MODELLING GLOBAL BEHAVIOUR WITH SCENARIOS IN OBJECT-ORIENTED ANALYSIS

THÈSE N^{O} 1655 (1997)

PRÉSENTÉE AU DÉPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Dorothea BERINGER

lic.phil.nat. Informatikerin, Université de Berne
originaire d'Adelboden (BE), Elgg (ZH) et Ellikon (ZH)

acceptée sur proposition du jury:

Prof. A. Strohmeier, directeur de thèse
Dr Ph. Dugerdil, corapporteur
Prof. Y. Pigneur, corapporteur
Prof. S. Spaccapietra, corapporteur

Lausanne, EPFL
1997

Modelling Global Behaviour with Scenarios in Object-Oriented Analysis

THESE No 1655 (1997)

présentée au Département d'Informatique
ECOLE POLYTECHNIQUE FEDERALE DE LAUSANNE
pour l'obtention du grade de docteur ès sciences

par

Dorothea Beringer
Informatikerin lic. phil. nat.
originaire de Adelboden (BE), Elgg et Ellikon (ZH)

acceptée sur proposition du jury:

Prof. A. Strohmeier, rapporteur
Dr. Ph. Dugerdil, corapporteur
Prof. Y. Pigneur, corapporteur
Prof. S. Spaccapietra, corapporteur

Lausanne, EPFL
Mai, 1997

to Jürg

Abstract

The first object-oriented analysis methods focused on the specification of the classes of a system and on the static relationships between them. Dynamic relationships between classes and the functional view of the system as a whole were neglected, and no models were offered for capturing system requirements. This changed with the publication of use case driven approaches such as OOSE (1992) and Fusion (1994). Modelling global behaviour by scenarios in both requirements analysis and design has since been adopted by many object-oriented methods. Scenarios are also called use cases, system operations or business processes, and they are modelled using different notations.

Some of the analysis methods that use scenarios have common characteristics, namely *i)* the similarity of the relationship between the entities of a data model and the scenarios of a flat scenario model with a matrix, and *ii)* the assumption that the externally visible behaviour of the system can be subdivided into more or less independent scenario types. In the following, we will refer to these characteristics by the term matrix approach. While in many projects the matrix approach has been used successfully, several difficulties arise when more complex systems are modelled: relationships and similarities between different scenario types cannot be expressed, the dependencies between scenario types are not modelled, only one abstraction level can be represented, and the apparently seamless transition from the analysis to the design model may result in a low quality object model with a strong bias towards data modelling.

These difficulties lead us to propose an enhanced scenario modelling technique (called SEAM) which overcomes some of the weaknesses of the matrix approach. This modelling technique includes composition, aggregation, specialisation and extension hierarchies of services, and is based on the paradigm of interacting objects (which can be atomic objects, subsystems or systems) offering services. Scenario types, showing the possible interaction sequences for a specific service, can be modelled on several abstraction levels, and can describe the services of any kind of object (and thus also the global behaviour of a whole system) from both an internal and external point of view. We describe the concepts and the notation of SEAM, and we show how it can be integrated into the Fusion method.

The difficulties that may arise in projects using methods based on the matrix approach are not only due to the limitations of scenario modelling techniques. A major factor is the often contradictory definition of the analysis model goals, which leads to clashes of intent. Therefore we discuss the nature of such intent clashes and analyse how the different software development methods deal with them.

Finally we give an overview of the various notations and basic concepts used by different scenario modelling techniques, and we provide summaries of current, mostly object-oriented, approaches to modelling global behaviour.

Résumé

Les premières méthodes d'analyse par objets se concentraient sur la spécification des classes du système et sur les relations statiques entre celles-ci. Les relations dynamiques et la vue fonctionnelle du système entier étaient négligées, et aucun modèle n'exprimait les exigences envers le système considéré dans son ensemble. La situation a changé avec l'apparition de méthodes comme OOSE (1992) et Fusion (1994), et beaucoup de méthodes par objets ont adopté aussi bien pour l'analyse que pour la conception une modélisation du comportement global fondée sur les scénarios.

Dans les méthodes d'analyse qui utilisent des scénarios, ceux-ci sont également appelés cas d'utilisation, opérations du système ou fonction du système, et ils sont représentés au moyen de diverses notations. On trouve cependant des caractéristiques communes à plusieurs de ces méthodes, en particulier: *i)* l'existence d'un lien matriciel entre les entités du modèle des données et les scénarios du modèle des scénarios, et *ii)* l'hypothèse que le comportement du système peut être divisé en des scénarios relativement indépendants. Par la suite, nous désignons ces caractéristiques par le terme d'approche par matrice. Quoique l'approche par matrice ait été utilisée avec succès dans beaucoup de projets, plusieurs difficultés surviennent lors de la modélisation de systèmes plus complexes: on ne peut pas exprimer les relations et similarités entre différents scénarios, on ne peut pas modéliser les dépendances entre scénarios, on ne peut représenter qu'un seul niveau d'abstraction, et la transformation directe du modèle d'analyse en modèle de conception peut conduire à un modèle des objets de piètre qualité et trop influencé par une vision statique des données.

Nous proposons une technique de modélisation à base de scénarios (appelée SEAM) qui surmonte quelques-unes des faiblesses de l'approche par matrice. Cette technique inclut la modélisation des hiérarchies de composition, d'agrégation, de spécialisation et d'extension des services, et elle est basée sur un paradigme d'objets qui interagisssent et offrent leur services. Ces objets peuvent être des objets atomiques, des sous-système ou des systèmes. Les scénarios montrent - aussi bien d'un point de vue intérieur qu'extérieur - les séquences d'interactions possibles pour chaque service d'un objet quelconque. On peut ainsi décrire le comportement global du système entier de la même manière que le comportement d'un sous-système, tout en ayant plusieurs niveau d'abstraction pour les scénarios. Nous décrivons les concepts et la notation de SEAM et nous montrons son intégration dans la méthode Fusion.

Les difficultés rencontrées dans des projets qui utilisent des méthodes basées sur l'approche par matrice ne sont pas toutes causées par les restrictions des techniques de modélisation. On trouve également fréquemment une définition contradictoire des buts des modèles d'analyse, ce qui conduit à des conflits entre différentes intentions. Nous discutons des contradictions qui résultent de ces différents buts et nous montrons comment différentes méthodes traitent ces contradictions.

Finalement, nous présentons un aperçu des notations et concepts utilisés dans différentes techniques de modélisation par scénarios, et nous résumons plusieurs approches actuelles de modélisation du comportement global.

Zusammenfassung

Die ersten objektorientierten Analysemethoden unterstützten in erster Linie die Spezifikation der Klassen eines Softwaresystems und der statischen Beziehungen der Klassen untereinander. Die dynamischen Beziehungen zwischen Klassen und vor allem das globale Verhalten des gesamten Systems wurden kaum modelliert. Dies änderte sich, als Methoden wie OOSE (1992) und Fusion (1994) publiziert wurden. Seither benutzen viele objektorientierte Methoden Scenarios, um das globale Verhalten eines Systems zu modellieren. Scenarios werden auch Use-cases, Systemoperationen oder Geschäftsvorfälle genannt, und es gelangen unterschiedliche Notationen zur Anwendung.

Einige der Analysemethoden, welche Scenarios verwenden, haben gemeinsame Merkmale, nämlich *i)* die Entitäten des Datenmodells und die Scenariotypen des Verhaltensmodells bilden eine Matrix; dies ist möglich, weil beides flache Modelle sind, *ii)* und es wird davon ausgegangen, dass das externe globale Verhalten eines Systems in voneinander mehr oder weniger unabhängige Scenariotypen aufgegliedert werden kann. Im folgenden bezeichnen wir diese Merkmale als die Eigenschaften des sogenannten Matrix-Approach. Methoden basierend auf dem Matrix-Approach werden zwar in vielen Projekten erfolgreich angewandt, aber bei der Modellierung komplexerer Systeme können Schwierigkeiten auftreten: Aehnlichkeiten, Abhängigkeiten und weitere Beziehungen zwischen verschiedenen Scenariotypen können nicht dargestellt werden, nur eine einzige Abstraktionsebene kann modelliert werden, und der anscheinend so problemlose und direkte Uebergang vom Analyse- zum Designmodell führt oftmals zu einem Objektmodell von schlechter Qualität und zu einseitiger Ausrichtung auf die Daten des Systems.

Diese Schwierigkeiten haben uns dazu veranlasst, eine Modellierungstechnik zu entwikkeln (wir nennen sie SEAM), die einige der Probleme des Matrix-Approach löst. Diese Modellierungstechnik basiert auf dem Paradigma interagierender Objekte. Dabei kann ein Objekt ein atomares Objekt, ein Subsystem oder ein gesamtes System sein, und jedes Objekt bietet sogenannte Dienste an. SEAM enthält Konzepte und Notationen zur Komposition, Aggregation, Spezialisierung und Erweiterung von Objektdiensten beliebiger Objekte. Die Objektdienste und ihre Interaktionen werden mit Hilfe von Scenariotypen modelliert. Dies geschieht auf verschiedenen Abstraktionsebenen und sowohl von einem externen als auch von einem internen Blickwinkel auf das Objekt, welches den Dienst anbietet.

Nicht alle Schwierigkeiten, welche durch die Verwendung des Matrix-Approach entstehen können, lassen sich auf Limitationen in der Modellierung von Scenarios zurückführen. Ein weiterer sehr wichtiger Faktor sind widersprüchliche Ziele eines Analysemodells. Da sich gewisse Schwierigkeiten in der Zieldefinition nicht vermeiden lassen, gehen wir auch darauf ein, wie verschiedene Methoden damit umgehen.

Schliesslich geben wir auch eine Uebersicht über grundlegende Konzepte und Notationen diverser, meist objektorientierter Methoden sowie Zusammenfassungen der meisten von uns untersuchten Modellierungstechniken.

Acknowledgements

I am very happy to take the opportunity to thank all the colleagues and friends who have contributed to this thesis. In particular I would like to thank:

- Prof. Alfred Strohmeier for offering me the opportunity to work in the Laboratory of Software Engineering at the Swiss Federal Institute of Technology in Lausanne, and for the supervision of my thesis.
- Prof. Yves Pigneur, Prof. Stefano Spaccapietra and Dr. Philippe Dugerdil for accepting to be the referees of my thesis and for many helpful comments.
- Dr. Philippe Dugerdil for several interesting discussions on use cases and scenarios, and for his detailed feedback on my ideas.
- All the colleagues in the Laboratory of Software Engineering for the collaboration in the Software Engineering course and for helping me to improve my French. In particular I thank Gabriel Eckert for proofreading all my French texts.
- My dear friends, Cornelia and Peter Berthold, Cornelia and Martin Imboden, Eva and Lorenz Malmstroem, and Cornelia and Stefan Schranz who did not only encourage me to go back into research and to start a PhD thesis, but who also supported me and waited faithfully when weekends and evenings were eaten up by my work.
- A warm “Thank you!” goes to my parents, who worked hard to make it possible for me to get a higher education and to whom I am deeply indebted in many respects.
- I would also like to thank the Swiss National Foundation which has supported the research for this thesis under the grant number 2000-043648.95.

Last but not least, I am deeply indebted to my dear husband Jürg for his continuous love, encouragement and support.

Table of Contents

Abstract i

Résumé ii

Zusammenfassung iii

Acknowledgements iv

Table of Contents v

List of Figures ix

Chapter 1 Introduction 1

Chapter 2 The modelling of global behaviour in current object-oriented analysis methods 9

2.1 Modelling global behaviour: the usage of use cases, scenarios and interactions diagrams 9

2.1.1 Introduction 9

2.1.2 The global behaviour of an object-oriented system 10

2.1.2.1 Global versus local behaviour 10

2.1.2.2 Internal versus external view of the global behaviour 11

2.1.2.3 Conceptual versus technical interactions 11

2.1.2.4 Requests versus notifications 13

2.1.2.5 Design versus analysis 16

2.1.3 Modelling global behaviour by modelling scenarios 17

2.1.3.1 Scenarios of one object versus scenarios between several objects 17

2.1.3.2 Scenario instances versus scenario types 18

2.1.3.3 Scope of a scenario type, classification schemes 20

2.1.4 Notational limits in modelling interacting objects 24

2.1.4.1 Modelling the dynamics versus modelling the statics 25

2.1.4.2 Dynamic models: transitions between states versus interactions between objects 26

2.1.4.3 Interactions between objects: time-line diagrams versus object diagrams 29

2.1.4.4 Some further aspects of interaction diagrams 31

2.1.5 Summary 33

2.2 The Matrix Approach 35

2.2.1 Characteristics of the matrix approach 35

2.2.1.1 Notation 35

2.2.1.2 Process 39

2.2.2 Difficulties with the matrix approach 41

2.2.2.1 Flat list of scenario types 41

2.2.2.2 The matrix approach supports the modelling of only one abstraction level 44

2.2.2.3 System boundaries 46

2.2.2.4 Difficulties in the transition to a good oo-design 47

2.2.2.5 Is the matrix approach really object-oriented? 50

2.2.3 Reasons for these difficulties 51

2.3 The relationship cardinality domination 52

Chapter 3 Goals of Analysis 57

3.1 What is an analysis model? 57

3.1.1 Motivation, notation, intent and content of models 58

3.1.2 The term “analysis” from the perspective of the problem solving cycle reference model 59

3.1.2.1 The problem solving cycle 59

3.1.2.2 Applying the problem solving cycle to software engineering 61

3.1.2.3 Where does the term “analysis” fit into the problem solving cycle? 63

3.1.2.4 Consequences 65

3.1.3 Criteria for defining the goals of analysis models 66

3.1.3.1 Model of the current system versus model of the future system 66

3.1.3.2 Application-oriented versus reuse-oriented 67

3.1.3.3 “Real world” model or abstracting a software system 67

3.1.3.4 Technology independent models 68

3.1.3.5 External versus internal model 72

3.1.3.6 Complete and unambiguous models versus essential models 73

3.1.3.7 Targeted audience 74

3.1.3.8 Consequences 75

3.2 Intent clashes 77

3.2.1 Difficulties with the ideal analysis model 77

3.2.1.1 The ideal analysis model 77

3.2.1.2 First misconception: stable model 78

3.2.1.3 Second misconception: objective real world model 78

3.2.1.4 Third misconception: initial final high-level view 79

3.2.2 The two intent clashes 80

3.2.3 Approaches to handle these intent clashes 82

3.2.3.1 The idealistic approach 83

3.2.3.2 The one-model approach 83

3.2.3.3 The two-model approach 85

3.2.3.4 Consequences 86

Chapter 4 SEAM: an Enhanced Scenario Modelling Technique 87

4.1 The starting point 87

4.1.1 Models of the system 88

4.1.1.1 Only one model 88

4.1.1.2 Evolution of the final object model 90

4.1.1.3 Viewpoints 91

4.1.2 The object-oriented system 92

4.1.2.1 Excursus: Groups of objects in various methods 92

4.1.2.2 The system of interacting objects in SEAM 95

4.1.2.3 Implications of this approach 99

4.2 Services and scenarios ... 102
4.2.1 Services ... 102
4.2.2 Various kinds of interactions ... 105
4.2.3 Scenarios ... 108
4.2.4 Notation: schemas for objects and services ... 110
4.2.5 Notation: interaction diagrams ... 112
4.2.5.1 Specifying interactions ... 113
4.2.5.2 Basic notations for interaction diagrams ... 114
4.2.5.3 Modelling scenario types of services ... 118
4.2.5.4 Grouping and hiding objects and interactions ... 121
4.2.5.5 Further notations for interaction diagrams ... 124
4.2.5.6 Why interaction diagrams? ... 128
4.2.6 Detailing scenario types ... 129

4.3 Hierarchies of services ... 133
4.3.1 Composition hierarchies ... 134
4.3.2 Aggregation hierarchies ... 136
4.3.2.1 Complete aggregation of services ... 136
4.3.2.2 Partial aggregation of services ... 138
4.3.2.3 Notation: interaction diagrams for aggregate services ... 139
4.3.3 Inheritance hierarchies ... 142
4.3.3.1 Specialisation and generalisation of services ... 142
4.3.3.2 Notation: interaction diagrams for specialised services ... 144
4.3.3.3 Extending services ... 145
4.3.3.4 Notation: interaction diagrams for extended services ... 148
4.3.4 Service diagrams ... 148
4.3.4.1 Aggregation and composition graphs ... 148
4.3.4.2 Inheritance graphs ... 150
4.3.4.3 Context diagrams ... 151
4.3.5 Excursus: comparison to other methods ... 152

4.4 Life-cycles of objects ... 155
4.4.1 Order of services and essential states ... 155
4.4.2 Notations for the object life-cycles ... 158

4.5 Transitions between scenario models ... 163
4.5.1 Transformations of interactions ... 163
4.5.2 Transformations of scenario types ... 166
4.5.3 Transformations of service hierarchies ... 167
4.5.4 Why transitions between scenario models? ... 170

4.6 Developing scenario models ... 172
4.6.1 The development process ... 172
4.6.2 Completeness of scenario models ... 175
4.6.3 Consistency of scenario models ... 178

4.7 Summary ... 182
4.7.1 Summary of the basic concepts ... 182
4.7.2 Reasons for the concepts as defined in SEAM ... 188

Chapter 5 Case Studies ... 193

5.1 Mail Order Firm ... 193

5.2 ECO-System ... 203

Chapter 6 Summary and Outlook ... 213

Appendix A Overview of some methods ... 217
A.1 Introduction ... 217
A.2 OOSE ... 218
A.3 Fusion ... 222
A.4 Modern Structured Analysis ... 228
A.5 OBA ... 230
A.6 FORAM ... 232
A.7 BON ... 235
A.8 Booch and OMT ... 239
A.9 Further notations for modelling global behaviour ... 241
A.9.1 Path Expression Groups ... 241
A.9.2 Scenario trees ... 243
A.9.3 Composition of scenarios based on statecharts ... 245
A.9.4 CRC cards ... 245
A.9.5 Requirements scripts in OSMOSIS ... 246
A.9.6 Storyboarding ... 247
A.9.7 Business processes (“Geschäftsvorfälle”) ... 248
A.9.8 Extending OOSE by use case levels ... 249
A.9.9 Use case maps ... 251
A.9.10 M.E.R.o.DE ... 251
A.9.11 Behavioural models of Kowal ... 252
A.9.12 Further approaches ... 253

Appendix B Examples of diagrams from various methods ... 255

Appendix C Enhancing Fusion ... 279

References ... 285

Curriculum Vitae ... 291

List of Figures

Figure 1: Overview of the thesis 4
Figure 2: External versus internal views 11
Figure 3: Conceptual and technical interactions for different abstraction levels 13
Figure 4: Modelling on several abstraction levels 14
Figure 5: Using notifications for the external and requests for the internal view 16
Figure 6: Modelling scenario instances and types, part 1 21
Figure 7: Modelling scenario instances and types, part 2 22
Figure 8: Determining the end of a scenario 23
Figure 9: Transitions between states versus interactions between objects 27
Figure 10: Dynamic model of a system: life-cycles of objects or scenarios of system functions 28
Figure 11: Overview of the concepts for modelling global behaviour 34
Figure 12: Modelling global behaviour from an external viewpoint. 36
Figure 13: The matrix between the object model and the scenario model. 38
Figure 14: Modelling complex patterns 41
Figure 15: Abstraction levels 45
Figure 16: Composed systems 47
Figure 17: Objective, content and intent of a model 58
Figure 18: The problem solving cycle 60
Figure 19: Goals of analysis models 66
Figure 20: The ideal analysis model 77
Figure 21: Seamless expansion of the analysis model 79
Figure 22: Initial analysis model versus final high-level view 80
Figure 23: The first intent clash 82
Figure 24: Various views of the object model. 89
Figure 25: Service calls in a white-box subsystem 97
Figure 26: Service calls in a black-box subsystem 97
Figure 27: Properties of services 102
Figure 28: Interaction diagram for modelling scenario types 115
Figure 29: Notations for interaction diagrams 117
Figure 30: External view of a system service 118
Figure 31: Internal view of a system service 119
Figure 32: External view of an atomic service 119
Figure 33: Service triggered by time-out 120
Figure 34: Service triggered by two different interaction types 120
Figure 35: Service triggered by an interaction from the server object to an agent 121
Figure 36: Object groups in interaction diagrams 122
Figure 37: Indirect interactions 123
Figure 38: One-agent view for a system service 124
Figure 39: One-agent view for an atomic service 125
Figure 40: Control bars 125
Figure 41: Explicit return interactions 126
Figure 42: Object creation and lifetime 127
Figure 43: Two-dimensional object diagram 128
Figure 44: Detailing scenario types: an example, part 1 131
Figure 45: Detailing scenario types: an example, part 2 132
Figure 46: Detailing scenario types: an example, part 3 132
Figure 47: Internal view of a service with service brackets 134
Figure 48: Internal views of a black-box subsystem 135

Figure 49: Deriving internal and external view for an aggregate service 140
Figure 50: Making_phone_call as complete aggregate service 141
Figure 51: Detailing versus specialising scenario types 145
Figure 52: Extending services 146
Figure 53: Modelling making a phone call by an extension hierarchy 147
Figure 54: Aggregation graphs for services 149
Figure 55: Inheritance graphs for services 150
Figure 56: A context diagram 151
Figure 57: Essential states 157
Figure 58: Life-cycles in pseudo-code notation 159
Figure 59: A state transition diagram 161
Figure 60: Decomposition of state transition diagrams 162
Figure 61: Transitions between scenario models 164
Figure 62: Selection criteria in interaction specifications 165
Figure 63: The service deliver_drums with an alternative design 166
Figure 64: Transforming a complete aggregate service 168
Figure 65: Transforming a generalized service 169
Figure 66: Transforming an extension into a specialisation 170
Figure 67: Overview of the relationships among services 183
Figure 68: Metamodel for SEAM 184
Figure 69: Summary of some terms as used in SEAM 186
Figure 70: Overview of the documentation of a scenario model 187
Figure 71: Schema of the system Mail_Order_Firm 193
Figure 72: Context diagram of the system Mail_Order_Firm 194
Figure 73: Schemas of the services make_orders_for_suppliers and order 194
Figure 74: Specialisation hierarchy of the service Mail_Order_Firm :: order (external view) 195
Figure 75: Service Mail_Order_Firm :: make_orders_for_suppliers (external view) 196
Figure 76: Second schema for the system Mail_Order_Firm 196
Figure 77: Internal view of the specialisation hierarchy for the service order, part 1 197
Figure 78: Internal view of the specialisation hierarchy for the service order, part 2 197
Figure 79: Internal view of the specialisation hierarchy for the service order, part 3 198
Figure 80: Internal view of the service order without specialisation 199
Figure 81: External view of the service order_advance_payment of the subsystem Order_System 200
Figure 82: Schema of the subsystem Order_System 200
Figure 83: Object model of the subsystem Order_System 201
Figure 84: Schema for the object Order 201
Figure 85: Schema for the elementary service taking_down_order 202
Figure 86: Internal view of the service taking_down_order 202
Figure 87: Schema of the service set_client 202
Figure 88: Context diagram of the system ECO-System 203
Figure 89: Schema of the system ECO-System 204
Figure 90: Schema of the service rearrange_drums 204
Figure 91: Schema of the service deliver_drums 204
Figure 92: Schema of the service get_status 205
Figure 93: External view of the service deliver_drums 205
Figure 94: External view of the service get_status 205
Figure 95: The service deliver_drums as a partial aggregation 206
Figure 96: Schema for an extended service 207
Figure 97: Scenario type for the extended service deliver_drums_exception 207
Figure 98: Aggregation and inheritance graphs of the service deliver_drums 207
Figure 99: The service deliver_drums as a complete aggregation 208

Figure 100: Aggregation graph of the service deliver_drums 208
Figure 101: Life-cycle of ECO-System 209
Figure 102: Service check_in_drum: user interface design 210
Figure 103: Service get_status: user interface design 210
Figure 104: Internal view of the service overview_drums 211
Figure 105: Schema for the service overview_drums_for_building 212
Figure 106: Schema for the service get_types 212
Figure 107: Object model with communication associations (arrows with no labels) in OOSE [Jacobson92, page 189] 255
Figure 108: Description of a use case in OOSE [Jacobson92, page 349] 256
Figure 109: Modelling the relationship between abstract and concrete use cases in OOSE [Jacobson92, page 343] 256
Figure 110: Interaction diagram in OOSE (example with two interface objects) [Jacobson92, page 381] 257
Figure 111: Interaction diagram in OOSE (output signals) [Jacobson92, page 218] 258
Figure 112: Event trace diagram in Fusion (analysis model) [Coleman94, page 47] 258
Figure 113: Life-cycle expressions in Fusion (analysis model) [Coleman94, page 33] 259
Figure 114: Scenario diagram of extended Fusion showing a use case instance with a sub use case [Coleman95] 259
Figure 115: System interaction graph in extended Fusion [Coleman95] 259
Figure 116: Schema of a system operation in Fusion (analysis model)[Coleman94, page 31] 260
Figure 117: Object interaction graph in Fusion (design model) [Coleman94, page 75] 260
Figure 118: Decomposition of object interaction graphs in Fusion [Coleman94, page 79] 261
Figure 119: Context model, message table and task cards in FORAM [Graham94b] 261
Figure 120: Event trace in FORAM [Graham94b] 262
Figure 121: Use case map for a design pattern [Buhr96] 262
Figure 122: Event list in Modern Structured Analysis [Beringer92c] 263
Figure 123: Context diagram in Modern Structured Analysis [Beringer92c] 263
Figure 124: Scripts in OBA [Rubin92] 264
Figure 125: Textual description of a scenario [Kilberth93, page 99] 264
Figure 126: Event charts and scenario charts in BON [Walden95, pages 253-255] 265
Figure 127: Object Scenario in BON [Walden95, page 259] 265
Figure 128: Object Scenario in BON (with grouping of objects into several subtasks) [Walden95, page 114] 266
Figure 129: State chart as used in Booch [Booch94, page 207] 266
Figure 130: Interaction diagrams in Booch [Booch94, page 218] 267
Figure 131: Object diagram for one mechanism [Booch94, page 402] 267
Figure 132: Add-relationships between use cases in OMT [Rumbaugh94b] 267
Figure 133: Description of a use case in OMT [Rumbaugh94b] 268
Figure 134: Event flow diagram in OMT [Rumbaugh95] 268
Figure 135: Two different kinds of event trace diagrams in OMT [Rumbaugh95] 269
Figure 136: Concurrent object interaction diagram in OMT [Rumbaugh95b] 269
Figure 137: Object interaction diagram showing the design of the operation ‘redisplay’ [Rumbaugh95b] 270
Figure 138: Operation specification in OMT [Rumbaugh95b] 270
Figure 139: Object-event table and structure diagrams in M.E.R.O.DE [Dedene94] 270
Figure 140: Scenario tree and grammar of the user-view of the ‘caller’ [Hsia94] 271
Figure 141: Scenario tree and conceptual state machine of the user-view of the ‘callee’ [Hsia94] 271
Figure 142: Collaboration graph in Responsibility Driven Design [Wirfs90, page153] 272
Figure 143: Class card in [Wilkinson95, page115] 272
Figure 144: Scenario description in [Wilkinson95, page146] 272

Figure 145: Class diagram with main collaboration in [Wilkinson95, page143] 273
Figure 146: Action diagram with brackets, used for the specification of a scenario [Kowal92, page292] 273
Figure 147: Composition of scenarios in [Glinz95] 274
Figure 148: State chart for one scenario in [Glinz95] 275
Figure 149: Event-response list and sample storyboard in OORD [Umphress91] 275
Figure 150: Graphical representation of a “Geschaeftsvorfall” in [Mueller93] 276
Figure 151: Bergen interaction diagram, captures also the inheritance hierarchy of the objects [Baklund95] 277
Figure 152: Description of a group of interacting objects using path expressions (PEG) [Adam94] 277

Chapter 1
Introduction

Motivation

From 1991 until 1994 I worked as a software engineering consultant in a medium sized software company, where we advised customers in the areas of software development methods, quality assurance and project management. In the beginning, our consulting was based on modern structured analysis (MSA) as published in [Yourdan89] and [Brantschen91], but step by step we started switching to object-oriented analysis methods. This was the time when object-oriented development methods were becoming popular, and there was much debate as to what an object-oriented analysis method really is about. On one hand, there were the purists who avoided any bias towards structured methods and started the development process by specifying classes with their attributes and operations. On the other hand, there were the pragmatics who wanted models which could be used for requirements specification, contracts, as a link between the object model and the functional requirements, and as a design-independent starting point for determining the objects (see also [Berard93, page 36] and [Beringer93]). They used structured methods (e.g. certain models of MSA as in [Briod93] or descriptions of business transactions as in [Mueller93]) at the beginning of a project in order to define the external view of the global behaviour and to determine the requirements of the system. These approaches were often quite similar to use cases, but only with the publication of Jacobson's book on OOSE [Jacobson92], were use cases and similar modelling techniques accepted to be genuine object-oriented approaches.

As consultants, we came across several weaknesses and deficiencies in structured analysis methods. Therefore we adapted and enlarged the original analysis model of MSA by introducing some new modelling elements (see [Beringer92b] and [Beringer92c]). For some of the deficiencies we just found work-arounds, which were usually very pragmatic, because there was no time for more thorough research. As everybody else we hoped that newer methods would solve these problems anyway. However, when we learned about OOSE, we were quite surprised to find that its use case model was very similar to what we had done so far, and that many of the difficulties we were having with MSA were not resolved at all. At this point, I started to think about doing some research myself in this area. I started with the following two questions: What should an object-oriented analysis model, which also allows modelling of external system behaviour, resemble? Can the difficulties I had previously encountered with the scenario model of MSA (i.e. with the event response lists and with the high-level data flow diagrams of the business processes) and of OOSE (i.e. with its use case model) be overcome?

History of the thesis

Since I started in 1993 researching the modelling of global behaviour in object-oriented analysis, much has changed in the realm of object-oriented analysis methods. Many methods have adopted, in one form or other, the use cases and interaction diagrams of Jacobson, and 2nd generation methods have been published (e.g. Fusion [Coleman94], [Booch94], 2nd generation OMT [Rumbaugh94]). However, usually these methods do not go much further than what Jacobson proposed initially, and the difficulties with the scenario models remain more or less the same. Only very recently has this begun to change. At the workshop on use cases at the OOPSLA'95 conference as well as in other recent publications, different authors have described deficiencies of the current approaches to modelling global behaviour and have proposed enhancements to scenario modelling. Overall, these publications back up many of my own observations.

Although at the beginning it looked quite easy to propose an approach which would overcome at least some of the weaknesses of current approaches to modelling global behaviour, the path to such an enhanced scenario modelling technique has not proved straightforward, a fact reflected in the present thesis. One of my starting points was to consider the seamlessness claimed for many methods. The results of this research activity are not included in this thesis (they have been published 1994 in [Beringer94]), although they have influenced the enhanced scenario modelling technique (SEAM) proposed here. Another question at the beginning was what object-oriented analysis really is about. While trying to find some basis that I could use for my further research, I gained some insights into the difficulties that arise when using a term without stating its meaning explicitly. I realized that some difficulties we had encountered in modelling global behaviour were not due to weaknesses in the notation but to contradictions in our expectations concerning the analysis model. The resulting research results have been published with some delay in a technical report [Beringer96b] and have become chapter 3 of this thesis.

The investigation of the concepts used in modelling global behaviour was another interim milestone. Though many articles and columns have been published about modelling global behaviour, hardly anyone has tried to clarify the differences and similarities of the various concepts, terms and notations used by the different methods. Therefore I looked at a number of methods and tried to identify the main concepts they offer for modelling global behaviour and for using scenarios. The resulting overview has been published in [Beringer95b] and an updated version of it in [Beringer96], which corresponds to chapter 2.1 and appendix A of this thesis. The results of this investigation have heavily influenced SEAM.

The investigation of various methods also helped to answer another important question: what kind of relationships exist between scenarios? The results were twofold; a first set of relationships that relate scenarios within one scenario model - these have been integrated into the semantics and notation of SEAM - and a second set that relate scenarios belonging to different scenario models and thus play an important role in the process of developing scenario models with SEAM.

The enhanced scenario modelling technique SEAM is not just an enhancement of one specific modelling technique. Rather, SEAM is based on the fundamental insights and concepts gained through the research activities mentioned above. In addition, it has of course also been influenced by some of the methods I investigated and by my personal experiences in modelling global behaviour. This thesis contains a proposal of SEAM as it has evolved from the investigations and case studies carried out during my research. Of course, its large scale validation is yet outstanding.

Overview of the thesis

In chapter 2.1, we start the thesis with an analysis of current approaches to modelling global behaviour. The focus of our discussion is on the usage of use cases, scenarios and interaction diagrams. Chapter 2.2 describes some common characteristics and weaknesses of analysis methods which are based on a matrix between a flat scenario model and a data model; we refer to these characteristics as the matrix approach. We have identified several potential sources for the difficulties with the matrix approach. One of them, the domination of the structure of the object model by the relationships and cardinalities between data elements is explained in chapter 2.3. Another source of difficulties is the existence of contradictory goals and expectations concerning the analysis process. Therefore, in chapter 3, we discuss possible interpretations of the term analysis, we present different goals of analysis models, we describe the intent clashes which may arise due to contradictory goals and misconceptions concerning the development process, and finally we categorize object-oriented development methods according to how they cope with these intent clashes.

In chapter 4 we propose an enhanced scenario modelling technique (SEAM). SEAM is only a modelling technique and not a full fledged method since it does not cover all the aspects a complete method should cover. Our discussion is focused on the concepts and models that represent the global behaviour of a system. First of all this concerns scenarios and services. But models showing scenarios and services cannot be regarded as disconnected from the object model, and therefore we present concepts and notations for the object model as well. Basic assumptions made concerning the object model and the development process are described in chapter 4.1. Chapter 4.2 defines the concepts of SEAM and proposes various hierarchies of services. A summary of the concepts is found in chapter 4.7. The notation of SEAM is explained in chapter 4.3. Chapter 4.4 deals with the evolution of scenario models and describes various possible transitions between scenario models. Some reflections concerning the documentation of scenario models, possible CASE-tool support, consistency and completeness of scenario models and the

process of developing scenario models are found in chapter 4.5 and 4.6. In chapter 5, the description of SEAM is complemented by two partial case studies. Throughout chapter 4, SEAM is described independently of its possible integration into current object-oriented methods. Such an integration is sketched out in appendix C for the Fusion method.

The thesis is rounded off with two appendices. Appendix A gives a short summary of different modelling techniques with respect to their approach to modelling global behaviour. The following methods and publications are considered: OOSE, Fusion, MSA, OBA, FORAM, BON, Booch, OMT, PEGs, CRC-cards, Behavioural Models [Kowal92], M.E.R.o.DE, OSMOSIS, Storyboarding, Use Case Maps, Scenario Composition [Glinz95], Scenario Trees [Hsia94] and enhancements to use case modelling by abstraction levels as proposed by [Regnell96] as well as [Armour95]. Finally, Appendix B contains examples of diagrams from these methods.

A graphical overview of the different parts of the thesis and their dependencies is given in figure 1.

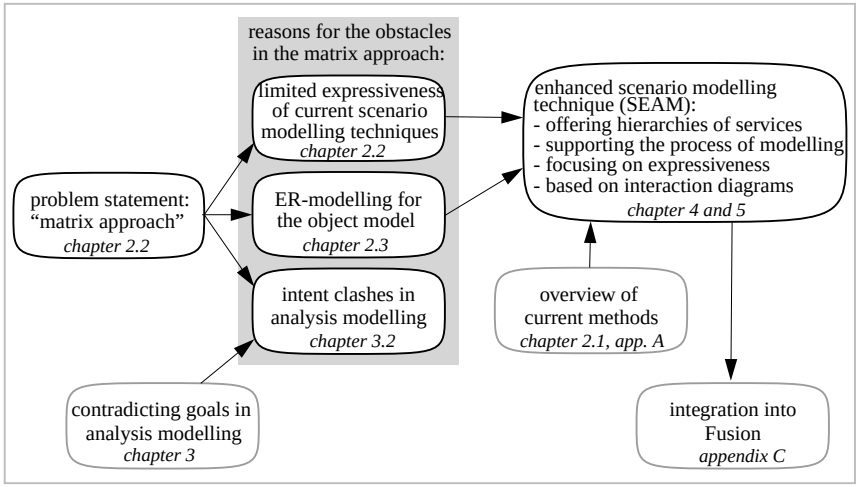


Figure 1: Overview of the thesis

Positioning of the thesis

Domain of the thesis

The thesis is about *modelling techniques* for modelling the global behaviour of software systems with scenarios. As such, the thesis belongs into the domain of *software development methods*, part of the domain of *software engineering*.

In the first place, a description of modelling techniques is concerned with the *concepts, notations and usage of models*. But, without also taking the modelling process into account, we cannot show why we have chosen certain concepts and notations or how a certain model might be used. However, we will not discuss all those other issues of project management or requirements engineering which do not have a direct influence on the modelling techniques.

The subject of validating and verifying scenario models is briefly discussed in chapter 4.6, but the thesis does not elaborate on software validation and verification and does not investigate validation and verification techniques for scenario models.

Scope of the thesis concerning the software development life-cycle

Modelling the global behaviour of a system can be done from two perspectives, both considered in this thesis. On one hand, we can model the externally visible behaviour of a system. In this case the system is regarded as a black box and the model shows the externally visible state changes of this black box as well as its interactions with any agents. An external view of the global behaviour is part of an analysis model and is developed when eliciting the requirements of a system, when determining the system boundaries, or when specifying the interfaces of the system. Models showing the external view of global behaviour may also appear in the design documentation.

On the other hand, we can model the internal view of an object-oriented system and show which internal interactions among objects are necessary to supply a specific externally visible behaviour. Models showing the internal view of global behaviour are certainly part of the design activities, but, depending on the complexity of the system and on the life-cycle model adopted, they can also be part of the analysis activities. Models showing the internal view are closely related to models showing the external view of global behaviour. When discussing the latter without considering their links to the models for the internal view, one would neglect a very important point, and therefore the thesis touches also on issues belonging clearly to the design phase. *However, our main focus is on analysis*. Implementation issues are not treated at all.

Business process modelling

Many of the models discussed in this thesis could also be used for systems of interacting objects which belong more properly in the domains of workflow management or business reengineering rather than software systems. In order to limit the scope of the thesis, we have not scrutinised such domains, although there is clearly a close relationship to business process modelling.

Level of formality

Modelling global behaviour by using scenarios can be done in a very informal way where neither the concepts to be specified nor the form of specification is precisely defined. In opposition are the highly formalised approaches, where even the language used

for the description of the global behaviour is a formal language. Methods such as OOSE and Fusion, as well as the modelling approach we propose in this thesis, are *semiformal approaches*. They define precisely what should be modelled and how, and they propose a syntax for the specification of the scenarios. But this syntax contains also informal textual descriptions, and quite often no complete model is required or even possible. The goal of such a semiformal notation is to provide a maximum of expressiveness in an easily understandable way, and to allow the developers and users to represent the global behaviour as they perceive it. Furthermore, by supporting explicitly a certain amount of fuzziness, semiformal and informal notations allow concentrating on essential aspects and prevent getting lost in details not yet important. A disadvantage of semiformal (and informal) methods is, however, that a far-reaching automatic verification of consistency, completeness and absence of deadlocks is usually not possible. For this purpose, more formal approaches would be required[1].

The motivation for using scenarios in modelling global behaviour

The use of scenarios during the software development process is neither new nor is it limited to object-oriented software development. Scenarios have been used for a long time - albeit often in a very informal way - in the various phases of the development process. Examples are numerous. If users describe how they work now and how they would like to work with a future software system, if developers discuss and determine the requirements based on the behaviour a system offers, if test scripts are used for system tests, if a developer tries to figure out the event trace caused by a specific triggering event inside a system - then scenarios are used. The use of scenarios is not limited to a specific kind of software or to specific phases of a project. Even apart from describing software systems, scenarios are used in workflow management, business reengineering and in quality assurance, e.g. for defining the necessary procedures for demanding holidays, or for specifying how orders are made.

In the literature we find the following reasons for using scenarios in the external view of the global behaviour of a software system (see e.g. [Holbrook90], [Firesmith95], [Jacobson92], [Wilkinson95] etc.):

- Scenarios allow us to divide the external view of the global behaviour of a system into meaningful pieces.
- Scenarios specified by text, tables or event trace diagrams are very easy to understand and to verify by users. They support communication with the user,

1. We adopt in this thesis the view expressed in [Bowen95]: Formal specifications are used if the emphasis is put on the verification of the software specification and if the additional effort due to the formal specification can pay off. Traditional modelling languages are used if the goal is to get easily understandable models and to communicate with the user. Both kinds of modelling techniques are necessary for successful software development. See also [Kotonya96], where formal and informal modelling techniques are considered as part of a large toolkit. He suggests that when requirements are modelled, the technique that is most appropriate is used for each viewpoint.

they can be used for role-plays and walk-throughs, and they help to produce a complete and correct requirements model.

- Scenarios can be used both to define the requirements and to test the final system. The test scripts for system tests can be derived directly from the scenarios of the requirements specification.
- In an incremental development, the increments can be defined along the scenarios.
- Scenarios can be used as the starting point for the detection and specification of the objects in an object-oriented system.

Using scenarios for the internal view of a system is not new. In the telecommunication industry scenarios have long been used to show the complex interaction patterns between various hardware or software components. When building an object-oriented system, scenarios are a natural way to show how the different objects work together to provide the required system services. A documentation of object-oriented system internals that includes scenarios allows investigators to find out quickly and easily what should happen within the system under certain circumstances in response to a given external event. Even if not all special cases are modelled in this way, the scenarios are an invaluable help for all those who are not very familiar with the architecture of the system. Scenarios can also be used in the design process to help determine the objects and their operations. Such a design approach is proposed e.g. by [Wilkinson95] in connection with responsibility driven design and CRC-cards. Kruchten even uses scenarios as the link among the four other views of the system architecture which he represents by a 4+1 view model [Kruchten95].

Finally, various methods propose using scenarios as the driving force for the development process.

Other research activities on Scenarios

During the last few years, research has been started in a number of different places in order to investigate the use of scenarios in software engineering. We are currently aware of the following efforts directly concerned with the use of scenarios in software engineering:

- Zorman from the University of Southern California developed the tool REBUS that supports storyboarding techniques for requirements analysis [Zorman95] (see also chapter A.9.6).
- Alvarez et al. from the Universidad Nacional de La Plata, Argentina, investigating hypermedia CASE-tools that allow the documentation of use cases in various versions [Alvarez95].
- Berteaud95 et al. from the University of Nantes developed the experimental CASE-tool OSMOSIS, which is essentially a browser for a semantic network and supports the definition of requirements scripts [Berteaud95] (see also chapter A.9.5).
- Glinz from the University of Zuerich works on a scenario modelling technique for the external view of system behaviour; the notation is based on statecharts [Glinz95] (see also chapter A.9.3).
- Buhr from the Carleton University of Ottawa developed use case maps as a link between the use cases of requirements analysis and the interaction diagrams of the detailed design. Use case maps are used in the system design to show which objects are involved in a use case or a design pattern [Buhr95] (see also chapter A.9.9).
- Notations for use cases as well as interaction diagrams will be part of the unified modelling language (UML). UML is developed at Rational and a preliminary version has been published in [Booch95]. The main developers of UML are Booch, Rumbaugh and Jacobson, although others such as HP (Fusion method) have joined the effort to develop a commonly accepted modelling language for object-oriented development.

In addition, many other research projects not listed above incorporate formal or informal scenario models in one way or another.

Limits of the thesis

In this thesis we investigate a number of difficulties related to the modelling of global behaviour in current object-oriented analysis methods, and we suggest how some of these difficulties could be overcome. Did we find the silver bullet for modelling global behaviour and for developing scenario models? I doubt that there are any silver bullets in software engineering. But even if there were such silver bullets, it would be an overestimation of the ability of a thesis to expect to find the particular solution that everybody else in the large community of software engineers has been missing so far. Thus I do no claim to offer a silver bullet. However, I expect and hope that in this thesis there are some concepts and ideas that will be incorporated in other approaches and that can stimulate other people to see certain things from a new point of view. A summary of what I consider as my main contributions to the field of object-oriented software development can be found in chapter 6.

Chapter 2
The modelling of global behaviour in current object-oriented analysis methods

2.1 Modelling global behaviour: the usage of use cases, scenarios and interactions diagrams

2.1.1 Introduction

Investigating current notations for modelling global behaviour can be very confusing. At first sight, some of the notations differ in their graphical symbols and the additional information they represent. But if there were only these differences, then all these diagrams should be easily transformable into each other. This is not the case because there are also divergences in the basic concepts that are used. The result is similar notations for totally different concepts, and different notations for the same concepts. Worse, the same terms are used for different concepts in different methods. This is possible because there does not yet exist any reconciled set of definitions or any classifications of the different terms, concepts, approaches and notations that are generally accepted.

In this chapter we discuss the underlying concepts of various notations. We also describe what aspects of global behaviour can be reflected by these notations, and how these aspects can be combined into one diagram. The starting points of the investigation were those approaches that use interaction diagrams or event trace diagrams to model sequences of interactions between a system and its agents, or among its internal objects, and also those notations providing textual descriptions of such sequences (e.g. use cases or use scenarios). Because the same content can also be expressed with regular expressions or state charts, we also included such techniques. The emphasis, however, was put on interaction diagrams and use case descriptions.

We do not provide a comparison or evaluation[1] of different methods, nor do we present a general metamodel[2]. We focus on the various concepts found in techniques that model global behaviour, we give an overview of these concepts and their relationships, and we provide a short description of various methods in the appendix.

2.1.2 The global behaviour of an object-oriented system

Berard [Berard93] defines an object-oriented system as a *system of interacting subsystems or objects*. The *structure* of such a system is defined by how each component is composed of and connected to other components. The *behaviour* is defined by the interactions within the system, and between the system and its environment. The behaviour specifies how an object or system acts and reacts in terms of its state changes and interactions, and defines its outwardly visible activity [Booch94, page86].

2.1.2.1 Global versus local behaviour

In any system of interacting objects we can look at the behaviour of the individual objects or at the behaviour of the whole system. When describing *local behaviour*, we focus on one single object and how it interacts with its environment under certain conditions (e.g. by specifying the class interface with the contracts for all the services the object offers and for all the services the object needs from other objects). When describing *global behaviour*, the whole system of interacting objects comes into focus, and we concentrate on the behaviour that is provided by these objects together.

1. A quantitative comparison of object-oriented methods, mainly focusing on notational issues, can be found e.g. in [Stein93, Stein94]. There 4 macrocomponents (or characteristics) are distinguished, each being divided into about 40 microcomponents. For each method these microcomponents are rated with a value between 0 and 2 (no support, average support, good support). The interpretations of this data are very delicate, because they cannot take into account the underlying concepts, orientation and philosophy of a method. The conclusions are purely quantitative, and in my opinion do not really do justice to those methods that do not do well in respect to the implicitly chosen evaluation criteria.
2. There does not yet exist a commonly accepted metamodel that could be used for a comparison. OMG has tried to get all different object-oriented methods described on the basis of a common metamodel [OMG92]. Yet as far as I know, this effort was not continued.

2.1.2.2 Internal versus external view of the global behaviour

Modelling global behaviour of a system can denote either an internal or an external view of the system (see figure 2). In the external view, the model shows the services that the system as a whole offers to its environment, without considering how this behaviour comes about internally. The system itself appears as one single object; we only describe the interactions between the system and its environment. The environment may be further detailed into different agents[3]. In other words, the external view shows the interface of a system or subsystem. Yet the interface definition does not only mention all the services offered but also all the services needed from other systems in the environment. Examples of external views of global behaviour are the use cases of OOSE or the interface model of Fusion.

A special case of modelling the external view is by using user views, as in the scenario trees of [Hsia94]. There, the behaviour of the system is modelled from the perspective of one single agent, i.e. only the interactions between the system and this agent are taken into account, while interactions with other agents are shown solely in the user views of these other agents (see figures 140 and 141).

An internal view not only contains the interactions with the environment but also the interactions among the different internal objects of the system. The internal model thus shows how these objects work together to provide a certain global behaviour. Examples are the object scenarios in BON and the object interaction diagrams in OMT, Fusion and OOSE.

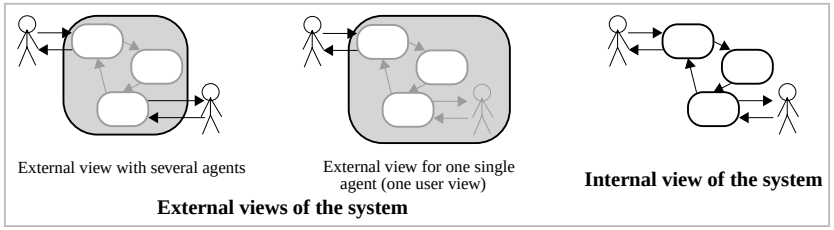


Figure 2: External versus internal views

2.1.2.3 Conceptual versus technical interactions

We define an *event* as something that happens at a certain point of time (see definition in [Rumbaugh91]) and that is of a certain importance to a given system or object. Every event has a name, it may also have parameters that give additional information.

3. An agent may be anything (a human being, a software system, a hardware component) that is outside of the system and can send or receive events to or from the system. The various agents are normally distinguished by the roles they have in respect to the system.

We define an *interaction* as an event that is a message from one object to another object (or to itself). In a diagram, an event that is an interaction can be modelled as an arrow between two objects. The term event is more general than the term interaction, though in an object-oriented system most or all events are modelled as interactions between objects. If the term event or the term interaction is used and if all interactions are considered to be events is method specific.

