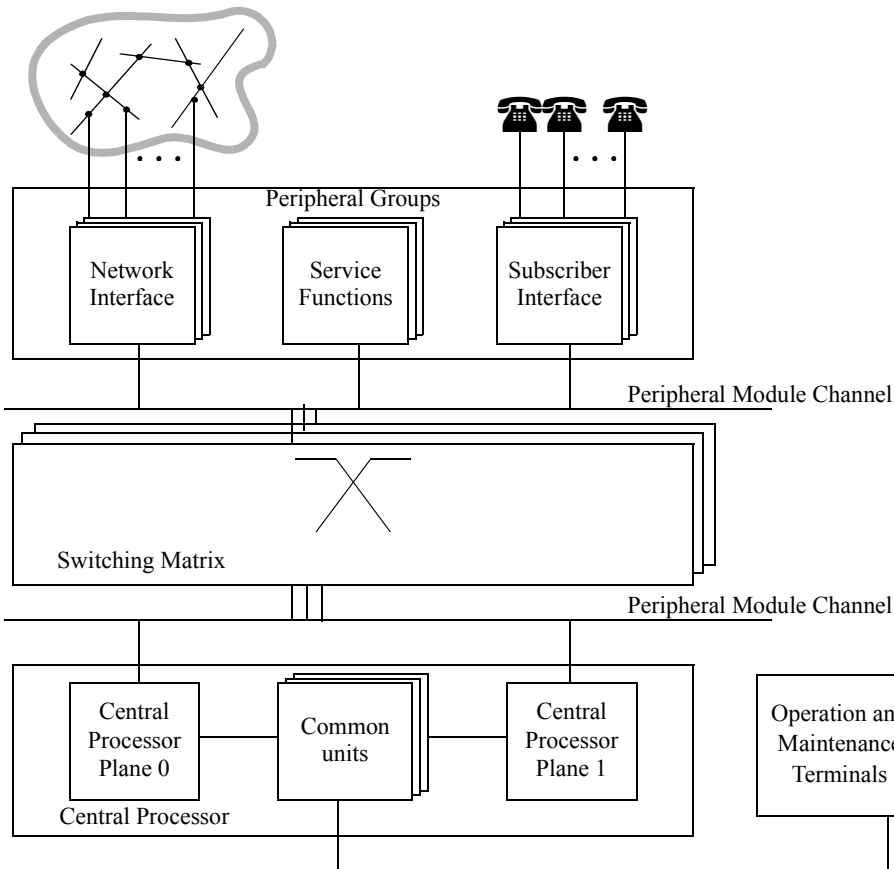


Figure 76: tss Hardware Architecture

The central processor, the switching matrix and the peripheral groups communicate via a bus, called the peripheral module channel (PMC). It carries data and control information. The switching matrix is directly connected to both the central processor and the peripheral groups. The communication between the central processor and the peripheral groups is always relayed through the switching matrix in the same way that speech and data are exchanged between peripheral groups. The central processor and the terminals communicate via terminal connection units, which have direct connections to the terminals themselves. The terminal control units are among the common units in the central processor.

The central processor and the switching matrix both have clocks. The clock of the central processor acts as a real-time clock used to implement timestamps and absolute timers, and gives triggers to the software. The triggers are used by

the process management BB and by the BB which implements relative timers. The clock of the switching matrix is used to synchronise the tss system with the public network.

In figure 77 three peripheral groups are shown. The first and the third consist of only one card each. The first is the speech announcement card (SAG) which is a service function card. The third group is the digital trunk group (DTG) which is a network interface card. The second group consists of the peripheral group controller (PGC) and a set of circuit cards, e.g. analog subscriber card (ASC), digital subscriber card (DSC) and three wire analog subscriber card (TSC). There are about 20 different types of circuit cards. All circuit cards are connected to the universal peripheral slot bus (UPS).

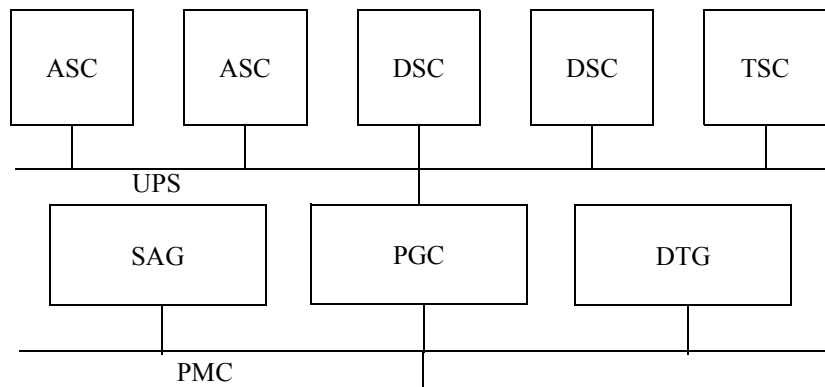


Figure 77: Three Peripheral Groups

### A.2.2 Redundancy

The central processor contains two identical planes, viz. plane 0 and plane 1 (figure 76). This is to improve the reliability and availability of the switching system. Each plane consists of a microprocessor (called central controller (CC)), disk controllers, a memory and communication hardware. One of the planes performs the operational software tasks while the other performs tests in cold stand-by mode. When the operational plane fails, the planes are interchanged with only a short down-time. Three processor control switches in triple modular redundancy decide whether the planes must be interchanged. The decision is taken via majority voting upon special messages from the processor planes. The two planes share a collection of common units consisting of the processor control switches, the real-time clock, the disk units and the terminal control units.

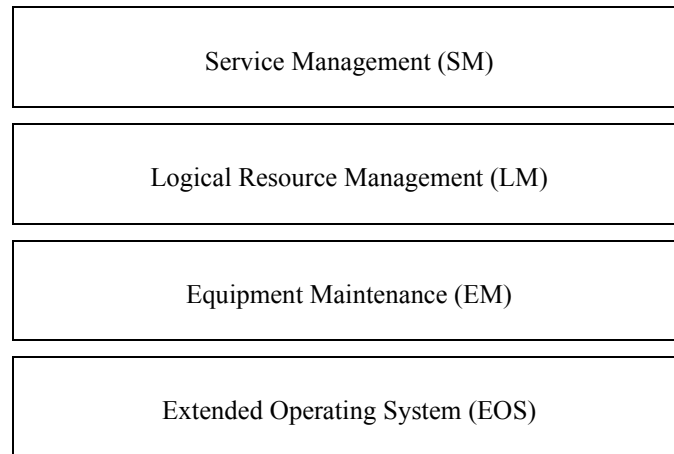
All common units are doubled in the central processor. The common units have connections to both processor planes. The real-time clocks are synchronised and they can be connected to an external reference clock. This external clock may be the synchronisation clock of the switching matrix or a clock transmitted through a radio connection.

Part of the switching matrix, viz. the part dealing with time stage switching, has triple modular redundancy. Voting upon output data is used to detect and correct errors. The synchronisation clock within the switching matrix is doubled. The clocks are synchronised with a reference clock obtained from the network itself. In special cases, e.g. during testing, the synchronisation clock may run in free mode, i.e. without a reference clock. The remainder of the switching matrix does not have hardware redundancy.

Service functions in the peripheral groups have redundancy in order to enable graceful degradation after failures. The trunk cards have redundancy through the configuration of the network. Alternative service function cards and trunk cards, which are connected to alternative trunks, are connected to different cards of the switching matrix. There is no redundancy of the subscriber cards.

Of all of the SW in the CC only some part of the extended operating system and the handling of the message channels between central processor and the switching matrix are aware of the HW redundancy. The active plane runs the *actual* SW while stand-by plane runs test SW.

### A.2.3 Software Architecture



*Figure 78: Layered Subsystems*

The software of the tss systems is based on the hardware shown in figure 76 and figure 77. The software of the tss system is layered as shown in Figure 78. Note that the term subsystem is used here in the sense of horizontal subsystems. They are the extended operating system, equipment maintenance, logical resource management and service management. Each subsystem uses only functionalities in lower-layer subsystems.

The subsystems are distributed over the entire hardware configuration tree. This is shown in Figure 79. Layered peer-to-peer communication (section 7.4.3.4) crosses hardware boundaries. The figure also shows the internal structure of the equipment maintenance subsystem. The equipment maintenance subsystem of the CC mirrors its controlling hardware configuration. The same holds for the equipment maintenance of the PGC. The figure shows PUs which have lines, e.g. network interface cards and subscriber cards.

---

## A.3 The SW Architecture of the CC

The BBM is applied in the SW design for the central controller (CC) of the tss systems. The peripheral SW is designed using compile-time modules only. This section describes the application of the different methodical steps of the BBM.

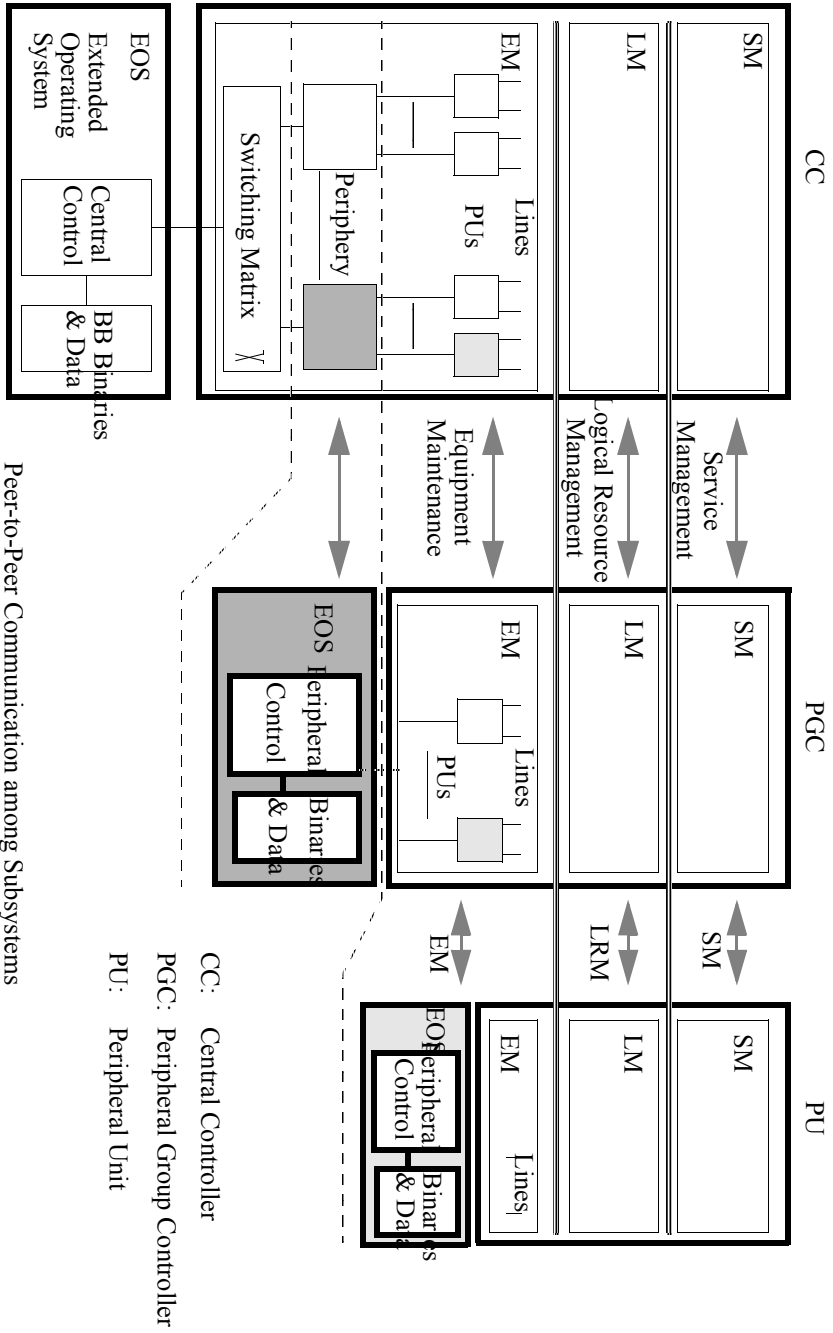


Figure 79: Peer-To-Peer Communication

### A.3.1 Object Design

The SW architecture of the CC is based on object modelling of the application domain, of the peripheral units (PU) and of the hardware of the CC (figure 80). Layers are used for identification of objects. The layers serve the purpose of realizing stability in the object structure. Layers of managed objects of the application domain (SM + LM) are built on top of a layer of managed objects reflecting the hardware.

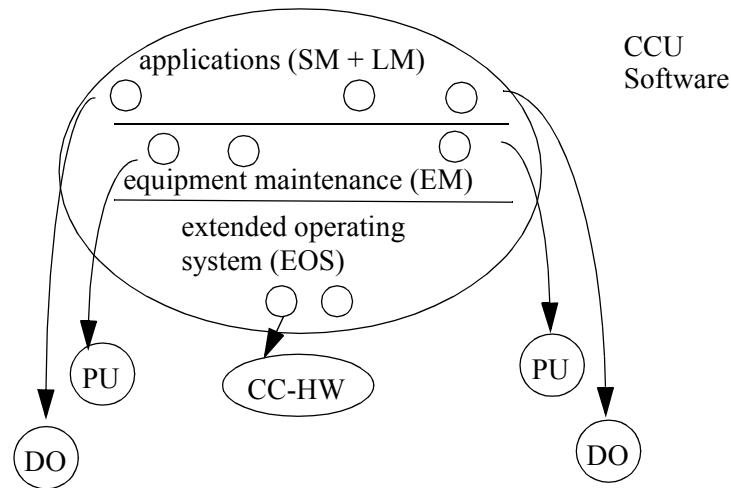


Figure 80: Mapping of Objects to Layers

The object model is on the one hand used as a basis for the design of the BBs, on the other hand it is used to determine configuration data (see section A.4.3). Instances of objects are kept in instance tables that can be viewed and updated from the system management interface. For a more systematic introduction of object design see chapter 4.

### A.3.2 Functionality of the CC Subsystems

The CC software is structured into four layered subsystems and consists of the following functionality:

The extended operating system (EOS) contains amongst others the run time kernel, timer services, exception handling, the data base, recovery mechanisms, the BB binary administration, the message transfer service to and from the peripheral areas, file handling and memory management of the central controller.

The equipment maintenance (EM) subsystem consists of the control layer for the switching matrix, the peripheral hardware and its interconnection structure which is controlled by the central hardware. It handles aspects of e.g. data distribution, recovery of peripheral software and fault handling of the peripheral hardware. Together with the extended operating system it forms the distributed operating infrastructure that constitutes the basis for the applications.

The logical resource management (LRM) subsystem provides a management layer of logical resources for the higher-level subsystem. Logical resources can be roughly divided into two classes. The first consists of abstractions of hardware objects and constitutes the basis on which the application processing is performed. The second class has to do with non-hardware-related logical objects, for example data for signalling, logical lines, and facility data.

The service-management subsystem comprises all the services of the application, that is, it handles calls, call signalling and call facilities such as call forwarding and automatic call distribution.

### A.3.3 Aspect Design

The *actual* or operational functionality of a telecommunications system is usually only a small part of the total functionality (ca. 20%). Other parts of the functionality deal with e.g. service of the system, or with fault handling. Aspects are those functionality which crosscuts (almost) all (hardware and domain) objects. The functionality of the product family is also structured according to *aspects*. For a more systematic introduction of aspects see chapter 5.

#### A.3.3.1 tss SW Aspects

We shall give the list of SW aspects of the tss system [Bau95]. These aspects are either designed to meet the requirements or a consequence of the system architecture (see section 10.1.1 and section A.2):

##### *system management interfacing*

The system management interfacing aspect concerns the system's external control. This may be a man-machine interface including formatted input and output or a message-based interface.

The system management interfacing aspect is a consequence of the requirements that most of the objects need to present specifically formatted information via the system management interface and accept input from it.

##### *recovery*

The recovery aspect relates to the system's proper initialisation. A recovery is initiated by either an operator or the error handling aspect.

The recovery aspect will be present in most product families; recovery is complex enough to factor functionality out into a recovery aspect.

#### *configuration control*

The configuration control aspect relates to the impact of changes in the physical (hardware) and/or logical configuration. Changes may be induced by failures or reconfigurations via system management.

The configuration control aspect constitutes the software implementation of the configuration management system aspect (see section 5.2.2).

#### *data replication*

The data replication aspect relates to the replication of data across processor boundaries. Configuration data from the central controller (controlling equipment) are replicated whenever a peripheral device (controlled equipment) requires a local copy of part of the configuration data.

The data replication aspect is an example of a software aspect which is a consequence of the distributed system architecture.

#### *test handling*

The test handling aspect comprises built-in functions that either run periodically or are invoked following specific events in order to detect and identify internal or external hardware faults or corrupted data. Test functions do not generate a resulting event unless to indicate a fault.

The test handling aspect is a consequence of the fact that generic test functions, which could be localised somewhere in the system, are not sufficient. Many of the objects require their own test functions.

#### *error handling*

The error handling aspect is accessed when a failure occurs. The functions concerned take the appropriate actions following a failure. In particular this includes damage confinement and fault localisation.

The error handling aspect constitutes the software implementation of the fault management system aspect (see section 5.2.2).

#### *diagnostics*

Functions of the diagnostics aspect are invoked by

- test handling, as part of preventive maintenance, in order to detect hidden faults,
- error handling in order to localise hardware faults,
- configuration control in order to verify the repair or correct re-configuration of physical or logical objects.

The diagnosis aspect is a consequence of the fact that generic diagnostic functions, which could be localised somewhere in the system, are not sufficient. Many of the objects require their own diagnostic functions.

#### *performance observation*

The performance observation aspect relates to the collection and processing of data for statistics and quality measurement purposes.

The performance observation aspect constitutes the software part of the performance management system aspect (see section 5.2.2).

#### *debugging*

The debugging aspect covers the functions required to interactively debug the on-line software in test-floor operation as well as field operation.

The debugging aspect is a consequence of the fact that generic debugging support which could be localised somewhere in the system is not sufficient. Many objects must provide their own functionality for debugging.

#### *overload control*

The overload control aspect implements the functionality to prevent the system from malfunctioning in overload conditions. If in a situation external requests are exceeding system capabilities, the system will internally still be within the margins of the specified quality of service.

The overload control aspect is a consequence of the fact that different objects need to take different measures to defend themselves from exceeding requests.

#### *operational*

The operational aspect has a specific character. It represents the system's functional behaviour, i.e. in the case of a switching system the ability to handle calls.

The tss software aspects make reference to the system management functional areas (see section 5.2.2). However, the areas accounting management and security management are not regarded as aspects. The reason is that accounting management and security management can be localised in functional packages. Accounting management is implemented as a set of BBs in the logical resource management layer and security management is implemented generically via login procedures and user profiles in the operation and maintenance terminals.

### **A.3.3.2 The System Quality Reliability in tss**

The implementation of the system quality reliability in tss is given (see appendix A.2.2) to illustrate a more complex mapping of a system quality. Reliability is realised in the following ways:

not in software:

- the central processor is duplicated with one of them operating in cold stand-by mode;

- the switching matrix executes in triple modular redundancy with majority voting,

- the following holds for the three classes of peripheral cards, subscriber cards, trunk cards and service cards:

  - specific service cards are configured in load sharing or hot stand-by for dynamically allocated resources,

  - redundancy of trunk cards is handled by the network of switching systems which has alternative routes configured in case one fails.

  - subscriber cards are not redundant

in software:

- by the system management interfacing software aspect: changes in the card configuration, states of card and logical objects and parameters thereof made by the operator occur either completely or not at all (transactions and roll-back),

- by the configuration control software aspect: persistence of the configuration is realised in a database,

- by the error handling software aspect: fault management concepts are implemented for card faults to maintain the system in a consistent state.

### A.3.3.3 The tss State Model

We present the tss state model as an example of functionality of a software aspect. It is the core part of the configuration control aspect, which is used for managing physical and logical resources. The state model is small and distinguishes between persistent and dynamic states. Persistent states survive system crashes by being handled by the system configuration database. Dynamic states are reset after a crash. All managed objects in the system controller implement this state model.

The state model consists of three states taken from the ITU standard [X731]: the administrative state, the operational state and the usage state.

The operational state reflects the actual state of a real resource, for instance a hardware board. The usage state applies to a resource which is used by other resources. The operational and usage states have dynamic values since they reflect the dynamic state of equipment or other resources. The administrative state reflects actions of the operator and has, therefore, persistent values. The three states have the following values (figure 81):

administrative state: *not managed, locked, unlocked, broken*

The actual names used by tss are: not installed, out of service, in service and error, respectively.

operational state: *disabled, enabled*

usage state: *idle, active, busy*

The administrative state values have the following meaning:

The value *not managed* means that a data item representing a physical device is present in the system controller data base, but no action is taken to connect it to the actual equipment.

The value *locked* means that the equipment should be completely prepared to start its operation and already supervise for errors. Operation, however, may not be started.

The value *unlocked* means that the equipment starts its operation or is in operation.

The value *broken* means that an error in the equipment cannot be corrected by the system itself. Without intervention of the operator the system does not do anything with the failed equipment.

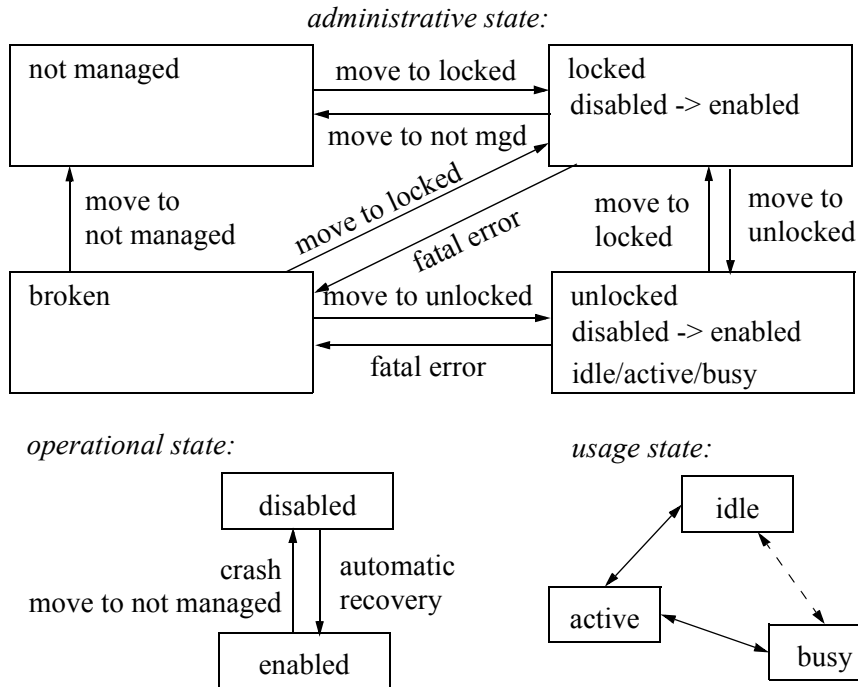


Figure 81: tss State Model

Connecting operational state with administrative state is straightforward. If a device has the value *locked* or *unlocked* in the administrative state, the system will automatically try to bring the device to the operational state *enabled* via an automatic recovery. A crash will bring the device in the operational state *disabled*, an operator command *move to not managed* will also bring the device to the operational state *disabled*.

The values *not managed* and *broken* of the administrative state are extensions to the standard for allowing good coupling of administrative and operational states. The operational state always has the value *disabled* if the administrative state has either the value *not managed* or *broken*.

The usage state shows whether a resource is idle or in use. The *active* value means that a resource is partially used, while the *busy* state indicates full use. A resource which does not provide for multiple usage does not have an *active* value. The transitions are made by allocating and releasing resources. As shown in figure 81, the usage state only has meaning when a system is *unlocked*.

#### A.3.3.4 The tss Recovery Model

We shall now describe the tss recovery model as a further example. It is the core of the recovery aspect. The recovery model describes the standardised part of the recovery aspect.

A simple recovery model distinguishes only between a recovery phase and an operational phase. During recovery, equipment is started and configuration parameters are initialised with values from a configuration database. Operation starts when the whole system is initialised.

The tss recovery model is an extension of this simple model. It is based on two concepts: recovery levels and recovery phases. The intention is to make operational actions very efficient and perform initialisation and preparation actions during recovery only.

#### Recovery Levels

Recovery of the central processor is realised according to a specific recovery level. Recovery levels define the actions taken to bring the system (back) into operation. There are two *types* of recovery levels:

*System-initiated recovery levels* are recovery levels which are set by the system itself to perform a recovery. They comprise the recovery levels: *restart without persistent data reload*, *restart with persistent data reload* and *reload*. System-initiated recovery levels are initiated after a recoverable error has been detected in the system.

*Operator-initiated recovery levels* are recovery levels set by an operator. They comprise the recovery levels *new version load*, *new version load with new DB scheme* and *initial load*.

The following *recovery levels* are defined in terms of the affected classes of data: dynamic data, persistent data or binaries.

*Restart (or restart without persistent data reload):*

This is the weakest recovery level. The operational work is stopped and dynamic data are re-initialised. Then the application processes are started and operational work resumes.

*Reconstruction (or restart with persistent data reload):*

In the reconstruction recovery level, the persistent data are reconstructed. This is done by loading the data base tables from persistent back-up memory to the in-memory database and re-executing the logged data base transac-

tions. Then, dynamic data are initialised, application processes are started and operational work resumes.

*Reload:*

Hard data (BB binaries) are restored from the system disk and the actions of the *reconstruction* recovery level are executed.

*New version load:*

A new version of BB binaries is loaded from the system disk, and the actions of the *reconstruction* recovery level are executed.

*New version load with new DB scheme:*

The back-up database is updated with the logged database transactions. A new version of BB binaries is loaded from the system disk, dynamic data are initialised and persistent data from the previous version are transformed into the new data base scheme. Then the actions of the *reconstruction* recovery level are executed.

*Initial load:*

The binaries are loaded from the floppy disk, dynamic and persistent data are initialised. A minimum set of persistent data necessary for the system to be able to run consistently, is provided. Journalling files are created. This recovery level is intended to be executed only once in a system's life time because it initialises all data including journalling files.

## **Recovery Phases**

The recovery levels are implemented by a number of recovery phases. Each recovery level is an ordered selection from the set of all the recovery phases. The phases have been standardised for all the BBs. Each BB can provide methods for each recovery phase. The recovery phases constitute the second step in a series of three recovery steps:

The first step is the loading of a BB executable into memory. This step is performed only if necessary for the recovery level.

The second step is based on the loaded BBs. Each BB provides a set of initialisation methods for the recovery phases. Each phase has its own semantics, which the recovery methods may not violate (see below). These methods are executed in the predefined order of recovery phases (see below). The actions of the recovery phases comprise all the recovery actions of the BBs.

The last step is the starting of the application processes. This step is implemented as the last recovery phase and starts the running of the system, i.e. it is a transition from recovery to operation.

The recovery methods of a BB must ensure that a proper recovery is performed for each recovery level. The recovery phases are ordered in a specific sequence. During system recovery the recovery methods of one recovery phase are executed for all the BBs before the recovery methods of the next phase are executed. It is important to note that a recovery method of a specific BB may rely on the initialisation of another BB, which therefore must be done in an earlier phase.

The tss recovery model allows a BB to be loaded incrementally, but the recovery phases are executed for all BBs. Extending the recovery model to allow new applications to be loaded while the system is running requires a different strategy. The recovery phases would be executed per BB, except the last phase, i.e. the starting of the application processes. BBs may in their initialisation methods rely on the fact that recovery methods of BBs of lower layers have been executed. After all the BBs have been initialised, the application processes are started. To be able to extend a running system it must be cared for that no process enters the code of an initialising BB before initialisation is finished. It is more difficult to reduce a running system because all resources held by the respective BBs have to be freed and there may be no running processes inside these BBs, nor may other BBs depend on them.

The execution sequence of initialisation methods in one recovery phase is arbitrary. Each recovery method is executed only once during a recovery. The meanings of the different recovery phases are as follows:

*Post Load Bind phase (PLB):*

Binding of references (linking) between BB binaries during recovery.

*TeRminate Bind phase (TRB):*

TRB is inverse to the PLB phase; it is used for unbinding and is intended for the unloading of single BBs while the system is operational.

The term *binding* is used for registration of call-back methods during recovery (see section 7.2.4)

*Initialisation of dynamic data (Dat0):*

This phase is equivalent to the data initialisation realised by the runtime system usually generated by the compiler. The tss system did not rely on this feature of the compiler.

*Dat1:*

Initialisation of persistent data to *non-existent* in the sense of database management.

*recovery levels*

	<i>mnemonic</i>	<i>restart</i>	<i>reconstruction</i>	<i>reload</i>	<i>new version load</i>	<i>new ver. load w. new DB</i>	<i>initial load</i>
<i>execution order</i> ↓	<i>TRB</i>			●	●	●	●
	<i>PLB</i>			●	●	●	●
	<i>Dat0</i>	●	●	●	●	●	●
	<i>Dat1</i>		●	●	●	●	●
	<i>Dat2</i>	●	●	●	●	●	●
	<i>NewDB</i>					●	
	<i>DBRec</i>		●	●	●	●	
	<i>ILD</i>						●
	<i>Dat3</i>	●	●	●	●	●	●
	<i>ProcAct</i>	●	●	●	●	●	●

Figure 82: Recovery Phase Hierarchy

This is necessary for the proper functioning of the DB handler's persistent data reconstruction mechanism.

*Dat2:*

Initialisation of dynamic data similar to *Dat0*, but to be used if expression evaluation requires values generated during the execution of *Dat0* and/or *Dat1*.

*New DB scheme (NewDB):*

The functions of this phase are executed in a new version load with a new DB scheme. Examples are new tables filled with default values. DB transforma-

tions from an old database scheme to the new one are executed. BBs with files perform possible file transformations.

*Reconstruction of the data base (DBRec):*

Tables of the in-memory database of BBs are reconstructed.

*Initial Load phase (ILD):*

The functions of this phase are executed only in an initial load. An example is the creation of journalling files. Another action is the construction of a minimal version of the database in memory. An initial load deletes all history information and therefore all actions which have to be performed only once are to be executed in the initial load phase.

*Dat3:*

Initialisation of dynamic data similar to Dat2. To be used if expression evaluation requires persistent data values generated during DBRec or ILD execution. Examples are transformations of tables for fast access.

*Process Activation phase (ProcAct):*

This is the last recovery phase. After the execution of this phase the system is recovered.

Figure 82 shows the recovery phases executed per recovery level.

### A.3.4 Concurrency Design

Concurrency design maps the functionality of a system to processes and threads to allow for parallel execution and timeliness. This section describes the concurrency design of the tss systems. For a more systematic treatment see section 6.2.

The concurrency design of tss differentiates between thread types (called process) and thread instances and is guided by the following principles:

autonomous external behaviour is represented by a thread internally, e.g. each operator session and each call has its own thread instance;

each communication channel arriving at a hardware unit is handled by one thread instance;

tasks which have different priorities are represented by thread types for each priority;

sets of tasks that require fixed amounts of time and should not be delayed because of processor contention are implemented on different real or virtual processors.

These principles lead to few thread types and relatively large numbers of process instances. Programming of process synchronisation is limited while the dynamic behaviour profits from the number of independent working process instances. Design of the process types can be carried out as if working in a single instance environment.

One very important fact is the independence of object design and concurrency design. The designer is free to carry out the concurrency design without consequences the other dimensions.

The guidelines for the concurrency design as given above actually are an extension to the MO view of the world (see section 10.2.1): MOs reflect *static* physical or logical entities of the real world (with types and instances). The processes as implicitly defined above reflect *dynamic* entities of the real world (human beings like operators or speaking subscribers or microprocessors communicating via message channels) also with types and instances.

The system infrastructure generic *Process Management* (see section 7.5.4) deals with processes. Any Building Block containing threads is a specific of Process Management. Process Management supports the concept of a virtual processor [LM97] called thread category in tss.

A thread category realises a virtual processor with a certain percentage of processor time. A thread category has priorities. Two important attributes of a thread are its category and its priority. There are four different thread categories. A main category, an MMI category, a debugging category and a background category. Almost all threads are allocated to the main category. Threads for system management interfacing are allocated to the MMI category, the debugging threads are allocated to the debugging category and a background thread might be allocated to the background category. All categories are configured to receive 10% of the real processor time, except the main category which receives 70%. In the case of underflow, i.e. no ready to execute thread in a category, the processor time is given to the main category. System management interfacing threads and debugging thread execute with predictable performance independent of system load. A background thread which needs to make progress even in a maximum load situation is allocated to the background category, all other background threads are allocated to the main category.

Basically, there are the following mechanisms applied in the tss system to ensure data consistency for concurrent data accesses:

- critical regions for real-time critical accesses, and
- transactions for data base modifications

The concurrency design of the entire application functionality (Equipment Maintenance, Logical Resource Management, and Service Management) is based on one address space and comprises six process types.

tss service management performs call processing. A single thread instance receives the call-initiating messages from the peripheral cards. A call thread is started to handle a new call. The call thread type has instances for the maximum number of parallel calls allowed in a system, e.g. several thousands.

tss equipment management and logical resource management perform control processing in two thread types. They go along aspects and cross many objects. A fault handler covers the operational, recovery and fault management aspects, while a configuration handler covers configuration control and data replication. A separate thread instance is used for each equipment instance at the peripheral bus.

A further thread type covers the system management interfacing aspect of all three layers. It is instantiated per user instance.