When modelling interactions between objects (among internal objects or between the system and the agents), we can do this on different abstraction levels. Therefore we introduce here the distinction between *technical* and *conceptual* interactions and events.

- *Technical:* A technical interaction or event corresponds to a certain software construct in the final software system. This may be a procedure call, an entry in a dialogue window, a menu item, or a hook in an UIMS. When modelling technical events, we focus on the software design. Examples are the method calls in the interaction diagrams of Fusion, OOSE and OMT, and the system input events (system operations) of Fusion.
- *Conceptual:* When modelling conceptual interactions or events, we focus on the concepts of the problem domain, the implementation is of no concern. A conceptual event need not correspond to any software construct, it may be more abstract. Examples are transactions in the use cases of OOSE, events in the event lists of Modern Structured Analysis, M.E.R.O.DE and OORD, or again the input events of Fusion.
- *Purely conceptual*: Purely conceptual events are conceptual events that do not correspond to technical events. Yet in analysis or high-level design models, whether an event is a purely conceptual event or a technical event can often only be determined in retrospect, i.e. when the design is completed.

Of course within the technical events as well as within the conceptual events we may again distinguish several abstraction levels. For example, a technical interaction provided by a UIMS can again be mapped into the single keyboard entries necessary to generate that interaction. When building an information system, the lowest necessary abstraction level to be considered will be the events provided by the UIMS. Yet when building an UIMS, we will also model more low-level interactions. The same is true for conceptual events. Moreover, whenever the actual interactions found in the software system correspond to concepts of the problem domain (e.g. system operations in Fusion), a given abstraction level is conceptual as well as technical. When modelling global behaviour, any possible abstraction level can be chosen. Figure 3 shows the same scenario twice, once on a purely conceptual level and once on a technical level. Figure 4 shows the beginning of the scenario making_phone_call even on three abstraction levels. The first abstraction level is good for giving an overview to the functionality of the system. The second may be suited for determining requirements with the user. For specifying the detailed requirements that serve as a basis for developping the internal design models, the most detailed abstraction level is needed.

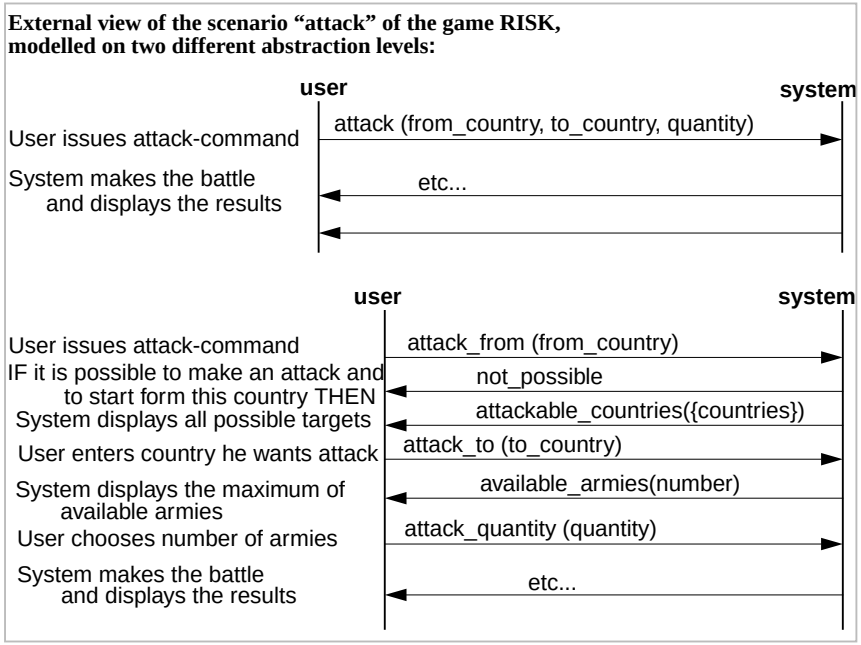


Figure 3: Conceptual and technical interactions for different abstraction levels

An oft-heard criticism of certain object-oriented analysis methods is the weak connection between the static and dynamic models. The static model shows the object types and their operations, and the dynamic model consists of a set of state transition diagrams. Frequently the state transition diagrams are found to show conceptual events that are on a higher abstraction level than the events calling the object operations shown in the static model. Unfortunately the explicit mapping between these two levels of events is often missing.

2.1.2.4 Requests versus notifications

Besides the decision for a specific abstraction level there are two further points to decide upon concerning the interactions: Do we consider the objects as being active, each having its own thread of control, or as passive? Do we consider the interactions as being one-way messages (notifications), or requests?

If we decide for passive objects and thus for only one thread of control in the whole system, the flow of control coincides with the flow of interaction. We have *intraprocess* interactions. Sequential programs only have intraprocess communication, and also languages such as Smalltalk and C++ or methods such as Booch [Booch91], first generation OMT [Rumbaugh91] and Fusion [Coleman94] only support intraprocess interactions. In contrast to the intraprocess communication, *interprocess* communication takes place between two active objects, each of them having its own thread of control. Interprocess communication is the basic concept for all concurrent systems, for the models of workflow management systems and for languages such as CSP and Occam. Interprocess communication is also that type of communication we encounter in daily life between people, between organisations and between machines.

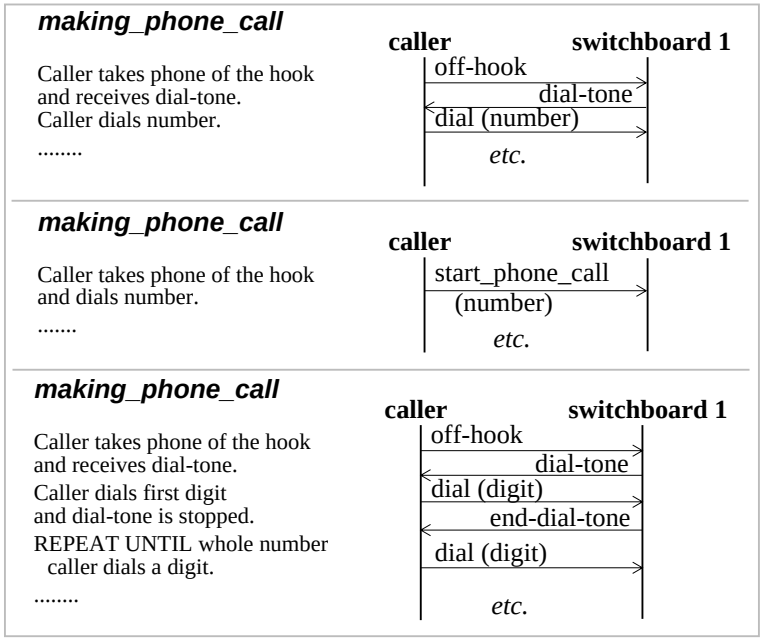


Figure 4: Modelling on several abstraction levels

An interaction between two objects can be a simple *notification* (or *one-way message*), or it can be a *request*. In the notification case, one object notifies another by sending it a message with some information. No return interaction takes place after the notification. In the request case, a client object requests a service from a server object and waits for its reply. Requests imply a client server- (or uses-) relationship between two objects. Object-oriented languages for sequential systems only support requests.

When modelling object-oriented systems, we thus get four possible forms of interaction, though only two of them play a major role when discussing the modelling of global behaviour:

	interprocess	**intraprocess**
notification	**signal, notification**	**“goto”**
request	**remote procedure call**	**procedure call**

Sometimes the same notations are used for models showing notifications as well as for models showing requests. An example of this is the event trace diagram of OMT in figure 135: the left diagram shows notifications, the right one shows requests using two arrows for each call. In contrast to this, the concurrent object interaction diagram of OMT (figure 136) uses slightly different symbols for interprocess notifications and for intraprocess requests (procedure calls). Also, in the interaction graphs of Fusion, arrows can denote notifications (in the system interaction graphs) or procedure calls (in the object interaction graphs). Though the interpretation of the diagram is quite different, in both cases the same symbols are used, which may be quite confusing.

Often an analysis model showing the external view of a system models the system on a more abstract level, thus it uses notifications for the interactions between the active system and the active agents which represent purely conceptual interactions. For example the input and output events of the Fusion interface model are notifications. They may have parameters but have of course no return values. Figure 112 shows an event trace, figure 115 a system interaction graph with notifications. When moving to the design, the perspective then often changes, because in fact it is the system that prompts the user for some command or information. The interaction with the user is a procedure call from some controller or problem domain object to an interface object which acts as mediator. Figure 5 shows the transition from notifications to requests when changing from the external view to the internal view.

In a sequential system, the interactions among internal objects are always modelled as requests. For an example see the object interaction graphs of Fusion (figure 117). Yet when describing the interactions among internal objects as well as the interactions with the environment of the system in one interaction diagram, we often would like to show the effective procedure calls for the internal interactions, and at the same time to abstract away the interface mechanism, thus modelling the user inputs as notifications[4]. Examples for this are found in the interaction diagrams of OOSE (figure 111) where even different kinds of arrows are used for the notifications and the requests (procedure calls), and in the object scenarios of BON (figure 127).

4. For the user interactions in sequential programs, the transition from notifications to procedure calls often is deferred to the implementation phase. There an active object is introduced that handles the interactions between the problem domain objects and the user interface components (e.g. by having an event loop). This object may also control the system life-cycle (e.g. as suggested by Fusion).

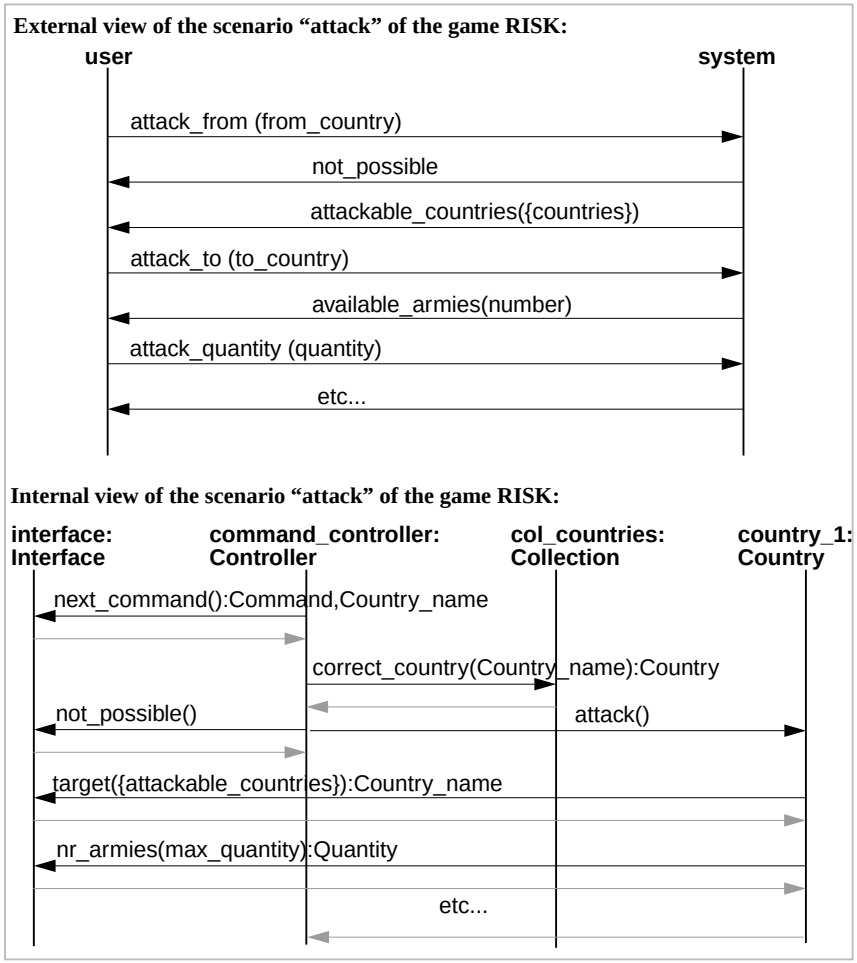


Figure 5: Using notifications for the external and requests for the internal view

2.1.2.5 Design versus analysis

Modelling the global behaviour of a system may be part of the analysis as well as of the design. What view of the global behaviour is recommended in which phase differs from method to method. Some methods model technical interactions only during design, others already during analysis. Some determine the internal view only in the design; for others it is part of the analysis. In some methods certain notations such as state charts or interaction diagrams are used during analysis, in other methods they belong to the de-

sign. Furthermore, there are also many methods that do not have a clear distinction between analysis and design at all.

In this overview we primarily want to discuss modelling techniques for the analysis. Yet due to the often quite arbitrary distinction between analysis and design, we also consider certain design notations.

2.1.3 Modelling global behaviour by modelling scenarios

We distinguish between the generic term scenario as we use it in this chapter, and the method-specific term scenario as it is defined by various methods. We define the generic term scenario in the following way: "A *scenario* is a sequence or a set of possible sequences of events or interactions." The events of a scenario provide together some service or system function. The sequence is triggered by an initial event, which is also called the *stimulus* of the scenario.

Many methods provide one or several modelling concepts that fulfil above definition for scenarios. Though we consider them all as scenarios, they have different names, may have different syntax and semantics, and are used for different purposes. Examples are: use case (OOSE), scenario and system operation (Fusion), script and use scenario (OBA), scenario, user case and top-level system operation (2nd-generation OMT), use case, scenario and function point (Booch), scenario (Kowal), essential activity (MSA), business transaction (MSA-GfAI). For more details see appendix A.

2.1.3.1 Scenarios of one object versus scenarios between several objects

In above definition of scenarios we have not distinguished between the scenario models that show the interactions among several objects and those that show the interactions of one object with its environment. Yet there are some essential differences between these two kind of models.

In the first case, *the scenario between several objects*, we have several objects, subsystems or systems which can all interact with each other. These interactions can be either notifications in the case of a concurrent system or of several interacting systems, or requests in the case of a sequential system.

In the second case, *the scenario of one object*, we show the interactions between this object, subsystem or system with its environment, the agents. Direct interactions between the agents are not considered. These kinds of models are mainly used for two purposes: to model the local behaviour of one single object which is part of a larger system (e.g. state charts showing object life-cycles), or to model the global behaviour of a system or subsystem from an external point of view (e.g. interface model of Fusion). As we are focusing on only one object, we can divide the interactions into input (from an agent or a client to the object) and output events (from the object to an agent or a client). One input event causes only one output event if it is a request (see the operation schemas of internal methods in Fusion), or may cause several output events if it is a notification (see the schemas of system operations in Fusion).

A scenario that contains for one specific system function all the interactions between the system and its environment shows an external view of this system. It can be refined into a scenario that shows also the interactions between the components of the system (internal view of the system, external views of the components). For each component, the external view showing input and output events can again be refined into an internal view showing also the interactions among the components of this component. Many methods do not have a recursive refinement of this kind, but stick to the two levels of system and objects. They only differentiate between an external model of the system as a whole and the internal model on the level of objects and their interactions. There are two reasons for this:

- Somewhere during the refinement process we switch from purely conceptual events to technical events and from notifications to requests. Many methods do this when moving from the external view of the system, where we model on the level of the system as a whole, to the internal view of the system, where we model on the level of objects. Often also the modelling technique is changed (see e.g. OOSE and Fusion). Thus they only know two levels of refinement.
- Many of the investigated methods do not really support the elaboration and modelling of concurrent systems or of systems containing subsystems[5] that are systems themselves.

2.1.3.2 Scenario instances versus scenario types

Event instances and event types

An event that has no parameters or only instantiated parameters (i.e. each parameter has taken a value that is a member of the set of values defined by its type) is an *event instance*. An event that has at least one parameter that is not instantiated (i.e. which is a parameter type specifying a set of possible parameter values) is an *event type*.

Scenario instances and scenario types

A *scenario instance* is a sequence of event instances. A diagram modelling a *scenario instance* contains neither any event types nor any optional or alternative events. A *sce-*

5. BON uses object groups in its object scenarios; these serve only for simplifying the diagrams and have no further meaning within the system.

nario type specifies a set of possible scenario instances. A diagram modelling a *scenario type* contains either event types or optional or alternative events. For a scenario type we do not require that all events are event types. Thus a diagram modelling a sequence of event instances and event types also shows a scenario type.

The first interaction diagram in figure 7 shows the scenario instance Peter_calls_-James_busy. This scenario instance shows only one sequence of interaction instances - all the interaction parameters are instantiated. The other interaction diagrams in figure 7 show scenario types. In these scenario types the parameter *number* is not instantiated and in the last interaction diagram certain interactions are even optional.

Modelling scenario instances

Diagrams of scenario instances show snapshots of system behaviour. They are suited for communication with people not used to abstract thinking, for finding the boundaries of the system, or for comprehending a complex internal mechanism. An example of such a diagram is found in figure 135. There, nothing is parameterized. Even all the parameters that do not influence the course of actions are replaced by concrete values or object instances. Of course, diagrams showing only scenario instances are not suited to the acquisition of a complete behavioural model. Therefore, much more often scenario types are modelled. Data or object types are used as parameters in the event types, and several possible courses through the scenario are described.

Algorithmic versus declarative specification of scenario types

When making a complete model of a scenario type, we have to specify all possible orders of events. We can do this in an *algorithmic* way, annotating the diagram with *sequence information* that shows iteration, alternation and concurrency (see e.g. scripts for interaction diagrams of OOSE and Booch in figures 111 and 130, dotted lines for iterations in the event trace diagrams of Fusion in figure 112, sequence labels for interaction graphs of Fusion in figure 117 and operators in the life-cycle expressions of Fusion in figure 113 or in the path expressions of PEG in figure 152). Not all notations allow the modelling of any kind of algorithm, e.g. the event traces of Fusion only know iterations. Others have symbols for alternative, optional and iterated events. Some even offer a syntax for concurrent or interleaving sequences of events.

In contrast to the algorithmic description, we can specify the reactions to the initial event in a *declarative* way. In this case, only the state changes and all the interactions that took place are specified, but not their order. An example are the system operations in Fusion (see figure 116).

2.1.3.3 Scope of a scenario type, classification schemes

Scope of a scenario type

The *scope of a scenario type* has two dimensions: the length of the scenario instances and the number of possible scenario instances specified by this type. When specifying scenario types, there are several possibilities we can choose from for both dimensions. For example all three scenario types in figure 7 start with the event *off-hook* and end with the event(s) *on-hook*. Instead of modelling the whole length of a phone call in one scenario type, we can also split it up as it is done in figure 6. There, we have shorter scenario types; for a complete phone call a sequence of up to four scenario instances is executed, instead of just one scenario instance as is the case according to the specification of figure 7. Thus, between figure 7 and 6 we have a difference in the first dimension of the scopes of the scenario types.

But even if we have decided on the length of the scenario instances we want to specify by scenario types, we still have several possibilities concerning the second dimension of the scope. For example in figure 7, the same problem - making a phone call - is once modelled as three scenario types (in the middle: scenario types *making_phone_call_-busy*, *making_phone_call_no_answer* and *making_phone_call_connect*) and once as one scenario type (bottom: *making_phone_call*). The two possibilities differ concerning the second dimension of the scope, i.e. in the number of possible scenario instances. The scenario type *making_phone_call* contains more possible scenario instances than the scenario types *making_phone_call_busy* or *making_phone_call_connect*, because it covers all possible variants that can occur when making a phone call, whereas e.g. *making_phone_call_busy* only covers phone calls where the callee is busy.

For both dimensions of the scope of scenario types there exists a set of possible criteria. Normally, when developing a scenario model, we choose a tuple of criteria to determine the scope of the scenario types, i.e. we choose one criterion for determining the end of the scenario instances and one criterion to classify these instances into scenario types. Such a tuple of criteria we call a *classification scheme.* When using a development method that does not offer any hierarchies of scenarios, such a method either enforces itself a specific classification scheme (e.g. Fusion), or it requires that the user chooses one. This classification scheme either determines or just influences (in case it needs not be followed strictly) how the scenario model is structured into scenario types.

In the following we describe criteria for both dimensions of the scope of scenario types. These criteria have been formulated based on the classification schemes found in the different methods we investigated.

Possible criteria for the end of a scenario

We have to determine when one scenario ends and the next one starts. There are several possible criteria we can use to determine the length of the scenario instances, and thus the length of the scenario types which contain these instances:

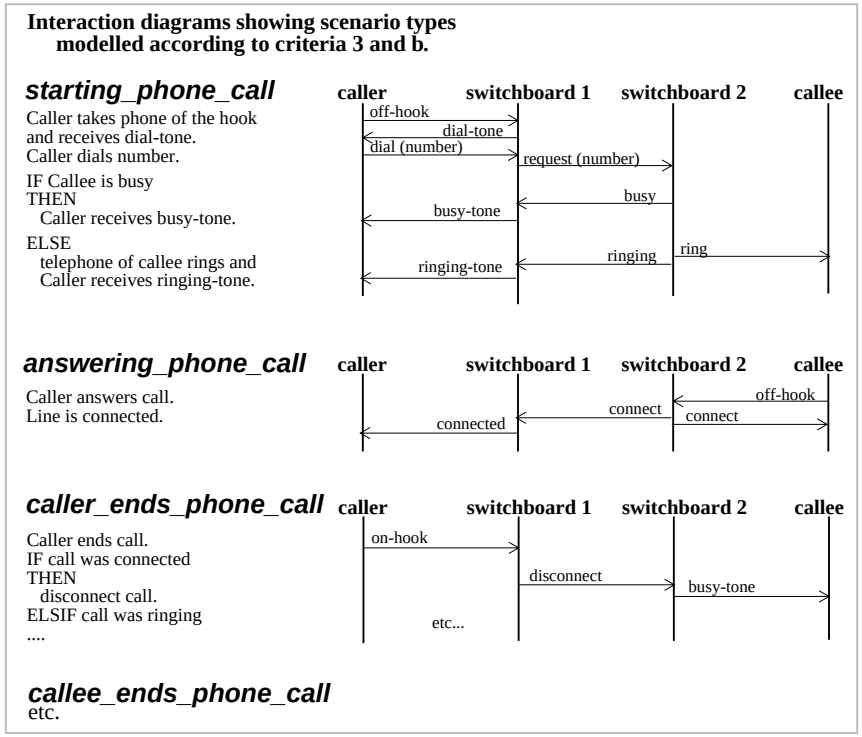


Figure 6: Modelling scenario instances and types, part 1

a) One scenario instance consists of one input event and zero, one or several output events. Intermediate interactions between the agents and the system are not allowed (see system operations of Fusion or essential activities in [McMenamin84]).

b) A scenario instance ends as soon as a new essential system state is reached. Such an essential system state may appear in the system life-cycle model (see scripts in OBA).

c) Intermediate interactions are allowed as long as further input events cannot themselves trigger scenarios of their own (see business transactions in MSA-GfAI).

d) The whole sequence of input and output events that is perceived as belonging together, as being part of one and the same overall task, is one scenario.

e) The whole life-cycle of an object or a system, i.e. the whole sequence of events from initialisation until shut-down, is considered as one large scenario.

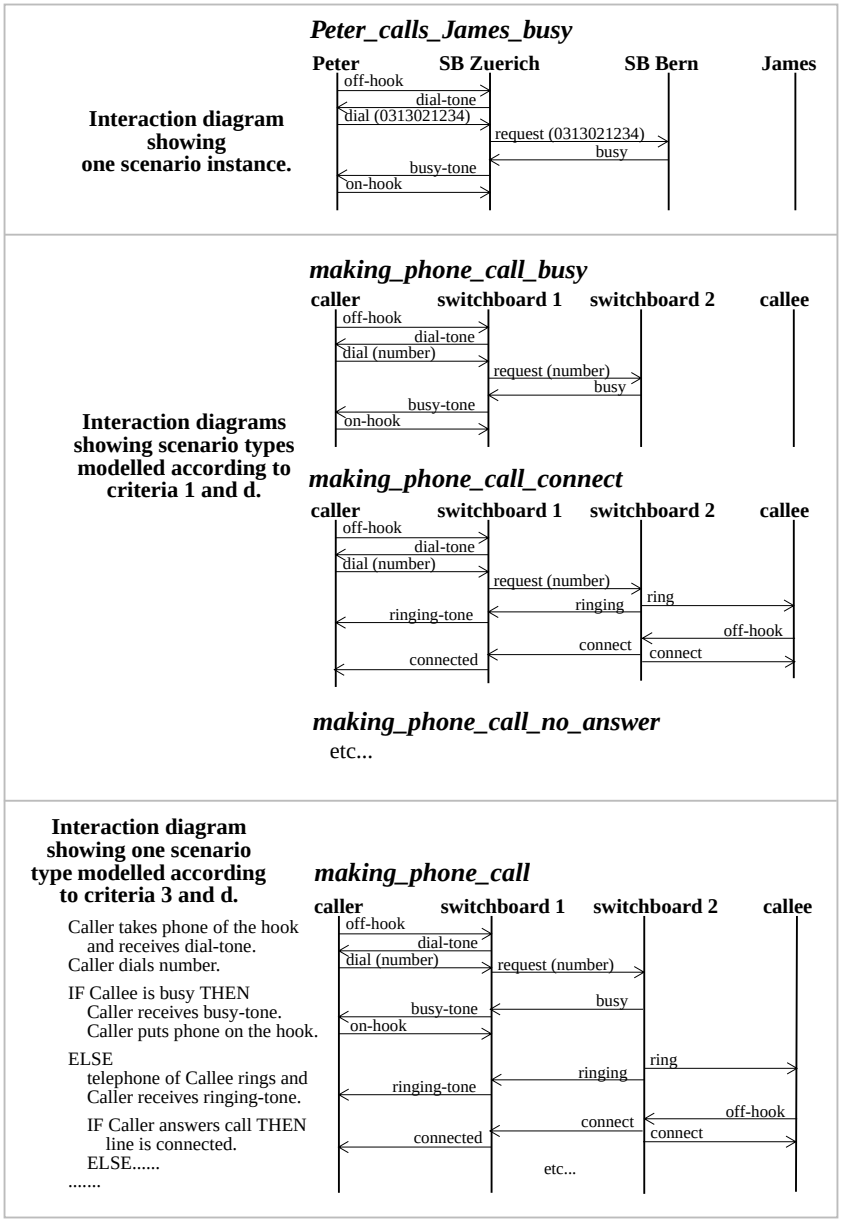


Figure 7: Modelling scenario instances and types, part 2

The scenario type *making_phone_call* of figure 7 is modelled using criterion d. In contrast, the scenario types of figure 6 are modelled based on criterion b.

The only really precise criterion for the end of a scenario is the first one. It also satisfies the notion of having perfect technology within the system and thus no duration for the execution of scenarios. Yet its disadvantage is that it necessitates that the input event has in its parameters all necessary information that the system needs for this scenario, because several interactions between the system and the agent for obtaining further information would result in several scenarios. Also, the system boundary with further agents influences the classification, as is shown in figure 8.

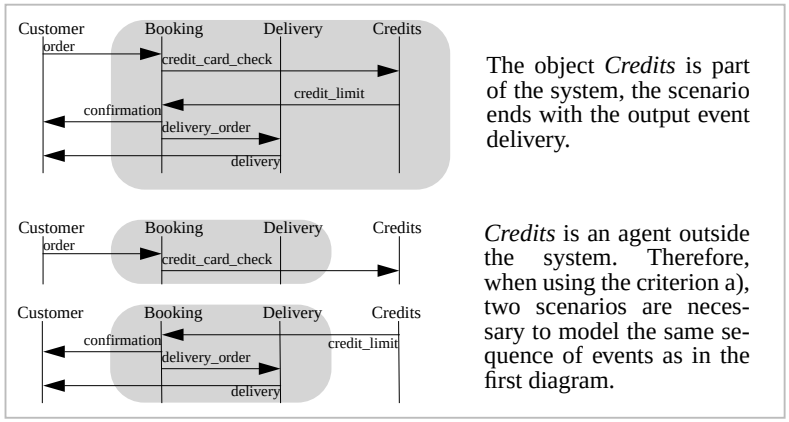


Figure 8: Determining the end of a scenario

Possible criteria for classifying given scenario instances into scenario types

Once we have decided on the length of the scenario instances and types, we have also to decide on how many variants are contained in one scenario type, i.e. how we divide up the set of all possible instances into the sets of instances specified by scenario types. In the following we present various possible criteria that might be used.

One scenario type may include the following number of scenario instances:

1) All scenario instances with exactly the same sequence of event types; only the values of the system state and of the event parameters may differ.

2) Some scenario instances with the same initial event type (stimulus) and a subset of all the possible sequences of further event types,
  2a) the scenario type is determined by the content of the input events.
  2b) the scenario type is determined by the system state at the time of the initial input event.

3) All scenario instances with the same initial event type (stimulus); the rest of the scenario may differ arbitrarily.

4) All scenario instances that have certain common parts or handle similar tasks.

In most of these criteria, the specification of the initial event types (i.e. the classification of event instances into event types) determines how the scenario instances are grouped into scenario types. Thus these criteria presuppose well defined event types. For example, if we have the event type *order (payment_info, client, items)* as an initial event type, criteria 3 determines that the model will have a scenario type *order* that handles all the different variants of ordering. If we assume the initial event types *order_cash (agency, client, items)*, *order_advance_payment (client, items)*, *order_COD (client, items)* and *order_credit_card (card_info, client, items)*, then we will get four different scenario types, each one handling one way of payment. Some of the difficulties that arise due to dependencies between defining event types and scenario types are discussed in chapter 2.2.2.1.

The first criterion is only used in partial scenario models. Due to the vast number of possible different reactions to an initial event it is not feasible to model the whole system in this way (see the use of scenario diagrams in Fusion and [Kilberth93], figures 112 and 125). The fourth possibility is the other extreme: a very rough classification covers all possible scenarios, yet the types cannot be modelled in depth to consider alternative courses, intermediate interactions, event parameters and other details (see BON, where only for some scenario types a high-level diagram is made; a more detailed modelling of scenarios is considered as being too complex [Walden95, page 170]). The third possibility is chosen by methods such as Fusion and MSA, where a scenario contains restricted or no intermediate interactions. So one scenario type contains all the scenario instances which start with the same input event type, but different output event types are possible. When a scenario may contain longer sequences of interactions, the third possibility may lead to very few, yet very complex scenario types (OOSE mentions as an example a telephone system, where most scenarios or use cases start with the event “handle lifted”). Therefore, methods such as OOSE prefer the second possibility for the classification of scenarios. They leave it to the judgement of the developers to decide which scenario instances are considered as variants of one and the same scenario type, and which ones are handled as separate scenario types.

2.1.4 Notational limits in modelling interacting objects

When modelling interactions, we would like to show many different aspects all at once. We name only a few of them:

- the names of the system, agents or objects involved in the interactions
- the dimension of time, i.e. the order of events

- all possible sequences in which the events can occur together with the corresponding conditions
- the events with their parameters and possible return values
- the different states of the objects or of the system and the state changes

Yet any diagram is limited to two dimensions. Of course, we can decompose diagrams. We can also use annotations, maybe even supported by a hypertext tool. But the restriction remains. We always have to choose a few main aspects we want to focus on; the others need to be neglected or dealt with by annotations. In the war for the 'best' method or the 'best' tool, it is often forgotten that depending on the type of system we are modelling, on the momentary intent behind the model, and on personal objectives, certain techniques are better suited than other ones. Moreover, some views can even be automatically transformed into other ones.

In the following paragraphs we are going to discuss possible aspects one might like to describe in models showing object interactions, and we will show which modelling techniques focus on which aspects.

2.1.4.1 Modelling the dynamics versus modelling the statics

According to standard IEEE 610.3, a *dynamic model* shows the change that occurs in a system in which there is change, such as the occurrence of events over time or the movement of objects through space, whereas the *static model* shows those aspects that do not change over time. Also when modelling the global behaviour of a system, we have to distinguish between those models that show the static aspects of interactions and those models that show the dynamics of the behaviour.

As a *static interaction diagram* we define an interaction diagram that shows all possible interactions between its objects; it contains no information concerning the sequence of these interactions or their preconditions. Examples of static diagrams showing interactions are: collaboration graphs (see figure 142 and 145), event flow diagrams of OMT listing all possible events between certain objects (see figure 134), and the communication associations in the object model of OOSE (see figure 6, the arrows denote that over these associations events can be sent, the events themselves are not listed).

A *dynamic interaction diagram* shows one or several possible sequences of interactions between several objects (see figures 117 and 110). State transition diagrams also belong to the dynamic models; they show all possible sequences of interactions with one specific object.

In the static model object types are often used whereas the dynamic model shows object instances. Yet these object instances are often rather used as prototypes of their object types. The concrete values of these object instances are not important. Considering the prototypical nature[6] of the objects in the object diagrams, the distinction between object instances and object types gets blurred. The formulation or the symbol for "object x" is

interpreted as "an object of type x" when interactions are concerned, and as "object type x" when the inheritance hierarchy is specified.

2.1.4.2 Dynamic models: transitions between states versus interactions between objects

An event in an object-oriented system may have the following two aspects:

- it may be an *interaction between two objects*, modelled by interaction diagrams or equivalents,
- at the same time it may be a *transition between two states* of the object that receives the event, modelled by state transition diagrams or equivalents.

We can compare this to looking at a coin from different angles of vision (see figure 9). At each angle there are certain parts we see better and others we see less well. If we look only at one side (e.g. at a two-dimensional object interaction diagram), this side is very clear to us, yet the other side (the different states of these objects) is totally hidden. If we hold the coin very near to our eyes with the edge towards us, we can get a glimpse of both sides, yet neither is very clear, and only the edge (events) is really visible (this is somehow the case in the life-cycle expressions of Fusion).

The two counterparts

State transition diagrams show the whole life-cycle of one single object, i.e. all the events the object can receive in all its states. If the diagram is also annotated with the events the object sends and the recipients of these events, all the state transition diagrams together with a list of external events define the whole dynamic behaviour of a system. Interaction diagrams that show the interactions caused by a specific external event can be derived from a complete set of annotated state transition diagrams.

When we make for each external event an interaction diagram that shows all necessary interactions between objects, and annotate this diagram with the preconditions that must be fulfilled by the objects involved, we also get a complete model of the dynamic behaviour of the system from which we can derive the state transition diagrams of the individual objects. Figure 10 shows the two possible views: life-cycles and scenarios.

For system development both views are necessary. If we design the classes, we need state transition diagrams. If we focus on the system functions and the overall design of the interactions, we prefer interaction diagrams. In some methods, both views are explicitly modelled; yet often one view does not contain all the information necessary to derive from it the other view. Other methods combine the different techniques, e.g. Fusion uses life-cycle expressions (for the life-cycle of the system as a whole) and preconditions (re-

6. See also object-oriented languages such as SELF [Ungar91], which do not differentiate between object types and object instances; see also diagrams as in figure 131, which show interactions between instances as well as further associations between the corresponding types.

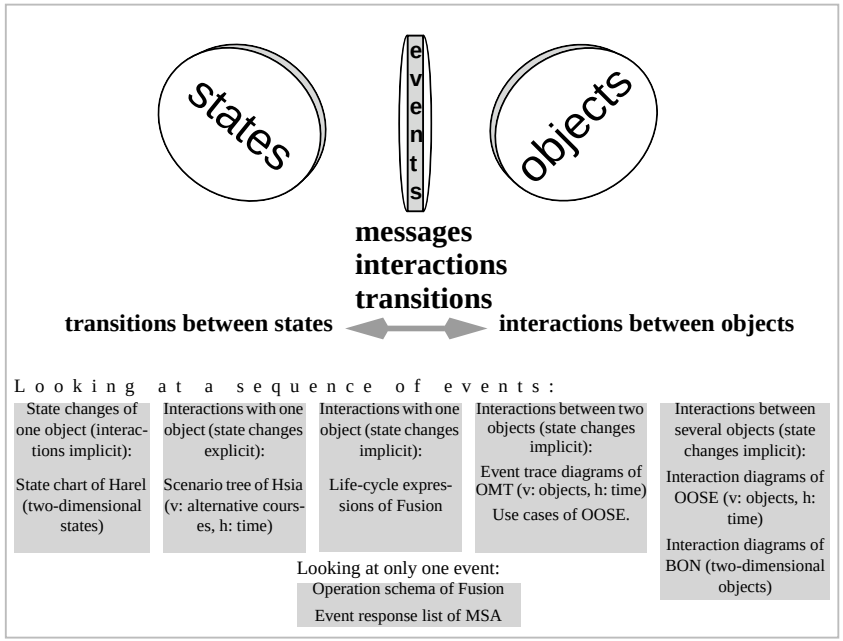


Figure 9: Transitions between states versus interactions between objects

ferring to the life-cycles of individual objects) to specify the possible sequences of system operations.

Another reason why changing from one view to the other one is often not possible is that sometimes only the most important events are included in the state transition diagram of an object (e.g. leaving out all events that can occur in any state) or in an interaction diagram (e.g. leaving out all events that an object sends to itself). Also, often the object states in the preconditions of an interaction diagram are not described in the same manner as in the state transition diagrams.

Figure 9 gives an overview of the different possibilities to combine objects and states, and it mentions also examples of modelling techniques.

The system life-cycle

For the special case where the modelling effort focuses on the interactions between a system and its environment, both an interaction diagram that is enlarged to include all interactions (in the form of an event trace diagram with script or in the form of regular expressions) and a state transition diagram of the system as a whole can contain the identical information and can show the same view, namely the whole system life-cycle. For

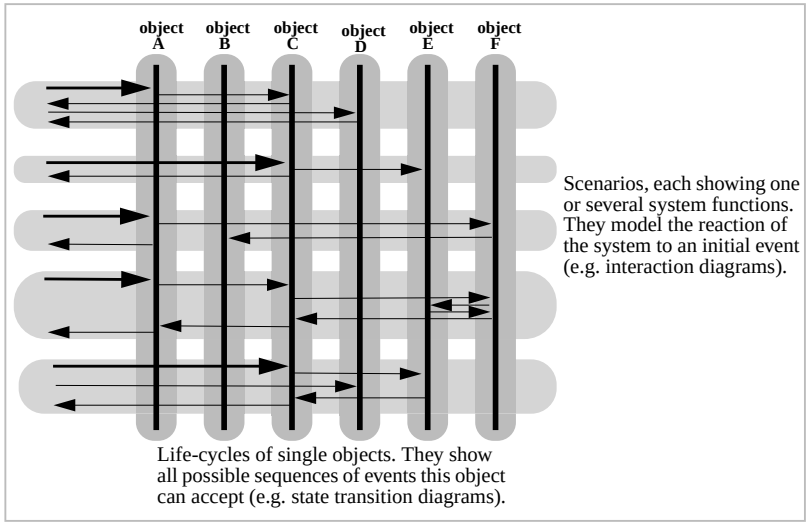


Figure 10: Dynamic model of a system: life-cycles of objects or scenarios of system functions

this reason, several methods define automatable transformations between corresponding notations, e.g.:

- life-cycle expressions --> state transition diagram (Fusion)
- scenario trees --> regular grammar --> state machine ([Hsia94])

Hsia uses transformations to verify the consistency of the dynamic model of analysis, whereas in Fusion the transformation is one of several steps for implementing the system life-cycle, which is specified by regular expressions.

Assertions and rules

Instead of using state transition diagrams or interaction diagrams, the dynamic behaviour of a system of interacting objects can also be specified by using rules and assertions. This is done in FORAM, where for each object type not only the operations are defined, but also any business rules that affect this object. In the domain of information systems, business rules are often already formulated and can easily be allotted to the rulesets of the object specifications; also they are better understandable to the users verifying the models than a set of state transition diagrams. During design, assertions on the object operations are then derived from the rulesets and any interaction diagrams. Of course, other methods than FORAM also use assertions in the object specification during design, often replacing earlier dynamic diagrams by assertions.

We may mention here also the approach of Odell: the relationships in the class diagram may be annotated by different types of business rules, some of them also expressing dynamic aspects [Odell93].

2.1.4.3 Interactions between objects: time-line diagrams versus object diagrams

Time-line notations

Time-line diagrams

Time-line diagrams are two-dimensional graphs where the vertical axis is time and on the horizontal axis the different objects are listed (see e.g. figures 111, 112, 130 and 135). Sequence information is found in two places: a basic sequential ordering of the interactions is given by the order of the arrows in respect to the time axis. Iterations and alternations are shown by additional symbols and/or pseudo-code that is written to the left of the diagram, also called script (not to be confused with the scripts of OBA). Historically this notation comes from the telecommunication industry (message sequence charts as specified by the ITU-T Z.120 standard).

Time-line diagrams have the advantage that they emphasize the time related aspects and concentrate only on dynamic aspects. The possible sequences of events can be seen at a glance. Furthermore, reading a text from top to bottom, even with iterations and alternatives, is very familiar not only to engineers but also to users coming from other domains. The disadvantage of time-line diagrams is that for very complex scenarios, involving either many objects or many different courses, the diagram rapidly becomes confusing. Also, most methods do not provide any decomposition for time-line diagrams.[7]

Tables

OBA uses scripts (not to be confused with the annotations to time-line diagrams often also called scripts) to model scenarios. These are tables, the rows showing the interaction and objects involved, the vertical axis showing the sequence of time (see figure 124). OBA-scripts normally only show one possible sequence, so no further information for additional courses is necessary.

Text only

When the diagram only shows the interactions between a system and its environment, then the graphical part can be replaced by a textual description. This text may be in structured English, or even in plain text as e.g. in the use cases of OOSE (figure 108). Yet still the vertical axis shows the flow of time.

7. In OOSE, which uses time-line interaction diagrams in the design model, interaction diagrams are decomposed whenever parts of use cases have been factored out into abstract or extend use cases. Yet no decomposition of interaction diagrams of the design model is supported that is not linked to the use case descriptions of the analysis model.

Two-dimensional object models

Object diagrams

Object diagrams represent the objects as two-dimensional symbols and not as mere lines. Therefore they use both dimensions to show the objects and interactions involved in a scenario. The dynamic aspects are only described by numbering the arrows and by additional text. Information about the different possible sequences of interactions is found in the textual description. Fusion duplicates some of the sequence information by having a more complicated numbering scheme, yet these numbers can only represent simple dynamic behaviour (see figure 117). Fusion also enforces one textual description for each method, which results in further redundancy between the graphical and the textual part of the diagram, whereas most other notations suggest having only one textual description for the whole interaction diagram (e.g. BON, figure 127).

The advantage of object diagrams is that they look very similar to class diagrams, so people do not have to get accustomed to two very different notations. Also, the grouping of objects (e.g. figure 128) or the decomposition into several diagrams (e.g. figure 118) is quite easy. Yet the dynamic information is not very obvious. It is mostly in the textual part. Of course, there are no limits to decomposing the dynamic aspects to lower levels of detail.

Modelling further associations between objects

Object diagrams can be used for showing both the dynamic and the static aspects of behaviour. These aspects can also be mixed. For example we can make an object diagram that shows all interactions involved in one specific scenario, but also all further associations between the objects involved. So notations for showing visibility can also be integrated into the interaction diagrams (see e.g. object diagrams of Booch, figure 131).

Transformation into time-line diagrams

Time-line diagrams emphasise the *call-sequence* and thus the dynamic aspects of the system. This is in contrast to the object diagrams which emphasize the *call-structure* and thus the static aspects of the system. These two types of diagrams can be converted automatically into each other, if they are limited to show the objects, interactions and possible sequences, and if no further aspects are mixed in. There is much arguing as to which kind of interaction diagram is the better one. A closer look suggests that these arguments stem from a different weighting of the disadvantages, and reflect differing backgrounds and tastes.

2.1.4.4 Some further aspects of interaction diagrams

Implicit versus explicit return events, flow of control

Whenever we model requests, we have to choose between:

- modelling the return event from the request explicitly as an arrow for itself (e.g. some event traces of OMT, figure 135),
- including the return event in the call event (e.g. the interaction diagrams of OMT and Fusion, where the label of the message includes always both, the name of the operation to be called with its parameters as well as any return values),
- neglecting the return event at all (e.g. object scenarios of BON).

When the return event is modelled explicitly, the *control flow* and the *scope* or *duration* of the operations can be deduced from the order of the events, as long as it is clear that requests and interprocess interactions can be assumed. When the return events are modelled implicitly, then the flow of control and the scope of the operations need to be shown explicitly. OOSE shows the scope of operations by rectangles (figure 111), OMT uses the same symbol to show not the scope of the operation but the flow of control (figure 135), and Fusion uses a special numbering scheme for the sequence labels to indicate with them the flow of control and the order of the return events (figure 117). When we model concurrent systems using notifications, then the flow of control and the scope of the operations cannot any more be deduced from the sequence of the events.

Great confusion may arise, when it is not clear if a diagram is overspecified because it shows both return events and control regions in case only requests are allowed, or if it just uses both requests and notifications. The interpretation of interaction diagrams may also become difficult if it is not clear whether a sequential or a concurrent system is being modelled.

Inheritance hierarchies of objects

If time-line interaction diagrams are used for the detailed design, it is possible to add annotations concerning the object inheritance hierarchy. For this, the lines of the objects are replaced by rectangles. If an object inherits operations, its superclass is shown by a rectangle inside its own rectangle, and the messages are annotated with the symbols “i” for implementation and “d” for declaration. Depending on where the arrow points at (inner or outer rectangle) and where the annotations are placed, we know where the operations are declared and where they are implemented [Baklund95]. In figure 151, for example, the operation *createShape()* is declared in the supertype *CreationTool* (represented by the inner rectangle) and is implemented in the subtype *RectangleCreationTool* (represented by the outer rectangle).

Real-time constraints

When interaction diagrams are used for higher level models or when time is not a delimiting factor, real-time aspects are not reflected in the interaction diagrams. All possible sequences of interactions are modelled, but the effective time used up by an interaction or an operation is not considered. However, it is possible to annotate the interaction diagram with the worst-case assumptions concerning the execution time of operations and their operation calls.

Modelling all possible courses versus focusing on the main course

Modelling with interaction diagrams all possible courses of all scenario types is often not feasible. Therefore, in order to avoid too many or too complex diagrams, certain methods recommend concentrating on the most important or most complex scenario types, or showing only the main course of execution and maybe some important exceptions (e.g. object scenarios in BON, scripts in OBA, interaction diagrams in OOSE). By way of contrast, a complete model of the global behaviour containing all scenario types and all possible courses of execution is made in Fusion.

Abstracting away low-level details

To make a diagram more readable we may neglect low-level interactions and objects considered inessential. This may be done in two ways:

- by leaving out certain interactions, i.e. we do not show all the interactions with further objects or agents a specific object or system would need in order to respond to a certain event,
- by using conceptual events, i.e. we abstract out effective interaction mechanisms by omitting mediator objects such as interface and collection objects; the effective technical events are replaced by a more abstract conceptual event that is not explicitly represented in the implementation and may connect objects that do not interact directly in the implementation.

It is not always clear if a specific modelling technique or diagram shows all the interactions, or if some interactions are neglected for the sake of simplicity. Also, it is not always clear if the interactions shown are on a technical or on a purely conceptual level.

Furthermore, some or all of the event parameters may be omitted. This is done in most notations using regular expressions (e.g. life-cycle expression in Fusion, path expressions in PEG), but also in some interaction diagrams (e.g. object scenarios of BON, event traces of OMT). Of course, this necessitates that a complete specification of the events is found somewhere else.

Structuring of diagrams

To avoid too heavily overloaded diagrams we have four possibilities:

- we choose a classification scheme that avoids too complex scenarios,
- we deliberately aim at incomplete interaction diagrams,
- we have some notational aids for structuring and decomposing a diagram,
- we have several models which represent the interactions on various abstraction levels.

OOSE, for example, allows in its analysis model to factor out common and exceptional parts of use cases into abstract or extend use cases. In the design, the abstract and extend use cases are then modelled by separate interaction diagrams. Fusion allows multiple operations schemas for one system operation, each schema describing the system response for another set of preconditions. The interaction diagrams may be decomposed into an arbitrary number of diagrams, one for the system operation itself, the others for the more complex methods. Another approach is chosen in BON: object instances may be grouped into object groups; these are used to simplify the object scenarios. The last of the possibilities above mentioned is not supported by any of the investigated methods.

2.1.5 Summary

An overview of the different concepts and their most essential relationships as presented in this chapter is given in figure 11. For the sake of simplicity, not all the details mentioned in the text are represented in the graphic.

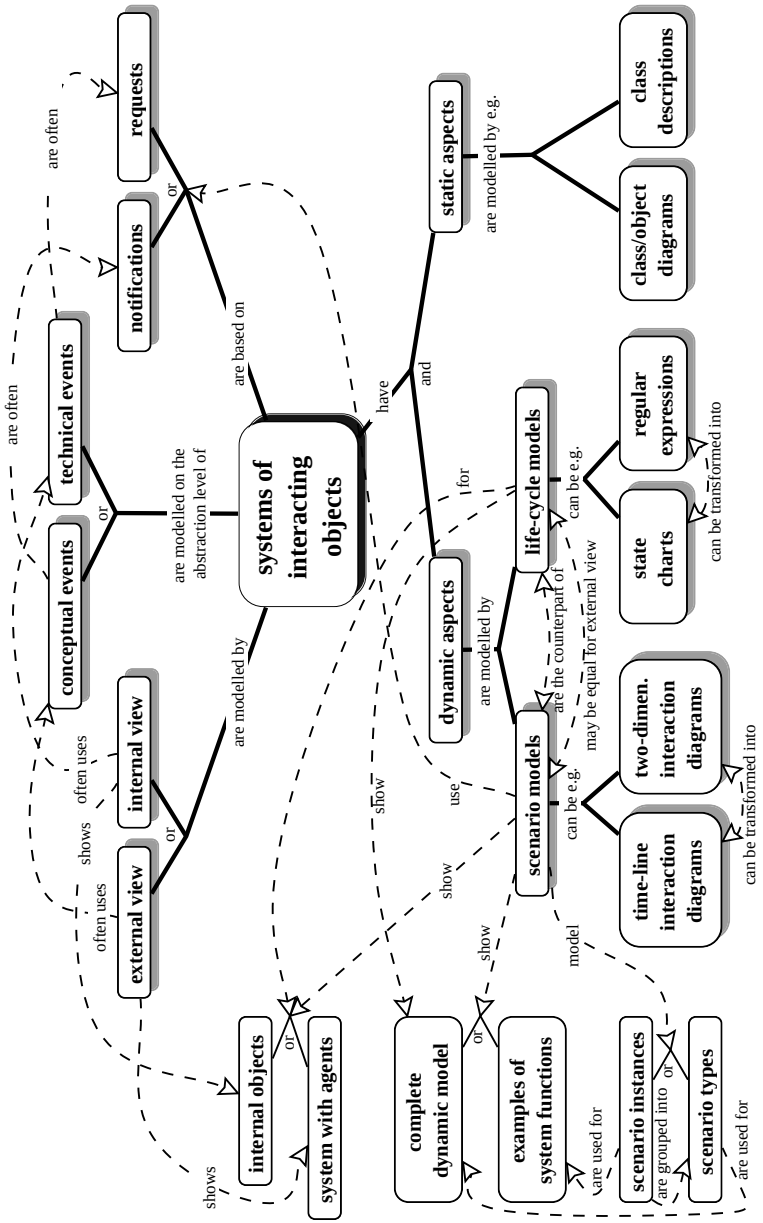


Figure 11: Overview of the concepts for modelling global behaviour

2.2 The Matrix Approach

In this chapter we will have a closer look at the phenomenon which caused the research work on this thesis. We call it the “matrix approach”, and we use this term to characterise certain aspects that appear to varying degrees in several object-oriented analysis methods. We examine three methods which serve here as examples for methods using to a large degree the matrix approach: OOSE [Jacobson92], Fusion [Coleman94] and Modern Structured Analysis (MSA). The latter is not an object-oriented method, but in its analysis model, there is a striking similarity between MSA and many object-oriented analysis methods. Further, many of the drawbacks and difficulties have not changed by shifting to object-oriented analysis. Before methods such as OOSE became popular, certain companies even combined MSA in an adapted form with object-oriented design (see [Briod93] and [Mueller93]); the resulting models were quite similar to those of “pure” object-oriented methods such as OOSE, Fusion or second generation OMT. MSA exists in several variants. Its main elements were introduced by [McMenamin84], and promoted further by [Yourdon89]. We mainly reference here the variant used by the GfAI [Brantschen91]. There are also other methods having many aspects of the matrix approach e.g. M.E.R.o.DE. [Dedene94], but we do not focus on these in this chapter.

We need to emphasize that the matrix approach is only a generalisation of some common features. Of course, not all the characteristics of the matrix approach as we describe them below apply to all of the above methods to the same degree. Moreover, certain methods already offer notations and development processes that overcome some of the obstacles.

2.2.1 Characteristics of the matrix approach

2.2.1.1 Notation

The matrix approach assumes that there is a single, monolithic system that interacts with agents (also called actors in OOSE or terminators in MSA). The analysis model focuses on the interactions between the system and the agents, thus modelling the external behaviour of the system (see figure 12). In spite of the external viewpoint concerning behaviour, the system is not modelled as a black box. In the description of system behaviour, assumptions about its internal structure are made.

In the matrix approach, the analysis model consists of two models: the object model and the scenario model.

The object model in the matrix approach

The object model reflects the data view of the system. It models the data by using entity, attribute and relationship types, and cardinality annotations. Operations are not included in the model and derived attributes may or may not be allowed. An enhanced entity re-

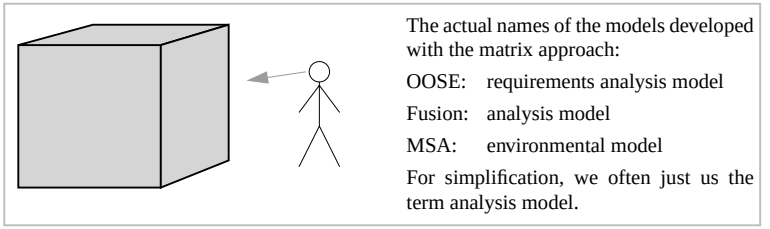


Figure 12: Modelling global behaviour from an external viewpoint.

lationship notation is used. In MSA, this object model is called data model, in Fusion system object model, in OOSE object model.

The object model contains:

- all information that needs to be *stored* by the system beyond the duration of one scenario,
- all data that *flows* over the system boundaries as content or parameters of an input or output event.

These two sets of data need not be identical, although for most systems, especially information systems, they coincide almost completely. Yet sometimes we need to store information concerning the system state that is necessary for the specification of the conditions concerning the reactions of a system but never appears on the outside of the system. Or we may have event parameters that are not relevant beyond a single scenario execution (in some methods this information is not modelled in the object model, but we do not consider these exceptions in the characterisation of the matrix approach).

The scenario model in the matrix approach

As we only focus on the external view of the system, the scenario is a sequence of messages or events, flowing into and out of the system. A scenario type describes a class of such scenarios. The scenario model is basically a list of the descriptions of the different scenario types. Instead of the term scenario type, OOSE uses the term use case, Fusion the term system operation, and MSA-GfAI the term business transaction (business process, business function)[1].

The matrix approach has a flat list of scenario types. When considering the classification criteria presented in chapter 2.1.3.2, these scenario types are determined by the classification scheme two or three, i.e. there is a close mapping between the types of the triggering events and the scenario types. In most methods using the matrix approach, this mapping is one to one (MSA, Fusion). Criteria for the end of the scenario may allow in-

1. In german, the terms “Geschäftsvorfall” or “Geschäftsvorgang” are used, which come from the problem domain of banks and administrations.

termediate interactions or not, but within one method all scenario types must meet the same criteria.

The description of a scenario type contains the following elements:

- the **name** of the scenario type, which is often also the **name** of the input event that triggers the scenario (stimulus of the scenario),
- the **reaction** of the system: output events and/or state changes of the system,
- the **conditions** which determine the reactions of the system,
- **intermediate interactions** with agents (input and output events), necessary to provide the desired reaction (only if the criteria determining the end of a scenario allow intermediate interactions).

The stimulus may be either an input event from an agent or a temporal event. In this chapter, to avoid treating the temporal events as a special case, we consider the clock as also being an external agent. Also, for each event both the agent that sends or receives this event and the parameters of the event are specified. The parameters of the input and output events give the information that is transferred between the system and agents. For parameter specification, entity and attribute names from the object model are used.

The description of the reaction may be algorithmic (e.g. OOSE) or declarative (e.g. Fusion). The specification of the state changes consists at least of a list of the entities that are read or changed, and the description may contain further details. Both state changes and conditions refer again to the entities and attributes of the object model.

The matrix

The matrix approach gets its name from the matrix that can be derived from the object and the scenario model. The first axis of the matrix represents the flat list of scenario types which model an external view of the system. The second axis shows the flat list of entity types (see figure 13). The matrix does not provide any additional information that is not yet contained in the scenario model. So most methods do not suggest making an explicit matrix. An exception is M.E.R.o.DE with its object event table [Dedene94]. Jacobson uses the matrix to show the dualism between use cases and objects [Jacobson95]. The CASE-tool ADW even supports automatic derivation of this matrix [Beringer92b].

That matrix is possible is due to the following characteristics of the matrix approach:

- *Well defined system boundaries*
  The system is expected to have well defined system boundaries. Along these boundaries, the object model (showing the data view of the system) and the scenario model (showing the functional view) are defined.
- *Flat models*
  The data of the system is subdivided into entities, the functionality into scenarios. Both models are flat, i.e. there are no hierarchies and there is no

scenarios \ objects	Oa	Ob	Oc	Od	Oe	Of	Og
S1		x			x		
S2				x			
S3	x	x					
S4						x	
S5			x				x
S6				x	x		
S7		x					x

The symbol ‘x’ shows which scenarios access which objects. If the matrix were a CRUD-matrix, we would use the letters C,R,U and D in the place of ‘x’. These letters would stand for creating, reading, updating and deleting entity instances.

Figure 13: The matrix between the object model and the scenario model.

further grouping or refinement into further entity types or scenario types. These flat models have only three levels of granularity: the system as a whole, the entity types and scenario types, and finally the attributes of the entity types and the events or transactions of the scenario types.

- *Independent models*
  Apart from using the elements of the object model for the specification in the scenario model, the two models are mutually independent. There may even be different teams for developing the object model and for determining the scenario types. Changes in the structure of the scenario model do not affect the object model, and changes in the object model only affect event parameters and state change descriptions, but not the classification of the scenarios into scenario types.

Example Fusion

The scenario model is called the interface model, and is divided into the life-cycle model and the operation model (for more details see chapter A.3). The operation model specifies system operations. The schema of the system operation gives a textual description of the operation. Further, it specifies the data items that are supplied by the input event, the data items that are read, changed or created by the operation, the names of the output events, the names of the agents they are sent to, the preconditions under which the input event can be accepted, and the results of the operation formulated as postconditions. Intermediate input events are not allowed.

Example OOSE

In OOSE, the scenario model is the use case model (for more details see chapter A.2). The use case model divides up the complete functionality of the system into use cases. In contrast to Fusion, the use cases describe scenarios that contain a whole course of input and output events. A textual description of the normal course and of the alternative

courses is given for each use case. The description may be divided up into transactions, and words referring to the object model may be highlighted.

Example MSA

In MSA-GfAI, a scenario type corresponds to a business transaction description, also known as essential activities. The external view of the scenarios is given by the following diagrams:

- Event response list: Each row describes one business transaction. There are columns for the stimuli which are either input data flows or temporal events, for the reaction of the system, for the output data flows, for the conditions, for the reactions and for the agents receiving and sending data flows.
- Context diagram: A dataflow diagram shows the system, all agents, all input and output dataflows, as well as all intermediate interactions. Additionally, for each dataflow the parameters are specified, using entities, relationships and attributes of the entity relationship diagram.

Intermediate interactions are only allowed for very restricted cases [Beringer92b], where the notion that a business transaction has no significant duration (assuming perfect technology) is not violated too much, and where the business transaction is not considered as being finished or being in a stable intermediate state. In all other cases, the business transaction ends with the request for further information. The answer from the agent then triggers a new business transaction. For more details see chapter A.4.

2.2.1.2 Process

The analysis process

The analysis phase is used to analyse and model the requirements and the problem domain. Technological aspects are abstracted out, i.e. the analysis model is technology independent, and of course also implementation independent. The focus lies on a real world model that is free of any design decisions. Furthermore, the model is expected to include all functional requirements of he system, and to be more or less stable for the rest of the development project and even beyond the life-time of any specific implementation.

Before the analysis process can start, the system boundaries must be clear. As the scenario model describes the behaviour of the system at the system boundaries, changing these boundaries after the start of the modelling process necessitates the restart of the modelling process. After the boundaries are known, the development of the scenario and of the object model can be done quite independently. The classification of system functionality into scenario types can be done in parallel or prior to the object model. Yet for the specification of the state changes and the event parameters, the object model is needed as input. Consistency between the two models is reached when all the data elements

mentioned in the scenario descriptions are modelled in the object model, and all the entity types of the object model are used somewhere in the scenario model.

The analysis process finishes as soon as the model is complete and has been validated by the users for completeness and accuracy.

Use of the analysis model for other tasks

The completed analysis model is used for various tasks throughout the rest of the development process:

- *Contracting functional requirements*: The model can be used to validate the functional requirements of the system with the user. In the case of users trained in the easily understandable notation, the users can directly participate in reviews and sign off the models. The model can also become part of the contract for the design and implementation phases of a system.
- *Testing*: Scenario descriptions can also serve as scripts for the system and acceptance tests.
- *Design model*: When moving to the design phase, the analysis model determines the functional requirements of the design model. In addition, the design model may be directly derived from the analysis model.

The transition to the design

The matrix approach advocates a direct and quite seamless transition to the design model. All the objects of the analysis object model become objects in the design model (often called entity objects or problem domain objects) and have the necessary operations added. Additional objects are added for implementing the interface mechanisms, control of the scenario execution (often called controller objects), and coordination among the various objects.

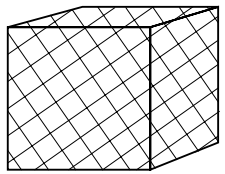
The operations of the entity and some of the controller objects are found by dividing up the scenario descriptions into various steps and allocating these steps as operations to the entity objects or to the controller objects. For the distribution of the steps among objects, various approaches can be chosen (see also [Mueller93]). One extreme is to introduce one controller object for each scenario type; these controller objects take over as much functionality as possible, the entity objects serving only as data containers. The other extreme is to use hardly any controller objects; the various steps of the scenario are distributed completely onto the entity objects. In this case, both the knowledge about how the scenario works and the control over the correct execution of the various steps are completely dispersed.

2.2.2 Difficulties with the matrix approach

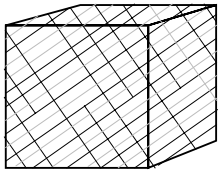
The difficulties we are going to mention here do not apply to the same degree to all methods that can be classified as adopting a matrix approach. Moreover these obstacles cannot be made responsible for all difficulties in a specific project - many other factors determine the success or failure of a project to a much greater degree, and many projects have been carried out more or less successfully though they fought with some of the drawbacks of the matrix approach.

2.2.2.1 Flat list of scenario types

The matrix approach assumes that the external view of the behaviour of a system can be subdivided into a set of more or less independent scenario types which are clearly distinct and do not need further structuring. For many systems this assumption holds. Yet there exist also systems that exhibit a more complex structure in their behaviour. The following figure illustrates the slight yet important difference between a system with a flat event structure and a system which has more complex interconnections between the different possible scenarios and events.



"matrix approach"



more complex patterned surface

Modelling the external behaviour of a system is like sketching an outline of the patterned surface of a cube. When having in the model a flat structure of scenario types, we either have a cube with a very easy pattern, or we have simplified the pattern to such an easy one (left cube). With a model that would allow a more complex structure, we could take irregularities of the pattern into account in our analysis model (right cube).

Figure 14: Modelling complex patterns

If we consider the possible relationships between the scenarios, we discover that some scenarios show a certain closeness to each other, which may express itself in various facets:

- Some scenarios may be extensions or special cases of other scenarios.
- Some scenarios may have common parts.
- A scenario that can be directly invoked by the user may also be part of another scenario.
- Some scenarios may be part of a larger task and are often executed together.
- It may happen that a certain scenario can only be executed if certain preconditions are fulfilled. Such preconditions are the previous execution of certain other scenarios, or specific output events or state changes of the previous scenario.

By classifying scenarios into scenario types, some scenario instances that are very close to each other are grouped into the same scenario type. Yet for the rest of these relationships between scenarios the assumption of having a system that can be modelled by a flat list of independent scenario types leads to some severe limitations. These are presented in the following paragraphs.

Specialised scenarios

In Fusion and MSA, all scenarios that start with the same input event are classified into one scenario type. In OOSE, they may be classified into more than one type in order to avoid the situation where a single type handles too many variants. But whenever scenarios start with different input events, they are classified into different types. Yet these types may only be variants of each other. From a logical viewpoint, they are all special cases of a more general scenario type that would be triggered by a more general event type. In a flat list of scenario types, this cannot be expressed.

The influence of the names of the input events on the classification of scenarios

In the matrix approach, the structure of the scenario model is determined by the types of the triggering events. There is at least one scenario type for each stimulus type. Yet the specification of the stimuli is quite arbitrary. As a simple example, consider the scenario model of an address system. In a first model, we might find the external event *access_address (mode, name)*, where mode has the values read, update or delete. In a second model, we might find the events *read_address (name)*, *update_address (name)* and *delete_address (name)*. In the first case we get one scenario type with several variants, in the second we get three different scenario types. We may judge the first model to be too unspecific for our purposes, and prefer the second model. Yet in the second model, we cannot represent any more the fact that these three scenario types are very closely related.

If the first criterion for the end of a scenario type is chosen (see chapter 2.1.3.3), then the specification of the triggering event determines also the end of the scenario. If the parameters of the stimulus already contain all necessary information for a certain task, this task is modelled as one single scenario. If during the scenario execution further information is needed, then this information must be requested from the agents by further interactions. Each input event of these interactions starts an new scenario type. A sequence of several scenarios results. The fact that these scenarios are closely related to each other or have to follow each other immediately cannot be shown. Example: ordering an item can be modelled using one single stimulus *order (name, set of [item_nr, quantity])* or using the stimuli *order_for (name)*, *request_item (item_nr, quantity)*, *submit_order*. In the second case we have three scenario types, normally executed in the given order with several repetitions of the second scenario type.

No handling of redundancy across different scenario types

When we assumed a flat list of scenario types, ideally there were no redundancies between the different scenario types. This would imply that the classification would be completely disjoint, and that all scenario instances that have identical parts would be also triggered by the same input event and thus grouped into one single scenario type. Yet reality is often different, and the discovery and elimination of redundancy is of great importance in producing an understandable, maintainable and useable scenario model.[2]

A special case of redundancy is given by those scenarios that can be both directly triggered by an external agent, and also appear at the end of another scenario. In both cases the scenario is triggered by the same event, once externally and once internally. In the matrix approach such a scenario type is modelled twice, once as a scenario type, and a second time as part of another scenario type.[3]

No support for dependencies between scenarios

When dividing up the scenario model into many short scenario types, we neglect the fact that some of them appear in a certain order and together make up a longer scenario type.[4] Such a longer scenario type cannot be represented in the matrix approach.[5] To solve the problem by grouping such scenarios together into one chapter in the project documentation is a pragmatical solution that is hardly satisfactory.

Furthermore, certain scenarios can only be triggered if certain other scenarios have been executed beforehand. Consider the scenario model of a teller machine. We might have the scenario types *insert_card, change_code, get_balance, withdraw_money_with_receipt, withdraw_money_without_receipt, abort_session.* All but the first scenario type depend on the first scenario type. They can only be triggered after the first one has been executed successfully. In this example, the necessary order of scenarios is in respect to the system as a whole. We can distinguish different levels that are predicated by these dependencies:

- *Dependencies in respect to the system life-cycle*: the whole system goes through different states in which only certain scenarios are allowed, i.e. all scenario instances of a certain type are rejected or allowed.

2. OOSE diminishes redundancy with the *uses* relationship and the *abstract* use cases. In MSA and Fusion, the problem of redundancy is not that great, because these methods use the first criterion for the end of a scenario. But there the price for a low redundancy is a scenario model with many short scenario types. If these are not structured any further, the problem of redundancy is just solved by enlarging another problem.
3. To avoid this kind of redundancy, in MSA-GfAI [Beringer92b] the notion of internal events has been introduced. This allows us modelling scenario types triggered by two different “agents”, though one of the agents is not really an agent but another scenario sending an internal event. Internal events appear also in [Kowal92], where they are used to trigger those scenarios that are the consequences of certain anomalous system states.
4. This is not the case in OOSE, where the use cases normally represent quite long scenarios. With the *extend* and *uses* relationships structuring techniques are offered to factor out certain parts of a use case. But also OOSE offers no possibility to show further dependencies between its use cases.
5. For this reason Fusion has introduced the life-cycle expressions. These allow us grouping the system operations into longer scenarios, and also define the possible order of system operations.

- *Dependencies in respect to single object instances*: the triggering or not of a certain scenario instance depends on the specific object instance this scenario instance is referring to (parameters of input events or internal objects). Some scenario instances may be rejected, other scenario instances of the same type but referring to other object instances may be accepted.
- *Dependencies in respect to a subsystem*: instead of the system as a whole or of one single object instance, the preconditions for a scenario may also concern a whole subsystem.

Take a banking system. Before any other scenarios can be triggered, the system must be initialised. Only then is the system ready for other scenarios. These dependencies concern the system as a whole. But when we look at one specific instance of the object *account*, we uncover a further dependency. The bank account must be opened before any transactions can be made on it. This dependency concerns only one specific object instance.[6]

Dependencies between the scenarios are an important aspect for most systems. If they are not specified, an important part of the behavioural model of a system is missing. Therefore many methods have added pre- and postconditions to their scenario descriptions, or introduced some kind of dependency graphs or system life cycle specification (see also [Armour95] described in chapter A.9.8, Fusion in chapter A.3, and OBA in chapter A.5). If the method using the matrix approach does not provide any additional notations, all of above dependencies cannot be shown.

2.2.2.2 The matrix approach supports the modelling of only one abstraction level

In the matrix approach, the scenario types are modelled on one single level of abstraction. This level may vary from method to method, and there is also a certain freedom for the projects to select the level best suited for them. A very low level may be chosen, where all events appearing in the scenarios and especially also the input events are technical events. Or a higher level may be chosen, where the events abstract out whole interfacing mechanisms. Even details concerning variants of the scenarios or parameters of the events may be omitted, and one scenario type may encompass several scenario types of a lower level model.[7] But even when the method leaves it open to the user to choose the abstraction level that suits his needs in the best way, the scenario model can only be represented on one single level.[8]

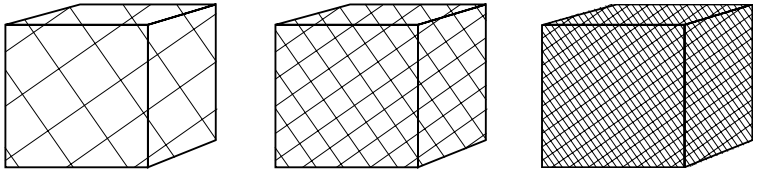
6. In Fusion, dependencies that concern the system as a whole can be represented in the system life cycle. Dependencies that concern only specific object instances, need to be modelled in the preconditions of the system operations. Other methods such as OMT have introduced state charts for object types to model such dependencies. If the dependency concerns a whole system or subsystem, then this approach requires that the object that handles the life-cycle of this subsystem is already known at analysis time.
7. In [Kolbe95] - an experience report on using use cases for requirements modelling - different abstraction levels are chosen in different projects, depending on the complexity and other factors.

The need for multiple views on different levels

Not every person involved in a project needs the same view of the system. A manager needs a model that shows only the most essential elements on a very high abstraction level. For validating the functional requirements (not the user interface) with the user, a conceptual level is necessary that abstracts out all design specific details. For the transition to the design, a model on the level of individual user interactions is preferable. Also in the final system documentation of a complex system it becomes necessary to show the scenario model on two or more abstraction levels, in order to help newcomers to become acquainted with the system.

The chosen abstraction level determines:

- to what degree the parameters of the events are specified,
- how the triggering events are defined and thus how fine grained the scenarios' classification into types is, and how detailed are the descriptions,
- if intermediate interactions between the system and the user are modelled at all, and if they correspond to the interactions in the user interface of the final system or not,
- if the scenario model focuses only on normal system behaviour or if exceptions and error handling are also taken into account.



Three views of one and the same system but on different levels of abstraction.

Figure 15: Abstraction levels

Integrating different abstraction levels

Of course, also in the matrix approach it is in principle possible to represent more than one abstraction level by having more than one scenario model. In [Jacobson95] making several independent use case models is suggested.[9] But if the relationships between the

8. [Armour95] states: "When modelling large business systems, we have found that a single, flat level of use cases is insufficient to effectively capture and partition the large amount of functionality present."

   And [Regnell96] states: "Furthermore, no modularisation concepts are given to manage large use case models."... "Another general problem with use case modelling is granularity. How detailed should we be when describing use cases? How large should the scope of each use case be? "

9. The integration of different abstraction levels of use cases into one single model is considered as being too close to functional decomposition. So only a very weak traceability between the use cases of the different models is provided.

scenario types in the different models are not specified precisely, traceability and maintenance become nearly impossible. What in a higher level model may be considered as a variant of an existing scenario type is modelled as a separate scenario type in a lower level model. Or what on a lower level are several independent scenario types, is represented as one single scenario type in a higher level model. Even worse, the mapping between the scenario types may be n:n instead of 1:n. Having several models or views of the scenario model on various abstraction levels is not realistic without a clear integration of these levels.

The process of defining scenario models

Once the method or the project team has defined on which abstraction level the scenario model is to be developed, the difficulty arises in hitting this abstraction level right away. In the matrix approach it is assumed that this is trivial and that the scenarios on the desired level can be picked directly in the real world. In reality, when there are two persons making a scenario model, then probably two different models emerge on slightly different abstraction levels. The matrix approach offers no help in merging these models. Furthermore, if it is decided to make a lower level model, the development team discovers that it is just not possible to find directly the correct triggering events and scenario types. Then the team has no other choice than to develop first a higher level model, and after that to derive from this model a more detailed one. The matrix approach does not provide any concepts or predefined transitions that support this process.

2.2.2.3 System boundaries

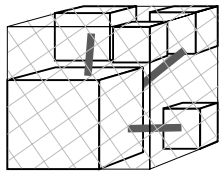
The matrix approach addresses only the modelling of monolithic systems. It models behaviour by showing the interactions that cross the boundaries of the system. It is assumed that these boundaries are known before the development of the scenario model starts, and that they will not change any more as long as the scenario model will be under construction or will be used as a basis for the design and testing activities. Though the assumptions of a monolithic system with known and fixed system boundaries holds for many development projects, there are also exceptions.

It is not always known in advance which parts of a system will be automated at all, or which parts will be in a first increment. So in a first analysis model the system boundaries may encompass more than the final system. Moving from this model to a model of a smaller system means redoing the scenario model more or less completely.

The matrix approach is also unsuitable whenever two views of the system are required: one view showing the behaviour of the system on its automation boundaries, another view showing the behaviour of the whole organisational system. These two views result in two independent models that cannot be linked to each other.

In the matrix approach, it is assumed that between the objects and the system as a whole there are no intermediate components such as subsystems. But as soon as we deal with systems that should be subdivided into one or several layers of subsystems, it becomes

necessary to model also the external view of these subsystems and how these subsystems interact with each other. This model should not be disconnected from the scenario model of the external view of the system as a whole, and we might want to reflect the subsystems also in the data model.



Viewing a system as a composition of several interacting subsystems.

Figure 16: Composed systems

2.2.2.4 Difficulties in the transition to a good oo-design

The matrix approach assumes that a seamless transition to the design is possible. The objects identified in the analysis and modelled in an ERD are taken over as problem domain or entity objects to the design and are completed there by operations. The operations are derived from the descriptions of the scenarios. This is achieved by subdividing scenarios into steps until the steps can be assigned as operations to the objects. Where necessary, new objects are introduced in the model (see chapter 2.2.1.2). When we look at textbook examples, this approach seems to work quite well. Yet when this approach is applied to real projects, where the implementation and the design models do not exist before the analysis model is made and where the models are not reworked again and again for demonstration purposes, certain difficulties arise. These difficulties do not prevent a team from getting a design model very quickly, but the quality of the design model may be bad.

Data driven design

In [Sharble93] two projects using two different design methods were compared on the quality of their models. The result was that the model developed by a responsibility driven method was much better concerning low coupling between the objects and high cohesion within the objects than the model developed by the data driven method[10]. In the responsibility driven method the objects were found by determining the responsibilities necessary in the system and not by deriving the objects from a data model or from an ER-like object model showing only the data view. A similar observation is documented in [Eckert95] concerning design object models developed in student projects by the Fusion method. Many had a bias towards data models and failed the quality criteria given by the method itself [Coleman94, chapter 4-6]. Deficiencies of object models that are derived from purely static data models are also discussed in [Wasserman92].

In the data driven design, many important design decisions concerning the assignment of attributes to objects are taken during analysis without considering the future behaviour and responsibility of this object and are thus quite arbitrary. First the data is encapsulated, then operations are added. Data and functions are not encapsulated in the same step. The capsules may be optimal concerning the data, but they are not necessarily optimal capsules of behaviour. By adding further objects such as interface and controller objects this bias towards the original data model is not eliminated. Only a complete redesign could overcome the paradigm shift between a data model with operations added and an object-oriented model (see also chapter 2.3).

Function driven design

When deriving a design directly from the scenario model of analysis, the following dangers exist (see also [Firesmith95]):

- Finding the object operations by dividing up the scenarios into its events, transactions or steps leads to a purely functional decomposition. These steps have been used during analysis in order to describe the external view of the system. They have not been designed to result in good objects.
- Working along the scenarios when designing the internal objects leads to the duplication of functions. To purge these duplications from the system later is quite difficult.
- A 1:1 mapping of scenarios to controller objects or a radical distribution of all control onto entity objects are quite easy, but both may lead to bad design.

The danger of a functional decomposition is also mentioned in [Walden95, pages 140-141]: deriving the system architecture directly from the scenarios often does not lead to a stable and robust system of interacting objects, because the separation of concerns is along system functions and not along object responsibilities. Therefore, if the local behaviour is not reconsidered based on other quality criteria and guidelines such as those given by RDD [Wirfs90] (determining first the responsibilities of the objects) or by [Berard93] (sufficient sets of primitive operations for reusable objects, composite operations for application-oriented objects), the object operations may well share many of the disadvantages we know from the functional decomposition in structured design.

10. We can group object-oriented modelling techniques into responsibility driven, data driven and use case-driven approaches. **Responsibility driven approaches** (the most popular representative is responsibility driven design (RDD) of Wirfs-Brock [Wirfs90]) start by determining the responsibilities of the objects. Though responsibility driven approaches are the best way for getting a high-quality design object model, they do not offer any techniques for specifying the external behaviour or the requirements of a system. **Data driven approaches** (e.g. the method of Shlaer/Mellor [Shlaer88], [Shlaer92]) start with a data model and derive from this data model the objects. Data modelling is part of analysis and can also be used for defining the data view of the system requirements. **Use case driven approaches** (e.g. OOSE and Fusion) also start with analysis and furthermore allow the modelling of the external behaviour. Some of them even explicitly support requirements determination. Many use case driven methods are to a large degree data driven methods as well, and most of them have the characteristics of the matrix approach presented in this chapter.

Architecture of the system

In modern applications the design of the system architecture becomes more and more important (see e.g. [Kruchten95], [Sadr96]). It is the backbone by which everything else has to orientate itself. Architectural elements such as predefined frameworks and design patterns are used. These help to get better understandable and maintainable models. But in the matrix approach the analysis model, i.e. the object model and the scenario model showing the external view of the system, is developed without having frameworks, design patterns and architecture structures in mind. These design aspects have to be introduced in the design. Yet the matrix approach and the methods having the characteristics of the matrix approach do not give any hints as how a design model can be derived from the analysis model and at the same time be based on specific architectural elements.

Changes in the scenario model

The matrix approach assumes that the scenario model developed during analysis reflects the scenarios of the final system and it is expected that the analysis model remains stable. But as [Walden95] mentions: “Once a clear understanding of the problem is reached and the right concepts established, a radically different set of use cases may instead fall out as a result of this understanding, perhaps fairly late in the process.” Enforcing compliance between the final system and the scenario model as developed in the analysis at the begin of the project may lead to suboptimal user interfaces. [Kilberth93] discusses the example of processes that have been manual up to now and should be supported by computers in the future. The scenario model of the analysis reflects the user’s view of the processes as he is used to carry them out with the old technology. A direct implementation of these work routines may lead to a modal system which allows only one standard way of carrying out the work. The positive effects that have been hoped for do not materialize, because in the resulting system the user is forced to follow technically schematized work routines. The system controls the user instead of being a tool in the hand of the specialist. Because the scenario model fixes the developers on one way to see the system, it prevents them from finding optimum computer support, which in the example of [Kilberth93] was a reactive system where the user is left as much freedom as possible in choosing and changing the sequence of small steps.

Traceability to the analysis model

There are two weak points concerning the traceability between the analysis and the design model. The first point concerns the user interface. The scenario model shows the system from an external viewpoint. The design model focuses on the internal interactions. We can show which internal interaction diagrams belong to which scenario type. For better traceability, intermediate models were needed that would show the effective user interactions of the user interface, and would link these interactions to the operations of the internal objects.[11]

Secondly, when in order to get a high quality design model the scenario model is changed and therefore the objects model is redesigned and the object operations do not

correspond to the steps of the scenario description, then the traceability between the design and the analysis model gets lost and the maintenance of the analysis model becomes awfully difficult.

The scenario model of analysis is a list of functions offered by the system. The design shows a network of collaborating objects. To bridge this gap and to provide a good traceability is not easy. [Firesmith95] even compares this gap to the semantic gap between data flow diagrams and structure charts.

2.2.2.5 Is the matrix approach really object-oriented?

We have taken MSA-GfAI as an example of a method that uses the matrix approach for the analysis. But though MSA-GfAI and similar methods have been used as front ends for object-oriented designs, they have never themselves claimed to be object-oriented. This is a first indication that lets us doubt the object-orientedness of the matrix approach.

Other indications are:

- Though encapsulation is one of the key concepts of object orientation, there is no encapsulation of data and functions into objects.
- Data and functions are not modelled on the same level of granularity. The data is modelled on the level of individual objects, the functions on the level of the whole system.
- The scenarios, system operations or use cases are orthogonal to objects. They are purely functional abstractions and could just as well be used as input for structured design.
- Instead of a system of interacting objects, the analysis model specifies a matrix over data entities and global functions.
- There is no concept of modularisation. Other concepts such as specialisation and aggregation are used for the object model but not for the scenario model.

So the question arises, which criteria should be met by an object-oriented analysis method. Object-oriented methods such as [Booch91] and [Wirfs90] are generally accepted as being really object-oriented. Yet they do not model the external view of the system at all. They assume that the requirements are known and incrementally develop the objects of the system as encapsulations of functions and data. As they do not support requirements determination and neither offer an easy starting point for the modelling process nor the basis for contracting projects, methods offering use cases, scenarios and system operations have been eagerly accepted in industry. But the fact that most object-oriented methods have integrated these concepts in one way or the other does not mean that they are

11. Even in Fusion, where the system operations represent technical events, it is not modelled how these events come from the user to the controller object. Also how the messages to interface objects correspond to the output events of the analysis is documented nowhere.

really all object-oriented, nor does it answer the question of what criteria an object-oriented analysis method would have to fulfil or what criteria are at all satisfiable[12].

2.2.3 Reasons for these difficulties

We have characterised the matrix approach and described some of its difficulties. In the following chapters we will take a closer look at some of these difficulties and the circumstances that cause them. We will also discuss and propose approaches and modelling techniques that might help to overcome some of the weaknesses of the matrix approach. The three main sources of the difficulties are:

- **Object model**: The object model is a very fine grained data model that does not allow any higher level abstractions. It does not necessarily satisfy object-oriented quality criteria. The global behaviour is modelled as a matrix between scenarios and these data entities.
- **Scenario model**: The scenarios are modelled on exactly one abstraction level as a flat list of scenario types, determined by the triggering events. Dependencies and other relationships between the scenario types cannot be shown.
- **Goals of the analysis model**: The goals of the analysis model are often defined contradictory, or they are not clear at all.

Modelling techniques that overcome to some degree the flat list of scenarios have already been proposed in various methods. Fusion has the regular expressions that allow us to specify to some degree the possible orders of the system operations. OOSE utilises the *uses* and the *extends* relationship to avoid redundancy and to model exceptional cases of scenarios. In chapter 4 we will introduce a scenario modelling technique that is not based on a flat list of scenarios at all.

Not all the difficulties that arise with the matrix approach are due to having a flat scenario and object model and having a matrix between them. Some of them are caused by the expectations connected to the analysis model. It often happens that the goals and expectations, explicitly defined or only implicitly assumed, are contradictory to each other or have proven to be too idealistic. In chapter 3 we will take a look at some of these goals and at the intent clashes that can arise. We will also show different approaches chosen by various methods to cope with these intent clashes. Our scenario modelling technique presented in chapter 4 will then be based on one of these approaches.

12. Determining what object-orientedness means for analysis does not become easier when we consider statements such as the one of [Chang93] who questions the claim of object-oriented approaches to be the most natural approach and states: “I tend to adopt a functional view when approaching a problem, and the top-down, hierarchical way of devising a functional solution is natural to me”.

2.3 The relationship cardinality domination

The matrix approach uses an extended entity relationship notation for the object model. In spite of many advantages, the use of this notation for an object model has two severe disadvantages that we mentioned in the previous chapter: the object model is very fine grained and flat, and the design model derived from this object model does not fulfil the quality criteria for good object oriented designs and cannot compete with design models developed by other (e.g. responsibility driven) approaches. Now the question arises as to whether the notation itself leads to these two weaknesses of the object model.

Rules for representing data in an ERD

If we model the data of a system with an entity relationship model, then there are some simple rules which dominate the structure of the model. These rules determine for instance if two attributes belong to the same entity type or not, and if a data element is an entity type per se or just an attribute of another entity type.

- Attribute types that have a 1:1 relationship are potentially part of the same entity type. If a group of attributes has a x:m or m:x relationship to another group of attributes, these two groups become separate entity types.
- If an attribute is multivalued (a set of values of the same type), it is factored out into an entity type that has a 1:m relationship to the original entity type.
- If an attribute is non-atomic (a list of values of different types), then this attribute becomes a group of attributes, and this group is factored out into an entity type of its own.[13]
- Whenever a relationship type concerns only one attribute (or attribute group) of an entity type, then this attribute is also factored out into an entity type of its own.

We can loosen these rules when we also allow complex entity types. These may have attributes that are lists of attributes or sets of attributes, and any functional dependencies among the attributes may be allowed. But still there exist certain limitations. For example as soon as we need to model a relationship that does not concern the whole entity but only a group of attributes, we have no other choice than to factor out this group into an entity type. Moreover, whenever two entity types have a relationship that is not a 1:1 relationship, these two entity types cannot be merged into one entity type. The determination of the entity types is dominated by the relationships and their cardinalities.

13. We could add here also the rules for the second and third normal form of ERDs. When using ERDs for object modelling also these are often intuitively followed, though due to the lack of the specification of keys, normalisation is not required and cannot be checked.

Another limitation for ERDs is that they cannot be modularized. A flat and fine grained model is automatically the result when determining the entities by above rules. Fusion advocates representing the object model on several diagrams, but this is only for representational purposes and does not introduce any higher level abstractions with information hiding. OMT introduces subsystems, but also there, relationships are allowed to cross system boundaries. Other methods have subsystems with information hiding (an overview of various concepts for subsystems is in chapter 4.1.2.1), but these models are not directly derived from an ERD. They are explicitly designed and reworked in order to have subsystems.

ERDs and object models

In object-oriented analysis, extended ER notations are often used to model either only the data view of the object model, or to model the whole structure (data elements and operations) of the object model. These extended ER notations extend one of the common classical ER notation by modelling elements for specialisation (inheritance that satisfies the substitution rules) and for aggregation. Both, specialisation and aggregation, can be considered as relationships with a predefined semantic. In a classical ERD, they would have been modelled as an “is-a” relationship, as a “contains” or as a “has-a” relationship. Apart from these differences, the data view of the objects is determined and decided upon in the same way as in an ERD. The result is an object model in which the capsules are not encapsulations of operations and data. They are data capsules with operations added. The model has been determined by the cardinalities of the attributes, by the existence of relationships between certain data elements, and by the cardinalities of these relationships. The following list shows the main characteristics of an object model that incorporates the principles of object-oriented development, in contrast to a data model or an object model biased towards data modelling:

- **Encapsulation**: In an object model several functions are packaged together into a capsule together with the data they are dealing with. The encapsulations are chosen in such a way that they offer optimal interfaces. For a given problem there are several possible ways to distribute the tasks and responsibilities onto the various objects.
- **Information hiding**: Objects are encapsulations with information hiding. Other objects cannot access directly the internals of an object, they are restricted to the services given by the interface. The kind of requests an object offers to its environment need not to map its internal data elements 1:1. The requests can also require some computing on the side of the requested object. Furthermore, objects can also be systems of objects. The access to the internal objects and their services is done by the interface of the subsystem.
- **Relationships**: The relationships in an ERD reflect some logical relations between two entity types. The relationships in an object model are references that are needed for requesting services from other objects. If between two objects a reference is necessary depends on the design of the services of these objects and on the design patterns used. So not every logical relationship between two objects has to be modelled as a reference. Yet if it is done, then probably no alternative design patterns and object interface designs have been tried out.
- **Quality criteria**: The quality criteria for optimal objects and object interfaces are low coupling, high cohesion, good maintenance and robustness against changes. These quality criteria have no meaning in the context of an ERD.

Of course it is possible to use an extended ERD notation to represent a given object model that fulfils the criteria of good object-oriented design. Such a diagram shows the attributes internal to the objects, the interface of the objects (all the services the object offers) and the references between objects. These references are abstracted into bidirectional relationships with cardinalities. ERDs may also be used to represent just the data view of an object model. Then the relationships may either represent effective references between the objects, or logical relationships. Even ERD notations that support subsystems may be used, if the object model that is represented has been designed to have subsystems.

But can’t we also start object modelling by using an ERD? And then gradually optimize the model in order to get an object model that meets above criteria? There is one great obstacle in this approach: the objects get too easily determined by the rules for an ERD. The result is an object model whose structure has been determined by the relationships between data elements and the cardinalities of these relationships, and not by objects as capsules of functions and data and the responsibilities of these objects. Once we have a flat model dominated by data modelling based on relationships and cardinalities, it is very hard to optimize such a model and to integrate into it design patterns that require a totally different model structure. The consequence of the optimization is often a total reworking of the model, resulting in total different sets of objects, operations and references.

Using ERDs in object-oriented development

As a consequence it follows that using an ERD to determine the objects of the system does not lead to the desired object model. If we look at methods such as Booch [Booch91], BON [Walden95], RDD [Wirfs90] or OBA [Rubin92], we realise that these methods do not use an ERD-like notation for determining the objects and thus avoid a domination of the object model by data modelling concepts. Nevertheless there are special purposes for which a data or object model modelled by an ERD is of great value also in the object-oriented development.

Avoiding any bias towards data modelling is not always the best decision for an object model. Though for many applications we may want to avoid such a bias, there are also systems where this bias is welcome. E.g. when we develop applications for database systems, then the optimal object model of the applications may be the one that is as close as possible to the data model of the data base.

When eliciting the requirements of a system, ERDs are just one means to capture the data view. It tells us the knowledge the system must have. This ERD is not the object model of the final system, it just represents the data view of the requirements, often from a user's or domain specialist's perspective.

A similar situation arises when using a scenario model to show the external view of a system. If we want to get the scenario model before we have designed the internal structure of the system, then we just need something as a data model for describing the data referenced in the scenario model. An ERD is one possible choice for this.

Chapter 3
Goals of Analysis

When developing an analysis model, the notation itself does not suffice for determining the content of the analysis model and the end of the analysis process. For this, the goals and intents of the analysis model are used, either implicitly or explicitly. In this chapter we take a look at possible ways to define the term analysis and the goals of the analysis model. We start by considering the reference model of the problem solving cycle. We examine the definitions that refer to analysis as the "what" as opposed to the "how", or as the problem definition phase of a problem solving cycle. We then continue with a discussion of various criteria (including technology independence, real word model, unambiguous or essential model) which are often used to specify the goals of an analysis model. Difficulties arising with these criteria are that they cannot be combined arbitrarily and that some of them are very vague; we propose therefore also some more concrete formulations. In the last part of the chapter we take a closer look at some misconceptions and intent clashes that are due to a too idealistic model of the software development process. Objective real world models, analysis models that are stable after the analysis phase and analysis models which are equal to the high-level view of the final software system are often attempted, but are normally not realistic. This leads to intent clashes with the desire for a seamless and easy controllable development process. We explain these intent clashes and we also describe the three different approaches taken by software development methods to cope with such clashes.

3.1 What is an analysis model?

The term analysis is heavily overloaded. Nevertheless, in many methods the goals of the analysis model are defined very vaguely, though the notational restrictions alone do not determine the content of the analysis model. This leaves much room for personal interpretations. Therefore we examine in this chapter various possibilities for defining the term analysis and the goals of the analysis model.

3.1.1 Motivation, notation, intent and content of models

When developing a model, we hopefully pursue a certain objective which gives us the motivation to undertake that task. Such a motivation may be to give a brief overview for managers, to evaluate the feasibility of a software system, to set up a contract for a specific software system for outsourcing, to provide the necessary requirements specifications to be used by sophisticated developers, to fulfil somehow the notational requirements of certain QA-guidelines, or to try out a new notation.

Depending on the motivation behind the modelling process, we then choose a notation and define the intent or goal of our model (we will use the terms "intent" and "goal" interchangeably). Only such information appears in the content of the model as is describable by the notation and corresponds to the intent of the model (see figure 17).

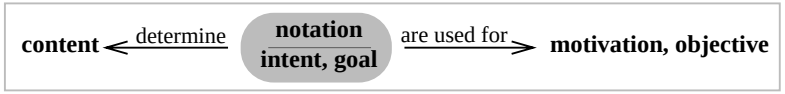


Figure 17: Objective, content and intent of a model

A given notation can be combined with many different modelling intents. For instance, an ERD can be used to show the logical model of all the tables found in a relational database. It can also be used to model all the data in a certain problem domain. Or it can be used to give a static view of all the objects in an object-oriented system, considering only attributes. Even if the notation is exactly the same, the content of the model, even if it is about the same system, may vary significantly.

Considering intents and goals of models is important for several reasons:

- Most modelling notations can be used in several contexts. Yet many methods do not state explicitly for which modelling intents their notations can or should be used.
- When comparing different methods, e.g. concerning their techniques for modelling global behaviour, the differences in model goals are at least as important as the differences in the notations used. For specific method selection, the achievable goals should even be of higher priority than notational issues.
- Terms such as analysis are heavily overloaded. Without making it more precise what the intent of a specific analysis model is, misunderstandings, endless discussions and wasted modelling efforts are inevitable. Stating only that an analysis model should be developed is not enough.
- Managers, end-users, system engineers and software engineers have different motives for making an analysis model.

- To judge the quality of an analysis model, we do not only want to determine if the notation is used correctly but also that the correct content is represented. But appropriate quality criteria depend on the goals of a specific model.
- The definition of the goals of the different models is also important when defining project management guidelines and project control. In a multi-method environment the intents of the individual models are actually the only bases on which a general model architecture can be defined (for more details see [Beringer95]).
- Discussions concerning the seamlessness of object-oriented methods cannot only be based on notational issues. They must take into account the limitations caused by the differing goals of models on different abstraction levels (for more details see [Beringer94]).

In the following we will discuss possible goals for analysis models. But first, we take a closer look at the term “analysis”, and we use for this the reference model of the problem solving cycle.

3.1.2 The term “analysis” from the perspective of the problem solving cycle reference model

Many methods use the term analysis for the first activities that are undertaken in order to solve a given problem. We therefore describe here the reference model of the problem solving cycle and examine how analysis could be defined, based on this model.

3.1.2.1 The problem solving cycle

A software development project tries to solve a certain problem by providing a computerized solution. Yet problem solving is not restricted to software engineering, it happens every day. The problem solving cycle is a reference model that describes the process of solving a problem in a very general way. It has also been used in the software development guidelines of NCR Switzerland and in the QA-guidelines of the GfAI Switzerland.

The four phases of the cycle

The four phases of the problem solving cycle are described in figure 18.

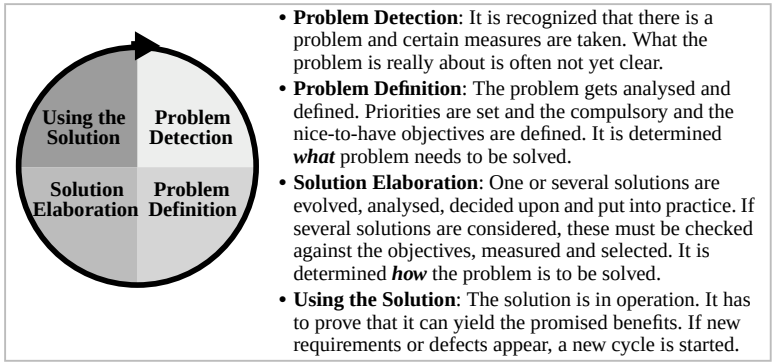


Figure 18: The problem solving cycle

Omitting the definition of the problem

The different activities during the four phases of the cycle vary widely depending on the domain we are looking at. In politics, the elaboration of the solution may demand proposing, discussing and voting on new laws, in engineering this may involve building and trying out prototypes. Yet common to all the different domains is the fact that either the problem is defined *explicitly* before or in parallel to the elaboration of the solution, or this is done *implicitly* during the elaboration or even use of the solution. Omitting the definition of the problem is not possible. It can only be deferred. The objectives can be hidden behind the solution but this normally causes unnecessary controversies and additional costs.

Recursive problem solving cycles

At any point in the cycle, the detection of new problems may give rise to the initiation of a new problem solving cycle within the existing cycle. In fact, any individual activity in the original cycle can be seen as another smaller cycle. This results in a recursive structure of an arbitrary multiplicity of problem solving cycles.

Iterations and backtracking within the cycle

During the problem definition phase it may turn out that the wrong problem has been identified. Or during the solution elaboration phase the problem definition may appear to be ambiguous or to be insoluble due to overconstricted objectives. Or, when putting a solution into operation, some defects may be detected which necessitate some further elaboration of the solution. In all these cases, we have to go back and iteratively redo the

work of previous phases. Of course, it is often a matter of definition whether we consider such backtracking as going back within the same problem solving cycle or as initiating a further problem solving cycle.