The entire incoming and outgoing communication is handled via one central BB that handles the bus connection. A singleton thread type covers the incoming direction, while for the outgoing direction functions are provided which run under the budget of the sending process. The incoming messages are distributed from this single process to the process representing equipment instances. It polls an incoming message buffer and thus determines the speed of the internal processing. The respective BB is part of the extended operating system.

### A.3.5 Building Block Design

Building Blocks are software components (see chapter 7). This section explains BBs of tss from a technical point of view. For a more systematic treatment see chapter 7.

BBs usually follow the object dimension, that is, a BB is a cluster of objects. However, this is no restriction. A BB can follow the other dimensions as well, or even encapsulate arbitrary parts of the three design dimensions. The main criteria are configurability and incremental integration, as will be outlined in this section.

A BB is a unit of design and deployment. It consists of *provides* and *requires* interfaces (see section 7.2). The dependency relation between BBs is uni-directional (see section 7.4). To allow communication from the *lower* BB to the

*higher* BB, the *higher* BB registers call-back methods with the *lower* BB (see section 7.2.4). Uni-directional layering is used as a basis for incremental integrability and incremental testability of the system. Each BB is designed such that contains sufficient behaviour to be tested independently. Furthermore, generic functionality of the product family is separated from specific functionality of particular products and embodied in generic and specific BBs, that is, component frameworks and plug-in components (see section 7.5).

#### **A.3.5.1 The tss Component Model**

The component model of the tss system comprises the component identity of BBs and the way in which interfaces of a BB are accessed.

##### **BB Access**

Inter-BB calling of tss always uses just one indirection. In the case of the call of the service interface the indirection is in a BB descriptor (see below). For call-backs the indirection takes place via a procedure variable (see below).

The tss component model works without a system registry. A BB has a description of all other BBs to access. References to BBs which may vary for different installations are always handled by a generic BB (see below). References to exported procedures in the BB descriptor are linked statically. The provides interfaces of used BBs are also known. The only place where knowledge about the entire BB configuration is present is the configuration file of the BB loader. The BB loader is responsible for loading the BBs from disk into the memory.

Interfaces are not separate units, they are part of BBs. Interfaces which are independent of BBs may be described at the design level but not at the implementation level.

##### **The tss BB Identity**

The BB identity used within the tss system is closely linked to the memory addressing scheme. The central processor uses a 32 bit addressing scheme. The resulting virtual memory address space of 4 GB is translated by the MMU to the physical memory. BBs are statically located in the virtual address space. Each BB is given a unique number during design to enable its identification. This BB number is mapped to a fixed memory location in the virtual address space. The linker uses the BB number to resolve all BB internal addresses. The loader programs the MMU before it transfers a BB from the disk into the RAM.

The 32 bits of an address are interpreted as follows (see figure 83):

10 bits for the BB number

4 bits for the BB relative sections

18 bits for the relative segments.

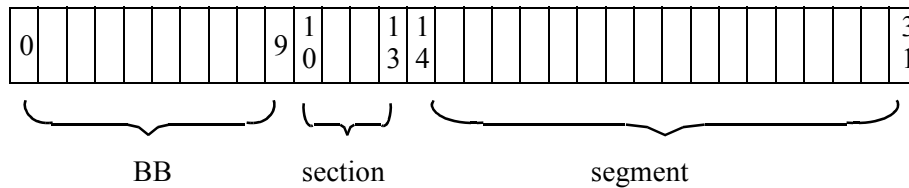


Figure 83: tss Addressing Scheme

A tss system can consequently handle a maximum of 1024 BBs. Since a typical product family had less than 1024 BBs, the BB number was assigned to the entire product family. The address space of 22 bits per BB was also sufficient. If necessary, a more dynamic scheme could have been used to allow a maximum of 1024 BBs per product. The sections are used for different types of *data*, such as the BB descriptor (see below), code, persistent data or dynamic data.

### The Provides Interface

A BB is accessed from other BBs through its provides interface, which is described in the BB descriptor. The BB descriptor contains a table of exported procedures of a BB (figure 84). The BB descriptor is generated during linking and is placed in a specific section of a BB. If the interface of a BB does not change, or is conservatively extended, a using BB will be left unchanged. A new implementation of a BB will be loaded without recompiling the using BB. Many of the errors in a deployed system requiring a SW update can be handled via this mechanism. If the order of procedures in the descriptor or parameters of procedures change, the using BBs will have to be updated.

Procedure calls from outside BBs are indirected via the BB descriptor (Figure 84). A calling BB is compiled and linked to the descriptor of the called BB. The actual address of the called procedure is determined at runtime. (This is comparable to a C++ virtual table call.) Each exported procedure is given a number. Besides the signatures of the exported procedures, the export specification of a BB contains their assigned numbers. To be able to optimise BB internal code, the compiler needs to know whether a call is a local call or a call to another BB. Table 7 shows the translation scheme for both situations.

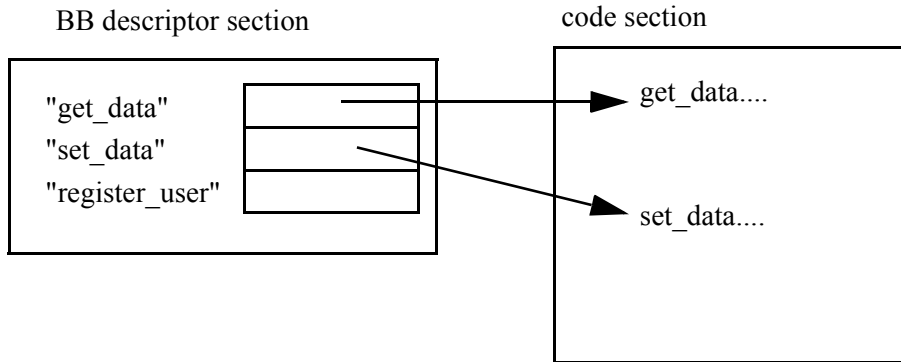


Figure 84: Service Interface of a BB descriptor

	BB internal procedure call	BB external procedure call
source code	get_data (...);	get_data (...);
assembler code	"move parameters to stack"; jsr get_data;	"move parameters to stack"; move get_data A jsr A;

Table 7: Procedure Call Translation Scheme

### Up-Calls

BBs are restricted to have only unidirectional relations. A BB importing a provides interface of a second BB is said to reside in a higher layer (see section 7.4). A call-back is always a call to a procedure in a BB of a higher layer. These procedures are called up-call procedures and are registered with the lower-layer BB (figure 85). In the implementation, arrays are used to store procedure references.

These arrays with procedure references are statically allocated in the tss component model. This poses two types of restrictions: first, the number of different BBs that can bind themselves to the generic BB is fixed at compile-time, and, secondly, BBs cannot be used by more than one application (cf. dynamic link library). This can be avoided by using dynamic data structures to store procedure references.



### Recovery Interface

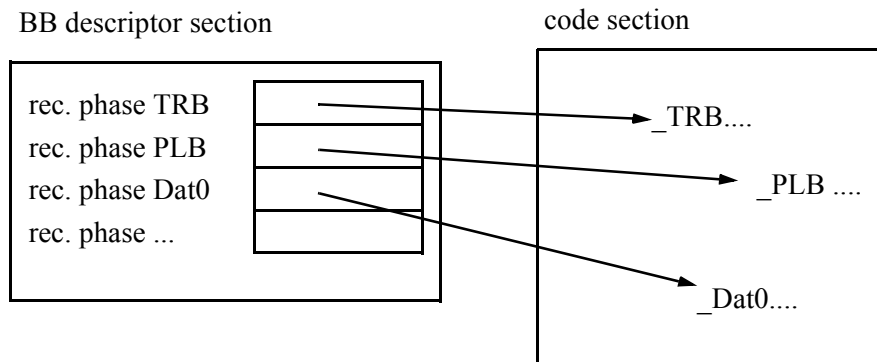


Figure 86: Recovery Interface of the BB descriptor

A BB is recovered by the recovery manager, a low-level BB of the extended operating system. After loading a BB, the recovery manager will activate the initialisation procedures of the BB. (For the tss recovery model see section A.3.3.4). Since recovery is the first activity of a BB, the BB cannot register its recovery procedures with the recovery manager. The solution chosen is to put the recovery procedures in the BB descriptor (figure 86). The recovery manager receives the BB number of a loaded BB from the loader and accesses the recovery procedures at their pre-defined location. One recovery phase is the bind phase in which registration procedures are executed to establish a binding between BBs.

A possible extension of the BB descriptor is to generally support other infrastructure BBs (so-called SIGs; see below) as well. Most BBs need to register with them anyway. Call-backs from these infrastructure BBs would rely only on the BB number and would find the appropriate procedures in the BB descriptor. This reduces the code size of these infrastructure BBs by making the table with call-back references superfluous.

### Implementation Languages

The tss system was programmed using three different programming languages. Most of the system was programmed in C. The call processing software was programmed using the ITU programming language CHILL. Core parts of the extended operating system were programmed in Assembler. Compilers for these three languages were adapted to support the tss component model.

### A.3.6 Generics and Specifics

There are several types of generic BBs (component frameworks) in the tss systems (see section 7.5.4). Note that to be a generic is a role of a BB. It is perfectly possible for a BB to have several (different) generic roles.

#### A.3.6.1 Abstraction Generics

An *abstraction generic* shields the differences of its specifics by providing a common interface towards the users of the generic. In addition, it implements common functionality. Figure 87 is a picture of an abstraction generic and its specifics. A use interface of the generic is used by the specifics to access the common functionality provided by the generic.

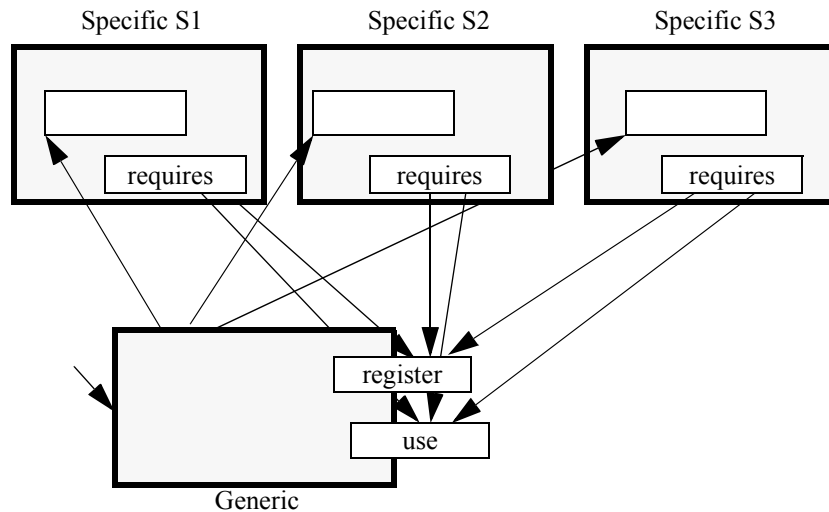


Figure 87: Generic and Specifics with Interfaces

Examples of abstraction generic include file handling, I/O device handling, digit analysis, call handling, etc. File handling includes the operations open, close, write and read. Specifics of file handling have specialised handlers for different file types such as sequential or indexed files. I/O device handling includes the operations mount, dismount, assign and de-assign. For the handling of different device types such as PCs, printers and disks I/O device handling has separate specific. Digit analysis has as input digit strings and specifics for different destination types such as analog or digital subscribers or other switching systems. Call handling contains the basic call model and is extendable towards different signalling handlers.

### A.3.6.2 System Infrastructure Generics

*System infrastructure generics* build an operating infrastructure for all Building Blocks in the system, i.e. all Building Blocks are potential specifics of them. System infrastructure generics administer system resources such as processor time, memory usage and standardise system functionality such as the man-machine interface (MMI) supporting generics. Since these generics offer very generic functionality, they are located low in the system, i.e. they are part of the operating system.

One characteristic of system infrastructure generics is that their specific functionality is parametrised by data. Thus all the algorithmic parts reside in the generics themselves and the specific parameters reside in the specifics. Figure 88 shows a Building Block together with three system infrastructure generics (SIGs). Despite the fact that the Building Blocks contain the data for a system infrastructure generic, they access such data through the system infrastructure generic.

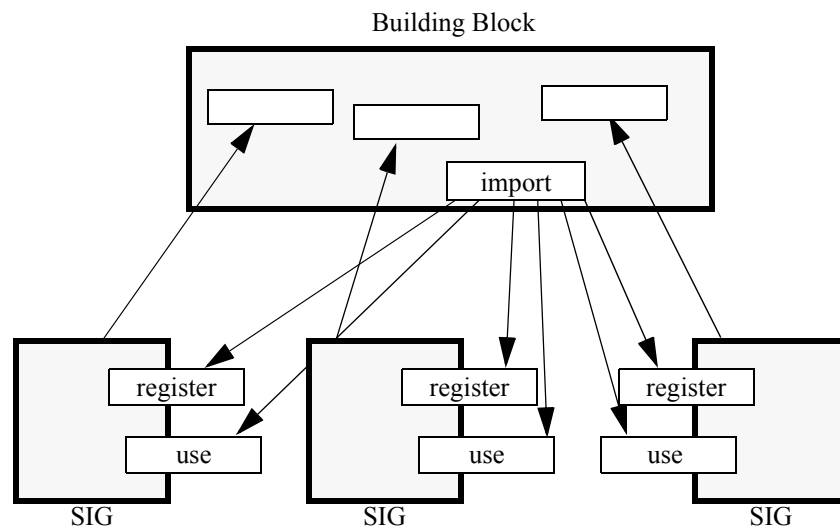


Figure 88: System Infrastructure Generics

One example is the *Exception Handling* generic. This generic defines the standardised functionality such as possible severity of the exceptions, whereas the actual severity, output formats and texts for the man-machine interface are located in the specific Building Block.

Using a generic to implement such a system function, as e.g. exception handling is an alternative to a case statement where each case alternative represents a specific instance of an exception. Maintaining lists of case alternatives for an evolving system leads to frequent changes in the source code of the exception handler. This is a potential source of errors. In contrast to this a generic is not changed. The list of different exceptions in a specific product is automatically built by the configuration of selected Building Blocks in the product.

The use of the system infrastructure generic reduces the code of the specific Building Blocks considerably. Furthermore, this standardisation allows code generation. There is a data definition database (DDD) tool where the specific data of the system infrastructure generics are globally administered. Because of their relevance of the whole system these data can be easily reviewed and changed without going into the code of the system. Furthermore, the DDD provides a variety of backend generators supporting the code generation, customer manual creation and the product configuration processes. Part of the production of a Building Block is the code generation for all relevant system infrastructure generics. An implementer of a Building Block may not even be aware that some of the system infrastructure generic data of his Building Block is part of the Building Block. (See also section A.4.2)

Examples of other SIGs are process management to handle threads, memory pool handling, data base handling for persistent handling of instance data, the message transfer system to send and receive messages from the peripheral units, operator report handling, recovery handling (section A.3.3.4) and management interface string handling

### A.3.6.3 Resource Generics

A resource generic is a generic designed to encapsulate a pattern that models a static communication path between Building Blocks. Within the control software it reflects the existence of a hardware bus, connector or plug. A resource generic (figure 89) administers the plugs or the slots of the bus. Thus, if a hardware module controls a bus and another is plugged into the bus, their controlling software modules communicate via the resource generic. The Bus Generic in figure 92 is an example of a resource generic.

A resource generic guards the supply and allocation schemes of so-called *abstract resources*, e.g. the bus slots. Physical resources, such as printed circuit boards and a logical resource, such as a line, are to be distinguished from abstract resources.

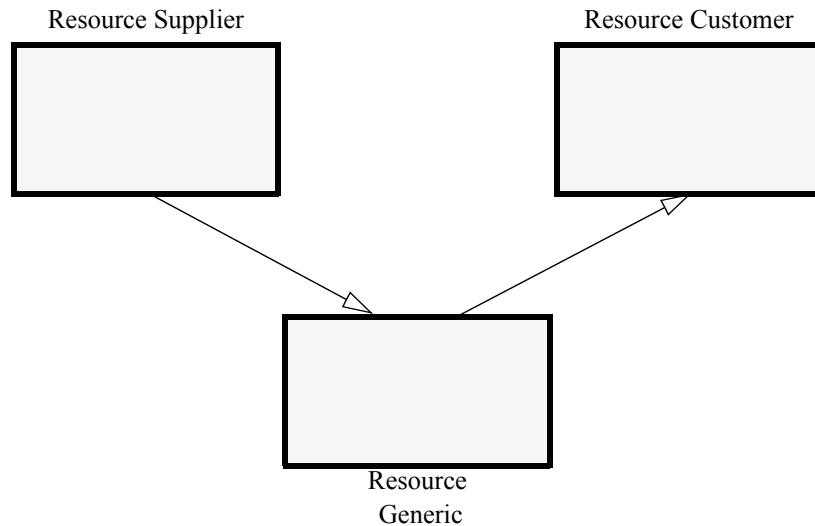


Figure 89: Connectable Resource Generic and Resource Flow

A resource generic has two types of specifics, viz. *suppliers* and *customers*; see figure 89. The suppliers supply the (abstract) resources, whereas the customers *allocate* these resources. The resources are said to flow from the supplier to the customer via the resource generic. The generic reduces the mutual knowledge of suppliers and customers to that of a resource. Supplier and customer Building Blocks know only their respective interfaces of the generic. Note that there is a bidirectional communication flow between suppliers and customers, mediated through the resource generic.