3.1.2.2 Applying the problem solving cycle to software engineering

Any work done in the realm of software development can also be seen from the viewpoint of the problem solving cycle. We may apply the reference model of the problem solving cycle to planning a new system, to maintenance work, to implementing a reusable class library, or to developing a new software application. Often the life-cycle of a whole software system is looked at as one single problem solving cycle. The development project starts due to a problem detection and ends when the solution is effectively used. The tasks of such a project can be subdivided into those which serve to define the problem and those which serve to elaborate the solution. They are carried out basically in a sequential order, yet always include some amount of backtracking and iteration.

The recursive nature of the development process

Yet applying the problem solving cycle to the whole life-cycle of a project or software system is only one of many possibilities. In fact each activity within the project and each subsystem or component corresponds again to a problem solving cycle. Due to its recursive nature, the problem solving cycle can be applied at any level of granularity. Restricting its application to the life-cycle of the project as a whole is an arbitrary decision. This becomes especially evident when considering more complex software developments where several project teams work on several applications which together make up some large system, composed of organisational, hardware and software elements. From the overall system view, many details that for the individual teams belong to the realm of problem definition are considered as implementation details. Thus exactly the same activities and models may in one project be considered as belonging to the problem definition of a problem A, and in another project as belonging to the elaboration of the solution to a problem B, without ever making it explicit that problem A could be considered as a recursive problem solving cycle within the solution elaboration of problem B. An example of such an approach is found in [Berard93]. Berard proposes a recursive/parallel life-cycle for software development with recursive cycles of problem definition and problem solution. Exactly the same methods, techniques and phases are used on several levels of decomposition, taking into account the recursive nature of the problem solving cycle.

"What" versus "How"

The problem definition sets out *what* the problem is about. The elaboration of the solution shows *how* the problem is solved. When applying this distinction to a software development project, we can say that in a first phase (or phases) the *what* of the system

under consideration needs to be defined, and in a second phase the *how*. But the *"what - how"* rule does not really help us to distinguish between these two phases or their corresponding models, because often it is not clear if a certain statement describes the "what" or the "how" of the system. This is because of the following reasons:

- The problem solving reference model only tells us that theoretically such a differentiation between statements belonging to the problem definition or to the solution of a specific problem can be made. It cannot give us any guidelines as how to differentiate. Such guidelines are not possible, because only by deciding for a particular case that a statement defines the *what* and not the *how* do we know what the problem is, not before.
- Davis summarises the "*what versus how*" dilemma succinctly as "one person's what is another person's how" [Davis93]. For some people the exact look of the user interface is the *how*, for others it belongs to the *what*. The truth, which is often relative and subjective, depends on both the concrete circumstances and on the perspective and expectations of the people involved. Where we set the boundary between defining the problem and finding a solution depends on what we perceive as being the problem. This again depends on our role in the project (manager, user, network specialist, programmer) and our personal opinion.
- Due to the recursive nature of the problem solving cycle, the *how* of an encapsulated recursive cycle refines the *what* of its outer cycle. Also the *what* of an inner cycle refines the *how* of the outer cycle. Thus a certain statement is at the same time the *what* of one problem and the *how* of another problem.

This dilemma also becomes obvious when considering the criteria mandated by a method to determine the end point of the problem modelling process. These criteria are often very vague and subjective, and they do not really help to define an objective paradigm within which to exercise judgement on the completion of the problem definition.

Limits of the reference model

Though the problem solving cycle is a very valuable reference model for software development and is mentioned in some form or other in many development methods, its significance for practical engineering projects has severe limits. Firstly, applying the reference model only to the level of a whole project contradicts its recursive nature. Secondly, the reference model only makes the statement that there exist such problem solving cycles and that these can be decomposed into four phases.

The reference model helps to analyse the software development process and its different activities from a retrospective view, providing one possible perspective of software development (see also [Floyd89]). It enhances our understanding of what is going on in the development project but it does not determine what in a concrete situation should be considered as problem detection, problem definition or problem solution. Thus it cannot

serve as a guideline for determining in a project if something belongs to the model of the problem detection, problem definition or to the model of another phase.

3.1.2.3 Where does the term “analysis” fit into the problem solving cycle?

Considering the problem solving cycle, there are different elements that we could call “analysis”. And in fact, there are several interpretations of what activities and results are called analysis.

Analysis defined as modelling the “what”

Often it is said that the analysis model defines the problem and describes the “what”, in contrast to the design model which describes the solution, the “how”. Examples:

- OMT: “Analysis is understanding a problem; design is devising a strategy to solve the problem...” [Rumbaugh94d].
- [Booch94, page 252]: “The purpose of the analysis is to provide a model of the system’s behaviour. We must emphasize that analysis focuses upon behaviour, not form. ...analysis must yield a statement of what the system does, not how it does it.”
- [Walden95, page 122] mentions that the current consensus on the meaning of object-oriented analysis seems to be “creating an object-oriented model from the external requirements (a model of the problem)”.
- [Shlaer88, page 93]: “The first thing to do is to build the information model to define the conceptual units you will be working with. In doing this it is important to stay focused on the real world - the problem - and not the solution to that problem, which will be supplied with the automated computer system.”
- In [Graham93] analysis is characterized as “identifying large scale structure, just enough to describe the *what* of the system, rather than the detail of the *how*.”
- [Rawsthorne95]: “...analytic models - a set of models describing *what* the system will do; design models - a set of models describing *how* the system will be implemented...”.
- In [Wilkinson95] analysis and design are differentiated in the following way: “Analysis, or problem modelling, in which the problem is described and represented... Design, or solution modelling, in which a solution to the problem is discovered and represented...”. In the design model “additional classes and mechanisms, which support *how* the system works, are built upon the existing classes, which describe *what* the system does”.

- Davis describes analysis as “understanding the problem and the user’s needs, identifying all possible constraints on a solution; organizing the plethora of assembled information”. This in contrast to the requirements specification: “describing the expected behaviour of the product to be built to solve the now understood problem”. As synonyms for analysis he uses requirements analysis and problem analysis. [Davis90, Eckert93].

Considering what we have said in the previous section, it becomes evident that these definitions help us to categorize an analysis model in a retrospective view of a project, i.e. when the problem is already defined and we know what belongs to its definition and what to its solution. Yet they cannot provide any precise and objective criteria which could help during the modelling process of analysis to specify the goals of the analysis process and to determine the content of the model[1]. For this, other criteria are necessary.

Analysis in its common meaning

If we consider the everyday meaning of analysis as understood in most other disciplines (e.g. as defined by [Webster]: “an examination of a complex, its elements, and their relations”), then analysis activities can be found in any of the different phases of the problem solving cycle. Analytical examinations are necessary to define the problem, but they are also necessary to create a solution or evaluate a solution. Every modelling activity involves the separation of a complex into its elements and relations. This usage of the term “analysis” is referred to by [Walden95, page 122] as being the traditional sense of analysis, prior to its reinterpretation which is specific to current software development methods. Examples for this interpretation of the term analysis are:

- Wirfs-Brock uses the term analysis in this sense when dividing the design process into an exploratory phase and an analysis phase[2] [Wirfs90].
- The term analysis is also used for domain analysis in the sense[3] of taking a closer look at a specific problem domain, business process or enterprise. The resulting domain analysis model has no connection to the future software system, though domain analysis may of course precede a system development effort. Examples are the domain analysis in [D’Souza93] and [Beringer94],

1. The usefulness of the “what - how” distinction for defining requirements engineering or analysis is quite often questioned, e.g. also in [Siddiqi94], where Siddiqi examines the “universal truth” that requirements describe the “what” of a system and not its “how”.
2. The analysis phase denotes the refinement of the objects into subsystems and subtyping hierarchies and the specification of the method signatures. It has nothing to do with requirements definition or problem analysis.
3. Other usages of the term “domain analysis” are:
   - Finding the problem domain objects for the analysis model of a software system. E.g. in OOSE the requirements model consists of a problem domain object model and a use case model [Jacobson92].
   - Defining concepts that are valid beyond one specific software application in order to develop reusable software components for one specific application domain. [Neighbors80, Berard93] state: “System analysis states what is done for a specific problem in a domain while domain analysis states what can be done in a range of problems in a domain...”

the enterprise modelling in [Jacobson92] or the strategic modelling in [OMG92].

- [Jacobson92] differentiates between the process of requirements analysis which has as input a requirements specification and produces a requirements model, and the process of robustness analysis which investigates this model concerning its robustness and produces the analysis model (and not a robustness model as could be expected). So in OOSE the term analysis has two distinct meanings and uses.

Any modelling activity includes analysis in this general sense. Therefore this original and common definition of analysis does not provide any basis for defining the content and intent of an analysis model in the software development process.

Analysis as being equal to specifying requirements

Another possibility is to use analysis as synonym for requirements specification. In the problem solving cycle we specify requirements during the problem definition as well as during the evaluation and selection of a solution. So this definition would cover the second phase and part of the third phase of the problem solving cycle.

In the realm of software development, not all specification issues are decided upon by the same people. For certain decisions the customer takes responsibility, for other decisions the developers have their full freedom, as long as no requirements constraints of the customer are violated. We could therefore say that analysis only contains all those requirements specifications which are under the responsibility of the customer (this can then be seen from retrospective as the problem definition phase). Such a definition of the term analysis is only possible if in the team where the term is used it is clear, which details of the requirements must be defined by the user and which are decided autonomously by the developers. So the content of the analysis model becomes dependant on the organisational context, on the specific project management approach and on contractual conditions. And whenever a participative approach to software development is chosen (as e.g. described by [Reisin90]), this definition of analysis does not work at all.

3.1.2.4 Consequences

All these possible definition of the term “analysis” fail to provide the necessary criteria to determine the content of the analysis model. Also, the reference model of the problem solving cycle only defines that such a model exists, and gives reasons for it. But due to the retrospective viewpoint of the analysis process, it is of limited help as a guideline for the analysis model.

3.1.3 Criteria for defining the goals of analysis models

In the literature, various criteria for the goals of the analysis model are mentioned that should support the developer in determining the content of the analysis model and the end of the analysis process. Figure 19 gives an overview of those criteria which go beyond the simple “what-how” discussed in the previous chapter. Some of the criteria we present in the following sections are very vague, so we propose more precise definitions where possible. Moreover, some criteria require or imply the exclusion of other criteria. We mention some of these contradictions.

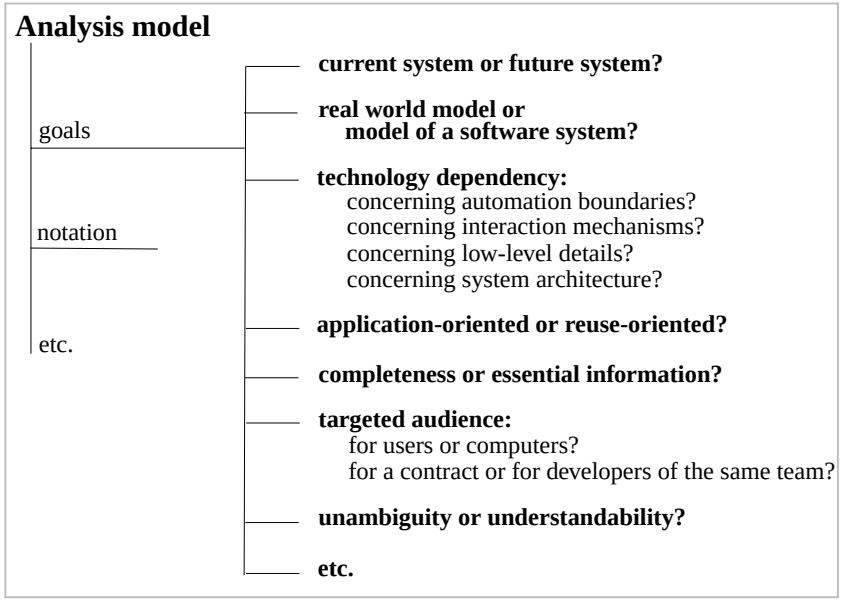


Figure 19: Goals of analysis models

3.1.3.1 Model of the current system versus model of the future system

Introduced by DeMarco in 1979 [DeMarco79], the differentiation between the model of the current system and the model of the future system has not lost any importance, though only in rare cases are both models made. Yet it is important to state explicitly which model is the goal, especially when starting by collecting information on the current system in order to model a future system. Even if the objective is only the replace-

ment of an old system or the mere automation of manual processes, a new system has potential to substantially change current organizations and decision making processes (a list of such changes can be found e.g. in [Huber90]). These differences to the status quo need to be taken into account in an analysis model which has the goal of describing the requirements or the high-level view of a new system.

3.1.3.2 Application-oriented versus reuse-oriented

Methods such as Fusion [Coleman94], OMT [Rumbaugh91] or OOSE [Jacobson92] focus on the development of one single application. Their analysis models are thus application specific. Their goal is to produce a minimal, accurate, and easily understandable description for one application. If the goal of analysing a problem domain is not a specific application but a library of reusable components or an application framework, then the analysis model should show reusable objects and reusable interaction mechanisms. As is shown in [Berard93][4], application-oriented models differ in their structure from reuse-oriented models.

3.1.3.3 "Real world" model or abstracting a software system

An analysis model that reflects the real world and models the problem as the user perceives it (also called real world model[5]) is not identical to a model that abstracts a software system (also called target- or solution-oriented model). Differences include what information is modelled, how data and operations are grouped into objects, how these objects interact with each other, and also how external objects interrelate with the objects within the system. A model that tries to be both, a real world and a target-oriented model, is with the exception of trivial systems either the one or the other. As it is mentioned in [McGinnes92], though many developers pretend to model the reality or to reflect the users' view, often reality is bent to fit into an object model which fulfils quality criteria coming from the software design. Therefore [Høydalsvik93, McGinnes92] emphasize the necessity of a clear distinction. They advocate a model that reflects reality as the user perceives it and is not influenced by any knowledge about what would be good software abstractions. Other methods start right away with a model of the problem domain that uses the abstractions of the software system. Such a model focuses on the targeted software system, and in this approach finding good software abstractions is more important

4. Reuse-oriented models use inheritance heavily, factor out associations and complex operations into classes of their own, parameterize classes, and provide classes with a complete set of primitive methods. For this, a more complex model must be accepted. In contrast, an application-oriented model describes the characteristics and requirements of exactly one application, and is as small and simple as possible. The focus is on the application specific relationships between perhaps reused objects. The classes only need a sufficient set of methods but may include also complex operations.

5. In [Høydalsvik93] this model is called problem-oriented model. The model describes a certain organisation, business, process or machine without considering how this problem will be reflected in the structure and elements of the software system.

than reflecting reality as the user would perceive it before he gets acquainted with the future software system.

3.1.3.4 Technology independent models

Technology independence

Many methods mention that the analysis model should be independent of the design and the implementation of the system, i.e. that technological details should be abstracted away. Formulations such as "describing the *what* instead of the *how*" or "modelling problem-oriented instead of target-oriented" also imply a technology independent model. Examples:

- DeMarco suggest developing logical models of the current and future systems which do not contain any physical information [DeMarco79, page 233].
- OMT gives the following guidelines for a technology independent analysis model: "A good analysis captures the essential features of the problem without introducing implementation artifacts that prematurely restrict design decisions." [Rumbaugh91, page 187]. "The successful analysis model states what must be done, without restricting how it is done, and avoids implementation decisions. The result of analysis should be understanding the problem as a preparation for design" [Rumbaugh91, page 148].
- Fusion: "Analysis is about describing what a system does rather than how it does it. Separating the behaviour of a system from the way it is implemented requires viewing the system from the user's perspective rather than that of the machine." [Coleman94, appendix A].
- The analysis model in MSA assumes perfect technology (no errors, no space or time constraints, no costs). Also no differentiation is made between human and automated processes [Yourdan89].
- OOSE: "In analysis an application-oriented specification is developed to specify what the system offers its users. This specification, which we call the analysis model, specifies the functional behaviour of the system under practically ideal circumstances and without regard to a particular implementation environment... It is important, however, to judge whether the analysis model can actually be realized..."[Jacobson92, page 15].
- Sommerville and Kotonya summarise the content of the requirements specification as: "A software requirements specification is a document containing a complete description of what the software will do, independent of implementation details" [Kotonya96].

Many methods do not specify further what exactly they mean by technology independence and from which technologies they want to abstract in the analysis model. For this

reason, [Bailer93] differentiates in the context of information systems between several types of technology from which we can abstract in an analysis model:

- *Information or implementation technology*: The system is modelled independently of its future implementation as software or manual system. The system itself with its processors, data stores and communication paths is considered as perfect.
- *Organisation technology*: The system is modelled independently of the organisational structures found in the present organisation; the organisational environment is considered as being of perfect technology.
- *Production technology*: The system is modelled independently of what it produces. This abstraction does not make any sense in the realm of software engineering, because any software system supports a specific production technology and cannot be modelled without this.

Abstracting away implementation as well as organisation technology brings the great advantage that the whole organisation in which a software system should be embedded can be rethought and redesigned, and hopefully also optimized, thus combining system development with business re-engineering. It seems ideal to make an analysis model that abstracts away any technological detail concerning implementation technology and organisation. Yet we must consider the following points:

- Even if the processes of business re-engineering and of requirements determination are intertwined they have two distinct goals. Only by making a technology independent model of the current (or future) software system, the business is not automatically re-engineered in an optimal way.
- The metaphor of perfect implementation technology may be useful for batch-type information systems. Yet when developing networking software or new user interfaces, the system under consideration can not exist if we assume perfect technology with no automation boundaries. In most projects, the goal is not a model that abstracts away all implementation technology, but abstracts away only certain aspects.

In practice there is therefore a great amount of uncertainty how to develop and use technology free models in an efficient way. As [Bailer93] mentions, in industry technology free models are often not created and even many examples found in books are not really technology free. But when we want to describe a model that abstract away only certain technical aspects, we need a further distinction. We propose the following criteria: abstracting the automation boundaries, the interface and interaction mechanism, the system architecture, and low level implementation details.

Abstracting away low level implementation details

When in 1979 DeMarco introduced the differentiation between the physical and the logical model [DeMarco79], it was not yet common to abstract away physical details such

as file structures, algorithms and help variables. Nowadays, abstracting away low level implementation details in an analysis model, and to some degree even in the design model, is taken for granted by object-oriented development methods. The encapsulation of data and operations into objects, the principle of information hiding and the possibility to define class interfaces by contracts help to create analysis and design models that are not programming language specific and that even do not contain yet all objects necessary for the implementation.

Abstracting away the system architecture

Many methods advocate an analysis model that does not consider the internal system architecture. The internal system architecture is either defined in a separate model after the analysis model (as e.g. in SSP [Jufer92] or OMT [Rumbaugh91]) or it is not dealt with by the method at all (e.g. Fusion [Coleman94]). As long as a method is used for monolithic and simple systems, abstracting away the internal system architecture may appear to be very obvious, a special high-level model that takes into account the system architecture but no other implementation details may be superfluous. For more complex software systems which may include distribution, off-the-shelf software components (networking software, user interface systems, database systems), or which also integrate hardware components, we certainly also need an analysis model concerning the system architecture, thus having two analysis models with different goals. But not every project starts with an analysis model that abstracts away the system architecture for the following reasons:

- Introducing distribution or off-the-shelf components can change the structure of an object-oriented analysis model totally, causing major remodelling when introducing the system architecture.
- For certain systems all the complexity lies in the system architecture, an analysis model that abstracts this away would be trivial.
- In certain projects the system architecture is already given at the beginning and is not subject to change.

So depending on the concrete circumstances we need to define explicitly for each analysis model whether the system architecture is abstracted away or not and which elements of the technical system architecture are taken into account and which ones not.

Abstracting away automation boundaries

The reason for abstracting away automation boundaries in analysis modelling is that to fix the boundaries of the future software system right at the beginning of a project may not only be difficult, it can also be very dangerous if they are meant to remain fixed as the analysis process goes on. In order that the new system really suits the needs of the users, the system boundaries and the organisational aspects of the environment need to be carefully examined and eventually redesigned. An abstraction from all implementation technology is needed. Therefore many methods advocate a universe of discourse

larger than the automated system. In MSA [Yourdon89] the automation boundaries are even abstracted away for the whole analysis model (also called essential model), they are introduced during the transition to the design. [Shlaer88] distinguishes between the analysis model, which does not specify which processes will be automated, and the external specification model which focuses on the interactions on the automation boundary. [Cook94] has two analysis models, the essential model which abstracts away any automation boundaries and the specification model which takes them into account. In OOSE and Fusion a problem domain object model is created which is free of automation boundaries, yet the use case and interface models are made after the automation boundaries have been introduced into the analysis or system object model.

Other methods, especially those which do not model global system behaviour explicitly, leave it open as to the degree to which the automation boundaries are abstracted away in the analysis model, often assuming that introducing the system boundaries into the object-oriented object model will not change the structure of the object model. This may be true for certain projects but is not the case in general, as is shown in [Eckert95] for an example using the Fusion method and in [Høydalsvik93] for a model of the OOPSLA conference registration problem.

Abstracting away the low-level interaction mechanisms on the system boundary

When we consider those models showing the interactions between the software system and its environment (scenario models that show the external view of the global behaviour of the system), two questions arise: What does it mean if we state technology independence as a goal for our model? Which details of the interface can be abstracted away?

As we model interactions at the system boundary (external events, system services, use cases or scenarios), technology independence certainly does not mean abstracting away the system boundaries. The system boundaries of the software system must be known, but we can decide on which level we want to model the interactions. We can choose a technical level, or we can choose a conceptual level (see also chapter 2.1.2.3 where we use the distinction between technical and conceptual events to characterise different scenario modelling techniques). We propose that a technology free scenario model should be on a conceptual level of interactions and events, i.e. these are chosen to reflect the current perception of the problem domain and not to correspond to the actual user interactions in the final software system.[6] Furthermore, the structure of the scenario model of a technology free analysis must not have any influence on the scenario model of the design[7].

6. Every scenario model, irrespective of the abstraction level chosen, has a bias towards one specific interface design. Therefore a really technology free scenario model cannot exist, we can only define explicitly that the structure of the scenario model of the analysis is not a prejudice for the design model.
7. In chapter 4.5 we will discuss possible changes that only affect the structure of the scenario model.

3.1.3.5 External versus internal model

When looking at a system, a subsystem, a software component, a process or an object, we differentiate between the internal view which describes the system by its components, and the external view which describes the system as a black box. Both, the external and internal view, contain dynamic as well as static aspects.

We could also say that the external view describes the “what” and defines the requirements, whereas the internal view describes the “how” and gives the design of a system, assuming that the external view defines the problem and the internal view a solution. These terms are often used as implying each other, thus defining the terms “what” and “how” by external and internal view. But if the external view really describes the “what” and the internal view the “how” is of course very subjective (see also chapter 3.1.2.2).

Having two different models

Even when describing only the external view of a component, we need a specific syntax and semantic for this. Basically there are two possible approaches:

- The same notation is used to describe the external behaviour as is used for the internal view. Without additional information you cannot tell if such a model represents an external or an internal view.
- Different notations (and maybe even different paradigms) are used for the external and the internal model. The two models cannot be confused, but two different notations need to be mastered and the transition between the two models may be further impeded.

The advantage of the second approach is the clear separation between the external and the internal view. Due to the different modelling techniques, the models need to be redone when changing the view point. In a top-down approach, where first the external view is modelled and afterwards the internal structure is determined, this may be an advantage. Yet in a bottom-up approach, where we already know the internal components and thus can use these as semantic elements for the external description, it hinders us from having only one model that represents both view points. Here lies the strength of the first approach which works well, as long as the goal of the model is still clear and the external view is not mistaken for the internal view.

Whenever we distinguish between the external and the internal view, we must accept having two models that look very similar and are about the same system, but differ in their contents and model elements (e.g. an external data model and an internal object model).

Correlations between the internal and the external views

Ideally, the external view does not make any unnecessary restrictions concerning the internal view, and information hiding is ensured. Far too often, this is not the case: the same model elements (e.g. object types and attributes) are used in the internal and the

external view (see also the discussion on the difficulties of the transitions to a good design when using the matrix approach in chapter 2.2.2.4).

3.1.3.6 Complete and unambiguous models versus essential models

Oft-mentioned goals of analysis models are that they stick to the essential information, do not go too low down into details, are unambiguous and complete. But these criteria cannot be fulfilled all at once. Also, the criteria completeness and unambiguity only really make sense if the analysis model is specified in a formal language.

Completeness and unambiguity of object models and scenario models

We might like to consider an object model to be complete, if we have found all the necessary objects and specified all the external views of all the object services. But in order to find all the objects and services we need to determine how the object services work and if they need further services from other objects. Otherwise we cannot verify whether all the necessary objects and services have been modelled[8]. Furthermore, for an unambiguous model, service names are not enough. The services need to be specified in such precise language. that no misunderstandings are possible. This could be a formal specification language, programming language or an unambiguous domain specific language. A formal language is also necessary if we want to verify the completeness formally and not only informally by human reviewers.

The same is true for the scenario model. Also an unambiguous scenario model would have to use such a language. And in order to be complete, it would have to cover all possible sequences of events, also all exceptional and error scenarios, and not only the most typical ones.

Limits to complete and unambiguous analysis models

There are several points that restrict the development of complete and unambiguous analysis models:

- *Redundancy in the design process*: Total unambiguity necessitates detailing the analysis model (which is not yet the design model) until an unambiguous language level is reached. One criticism of SA/SD has been that the work is done twice: first the system is specified down to the level of pseudo code in the process model of SA, modelling the problem domain irrespective of the design of the software system. Afterwards the same level is specified once again in the design, this time for software components. An effort that is normally not justified for every part of the analysis model. Also the effort for a complete analysis model, especially for a complete scenario model, is not always justified by the expected return of investment.
- *Targeted audience*: The language preferred by the targeted audience is not necessarily an unambiguous language. For example the scenario models of common object-oriented analysis methods such as OOSE, Fusion, OBA and OMT focus on easy understandability, but are not themselves unambiguous.
- *Evolving requirements*: In not every project are all the requirements known from the beginning, and not all requirements can already be fixed in all details. An unambiguous and complete analysis model can then only evolve with the design and implementation of the system (see e.g. also experience report in [Kolbe95].

8. Most analysis model provide a list of the services offered by an object, but they do not mention which other services are needed to provide theses services. No verification concerning the completeness of the object model is possible.

Reducing to the most essential informations

Many object-oriented methods do not try to provide a complete analysis model but advocate focusing on the essential aspects. For example Booch [Booch94, page 253] distinguishes between primary scenarios (which illustrate key behaviours) and secondary scenarios (behaviour under exceptional conditions); only the first are all included in the analysis model. BON [Walden95, page 170] provides only on a very high abstraction level a mere list of all external events and scenarios. A full covering of all possible scenarios on a more detailed level is considered as unrealistic, so only a selection of the listed scenarios are also modelled by object diagrams.

But if we specify as a goal of the analysis model the modelling of only essential information, we have a very vague criterion. On the one hand, what “essential” means, depends on the other goals chosen. On the other, its interpretation remains always subjective, and must be decided upon in the concrete situations by the developers and reviewers, maybe backed up by risk assessment techniques.

3.1.3.7 Targeted audience

To be understood by users (and developers) versus to be understood by machines

If we require that the analysis model is to be understood by those involved in or affected by the system or problem that is modelled, we have to choose an appropriate language and to model the system in such a way that it reflects the thinking and perception of the users. Ideally such a modelling technique is close to other techniques used in the daily work of the users. Examples for this are the concept maps, event-response-lists and storyboards of [Umphress91], the informal scenarios of [Kilberth93], or the scripts of OBA [Rubin92] (e.g. [Heeg94] likes to work with these scripts, as they are so similar to the tables secretaries work with every day).

On the other hand, there is the desire for an automated verification or even execution of an analysis model. This necessitates a language that can be interpreted by machines, e.g.

formal specification languages such as Z, B, VDM, CO-OPN or LOTOS. But the resulting analysis models can only be understood by users who are themselves software engineers. Otherwise the developers first need another analysis model that they can discuss with the users, and based on this they can develop the formal model. Of course, formal specification languages also necessitate that the software developers have the necessary educational background, training and experience for using formal specification languages; up to now this is only true for a minority of project teams.

"Natural" modelling techniques

The targeted audience determines also which modelling techniques are considered as being natural. Certain people consider a purely functional approach as natural [Ward94], others see the world as consisting of passive objects and of processes that manipulate these objects [McGinnes92], for others everything should be modelled as a system of interacting and active objects. Which techniques are perceived as natural is a question of experience, profession, culture, language and personal perspective. As a consequence, not every modelling approach does match well the thinking of the people involved in a specific analysis process, even if it is conceived as being very natural by its originators.

To be used for contracting versus to be used by the same team in an iterative approach

Will the results of the analysis model be used in-house by the same team that has specified the analysis model? Or will the analysis model be used for a contract? And will the analysis model be used for estimating the project size by certain estimation techniques (as e.g. the one proposed by [Moser92])? Also these goals must be known before the modelling process starts in order to avoid nasty surprises.

3.1.3.8 Consequences

Contradictions

An analysis model cannot fulfil all possible expectations at once. We have already formulated various pairs of opposite goals. But also between these pairs there are dependencies. For example an unambiguous model going down to a precise and thus formal language level is not targeted at the user. Application- and reuse-oriented models are always models of the software system. And a system architecture is out of place in a model of the real world. It does not make sense to have goals that contradict each other, neither prescribed by a method description, nor foreseen in the project plan, nor existing implicitly in the heads of the software developers involved in the project.

Many difficulties that arise with methods based on the matrix approach (and with other methods as well) are just due to the fact that the goals of the analysis model are only implicit or that contradicting goals are carried over from a method description. As the goals are not defined and discussed explicitly, the effort is undertaken to develop a model that should satisfy contradictory goals. And as this is not possible, frustration is the consequence. But all those difficulties that are due to unclear, contradicting or even inappropriate goals cannot be resolved by an enhanced notation or an improved analysis process.

Using goals for the project plan

In any project, somebody has to define how many different models are to be developed and what their goals are. If this is not done explicitly before the models are constructed, it will be done implicitly. Then the goals are imposed by the person with the greatest influence, or they are determined with much additional effort when arguing about modelling issues which are in fact uncertainties concerning the goal of the model. Far better that the goals are defined by a method, by QA-guidelines and by the project plan (more about general model architecture frames and project plans can be found in [Beringer95]).

Responsibility of the development team

Some of the criteria we have mentioned here can be defined quite precisely before the development of the analysis model starts. Other criteria (e.g. what the essential information is or how detailed the model must be to avoid too much ambiguity but be still not too complex and still understandable by the targeted audience) can only be stated more precisely during the development process. There it is important that the developers and reviewers are aware of the vagueness of these goals. There is no easy guideline that could replace human judgement and human awareness.

3.2 Intent clashes

Besides the goals mentioned in the previous chapter, we may have further expectations concerning the analysis model and especially concerning the analysis and the design process. Many of the previous goals and of the expectations mentioned here are achievable when the effective developing process (analysis and design processes) is not considered. When it is taken into account, further difficulties arise. We describe these difficulties here as three misconceptions and two intent clashes, and we also show how the various methods cope with these intent clashes.

3.2.1 Difficulties with the ideal analysis model

3.2.1.1 The ideal analysis model

When we are asked for the ideal analysis model, the ideal relations between the analysis and the design model and the ideal properties of the development process, we might mention the goals and expectations presented in figure 20:

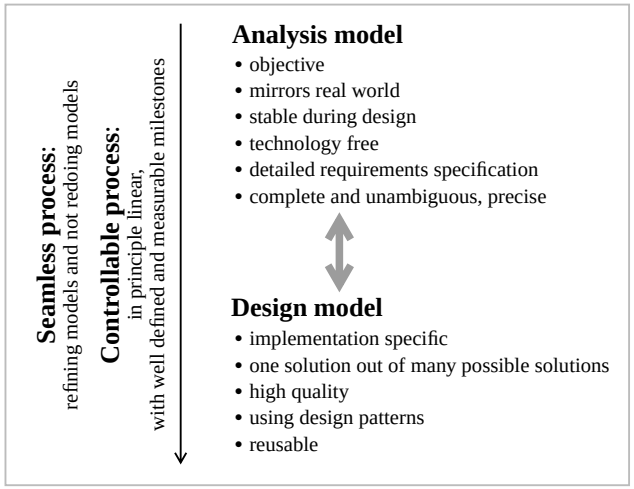


Figure 20: The ideal analysis model

If we specify the goals and expectations in this way, then we have to deal first of all with the problems we mentioned already in the previous chapter; some of these goals are contradictory (e.g. real world model and detailed requirements specification of the software system), and they are not precise enough to be useable guidelines in a concrete project (e.g. technology dependency). But there is also a further difficulty. Some of these expectations are often just not achievable. Though it would be very nice to fulfil them, they contradict the experiences of software development. In many projects it is not possible to make a stable analysis model, an analysis model that is an objective picture of the real world, or an initial requirements analysis model that can be seamlessly expanded into a design model.

3.2.1.2 First misconception: stable model

There exist some software development projects, where the functional requirements are well known from the beginning, and remain stable during the design. But often, this is not the case. On the contrary, there are projects where only a vague idea exists of what the system should do. Evolutionary prototyping or scenario modelling is done in parallel to the design and implementation of the system, in order to find out what the real needs are, and what the requirements should be like. Many people advocate the parallel advancement of design and requirements model, (e.g. [Holbrook90] mentions that the criteria and requirements for a complex design problem evolve at the same time as solutions are being formulated, and [Siddiqi94] quotes: “specification and implementation are intimately intertwined”)[9].

In other projects, the functional requirements are known, or people think they know them, but they refer to the old technology. With a new technology new possibilities arise e.g. for the user interfaces. And these new possibilities not only affect the details of implementation, but also influence the metaphors and work procedures used in the analysis model. Already in chapter 2.2.2.4 we have shown that this may result in changes in the scenario model.

A further difficulty for a stable analysis model is the immense complexity inherent in large projects. It is not possible to model first all the requirements, and then to proceed to the next phase. Instead, an iterative approach is chosen. First the most basic requirements are picked out, and a first system is made. Incrementally, other requirements are added. Though this approach prevents the project becoming bogged down in the analysis phase (paralysis by analysis), the analysis model is never complete, and substantial (and sometimes also increasingly expensive) changes can become necessary as new requirements are added. An example of such an approach to software development is given by Kruchten in [Kruchten95] and [Kruchten96].

3.2.1.3 Second misconception: objective real world model

It is often said that an object-oriented analysis model models the real world. And it is assumed that in analysis there exists a model that is objective and is free of any design

9. In [Wilkinson95] also an iterative approach is recommended. But there, stable does not mean that the model is not changed any longer. An analysis model is considered as being stable as soon as the model begins to be able to provide information about the problem domain rather than require it, i.e. people are getting information from the model and are no longer only putting information into the model.

decisions. This assumption is not backed up by everybody. [Floyd89] states that every model is a construction, and that this construction is always subjective, depending on the background of the person who developed or designed it (see also [Siddiqi94]). There is no objective way to look at the world around us, the model is always shaped by the personal perspective. Even if in a project a so-called objective real world model is made, it is just the subjective construction of that person in the team with the greatest influence.

Our perception of the real world also changes during a project. Metaphors are found and implemented during the development project, and after some time we may even use these metaphors outside this project when describing similar problems. And when we would make the real world model a second time, suddenly we would find us using the metaphors that came from the development of a previous software solution, even in the real world model.[10]

3.2.1.4 Third misconception: initial final high-level view

In a seamless approach the analysis model can be expanded step by step into the design and implementation model (see figure 21). The initial analysis model as it exists after the first phase or milestone of the project is just a subset of the final system model.The same assumption is made for an initial data model which may be part of the analysis model.

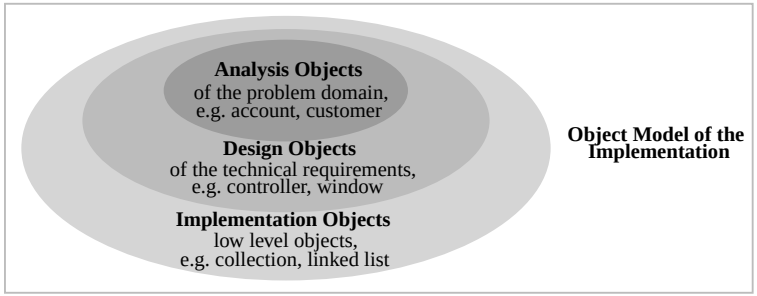


Figure 21: Seamless expansion of the analysis model

Both assumptions hold for text book examples and for trivial systems. But they do not hold when the developers do not know in advance what the structure of the final system will be. If this final system is of a good quality, then it cannot be a mere expansion of the initial analysis model, as we have shown in chapter 2.3 and in [Beringer94].

In reality the model is reworked several times until those design patterns, object abstractions and interaction mechanisms are found that suit the problem, satisfy the quality demands, and will be implemented in the final system. The iterative approach is not a mere expansion and detailing of the model, but includes also changes to it.

10. These metaphors are often very domain specific, and unfortunately rarely get published. An exception is the "tool-material-aspect" metaphor documented in [Gryczan92].

Also, the final system model can be shown on various abstraction levels. The higher level views abstract away certain details of the lower level views, but they still reflect the essential structures and concepts of the final model. But this final high-level view is not the same model as the initial analysis model, even if they use similar notations. If we use an approach where we first model the requirements in an analysis model and then develop the implementation, we have two high-level views of the system which are slightly or drastically different from each other (see figure 22). The differences may only lie in the model elements that are used for the description (e.g. the allocation of attributes to object types). Or the difference may even concern the overall structure of the scenario model and the object model.

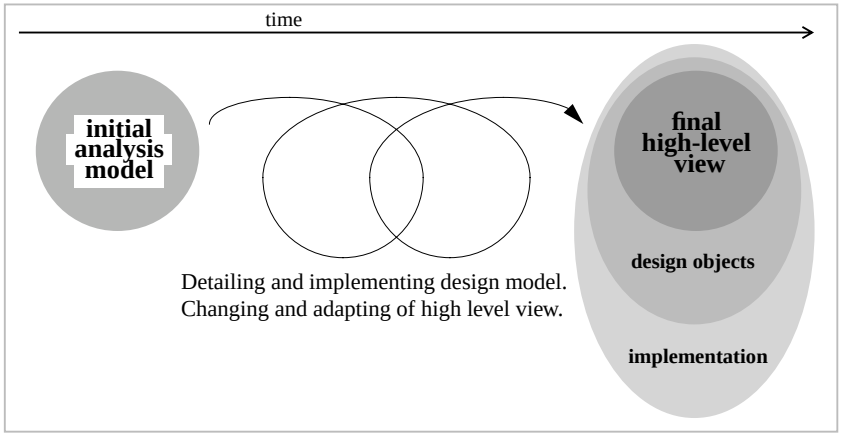


Figure 22: Initial analysis model versus final high-level view

3.2.2 The two intent clashes

The two intent clashes we mention in this chapter do not appear in every project. There are projects where the assumed goals do not lead to any contradictions due to the nature of the given project constraints. Yet in other projects, intent clashes arise and should not be ignored.

First intent clash: controllable process <--> no stable requirements, no real world model

In industry we need controllable processes with milestones. We must be able to define what we expect from these milestones, and we must measure the results. We need to verify that the required work has been done correctly, specifically that the appropriate work

has been done and that no later rework is needed. If we cannot guarantee and measure these criteria at a milestone, then project management becomes extremely difficult, and outsourcing and contracting software is a nightmare.

The consequences for the analysis model are straightforward: we want an analysis model that can be completed in a first phase of the project and remains stable afterwards. Further we require that it is a real world model, so the desired content is specified clearly and we can easily decide at the milestone that the model is complete and correct. With such an analysis model and analysis phase, we would have at least at the beginning of the project well defined and controllable milestones. We could also use the analysis model to estimate the size of the remaining work, and to work out contracts for the development of solutions.

As we have shown above, such an ideal analysis model does not always exist. We may fake an idealistic linear process in the system documentation (as proposed by [Parnas86]), but that does not work for the process itself. We thus have a clash between the constraints of project management, and the possible goals of analysis models and the needs for a highly iterative, and creative approach (see figure 23). The larger the project, the more difficult it becomes to reconcile these two intents. If we look at the proposals for project management e.g. by [Sadr96] or [Kruchten95] and at other guidelines for iterative object-oriented project management, we can see that they try to find compromises that do not satisfy the “purists” of either side. Their goal is a realistic and cost-efficient process that has some iterations, leaves some freedom for changes and creativeness throughout the development, and is still to some degree controllable.

Second intent clash: seamless process <--> initial final high level view

Even before object-oriented modelling techniques became popular, one tried to find well defined links between the analysis and the design model. There were at least two reasons why a well defined or even better a seamless transition between these models was required:

- A clear connection between the various models is needed in order to enable maintenance.
- Rules (and even automation) for deriving the design model from the analysis model and the implementation from the design model are requested, in order to facilitate and control the development process. Often the dream has even been to have automatic code generation from a problem model.

When object-oriented technologies have been introduced, great hopes have arisen that having one paradigm for the whole development process would finally deliver a seamless process. But as we have mentioned before, this works only when modelling the system from a retrospective perspective. So for most projects, the initial analysis model is not identical to the final high-level view. This is very evident in a linear process, when the initial model remains unchanged. But it is also true for an iterative process, only that in an iterative process often the analysis model evolves step by step to reflect the con-

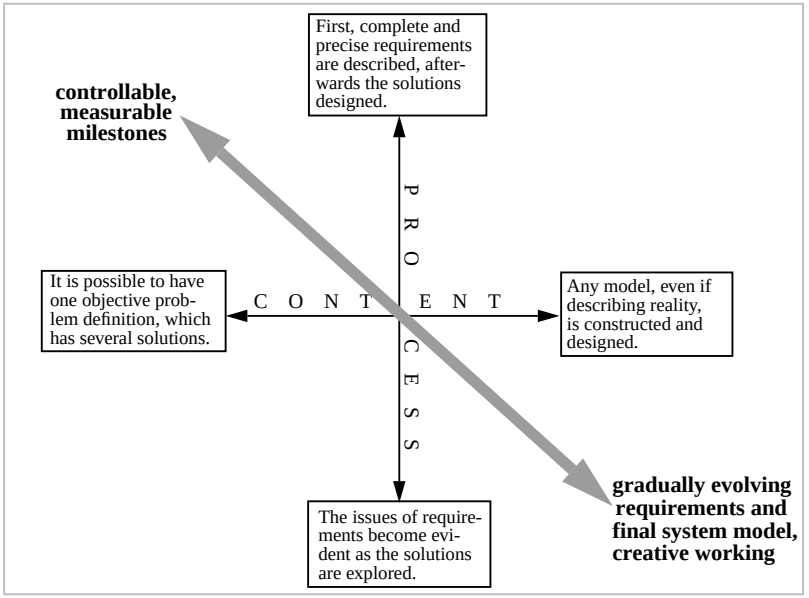


Figure 23: The first intent clash

cepts found in the final system. So the intent clash between the desire for a seamless process and the impossibility of having an initial analysis model that reflects the final high-level view remains.

3.2.3 Approaches to handle these intent clashes

Methods handle these intent clashes differently. We have categorized the methods into three groups concerning their treatment of the intent clashes: methods that have an idealistic approach, methods that just focus on the evolution of the final system model, and methods that bother with developing, reworking and maintaining two different models over the whole development process. This classification mainly concerns if and how the analysis and the design model are distinguished and what are the relations between these two models. But the relations between the analysis and design model also influence the kind of process model that might be chosen. Not every method can be categorized into one of the following groups. Some methods, such as OMT, leave it open to the user how he relates his design and analysis models to each other, and they may be combined with different kind of processes.

3.2.3.1 The idealistic approach

The idealistic approach does not really provide a solution to the intent clashes mentioned above. It just hopes that the clashes are not serious and can be neglected. A more or less linear approach is chosen, and the goal of the analysis phase is a stable and real world analysis model that defines the functional requirements for the system. At the end of the analysis phase, this model is reviewed and accepted. In the design it is directly enhanced or transformed into the final software model and is thus a high-level view of the design model.

Of course, it is admitted that there may be some overlap of the analysis process with the design process and that changes to the analysis model during the rest of the development process cannot be avoided. Yet to reach the ideal process as closely as possible is attempted, and also to produce at least documentation that fakes it. Most data-driven or use case driven object-oriented methods fall into this category, though much depends on how they are interpreted and applied in practical projects. Also, the matrix approach as we have described it in chapter 2.2 is an idealistic approach, and some of the difficulties of the matrix approach mentioned there are due to the two intent clashes.

An example: Fusion

In the first phase of the project, the functional requirements are described by an analysis model that consists of the object model and the interface model. The analysis model describes the intended behaviour of the system as it is externally visible, without any implications for the design and implementation of this behaviour. In the ideal process, the analysis model can then directly be refined and enhanced into the models of the design. Design objects are added and the functions are distributed onto the different objects. The objects and system operations in the analysis model do not only define the external visible behaviour of the system but ideally also correspond to the high-level view (subset of objects, attributes and methods) of the final object-oriented system model. The design model should of course also satisfy the quality criteria for a good object-oriented design.

3.2.3.2 The one-model approach

In the one-model approach, the goal is always the design model. No analysis model is made, work is started right away at this final model with an iterative and evolutionary approach. In the first versions of this model, a part (or all) of the user's view of the system is modelled on a high abstraction level. In the following versions, the model is gradually reworked and refined, until the final object structure, with the final metaphors, design patterns and object specifications has evolved. No model is kept of the initial users' view; it is assumed that his perspective also adapts to the high-level view of the final software system.

The one-model approach provides quite a seamless process (not by expanding the analysis model but by reworking it) and has only become possible by having the object-oriented paradigm throughout the whole development process. It overcomes the second intent clash by not maintaining any initial analysis model. But it is weak in process control. It is very hard to define easy controllable and measurable milestones which are meaningful. This becomes also evident when looking at the methods using this approach: they can only provide quite vague guidelines on how the project could proceed, but cannot provide any predefined milestones with well measurable criteria. Some methods even omit all mention of the difficulties of project control, which does not cause any problem as long as they are used in an environment where no project control is required, as is the case for many small or in-house projects.

Another drawback of this approach is that there is often no documentation of the user's view or of the high-level view. Even if it is assumed that the user's view adapts to the vocabulary and structure of the final software system, having only a detailed design model is not enough for the maintenance of the system.

An example: Booch

"Analyse a little, design a little, test a little" is the motto of the Booch development process. In addition to this microprocess, a macroprocess is suggested that helps to bring some structure into the development process [Booch94]. Each phase of the macroprocess consists of several iterations of the microprocess. Besides the static class diagram other views and diagrams are also supported. Yet their purpose is only to help evolving the object model of the final system.

An example: FORAM

"Object-oriented analysis is analysis, but also contains an element of synthesis. Abstracting user requirements and identifying key domain objects are followed by the assembly of those objects into structures of a form that will support physical design at some later state. The synthetic aspect intrudes precisely because we are analysing a *system*, in other words imposing a structure on the domain." (page 2 of training note 2 [Graham93]).

An example: BON

From the outset BON uses a notation which represents systems of interacting objects. During the development process the initial model is reworked to improve its structure, and further details are added. BON describes the development process using *activities* and *tasks*. Activities are orthogonal to tasks, though some occur mainly in the analysis tasks, others mainly in the design tasks. Analysis consists of tasks for gathering information and describing the gathered structure, and is mainly concerned with the problem domain per se (to which degree technical and implementation dependant details are considered from the very beginning depends on the constraints of the project). Yet since analysis is already concerned to find good object-oriented abstractions, the domain is described as solution-oriented. All tasks of the development process are targeted at evolv-

ing and specifying the interfaces of the classes. Other models than the class specifications, e.g. the object scenarios, are either discarded as soon as they are no longer needed to determine the class interfaces, or they are updated to reflect the final system and kept for documentation purposes. The amount of changes that can affect other views is minimized in that these views do not show much details, e.g. the object scenarios do not indicate operation and parameter names.

An example: Floyd

In [Floyd89] a constructivist perspective is the basis for the development process. Therefore the requirements are not analysed, but constructed. The goal is the implementation of the final system, user satisfaction has the highest priority. The participative design allows that users learn from developers, and developers from users, and that together the new system and the new ways how work should be carried out in the future are worked out (see also [Reisin90] and [Schauer93]). This will also result in new metaphors, work patterns and design patterns. But there is no predefined process for reaching this goal. The process evolves as the system evolves.

3.2.3.3 The two-model approach

Another approach is chosen by those methods which explicitly develop two models, an analysis model for the initial definition of the requirements and a design model for documenting the final software system. The first reflects the user's perspective as it is at the beginning of the project, and it describes the functional requirements as completely as possible, and in a precise and detailed manner. The notation is chosen in such a way that it mirrors best the thinking of the users (e.g. in [Ward94] this is a purely functional description of the system behaviour), and a seamless transition to the design is of low importance. Either the analysis model is finished and reviewed after the first phase of the project, is considered subsequently as more or less stable, and the user does not participate in the design. Or the analysis model is developed in parallel with the design model, but there is a clear distinction between the analysis activities and the design activities, even different teams may be involved. The analysis model is only changed when new or changing requirements arise, it is not adapted to the structure or model elements of the design model. In the two-model approach it is assumed that the design patterns, data structures and scenario structures used in the design do not affect the analysis model.

Most methods using a two model approach also use different modelling techniques and sometimes even differing modelling paradigms for analysis and design. Very naturally this prohibits carry over of any modelling decisions from analysis into design. Here, the analysis model only serves as an information source. In the design a totally new model has to be made, and the model elements of analysis cannot be directly reused, even if the two models go down to the same level of detail and precision. The clear separation of requirements capture and system design is considered as an advantage in the two-model approach. [Ward94] even considers the lack of seamlessness in moving from analysis to design, and the notation change, as an advantage, since it forces the developer to search

for a good design model, and prevents him from carrying over analysis constructs into the design. The disadvantage is a high effort for developing and maintaining two different models in different techniques for things that are quite similar - if no mapping between the two models is provided the maintenance of the system becomes a nightmare. Avoiding changes in the analysis model may also hinder the evolution of a new understanding of business processes, the creation of new concepts and their transfer to the users.

Examples for this approach are MSA-GfAI, MSA as defined by [Yourdan89], and the approaches proposed in [Høydalsvik93], [Holbrook90] and [Ward94].

3.2.3.4 Consequences

The best approach...

Which approach is best depends on the risks faced by a specific project. If the greatest risk is to miss the optimal object-oriented structure in the final system and to fixate on old work procedures, then probably the one model approach is the best one. If the greatest risk is that the project gets out of control, then maybe one of the other approaches is better. None of these approaches is the silver bullet. The intent clashes cannot be overcome, only weakened, and most projects have to juggle with them to some degree.

In this thesis...

For the rest of this thesis the subject of how to specify the goals and expected content of analysis models i not pursued any further. For the enhanced scenario modelling technique SEAM proposed in chapter 4, we assume a one-model approach, hence we emphasize the importance of changing and reworking a scenario model. We assume that the targeted audience of the models are users and any kind of developers, hence we have chosen a semiformal notation instead of a formal specification language. Concerning the other possible goals of analysis models we do not make any assumptions. SEAM may be used for modelling scenarios in various kinds of analysis models, serving different purposes and having different criteria for their content. Which of these goals are finally selected for the analysis model(s) of a given project is not specified by SEAM but must be determined by the project team and is linked to the project specific constraints and risks.

Chapter 4
SEAM: an Enhanced Scenario Modelling Technique

In this part of the thesis, we present the enhanced scenario modelling technique SEAM[1], which overcomes some of the weaknesses of the matrix approach. SEAM includes composition, aggregation, specialisation and extension hierarchies of services, and it is based on the paradigm of interacting objects offering services. In 4.1 we present the starting point of our approach. Chapter 4.2 describes the concepts of services, scenario types, interactions and hierarchies of services and scenario types. Chapter 4.3 presents the notation, which is mainly based on interaction diagrams. Chapter 4.4 introduces transformations of services and of scenario types. How scenario models are developed and documented is discussed in chapter 4.5. and 4.6. Finally, chapter 4.7 contains metamodels and summaries of the most important concepts of SEAM.

4.1 The starting point

There exist a number of differing interpretations of object-oriented (analysis) modelling and development. Therefore we present in this chapter the concepts that serve us as starting point for the definition of the enhanced scenario modelling technique SEAM. This chapter has been mainly influenced by the modelling approaches proposed by Nerson and Walden in BON [Walden95], by Berard [Berard93], by Goldberg and Rubin in OBA [Rubin92], and by Wirfs-Brock and Wilkerson in RDD [Wirfs90].

1. SEAM is an acronym for “Some Enhancements to Analysis Modelling” as well as for “an enhanced ScEnArio Modelling technique”. Furthermore, SEAM hints at the important concept and often misused catchword “seamlessness”.

4.1.1 Models of the system

4.1.1.1 Only one model

As we have mentioned in chapter 3.2.3, there are three fundamentally different approaches concerning the models developed during the development process. For our proposal of an enhanced scenario modelling technique we chose the one-model approach. We assume that the ultimate target of the development process is to arrive at the final system, and that we have basically one model throughout the whole development process, which of course may undergo major changes. This model is target oriented[2]. There are higher and lower level views; higher level views show the system from a more abstract viewpoint, summarizing or omitting details. Throughout the whole model, the same paradigm is used, in our case the paradigm of interacting objects. All views use the same model elements, and as far as possible also the same modelling elements[3].

This one system model can be subdivided into several **views**, which we often also refer to as models. A view shows only certain aspects of the system model. Different views may use different diagramming techniques, but they show the same system and use the same model elements where they overlap. Views may be mutually redundant, but when this occurs it is deliberate. Especially in system maintenance where one tries to understand a system structure which is more or less unknown, having multiple views is very helpful (see also [Meyers92]).

If we use the classification of [Kruchten95], we can partition the whole system model into the logical, process, development and physical views. In this thesis we are only concerned with the logical view, which is the **object model** when using an object-oriented approach. The object model in a one-model approach addresses both the functional requirements[4] and the logical design of the system. Non-functional requirements are mainly solved by the process, development and physical views.

The object model can again be partitioned into several views. A classical division is between the functional, dynamic and data views. This division is still used in some object-oriented methods (e.g. OMT [Rumbaugh91]). In an object-oriented model, these views do not represent different parts of the system in isolation - an object encapsulates functional, dynamic and data aspects - yet we can still meaningfully distinguish these views. The functional view shows all the services an object or a system offers, the dynamic view

2. When developing an application, this target is the final application. When using the modelling techniques presented here to improve an organisation, then of course the target is the future organisation.
3. Modelling elements are the syntactic and semantic elements provided by a method. Model elements are pieces of models that are modelled by using the modelling elements of a method.
4. [Hofmann93] defines the functional requirements as the functions or services that a system or system component must perform, whereas the non-functional requirements are constraints concerning performance, reliability etc. on the functional requirements. Other possible terms are: behavioural and non-behavioural requirements, business and technical requirements [Beringer94], functional requirements and system attributes [Gilb88].

shows the possible orders of these services, and the data view shows the information stored by the object and exchanged between the object and its agents.

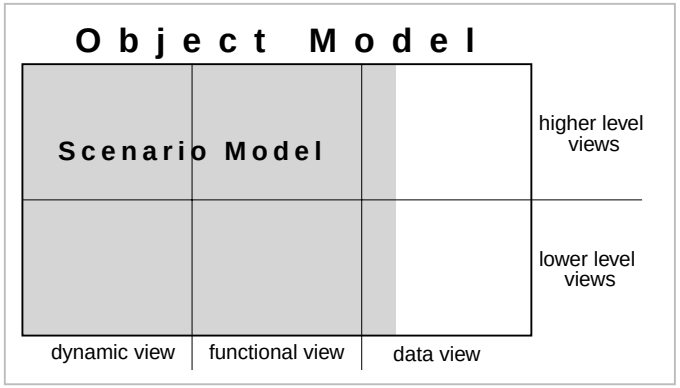


Figure 24: Various views of the object model.

Having a one model approach, the same model elements must be used in all three views. The views are closely interconnected (in contrast to e.g. the views of the original version of OMT [Rumbaugh91], where only in a second phase the correlation between events, functions and object operations is established). The functional and data view are static views: they show what kind of objects, functions and interactions are basically available, but they do not show in which order they can be instantiated. In contrast, dynamic views and diagrams show how the system executes (see also the distinction between static and dynamic interaction diagrams in chapter 2.1.4.1).

When examining the global behaviour of the system by using scenario types and interaction diagrams, we often use the term scenario model. The **scenario model** focuses on those aspects of the object model which show the global behaviour of the system. This includes the behaviour the system offers to its environment as well as the internal mechanisms necessary to provide this behaviour (see also chapter 2.1.2). As we are using an object-oriented approach, the scenario model defines dynamic, functional and data aspects of the system, yet its diagrams emphasize and highlight the dynamic and functional aspects. There is no strict line between the scenario model and the rest of the object model - the specification of an object is imperative for developing those parts of the scenario model using this object, even if it is not in the main focus of the scenario model (see figure 24).

We express and document the whole model or a certain view of it by using diagrams and textual specifications. For the sake of simplicity, we sometimes also use the term diagram for the textual specification.

4.1.1.2 Evolution of the final object model

Creating and approaching the object model

We assume that it is not possible to make the higher level object model (analysis model) by just picking up real world objects (see chapter 3.2.1). First, there is no objective real world model. Any perception or model is a subjective creation, as must be any analysis model. Second, we want to make a high-level model of the system to be built, and even if we wish to mirror the real world as closely as possible, there are some differences. The consequence of this is that the higher level models are also created and negotiated by the people making them, and as their perception of the system changes, the model also changes.

As we have chosen a one-model approach, we desire that the external view of any component is expressed by using its internal structure, and not some other model elements. But in the beginning it is not possible to create the final high level view, and it follows that it is also impossible to use the same model elements in the internal and external view of a component, unless we would simply expand and derive the internal view from the external view. But as we have shown, this would lead to suboptimal models. As a consequence, though the goal is always to make a view of the final model, we can only achieve this goal by encircling and approaching this model step by step. In fact, the model always has a preliminary character. It is the final one until more details have been defined, causing iterative changes in the whole model. Steadily approaching the final model by several iterations is a fundamental paradigm of our approach, but it is not permission for chaos. Nor does it mean that there are no more any project planning, QA-activities or milestones (see also [Beringer95]).

Requirements, analysis and design models

The highest level view of the object model is thus drafted during requirements engineering, reworked during the analysis, and once again redesigned during the design. Initially, the high-level view reflects the requirements as the users see them before having gone through the learning process of building a software system to solve their problem. At the end, the high-level view is the highest abstraction level of the software solution as it is installed at the users' site.

So the terms requirements model and analysis may designate on one hand the results accepted at certain milestones quite at the beginning of the project. On the other hand they designate those parts of the final system model which describe the high-level view of the system and show which requirements the system fulfils.

Seamlessness

Our approach can be called seamless in the sense that the result of the modelling process is one single and coherent object model. All the modelling is based on the paradigm of interacting objects, and only one modelling technique is used for the whole process from

requirements elicitation down to design. Yet our approach is not seamless in the sense that the object model of analysis is only extended into the object model of design. Detailing the model means reworking and changing it. At each milestone we have a different model, fulfilling the goals defined for this milestone and reflecting the final system as it is perceived by the developers and users at this moment.

Furthermore, we do not include and address technology independent domain modelling or business modelling (such as a company wide data model). The integration of such models causes additional difficulties, because an object model reflecting business processes without any automation in mind may differ significantly in its content from an object model that shows a possible computerisation of this business[5], especially concerning the data and static aspects of the object model.

Permanent and transient models

In the development process we distinguish between permanent and transient models (see also [Beringer95]). A **permanent** model is a model or part of a model that after its initial creation (e.g. as result of a certain phase) and acceptance at a certain milestone will be maintained, and is thus subject to change. Some of these changes are due to errors and changing requirements. Others are due to the ongoing evolution of the system model and the changes arising in moving from the initial problem perception and initial design ideas through to the final system.

Other results of the milestones are the **transient** models. They are not updated, and they are either discarded after some time, or archived for administrative purposes. Transient models are also created to support meetings, presentations, walk-throughs or brainstorming sessions.

4.1.1.3 Viewpoints

For capturing the requirements and expectations of the different kinds of users (including those installing and operating the system), we can choose the approach of modelling different viewpoints. Each user-group makes its own model of the system, and we thus arrive at having several different models of the same system when eliciting the requirements. These models may complement each other (as in the caller- and callee-viewpoints in a telephone system, see e.g. [Hsia94]), or they can overlap (as in the viewpoints of different officials having differing needs for an information system). Having multiple viewpoints leads to the following complications:

- We need a framework that allows us to manage these different models and not to lose the simultaneous overview of them all and on their connections (such a framework is e.g. described in [Kotonya96]).
- We may want to verify the consistency between these models.
- We need to integrate the various viewpoints into one single model. This is not trivial, as the different models may use different model elements. Therefore we need ways to transform these model elements into each other.

We will take up the last point again when we discuss the possible transformations of interactions and services in chapter 4.5.

5. Real world objects may appear in the model of the software system twice, once as surrogates and once as agents (e.g. the customer of a bank is an agent of the banking system, but the banking system stores also some information about it in a surrogate object), and the responsibility and behaviour of objects may change drastically (no real-world circle draws itself, circles are passive objects, yet in a software system they normally know how to draw themselves), see also [Hoydalsvik93].

4.1.2 The object-oriented system

4.1.2.1 Excursus: Groups of objects in various methods

Many methods offer modelling elements to modularize the object model. In this chapter we give a short overview of various concepts and terms. We use here the term “group” as a generic term for any kind of grouping. In the different methods the following terms are used: system and aggregation (most methods), physical and catalogue aggregation, subsystem, module and composite object (OMT, [Blaha93, Rumbaugh94c]), class categories, module and subsystem (Booch [Booch94]), subsystem (Wirfs-Brock [Wirfs90]), cluster, subsystem and group (BON [Walden95]), subsystem and block (OOSE [Jacobson92]), composite object (Kilov [Kilov94]), kits and system of interacting objects (Berard [Berard93]), layers (Graham [Graham94]), domains (Shlaer/Mellor [Shlaer92]), etc.

Grouping of object instances versus grouping of object types

A first important distinction is between grouping of object instances and grouping of object types (classes, specifications). For example the *groups* of BON, the *subsystems* of OOSE and the *composite objects* of OMT assemble instances. Thus the same object type may appear in different groups. In contrast to this are the *clusters* of BON, the *subsystems*[6] of OMT and the *class categories*, *modules* and *subsystems* of Booch group object types. In the majority of these kinds of groups, one object type may appear only in one group. The main purpose of this grouping is to get manageable class diagrams and to distribute the object type specifications onto several files and work-units. Grouping of object types is orthogonal to the grouping of object instances.

6. Under certain circumstances, a subsystem in OMT may be both, a grouping of object specifications and a grouping of object instances, see [Rumbaugh94c].

Grouping object instances

When grouping object instances, there exist various possibilities for defining the semantics of these groups. Different methods use different semantics, yet not every method precisely defines the semantics of their terms for groups. Often certain aspects are left open, or they are mentioned briefly and the definitions are scattered throughout a book or several articles. Also the same terms are used in different methods with differing syntax and semantic. The following list shows the main aspects in which the various definitions differ:

- The group as a whole is considered as a **first-class object** and has a name. This is the case for most of the groups called *subsystem*, for the *composite objects* of OMT and for the *groups* of BON. It is normally not the case for the groups called *aggregations*, though in OMT physical aggregations can be considered as being composite objects. The group as a whole may also itself offer services and may be referenced by other objects or groups (e.g. *subsystems* in Wirfs-Brock, *layers* of Graham).
- The group has a **dominant object** (dispatching object, interface object, controller object, owner object). The concrete responsibilities of the dominant object differ from method to method, depending on other semantic aspects of the group (e.g. if it is a white-box or black-box). Dominant objects are found in *aggregations* (one object knows all the other objects). In the *subsystems* of OMT, dominant objects are optional. The *Groups* of BON and the *subsystems* of Wirfs-Brock do not recognise the concept of dominant objects.
  The role of the dominant object may be played by the group itself (*aggregations*, *layers* of Graham), or by one of its component objects (*subsystems* of OMT).
- One object instance belongs to **several groups** which are not necessarily in a hierarchy. This is the case for some kinds of *aggregation* (e.g. catalogue aggregation of OMT) and for the *groups* of BON. Life-time dependencies are of course not possible.
- The group is a **white-box**, allowing objects outside the group to directly create component objects, to reference component objects or to call the services offered by the component objects (e.g. all kind of *aggregations*, *groups* of BON, *kits* of Berard).
  Or the group is a **black-box**, an encapsulation that offers one single interface and hides the internal structure. References can leave the group, but outside objects can only reference and send messages to the group itself (e.g. *systems of interacting objects* of Berard, *layers* of Graham).
  There exist also many kinds of mixed solutions. In the *subsystems* of OMT objects can only be created by the group, but afterwards objects outside the group can ask for the references to directly address these objects. In [Jacobson95b], a *subsystem* offers several contracts. Each contract is associated with a specific internal class, and these classes are visible to the outside. The rest of the internal implementation is hidden and can be arbitrarily changed[7].
  If the group is a black-box, then the life-time of the component objects depends on the **life-time** of the whole group or of its dominant object respectively. If the group is a white box, life-time dependency is not compulsory for all kinds of groups. For example the *catalogue aggregations* of OMT and the *groups* of BON do not have life-time dependencies.
- The groups are only a **conceptual modelling element**, and are not reflected by any construct in the program. *Subsystems* of Wirfs-Brock are only a conceptual help to structure the model. In contrast, *subsystems* of BON, *layers* of Graham and *systems of interacting objects* of Berard are reflected also in the code.
  If groups are only a conceptual modelling element, then somewhere it must be somehow specified which component objects deliver which of the services that are offered by the group as a whole. Wirfs-Brock records this using collaboration graphs, OMT with low-level and high-level class diagrams and with the rule that services or relationships of the subsystem are automatically handled by the dominant object if not otherwise specified. If the group is also reflected in the code, then there is either a dispatching object (which may be the group object itself, e.g. *layers* of Graham) that dispatches the service calls to the appropriate component objects, or there is a controller object that receives and handles all service calls. As most programming languages do not offer any constructs that would correspond to the semantics of the groups as used in the modelling techniques, a direct mapping is normally not possible.
- Groups correspond to processes; each group has an **active object** and its own thread of control. An example is the use of *subsystems* in BON.
- Some methods allow **only one layer of groups**, so the system as a whole may be subdivided into groups, and these then contain the objects (e.g. *subsystems* of BON). Other methods allow or even recommend hierarchies of groups (e.g. Berard and Wirfs-Brock). They also support a recursive or top-down development of components.

Which kind of groups are found in a method depends also on the concept of relationships. Interestingly enough, the two methods mentioned above which foster recursive development and hierarchies of groups (Berard, Wirfs-Brock), emphasize the modelling of the object behaviour and not the modelling of static relationships. They provide collaboration diagrams and specify one-way references to other objects, but they do not integrate extended entity relationship modelling.

7. The subsystems are called to be black-boxes, in our terminology they were only grey-boxes, because the classes providing the contracts are externally visible. Having visible classes contradicts in my opinion also the goals Jacobson has defined for having subsystems, such as implementation independence and plugability. Jacobson uses the same approach (subsystems with several contracts given by the providing classes) as chosen by Wirfs-Brock [Wirfs90], only there the goal of the subsystems is to improve the design by simplifying the communication flows and streamlining the collaborations between classes.

4.1.2.2 The system of interacting objects in SEAM

In SEAM, we consider an object-oriented system to be a **system of interacting objects**, where each object can again be a system of interacting objects.

Behaviour and state

Each object has an external visible **behaviour**, i.e. it offers services.

Definition 1: *A **service** is a functionality offered by an object. The results of a service are state changes in the object offering the service (these state changes will affect the results of other services) and/or information exchanges with other objects. The services that an object offers make up its behaviour.*

In order to fulfil a service, an object may also use services from other objects and thus may have to interact with these objects. For this it has to know the references of these objects. This information it can either get dynamically, or it can store it in static **references**. In order to react properly to service calls and to produce the desired results, an object not only needs informations from other objects, but has also to store information itself. This is done by its internal component objects or by state variables. Together with the static references to the collaborating objects, these make up the knowledge an object has, i. e. they define the **state** of an object.

Definition 2: *The **state** of an object is given by the values of its internal components (internal objects or state variables) and by the static references to collaborating objects.*

The state of an object is determined by all the services instantiated since the creation of the object. The state thus reflects the history of an object and determines its future behaviour.

Interactions

In order to communicate with each other, objects interact by interactions.

Definition 3: *An **interaction** is a message from one object to another object (or to itself). An interaction has a name and it may have parameters.*

Definition 4: *An interaction of which all parameters are instantiated (i.e. each parameter has taken a value that is a member of the set of values defined by its type) is an **interaction instance**.*

Definition 5: *An interaction that has at least one parameter that is not instantiated (i.e. which is a parameter type specifying a set of possible parameter values) is an **interaction type**.*

An interaction type defines a domain of possible interaction instances by specifying their name as well as the types or values of their parameters. An interaction name without any

parameters denotes either an interaction instance or an interaction type that defines a domain of only one value. Examples of interaction types are *deposit (amount: NumberType, currency: CurrencyType), account_entry (kind: Kind_of_entry, amount: NumberType, currency: CurrencyType)* or *account_entry ("Deposit", amount: NumberType, currency: CurrencyType)*. An example for an instance of the first interaction type is *deposit (354, CHF)*.

Hierarchies of objects

We already mentioned that each object can again be a system of interacting objects. This gives us a hierarchy of object instances (not of object specifications!), each one being a composition of lower level objects. These objects are then called its **component objects** or its **internal objects**. The highest level object is the **system as a whole**. The lowest level objects are the **atomic objects**. In between we have the **subsystems**.

Definition 6: *A **system** or **subsystem** is an object that is a composition of objects (object instances). An object that is not decomposed into further objects is an **atomic object**.*

On each level, the dividing up of the object instances into subsystems is disjoint, i.e. one object instance can only belong to one subsystem (and of course to the subsystems this subsystem is part of). The life-time of a component object depends on the life-time of its system.

Every object offers services, independently of being a system, a subsystem or an atomic object. In the case of systems or subsystems, the services are **system services**, i.e. they can be decomposed into a composition of services offered by component objects of the system. Among the services offered by an atomic object we call those services that are not further decomposed **atomic services**; we will define this term in the context of aggregation hierarchies of services in chapter 4.3.2.1. Whenever we use the term object, this can denote a system, a subsystem or an atomic object. The term service also encompasses both system services and services of atomic objects.

In principle we can have several levels of subsystems. Yet many projects will have either only one level of subsystems, or several levels of subsystems only in one part of the system. Smaller projects even go without any subsystems; there we just have the highest level object which is the system as a whole, and the atomic objects.

Subsystems: white-box and black-box

We differentiate between black-box and white-box subsystems. Both are encapsulations of object instances, both have life-time dependency, but they differ in regard to the transparency of the system boundary.

The internal structure of a **white-box subsystem** is known to its collaborating objects. These may directly reference the component objects or call their services (see figure 25). A dominant object is not compulsory. The services of the subsystem are the union of all the services of the component objects that are relevant to external collaborators (in prin-

ciple another collaborating subsystem may call any service offered by one of the component objects). White-box subsystems serve only as a conceptual modelling element. They are not reflected in the code. White-box subsystems are semantically very close to the composite objects defined by OMT or the subsystems of Wirfs-Brock and are especially useful in a top-down development of the object model. The system as a whole and the problem domain subsystems are normally white-box subsystems.

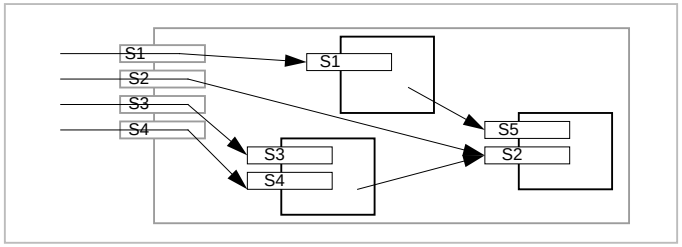


Figure 25: Service calls in a white-box subsystem

In contrast to white-box subsystems, **black-box subsystems** have information hiding. To the outside, only the services of the subsystem as a whole are visible. Collaborating subsystems can only call services of the subsystem, and they only reference the subsystem (see figure 26). The internal structure of the subsystem is not known to them. The subsystem is either itself the dominant object of the subsystem (i.e. the dominant object has the same name as its subsystem) and dispatches all incoming messages to the appropriate component objects, or the subsystem has one specific controller object that receives and handles all incoming messages. Thus the subsystem and the restrictions imposed by it are also existent in the final code. Black-box subsystems are semantically very close to the layers of Graham, and, like Graham, we allow only outgoing references and message calls to pass the system boundaries. Whenever we have wrappers around an existing system, or we want to have one single interface to a subsystem like a DBMS, we use black-box subsystems. Also when modelling distributed or client-server applications it may be convenient to use black-box subsystems for the individual applications.

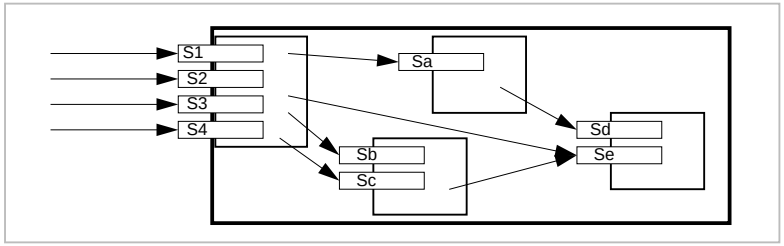


Figure 26: Service calls in a black-box subsystem

Groups

Sometimes we would like to group some object instances together only in the context of a certain diagram; such a grouping should have no further semantic meaning. As in BON we use groups for this.

Definition 7: *A **group** clusters together several object instances in the context of a diagram; one object instance may belong to several groups, and each diagram may group objects differently.*

Like for a white-box subsystem, the services of the group are the services of its objects relevant to the outside. Yet in a group there is no life-time dependency; an object instance may belong to several groups at the same time. Groups have no effect on the object-model. They are only used as a replacement for some objects we do not yet know, or to make diagrams more legible.

Internal and external views of objects

The **external view**[8] of an object describes an object as it is seen from the outside. It tells us how this object works together with other objects on the same abstraction level and what its externally relevant behaviour is. In the external view we describe:

- Services the object offers including the possible orders of these services (life-cycle of the object), preconditions and results of the services, information exchanged by the services (parameters of the interactions), the state of the object as perceived at the outside of the object (external data view) and as used for the description of preconditions and parameters.
- Services the object uses, and references to the objects offering these services.

The **internal view** is a mere expansion of the external view. It shows what the internal structure of the object is, and how this internal structure is used to provide the services as defined by the external view. In the case of an atomic object, the internal view consists merely of state variables and private services. The external view is not independent of the internal view; the component objects defined in the internal view are also used in the specification of the data view of the external view.

The highest level view of a system as a whole need not contain only the external view of the system; it may also contain a high-level view of the internal view. Thus the terms high-level view and external view are not synonyms.

8. We do not use here the terms “interface” and “body” because they are too much linked to the process of programming where first the interfaces are specified and then the bodies are implemented. External and internal views do not imply any chronological order (the external view of a subsystem is often determined after its internal view is defined), and neither the interfaces nor the bodies are of more importance or less subject to change during the modelling process.

4.1.2.3 Implications of this approach

Specification of the services used

We said that the external view of an object includes also the services the object uses from collaborating objects. This has two advantages:

- It is a first step towards gaining composable components (defining all and not only some of the "plugs" of a component [Nierstrasz95]).
- We can use this definition of external view on all abstraction levels, for the system as a whole that communicates with its agents as well as for atomic objects that answer requests by sending requests to other objects.

By assuming that for an object not only the services offered but also the services used are defined, we go further than the class headers in programming languages or the contracts used to specify objects in [Meyer93], as these do not include the services used from other objects.

Parameters of interactions

When describing the parameters of the interactions of services we use data elements. In a complete object model these data elements are given by the internal view of the system (i.e. its decomposition into component objects plus fundamental data types), and also any constraints are defined there. But it may happen that we want to describe the parameters already before we know the decomposition of the system. We do this with a provisional data model which is transient and will be replaced as soon as the internal composition of the system is known. This provisional external data model may be:

- an **entity relationship model** (or extended entity relationship model),
- a **data dictionary** with **business rules** which give the most important constraints between the data elements,
- a **concept map.**

Which modelling technique is chosen for the provisional data model depends on the kind of system and on the other project constraints such as the experience of the people involved, the notation of the models already at hand etc. For an information system an entity relationship model is probably the most appropriate notation, whereas for a process control system a mere concept map or a simple data dictionary may be enough. The provisional data model is part of the external view of the target-oriented system model, and should not be confused with a problem-domain-oriented data model.

As soon as the internal decomposition of a system or subsystem is known, the provisional data model describing the data elements becomes obsolete and the definitions of services must be changed so that the parameters are based on the effective internal component objects. Of course, the internal decomposition must fulfil the constraints given by any provisional external data model, and in many cases looks quite similar to the provisional data model.

Entity relationship diagrams

Entity relationship diagrams (ERDs) or their derivatives (such as extended entity relationship diagrams, class diagrams focusing on data and relationships) still have their place in the development process we use here, even if they are not part of the object model.

- **Problem-domain modelling**: ERDs are well suited to model the static relationships between all kinds of entity types. This can be done independent of any object-oriented software application (e.g. company-wide data models, glossaries). Also, some development processes demand that a problem-oriented model, focusing mainly on static aspects and on data, has to be developed at the beginning of the project. Such a problem-oriented model helps a lot in the learning process all project team members have to go through, and it serves as an information source for any target-oriented model.
- **Provisional external view of systems and subsystems**: see previous section.
- **Modelling the data view of a system or subsystem**: Every system, subsystem or atomic object possesses a certain amount of knowledge. In the object model as defined so far and in the specification of objects as described in chapter 4.2.4, this knowledge is modelled by specifying the internal decomposition of an object and its references to other objects. Yet additionally, we might like to have a more abstract view of the data of a system. For this, we would probably use an ERD. This model is not part of the object model, it is additional and shows the knowledge found in a system, but its structure may be quite different from the one of the object model. As we focus in this thesis on the scenario model, we will not mention this additional data model again.
- **Models for relational database management systems**: In the case where the object-oriented application uses a relational database, it makes sense to have an additional model of the logical view of this database in the form of an entity relationship diagram. Such a model is of course a permanent model and coexists with the object model of the application. It may be that only the overall information content of these two models concerning the persistent data is equal. Or it may also be that each object corresponds to an entity in the ERD, and even that the object model is designed in order to match closely the logical model of the database, a model that may already exist before the targeted system is developed.

In the case of applications that consist primarily of a DBMS, other semantics for the object model than those discussed in this thesis may be preferable. This concerns the concept of references (e.g. two-way references as in the object model of the Object

DBMS Standard [ODMG93]), or the characteristics of object types (we could for instance distinguish between non-persistent controller and view classes and persistent entity and relationship classes). Yet for this thesis we have not investigated the integration of relational, object-oriented or legacy database systems.

Object aggregation

Most object-oriented methods define aggregation relationships between objects. The concrete semantic of this relationship varies widely (compare for instance object aggregation in OMT, Fusion and BON). In this thesis we do not use the term aggregation in connection with objects; we have only introduced the concepts “composition of objects” and “static references to collaborating objects”. The physical aggregation of OMT is quite close to the composition of objects, yet the composition adds the notion of encapsulation and hierarchy. Most other aggregations do not have any compulsory semantic that does not already exist for normal relationships, only their names (like “belongs_to”, “has_a” or “corresponds_to”) and the notation distinguish them from other associations. We model these aggregations with static references. Of course, it would be possible to distinguish between several kinds of static references, some of them being aggregations (e.g. [Walden95] classifies static references into associations, shared associations and aggregations). For the sake of simplicity, and because our main focus is on the scenario model and not on the static object model, we do not make such a distinction and use only the concept of static references.

4.2 Services and scenarios

In chapter 4.1.2.2, when discussing the starting point for the modelling technique SEAM, we gave the definitions for the terms service and interaction, but we did not yet explain in detail how services and interactions are linked together and what scenarios are used for in the description of the services of a system of interacting objects.

4.2.1 Services

Properties of services

When considering the services offered by an object, we can differentiate between the following properties:

- **Information flow**: The information that is exchanged between the server object and its agents
- **Interactions involved**: The input and output interactions that take place between the server object and its collaborating objects, the **possible sequences** of these interactions (scenarios of the service)
- **State changes**: Changes of the state of the server object
- **Precondition**: Preconditions concerning the state of the server object which need to be fulfilled in order that this service can be requested (because services cannot be executed in an arbitrary order), services as sequences of services

Figure 27 gives an overview of these aspects.

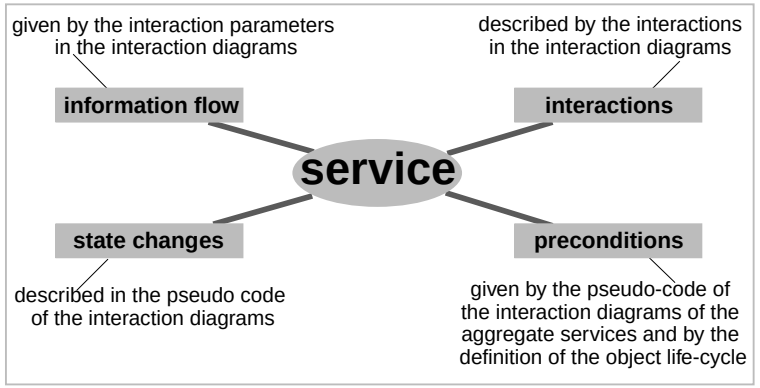


Figure 27: Properties of services

In order to distinguish the various objects involved in a service, we call the object that offers the service the **server object**. The objects it interacts with are its **agents** or its **collaborating objects**[1]. These terms denote only a momentary role of an object. The same object takes on different roles in different contexts. Of course, server object and agents may be atomic objects, subsystems or systems.

Information flow

Each service offered by an object has a certain **information flow.**

*Definition 8: The **information flow** of a service contains all the information that is exchanged between the server object and its collaborating objects.*

When considering the information flow of a service, we are not interested in which order the information is exchanged, and how the information may be packaged into individual interactions. The information flow indicates only the sum of all information exchanged. The total information flow of a service or of a group of services plays an important role when considering transformations of services (see chapters 4.5.2 and 4.5.3).

Triggering events, interactions and motives

Each service is triggered by an event.

*Definition 9: An **event** is something that happens at a certain point of time and that is of a certain importance to a given system or object.*

The **triggering event** may be an interaction from another object, an interaction of the server object with itself, a time-out, or a trigger from another service of the server object. Time-outs or triggers of the server object (internal triggers or internal events) can also be modelled as interactions: in the external view of the service they are messages from the server object to itself, in the internal view they are messages from some component object to another component object or to itself. Whether events that are time-outs or internal triggers are modelled as interactions or are only described textually, depends on the abstraction level chosen. On a technical level, they are always modelled as interactions.

In an object-oriented system, services are carried out by **interactions** between the server object and its agents (sometimes a server object also interacts with itself), and by interactions within the server object in the case of a decomposable system. The interactions are the mean by which a service may be triggered and by which information is exchanged between the objects involved in the service. There may be interactions from an agent triggering the service, and there may be interactions back to the triggering agent. A service may also affect other agents than the triggering one, either by sending them some results, or by asking them for some information that is needed to carry out the service. Furthermore there may be interactions internal to the object that offers the service. All these interactions do not occur in an arbitrary order; there is a **partial order** that determines which interactions have to follow each other sequentially, and which interactions can take place in parallel

Events and interactions are not to be confused with motives.

*Definition 10: A **motive** gives the reason why an event or an interaction takes place.*

An interaction or another event is the consequence of some motive, they are issued due to a motive. The interaction name expresses what is expected from the receiving object, its parameters provide the necessary information. For one and the same event or interaction there may exist several possible motives. The following table gives some examples contrasting the difference between motives and interactions:

motives	**interaction**
at midnight new log-files need to be created	*restart log-file (XY)*
system manager wants to restart log-file XY	*restart log-file (XY)*
user needs money	*withdraw money from bank account (amount)*

The interactions involved in a service, together with other actions such as state changes of the server object, make up the scenarios of the service. As we will see later on, the scenarios can be modelled on various abstraction levels. In those cases, where no information flow takes place between the server object and any agents, and if the triggering event comes from the server object, we may even abstract away all the interactions in the external view of the server object.

The states of the server object

There are three connections between the services and the states of an object:

- services carry out state changes in the server object
- the state of a server object influences the results of its services
- the state of a server object determines the possible order of its services

A service may change the state of an object in that it changes the information stored in the object. These are references to its collaborating objects, values of its component objects or values of its state variables.

1. We treat “agents” and “collaborating objects” as synonyms, though we often adapt to the current usage of the terms by using agents when talking about the collaborating objects of a system, and using collaborating objects when talking about the agents of an atomic object. The term “collaborators” is known from the CRC-cards (see [Wirfs90] and [Wilkinson95]), a technique that focuses on the internal design of a system. The term “agents” is known from context diagrams and other modelling techniques showing the external view of the whole system. Because we use for both abstraction levels the same concepts and modelling techniques and because we treat the agents of a system as objects just like the system itself is an object, we do not make a distinction between the terms agent and collaborating object.

The state information stored in an object does not only influence the result of an issued service. It also determines if a certain service can be triggered at all or not. As an example lets look at an object of type *BankAccount* with the services *Open, Deposit, Withdraw, Balance, Block, Unblock*. The services *Deposit* and *Withdraw* change the balance of the account and thus influence the result of the service *Balance*, and in the case where there is not enough money in the account, also the result of the service *Withdraw.* Yet before any of these services may be used, the service *Open* has to be initiated. If the account has the state open, the service *Open* can no longer be used. The service *Block* changes the state of the account to be blocked, so afterwards only the services *Unblock* and *Balance* can be called.

4.2.2 Various kinds of interactions

In chapters 2.1.2.3 and 2.1.2.4 we have differentiated between various kinds of interactions: interactions which must be interpreted as notifications and interactions which must be interpreted as requests, interactions on a technical level and those on a mere conceptual level. We introduce these distinctions also in our enhanced scenario modelling technique SEAM, though most other methods and modelling techniques do not make these distinctions. Either they do not need to because they only target at one specific abstraction level (e.g. only at the detailed design of sequential systems where all interactions are technical events and are requests), or they use different modelling techniques for the various abstraction levels. Other methods do not introduce different kinds of interactions because they leave the interpretation of the interactions to the user of the method.

Notifications and requests

*Definition 11: A **notification** is a one-way interaction from one object to another object.*

The sending object may continue execution after sending the message without waiting for a response. The receiving object may react to the interaction with zero, one or many other interactions. These may be addressed to any objects in any order, the object that sent the first interaction need not be among them. Instead of notifications we may also use the terms one-way event or signal (e.g. in OOSE). Notifications are used as technical interactions in concurrent systems or as conceptual interactions when modelling the high-level view of the interactions between a system and its users. In the detailed design, such high-level notifications are often replaced by more complex interactions mechanisms consisting of requests.

*Definition 12: A **request** is a two-way interaction where the sending object waits for a return-interaction from the receiving object.*

A request consists of two interactions. The sending object (in connection with requests also called the **client object**) sends a message to the server object. After sending the message, the client object waits until it gets the answer from the server object. The answer

is called the **return interaction**. Before the server object reacts with sending the return interaction to the client object, it may send other requests or notifications to other objects. A request can be considered as consisting of two notifications for which some additional constraints apply. In sequential systems, requests are usually implemented as procedure calls or method calls, in concurrent systems as remote procedure calls. Other methods use the term message instead of the term request (e.g. Fusion and OOSE).

Conceptual and technical interactions

*Definition 13: **Technical events** reflect constructs of the program code, **conceptual events** reflect concepts of the problem domain. A conceptual event that is not identical to a technical event is a **purely conceptual event**. If technical or conceptual events are modelled depends on the focus and the goals of the model.*

Technical interactions

When modelling the events and interactions of services, we can do this on various abstraction levels. One possibility is to choose a technical level. There we reflect the effective program code, and we focus on the effective communication among the objects. When we model on a technical abstraction level, the interactions correspond to software constructs in the final program, e.g. to method calls, to signals or to hooks of an UIMS. We call these interactions **technical interactions** or **technical events**. We assume that on a technical level all communication takes place via interactions, i.e. also events such as time-outs and internal triggers are modelled by interactions. Therefore a technical event is always an interaction. Technical interactions may be notifications or requests. An atomic service is always triggered by a technical interaction.

Conceptual interactions

When modelling events of services we may also choose a higher abstraction level, a conceptual level. There, the focus lies on the correspondence with concepts of the problem domain and not with software constructs. **Conceptual events** or **conceptual interactions** reflect events, messages or information flows from the problem domain[2]. They do not need to correspond to the effective interactions in the program code. Not all conceptual events are modelled as interactions.

During requirements determination we normally model all the events and interactions that trigger the services and that are needed for exchanging information with agents as conceptual events and interactions. We do not care yet if a conceptual interaction corresponds to an effective interaction between software objects in the actual system interface

2. Conceptual events can be compared to the semiotic acts that Graham uses in requirements analysis[Graham95]. A semiotic act is a message in the direction of the actual information flow.

or not. Conceptual events such as time-outs or internal triggers can also only be mentioned in a textual description.

Conceptual interactions can be both notifications and requests. But the use of conceptual requests needs to be well thought through. Normally when modelling an interaction pair between two software objects as a request, we know already that this request will be also a technical request, so we can directly model it as a technical request. When modelling conceptual interactions between users and a system, these interactions are normally notifications, because the system does not only answer to the user that sent the first interaction but to all agents on an equal level. The situation is different when modelling a one-agent view of the system. Then the interactions between this one agent and the system may of course be modelled by conceptual requests.

Purely conceptual interactions

In the course of the software development process, a conceptual event can either be directly transformed into a technical interaction, or it is replaced by a whole interaction mechanism that consists of several technical interactions. We call a conceptual event that does not correspond 1:1 to a technical event a **purely conceptual event**. Purely conceptual events or interactions abstract away complex interaction mechanisms. The direction of a purely conceptual interaction needs not be the same as the direction of the first technical interaction in the interaction mechanism that is abstracted away by this conceptual interaction.

The distinction between conceptual and purely conceptual events is only of interest from a retrospective point of view, and is therefore hardly ever used. Normally we only distinguish between conceptual and technical events, thus emphasising the focus of a model. If the focus of the interaction is on reflecting an effective program construct, then we use a technical interaction. If the focus is on modelling higher level events as this is normally done in requirements analysis or higher level designs where we do not care about how these events are represented in the program, we use conceptual events.

Examples of technical and conceptual interactions

A typical example of the use of conceptual and technical interactions is the modelling of the interactions between a system and a human user. In the interaction diagrams showing the external view of a system, purely conceptual events are used. One conceptual interaction contains all the information flow that can logically be transferred at once from the user to the system (see for example figure 3). When moving from requirements analysis to design, we want to model the actual interface mechanism. Several interactions take place to enter the information of the purely conceptual interaction. Furthermore, it is not the user that sends an interaction to a software object, but a software object that requests inputs from certain interface objects.

Modelling the interactions between a human user and a software system by technical interactions is quite delicate. It can only be done if the user really submits messages in the

form of message name and parameters to the software system, e.g. via a command line interface[3]. Normally we model the interactions as conceptual interactions which are then detailed by technical interactions between the various interface and problem domain objects. Conceptual interactions can also be used when modelling the interactions among atomic software objects, but very often these interactions are modelled straight away on a technical level.

4.2.3 Scenarios

Scenarios as sequences of actions or interactions

*Definition 14: A **scenario instance** is a sequence of **interaction instances** and/or **state changes** of objects. Interactions and state changes are also called **actions**.*

Most often[4] the focus of scenarios is on the interactions, so we can also say that a scenario is a sequence of interaction instances. A diagram modelling a scenario instance contains neither any interaction types nor any optional or alternative interactions or state changes.

The objects involved in a scenario may be systems, subsystems, or atomic objects. There may be arbitrarily many interactions within one scenario instance. In the case of a parallel system, some of the interactions may be executed in parallel. Yet for the sake of simplicity we always use the word “sequence” to denote one specific order of interactions. Furthermore, the description of a scenario may also contain the state changes of the objects and the calculations for the values of interaction parameters.

A scenario is triggered by a **triggering event**. This triggering event may be an interaction or another event such as a time-out or an internal event.

Scenario types

One of the basic concepts of object-oriented modelling is the difference between types and instances. The interaction and scenario instances which can be executed in a simulation or walk-through of a higher level model or at run-time of the final software program are specified by interaction and scenario types.

*Definition 15: A **scenario type** specifies a set of possible scenario instances.*

All potential scenario instances of a system are classified into scenario types. Whereas a scenario instance only shows one concrete path through the scenario and has concrete values for all the interaction parameters, a scenario type shows all possible paths through

3. Even in this case the difficulty remains that when modelling the internal view of the system, this interaction is issued by a command line interface that is part of the software system and not part of the agent user.
4. Very rarely are scenarios modelled that have no interactions. Examples are scenarios modelled on a higher level and triggered by time-outs or other internal event that only cause internal state changes. See also figure 33.

the scenario, all possible interactions and all possible state changes. Yet we do not require that all interactions specified in a scenario type are interaction types, some of them may also be interaction instances. Examples of a scenario instance and a scenario type can be found in figure 7 on page 22.

A **scenario type** specifies:

- the interaction types and interaction instances of the scenario,
- the various possible sequences of the interactions,
- the conditions that determine the order of the interactions,
- calculations of output parameters (parameters of output interactions),
- state changes of the objects involved in the scenario,
- the possible triggering events that may initiate the scenario.

The **possible orders of interactions** which can be defined are: sequence of interactions, optional interactions, repetition of interactions, concurrent or interleaving interactions, alternative interactions.

The connection between scenario types and services

Among the properties of a service mentioned in chapter 4.2.1 are the interactions between the server object and its agents, the interactions among the component objects of the server object, the state changes in the server object, and the triggering events. We describe these aspects with the help of scenario types. For one service we may define several scenario types. These scenario types describe the service on **different abstraction levels**; they must be in a detailing hierarchy, i.e. they are mutually compatible (see chapter 4.2.5.3). Furthermore, each scenario type of a service can be described from an internal and external point of view. The **external view** only shows the interactions between the agents and the server object, as well as state changes of the server object. The **internal view** also shows all the interactions among the component objects of the server object. The scenario types that model the actions of a service must fulfil the following conditions:

- The scope of the scenario type (definition see chapter 2.1.3.3) is determined by the responsibility of the service that is described by the scenario type.
- The information flow of the scenario type (i.e. the sum of all information that is exchanged by the interactions) is determined by the information flow of the service.

As services can be arbitrarily defined, there are no rules as how a scenario type has to look, with the exception of the scenario types for atomic services. There are no rules concerning triggering interactions (one or several triggering interaction types are possible, also other triggering events are possible that are not interactions), no rules concerning the direction of a triggering interaction, no rules concerning which variants belong to

which service, and no rules concerning the end of a service and thus of its scenario types. Instead of having classification criterias (such as e.g. those described in chapter 2.1.3.3) which determine the scope of a scenario type belonging to a service, we provide the possibility of building aggregation and generalisation hierarchies of services (see chapter 4.3), and of detailing scenario types.

In principle, different scenario types which are mutually incompatible are feasible for one and the same service, because normally there are several possible ways to produce the necessary results and to have the same information flow. The different possibilities concern how the necessary exchange of information between the objects is distributed onto various interactions as well as how these interactions follow each other. Yet within one model and on one specific abstraction level, only one scenario type is allowed for a specific service. All other possible scenario types are considered as belonging to other scenario models. During the development process several scenario types on the same abstraction level may be defined for the same service, e.g. by different persons or in order to ameliorate the model. In the integration and consolidation process, these scenario types need to be transformed into other scenario types with the preservation of the responsibility and information flow of the service they describe (see also chapter 4.5).

Using scenarios for things other than specifying services

We call any sequence of interactions between any kind of objects a scenario. Scenarios are not only used for specifying services, but can also be used to describe and to visualise any kind of important mechanisms, design patterns, frameworks or other architectural aspects of a system of interacting objects.

4.2.4 Notation: schemas for objects and services

We document a scenario model by **schemas** and by **diagrams**. The schemas[5] contain all the textual information. The graphical information is modelled by diagrams. In the case studies in chapter 5 we have divided up the textual specification of an object and its services into schemas for the object and for each of its services. Of course, in a hyper-text CASE-tool such a division is not necessary: the specification of an object and its services can be structured in a hierarchy of texts and diagrams (e.g. as outlined below), with links not only along the hierarchy of documents but also across the various elements of the specification. Examples for schemas of objects and services can be found in chapter 5. The syntax and semantic of interaction diagrams will be introduced in the following chapter. For a notation for diagrams showing graphically the various relationships among object types see for instance BON [Walden95]. In the following we give an outline of the information that needs to be recorded for the specification of the objects and their services.

5. We use the term “schema” in analogy to the operation schemas of Fusion.

Objects and services

As a consequence to what we said about the basic object model in chapter 4.1, we assume that the specification of an object type (system, subsystem or atomic object) contains the following information:

- **Name** of the object type, characteristics of the object type (atomic object or subsystem, subtype of), textual description of the object type
- **Services offered**:
  - high-level services offered (will be introduced in chapter 4.3.2.1)
  - elementary services offered (will be introduced in chapter 4.3.2.1)
  - life-cycle of the object (will be discussed in chapter 4.4.2)
- **Services used**, **references** (only if the agents of this object also appear in some model as server objects):
  - list of the services offered by an agent and used by this object
  - references to collaborating objects which need to be reminded
- **Essential states** (will be discussed in chapter 4.4.1)
- **Internal composition** or provisional external data model (see chapter 4.1)

Of course, not every element listed above is compulsory. The grade of detail varies with the kind of object (system or atomic object), with the level of abstraction and the goal of the model.

Internal composition

The internal composition shows the internal structure of an object, i.e. all the objects it consists of. In the case of a system or subsystem we specify:

- **Component objects**: List of all the component objects, indicating also the cardinality of the component objects.
- **Internal scenarios of all the services**: Interaction diagrams which show the internal view of the elementary services the system offers.

In the case of an atomic object, the internal composition consists only of the internal state variables[6] (also called attributes) and private services (services not offered to other objects). The internal view is therefore reduced to a list of all the state variables and all the private services. Interaction diagrams showing the internal view of atomic services could be drawn, yet they are not of much interest.

6. If these attributes are again considered as being objects or not, does not have any influence on the specification of services. For a discussion of this subject see [Eckert95b].