The resource generic standardises parts of the supplier interface and parts of the customer interface. Additionally it provides an internal resource administration. Besides this standardisation, each resource generic provides abstract interfaces for sending information from the supplier to the customer and vice versa. The resource generic does not actually deal with the information sent through the connection. An example is a configuration data change in a customer Building Block (see example in section A.3.8). Such a change has to be communicated to the controlled hardware component. The information goes from the customer via the resource generic to the supplier. The supplier sends it via a communication path to its controlled equipment, which then in turn sends it via the physical connection to the controlled equipment of the customer.

Also in the other direction the resource generic mediates information. Suppose that the supplier receives the information that its controlled equipment is

not available for operation any more. Then it has to inform the customers that their equipment cannot be reached any more either. Thus makes it possible that controlling Building Blocks are not informed separately about failures via messages. Instead, one message is enough to signal the failure of a whole subtree. Note that this kind of communication direction is always a reversed one concerning the actual hardware and the mirrored software structure.

Since hardware buses and other plugs are quite common in the structuring of hardware, the standardisation of software handling in the form of resource generics is an example of a domain-specific design pattern [GHJ\*94].

The UPS Generic is a resource generic which handles the slots of the universal peripheral slot bus (UPS) (figure 92). The TS Generic is a resource generic which administers the timeslots of the peripheral module channel (PMC) (see figure 76 on page 198). The PMC Handling is implemented as a usual generic not making use pattern of a resource generic out of historic reasons. The CGR Generic is a resource generic dynamically administering pools of timeslots of the PMC statically allocated with the TS Generic. The PG Feature Generic administered call handling resources for PUs. Call handling applications could supply resources which PU administration BBs could allocate. The resources were then send by the PU administration BBs to the PUs.

**A.3.6.4 Layer Access Generic**

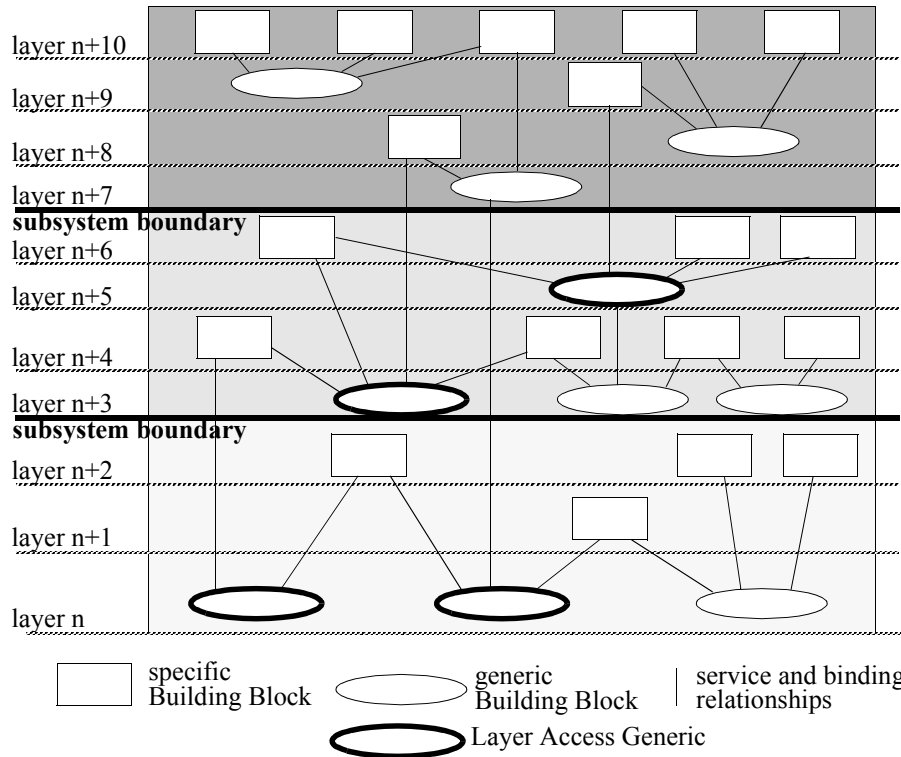


Figure 90: Layer Access Generics

The interfaces of the conceptual layers, i.e. the subsystems, consist of interfaces of the generics only; these *layer access generics* enhance the configurability within layers. All kinds of generics may act as layer interface generics. Changes within one layer that do not affect these generics keep the layer interface stable. Layer access generics are brokers for the subsystem functionality.

The interface between the equipment maintenance (EM) subsystem and the higher layer subsystems is build by the CCT Generic offering the allocation circuits of the PU to the line administration BBs, the TS Generic offering PMC timeslots to connect the circuits of the PU to the switching matrix, the CGR Generic offering pools of timeslots in case the timeslot allocation is to be determined dynamically and the PG Feature Generic allowing the supply of call handling resources for PUs.

### A.3.7 Self-Describing Components

All resource usage of a Building Block is defined by the Building Block itself, e.g. the use of processes, memory, etc. Resources are administered by system infrastructure generics of which a Building Block must be a specific if it needs the resources. Installing a new Building Block (or removing one) claims (frees) resources. A BB is self-describing.

As an example, the *Process Management* generic administers all processes of the system. A process is defined locally in a Building Block. The Process Management starts and handles the processes according to data such as category, priority, dispatch time or dynamic stack size, which is determined in the Building Block itself.

Consider the addition of some Building Block to the system. No recompilation, re-linking or reloading of the Process Management is necessary. Instead, the Building Blocks defines its own processes and makes them known to the Process Management via the bind (registration) mechanism described above. After adding the Building Block, the system will have an adapted list of processes.

There are thus no separate configuration files or global include files in the system. Instead, a BB which owns specific data registers it. Therefore, a BB registers itself with all relevant system infrastructure generic and with some of the other generic to be coupled to its direct operational context.

The only exception to the rule of having no global files is that there is a load set file that lists for a particular product all Building Blocks that have to be loaded.

### A.3.8 EM Layer Structure

As an example of applying the architectural concepts we describe the main structure of the equipment maintenance (EM) conceptual layer.

#### Control Structure

In this section we describe the concepts for dealing with hardware configurations. To be able to have a flexible hardware system, backplane bus systems are used throughout tss. The absence of closed communication loops (only star-type buses and tree-type hardware dependencies are used) reduces the overall system complexity. The central controller (CC) controls the complete system. This leads to a tree-type hardware dependency graph. Leaves of the graph are the peripheral

units (PU). Intermediate processing units are the Peripheral Group Controllers (PGCs); see figure 91. The products of the tss family have either 30 or 122 peripheral groups, many of them being PGCs. Such intermediate processing units are responsible for the monitoring of their subordinate hardware boards. The tree structure defines the controlling – controlled equipment relationship where peripheral group controllers have the double role of being controlled equipment and controlling equipment at the same time.

There are exceptional cases within a tss system which violate the pure tree structure and make it a directed graph, that is, equipment may be controlled by more than one controller. Therefore, the term *control hierarchy* is used instead of ‘control tree’. In these exceptional cases cables may connect two otherwise independent PUs. The logical connection between these PUs is visible as part of both PUs. This connection has to be administered by the controlling equipment. Explicit knowledge of the status of the connections is available in each piece of controlling equipment that oversees both end points, upwards in the control hierarchy.

As a consequence of the control structure the CC forms a processing bottleneck of the systems. Allocation of functionality takes this fact into account and all processing which has no coordinating character is allocated to the periphery.

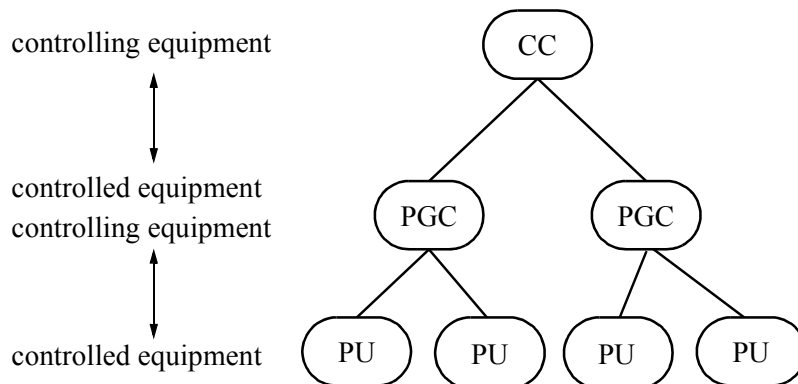


Figure 91: Tree-Type Control Structure

### EM Structuring

An important point in the equipment maintenance is the structure of the control software in the controlling equipment. Changes within a component low down in the control hierarchy have consequences for the control software of components higher up in the control hierarchy. When a card type is added or removed, the corresponding control software must be adapted to the new situation, e.g. by just

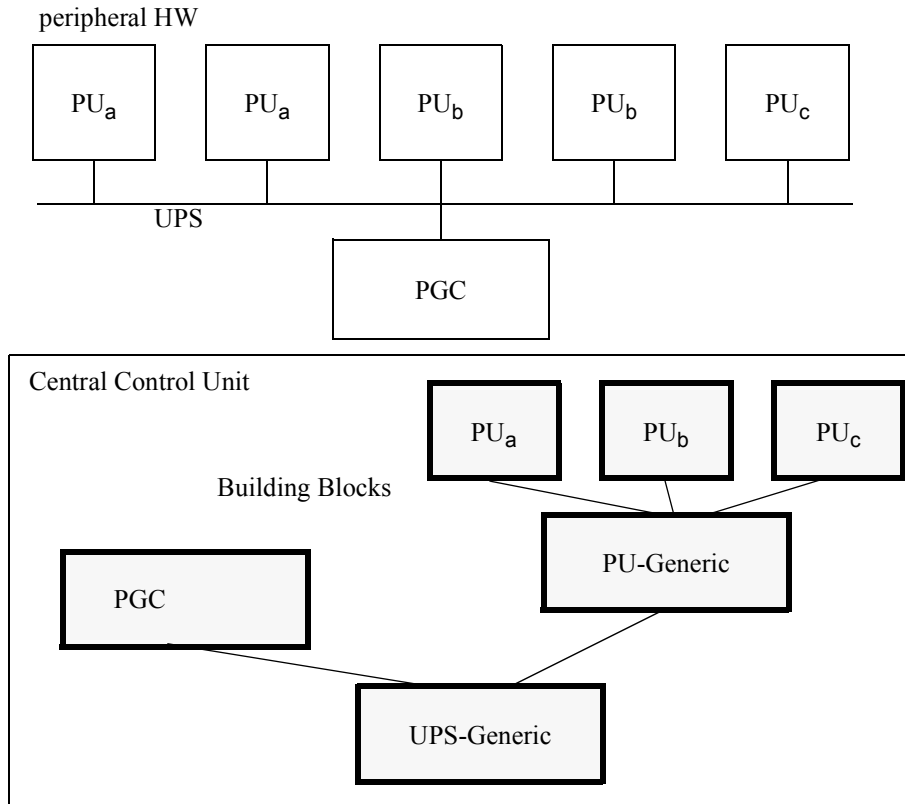


Figure 92: System Structure with HW Mirroring in EM

adding or removing Building Blocks. The configuration-to-minimum condition of the control software means that the control software contains only the software that is needed for the controlled hardware. This is supported by the concept, that the hardware topology is mirrored in software, see figure 92. Control software modules have the same interconnection structure as their controlled hardware counterparts. Each hardware module type, viz. PGC, PUA, PUb and PUC, is represented by one or more Building Blocks. Each level of hardware modules in the control hierarchy is represented by a generic control Building Block and a specific control Building Block for each hardware module type. For the situation of figure 92 there is a generic control Building Block, PU-Generic and corresponding specifics PUA, PUb and PUC. The PU-Generic, contains the standardised hardware handling, the specific control modules contain the specific deltas. Furthermore, each hardware bus is represented by a corresponding generic, the Bus-Generic. Instances of hardware modules are represented by data instances in

the instance tables of the corresponding Building Blocks. In particular, both  $PU_a$  and  $PU_b$  have two instances for the corresponding boards, and  $PU_c$  only one. In addition the PU-Generic has 5 entries for the same peripheral boards.

When a new hardware board is added, the software running on the board is added to the system as well. The remainder of the system has to be adapted to the new situation. This adaptation can be executed by adding a new control Building Block for the new hardware board within the controlling software. The configurability of the hardware thus imposes a corresponding requirement on the configurability of the software.

### A.3.9 The Use of Heuristics Within tss

In this section we give some comments about the application of the BBM heuristics in the tss product line.

List of Heuristics	tss Application Comment
<b>Heuristics of Object Design</b>	
<i>Heuristic 1: Use application domain objects and relations to generate an initial object model of the product family by mirroring them in the software system.</i>	The tss project did not do explicit domain modelling, however functionality of SM and LRM mirrors application domain objects.
<i>Heuristic 2: Remove objects, attributes and relations which do not describe required system functionality.</i>	see above
<i>Heuristic 3: Adapt the functionality of domain-induced objects to the required perspective of the system.</i>	see above
<i>Heuristic 4: Create one object per replaceable HW unit.</i>	This is the object structure of EM and, consequently, the BB structure of EM.
<i>Heuristic 5: Refactor domain-induced objects to objects of an application layer and an infrastructure layer.</i>	The separate subsystem SM and LRM result from the application of this heuristic.

Table 8: Application of Heuristics in tss

List of Heuristics	tss Application Comment
<i>Heuristic 6: Refactor large collections of application objects to objects of a basic application layer and an advanced application layer.</i>	The SM subsystem is substructured into basic call handling and into call facilities.
<i>Heuristic 7: Design objects which will be implemented by an operating system layer independent from an additional middleware layer.</i>	Generic functionality is factored out into EOS.
<i>Heuristic 8: Design messages which are sent between threads and processes in separate objects.</i>	Done as part of concurrency design.
<i>Heuristic 9: Design objects which hold message objects such as mailboxes, buffers, queues as separate objects.</i>	Each thread had one or more message buffers.
<i>Heuristic 10: Design protocol implementations as objects.</i>	Call signalings are separated into their own BB, local variants even further separated into a generic BB and specifics.
<i>Heuristic 11: Group interfaces of several domain-induced objects to one interface abstraction.</i>	Interfaces of different PU handling BBs are abstracted and access via a generic interface.
<i>Heuristic 12: Limit the visibility of attributes and operations of domain-induced objects behind interface objects.</i>	Done via interfaces of generic BBs.
<i>Heuristic 13: Model registration functionality as a separate design object.</i>	A circuit (CCT) is an example of such an extra object.
<i>Heuristic 14: Use container objects for explicitly handling instances of a class in lists and queues.</i>	No OO programming is used, lists are handled per BB.
<i>Heuristic 15: Use separate objects to model aspects with large amount of functionality.</i>	No OO programming is used, often one file per aspect is used.
<b>Heuristics of Aspect Design</b>	
<i>Heuristic 16: Take the complete functionality as the first aspect called operational aspect.</i>	Not used, useful for new projects.

Table 8: Application of Heuristics in tss

List of Heuristics	tss Application Comment
<i>Heuristic 17: Look for common behaviour of domain-induced objects. Allocate similar cross-cutting behaviour to one aspect.</i>	Performance observation is the latest aspect. It developed from divers implementation of similar functionality into a separate aspect.
<i>Heuristic 18: Use lists of architectural concerns from design of similar systems for analysing the required functionality for the identification of aspects.</i>	Not used in tss, next heuristic used instead.
<i>Heuristic 19: Use lists of aspects from other systems as starter sets for aspect identification.</i>	tss started with the list of aspects of the PRXD switching family.
<i>Heuristic 20: Select only those aspects which are relevant for the complete product family as SW aspects.</i>	Not used in tss.
<i>Heuristic 21: Support identified product-specific crosscutting functionality through the design of a generic BB during composability design.</i>	Not used in tss.
<i>Heuristic 22: Limit the number of different design concepts per aspect to increase conceptual integrity.</i>	Continuous discussions among architects, e.g. the state model (section A.3.3.3) is applied also to logical entities.
<i>Heuristic 23: Weigh the smaller number of aspects with potentially different designs against a larger number of small aspects with a unique design.</i>	tss avoids a long list of aspects, e.g. error handling used different design concepts for different parts of the system.
<i>Heuristic 24: Introduce a standard structuring for BBs by letting all aspects be present in each BB, even if some of the aspects are empty in a particular BB.</i>	In tss used for documentation and code of a BB.
<i>Heuristic 25: Use the list of aspects for checking completeness during review sessions. Structure large review team by allocating aspects to specific reviewers.</i>	Practised for reviews.
<i>Heuristic 26: Make a separate chapter per aspect in the BB documents.</i>	Such chapters are called page group which could be independently released.

Table 8: Application of Heuristics in tss

List of Heuristics	tss Application Comment
<i>Heuristic 27: Structure the implementation of a BB according to aspects.</i>	Supported by using separate files per aspect.
<b>Heuristics of Concurrency Design</b>	
<i>Heuristic 28: Use address spaces as failure containment units. Recovery from failure is realised within an address space.</i>	tss uses a single address space, MMU used for protection and of data classes (dynamic/persistent/hard) (recovery pre data class).
<i>Heuristic 29: Use address spaces to design for deployability. The freedom to relocate functionality to different processors depends on the absence of common data between address spaces.</i>	tss BBs are not relocatable, relocatability towards external PCs was under discussion for functionality of the system management interfacing aspect.
<i>Heuristic 30: Consider the use of a thread on the architectural level.</i>	Performance problems and the necessity for refactoring in tss led to this insight.
<i>Heuristic 31: An overview of all threads should be given in a global concurrency design.</i>	No document, only in the head of architects.
<i>Heuristic 32: Mirror independent behaviour of application domain objects by separate logical threads.</i>	First design step in tss, e.g. call facilities are designed as configurable state machines without separate threads.
<i>Heuristic 33: Use a separate thread to handle an external connection or external messages.</i>	Done by the BB MTS of the EOS.
<i>Heuristic 34: Cluster all functionality which is activated via object interaction by the external connection or messages into the thread.</i>	One call of a subscriber is handled by one thread instance, no change of thread during one interaction.
<i>Heuristic 35: Use a separate thread for the interaction of a user with the system.</i>	The system management interfacing aspect has a separate thread.
<i>Heuristic 36: Represent the receiving direction of an external channel or bus by its own thread.</i>	A thread instance per external connection is used for PG handling.
<i>Heuristic 37: Message sending over an external channel is done on the budget of the sending thread</i>	Done, MTS had a thread interface for the receiving direction and a procedural interface for the sending direction.

Table 8: Application of Heuristics in tss

List of Heuristics	tss Application Comment
<i>Heuristic 38: Refine the design of a separate thread per bus to have thread instances per connected equipment instance to the bus.</i>	Thread instances used per PU instance.
<i>Heuristic 39: Let internal consistency have priority over external reaction.</i>	Error handling and recovery run under the highest priority.
<i>Heuristic 40: Give operational tasks priority over background tasks.</i>	Background tasks have lowest priority.
<i>Heuristic 41: Use separate thread per different priority.</i>	Often priorities are assigned per aspect.
<i>Heuristic 42: Use a separate thread per cluster of objects with given priority.</i>	A good analysis is important for achieving good system performance.
<i>Heuristic 43: Split logical threads up into physical threads per address space.</i>	No separation into logical and physical thread used since address spaces are not used.
<b>Heuristics of Composability Design</b>	
<i>Heuristic 44: Cluster objects into BBs such that coupling of objects across BB borders is low and cohesion of objects within a BB is high.</i>	One of the first heuristics to use.
<i>Heuristic 45: Cluster objects into a BB which represent a feature.</i>	Configurability was always an important design criterion.
<i>Heuristic 46: Cluster objects into different BBs which belong to independently evolvable parts.</i>	Separate BBs for 2K and 8K switches.
<i>Heuristic 47: Cluster objects into BBs such that a BB can be used as a work allocation units for 1 or 2 persons.</i>	Such a need leads to the search for a stable interface and the precise roles of the BBs.
<i>Heuristic 48: If the variation point lies inside a BB, refactor the BB such that the variation point lies at the border of a BB.</i>	This often lead to the design of a generic BB, new features often induce such refactorings.
<i>Heuristic 49: Factor out functionality which is present in several BBs in a separate BB.</i>	Refactoring which is not really necessary lead to difficult discussions with project leaders.

Table 8: Application of Heuristics in tss

List of Heuristics	tss Application Comment
<i>Heuristic 50: Take as main criterion stability under evolution, that is, an interface should be such that it can serve for those implementations and those usages which are likely to happen.</i>	Stability considerations are very important.
<i>Heuristic 51: Factor generic implementation parts which are used by several specific parts into a generic BB.</i>	Modularity of new functionality evolves via such refactoring.
<i>Heuristic 52: Take the implementation of common aspect functionality as a candidate for a SIG.</i>	New SIG were not easily added since they were part of the EOS and had specific support tools.
<i>Heuristic 53: Resolve mutual dependence between BB A and BB B in the follow way: if A is expected to be more stable than B, then make B depend on A; and vice versa if the communication between A and B is expected to be the most stable part, factor the communication out into a new BB and let both, A and B, depend on it.</i>	The stability criteria is very important but not always easy to determine in practice.
<i>Heuristic 54: In the case of embedded systems, use importing of interfaces at compile time if needed for performance reasons. Otherwise use dynamic exploration of interfaces for more flexibility.</i>	Only compile time importing is used.
<i>Heuristic 55: Structure interfaces according to aspects.</i>	Used in documentation and code of a BB.
<i>Heuristic 56: Use layering for BBs on two levels. Subsystems, which are collections of BBs, are layered. These layers are based on the classification of layers of domain objects done during object design.</i>	One of the most important heuristics to achieve an incremental system structure.
<i>Heuristic 57: Individual BBs within subsystems are also layered in relation to other BBs.</i>	Complements previous heuristic. Is often used together with heuristic 53.
<i>Heuristic 58: A common principle for the layering of software is to separate hardware-technology-oriented functionality from application-oriented functionality.</i>	Very basic in the electronics industry.

Table 8: Application of Heuristics in tss

List of Heuristics	tss Application Comment
<i>Heuristic 59: Construct layers as virtual machines for higher layers.</i>	It is important to think about how one's own interface is used by others.
<i>Heuristic 60: Another way of introducing layers is to distinguish between generic and specific functionality.</i>	Basic rule for layering in tss.
<i>Heuristic 61: The usage of transparent layers is favourable to the usage of opaque ones.</i>	Mostly used throughout tss.
<i>Heuristic 62: Opacity is used for layers that function as facades, such as abstraction layers for hardware, operating system or middleware.</i>	Only used for the EOS.
<i>Heuristic 63: Use layers to structure communication in a system.</i>	Done in tss; leads to incremental and understandable system structure.
<i>Heuristic 64: Separate common functionality from specific functionality.</i>	This is a basic tss structuring principle.
<i>Heuristic 65: Look for the diverse parts in similar functionality for different features.</i>	Often more important than commonality analysis.
<i>Heuristic 66: Use inversion of control for designing the functionality of a generic BBs.</i>	Basic principle for component frameworks.
<i>Heuristic 67: A generic BB is stable if new specific functionalities may be based on the generic BB without changing it.</i>	The best results are achieved with PU handling.
<i>Heuristic 68: Use an abstraction generic to implement an abstract concept which is to be extended by specific BBs.</i>	Examples range from call handling, PU handling to file handling.
<i>Heuristic 69: Use a connectable resource generic to manage connectable resources which are supplied by HW boards.</i>	Used for almost all external connections except for MMI-PCs for historic reasons.
<i>Heuristic 70: Design a system infrastructure generic for functionality which provides an operating infrastructure for almost all application BBs.</i>	Establishes a 'domain infrastructure'.

Table 8: Application of Heuristics in tss

List of Heuristics	tss Application Comment
<i>Heuristic 71: System Infrastructure Generics must provide interfaces for application BBs for indicating their resource requirements.</i>	Supported through code generation from Data Definition Database.
<i>Heuristic 72: Design a layer access generic to restrict the visibility of the structure of a layer for higher layers.</i>	Done for access of SM and LRM to EM.
<i>Heuristic 73: Apart from a BB itself, the collection of BBs of a white box component can be packaged as unit of deployment.</i>	For functional extensions of a tss system BBs of a feature are loaded to an installed system.
<i>Heuristic 74: If two substitutable BBs are to be present in the same product, the BBM requires that there must also be some generic which switches between the two.</i>	Often generic BBs implement functionality to select a specific BB to which to communicate.
<i>Heuristic 75: Choose generic BBs in such a way that stability of architectural skeleton increases.</i>	Is achievable only over time, the harvest of good system design.
<i>Heuristic 76: Make a BB is a self-describing component by letting it communicate its characteristics such as its resource requirements to the infrastructure.</i>	Supported by Data Definition Database in tss.
<b>Heuristics of Deployability Design</b>	
<i>Heuristic 77: Select a set of objects in such a way that the set may be independently recoverable when an error occurs.</i>	tss has a simple recovery model where only PU can recover independently, recovery of the CC done per data class, requires complete system recovery of that data class.
<i>Heuristic 78: Align BB-, thread-, fault containment unit- boundaries to HW instances</i>	Not used since tss BB are designed for the CC only.
<i>Heuristic 79: Package BBs to deployment sets such that independent selling and evolution remains possible.</i>	Partial delivery in tss done per set of BBs implementing features.
<b>Heuristics of Composability Design</b>	
<i>Heuristic 80: Use a feature relation graph to describe relations between features.</i>	A feature DB contains these relations.

Table 8: Application of Heuristics in tss

List of Heuristics	tss Application Comment
<i>Heuristic 81: Design a system infrastructure generic to handle data-driven diversity.</i>	Functionality of some aspects of some BBs 'implemented' via 'deployment descriptors' only.
<i>Heuristic 82: A desirable non-conservative extension is the refactoring of a BB to a generic BB.</i>	'Never change a running system' is not always a good principle.
<i>Heuristic 83: A good family architecture is one whose BB structure resembles the feature structure, i.e. a good family architecture is feature-oriented.</i>	Good results in SM, LRM and EM.
<b>Additional Heuristics of the Specialised</b>	<b>BBM</b>
<i>Heuristic 84: A managed object may consist of an object in the CC and an object in the peripheral hardware.</i>	Most MOs of EM and LRM are split in that way.
<i>Heuristic 85: Hardware objects and hardware abstractions of the CC will often be part of the OS.</i>	Standard rule for separating EOS and LRM functionality.
<i>Heuristic 86: Maintenance replaceable units (MRU) are good candidates for hardware managing objects.</i>	Done for almost all plugable HW.
<i>Heuristic 87: Represent MRUs, which only together realise a specific function in the system, by one hardware managing object.</i>	Shelve extension cards and the PG card to which they are connected are represented by one managing object.
<i>Heuristic 88: For the specialised BBM, we will always have the two layers in the central controller, namely application and equipment management.</i>	The specialised BBM is applied in tss.
<i>Heuristic 89: When interface abstractions between the two layers have themselves state and behaviour create a new layer for these abstractions. They are then to be modelled as managed objects in their own right and be represented as an intermediate layer.</i>	Holds partly for LRM.
<i>Heuristic 90: A further division may be appropriate if additional abstractions are introduced to abstract from the distribution of the controller over several sites. The application functionality then runs on top of the multi-site abstractions.</i>	Not used in tss, because tss systems are mostly single site. Remote PUs for rural areas are handled as other PUs; modelling in some cases too simplistic.

Table 8: Application of Heuristics in tss

List of Heuristics	tss Application Comment
<i>Heuristic 91: A different division of layers may be appropriate if application functionality extends significantly. An application-specific platform encapsulates application infrastructure abstractions. Various advanced applications may run on this platform.</i>	A basic call handling layer is started but not completely worked out.
<i>Heuristic 92: Infrastructure functionality such as basic services which should be used by all the objects implemented on the system controller are modelled in the lowest layer.</i>	Since tss developed its own operating system, these services are made part of it.
<i>Heuristic 93: An important set of managed objects and their respective BBs concerns the handling of the PUs. The BBs in the EM layer will reflect the connection structure of the PU.</i>	This established the BB structure of the EM layer without generic BBs.
<i>Heuristic 94: The system management interfacing aspect consists of the functionality to communicate with a system management system and with the operators.</i>	The operator can not only interact with the application but can configure and tune the complete system.
<i>Heuristic 95: The recovery aspect consists of functionality for system initialisation and automatic recovery.</i>	tss did not separate initialisation and recovery in separate aspects because the large overlap of concepts.
<i>Heuristic 96: The data replication aspect is a consequence of the distributed architecture. It consists of functionality to replicate data within a managed object, that is, the control and management data of the control object is sent to the real resource object, and changes in the real resource object are propagated to the control object.</i>	Basic principle of telecom systems; important for structuring internal communication in such systems.
<i>Heuristic 97: The configuration management aspect establishes configuration parameters according to a system database or operator actions.</i>	Basic aspect; DB used throughout the system, state model implementations per generic of functional area.
<i>Heuristic 98: The fault management aspect supervises the system configuration and takes decisions on required actions in case of failure or other abnormalities.</i>	Basic aspect; uniform implementation only for error reporting part; errors of the CC and of the PUs are handled differently.

Table 8: Application of Heuristics in tss

List of Heuristics	tss Application Comment
<i>Heuristic 99: The performance management aspect has the task to monitor and register the quality of the system configuration. If certain quality thresholds are exceeded fault management is informed.</i>	The latest aspect of tss. Is not nicely implemented throughout the system.
<i>Heuristic 100: A quite typical design is to separate control functionality from processing functionality. Processing is allocated to the periphery, while control is allocated to the CC.</i>	A heuristic which determined the overall system architecture of tss.
<b>Heuristics of Organisational and</b>	<b>Process Issues</b>
<i>Heuristic 101: Develop a first product that can be used as a basis for the product family.</i>	Done for the tss product family, other projects had it to learn the hard way.
<i>Heuristic 102: Define and detail the architecture in such a way that BBs can be developed according to a simple waterfall model.</i>	Important quality criteria for architects.
<i>Heuristic 103: The architecture document describes the architectural models such as the BB dependency model, aspect designs, concurrency design and deployability design. The architecture document should be structured in a way which minimises the impact of changes.</i>	This was not achieved. There existed always a set of more or less important documents. Concurrency design did not have a separate document independent from BBs.
<i>Heuristic 104: The BB documentation consists of at least three documents: its specification document, its design document and its code document.</i>	Done for all BBs. The specification document also covered MO-related peripheral SW.
<i>Heuristic 105: Consider a deviation of the BB development process from the simple waterfall a quality problem of the architectural process.</i>	For BBs with completely new functionality architects sometimes had to work out sample designs.
<i>Heuristic 106: Proceed with the process of integration by extending a stable set of BBs with one or a few BBs.</i>	Very important for short integration times.

Table 8: Application of Heuristics in tss

---

## A.4 Making Products From BBs

tss products are configured according to the product feature list. The elements of a product are

- the mechanical shelves together with backplane buses and power supplies,
- the peripheral HW units with the according SW modules,
- the switching matrix modules,
- the operation and maintenance terminals and
- the central processor HW boards with a set of BBs.

The selected BBs of a product have to be consistent, that is, they all need to be of a compatible version.

In this section we describe concepts and ways of working for configuring, instantiating and evolving tss products. They are not considered part of the core BBM, however each practical application of the BBM needs to address the issues of configuration, instantiation and versioning.

### A.4.1 Construction Set

The use of SW components introduces great flexibility into product development. Each BB may be independently released and has own version number. From a management point of view concepts and a way of working are needed to effectively use this flexibility.

The tss development used the concept of a construction set. A construction set is the set of BB versions which are compatible and may be used to configure a product. Several products may be build from the same construction set. In an ideal world where each BB is ideal there would be only one construction set for the complete product family.

The presence of several versions of a BB in a construction set is possible. The situation where new BBs needed for a new product is handled by extending the construction set with these new BBs. If a set of existing BBs needs to be adapted three options are possible:

create a new construction set with same versions for the unchanged BBs and the new versions for the adapted ones. This is done if major changes are necessary such as restructuring BBs to introduce a generic BB its specifics;

make the adapted BBs variants with a new identity and add them to the construction set. This is done if new features are to be added;

allow different versions of the same BB in the same construction set. This done for minor changes

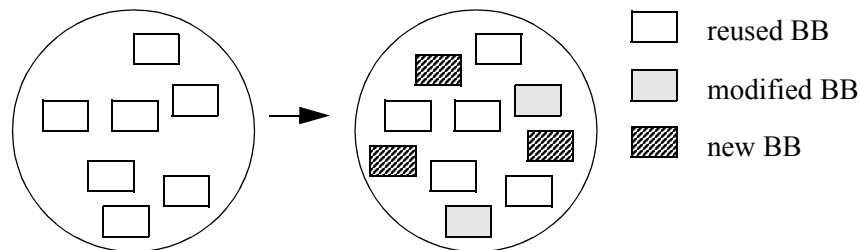
The last option requires to keep additionally the specific version of a BB in a product. Methods for ensuring compatibility between BBs such as testing have to take the selected option into account.