References

The references reflect the relationships or associations to other objects on the same abstraction level, i.e. to the collaborating objects. Component objects are not listed here as they are not external agents of the server object but internal to the server object, they appear in the internal composition of the server object. We model the associations among collaborating objects as unidirectional references and not as bidirectional relationships in order to simplify the composition of objects into subsystems. How these references are implemented is of no concern here. The specification of a reference indicates:

- name of the reference
- name of the referenced object type
- cardinalities, qualifiers, rules and conditions of the reference (optional)

Offered services

The specification of the services offered by the server object includes[7]:

- **Name**: Name of the service. In the case of an atomic service, the name of the service is equal to the name of the triggering interaction.
- **Summary**: A one-sentence summary of the responsibility of the service.
- **Inheritance** (in the case of extended or specialised services): Name of the service this service is an extension or specialisation of; conditions under which this service is chosen.
- **Description**: Textual description of what is expected from this service (outputs, state changes, conditions for the reactions). This description may be imperative or declarative. Depending on the goal of the modelling effort and depending on the abstraction level, the description of the service may be very informal and vague, a precise semiformal specification as in the operation schemas of Fusion may be used, or even a more formal approach may be chosen.
- **External scenario**: Interaction diagrams showing the external view of the scenario types of the service. In the case of an atomic service, we only give its signature.

4.2.5 Notation: interaction diagrams

In this chapter we propose a notation for modelling services and scenario types. The concepts and notation of SEAM have been influenced by various kinds of software develop-

7. Especially when specifying system services in order to model the external view of a whole software system and to model the requirements of this system, additional information elements may be added. These may be non-functional requirements such as performance, or QA-information such as author, version and date.

ment projects, including a process control system in the telecommunication, several information systems and, most recently, student projects implementing games as distributed software systems. The examples[8] shown in this chapter are taken from the two case studies of chapter 5.

4.2.5.1 Specifying interactions

Notifications are specified in the following way:

receiving_object :: interaction_name (parameter_name1 : parameter_type1,
parameter_name2 : parameter_type2, ...)

The specification of a request contains also the parameters of the return interaction:

receiving_object :: interaction_name (input_parameters) : (output_parameters)

The name of the receiving object may be omitted if it is clear from the context which is the receiving object. This is for instance the case when labelling interactions in an interaction diagram.

Interaction parameters

For the interaction parameters it is not compulsory to give both, the name of the object instance and the name of the object type. In higher level interaction diagrams, most often only the name of the object instance is specified. Either the type of the instance can be deducted if it is assumed that the name of the instance is equal apart from capitalization to the name of the type. Or the type is not yet known and is left open for later specification. Another possibility is to give only the object type; this must then be interpreted as "an object of this type". So instead of *invoice (delivery_note : Delivery_Note)* we can also use the following short-cuts[9]:

invoice (delivery_note)
invoice (Delivery_Note)

Sometimes, an interaction parameter is a list or set of several instances of the same type. This may be indicated by either curled brackets or text. Examples:

order ({item})
order (list of items)

8. There are two severe limitations for providing good examples of the notation in this thesis:
   - The notation described here is primarily targeted at being used with diagramming tools and CASE-tools. For nowadays computer systems running such tools, 17", 19" and larger monitors are state of the art. When diagrams are printed out at all, this normally happens in a A4 landscape format. Yet in this thesis the diagrams had to fit on A5 portrait format. Nobody would draw a gantt chart for project management on a little piece of paper. In the same manner, also notations for scenarios need not to be targeted at small pieces of paper.
   - SEAM has been developed having larger projects in mind. But diagrams from such projects could not be used as examples, they are too complex and too large to be included in a thesis.
9. We thus use a similar syntax as Fusion where also either only the type, only the instance name (TypedName) or both can be specified.

or for a set of tuples:

info_building ({drum_id, drum_type})

If not all parameters are listed, dots are used. Dots indicate that either a parameter is intentionally hidden (to simplify the appearance of a diagram when for the actual purpose of the diagram this interaction parameter is of no concern), or that some parameters are not yet specified. Example:

info_building ({drum_id, ...})

The name of the interaction does not represent the motive of the interaction, but the result desired from the object (e.g. *get_balance*, *print_document*, *do_weekly_cores*), or the information transferred to the object (e.g. *alarm*, *deadline_expired*). For conceptual interactions there are no further restrictions for the interaction name. Yet if the interaction is a technical event, then the interaction name is equal to the name of the atomic service triggered by this event, and the specification of the interaction is equal to the signature of the atomic service. The **signature** of an atomic service indicates the name of the server object, the name of the service, the names of the parameters of the triggering interaction, and the names of the parameters of the return interaction if it is a request.

When the object model of the system is already known (i.e. the internal decomposition of the system into subsystems, atomic objects and attributes), then the types of the interaction parameters are given by this object model. They correspond to the types of attributes, atomic objects or even subsystems, or to any other fundamental data types defined and used in the system. If the decomposition of the system has not yet been determined, then the possible types of the interaction parameters are given by the provisional external data model (see also chapter 4.1.2.3).

4.2.5.2 Basic notations for interaction diagrams

The main aspects that we want to show in a diagram specifying scenario types of services are the following (see also figure 28):

- objects involved,
- interactions involved (interaction names and interaction parameters),
- possible sequences of the interactions (control flow with conditions),
- begin and end of services.

We have chosen interaction diagrams with time-line notation (for a comparison between interaction diagrams as two-dimensional object diagrams or as time-line diagrams see chapter 2.1.4.3). The vertical bars are the objects (atomic objects or subsystems). The interactions are shown by horizontal arrows. They are annotated by the name of the interaction and optionally by the parameters of the interaction. To the left-hand side we have a pseudo-code annotation which specifies the possible orders of the interactions and

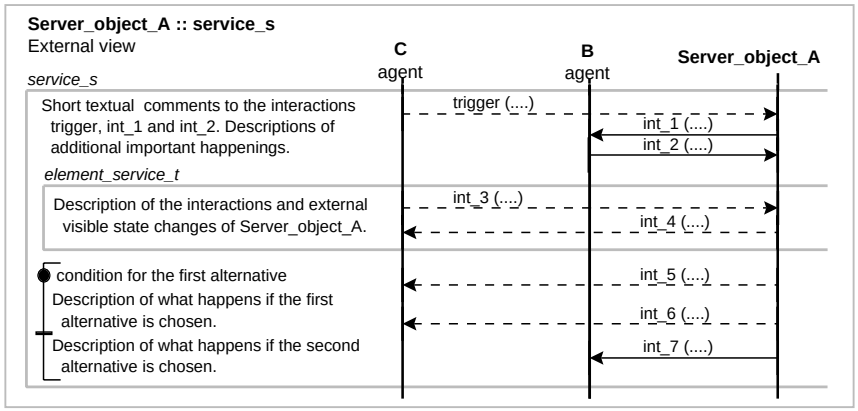


Figure 28: Interaction diagram for modelling scenario types

of other actions, and optionally the conditions under which a specific order is chosen. Furthermore, we show also the begin and end of services.

Pseudo-code

In the pseudo-code annotation we use brackets[10] to show the flow of control and plain text to describe actions and conditions. The following kinds of control flow can be expressed by the brackets defined in figure 29:

- **sequence** of actions
- **alternative** actions and **optional** actions
- **repetition** of actions
- **concurrent** or **interleaving** actions
- **start and end** of services

The default control flow is as sequence of actions, therefore the bracket for sequence need not be used. In the case of repeating or alternating actions, we can specify also the conditions for the repetition or the alternation. The specification of these conditions is optional. As a rule of thumb, we specify the conditions whenever the object that determines the flow of control is an internal software object (see for instance the repetitions in figure 104). Yet when the object is an agent of the whole system, for instance a person or another computer system, then we omit the conditions, because we cannot or do not

10. Using brackets to visualize the control flow in pseudo-code notations is not a new technique, see for instance [Martin85].

want to model the reasons why these agents take certain decisions (see for instance the alternation in figure 104).

The text in the pseudo-code gives additional information to the actions and interactions of the scenario. How detailed these actions are described depends on what for the model is made, and by whom it will be read.

The width of the pseudo-code annotation may vary from diagram to diagram. When modelling the external view of a service, we have lesser objects and may want more space for the pseudo-code. When modelling the internal view, the pseudo-code may be reduced to the control flow and only a few comments, needing less space. There are no fix rules in which situations which grade of detail is the best. An important criterion is that the targeted audience (which for these kinds of diagrams are normally humans with a certain amount of intelligence, knowledge about the problem domain and common sense) can understand it without overloading the diagram with non-essential information.

Objects

The vertical bars, which represent objects, are labelled at the top. The label contains the name of the object, and, in the case the object plays the role of the agent in that scenario type, also the annotation “agent”. Additional annotations such as “subsystem”, “group” or listing the objects that make up a group, are optional. The name of the object may be either the name of the object type (e.g. *Drum*), the name of the object instance and name of the object type (e.g. *drum1 : Drum*), or only the name of the object instance if the type is not yet known or can be derived from the instance name. When only the object type is specified, this must be interpreted as “an object of type ...”. We allow this short-cut because all the object instances represented in an object diagram are of a prototypical nature, being examples of instances which will be instantiated at simulation- or run-time of the system. Names of object instances do not have any other significance than distinguishing the objects from each other in the interaction diagram. It is advantageous if the same approach for labelling the object lines is used in all the interaction diagrams of a project.

If several object instances of the same object type appear in an interaction diagram, and if they receive the same messages at the same time (either concurrently or sequentially), then the bars representing these objects can be collapsed into one single bar representing a collection of objects. The collection is labelled by the type and/or name of the object instances, being put into curved brackets. *{Drum}, {drum}, {drum:Drum}* all signify that several objects of type *Drum* receive the same messages in the context of this interaction diagram.

Interactions

Interactions are modelled with arrows. We distinguish between the following kinds of arrows (see figure 29):

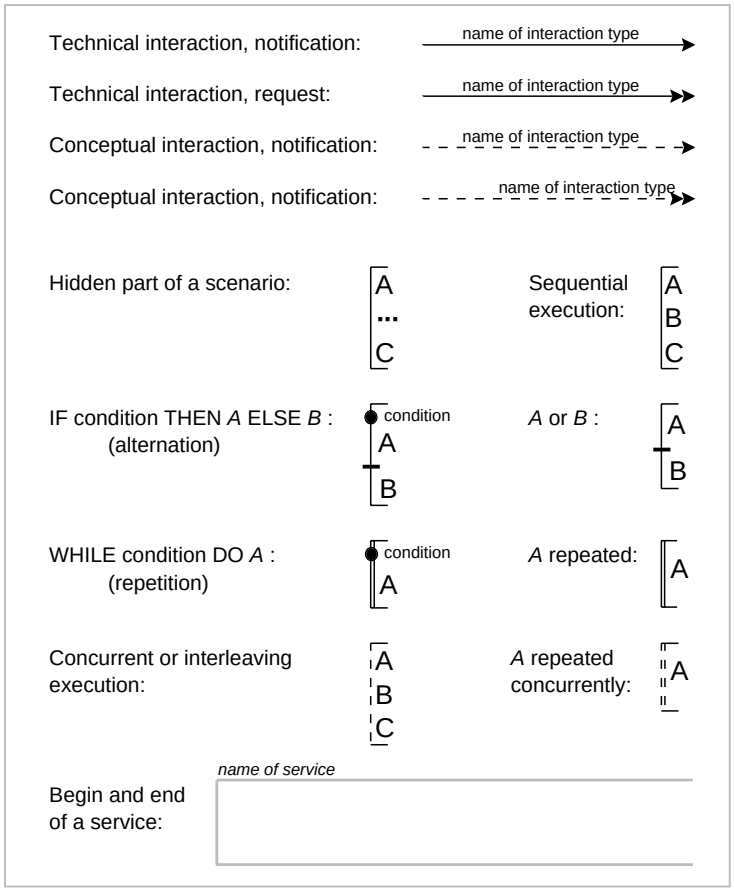


Figure 29: Notations for interaction diagrams

- double-headed arrow: **request**
- single-headed arrow: **notification**
- solid arrow: **technical** interaction
- dashed arrow: **conceptual**[11] interaction

The arrows are labelled either with the whole specification of the interaction or only with the name of the interaction. When only the name is used (e.g. for reasons of space), then the whole specification of the interaction may be added below the interaction diagram

(see for instance figures 99 and 86). If the goal of the diagram is to give a high-level view or to be used in a presentation, then the specification of the interactions can be omitted.

4.2.5.3 Modelling scenario types of services

External view and internal view

When using interaction diagrams to model scenario types of services, we have to distinguish between the internal and the external view of a scenario type. Figure 30 shows the external view of the service *ECO-System::enter_manifest*, figure 31 its internal view. Figures 74, 77, 78 and 79 show the external and the internal views of the services *Mail_Order_Firm::order_credit_card, Mail_Order_Firm::order_advance_payment* and *Mail_Order_-Firm::order_cash*. These examples are all services offered by a system. But we may also model the external view of a service offered by an atomic objects, as this is done in figure 32 for the atomic service *overview_drums_for_buildings* offered by the atomic object *Building_administrator*. The interaction diagram showing the internal view of a scenario type is just an expansion of the interaction diagram showing only its external view, i.e. all the interactions between the agents and the server object remain the same.

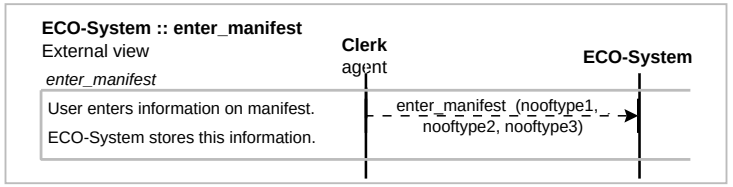


Figure 30: External view of a system service

Services triggered by time-outs

Most often, services are triggered by an interaction from another object. But there exist also cases where a service is triggered by a time-out or by an interaction of the server object with itself (either caused by some conditions or by another service). In the case of time-outs we have two variants to represent it in the interaction diagram:

- The time-out is modelled by an interaction of the server object to itself (example see figure 34) or from a clock-agent to the . This variant is chosen when modelling on a lower level and using technical events.

11. When we model on a conceptual abstraction level, we model the interactions as conceptual interactions, irrespective of them being purely conceptual or technical interactions. When we focus on the effective communication mechanisms (method calls) among software objects, we model the interactions as technical interactions. Therefore we only have two symbols, one for conceptual and one for technical interactions. This has also the advantage that any interaction can be modelled as a conceptual interaction if it is not essential for the goal of this interaction diagram or not known yet if the interaction is a technical one or not.

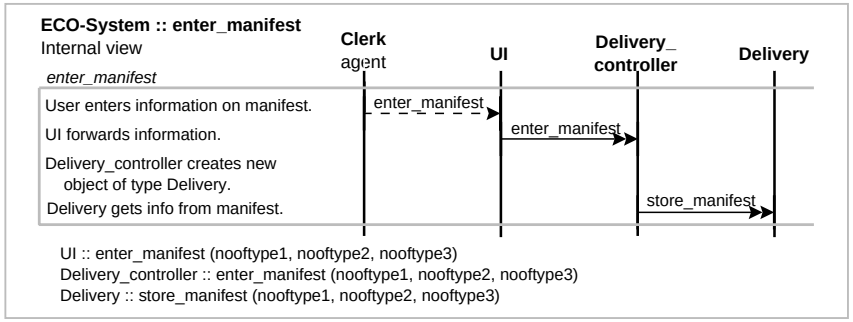


Figure 31: Internal view of a system service

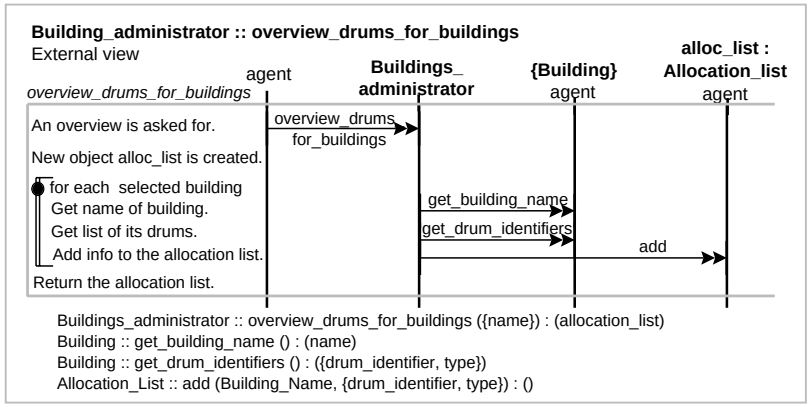


Figure 32: External view of an atomic service

- The time-out is not modelled by an interaction but only mentioned in the pseudo-code. This is especially helpful when modelling the external view of higher level system services. An example of this can be found in figure 75. In the case of a service where no results need to be send to any agents, this can result in an interaction diagram with no interactions at all (see figure 33).

Services with several possible triggering events

For non-atomic services we explicitly allowed having several events playing the role of the triggering event. These may be modelled as interactions, or may only be mentioned in the pseudo-code. Allowing several triggering events is very helpful for modelling high-level services which can be triggered by an external agent, by a time-out and by another service. Figure 34 models the service *Order_System::make_orders_for_suppliers* with two triggering events, both represented as interactions (compare also with the interaction diagram for the service *Mail_Order_Firm::make_orders_for_suppliers* in figure 75).

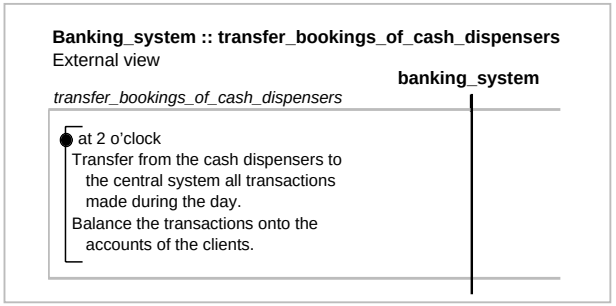


Figure 33: Service triggered by time-out

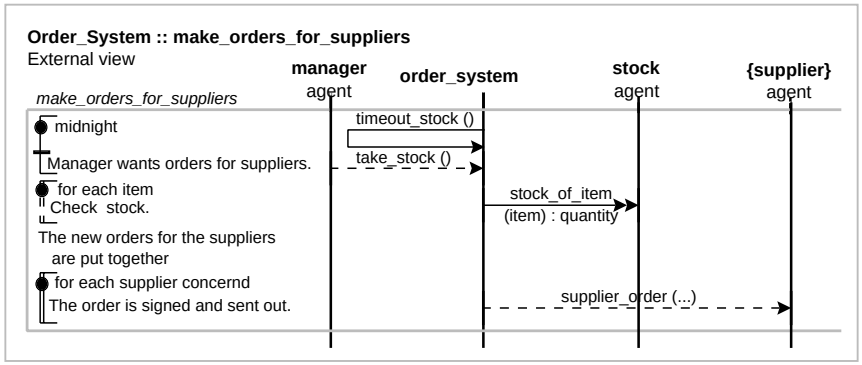


Figure 34: Service triggered by two different interaction types

Services triggered by the server object

Not all services are triggered by an external agent. There are services triggered by a time-out within the server system. There are even services that are triggered by an interaction from a system to an external agent. A typical example of this are games. There the driving force for progressing lies with the system, not with the user. It is the system which determines what comes next, it is the system which asks the user for its decisions and reactions. Figure 35 gives an example. Of course, any scenario model containing services triggered by the system can be remodelled into a scenario model containing only services triggered by the user. But the start and end of these services neither reflect how the users perceive the system nor correspond to the effective design and implementation[12].

12. For three successive years our students have modelled and implemented games. We used the Fusion method, so only services triggered by external agents were allowed. Though we always found a way to force the scenario model into this form, we were not always happy with it.

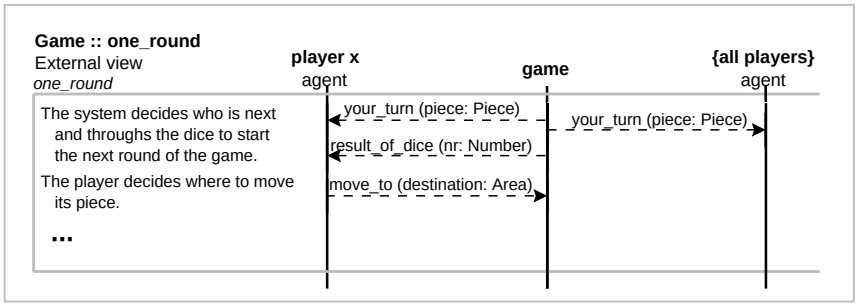


Figure 35: Service triggered by an interaction from the server object to an agent

4.2.5.4 Grouping and hiding objects and interactions

Group of objects

In 4.1.2.2 we did not only introduce white-box and black-box subsystems but also groups of objects. In contrast to subsystems, groups have no significance beyond the interaction diagram they are used in. They are not reflected in the final software structure and need not appear in other interaction diagrams. Their only purpose is to group objects in one or several diagrams, either because the objects that are part of the group are not yet known, or because the interaction diagram is quite complex and we want to decompose it. An object instance may be part of several object groups. These object groups may be overlapping.

In the interaction diagrams, groups are treated like other objects. The group is labelled with the name of the group. The label is further annotated by a list of all the objects that are part of the group, or by the qualifier “group”. Collapsing objects into a group allows us to simplify diagrams for presentation (compare figure 104 with the first diagram in figure 36) or to decompose a complex internal view into several diagrams (example see figure 36).

A special case of collapsing objects into one group is the collapsing of all agents into one agent. This is often done in the internal view where the main focus is on the internal mechanisms and not on the different agents. When all the agents are humans, then the group of agents may be named “user” (see first diagram in figure 36), otherwise the name is just omitted (see second and third diagram in figure 36).

Hiding objects and interactions

The need to hide objects or interactions may arise in the following situations:

- **Viewing**: When viewing an interaction diagram one might desire to focus only on those aspects that are of interest at the moment. This may be the case when viewing models on-line, when including diagrams into a

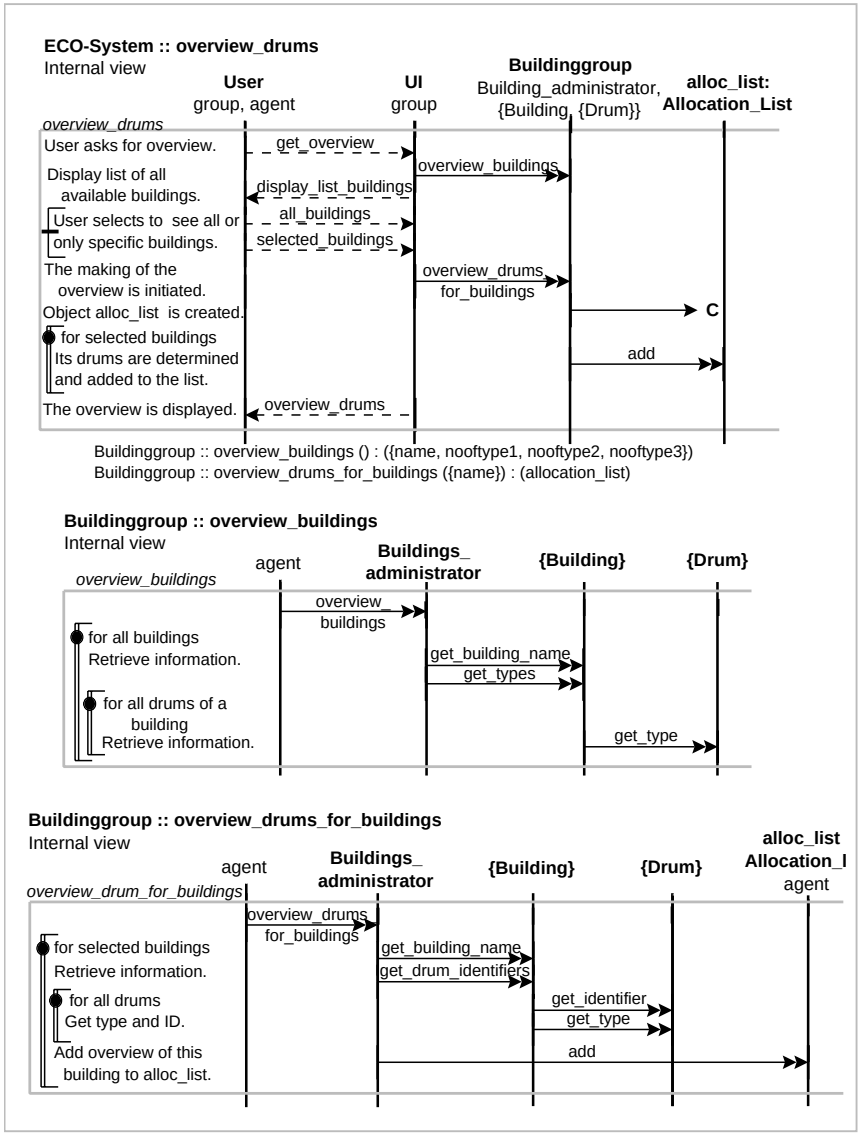


Figure 36: Object groups in interaction diagrams

documentation, or when printing out diagrams for presentations and discussions.

- **Developing**: When drawing a diagram it may happen that some details are not yet known, but nevertheless we would like to draw the rest of the diagram. So we need a replacement which shows that some objects or interactions are still missing.

We can hide an object by omitting it from the diagram. The interactions sent or received by this object are not omitted but replaced by either indirect interactions of by one-sided interactions (example see figure 37, where the objects *Buildings_administrator* and *{Drum}* have been omitted). **One-sided interactions** just show that an object receives or sends an interaction from or to a hidden object. An **indirect interaction** is a replacement for a sequence of interactions[13]. The two objects connected by an indirect interaction do not interact directly with each other but via another now hidden object. If we have for instance an interaction from an object *A* to an object *B*, followed by an interaction from object *B* to an object *C* and if we want to hide object *B*, we can do this by replacing the two interactions by an indirect interaction from object *A* to object *C*. This interaction has the name and the parameters of the interaction from object *B* to object *C*. In contrast to conceptual interactions, indirect interactions are not used to abstract away interaction mechanisms or to show higher level views. They are only used to enable the (temporarily and often automatic) hiding of an object and appear mainly in low-level interaction diagrams.

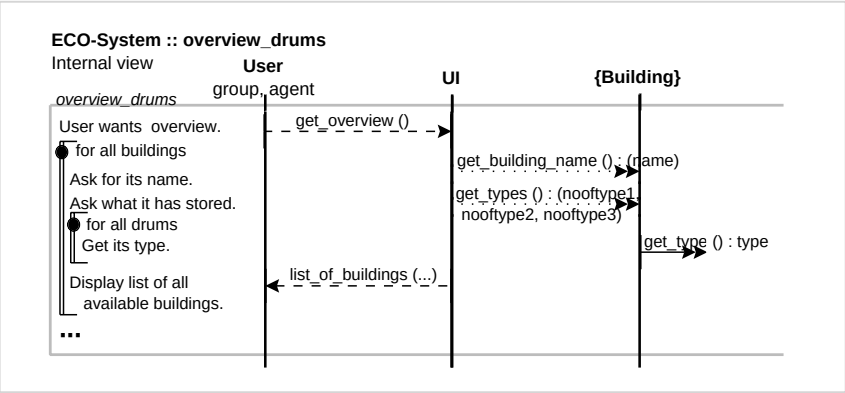


Figure 37: Indirect interactions

13. For example when developing distributed systems, we may like to hide the objects of the communication mechanism. The interactions between objects of different systems may then be drawn as indirect interactions, by dotted arrows.

We can also hide interactions and their accompanying pseudo-code. This is done by omitting the interaction arrows and replacing the pseudo-code by "**...**". Examples are found in figures 37, 99 and 97.

4.2.5.5 Further notations for interaction diagrams

In this section we introduce further modelling elements sometimes helpful for modelling scenario types. These are techniques for visualizing the flow of control, the lifetime of objects or the return events of requests, and for representing time-line diagrams as two-dimensional object diagrams. These modelling elements may be of help in certain circumstances, but they are not essential when using the scenario modelling technique SEAM and most of them are not used in the case studies in chapter 5.

One-agent views

In SEAM we have adopted an all-agent approach. Therefore the interaction diagrams of the external view show all the agents of the server object. Yet we have the possibility of using interaction diagrams also for describing one-agent views which show only the interactions between one agent and the server object. One-agent views are normally only made for higher level services of the whole system. Either they are derived from an all-agent view and are used as another way of presenting the scenario model in walk-throughs and in discussions with future users and domain experts. Or requirements elicitation is started with diagrams which only show one-agent views. For each potential agent of the system a one-agent view of all the services concerned is made - thus modelling the different viewpoints of the system (for instance in a telephone system the viewpoint of the caller and the viewpoint of the callee). In a second step these one-agent views are then integrated into all-agent views ([Hsia94] presents such an approach for determining requirements by using scenario trees and regular grammars, see chapter A.9.2). An example of a one-agent view can be found in figure 38 (compare with figure 93).

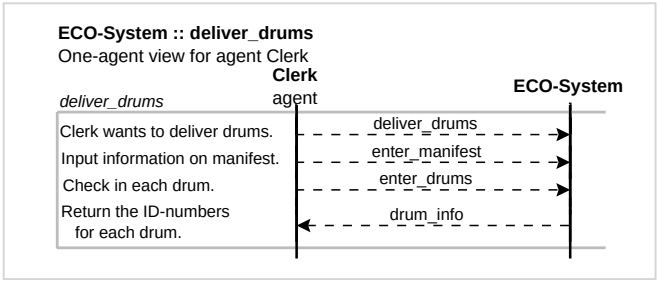


Figure 38: One-agent view for a system service

One-agent views could also be drawn for atomic requests (example see figure 39). In contrast to the all-agent view, the one-agent view does not show any requests initiated

by the server object. Diagrams showing one-agent views of atomic objects do not give any additional information compared to the schema or the signature of the atomic service, so they are hardly ever drawn.

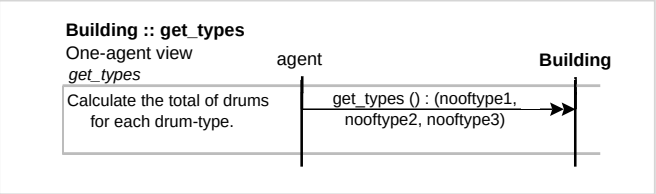


Figure 39: One-agent view for an atomic service

Control flow

In sequential systems it may be of interest to visualize how the control is passed from object to object. In time-line interaction diagrams the flow of control can be highlighted by using thicker bars for the object during the invocation of a service offered by this object. Figure 40 shows an example of SEAM, figure 111 shows an example of OOSE.

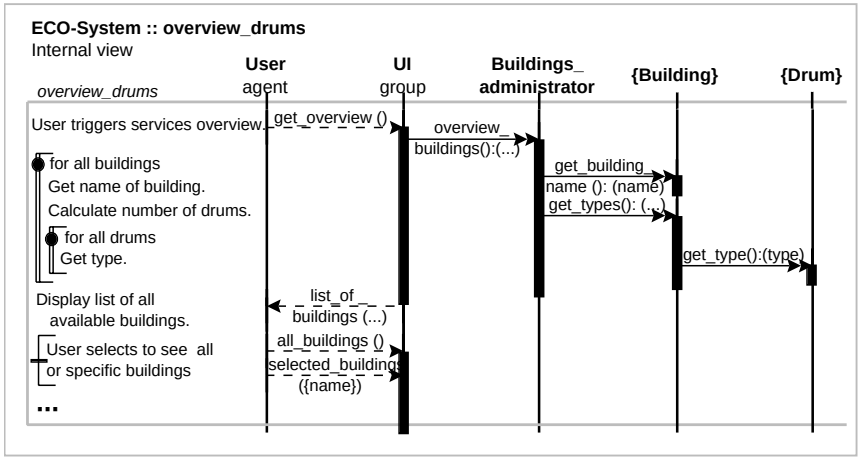


Figure 40: Control bars

Explicit return interactions for requests

In the basic notation for interaction diagrams the return interactions of requests are not explicitly drawn. In a sequential system it is implicitly clear when the return interactions take place. Yet if we want we can explicitly visualize the returns from the requests by using grey arrows for the returns as shown in figure 41. Because we always specify the

return parameters of a request with the arrow for the request, the grey arrows for return interactions do not have any labels.

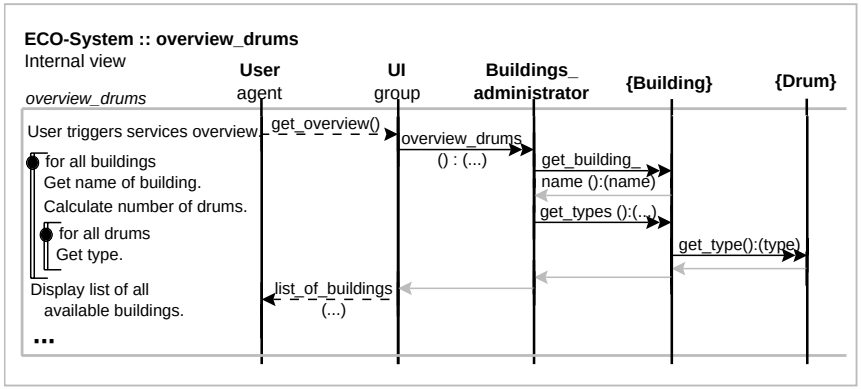


Figure 41: Explicit return interactions

Creation of objects

So far we did not show in the graphical part of the interaction diagrams when and by whom an object is created (see e.g. figure 31). The creation is only mentioned in the text of the pseudo-code annotation. But who creates an object can be easily shown graphically by a special arrow (see e.g. figure 104 or 42). In contrast to all other arrows, this arrow is not an interaction to the new object - the object to be created does not exist yet. The creation is either handled by the sending object itself or it is a message to the class of the object that needs to be created. If for the initialisation of the object additional information is necessary, this information transfer is modelled by a normal interaction.

[Mössenböck96] uses also special arrows to show who initiates the creation of an object. [Coleman94] uses in the interaction graphs create messages, yet there a create message may also have parameters and it initialises the new object. The message is sent to the object that is to be created - a mix that often causes confusion.

Life-time of objects

The life-time of an object can be denoted by having a bar instead of a line for the object during the life-time of the object. In order that no conflict arises with the bars for the control flow, an outlined bar can be used for the lifetime of the object, and a solid bar for the control flow[14]. An example of an interaction diagram which shows also the lifetime of the objects is found in figure 42.

14. [Mössenböck96] uses also bars to show the object life-time.

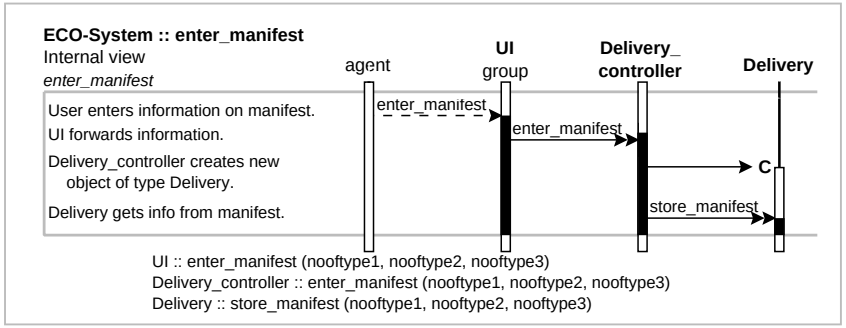


Figure 42: Object creation and lifetime

Alternative diagramming techniques

We have chosen to diagram scenarios by time-line interaction diagrams which are annotated to the left with pseudo-code. Here we show two alternative notations. All three notations are equivalent concerning the content they show. Depending on the tools available and on other project specific restrictions, one or the other of these notations may be favoured.

- Instead of using a time-line notation we may use a two-dimensional object notation as shown in figure 43 (the dashed object symbol represents a collection of objects and is equivalent to *{Building}*. The graphical part shows the objects and their interactions, but no services. The pseudo-code including the service brackets is moved to the foot of the diagram. The link between the pseudo-code and the interactions in the graphical part is made by numbers. Yet the numbers do not denote any order, the order is only expressed in the pseudo-code.[15]
  Instead of using brackets to show the flow of control and the start and end of services, we can also use structured English, enhanced by statements for the start and end of services (e.g. *START SERVICE xy, END SERVICE xy*).
- Also when using a time-line notation we may move the pseudo-code annotations to the foot of the diagram. All interaction arrows are numbered, these numbers are referenced in the pseudo-code. The brackets showing start and end of services appear twice, once in the pseudo-code and once in the graphical part.

Instead of using the time-line notation for all interaction diagrams, it were also feasible to use time-line interaction diagrams on the higher abstraction levels, and two-dimensional object diagrams for the internal view of scenario types of subsystems, i.e. on the

15. This is in contrast to the sequence numbers of Fusion (see e.g. figure 117), where the numbers are used to some degree to show sequences, alternations and repetitions, but cannot express all possible orders which may appear in a scenario.

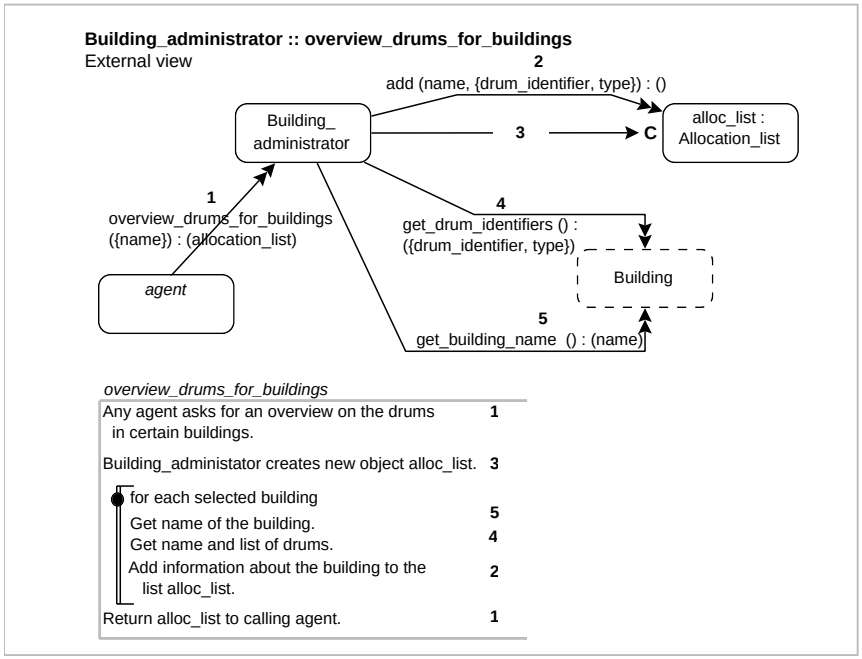


Figure 43: Two-dimensional object diagram

level of atomic objects. Another possibility would be to provide an automatic transition between the two representations in a CASE-tool.

4.2.5.6 Why interaction diagrams?

We have chosen to model the global behaviour by interaction diagrams and to represent interaction diagrams by time-line diagrams for the following reasons:

- Interaction diagrams in a time-line notation can be used on any abstraction level, but they are especially powerful on higher abstraction levels.
- They can easily show internal as well as external views of scenario types.
- They are easily understood by everybody. This is of great help not only in requirements elicitation and system design where the communication with any kind of potential users is of great importance (user centred design), but also in all those development teams where not everybody has the flair to master more complex notations.
- They really visualize the global behaviour of a system.

- We use them to specify the scenario types of the services, but they can also be used to visualise any other mechanisms in a system.
- In order to cope with the potential explosion of the size of diagrams, we have already introduced the decomposition of diagrams along object groups and zooming techniques like hiding objects or interactions, and we will further introduce the concepts of hierarchies of services.

One of the main arguments against interaction diagrams in a time-line notation and in favour of other notations for interaction diagrams or of not diagramming interactions at all is the complexity and size of diagrams (see e.g. the arguments in [Walden95, page 110]). This is true when using only a simple kind of interaction diagram. But by decomposing the object-oriented system into subsystems, by supporting subsystems and object groups in the interaction diagrams, and by having hierarchies of services, this is no longer a problem. The most predominant reasons for promoting or declining time-line interaction diagrams are probably personal preferences. These are often due to the personal background and positive or negative experiences which are also highly influenced by factors not connected with the advantages or disadvantages of a certain notation. One important factor is how well a certain notation is supported by a diagramming or CASE-tool, any notation can be made unusable by a bad support for developing, viewing and printing diagrams.

4.2.6 Detailing scenario types

Whenever a scenario type has one or several purely conceptual interactions, these interactions can be detailed into a more detailed interaction mechanism. This detailing process can be continued until we obtain a scenario type having only technical interactions. Also, additional interactions can be introduced without replacing a conceptual interaction when some parts of the scenario have only been described in the pseudo-code or have been abstracted away altogether. Detailing[16] allows us to describe a service on several abstraction levels, from a level with very abstract interactions down to a level with technical interactions (compare e.g. figure 94 with figure 103 and figure 81 with figure 86). This is especially helpful when modelling the interactions between a user and the system. On a higher abstraction level all information is entered at once. But on a lower level, one element after the other is entered, e.g. to make error checking after each input possible[17]. We thus allow that a service is described and specified by more than one scenario type. But we require that they are compatible with each other. Detailing is like zooming in and out on a service. Yet in contrast to automatable techniques such as hiding interactions (see chapter 4.2.5.4), detailing and abstracting scenario types cannot be automated. The conceptual interactions and pseudo-code annotation of the more abstract views must be created explicitly by a developer.

Definition 16: *A scenario type is **compatible** to another scenario type if it can be derived from this scenario type by detailing or abstracting.*

Definition 17: ***Detailing** a scenario type means that one conceptual interaction or a sequence of conceptual interactions are replaced by a more detailed interaction mechanism (consisting of technical or conceptual interactions) or that additional interactions are introduced.*

Definition 18: ***Abstracting** a scenario type means that certain interactions are left away or that a sequence of lower level interactions (technical or conceptual interactions) are replaced by some more abstract conceptual interactions.*

Properties of detailing and abstracting scenario types

It is quite evident that two scenario types having only technical interactions are only compatible if they have the same interactions and the same order of interactions. For scenario types having conceptual interactions it is more difficult to determine if they are compatible or not. This depends on their content, because when detailing a scenario type a whole sequence of conceptual interactions can be replaced by another sequence of interactions. Thus no syntactical rule can be given.[18]

Often the processes of detailing scenario types and the process of defining element services go hand in hand (see chapter 4.3.2 on aggregation hierarchies). The interactions of the service are detailed into more complex interaction mechanisms, and these are grouped together into element services (see e.g. figures 93 and 95).

When abstracting as far as is possible all the interactions of a service, we arrive at conceptual interactions which are very close to data flows (but are not data flows). Nearly all the information concerning the concrete order and how the information is exchanged has been abstracted away. What remains is the information flow of the service (see e.g. figure 94). In the case of services triggered by a time-out or by the server object itself and which do not have any information flow to collaborating objects, the resulting scenario type has no longer any interactions (see e.g. figure 33).

16. [Alvarez95] also supports the evolution of use cases (as reasons he mentions the better understanding of the problem domain and the changing viewpoint and modelling needs of the various stages of analysis and design). He offers the notion of having many versions of one use case. All these versions are documented by a hypermedia tool. In contrast to our approach, a tree of versions is supported, and the reasons for abandoned branches are documented. Also [Armour95] provides several levels of use cases, see A.9.8.

17. This is one of the major difficulties we encountered by using Fusion in students project: the system operations defined in the analysis were not technical but conceptual interactions. Trying to use these input events as messages to controller objects lead to very nasty designs. A user friendly error checking of user inputs became difficult or impossible.

18. In a previous version of our enhanced scenario modelling technique we allowed only the replacement of one conceptual interaction when detailing scenario types. This would have the advantage that:
    - The compatibility of scenario types was well defined (two scenario types are compatible if they can be derived from each other by replacing one conceptual interaction by a sequence of interactions).
    - In a CASE-tool each conceptual interaction could be expanded into a more detailed scenario.

    Though theoretically very nice, the application in several case studies showed that such a compatible hierarchy of scenario types could only be achieved in a reverse engineering approach, but not in forward engineering. There, often two or more conceptual interactions need to be replaced by a totally different interaction mechanism to get to the next lower abstraction level (see e.g. figures 93 and 95).

Documenting several scenario types for one service

In the documentation of the scenario model, the result of having one or several compatible scenario types for a service is a set of interaction diagrams, one for each abstraction level. Examples are the two interaction diagrams in figures 94 and 103 showing the external view of the service *ECO-System::get_status*, the two scenario types for the service *starting_phone_call* in figure 50 where the conceptual interaction *start_phone_call* is detailed into several technical interactions, or the diagrams in figures 44, 45 and 46.

When displaying interaction diagrams on a CASE-tool, it may be helpful to indicate by bars to the left-hand side of the bracket code, which parts of the current diagram can be detailed into a more detailed view, and for which parts there exists a more abstract view (preferably by using different colours for the bars). These bars may encompass one or several interactions for detailing and several interactions for abstracting, they may even cover a whole service. Such bars act like hyper-text links to further interaction diagrams of the same service. At the left side of figure 45, the left bar denotes that for the whole scenario type there exists a more abstract version in this scenario model - this more abstract scenario type is shown in figure 44. The right bar denotes that within this scenario model there exists a more detailed scenario type (figure 46) that models this part into more detail.

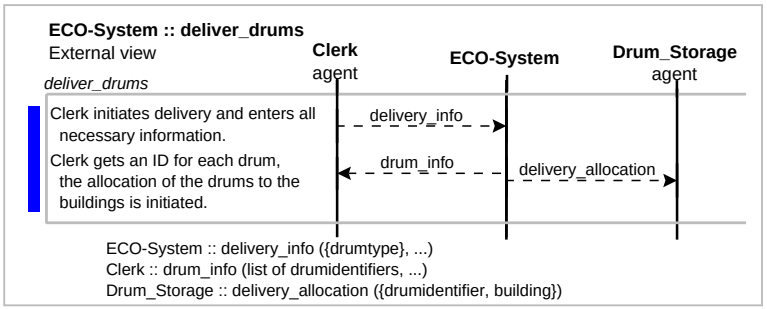


Figure 44: Detailing scenario types: an example, part 1

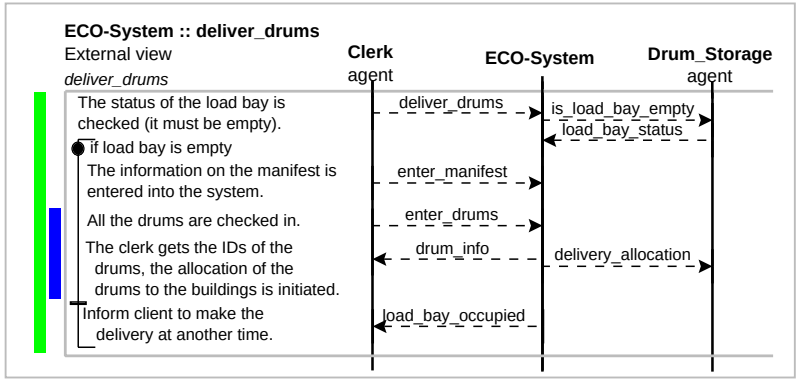


Figure 45: Detailing scenario types: an example, part 2

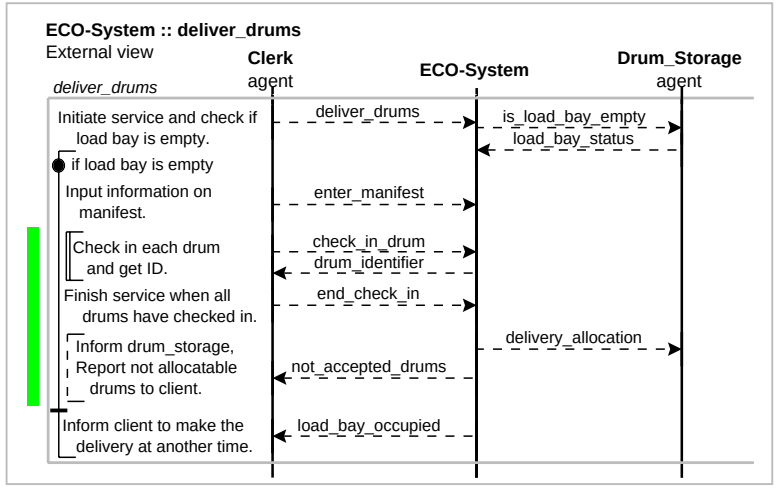


Figure 46: Detailing scenario types: an example, part 3

4.3 Hierarchies of services

Possible hierarchies

Inheritance, aggregation and composition[1] hierarchies are well known from objects. We adopt these terms for the various hierarchies of services, yet with slight differences in the semantic. Within the services of the same object we can build inheritance and aggregation hierarchies. Inheritance hierarchies show specialisation as well as extension relationships between services. We use the term inheritance hierarchy because specialisation and extension of services are the counterparts to the two possible uses of inheritance for object types: specialising an abstract object type into several concrete object types by implementing the abstract methods, extending an existing object type definition by further methods in order to treat additional cases. Aggregation hierarchies are used to group together services that follow each other. Aggregation does not imply any information hiding or encapsulation, it just allows us to consider and handle as one unit several services that are associated with one another concerning their order in the life-cycle of their server object (compare also with the definition of the term aggregate in [Webster]: “a mass or body of units or parts somewhat loosely associated with one another”).

Aggregation and inheritance hierarchies are especially helpful when defining the services of the system as a whole, where we often start off with very high-level services. These are specialised and broken up into lower level services, until elementary system services are reached. Having these hierarchies prevents us from ending up with a huge unstructured list of system services, and frees us from the need to find immediately the correct abstraction level.