#### A.4.2 Product Tailoring and Evolution

Often, in order to yield a product from the construction set of compatible BBs adaptations and modifications are required. In order to determine the required activities an initial product configuration step can be performed on the existing base of developed BBs. The result of these steps will lead to

new BBs to be developed

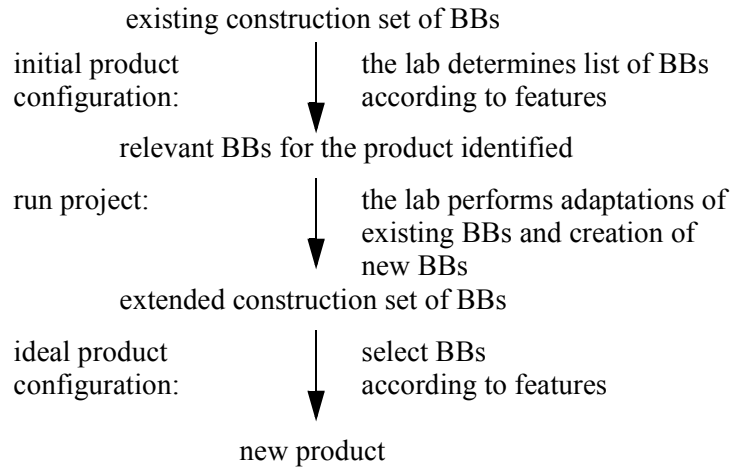
adaptations to existing BBs to be made (figure 93).



*Figure 93: Evolving the Construction Set*

The goal is to minimise the number of BB versions. An extended BB should be (upwards) compatible with the previous version of that BB, to the extent possible. The construction set is extended in a compatible way. This again makes it

possible to generate the new product by performing another (ideal) product configuration step (see figure 94).



*Figure 94: Development Steps to Extend the Construction Set*

A BB remains an entity, not only during design and implementation, but also during all phases of development. In particular, it is also an entity of documentation. In fact, as a product is configured out of the parts list of BBs, the documentation is configured as well. Different forms of documentation are supported through document generators of the DDD. In figure 95 an overview of the DDD is given.

Besides the code generators to support the creation of operator interfacing code, generators for office data and for manuals also have been developed. All the documentation is consistent with the system implementation by generation

Items related with EOS Generics:

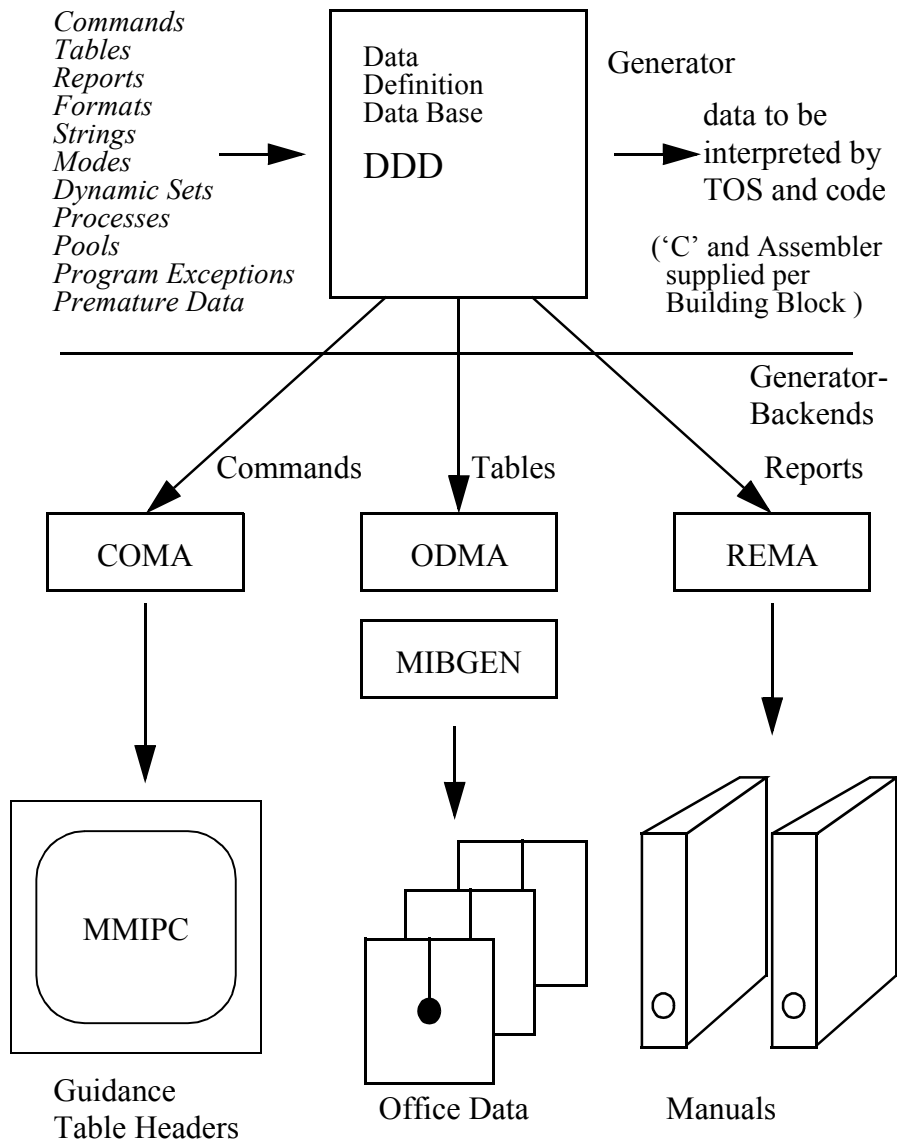


Figure 95: Overview of the DDD

from a single source. The DDD tool thus supports data consistency between different departments.

### **A.4.3 Product and Site Configuration**

In the above sections we described the concept of a construction set and how the tailoring of a product leads to evolution of BBs and construction sets. In this section we take a closer look how a specific instance of a product is build.

#### **Product Configuration**

A product is configured by using the construction set of BBs and the corresponding data definition database (DDD) contents. In order to identify the list of required BBs, that is the parts-list of the product, the following procedure is applied.

Select the number of features required for the product. Use the feature dependency model to select also dependent features. Then, use the feature implementation relation to select the relevant BBs. Again, the BB dependencies may lead to additional BBs.

Applying this procedure implicitly results in the subset of binaries required for the product. One of the goals of configuring a product is to keep the number of BBs on a minimum. The Software Factory, which is responsible for customer products, determines the BB binary configuration. The project data are documented in a 'Project Manual'. Among others, this manual contains the parts list of BBs and project dependent resources like stored announcements and tones.

The basis for configuring a specific product instance is the set of BBs and the corresponding soft data held by the DDD. The steps to be performed in order to yield product and site configurations roughly is as follows:

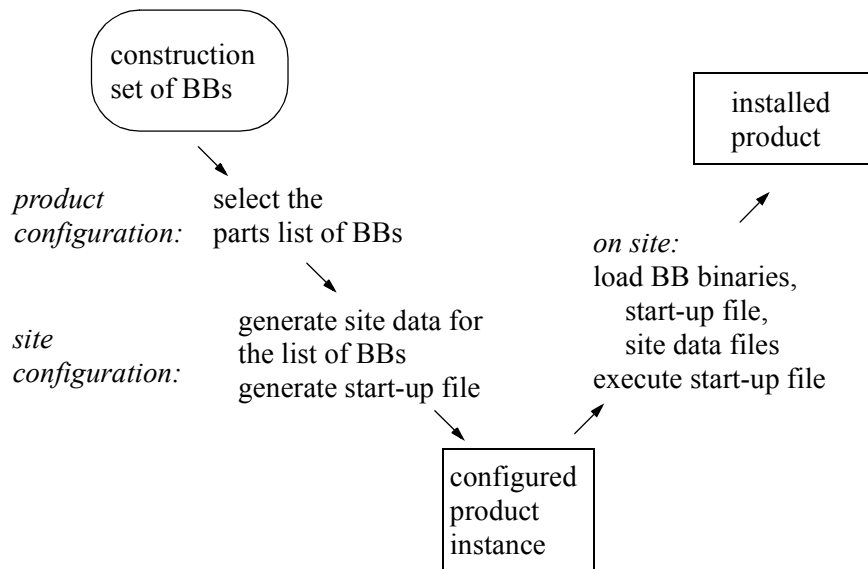


Figure 96: Process Steps for Configuring a Product Instance

### Site Configuration

BB binary configuration determines the set of tables contained in a product but not their contents (table entries). The contents of tables and data files (e.g. stored announcements) are site dependent and therefore are termed site data (see figure 97). As well as project administration, site administration is up to the software factory (see figure 97). The process of defining site data is termed soft data configuration. Site configurations are documented in so-called site data manual.

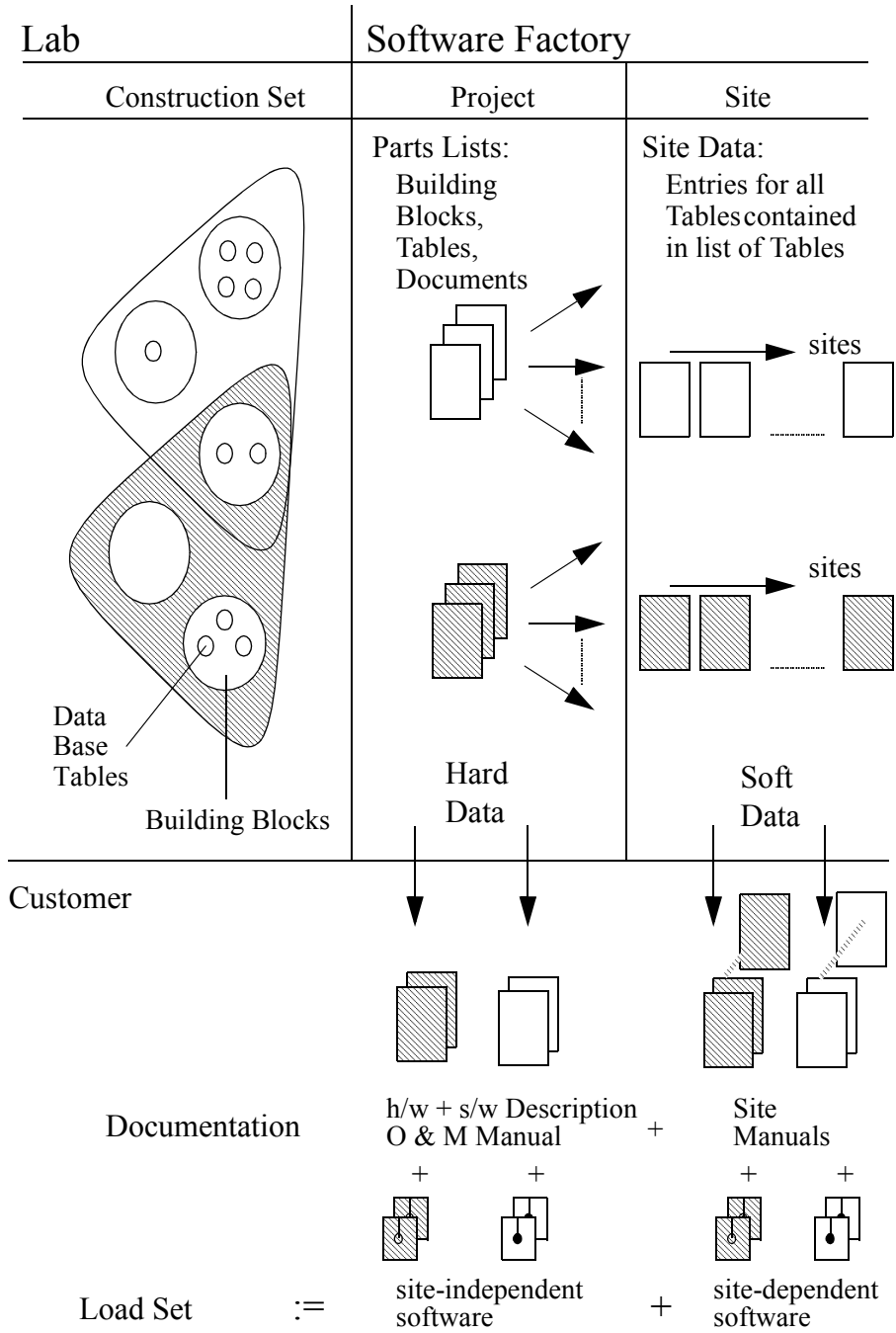


Figure 97: Process Overview of Product and Site Configuration

---

## A.5 Experience Data

In this section we give several sets of data collected from tss projects. The first data set (section A.5.1) describes the products of the tss family and the number of installations. The second data set (section A.5.2) gives typical performance characteristics of tss product. The third data set (section A.5.3) gives the software sizes of the complete product family. The fourth data set (section A.5.4) compares the sizes of two component frameworks (generics) with the sizes of their plug-ins. The fifth data set (section A.5.5) looks at the level of reuse achieved within several products. The last two sections have been taken from a study performed by Fleischer and Jäger [FJ95].

### A.5.1 The tss Products

The telecommunication switching system tss is supplied to various markets with a emphasis on the German market.

The tss platform is tailored to the specific needs of special applications. This includes applications like operator-based support systems, service provider network access points, mobile containerised exchanges, base station central control (BCC) for GSM networks and switching systems for authority networks.

The most important projects for the tss switching system are in table 9.

Product	Applications	#Systems	Country
1	national operator-assisted directory service, automated wake-up service	90	Germany
2	international operator-assisted directory service	8	Germany
3	operator system for the technical telecom service	153	Germany
4	network access point for private service providers	9	Germany
5	mobile, containerised exchange	10	Germany
6	operator system for inquiry and call completion service	1	Germany
7	local / toll exchange	10	CIS
8	local / toll exchange	4	Jordan
9	container exchange	2	Czech
10	local / toll exchange	23	PRC
11	base station central control for GSM	150	Germany
12	base station central control for GSM	570	(international)

*Table 9: Products of tss*

### A.5.2 tss System Performance Data

Some performance numbers of tss products are given below. A tss product can be configured for:

up to 20000 subscriber lines (8 lines per PU) or

up to 6000 trunk lines (30 lines per PU) (system without subscriber lines)

or a mixture of the two. There are two variants of the switching matrix:

2048 x 64 kbit/s lines or

8096 x 64 kbit/s lines.

This leads to the maximum of 930 (62x30/2) or 3782 (62x122/2) stable calls, respectively, at the same time in the system.

The system performance:

performance: 40 calls/sec = 144.000 BHCA (depends on signalling and facilities)

700 messages/sec (between central processor and switching matrix/peripheral groups)

Recovery times: (depends on image and data base size)

reload: < 8 minutes

restart with database load: < 4.5 minutes

restart < 2 minutes

The image size for the CC of a typical project:

150 building blocks

2,6 MByte code

1 MByte hard data (=strings and other constant values)

1,5 MByte soft data (= configuration data base)

1,5 MByte dynamic data

10 MByte process stacks and dynamic pools

### A.5.3 Software Sizes

The software sizes of the complete tss product family are:

Central Controller	No. of BBs	No. of ELOCS (=Effective Lines of Code)
Extended Operating System	175	462500
Equipment Maintenance	93	207146
Logical Resource Management	123	342157
Service Management	89	373469
Total Sum	480	1385272

*Table 10: Central Controller Software Sizes*

The 480 BBs of the CC (table 10) of the tss product family amount to ca. 14MB of binaries, that is a middle of 30 kB binary per BB.

Group	No. of Building Blocks / Modules	No. of ELOCS (=Effective Lines of Code)
Central Controller Software	480	1385272
Peripheral Software	186	416542
Personal Computer Software	111	177881
Switching Network Software	2	21700
SDE Tools	41	506930
Total Sum	820	2 508 325

*Table 11: tss Software Sizes*

The complete SW of the tss product family is given in table 11.

#### A.5.4 Comparing Generics and Specifics

The following comparison is made to compare the relative size a component framework (generic) has with respect to the complete functionality (framework and plug-in).

The degree of reuse due to the use of abstraction generics with two different examples is made. To this end the delivered source instructions (DSI) of the abstraction generic and of each type-dependent package are compared to the sum of both figures. To present fair figures, we have to differentiate between DSIs, that implement common characteristics within the abstraction generic, and DSIs, that do correspond to generic dedicated services, like for instance binding.

The first example stems from the equipment maintenance layer. Common functionality for the different types of peripheral cards includes image and data download, supervision mechanism, error handling, configuration management etc. In the software architecture we therefore defined an abstraction generic, peripheral group card (PGC) generic, and several specific BBs for the handling of different cards. The following figures illustrate what amount of reuse has been achieved.

PGC Generic		
total # DSI	common code DSI	generic dedicated DSI
7501	7148	353

*Table 12: PGC Generic*

Type Dependent Package	# DSI	sum of DSI	percentage abstraction generic	percentage specific BB
SAG maintenance	1515	8663	83	17
GCC maintenance	784	7932	90	10
DTG maintenance	793	7911	90	10
CGC maintenance	482	7630	94	6

*Table 13: PGC Specific BBs*

As a second example the channel associated signalling administration (CASA) of the logical resource management layer is chosen. This administration

software takes care about the different channel associated signalling systems for telephone lines, that are configured for a system, and additionally administers the corresponding line data.

CASA Generic		
total # DSI	common code DSI	generic dedicated DSI
3912	3521	391

*Table 14: CASA Generic*

Specific BB	# of DSIs	sum of DSIs	percentage abstraction generic	percentage specific BB
CAI trunk line admin	1754	5275	67	33
CAO trunk line admin	1277	4798	73	27
CAIRSU trunk line admin	1253	4774	74	26
CAORSU trunk line admin	1592	5113	69	31

*Table 15: CASA Specific BBs*

As can be inferred from the figures above, the degree of reuse in the first example is higher than in the second one. This is due to the fact, that the hardware architecture determined by the manufacturers is designed for commonality, whereas the signalling system exhibits the variety of application functionality. Throughout the tss system, the average reuse per abstraction generic, therefore, lies in between both extremes.

Considering all abstraction generics throughout the system and evaluating the amount of reuse contributed by them yields a reuse of about 25%. I.e. the current code will increase by about 25%, if we dispose of the abstraction generics.

In general, 70 of the 480 BBs are generics. 35 component frameworks reside in the application EM, LRM and SM. The EOS contains 25 component frameworks and 10 system infrastructure generics.

### A.5.5 Inter-Product Reuse

In this section the reuse of software of several product development projects is evaluated. Before the actual figures are presented, a description of the method of evaluation of reuse is given. Thereafter the products are briefly introduced and their differences are discussed.

As is normally the case, the available BBs within the construction set do not cover all features required for a new product. Therefore it is necessary to implement new BBs for the product. These newly implemented BBs in turn extend the construction set of BBs. The extended construction set is available for use at a later project. This is illustrated in figure 98.

New BBs are also implemented for another reason. As time proceeds, shortcomings in the system software architecture inevitably will be detected. In order to improve on them, redesigns of specific BBs will have to take place. Redesigns, however, are always associated with a new project and never occur for their own sake. Again this is sketched in figure 98.

As a consequence three categories of BBs contributing to a particular project can be distinguished:

*Unchanged BBs:*

these BBs are taken over from the existing construction set without any modification. They therefore directly contribute to reuse of software.

*New BBs:*

these BBs implement new system features. Their functionality has not been available before and they therefore do not contribute to reuse of software.

*Modified BBs:*

modification of BBs can occur due to the following reasons:

- change of the user interface of the BB;
- redesign of the BB due to architectural needs (refactoring).

In evaluating software reuse one therefore has to deal with three different categories of BBs. New, modified, and unchanged BBs. For each of these categories, two different ratios have been determined as a measure of reuse within the tss products. The first ratio is the number of BBs in each category to the total number of BBs within the project. The second ratio is the added number of source lines per category to the total number of source lines per project.

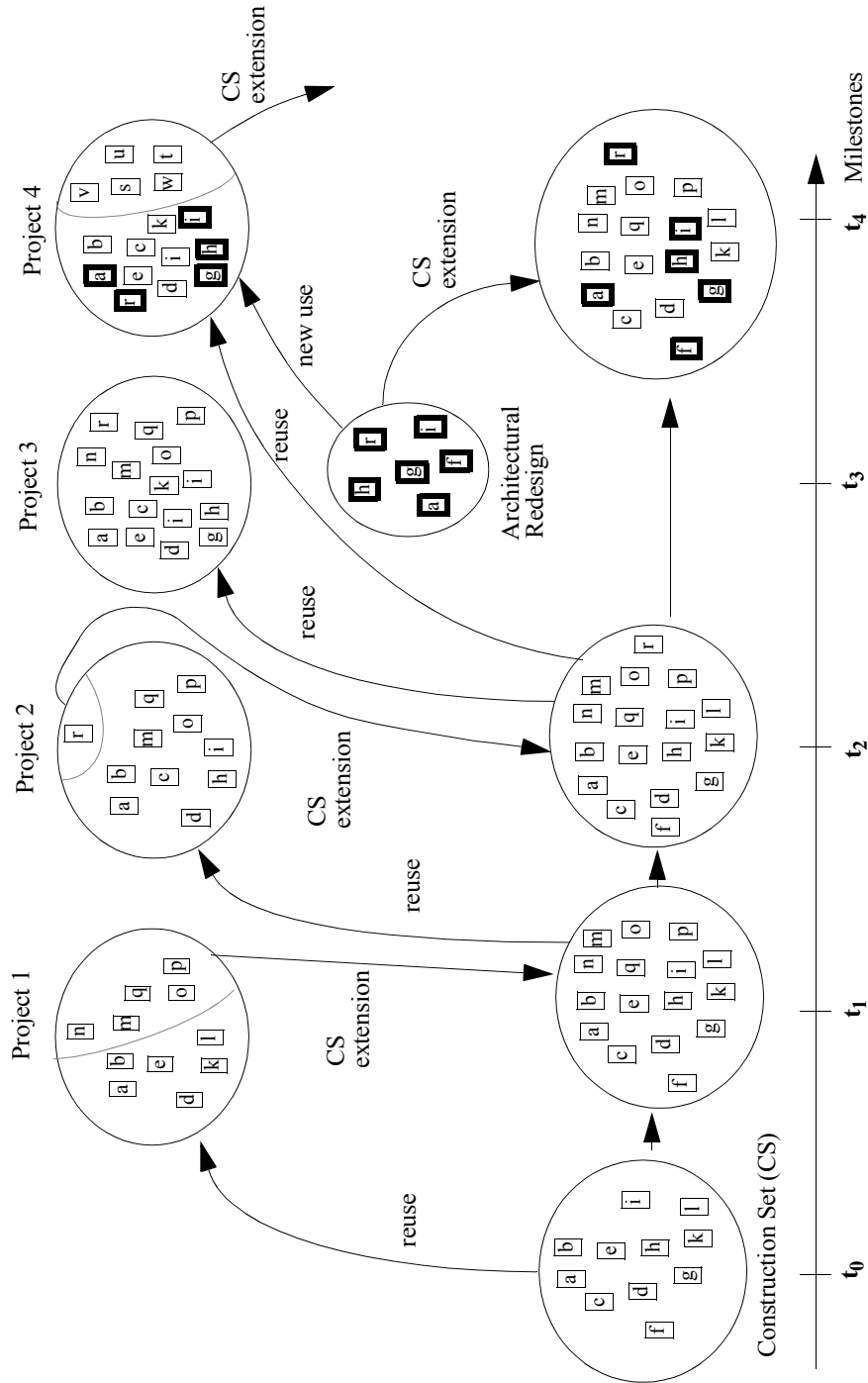


Figure 98: Construction Sets and Projects

For the categories of *new* and *unchanged* BBs these measures are clearly defined. An unchanged BB has been reused completely, a new BB corresponds to newly developed software and does not contribute to reuse.

For the category of *modified* BBs the interpretation is not as clear. BBs within these categories are partly reused and partly implemented anew. The fraction of reuse, however, could not be resolved with the available data. Therefore a *worst case strategy* has been adopted. For both measures, modified BBs are assumed not to contribute to reuse of SW.

Before the numbers are given, the projects, that have been evaluated, are shortly introduced.

The first product represents a digital switching system that serves as a combined local and toll exchange. It supports various signalling systems for trunk and subscriber lines. Additionally it offers various facilities, as for instance: Incoming call barring, traffic restriction, fixed destination call, three party services, abbreviated dialling, access code handling, traffic telephony measurements, call forwarding and others more.

The second product represents a container local exchange. It includes a charging service and signalling systems that are different from the signalling systems of the first project. Furthermore it only comprises few of the services listed with the first project, but offers others in return. E.g. the features “concentration of analogue subscriber lines” and “party lines” are additionally present.

The third product represents a service switch, that provides operator assisted value-added services. The implementation of those services includes functions like: automatic call distribution, call waiting, generated announcement of directory numbers, data links, call forwarding, traffic measurement, service statistics. This system is used by service providers of radio communication networks and within public networks.

The fourth product is a combined digital local and toll exchange similar to the first project. Additionally it comprises features of an operator assisted value-added service switch, corresponding to project 3. Therefore it practically covers the same facilities as were presented for the first and the third project. It differs, however, in the signalling systems, that are being used.

Table 16 presents a short summary of the projects analysed. It gives the number of new features introduced with the project and the total amount of BB and DSIs per project.

projects	number of new features	total	
		number of BBs	number of DSI
project 1	14	87	196266
project 2	12	75	189921
project 3	34	99	310184
project 4	22	103	288010

*Table 16: Basic Project Data*

unchanged BBs		new BBs		modified BBs	
BBs		BBs		BBs	
#	%	#	%	#	%
72	82.8	0	0	15	17.2
63	84.0	6	8.0	6	8.0
47	47.5	13	13.1	39	39.4
72	69.9	24	23.3	7	6.8

*Table 17: Number of BBs per Category*

unchanged BBs		new BBs		modified BBs	
DSI		DSI		DSI	
#	%	#	%	#	%
163945	83.5	0	0	32321	16.5
154295	81.2	16497	8.7	19129	10.1
74396	24.0	22579	7.3	213209	68.7
176567	61.3	48696	16.9	62747	21.8

*Table 18: Number of DSIs*

projects	degree of reuse
project 1	> 82%
project 2	> 81%
project 3	> 24%
project 4	> 61%

Table 19: Degree of Reuse of Application Software

The data shown in table 17 through table 19 result from projects developed from 1991 through 1993. They show that the underlying concepts and principles have led to a stable system architecture. With this architecture a parts-list of BBs has been realised implying configurability and a high degree of inter-product reusability. As can be inferred from the tables, the fraction of reused DSIs for a product ranges between 60 to 80 percent with one exception that is discussed below.

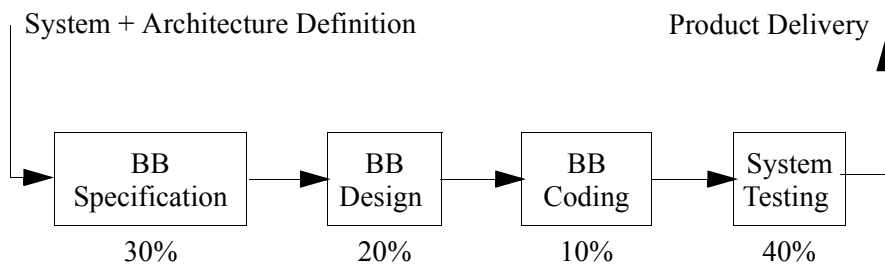
At the first project, the high fraction of modified BBs (16.5% of DSI) contrasts the number of newly developed BBs (0% of DSI). This traces back to the fact, that the new requirements for this project were accomplished by using BBs of former projects, only. Those BBs, that mainly implement signalling systems, had to be adapted to the new requirements and therefore appear as modified BBs in the statistics.

The data from the second and fourth product indicate a stable system architecture, that necessitates only modest modifications in order to bring in new features. The degree of reuse of the application software is high.

Note that these numbers *do not include the operating system software*. If this were the case, the degree of reuse would be higher from the very beginning. The number of DSIs for the operating system software, which besides the kernel includes infrastructure elements as for instance database management services, is about 460,000. This compares to the number of DSIs for the application software.

Outstanding with respect to the figures of reused and modified BBs is the third product. The high fraction of modified BBs (68.7% of DSI) is because a basic data structure of the system had to be extended. This data structure was used by several BBs.

Further data from the four projects show the same distribution of efforts over the development phases.



*Figure 99: Empirical Data on the Distribution of Efforts*

---

## References

- [AMO\*00] Pierre America, Jürgen K. Müller, Henk Obbink, Rob van Ommering: *COPA - A Component-Oriented Platform architecting Method for Families of Software-Intensive Electronic Products*, Tutorial given at the 1.st Software Product-Line Conference (SPLC1), 2000 [http://www.extra.research.philips.com/SAE/COPA/COPA\\_Tutorial.pdf](http://www.extra.research.philips.com/SAE/COPA/COPA_Tutorial.pdf)
- [Bau95] Lothar Baumbauer: *System Level Documentation*, Volume 6014 (internal documentation) Philips Kommunikation Industrie AG, 1995
- [BCK98] Len Bass, Paul Clements, Rick Kazman: *Software Architecture in Practise*, Addison-Wesley, 1998
- [Ben97] Douglas W. Bennett: *Designing Hard Software - The Essential Tasks*, Manning, Greenwich, 1997
- [BGK\*99] Dirk Bäumer, Guido Gryczan, Rolf Knoll, Carola Lilienthal, Dirk Riehle, Heinz Züllighoven: *Structuring Large Application Frameworks*, in Mohamed E. Fayad, Douglas C. Schmidt, Ralph E. Johnson (Eds.): *Building Application Frameworks*, Wiley, 1999
- [BGP00] Laszlo Boszormenyi, Jürg Gutknecht, Gustav Pomberger (Eds.): *The School of Niklaus Wirth - The Art of Simplicity*, dpunkt.verlag, 2000
- [Bir96] Kenneth P. Birman: *Building Secure and Reliable Network Applications*, Manning Publications, 1996
- [BM99] Jan Bosch, Peter Molin: *Software Architecture Design: Evaluation and Transformation*, Proceedings of the Engineering of Computer-Based Systems Conference, August 1999
- [BMR\*96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: *Pattern-oriented Software Architecture - A System of Patterns*, Wiley and Sons Ltd., 1996
- [Boa93b] Maarten Boasson: *Control Systems Software*, IEEE Transactions on Automatic Control, Vol. 38, No. 7, 1993, pp. 1094-1107

- [Boa93a] Maarten Boasson: *Complexity may be our own fault*, IEEE Software, March 1993
- [Boe87] Barry Boehm: *A spiral model of software development and enhancement*, IEEE Computer, May 1988
- [Bro75] Frederick P. Brooks, Jr. *The Mythical Man-Month - Essays on Software Engineering*, Addison-Wesley, 1975
- [CE00] Krzysztof Czarnecki, Ulrich W. Eisenecker: *Generative Programming - Methods, Tools and Applications*, Addison Wesley 2000
- [CG89] N. Carriero, D. Gelernter: *Linda in Context*, Communications of the ACM, Vol. 32, No. 4, 1989, pp. 444-458
- [CHW98] James Coplien, Daniel Hoffman, David Weiss: *Commonality and Variability in Software Engineering*, IEEE Software, November 1998, pp. 37-45
- [Cla85] David D. Clark: *The Structuring of Systems using Upcalls*, Proceedings of the 10th ACM Symposium on Operating System Principles, ACM Operating System Review, Vol. 19, No. 5, pp. 171-180
- [Cle95] Paul C. Clements: *From Subroutines to Subsystems: Component-Based Software Development*, The American Programmer, Vol. 8, No. 11, November 1995
- [Con80] Larry L Constantine: *Objects, Functions, and Program Extensibility*, Computer Language, January 1980
- [Dav93] Alan M. Davis: *Software Requirements - Objects, Functions, & States*, Prentice Hall, 1993
- [Dij68] E. Dijkstra: *The structure of the "THE" -Multiprogramming System*, Communications of the ACM, Vol. 11, No. 5, 1968
- [Dit97] K. Dittrich: *Datenbanksysteme*, in Rechenberg, Pomberger: Informatik-Handbuch, Hanser, 1997 (in German)
- [DK98] Arie van Deursen, Paul Klint: *Little Languages: Little Maintenance?*, Journal of Software Maintenance, 1998
- [DKO\*97] David Dikel, David Kane, Steve Ornburn, William Loftus, Jim Wilson: *Applying Software Product-Line Architecture*, IEEE Computer, August 1997
- [DMN\*97] Serge Demeyer, Theo Dirk Meijler, Oscar Nierstrasz, Patrick Steyaert: *Design Guidelines for Tailorable Frameworks*, Communications of the ACM, Vol. 40, No. 10, October 1997