Another hierarchy, the composition hierarchy, results from the decomposition of subsystems into objects. The services of component objects are composed into the services offered by the subsystem. Composition hierarchies imply information hiding and encapsulation, and always affect objects as well as services. The composite service is a compound formed by a union of the component services (see also the definition for the term compound in [Webster]: “something formed by a union of elements or parts”). Composition is needed when transitioning from the external specification of the desired system to its internal design. But it is also used whenever subsystems are modelled.

1. Originally, we called the aggregation of services aggregation in time (because several short services are assembled into a longer service) or vertical aggregation (hinting at the vertical axes of time in the interaction diagrams), and the composition of services aggregation in space (because several services of different objects or subsystems are assembled into one service) or horizontal aggregation (hinting at the horizontal axes showing the various objects of the interaction diagrams). As this terminology did not satisfy, we finally settled with aggregation and composition.

4.3.1 Composition hierarchies

Definition and properties of the composition of services

We have based our modelling technique SEAM on the paradigm of systems of interacting objects that can again be systems of interacting objects. The result is a hierarchy of objects, with the system as a whole being decomposed over several levels into subsystems and finally into atomic objects. Just as in this hierarchy component objects are composed into systems, so also the component services of these component objects are composed into the services of the system.

*Definition 19: The **composition of services** composes services of component objects into a service of the system these component objects are part of.*

Any **internal view** of a service shows the composition of the service of its component services. Figure 47 contains an interaction diagram showing the internal view of the system service *ECO-System::enter_manifest*, which is composed of the component services *UI::manifest*, *Delivery_controller::enter_manifest* and *Delivery::store_manifest*. In contrast to figure 31, where the start and end of component services is not explicitly shown, figure 47 uses service brackets to make it explicit.

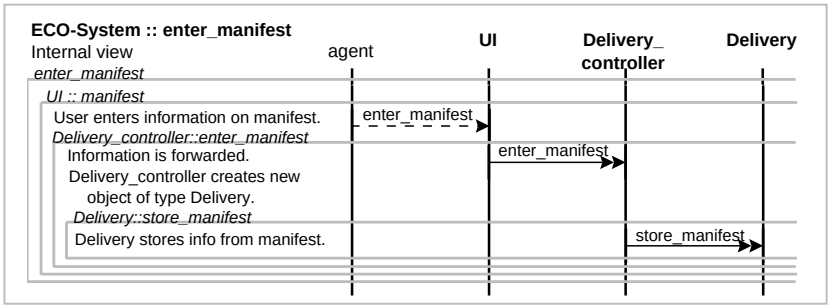


Figure 47: Internal view of a service with service brackets

If the diagram shows technical interactions, then service brackets do not add any additional information as there is a one-to-one correspondence between technical interactions and atomic services.

The composition of objects and services also leads to a hierarchy of interaction diagrams showing alternatively external and internal views of services. For example figure 74 and 75 show the external view of services of the system *Mail_Order_Firm*, figures 77, 78 and 79 the internal view of the system *Mail_Order_Firm*, figure 81 the external view of a services of the subsystem *Order_system* which is a component object of the system *Mail_Order_Firm*, figure 86 shows the internal view of one of the services of the system *Order_system*, and finally schema 87 the specification of a service of the atomic object *Order* which is a component object of *Order_system*. Of course, the external view of the

services of component objects (e.g. figure 81) must be compatible with the internal view of the system services the component services appear in (e.g. figure 78).

Internal views of white-box and black-box subsystems

In the case of white-box subsystems, the agents know the composition of the system services. Any agent can directly communicate with any internal object. But of course, every internal service which can be called directly by an agent of the system must be specified also as a system service.

In contrast to this, in black-box subsystems the system services hide their component services from any agents of the system. Therefore, the incoming messages always go to the dispatching object. This object has often the same name as the black-box subsystem. Figure 48 shows the internal view of the subsystem *Clients* which offers the services *add_client*, *get_client_info*, and *change_client*. In the resulting interaction diagrams for the internal views of these services, the interactions from agents always go the object *client_controller*. This object dispatches the messages to the objects *client_col* and *client*.

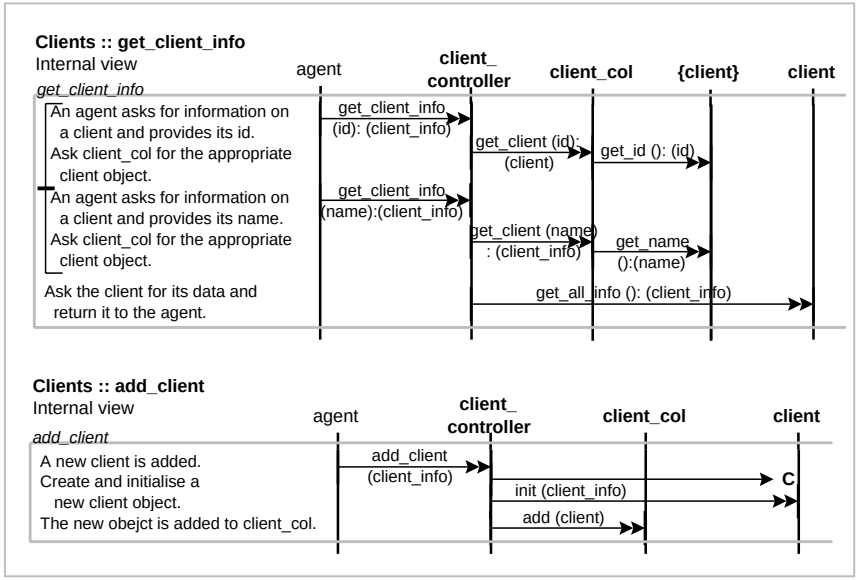


Figure 48: Internal views of a black-box subsystem

Of course, in this example, a white-box subsystem with direct access from the agents to the objects *client_col* and *client* would probably have been the better modelling approach.

Decomposition of interaction diagrams: subsystems versus groups

We thus get several possibilities to decompose an interaction diagram: we can decompose it along groups (see chapter 4.2.5.4), or we can decompose it along subsystems. In the interaction diagrams, both decompositions look quite similar, yet there are some essential differences:

- **Group**: A decomposition based on groups is used just for the decomposition of one single internal view; a group is not necessarily used in other interaction diagrams in which the same objects appear as well.
- **Black box subsystem**: A decomposition based on black box subsystems is relevant and must be used also in all other interaction diagrams. It also appears in any specification and diagrams showing the structure of the system. All the incoming interactions are handled by one specific controller or dispatcher object that often has the same name as the subsystem itself.
- **White box subsystem**: A decomposition based on white box subsystems is also relevant throughout the whole system. It also appears in any specification and diagrams showing the structure of the system. Yet the incoming interactions can go to different component objects.

4.3.2 Aggregation hierarchies

In the composition hierarchy we compose services of component objects into services of the system the objects are components of. In contrast to that, in aggregation hierarchies we aggregate services of an object (which may be an atomic object, subsystem or whole system) into services of the same object.

4.3.2.1 Complete aggregation of services

When considering the services offered by an object, we may realise that some of these services are often executed in a certain order, that in fact together they form a higher level service. These services can be aggregated by complete aggregation into a complete aggregate service.

*Definition 20: A service that is broken up into several services offered by the same server object is a **complete aggregate service** of these services - latter are called the **element services** of the complete aggregate service.*

We say that a complete aggregate service is a high-level service, because it represents a higher level view of the services of this object. Complete aggregate services can again be aggregated into higher level services. The lowest level element services in an aggregation hierarchy, i.e. those services that are no more complete aggregate services of lower level services, are called elementary services. Of course, an elementary service

offered by a system may still be decomposed into the component services of the component objects of this system.

*Definition 21: An **elementary service** is no more divided up or specialised into further services of this same object, i.e. it is neither a complete aggregate nor a generalized service.*

*Definition 22: A non-elementary service is a **high-level service**. It is always **abstract**[2], i.e. at execution or simulation time not the high-level service itself but its element or specialised services are triggered.*

*Definition 23: An elementary service offered by a system or a subsystem is an **elementary system service**.*

*Definition 24: An elementary service offered by an atomic object is an **atomic service**; it must be triggered by a technical interaction.*

Properties of atomic services

Atomic services are the smallest buildings blocks for modelling global behaviour in an object model. They cannot any more be decomposed into further services, they are neither complete aggregate nor composite services.

For an atomic service we further require that its triggering event is always a technical interaction, either a technical notification of a technical request. In the case of a request, the atomic service ends as soon as the return event is sent to the client object. The next interaction is already the triggering event for the following service. Between the request from the client object and the return event to the client object, the server object may send out requests or notifications to other objects and receive their answers. If the triggering event is a notification, the atomic service ends as soon as a further notification is received

Properties of complete aggregate services

Like an aggregate object type, which is more than just a collection of some objects (cardinalities, constraints, etc.), an aggregate service is also more than just an assembling of element services. The aggregation of services specifies not only which are the element services, but also the possible orders of the element services and the conditions that determine the order of element services. The possible orders that can be specified are the same as for interactions, namely: sequence of services, repetition of a service, optional services, alternative services and concurrent / interleaving services.

The complete aggregation of services is used to build abstraction hierarchies of services (see example in figure 99). Several lower level services are assembled into a higher level service. A complete aggregation thus has the following properties:

2. Our usage of the term abstract service is in analogy to the term abstract object type, and does not correspond to the abstract use cases of OOSE (see also appendix A.2).

- The aggregate service is completely divided up into element services, i.e. all interactions and state changes are part of an element service.
- The aggregate service is an abstract high-level service.
- The element services are of the same server object as the aggregate service.
- The element services are elementary or high-level services.
- The element services may also be used in other services.
- Within one scenario model for each complete aggregate service there exists only one possible way to break up this aggregate service into element services; alternative decompositions result in another scenario model (see chapter 4.5).

Concerning the terminology there are some points which could cause confusion:

- We have chosen the term *aggregation* of services to convey that several services are collected or put together into a more abstract service of the same object. We have redefined this term in the context of services. We have not taken over the semantics that exist for aggregations of objects.
- We use the word decomposing whenever we look for the parts of a service, independent of whether this decomposition takes place in the external view of an object or in the internal view of an object. If it is the *external view* of the object, we *decompose* an aggregate service of this object into *element services* offered by the same object. If it is the *internal view* of the object, we *decompose* a service of this object into its *component services* that are offered by its *component* objects.
- The terms *element service* and *elementary services* are to be distinguished.

4.3.2.2 Partial aggregation of services

Like the complete aggregation of services, the partial aggregation is also a relationship between the services of the same object. Yet the aggregate service is not completely broken up into element services. The aggregate service only uses other services at some points and does other parts itself.

*Definition 25: A service that for part of its scenario reuses other services of the same object is a **partial aggregate service**.*

A complete aggregate service is an abstraction of several lower level services. All the interactions take place in the lower level services. At execution time, the first element service is triggered and not the complete aggregate service itself. The order of the services and the conditions for these orders are known to and observed by the agents calling the element services[3]. In contrast, a partial aggregation is only used to enable reuse of parts of services used in several services. These element services can also be called directly by an agent. Complete aggregate services are always high-level services. Partial

aggregated services are always elementary services. An example of a partial aggregate service is found in figure 95.

We can subdivide the elementary services into two categories: partial aggregate services and simple services.

*Definition 26: A **simple** service is an elementary service which uses no other services of the server object.*

Of course, we can develop a scenario model of the external view of a system using only partial aggregation, and using neither complete aggregation, specialisation nor extension. The result were an unstructured list of elementary system services yet without any redundancy, comparable to the use case model of OOSE.

4.3.2.3 Notation: interaction diagrams for aggregate services

External view of a partial aggregate service

The external view of a partial aggregate service is similar to the external view of an elementary service, only service brackets are added to show the start and end of each of the element services. Figure 95 contains an example of a partial aggregate service.

The pseudo code and the interactions of the element services may be hidden or may be shown. When they are shown, they must of course be compatible with the interactions and pseudo-code annotations shown in the interaction diagrams describing the element services!

If the control flow of the aggregate service contains alternations of element services, and if no conditions are specified, then the alternative parts must start with different element services. This condition is necessary in order that the model remains deterministic, and is also valid for complete aggregate services. This condition also implies that if during the process of modelling two different alternative services without conditions become each an aggregate service which both start with the same element service, then the model must be reworked. The common element service must be put in front of the alternation.

External view of a complete aggregate service

In the scenario type of a complete aggregate service all the actions take place within the element services. We have therefore two possibilities for modelling the scenario type of a complete aggregate service. We can either do it analogue to the partial aggregate service in figure 95. Or we can hide the actions of the element services, and only show which element services appear in which order. The result is an interaction diagram that consists only of the pseudo-code annotation. This has the advantage that there is no redundancy

3. In the case of system services that are triggered by users, often mechanisms are implemented that force the user to observe the possible order of the services. Such mechanisms are e.g. grey menu-items that can not be selected, or panels that must first be exited before the next service of the aggregation can be triggered.

to the external view of the element services, and thus also no danger of inconsistency. In a CASE-tool, the interaction diagram of the aggregate service can always be automatically expanded by the interaction diagrams of its element services. Examples for complete aggregate services can be found in figures 99 and 50. Figure 50 models the service *making_phone_call* as a complete aggregate service and it also shows the internal view of the two scenario types of the first element service *starting_phone_call* (compare also with figure 6 where making a phone call were modelled by several independent services and the dependencies between them could not be shown). Further interaction diagrams such as the external views of the element services, the expanded external view of the aggregate service or the internal view of the aggregate service can easily be automatically derived from the diagrams in figure 50 plus the internal views of the other three element services. Figure 49 contains the first part of the derived interaction diagrams for the expanded external and for the internal view of the aggregate service *making_phone_call*.

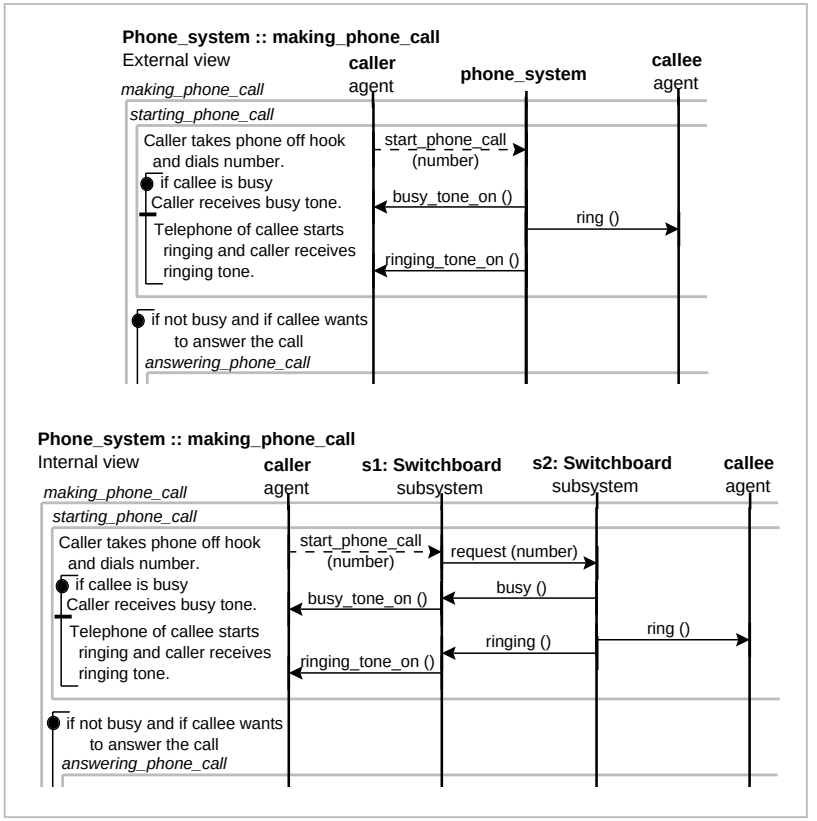


Figure 49: Deriving internal and external view for an aggregate service

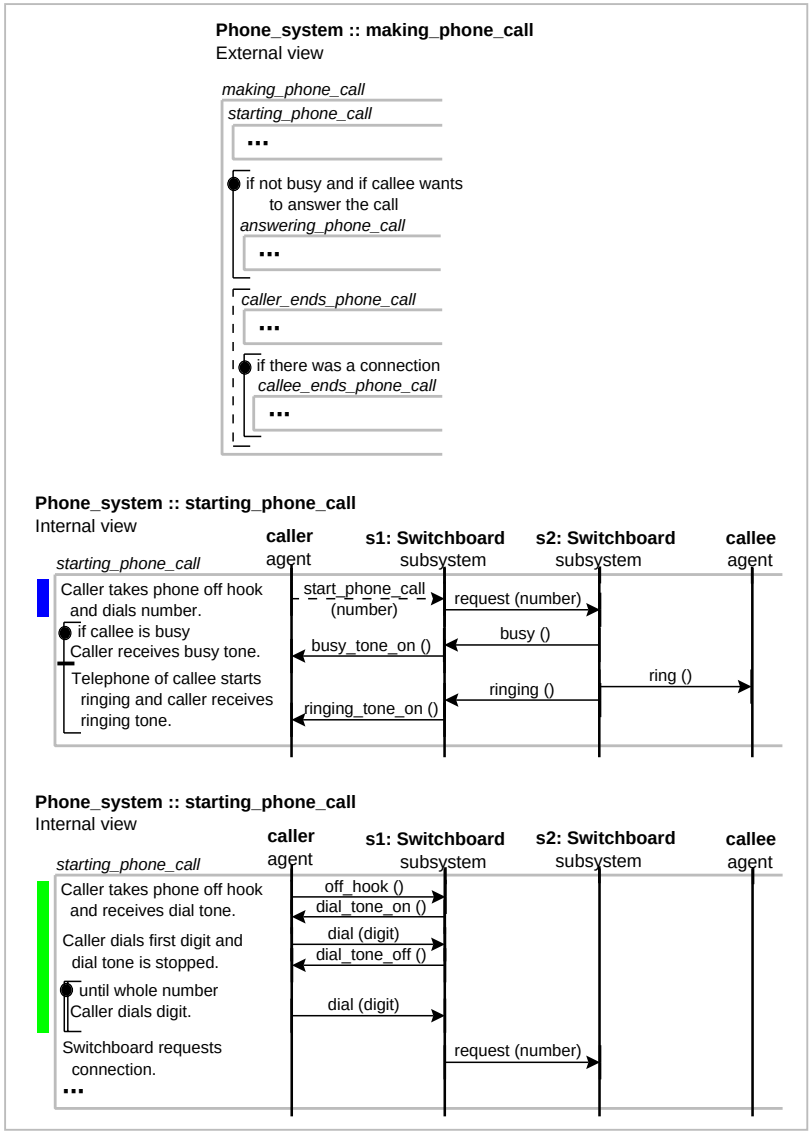


Figure 50: *Making_phone_call* as complete aggregate service

Internal view

The internal view of an aggregate service is a combination of the external view of the aggregate service and the internal views of the element services. Normally no diagrams are made for the internal view of aggregate services themselves, as they would only add redundancy to the specification of global behaviour. When viewing and discussing global behaviour, diagrams showing the internal view might be sometimes of interest. Ideally, they can be automatically generated by a CASE-tool. Example see figure 49.

4.3.3 Inheritance hierarchies

We differentiate between two kinds of inheritance hierarchies, the specialisation of services and the extension of services. Both hierarchies are relationships among the services of the same object.

4.3.3.1 Specialisation and generalisation of services

In analogy to the specialisation of objects where we may describe the common characteristics of object types in an abstract generalized supertype and where the virtual methods of the abstract supertype are implemented by the subtypes, we can also define a specialisation relationship among the services offered by the same object.

Example

Let us consider an order system where a client has four possible services for ordering items:

- *order_advance_payment*: If the ordered items are available, the client gets an invoice. The items are sent out as soon as the client has paid.
- *order_cash*: If the ordered items are available, the client gets a notice that he can fetch them at an agency of the company and pay cash.
- *order_credit_card*: If the ordered items are available and the credit card can be credited, the items are sent to the client.
- *order_COD*: If the ordered items are available, they are sent to the client with collect on delivery.

All four services have many things in common. In fact they are specializations of a more general service *order*. The specialisation is complete and disjoint, i.e. each possible scenario instance of *order* belongs to exactly one of its specialized services - a constraint we impose on the specialisation hierarchy of services.

Definition

*Definition 27: A service is a **specialisation** of a generalized service offered by the same object, if it handles some special cases (determined by the types or the parameters of the input interactions, or by the type or the state of the server object) of the generalized service.*

Properties of specialisation hierarchies

Above definition allows specialisation hierarchies of services only within services belonging to the same object. We do not consider hierarchies involving services of different objects because we have not detected any need for this in modelling global behaviour. Yet within the services of one object, we allow specialisation as well as aggregation hierarchies, because we consider generalisation to be orthogonal to aggregation.

Furthermore, specialised and generalized services have the following properties:

- We may build specialisation hierarchies over **several abstraction levels.**
- The specialisation of services is always **complete** and **disjoint**, i.e. every scenario instance of a generalized service is specified by a scenario type of exactly one specialised service of this generalized service.
- The **scenario type** of the specialized service is like a detailed scenario type of the scenario type of the generalized service, yet it covers only some of all the possible scenario instances of the generalized scenario type.
- A generalized service is an **abstract** service. The specialised services may be abstract high-level services or elementary services.

Polymorphism

Whenever we use in the specification of a system the name of a generalised service, it is decided at run- or simulation-time which one of its specializations has to be chosen. This is determined:

- by the input interaction type that triggers the service,
- by another input interaction type,
- by the values of certain parameters of input interaction types,
- by the state of the object offering the service,
- or by the type of the object in case the server object has subtypes.

In the first and last case, which service is to be executed can be determined when the service is triggered. In the other cases, the system model becomes non-deterministic, because only during the execution of the service can the appropriate specialisation be chosen. In these cases, this modelling construct cannot be used for low-level design models. The specialised services must be replaced by one single service, either a simple or an

aggregate service (see chapter 4.5.3). Yet when eliciting requirements and developing the initial analysis model, specialisation hierarchies with any kind of preconditions are of great value.

4.3.3.2 Notation: interaction diagrams for specialised services

When having a specialisation hierarchy, the generalized service is not always described by an interaction diagram. More often only a schema is made for it. If an interaction diagram is made for both, the generalised service as well as all the specialised services, then the specialised scenario types must be deductible from the generalised scenario type similar to detailing. An example of this is given in figure 74.

Most often, by specialising a scenario type we replace some conceptual interactions by a more detailed sequence of interactions. This may affect the internal or the external view. Yet specialising a scenario type may also affect the state changes of the server object or the parameter values of the output interactions. In any case, several specialised scenario types are derived from the scenario type of the generalised service. These are described in more detail than the generalized scenario type and each one of them covers some of the possible paths through the generalized scenario type.

The conditions under which a certain specialised scenario type is invoked is found either in the pseudo-code of the interaction diagram (for instance conditions concerning parameter values), in the interactions (a specific interaction type), or as additional text below the diagram. Furthermore, the conditions are also mentioned in the schema of the service. Just as for any other service, a specialised service may be modelled with interaction diagrams for its external view as well as for its internal view (see for instance figures 74, 77, 78, 79 and 81).

Detailing a scenario type versus specialising a scenario type

Specialising a scenario type is similar to detailing a scenario type. In both cases, a scenario type containing conceptual events is refined in more detail. The result is either one (only detailing) or several (specialising) scenario types. Yet a specialised scenario type is not compatible to its generalized scenario type, because it does not cover all possible cases of the generalized scenario type, i.e. it specifies a smaller set of scenario instances. When only one detailed scenario type results, we consider it as being a scenario type of the same service, just showing more details. When several scenario types result, we differentiate between the high-level service that only contains the high-level scenario type, and the specialised services with the specialised scenario types. Figure 51 visualizes the difference for the case where in specialising and detailing a certain conceptual interaction is replaced.

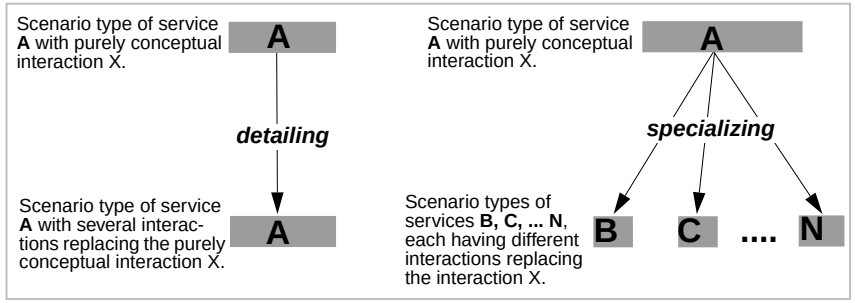


Figure 51: Detailing versus specialising scenario types

4.3.3.3 Extending services

In some situations we would like to specify a new service based on an existing service specification. Extending services[4] gives us the possibility of reusing the definition of a service and of its scenario types by adding further actions at certain points or replacing part of the scenario. The extension of services can be compared to inheritance from concrete classes: all methods and their implementations are inherited and new methods are added. By analogy, the extension of services allows the specification of some simple or normal scenario instances in one service, and more complicated variants in its extended services. Furthermore, in an incremental development process, when adding additional functionality we can define additional extended services instead of re-specifying the existing services.

*Definition 28: A service is an **extension** of another service offered by the same object if it handles some additional cases (determined by the state of the object or by the values of some input parameters) not included in the original service. The scenario type of the extended service adds some actions (state changes, interactions) to the original scenario type or replaces parts of the original scenario type.*

Properties of extension hierarchies

In contrast to specialised scenario types, an extended scenario type is not included in the original scenario type. On the contrary, all possible scenario instances of the original scenario type may also be covered by the extended scenario type. Specialising a scenario type narrows the conditions under which the scenario type can be applied. Extending the scenario type widens the conditions under which the scenario types can be applied.

4. The extension of services is not to be confused with the extend construct of OOSE. The extend relationship of OOSE inserts a use case A into another use case B under certain conditions. After the completion of use case A, use case B continues. In SEAM either the original or the extended service are executed, the extended service is not inserted into the original service. In contrast to OOSE, the extended service may not only contains additional actions but also replace certain actions of the original service.

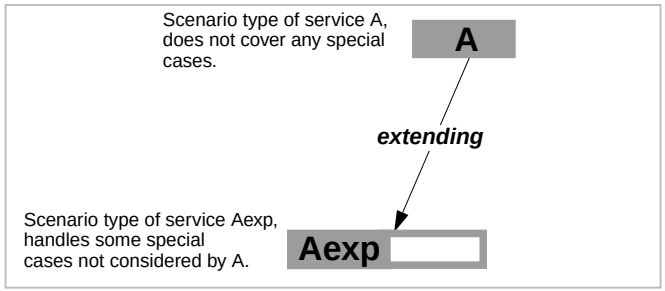


Figure 52: Extending services

Examples:

A typical example for using extended services is a dispenser:

- *Dispense_bottle_normal* is a service of the object *dispenser* that dispenses a bottle of the desired brand to the filing cabinet. The service is triggered by the interaction *choice_of_bottle (brand)* and can only be executed if everything is okay.
- The extended service *dispense_bottle_no_bottle* models the exceptional case where the dispenser has no more bottles of the desired brand. This extended service is also triggered by the interaction *choice_of_bottle (brand).*
- *Dispense_bottle_not_enough_money* is another extended service for the exceptional case where the client has not inserted enough money.

In figure 53 making a phone call is modelled by an extension hierarchy. The first service only specifies the successful phone call (only the first part of the interaction diagram is shown in figure 53). All other cases such as no answer or busy are treated by extended services (figure 53 shows the interaction diagram for the extended service *making_phone_call_busy*). Using an extension hierarchy to model making a phone call may be helpful when starting the process of requirements determination. Yet most probably this extension hierarchy would be transformed into an aggregation hierarchy as soon as the desired behaviour has been determined (see also chapter 4.5.3).

Another example (inclusive service schema and interaction diagram) for an extended service can be found in figures 96 and 97.

Polymorphism

An extended service is triggered by the same event as its original service. The criterion that determines whether the normal or the extended service is executed is either the value of some input parameter, or the value of the system state at some point of the service execution. As in certain cases of specialised services, this can not be determined when the service is triggered. Therefore, using extension hierarchies is helpful when eliciting re-

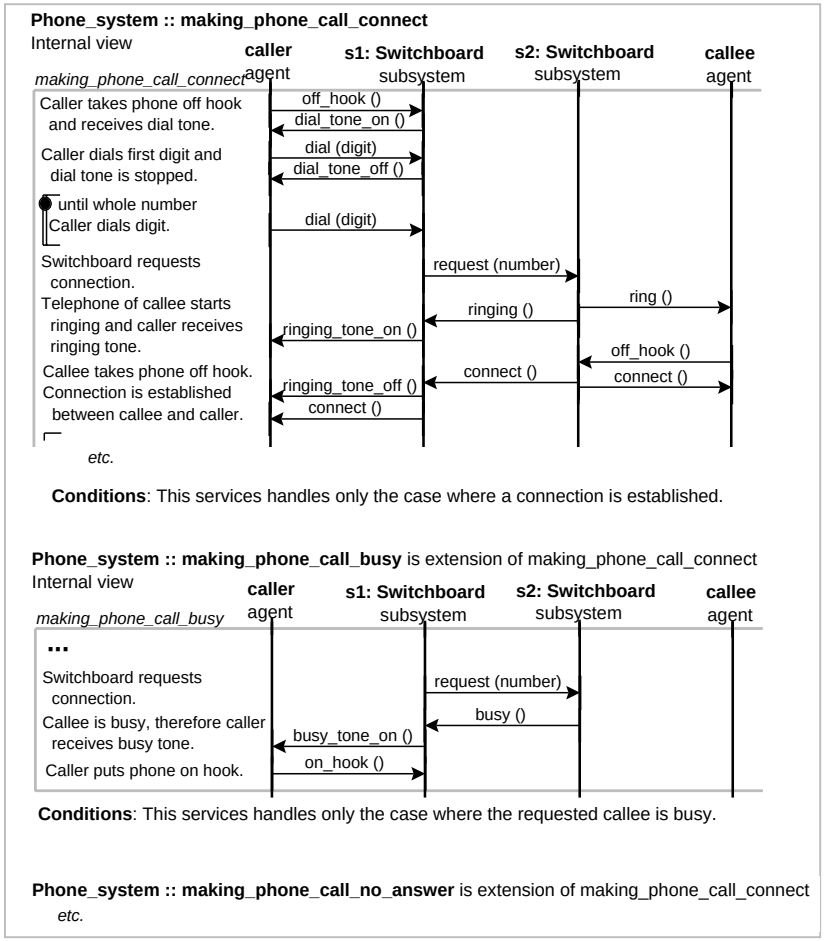


Figure 53: Modelling making a phone call by an extension hierarchy

quirements for a system, when comparing and integrating differing high-level scenario models, or when defining different increments of scenario models, to be implemented at different milestones. But for the final model, the hierarchy of extended services is replaced by one single service, most often an aggregate service.[5]

Orthogonality

Specialisation, extension and aggregation hierarchies are mutually orthogonal. In principle, a service can be an aggregate service, an extended service and a generalized service, though this is probably not be best modelling practice. The orthogonality of these three hierarchies can be compared to the orthogonality of the different hierarchies of object types.

4.3.3.4 Notation: interaction diagrams for extended services

In an interaction diagram for an extended service we normally hide all those parts of the scenario type that are identical to the original scenario type. Yet it must be unambiguously clear where the original scenario type is changed. Figures 96 and 97 contain the schema and an interaction diagram for the extended service *ECO-System::deliver_drums_-exception*. As with specialised services, the conditions that determine if the extended or the original service are chosen can be seen in the interactions or the pseudo-code of the interaction diagram, and are furthermore mentioned in the schema of the service.

4.3.4 Service diagrams

Service diagrams represent services as first-class objects. They focus on the relationships between services. We distinguish between composition graphs, aggregation graphs, inheritance graphs and context diagrams. The symbols for composition, aggregation and inheritance are derived from the Fusion method. Because we model relationships between services and not relationships between objects, the semantic is not identical. As a symbol for the services we have chosen rounded rectangles in order to distinguish services from the objects offering them.

Even if we consider services as first-class objects, they are still connected to their server objects. A service never becomes a server object, even if it is the only service offered by its server object, for instance in the case of a controller object. In contrast to other methods such as [Jacobson92] or [Graham94b] where the use cases or the higher level tasks are not linked to any objects, in our approach every service is offered by an object, either by an atomic object, by a subsystem or by a system.

4.3.4.1 Aggregation and composition graphs

Aggregation and composition graphs show the aggregation and composition of services. Figure 54 shows all the different symbols for aggregation and composition graphs.

5. If instead of the one-model approach a two-model approach were chosen then we could have a final analysis model that uses the extension and specialisation of services. The design model would only use the deterministic cases of the specialization and no extension at all. The mapping could be done in that the design would have one large service reflecting the specialization or extension hierarchies of the analysis, having the same name as the generalized or original service of the hierarchies in the analysis model.

Cardinalities

The cardinalities to the lefthand side of the element services have the same meaning as in the Fusion method:

- ***** denotes that the component or element service is executed zero or more times in each instance of the composite or aggregate service,
- **+** denotes 1 or more times,
- **1** denotes exactly one time.

In most cases, the cardinalities are omitted, either because they are not yet known when the diagram is drawn, or because they are of minor interest (why showing how many times a component or element service is executed when their order cannot be expressed?).

Aggregation graphs versus interaction diagrams

When using aggregation graphs for objects, it is possible to display all essential information in the diagram. When using aggregation and composition graphs for services, one important aspect cannot be shown: the order of the services, i.e. the flow of control.

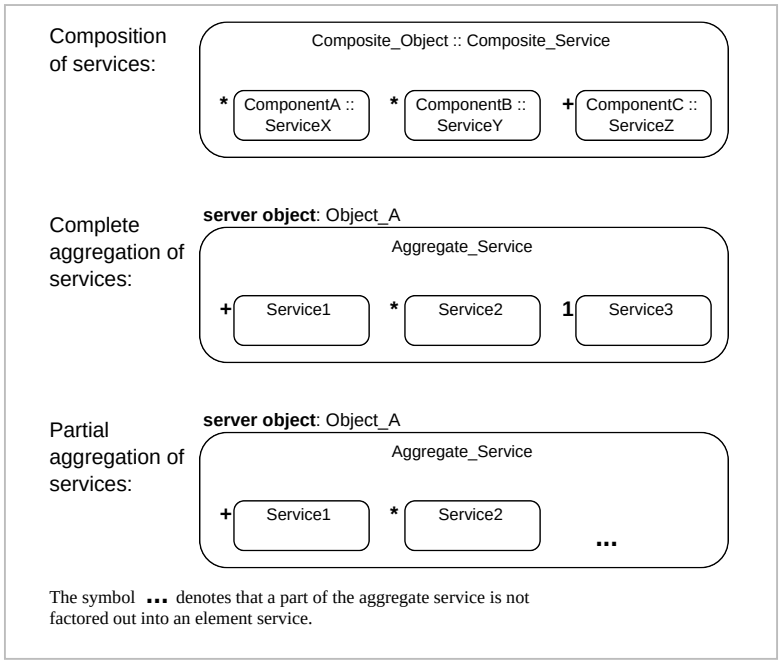


Figure 54: Aggregation graphs for services

In contrast to objects, where the order of attributes or component objects is irrelevant (there does not exist an equivalence to the control flow), constraints concerning the order of the services is an important aspect of composite and aggregate services. Therefore interaction diagrams are better suited for reflecting the aggregation and composition of services than service diagrams. Service diagrams are merely used to give a high level overview of the relationships between the services. They are redundant to the interaction diagrams and can be derived from them.

Suppressing the difference between aggregation and composition

If in the composition and aggregation graphs the names of the server objects are omitted, the aggregation and the composition of services can no longer be distinguished in the service diagrams. This may be of great value when making the first drafts of services. When it is not yet clear if the parts we are identifying are element services offered by the same object as their aggregate service or if these parts are component services offered by component objects, then we can draw a service diagram that does not yet make this distinction.

4.3.4.2 Inheritance graphs

Inheritance graphs are used to model specialisation and extension hierarchies of services. For modelling a specialisation relationship we use a solid triangle, for modelling an extension relationship we use an outlined triangle (see figure 55).

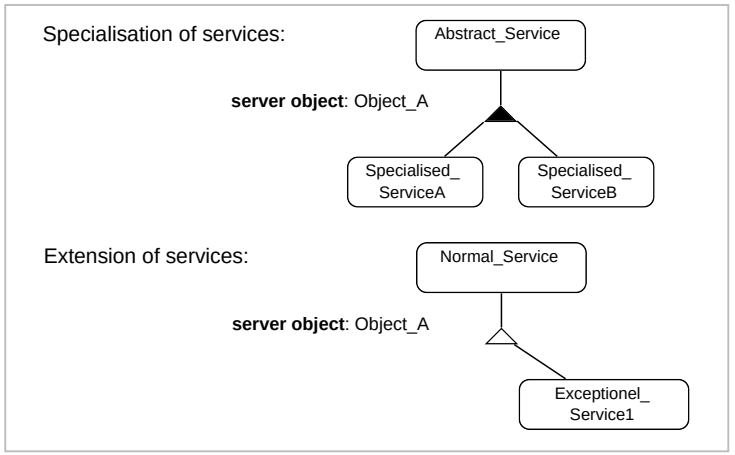


Figure 55: Inheritance graphs for services

In Fusion, a solid subtyping triangle denotes that the supertype is an abstract type and that its partitioning into subtypes is complete and disjoint [Coleman94, Appendix C].

This constraints are also valid for the specialisation of services. In contrast to this, an extension of services is neither disjoint nor complete, which is denoted by the outlined subtyping triangle.

4.3.4.3 Context diagrams

A context diagram shows for one object all the services it offers and whom these services interact with. In case the services are defined on several abstraction levels, just one of the abstraction levels is chosen, normally the highest one. Some of the aggregation, specialisation and extension relationships of the services may also be shown, in case this does not overload the diagram. In a context diagram we do not only show services triggered by external agents. We also show services triggered internally (e.g. by time-outs), whether or not they interact with any agents at all (example see figure 56).

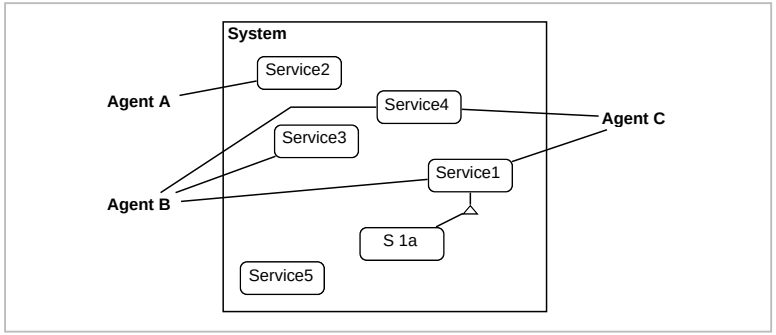


Figure 56: A context diagram

Context diagrams on the level of the whole system have already been used in structured analysis (see e.g. [Yourdan89]) and have been promoted in the object-oriented community by [Jacobson92] and [Graham94b], context diagrams have also been adopted by [Booch95]. Context diagrams are especially valuable at the beginning of the project to find out the scope of a system (especially in finding user groups and in allocating services to agents) and in the documentation of the system to give an introduction for people unfamiliar with the domain. Context diagrams could be used on any level, even for atomic objects and their collaborators, yet there their usefulness must be doubted, even more so as context diagrams are redundant to a complete scenario model consisting of interaction diagrams.

4.3.5 Excursus: comparison to other methods

Terms and hierarchies of services in other methods

The following table tries to map some terms used in other methods to the terms of SEAM. In most cases the mapping is only approximate.

	SEAM high level system service	**SEAM elementary system service**	**SEAM atomic service**
OOSE	use case		operation
Fusion RE[a]	use case	system operation	method
FORAM	task	task, atomic task	task, atomic task
[Glinz95]	scenario		-
[Regnell96]	use case, episode	episode	-

The following table gives an overview of modelling elements for hierarchies as provided by other methods. The table must be read with some caution, because the syntax and semantic of the modelling elements do not correspond precisely. More details about the methods can be found in the appendix.

	SEAM compl. aggregation	**SEAM specialisation**	**SEAM extension**	**SEAM partial aggregation**	**others**
OMT				add	
OOSE			alternative courses	uses/ abstract use cases	extends
Fusion RE[a]	use cases - system operations[b]		extends[c]	uses	
FORAM	component tasks	super-tasks	side-scripts	side-scripts	
[Glinz95]	scenario composition				
[Armour95]	high level use cases[b]	detailed use cases		abstract use cases	
[Regnell96]	flow charts with episodes			reuse of episodes	

a. Fusion extended for requirements engineering, as described in [Coleman95].

b. Only one level of abstraction.

c. The extend construct of Fusion for RE has the same syntax and semantic as the one of OOSE, but in contrast to OOSE it is only used for requirements elicitation and not throughout analysis and design; its goal is more like the goal of our extension construct.

Decomposition along subsystems or groups versus decomposition along call hierarchies

In SEAM we decompose the behaviour of an object-oriented system **along subsystems**. This approach is based on the following concepts:

- **Object-oriented**: We use the object-oriented and not the functional paradigm. Services are not independent procedures but are part of the objects offering them. Therefore the decomposition of services is not disconnected from the decomposition of objects. On the contrary, primarily the objects are decomposed, and together with them their services are also decomposed into the services of their component objects.
- **All-agent view**: Our approach is based on an all-agent view of the object offering the service. The specification of the external view of a service shows not only the interactions between the triggering object and the server object, but includes also all interactions with other collaborating objects. We only can adopt an all-agent view because we look at the services in connection with the objects offering them. If we looked only at the service itself, we could not discriminate between the server object and any other collaborating objects, and would thus have to adopt a one-agent view in the external specification of services. OOSE [Jacobson92] uses an all-agent approach in its interaction diagrams of use cases, too.

For the description of scenario types of services this has the following consequences:

- The external view of a service contains the server object with all its collaborating objects, yet it does not show any interactions among the collaborating objects. The whole course of interactions between the server object and its agents is shown, starting with the interaction that triggers the whole service.
- The internal view of a service contains all the component objects of the server object and all its collaborating objects (often collapsed into a group of objects). The whole course of interactions is shown, starting with the interaction that triggers the whole service.
- The external views of services of component objects need not be modelled separately, they are implicitly defined by the internal views of the system services of which they are components.

Even when the elementary system services have been chosen to be as short as possible, this approach can lead to very large diagrams for the internal view of services, especially when the system as a whole is composed directly from atomic objects and there are no intermediate subsystem layers. To further decompose these diagrams and the elementary system services they show, we offer the following possibilities: we can introduce subsystems, so establishing a hierarchy of composite services. Or we can introduce groups of objects and decompose a diagram along the groups. The subsystems are treated as single objects in the diagram for the internal view of the system service. Further diagrams

then show the internal view of all the services offered by the subsystem. If we only want to decompose one interaction diagram we use groups instead of subsystems.

An approach we have not chosen here is the **decomposition along call hierarchies**. This can be used when having only one level of objects, and all interactions are requests in a sequential system. We then could define external and internal view differently, based on a one-agent paradigm and on structured programming (functional decomposition). The specification of the “external” view would then include only the interactions with the calling object (see e.g. the contracts of [Meyer93], the operation schemas of object methods in [Coleman94], the classical way to specify the header of procedures or subroutines). The “internal” view would consist of all the requests to other collaborating objects, and of all the requests caused by these requests. The chain would continue until certain objects could answer the requests without needing the help of other objects. The chain of requests is a call hierarchy. The decomposition of services could be done along this call hierarchy: the chain of requests can be divided at any point. At each request we consider the rest of the chain as being the internal view of a service; the external specification of the service is only from a one-agent view. In this approach a functional view is adopted: the external and internal views of services are no longer connected to the objects offering the services. The advantage (or disadvantage) is that the decomposition of diagrams is possible without grouping objects into groups or subsystems.

Fusion uses call hierarchies to decompose the internal view of its system operations. The same approach has been adopted by OMT. However to avoid too strong a bias towards functional decomposition, the textual annotations of its interaction diagrams are divided up into one description for each method instead of one description for each diagram. Though in SEAM we use the decomposition along subsystems and groups, we can nevertheless also describe the call hierarchy of some requests, just as we can describe any kind of sequence of interactions by a scenario. Yet these diagrams no longer describe the internal or the external view of a service. Services are always modelled from an all-agent view and interaction diagrams are always decomposed along subsystems or groups. We consider this to be closer to the object-oriented paradigm than one-agent views and decompositions along call-hierarchies.

4.4 Life-cycles of objects

Though the main focus of this thesis lies on the concepts for modelling services and scenario types, we also mention the modelling of object life-cycles for two reasons. First, one aspect of services is their possible order and the preconditions that must be fulfilled in order that a service can be triggered. Second, we already model part of the object life-cycle in the specification of higher level services. When we further abstract the services of an object by complete aggregation, we finally get one single service encompassing the whole functionality the object offers; as a side effect this service specifies also the life-cycle of the object.

4.4.1 Order of services and essential states

Order of services

The object life-cycle specifies all possible sequences in which the services of the object can be executed. Part or all of the life-cycles of the system, its subsystems and its atomic objects are already given by the specification of the services: if the elementary services of the system as a whole are aggregated into higher level services and if these higher level services are aggregated again, then these aggregate services define the life-cycle of the system as a whole. Part of the life-cycles of subsystems and atomic objects are also defined by the aggregate services of the whole system, in combination with the specification of the internal view of the elementary system services (i.e. the representation of the elementary system services as compositions of services of subsystems and atomic objects). But the aggregation of system services and the composition of services of internal objects (subsystems and atomic objects) do not necessarily contain all the conditions for the life-cycles of internal objects; they only determine the dependencies between services in respect to the system as a whole (see classification of dependencies between services introduced in chapter 2.2.2.1, page 43). For all dependencies between services in respect to the life-cycle of internal objects, we need to have additional specifications. This can be done by using aggregation of services also for internal objects, and continuing this aggregation for each internal object until the complete life-cycle of this object is specified. Or we can use other modelling techniques to specify the life-cycles of the internal objects, or even for specifying the life-cycle of the system.

Another aspect of life-cycles is the question of who is responsible for preserving the life-cycle of an object. The responsibility for avoiding any violation of the life-cycle may lie with the object offering the service. This is normally the case when the agents triggering the services are human. A user-friendly user interface prevents the user from violating the constraints of the life-cycle. The responsibility for avoiding any violation of the life-cycle of the server object may also lie with the agents triggering these services, as is quite often the case if the triggering agents are software objects.

Essential states

In chapter 4.1.2.2 we have said that the state of an object is given by the values of its internal components (component objects or state variables) as well as its references to collaborating objects. Each possible combination of these values is a different state; we call these **micro states**. The actual micro state is determined by the history of service invocations, and influences the acceptance and outcome of future service invocations. Yet when considering only the preconditions concerning the acceptance of services, there exist many different micro states that are possible for accepting a certain service. We therefore group micro states together into sets of states, and call these sets **macro states**.

*Definition 29: Each possible value of an object is a different **micro state** of this object.*

*Definition 30: A **macro state** of an object is a set of possible micro states of this object; these micro states have in a certain context the same effect on the services of this object.*

*Definition 31: An **essential state** is a macro state that is used in a specification of the life-cycle of this object.*

Which macro states are considered as essential depends on how we define the life-cycle of an object. Essential states may overlap, i.e. they may contain identical micro states. An essential state may also be a subset of another essential state. Not all micro states of an object need to be part of an essential state, this depends on what for and how essential states are used.

We might wish to visualize the essential states of an object in an interaction diagram in which the object appears (for instance when modelling telephone switchboards). The interaction diagram in figure 57 shows the essential states *registered*, *awaiting_payment* and *completed* of the object *order_advance_payment*.

The essential states of an object may be listed in its schema. But normally we mention the essential states in the object schema only if they are used in the interaction diagrams or the state transition diagrams of this object. Otherwise we can omit them. For each essential state we specify its name and give a short description. This description is either textual or it is a predicate over the possible values of component objects and references.

Excursus: the definition of states in other methods

Most methods do not differentiate between micro states and macro states. On one hand, any operation that changes any values of attributes, variables or relationships, changes the state of the object. On the other hand, states are also used in the state transition diagrams that show the life-cycle of the object, and these states do not always change when the state (i.e. the value of the object) changes. Many methods simply use the term “state”, and it is up to the reader to infer the correct meaning from the context.