- [DW99] Desmond Francis D'Souza, Alan Cameron Wills: *Objects, Components, and Frameworks with UML - The Catalysis Approach*, Addison Wesley 1999
- [DZ83] J.D. Day, H. Zimmermann: *The OSI Reference Model*, Proc. of the IEEE, Vol.71, pp. 1334-1340, Dec. 1983
- [FJ95] Wolfgang Fleischer, Horst Jäger: *Software Reuse through Configurability and Conceptual Integrity*, Proceedings of the 2nd Philips Software Conference, Feb. 1995
- [FO94] Ove Faergemand, Anders Olsen: *Introduction to SDL-92*, Computer Networks and ISDN Systems 26, pp. 1143-1167, 1994
- [Fow97] Martin Fowler: *UML Distilled, Applying the Standard Object Modeling Language*, Addison-Wesley, 1997
- [Fra97] Michael Franz: *Dynamic Linking of Software Components*, IEEE Computer, March 1997
- [FS97] Mohamed E. Fayad, Douglas Schmidt: *Object-Oriented Application Frameworks*, CACM Vol.40, No.10, October 1997, pp.32-30
- [GFA98] Martin L. Griss, John Favaro, Massimo d'Alessandro: *Integrating Feature Modeling with the RSEB*, P. Devanbu, J. Poulin (Eds.) Proceedings of the Fifth International Conference on Software Reuse, IEEE 1998
- [GHJ\*94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns: Elements of Object-Oriented Software Architecture*, Addison-Wesley 1994
- [Gom93] Hassan Gomaa: *Software Design Methods for Concurrent and Real-Time Systems*, Addison-Wesley, 1993
- [Gom95] Hassan Gomaa: *Reusable Software Requirements and Architectures for families of systems*, Journal Systems Software 28, 1995, pp. 189 – 202
- [HFC76] A.H. Haberman, Lawrence Flon, Lee Coopridner: *Modularization and Hierarchy in a Family of Operating Systems*, Communications ACM 19 No. 5, 1976, pp. 266 – 272
- [Jac98] Michael Jackson: *Formal Methods and Traditional Engineering*, Journal on Systems and Software, Vol. 40, pp. 191-194, 1998
- [JB95] K. Jackson, M. Boasson: *The importance of good architectural style*, Proc. of the Workshop of the IEEE TF on Engineering of Computer Based Sytsems, Tucson, 1995

- [JCJ\*92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard: *Object-Oriented Software Engineering - A Use Case Driven Approach*, Addison-Wesley 1992
- [JGJ97] Ivar Jacobson, Martin Griss, Patrik Jonsson: *Software Reuse - Architecture, Process and Organization for Business Success*, Addison Wesley 1997
- [Kan90] K. Kang et al. *Feature-Oriented Domain Analysis Feasibility study*, SEI Technical Report CMU/SEI-90-TR-21, November 1990
- [Kar95] Even-André Karlsson: *Software Reuse, A Holistic Approach*, Wiley 1995
- [KBP\*95] R. Kamel, T. Borowiecki, N. Partovi, L. Galvin: *The Evolution of Digital Switching Software Architecture*, Bell Northern Research, Ottawa Canada, ISS '95 Vol.2 1995
- [Kic96] Gregor Kiczales: *Beyond the Black Box: Open Implementation*, IEEE Software, January 1996
- [KLM\*97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, John Irwing: *Aspect-Oriented Programming*, Xerox Corporation, 1997
- [KMM96] B. Keepence, M. Mannion, C. McCausland: *Pattern Based Transformation of Domain Features*, IEEE Symposium on Engineering of Computer Based Systems, Friedrichshafen, Germany, March 11-15, 1996
- [Kri99] Rene Krikhaar: *Software Architecture Reconstruction*, PhD. thesis, University of Amsterdam (UvA), 1999
- [Kro93] Klaus Kronloef (ed): *Method Integration: Concepts and Case Studies*, John Wiley, Baffins Lane, Chichester, Wiley Series on Software Based Systems, 1993
- [Kru95] Philippe Kruchten: *The 4+1 View Model of Architecture*, IEEE Software, Nov.1995
- [Kru99a] Philippe Kruchten: *The Software Architect*, in Patrick Donohoe (Ed.): *Software Architecture*, TC2 First Working IFIP Conference on Software Architecture (WICSA1), 1999, pp. 565-583
- [Kru99b] Philippe Kruchten: *The Rational Unified Process - An Introduction*, Addison-Wesley, 1999
- [Lak96] John Lakos: *Large-Scale C++ Software Design*, Addison-Wesley, 1996

- [Lea00] Doug Lea: *Concurrent Programming in Java - Design Principles and Patterns - Second Edition*, Addison-Wesley, 2000
- [LM95a] Frank van der Linden, Jürgen K. Müller: *Composing Product Families from Reusable Components*, Bonnie Melhart, Jerzy Rozenblit (eds.) Proceedings 1995 International Symposium and Workshop on Systems Engineering of Computer Based Systems, IEEE, pp. 35 – 40 (1995)
- [LM95b] Frank van der Linden, Jürgen K. Müller: *Creating Architectures with Building Blocks*, IEEE Software, Nov. 1995
- [LM95c] Frank van der Linden, Jürgen K. Müller: *Software Architecture with the Building Block Method*, IST report RWB-506-re-95055
- [LM95d] Frank van der Linden, Jürgen K. Müller: *Architectural Elements of the Building Block Method*, IST report RWB-506-re-95046
- [LM97] Frank van der Linden, Jürgen K. Müller: *Virtual Processors*, Nat.Lab. Technical Note 012/97
- [MF93] Charles R. Morris, Charles H. Ferguson: *How Architecture Wins Technology Wars*, Harvard Business Review, March - April 1993
- [MHM98] Jacques Meekel, Thomas B. Horton, Charlie Mellone: *Architecting for Domain Variability*, Second Int'l Workshop on development and Evolution of Software Architectures for Product Families, LNCS 1429, Springer, Berlin, 1998
- [Mil85] John A Mills: *A Pragmatic View of the System Architect*, Communications of the ACM, Vol. 28, No. 7, July 1985
- [ML97] Marc H.Meyer, Alvin P. Lehnerd: *The Power of Product Platforms - Building Value and Cost Leadership*, The Free Press, 1997
- [Mon00] Richard Monson-Haefel: *Enterprise Java Beans - 2nd Edition*, O'Reilly, 2000
- [Mul98] Gerrit Muller: *Systeem ontwerper een twintig koppig monster?* personal communication, 1998
- [Mul02] Gerrit Muller: *Requirements Capture by the System Architect*, <http://www.extra.research.philips.com/natlab/sysarch/RequirementsPaper.pdf>, 2002
- [Mül95] Jürgen K. Müller: *Integrating Architectural Design into the Development Process*, Bonnie Melhart, Jerzy Rozenblit (eds.) Proceedings 1995 International Symposium and Workshop on Systems Engineering of Computer Based Systems, IEEE, pp. 114 – 121 (1995)

- [Mül97] Jürgen K. Müller: *Feature-Oriented Software Structuring*, Proceedings of CompSAC'97, pp. 552-555, August 1997
- [Mül99] Jürgen K. Müller: *Aspect Design with the Building Block Method*, in Patrick Donohoe (Ed.): *Software Architecture, TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, 1999, pp. 585-601
- [MSG96] Randall R. Macala, Lynn D. Stuckey, David C. Gross: *Managing Domain-Specific, Product-Line Development*, IEEE Computer, May 1996
- [P1471] IEEE: *Draft Recommended Practise for Architectural Description*, IEEE P1471/D5.2, 1999
- [Par72] David L. Parnas: *On the Criteria to be used in Decomposing Systems into Modules*, Communications ACM 15, 1972, pp. 1053 – 1058
- [Par76] David L. Parnas: *On the Design and Development of Program Families*, IEEE Transactions on Software Engineering, March 1976, pp. 1-9
- [Par79] David L. Parnas: *Designing Software for Ease of Extension and Contraction*, IEEE Transactions on Software Engineering, March 1979, pp. 128-138
- [PC86] David L. Parnas, Paul C. Clements: *A Rational Design Process: How and Why to Fake It*, IEEE Transactions of Software Engineering, Vol. SE-12 No. 2, February 1986
- [Per94] Dewayne E. Perry: *Dimensions of Software Evolution*, International Conference on Software Maintenance 1994, Victoria BC, September 1994
- [Pla99] David S. Platt: *Understanding COM+ - The Architecture for Enterprise Development Using Microsoft Technologies*, Microsoft Press, 1999
- [Pla01] David S. Platt: *Introducing Microsoft .Net*, Microsoft Press, 2001
- [Pro99] Ben J. Pronk: *Medical Product Line Architectures*, in Patrick Donohoe (Ed.): *Software Architecture, TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, 1999, pp. 357-367
- [RBP\*91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorenson: *Object-Oriented Modeling and Design*, Prentice Hall 1991

- [RE99] Andreas Rosel, Karin Erni: *Experiences with the semantic Graphics Framework*, in Mohamed F. Fayad, Douglas C. Schmidt, Ralph E. Johnson: *Implementing Application Frameworks*, Wiley, 1999
- [Rec91] Eberhardt Rechtin: *Systems Architecting - Creating & Building Complex Systems*, Prentice Hall 1991
- [Ree96] Trygve Reenskaug: *Working with Objects - The OOram Software Engineering Method*, Manning Publications 1996pf
- [Ren97] Klaus Renzel: *Error Handling - A Pattern Language*, sd&m, 1997
- [RF96] Muthu Ramachandran, Wolfgang Fleischer: *Design for Large Scale Software Reuse: An Industrial Case Study*, 4th International Conference on Software Reuse, Orlando, Florida, April 1996
- [RJ96] Don Roberts, Ralph Johnson: *Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks*, in Martin, Riehle, Buschmann, Vlissides (Eds.), *Pattern Languages for Program Design 3*, Addison-Wesley, 1997
- [RM97] Eberhardt Rechtin, Mark W. Maier: *The Art of Systems Architecting*, CRC Press 1997
- [Rum94] James Rumbaugh: *The OMT Process*, Rational Whitepapers at [www.rational.com](http://www.rational.com) May 1994
- [RWL96] Trygve Reenskaug, Per Wold, Odd Arild Lehne: *Working with Objects - The OOram Software Engineering Method*, Manning, 1996
- [Sch97] Hans Albert Schmid: *Systematic Framework Design by Generalization*, CACM Vol. 40, No. 10, October 1997, pp. 48-51
- [SG96] Mary Shaw and David Garlan: *Software Architecture - Perspectives on an Emerging Discipline*, New Jersey 1996
- [SGM\*92] Bran Selic, Garth Gullekson, Jim McGee, Ian Engelberg: *ROOM: An Object-Oriented Methodology for Developing Real-Time Systems*, Proceedings of the 5th International Workshop on Computer-Aided Software Engineering (CASE 92), pp. 230-240, 1992
- [SNH95] Dilip Soni, Robert L. Nord, and Christine Hofmeister: *Software Architecture in Industrial Applications*, Proceedings of the International Conference on Software Engineering (ICSE'95), Seattle 1995
- [Szy92a] Clemens Alden Szyperski: *Insight ETHOS: On Object-Orientation in Operating Systems*, Dissertation ETH Zürich No. 9884, 1992

- [Szy98] Clemens Szyperski: *Component Software - Beyond Object-Oriented Programming*, Essex 1998
- [Szy00] Clemens Szyperski: *Modules and Components - Rivals or Partners?*, in [BGP00]
- [TOH99] Peri Tarr, Harald Ossher, William Harrison, Stanley Sutton: *N Degrees of Separation: Multi-Dimensional Separation of Concerns*, Proceedings of the International Conference on Software Engineering (ICSE'99), May, 1999
- [X700] ITU: *Management Framework for Open Systems Interconnection (OSI) for CCITT Applications*, Recommendation X.700, September 1992
- [X731] ITU: *Information Technology - Open Systems Interconnection - Systems Management: State Management Function*, Recommendation X.731, January 1992
- [YC79] Edward Yourdon, Larry L. Constantine: *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice Hall 1979
- [Web13] Noah Porter: *Webster's Revised Unabridged Dictionary*, Version published 1913 by the C. & G. Merriam Co., Springfield, Mass
- [Wei88] Gerald M. Weinberg: *Rethinking Systems Analysis & Design*, Dorset House Publishing, 1988
- [Wie96] Roel Wieringa: *Requirements Engineering - Frameworks for Understanding*, John Wiley, 1996
- [Wie98a] Roel Wieringa: *Traceability and Modularity in Software Design*, Ninth IEEE International Workshop on Software Specification and Design, 1998
- [Wie98b] Roel Wieringa: *A Survey of Structured and Object-Oriented Software Specification Methods and Techniques*, ACM Computing Surveys, Vol. 30, No. 4, December 1998
- [Wij00] Jan Gerben Wijnstra: *Supporting Diversity with Component Frameworks as Architectural Elements*, 22nd International Conference on Software Engineering, Limerick, 2000, pp. 51-60
- [Wij01] Jan Gerben Wijnstra: *Quality Attributes and Aspects in a Medical Imaging Product Family*, Hawai, Proceedings of HICSS-34, January 2001
- [Wir95] Niklaus Wirth: *A Plea for Lean Software*, IEEE Computer, February 1995

---

# Index

Bold numbers give the page of the definition.

## A

architecting model. . . . .	14
architectural meta-model. . . . .	12, <b>149</b>
architectural skeleton . . . . .	38, <b>126</b> , 141
architectural style . . . . .	51, 156
architecture	
control architecture. . . . .	165
functional architecture . . . . .	164
product family architecture. . . . .	3, 4, 6, <b>11</b> , 38, 139–146, 192
software architecture. . . . .	<b>10</b> , 150, 159, 192, 201
system architecture . . . . .	<b>10</b> , 79, 153, 164–168, 204
aspect. . . . .	44, <b>67</b>

## B

behaviour. . . . .	17, 18, 35, 51, 56, 89
blackboard. . . . .	43, 51, 101, 156
Building Block . . . . .	<b>95</b>
generic BB . . . . .	37, 97, <b>113</b> , 192, 222, 249, 253
specific BB . . . . .	37, <b>113</b>

## C

call-back . . . . .	<b>102</b>
component	
hardware component . . . . .	197
software component . . . . .	10, <b>95</b> , 139
component framework . . . . .	26, 29, 37, 53, 73, 97, <b>113</b> , 119, 126, 222, 249, 253
component model . . . . .	95, <b>103</b> , 187, 191, 217–221
conceptual integrity. . . . .	29, 42, <b>54</b> , 74, 142, 193

configurability . . . . . 3, 29, 62, 95, 121, 127, 216  
 configuration management . . . . . 13  
 connector . . . . . 101–102, 118, 158  
 COTS . . . . . 52

## D

design  
   aspect design . . . . . 34–35, 67–85, 175–177, 204–214  
   composability design . . . . . 36–38, 95–148, 171–175, 216–231  
   concurrency design . . . . . 35–36, 87–93, 214–216  
   deployability design . . . . . 38, 128–130, 177  
   object design . . . . . 34, 55–65, 169–171, 203–204  
 design artifacts . . . . . 23, 39–43  
 design dimension . . . . . 44–48, 95, 148, 149, 193  
 design tasks . . . . . 33–53  
 diversity . . . . . 138

## E

evolution . . . . . 18, 47, 60, 139, 143, 147, 185  
 extensibility . . . . . 3, 50, 142, 166

## F

fault management . . . . . 13, 27, 64, 75, 92, 120  
 feature . . . . . 19, **135**  
 future-proof . . . . . 11, 147

## I

incremental . . . . . 4, 11, 103, 111, 115, 127, 212  
 incremental development . . . . . 29  
 incremental integration . . . . . 3, 187  
 incremental layer . . . . . 37, **107**, 123, 126  
 incrementality . . . . . 29, 125, 148  
 inheritance . . . . . 56, 115  
 interface . . . . . 52, 99–103, 124  
   abstraction interface . . . . . 99  
   call-back interface . . . . . 102, 124  
   first-access interface . . . . . 124

open implementation interface . . . . .	100
provides interface . . . . .	<b>99</b>
requires interface . . . . .	<b>99</b>

**L**

layer . . . . .	<b>104</b>
layered development process . . . . .	29, 185
layering . . . . .	28, 59, 104–111, 115, 173

**M**

meta-model . . . . .	12
method . . . . .	<b>11</b>
model . . . . .	12

**O**

object	
application domain object . . . . .	17, 34, <b>55</b>
design object . . . . .	34, <b>55</b>
domain-induced object . . . . .	34, 44, <b>55</b>
hardware domain object . . . . .	34, <b>55</b>
implementation object . . . . .	34, <b>55</b>

**P**

performance . . . . .	48
plug-in . . . . .	37
product family . . . . .	3, 26, 47, 80, 95, <b>135</b> , 135–148, 182, 195
product line . . . . .	135

**R**

reliability . . . . .	49, 166, 167, 199, 207
reuse . . . . .	3, 11, 74, 151, 155, 193

**S**

SIG . . . . .	119
system . . . . .	<b>7</b> , 13

system control .....	13
system management .....	13

**T**

testability .....	50
testing .....	3, 104, 109, 154, 186, 200
thread .....	35, 39, 44–47, 48, 62, 87–93
logical thread .....	<b>88</b>
physical thread .....	<b>88</b>