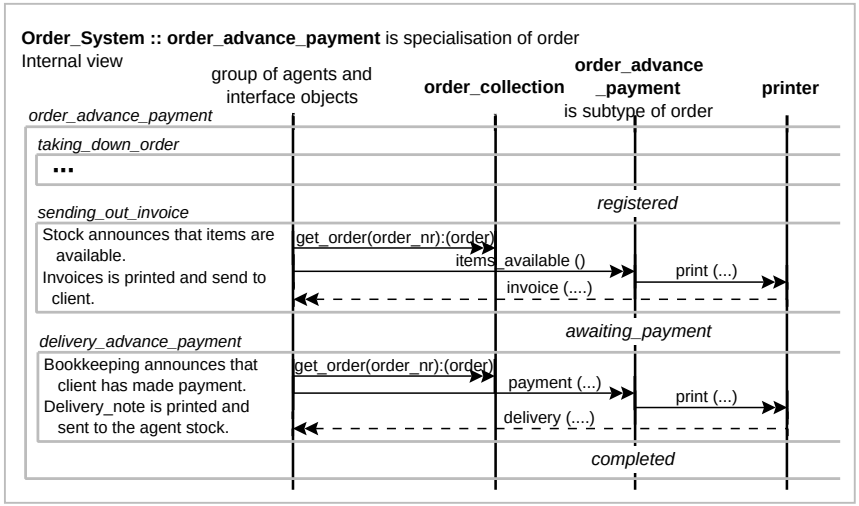


Figure 57: Essential states

Some definitions of the term "state":

- In **Fusion**, the term "state" is not used for individual objects but only for the system as a whole. Yet for the system it is used in two contexts. First there is the system state, which "consists of all the objects that have been created since the system started, and of all the relationships that appear on the object model" [Coleman94, Appendix C]. This would correspond to our micro state. Second, states are also used in the state machine which implements the system life-cycle [Coleman94, Appendix D]; these states would correspond to our essential states.
- **OMT** uses the term state in the sense of our macro states: "A state is an abstraction of the attribute values and links of an object. Sets of values are grouped into a state according to properties that affect the gross behaviour of the object [Rumbaugh91]." An object that always responds to an event qualitatively in the same way is said to be stateless [Rumbaugh93]. States are used in the state event diagrams of the dynamic model. State generalisation and nested state diagrams allow the modelling of the states of an object on different abstraction levels.
- **Booch**: "The state of an object encompasses all of the properties of the object plus the current values of each of these properties". The properties are the totality of the object's attributes and relationships with other objects. The state space of an object encompasses an unquantified yet finite number of possible states [Booch94]. This definition corresponds to our micro states.

States can also be named and are also used in the state transition diagrams; these states would correspond to our macro states.

- **OOSE** [Jacobson92] distinguishes between the internal states, which correspond more or less to our micro states, and the computational states, which can be compared with our macro states. An object behaviour is described in terms of computational states, though underlying them there are the internal states. State-controlled objects select operations not only from the stimulus received, but also from the current state, whereas stimulus-controlled objects perform the same operation independent of the state when a certain stimulus is received; they have only one computational state.
- In the **OMG** reference model, the state of an object is defined by the set of values and relationships associated with that object [OMG92], thus the term state corresponds to our term micro state.
- **ODMG**: The state of an object is given by its properties (i.e. attributes and relationships) [ODMG93].

4.4.2 Notations for the object life-cycles

We show here some possible ways to model object life-cycles although a thorough discussion of this subject is beyond the scope of this thesis. We propose to define the object life-cycle either by regular expressions, by state transition diagrams, by preconditions, or by pseudo-code. We leave the choice between regular expressions, pseudo code, preconditions or state transition diagrams open, because depending on the requirements and characteristics of the project, one or the other approach is better suited, and because all four variants can be used in connection with our approach to modelling services and scenarios. In some cases it is more natural to focus on the sequences of services (for instance in the case of the life-cycle of a coffee machine we think at once at the services *insert_money*, *choose_product*, *take_cup*), se we might choose pseudo-code or regular expressions. In other cases it is more natural to think in states (for instance a bank account: *open*, *blocked*, *closed_down*), state transition diagrams may thus be more appropriate.

The object life-cycle can be thought of as being the continuation of the process of abstracting element services into higher level services by complete aggregation. So the object life-cycle can be refined along the whole aggregation hierarchy until elementary services are reached. The specification of the life-cycle of an object must be consistent with the specification of the aggregate services of this object, with the specifications of the internal view of any system services this object appears in, and with any aggregations defined of these system services.

Modelling the object life-cycle is not only important for the system as a whole. Also for every subsystem or atomic object that does not allow that its services are called in an arbitrary order, life-cycle models may be necessary.

Pseudo-code for the object life-cycle

In the same way as we have specified the order of interactions and services in the pseudo-code annotations of interaction diagrams, we specify the order of the services for the whole life-cycle of an object. The only difference to the pseudo-code annotations of interaction diagrams is that we simplify the notation by leaving off the service brackets of the lowest level services. When specifying the life-cycle with pseudo-code or regular expressions, we normally do not go down to the level of elementary services, but only to the level of the high-level services which are further specified by a service schema and by interaction diagrams. Just as for aggregate services we also require for the system life-cycle modelled by pseudo-code that the model is deterministic. Therefore in the case of alternations, either a condition must be specified, or the alternative parts may not start with the same element service. Also interleaving parts may not start with the same element service.

The left example in figure 58 shows a simplified life-cycle model of the game Risk, the right example shows the life-cycle of the first case study of chapter 5.

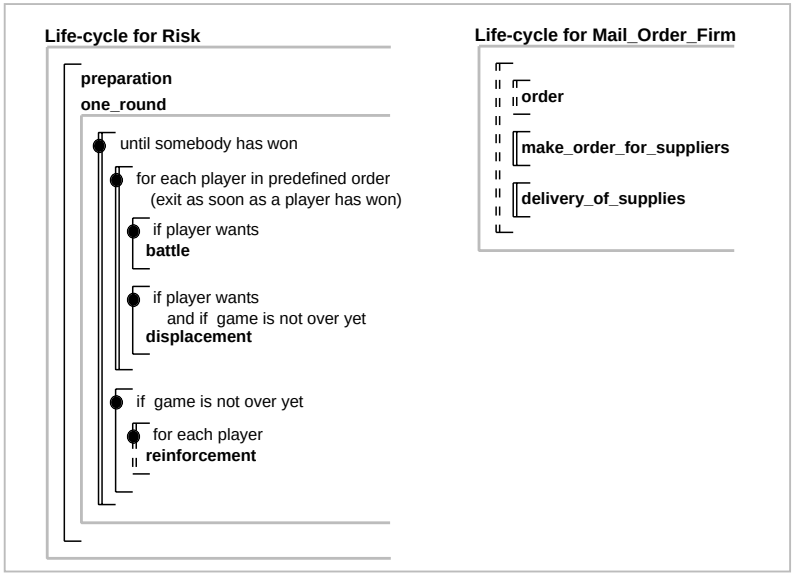


Figure 58: Life-cycles in pseudo-code notation

Regular expressions

Regular expressions show the possible orders of services offered by an object[6]. A regular expression consists of operators and of service names. The services may be elementary

or high-level services. Each regular expression specifies a high-level service. The available operators are:

- sequence of services: A **.** B
- interleaving or concurrent services: **A || B**
- optional service: **[**A**]**
- alternation: A **|** B
- repetition (zero or more): A*****
- repetition (one or more): A+
- interleaving or concurrent repetition: A **||***, A **||+**

Operator precedence is given by:

[ ], *, +, ||*, ||+, ., |, ||

The operator precedence can be overridden by bracketing the expressions. An example is the following life-cycle for the game Risk:

Risk: preparation . one_round
one_round = ((battle . displacement)* . reinforcement ||*)*

Regular expressions show all possible orders of services, but they do not specify under which conditions which sequence has to be chosen. This in contrast to the pseudo-code notation and the interaction diagrams, where with the help of the bracket code and the comments the conditions can be specified explicitly[7]. Furthermore, because regular expressions cannot take into account the state of the server object, they allow more sequences of services than are really possible. It is not possible to show under which conditions an object must refuse a certain service. For any optional, alternative or repeated service there are basically too cases to distinguish:

- The decision if a service is invoked or not lies with the object offering the service, i.e. if the service is optional or not depends on the state of the server object (and thus on its history). This case cannot be modelled adequately by regular expressions based on services[8] (compare also above example with figure 58).

6. In contrast to Fusion, we use regular expressions to show the order of services offered by an object (which may be an atomic object, a subsystem or a system), and not to show the order of input and output events of a system.
7. Because conditions cannot be specified, regular expressions can also be non-deterministic. This in contrast to the pseudo-code notation, where either conditions are required, or alternative or interleaving parts may not start with the same service. In order to keep the regular expressions a short-hand notation of the pseudo-code notation, we allow non-determinism.
8. In Fusion these conditions can be modelled by output events, supposing that the output events are chosen in such a way that they reflect essential state changes of the server object, for instance by choosing the output events *okay()* and *notokay()* instead of *status(status)*. Yet this may lead to output events which are really parameter values of a more abstract interaction. Furthermore, in [Coleman94] it is recommended to refrain from using regular expressions in this way.

- The decision if a service is invoked or not lies only with the object calling the service. From the point of view of the server object, the service may or may not be called. This case can be modelled adequately by regular expressions.

The advantage of regular expressions over interaction diagrams and over the pseudo-code notation is of course that they are very condense and that they can be used without a diagramming tool.

State transition diagrams

Instead of regular expressions or pseudo-code, we can also use state transition diagrams to show the life-cycle of an object. The states in a state transition diagram are the essential states listed in the schema of the object. The states may further reappear in interaction diagrams. The transitions[9] are the services an object offers; these may be high-level or elementary services. The symbol for a high-level service is a dashed arrow, for an elementary service a solid arrow. The arrows are labelled by the name of one or several services. If the label contains several services, then all these services must be executed in order that the transition can terminate and the next state can be reached. The label may further contain conditions. These conditions specify under which circumstances a service can be triggered, thus corresponding to the conditions we have in the pseudo-code notation for the life-cycle. In the case where more than one state can be reached by the same service, we also specify the conditions (i.e. postconditions of the service) which determine which state is chosen. Figure 59 gives an example of a state transition diagram.

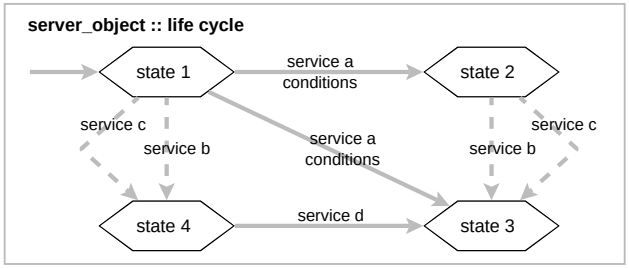


Figure 59: A state transition diagram

State transition diagrams can be nested[10]. This allows us having a higher level diagram showing the life-cycle of the object using high-level services, and having recursively

9. In this we differ from the classical state transition diagrams where the transitions are events, and we differ also from the notation from Glinz [Glinz95], where the states can encompass whole scenarios.

10. In contrast to the state charts of Harel [Harel87] and of OMT [Rumbaugh91], we focus on the services and the decomposition of services and not on the decomposition of states. Thus the nested diagrams do not zoom in on the states but on the transitions. The result is an aggregation hierarchy of services and not of states, and a simple list of states and not of services or events.

state transition diagrams for the high-level services that show their element services. Each diagram corresponds to a complete aggregate service. The example in figure 60 shows the state transition diagrams of the two complete aggregate services *service c* and *service b* of figure 59. A state transition diagram showing the whole life-cycle of an object can either use directly the elementary services, it can show only higher level services, or it can be decomposed showing both, higher level and elementary services.

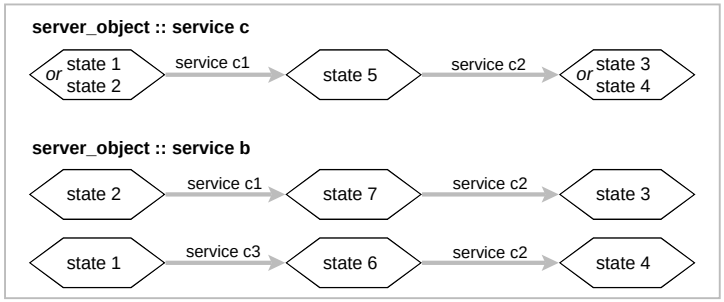


Figure 60: Decomposition of state transition diagrams

In principle, life-cycle descriptions in pseudo-code notation and life-cycle descriptions as state transition diagrams could be transformed into each other. This would also allow the verification of consistency between decomposed state transition diagrams and the aggregate services defined by interaction diagrams.

Preconditions

In certain systems, the high-level system services can all be applied at almost any time. There is not any typical order among them, lots of combinations are possible. Yet still there are certain constraints for the applicapability of the system services. Examples of such systems are text processors and diagramming tools. Modelling the life-cycle of such a system with pseudo-code or regular expression is a nightmare, the success is not guaranteed and the resulting model does not necessarily match the perception of the user. Much easier, and normally totally sufficient, is the use of preconditions for such systems. The schemas of the highest level services are enlarged by pre- and postconditions. These use the values of internal component objects or of essential system states.

Preconditions may also be used in combination with other notations for the system life-cycle. A possible combination is to use pseudo-code or regular expressions to show the life-cycle of the system as a whole and to use preconditions in the system services to specify indirectly the life-cycles of the internal objects, instead of having separate life-cycle models for the internal objects (see also chapter 2.2.2.1, page 43, where we distinguished between dependencies which refer to the system as a whole and dependencies which concern only certain object instances).

4.5 Transitions between scenario models

In chapter 4.2 we have introduced hierarchies of services and of scenario types. These hierarchies reflect relationships between services or scenario types within one scenario model. There are other relationships between services or scenario types which do not relate services or scenario types that can belong to the same scenario model. These relationships reflect transformations of services or scenario types. Because these relationships transform one scenario model into another one, they are not relevant as long as we just want to document one scenario model. Also they are not reflected in any modelling element. But they become very important as soon as we address the process of developing scenario models. During the development process, we do not only enhance a model, but we also rework it again and again. Whenever we change the definition of an interaction, the structure of a service hierarchy or an object in the underlying object model, we transition to a new scenario model. In this chapter we take a closer look at some of the possible transformations affecting the scenario model.

Definition 32: *A and B are interactions, scenarios, services or a set of interactions, scenarios or services. A **can be transformed into** B if in a given scenario model containing A A could be replaced by B without changing the content of the scenario model, i.e. only the structure of the model changes, but not the information expressed by it. The symbol **<-t->** denotes "can be transformed into".*

Observation 1: *(A <-t-> B) ==> (B <-t-> A) , i.e. "<-t->" is symmetric.*

Figure 61 shows two scenario models, *SM1* and *SM2*. These scenario models differ in that they have a different structure. But they specify the same global behaviour. By transforming parts of the models, we can transition from scenario model *SM1* to scenario model *SM2*. This transition can affect whole service hierarchies (e.g. *HS1* might be transformed into *HS2*), it can affect only specific scenario types (e.g. *ST1* might be transformed into *ST2*), or it can even be limited to certain interactions (e.g. *I1* might be transformed into *I2*).

4.5.1 Transformations of interactions

Two interactions may transport the same information flow and may have the same meaning, but they can still differ in their names and in their parameters. The trivial case is when only the name of the interaction needs to be changed in order to transform one interaction type or interaction instance into another one. Other possibilities are that the parameter types have to be changed or that the selection criterion has to be moved into the interaction name.

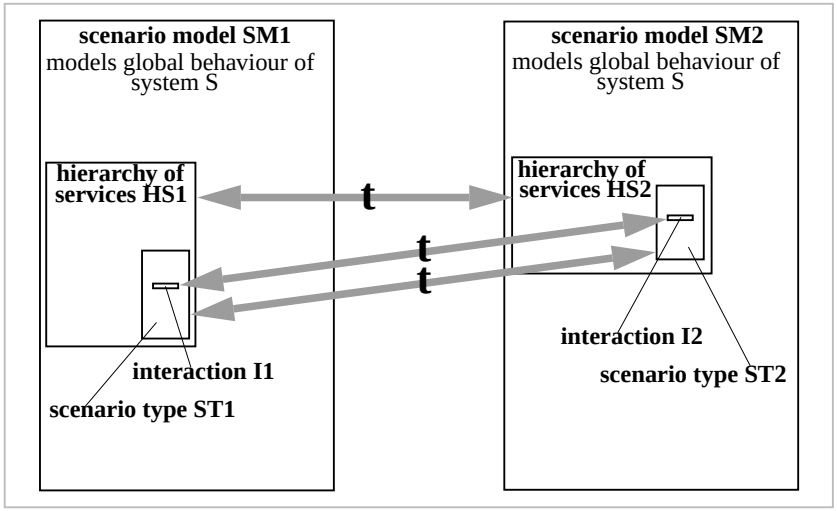


Figure 61: Transitions between scenario models

Observation 2: *If interactions A and B have the same information flow and convey the same meaning, then A <-t-> B.*

Changing the interaction name

Observation 3: *(A is an interaction with the name int_name1, B is an interaction with the name int_name2, A and B are identical apart from their names, and int_name1 and int_name2 convey the same meaning) ==> (A <-t-> B)*

A <-t-> B cannot be checked formally, it can only be decided by the developers in the context where these interactions are used.

Example: The interaction *enter_manifest (nooftype1, nooftype2, nooftype3)* in figure 93 could be transformed into the interaction *register_delivery (nooftype1, nooftype2, nooftype3).*

Changing the number and types of parameters

When two scenario models have different underlying object models or provisional data models, then the parameters of the interactions may be different though the interactions have the same name and transport the same information flow. Changing the parameters of an interaction type thus goes hand in hand with changing the underlying object or data model.

Observation 4: *(A and B are interaction types with the same name and the same information flow but have different parameters) ==> (A <-t-> B)*

Example: The interaction type *enter_manifest (nooftype1, nooftype2, nooftype3)* in figure 93 could be transformed into the interaction type *enter_manifest (no_drum_types)*, where *no_drum_types* is a data type containing information concerning the numbers of all three possible types of drums.

Changing the place of a selection criterion

All the parameters of an interaction have an influence on the future actions of the receiving object. Yet in some cases, we have a parameter that actually divides up the future actions of the receiving object into a small number of categories, and these categories are significantly different from each other. We call such parameters **selection criteria**. An interaction having as parameter a selection criterion can be transformed into several interactions, one for each category. In these interactions the selection criterion is then part of the interaction name (see example in figure 62).

Definition 33: *A **selection criterion** is either a parameter type having a small set of possible values, or it is a value of such a set and is part of the interaction name.*

Observation 5: *(A is an interaction type) ∧ (B is set of interaction types B_1, B_2, ... B_n) ∧ (A has a selection criterion as parameter) ∧ (B_1, B_2, ... B_n each have a selection criterion in their name) ∧ (A, B_1, B_2, ... B_n are identical apart from the selection criterion) ∧ ({y | y is a selection criterion of an interaction type of set B} = {x | x is a possible value of the selection criterion of interaction type A}) ==> (A <-t-> B)*

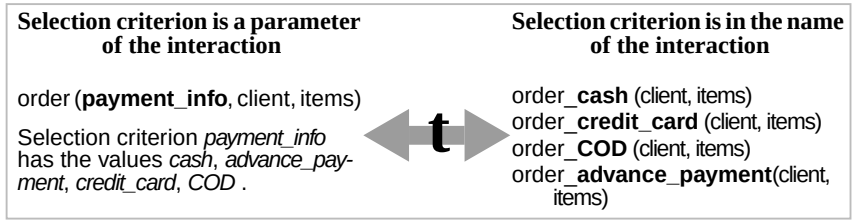


Figure 62: Selection criteria in interaction specifications

Moving the selection criterion of an interaction from its name to its parameters is analogue to transforming a specialisation hierarchy of services into one single service covering all different cases of the specialised services.

An interaction can be transformed into another interaction without affecting the service in which the interactions appear (with exception of atomic services). This is possible, because we differentiate between services and interactions. Services are neither determined by their triggering or other interactions, nor must the service name be equal to the name of the triggering interaction. Changes in a scenario model as the one in figure 62

only affect the specification of an interaction. They do not require further changes in the specification of the services in which these interactions appear (see also “first alternative model” on page 198).

4.5.2 Transformations of scenario types

Changing the algorithm of a scenario type

We already mentioned in chapter 4.2.1 and 4.2.3 that in general there are several possibilities to distribute the information flow of a service onto interactions. Also for the order of the interactions as well as the order of state changes there are most often several possibilities to choose from. On a given abstraction level, we have to decide for one variant. All other variants result in different scenario models. But these variants of scenario types can be transformed into each other by changing their algorithms.

Observation 6: *A and B are scenario types that have the same information flow and cause the same state changes in the server object ==> A <-t-> B.*

The possible changes that can be applied to A in order to be transformed into B are:

- replacing its set of interactions by another set of interactions
- changing the order of interactions
- changing the order of state changes

Example: In figure 63 we have replaced the two interactions *enter_manifest* and *end_check_in* of figure 95 by the interaction *complete*. This interaction has as parameters the information about the manifest which is now transferred after the individual drums have been checked in.

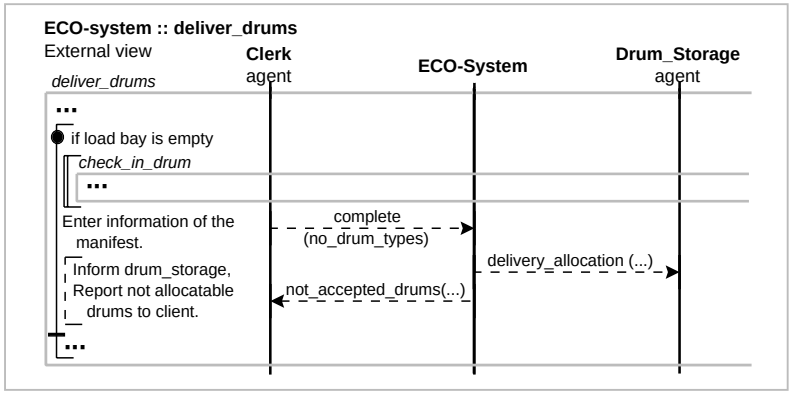


Figure 63: The service *deliver_drums* with an alternative design

4.5.3 Transformations of service hierarchies

Changing the structure of composition hierarchies

Whenever the component objects of a system change, then also the composition of the system services changes. Two services of two different scenario models specifying the same system may have exactly the same external view. But we may decompose the system differently. The result are not only totally different component objects having different responsibilities and offering different services, but also different algorithms and decomposition in component services in the internal view of the system services. In the process of designing the internal structure of a system, the composition hierarchies of the objects as well as the services they offer are normally changed several times until a satisfying design is found.

Observation 7: *A is a system service of system AS, B is a system service of system BS. A and B have the same external view but differ in their internal views ==> A <-t-> B by changing the structure and responsibility of the component objects of AS or of BS and by using other component services for A or for B.*

Transforming a simple service into an aggregate service

If a simple service contains several technical interactions or several state changes or can be further detailed into several technical interactions, it can be divided into several simple services that are part of a partial or complete aggregate service. The transformation into a partial aggregate service is mainly used when part of the service appear also in other services. So these common parts are factored out into element services. The transformation into complete aggregate services is mainly used for structuring the model and splitting up complex services. If a complex service is modelled as a simple service or as a complete aggregate service is a matter of the actual abstraction level and the actual goal of the model. As development proceeds, the need may arise to transform complex simple services into complete aggregations of several simple services.

Observation 8: *Service A is a simple service, service B is an aggregate service. Apart from the service brackets of the element services, the scenario types of B are either identical to or can be derived by detailing from the scenario types of A ==> A <-t-> B.*

An example of this transformation is found in figures 93, 95 and 99, where the service *ECO-System::deliver_drums* is first a simple system service, then a partial aggregate system service, and finally after the second revision of the scenario model a complete aggregate system service.

Changing complete aggregation hierarchies

When building complete aggregation hierarchies out of elementary services, there are several possible ways to abstract these services, resulting in different intermediate hierarchy levels. Also when dividing up a high-level service into its element services, there are often various possible ways to define these element services, resulting in different models having different sets of high-level and elementary services. An example for this is given in figure 64. The service *making_phone_call* is broken up into element services into two different ways. Both models model the same behaviour, both have their advantages and disadvantages. Which one is preferred in a specific telephone software system depends on the project and may evolve only after some time.

Observation 9: *(Services A and B are complete aggregate services,*
∃AS ∃BS ((AS and BS are simple services)
∧ (A <-t-> AS) ∧ (B <-t-> BS)
∧ (AS is identical to BS)))
==> A <-t-> B.

In analogy, also when making partial aggregate services there are often several possibilities as to which parts are factored out into which element services.

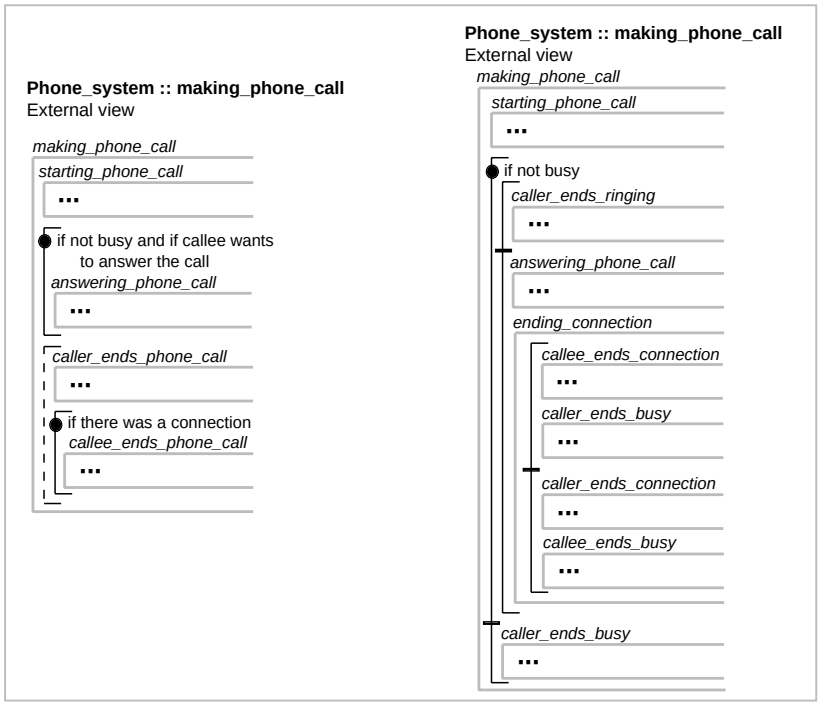


Figure 64: Transforming a complete aggregate service

Generalising services

When we have several services that are different from each other only in certain parts but on a higher abstraction level fulfil the same duty, it may be possible to model these services in three ways: as several independent services, as a generalisation hierarchy, or as one service encompassing all the different cases (see figure 65).

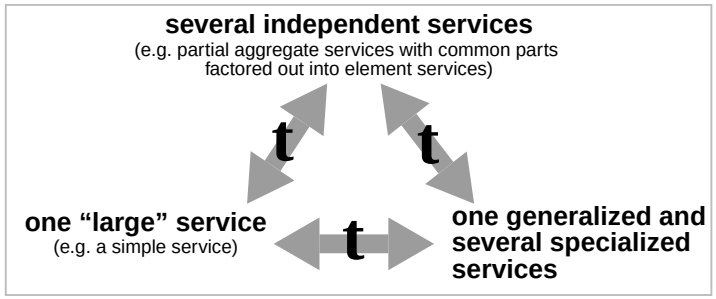


Figure 65: Transforming a generalized service

Observation 10: Any specialisation hierarchy can be transformed into a single service of which the scenario types
- *are modelled on the same abstraction levels as the ones of the specialised services and*
- *specify the union of all the scenario instances specified by the scenario types of the specialised services.*

Observation 11: Any specialisation hierarchy consisting of the generalised service A and the specialised services A_1, A_2, ... A_n can be transformed into a set of services B_1, B_2, ... B_n
- *where for each i B_i specifies the same behaviour as A_i*
- *where B_1, B_2, ... B_n are not the specialised services of another service*
- *where B_1, B_2, ... B_n are partial or complete aggregate services*
- *where the common parts of the services B_1, B_2, ... B_n are factored out into element services.*

Example: The service *Mail_Order_Firm::order* is modelled in figures 74, 77, 78 and 79 as a specialisation hierarchy and in figure 80 as a simple service (only internal view is shown).

Service specialisation and service aggregation are closely related. In a generalisation hierarchy, the parts that are identical in each service are described into detail in the generalized service. But instead of modelling these parts in a generalized service, we can use aggregation and factor them out into element services. Yet in contrast to a specialisation hierarchy, in the case of using aggregation we cannot express any conceptual specialisation relationship between services. Furthermore, all those parts that on a higher abstraction level are modelled identical but differ on a lower abstraction level have to be repeated in all the services.

Transforming an extension hierarchy into a specialisation hierarchy

Whenever we have a service that is extended by other services, we can transform this extension hierarchy into a specialisation hierarchy. A generalised service is created, the original service and its extended services become specialisations of this new generalised service (see figure 66).

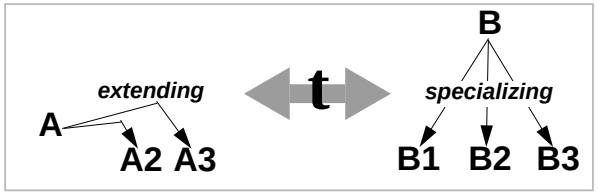


Figure 66: Transforming an extension into a specialisation

Observation 12: Any extension hierarchy with the services A_2, ... A_n being extended services of a service A can be transformed into a specialisation hierarchy with the services B_1, B_2, ... B_n being specialisations of a generalised service B
- *where for (i=2 to n) B_i specifies the same behaviour as A_i*
- *where B_1 specifies the same behaviour as A.*

Example: In chapter 4.3.3.3 we have modelled the services *dispense_bottle_no_bottle* and *dispense_bottle_not_enough_money* as extended services of the service *dispense_bottle_normal*. In another scenario model we could model this with a generalised service *dispense_bottle* and its specialised services *dispense_bottle_normal*, *dispense_bottle_no_bottle* and *dispense_bottle_not_enough_money*.

4.5.4 Why transitions between scenario models?

When looking at examples in text books, the impression may arise that the presented models have been developed all at once. Yet in reality, a model gets partially or totally redesigned several times. These redesigns are transitions from one model to another model. We mention here explicitly the need for having transitions between models because controlled changes are not only reality but also part of a good development practise, not the preservation of the first ideas. There are various reasons that lead to a redesign of and to changes in the scenario model:

- **Enhancements**: Additional functionality is added with or without changing the existing parts of the model.

- **Debugging**: Errors are removed by changing existing parts of the model.
- **Moving from an initial scenario model to the final scenario model**: If the initial model reflects the initial perception of the problem domain, then the model undergoes changes as the perception of the problem domain changes and as better modelling patterns evolve.
- **Replacing a provisional data model by an object model**: The external view of the system may be first described using a provisional data model. When the internal structure of the system gets determined, this provisional data model may be no longer necessary. As a consequence, all the interaction parameters and state changes specified in the external views of services need to be updated.
- **Improving the quality of the scenario and the object model**: The model is partially or totally redesigned in order to improve its quality (robustness, maintainability, low complexity, low coupling, good cohesion within classes and subsystems). For an example let's consider the selection criterion: at the beginning, it is not clear at all if the selection criterion is better part of the interaction name or better modelled as a parameter. It is even not known if there is a fix set of categories or not, if the services were better modelled as one service with one triggering event, as one service with several triggering events, as a specialisation hierarchy with conditions on the parameters, or as independent services. So at the beginning, arbitrarily one possibility is chosen. Later on, the initial solution may show itself as an unsuitable modelling approach and needs to be redesigned.
- **Integrating design patterns and frameworks**: Depending on the goals of the analysis model, design patterns and frameworks are often not used right away. It may either be too complex to develop an initial model and to integrate design patterns at the same time, or the possible range of applications may only evolve over time. Integrating design patterns and frameworks later on requires a partial redesign of the scenario model.
- **Integrating user viewpoints**: In order to integrate independently developed user viewpoints, the common parts of the models must be changed until they are identical.

We have mentioned some possible transformations of scenario models, these transformations are among those most often used when developing a scenario model. Being aware of these transformations helps tremendously when comparing different scenario models or when trying to improve a given scenario model. By giving a name to these transformations and by describing them explicitly, we help the developers in using the transformations and in communicating better on this subject.

4.6 Developing scenario models

4.6.1 The development process

Phases, increments and activities

The classical waterfall development process just consists of several phases which are carried out sequentially. The different abstraction levels of modelling, the contractual milestones, and the modelling activities are not really distinguished. In contrast to this, we distinguish in SEAM between the organisational aspects of a project (phases, milestones, contracts), the different abstraction levels and views of the model, the different versions of the model (preliminary and final system model) and the increments of the system. Various process models can be established, each one combining these aspects differently, and each one suited for specific project and domain constraints. Because a development process is project specific and must be adapted to the actual circumstances, constraints, risks and goals (see also [Beringer95]), we do not specify here a detailed process model. Concerning general process models, SEAM can be combined with any process model which does not contradict the assumptions we made in chapter 4.1.1. Two of the various possible process models are the one of Kruchten and the one of BON.

Kruchten subdivides the whole development process into the phases inception, elaboration, construction, transition, and evolution [Kruchten96]. The last phase contains a new development cycle and recursively contains again all the different phases. Each phase ends with a milestone for the management. Within each phase there are several increments. We can define for each increment its proper goal which affects also the content of its results. This goal in mind, those parts of the scenario model are developed that are concerned, with the appropriate grade of detail, precision and reworking. Analysis and design activities such as finding classes, selecting object scenarios, prototyping, incorporating reusable classes can appear in any increment, but in various degrees. Also the development of one specific abstraction level of the model is not restricted to one increment.

SEAM could also be used with the process model of BON (for more details see appendix A.7 and chapter 3.2.3.2). Like SEAM, BON also uses a one-model approach. It divides up the development process into tasks, having activities occurring in all the tasks. The deliverables of these tasks could be easily adapted to encompass the concepts and notations of SEAM.

The steps in developing a scenario model: an example

For an example, let us assume a small project where the scope of the desired system is well known, yet no similar system has yet been made (i.e. there is no preceding knowledge about the user interface, the system services and useful design patterns). In the fol-

lowing we give an example how the milestones might be defined. We only mention those aspects of the milestones which concern the scenario model.

- *First milestone:* The result is an initial scenario model of the external view of the system services. The parameters are based on a provisional ERD. Not all parameters need to be modelled. The system life-cycle is not yet modelled, no consistent level is required for the system services. They may be described and structured as is most convenient.
- *Second milestone:* The result is the internal view of the two most typical system services, down to atomic objects and atomic services (in order to better find the actual user interactions in the external view). Furthermore, the user interface gets specified. The external view of all system services is redesigned on two or three abstraction levels: a lower level mirroring the actual user interactions, and one or two higher levels giving a good overview of the system (all higher level services are complete abstractions of the lower level services). Specialisation and extension hierarchies are allowed. The life-cycle of the system as a whole is specified.
- *Third milestone:* The most important patterns used in the design are determined. Based on these, the internal view of all the services is developed. Whenever appropriate, subsystems are used.
- *Fourth milestone:* The quality of the model is carefully reviewed, especially the patterns and metaphors chosen are investigated. Where appropriate, a redesign is made. The external view of the services (parameter types of interactions) is adjusted to the internal object model.

Factors determining the steps and their order

There are various factors that determine which are the transient and permanent views and models in a specific project, how these models are linked with prototypes or implementation increments, how they are attributed to steps in the process, and in which order these steps are carried out. Examples of such factors are: kind of problem domain (process automation, scientific calculations, data management systems, low-level technical systems such as data conversion), size and complexity of the project, need for integration into larger systems or for cooperation with legacy systems, possibilities for reusing frameworks or components, precision and trustworthiness of given requirements and system architecture documents, starting point of the project (discussing system scope or detailing given system requirements and system designs), experience with similar projects, knowledge of the problem domain, further risks involved in this particular project.

Avoiding functional decomposition

A very straightforward approach to developing the internal view of a system were the following: an ERD is made which captures all the parameters of the external view, the

services are divided up into small functional units, these functional units are allocated to entity classes, if necessary further classes for control and display are added. Such an approach would take over characteristics of the matrix approach as discussed in chapter 2.2.2.4. It would combine the disadvantage of functional decomposition and of a bias towards data. With good reason, such approaches are criticised for example by [Sharble93] (bias towards data models) and [Firesmith95] (functional decomposition).

When using the concepts and notations of SEAM, a functional decomposition is not avoided automatically. Also with SEAM it would be no problem to produce low-quality object models. Yet there are several concepts helping to avoid it:

- *Objects and services are not decomposed independently*: In the internal view of a service no component services can be specified that are not offered by an object (interaction diagrams allow no other modelling). This can be used to determine the services or component objects according to the responsibility of these objects, using the criteria of high quality object-oriented design.
- *Several abstraction levels for system services*: It is possible to define first the global behaviour of the system or of a subsystem on a higher level and to define the elementary system services in parallel with the internal component services. This allows us adjusting the external view to the internal view given by an object-oriented decomposition, instead of following in the internal decomposition the external functional decomposition.
- *Provisional data models*: We explicitly adapt the interaction parameters used in the external view of a system to its internal component objects. As long as this object model does not exist yet, a provisional data model is used. But it is not assumed that this data model is already the design of the internal object model.
- *Iterations and change*: Even when we want to avoid it, we may get a functional decomposition. Therefore, the first model cannot be the final model. We first have to find the ideal patterns and responsibilities. We need one or more redesigns for the improvement of the quality and for the integration of object-oriented design patterns and reusable components. Our perception of the subsystem under consideration has to change gradually.

Functional decomposition or bias towards a data model must not always be bad! Depending on the circumstances, having a bias for example towards an existing relational database system may allow the easier and better maintainable system than having a large discrepancy between the objects in the application and the entities in the database.

4.6.2 Completeness of scenario models

Criteria for complete scenario models

The definition of the completeness of a specific version of a scenario model depends on the goals defined for this version (see chapter 3.1.3). A scenario model of an initial analysis model has other goals, and thus other completeness criteria, than a scenario model of a detailed design. Also not the same completeness criteria are appropriate for a model serving as the basis for outsourcing a development project, as for a preliminary model of an in-house project with an evolutionary approach. Therefore we cannot provide any general criteria for the completeness of a scenario model. The definition of these criteria make up part of the task of project and risk management, and the criteria are specific to each project and application domain.

In the following we give three examples of possible definitions for the completeness of a scenario model:

- *For a requirements analysis model:* The model has to show the external view of the system with all elementary services and their aggregates on an abstraction level that does not yet consider any details concerning the user interface and the actual user-system interactions, based on a provisional external data view. Special cases of lesser importance and error treatment needs not be taken into account. The goal is to define the scope of the system and the main tasks it has to carry out. The targeted audience are the users and the management. The model will be used for a go - nogo decision.
- *For a user interface model*: The model has to show the internal view of the user interface subsystem down to the level of single menu events or text entries. The model has two goals: to specify the external design of the user interface and to validate it with the users, and to make the detailed internal design of the user interface subsystem ready for implementation. Targeted audience: users and system designers.
- *For a design model*: The services must be decomposed down to atomic services and technical interactions. All interactions and references must be fully specified. Targeted audience: programmers implementing the model.

Determining elementary services in an analysis model

When modelling the external view of a system in an analysis model, criteria are needed to decide when the level of elementary services is reached, what kind of services are considered[1], and thus when the analysis model is complete. The easiest and most precise criteria is to decomposing services as long as it is somehow possible; this leads to a

1. The “complete” analysis model does it also model operational aspects such as services needed by system managers for maintaining and operating the system? See also the discussing of the notion “technology independent” in chapter 3.1.3.4 and the comments on this point in [Kotonya96].

decomposition down to single user inputs such as selecting menus or buttons and entering strings, and to a list of all services triggered by time-outs. Yet it is hardly ever the goal of the analysis model to specify already the details of the user interface. Therefore, more often the completeness criteria target at some higher abstraction level. The specification of such completeness criteria is often best done with project specific examples. But it is also feasible to have completeness criteria which restrict decomposition into very detailed services to the more complex and important services, and allow us leaving the other ones on a higher abstraction level.

Several scenario types for one service

For each service, several scenario types on different abstraction levels are possible. Most of these are of no value once a more detailed one has been drawn; they are transient diagrams which can be discarded. In the final model, only some of the more abstract views are kept in order to have an introduction into complex services.

Verifying the completeness of scenario models

In SEAM, automatic verification of the completeness is only possible concerning missing specifications of element and component services as well as of interaction parameters. Missing services not used by any other services can only be detected by human reviewers. For reviews it is crucial to have a precise definition of the goal of the model and of the abstraction level to be achieved. Only when the right review team and the right focus for the investigations is chosen, is it likely to have a trustworthy review result.

However, event the limited automatic verification mentioned above is not always possible. Depending on the goal of the model, we may only want to model the most essential services of a system for an analysis model, or only develop the internal view of some typical and some very complex services for the design model, or we may define only some of the interaction parameters. In these cases, no automated checks at all are possible. Also no precise rules can be given what should be part of the model and what can be left off. Nevertheless, partial scenario models may be the better choice when looking at the risks, benefits and costs of a project.

Excursus: completeness of scenario models in other methods

When we consider how the various methods attack the subject of completeness, we can roughly classify them into three categories.

Methods designed for verifying completeness and consistency

First, there are those methods which have as goal a model of the (external) global behaviour which should allow the running of formal checks on consistency and completeness and the automate simulation and prototyping. Examples are [Hsia94] and [Glinz95], which in contrast to our approach are methods targeted at verifying the completeness and consistency of requirements, even at the cost of other drawbacks and of a very narrow

scope of usage. The notation is formal and is chosen in such a way that extensive consistency checks are possible.

"Complete" models for documenting all the requirements and all the internals of global behaviour

There are other methods which target at having a complete scenario model and having easily understandable models. Yet they do not target at automatic verification, simulation and prototyping. Examples are methods such as Fusion, OOSE, and to some extent also MSA. In analysis, their goal is to have a scenario model which contains all the requirements. Therefore they use scenario types and not scenario instances. But inevitably, their criteria for completeness are only superficially precise (e.g. "essential" scenarios, "meaningful" abstractions). In order to avoid too large models, the targeted abstraction level cannot be chosen too low. But event then, experiences have shown that a complete scenario model of the external view is only feasible when the use cases, system operations or business processes do not go into the hundreds, because these methods do not offer any hierarchies of scenarios.

Fusion and OOSE model the complete internal view of the global behaviour of a system. If in larger industrial projects using these methods really all the interaction diagrams for the internal view are made, is beyond my knowledge and has probably not yet been investigated on a larger scale. When looking at the examples given in [Coleman94], it is conspicuous that the interaction diagrams ignore and omit the effective interface mechanisms (interface objects, detailed user system interactions). Also further system operations not modelled in analysis but nevertheless necessary for the functioning of the system, are also not modelled in the design.

Methods with partial scenario models

The third category consists of all those methods which do not strive for a complete scenario model at all, for instance OBA, BON, Booch and many other methods used nowadays in industry. Some of them model scenario instances, some scenario types. Most often they model the external and internal view of the system. BON for example lists in the scenario chart all of the scenario types, chosen on an high enough abstraction level to avoid two many details. Some of the scenarios are then documented by object diagrams, but not necessarily all. Details are again ignored (see example in figure 127). Common to these methods is that scenarios are only a mean to discuss about the scope of the system and to find the objects of the system. They are not the requirements specification and are not necessarily part of the final documentation.

4.6.3 Consistency of scenario models

Expressiveness versus rigidness

The notation and concepts we have chosen aim at:

- Showing scenarios and services on various abstraction levels.
- Modelling scenarios as they are perceived (aggregate, generalized and extended services).
- Documenting the scenario model also for outsiders and newcomers in a way which allows them comprehending the "big lines".
- Supporting the process of discussing the scenario model with participants of different backgrounds (not only specialists in analysis modelling).

Thus the modelling technique aims at expressiveness and it allows much freedom in how things get modelled. For example there are no strict rules how a purely conceptual interaction is detailed into further interactions, everything is possible (for instance a whole sequence of interactions can be detailed into another sequence, a direct mapping of lower level interactions to exactly one higher level interaction is no longer possible). There are no strict rules if the triggering event of a non-atomic service is an interaction from another agent or if it is another event not modelled as an interaction. There are several notations for the object life cycle. Last but not least, the pseudo-code is mainly text and not a formal language. Both, conditions as well as actions (description of interactions and of state changes) may be formulated in the language best understood by the team. The result is a modelling technique which consists of a set of simple and consistent concepts and a notation which is easy understood and useful for a vast variety of problems, but which is semiformal. It allows some consistency checks, but it is not formal and rigid enough for extensive consistency checks covering all the semantics of the model.

Pohl and Jarke propose in [Jarke94] to see the requirements engineering process as a cube with the three dimensions representation (informal - formal), agreement (personal view - common view) and specification (opaque - complete). Because we have chosen in our approach a simple yet coherent set of concepts and an easily understandable and presentable notation, our approach allows us starting the RE-process in the corner {informal, personal views, opaque}. During the integration of various viewpoints, the development of more detailed and precise scenario types and the determination of the internal views of services, we gradually circle towards the corner {formal, common view, complete}. Yet in the dimension formal - informal the progress is limited, as we do not provide a transition to formal specifications.

Consistency checks

The following very basic consistency checks are straight forward and can be easily automated:

- *Completeness*: In case a complete and not a partial model is the goal, it can be checked that all component services and component objects are specified by a schema, all parameters are part of the object model or preliminary data model, every system service has an interaction diagram showing its internal view, etc.
- *Ambiguity*: No object or service may be specified by more than one schema.

In the case of a complete model, the following consistency requirements could also be automatically checked, but they require more complex algorithms. Furthermore, whenever the conditions for repetitions and alternations in the pseudo-code need to be taken into account, these conditions cannot be compared automatically for equivalence, because no formal syntax is used and the text describing the conditions need not be the same in the various interaction diagrams, even if they express the same condition.

- *Internal view and external view of the same service:* The internal view must be a direct expansion of the external view, i.e. all interactions between the agents and the server object must be the same in the scenario type showing the internal view of the service as in the lowest level scenario type showing the external view of the service. Also the possible orders of these interactions must be the same in both views.
- *Life-cycle of an object and aggregate services*: Any aggregate service must not contradict the life-cycle specification of its object. If the life-cycle is specified by pseudo-code or regular expressions, then the checks are quite trivial. If the life-cycle is specified by state transition diagrams, then ideally first a specification in pseudo-code is derived. The algorithms for this are not discussed here as this would go beyond the scope of this thesis.
- *Composition of services in the interaction diagrams*: The external view of a component service must be compatible with all the internal views of all system services in which this component service appears. Compatibility includes that also the interactions reappear in one of the scenario types showing the external view of the component service, and also that the order of the interactions is preserved.
- *Composition of services and life-cycles of component objects*: The internal view of any system service must preserve the life-cycles of its component objects.
- *Services of specialised objects*: Whenever an object type is a specialisation of another object type, then its life-cycle may not restrict any order of services allowed by the life-cycle of the supertype (substitutability).
- *Reachability of services*: Each elementary service of the system as a whole must either directly appear in its life-cycle model, or it must be an element service of an aggregate service used in the life-cycle model of this system. Also each elementary service of a component object must either be directly used in at least one of the elementary services of the system this component object is part of (i.e. it must appear in at least one internal view), or it must be a element service of an aggregate service fulfilling this condition.

There are other consistency conditions which cannot be automatically checked, because they require semantic interpretation of the model:

- *Compatibility between scenario types of the same service*: One service may be modelled by several scenario types on different abstraction level. These must be mutually compatible (for more details see chapter 4.2.5.3).
- *Integrating viewpoints*: When an approach is chosen where first independent models are made for the various user viewpoints and later on these models are integrated into one model, then these individual models have to be transformed in order to get one single model with no inconsistency and no redundancy. Some inconsistency between the different models can be found automatically. Yet the very essence of the models, i.e. how these different views perceive the system behaviour, can only be unified manually. Especially important is here to detect all those services which reflect the same behaviour but have different names and use different scenario types to model them.

Whenever the system under consideration is a concurrent system, then it should be free of deadlocks. Although the interaction diagrams contain all the information necessary to check for deadlocks, this cannot be done directly. First other models would be needed to be derived from the interaction diagrams. This we do not discuss within the scope of this thesis.

Excursus: consistency of scenarios in other methods

In most methods covering the whole life-cycle from analysis to implementation, the checking of consistency plays a minor role. In all those methods which do not require a complete scenario model, either for the external view of the system or for the internal view, consistency checking is of course very limited (these are all the methods falling into category three concerning the completeness of the scenario model). Consistency checking is also very limited in all those methods, where different models are made for the dynamic, functional and data view, and where there are no well defined links between these models (for instance first generation of OMT).

Other methods require a complete scenario model, but the scenarios are described textually to a large extent (e.g. Fusion). In the Fusion method, some consistency checks are prescribed, but they are quite straight forward. For the analysis model, consistency criteria are for instance that the outputs mentioned in the operation schemas must be the same as those mentioned in the life-cycle expressions and that all system operations in the life-cycle expressions must be described by an operation schema. The content of the textual descriptions cannot be checked automatically. Whether they meet the requirements of the system and whether they are consistent can only be determined by reviews

and walk-throughs. More thorough consistency checking is only possible with formal notations. For example [Hsia94] and [Glinz95] propose various algorithms to check certain aspects of consistency in their modelling techniques (see A.9.2 and A.9.3 in appendix A).

4.7 Summary

4.7.1 Summary of the basic concepts

We have introduced an enhanced scenario modelling technique which allows:

1) modelling the external view of the global behaviour of a system or subsystem **on several abstraction levels** using the same concepts and notations,
2) modelling **internal and external views** of the global behaviour of a system using the same concepts and notations,
3) treating a system as a system of interacting subsystems that can be further **decomposed into subsystems**.

This allows us to choose an iterative process for capturing and designing the system behaviour. No longer does the best abstraction level for modelling external behaviour need to be found right away. In the case of systems having a global behaviour which is inherently structured, system services can be modelled as hierarchies and are no longer forced into a flat list (point 1). Point 3 allows us to start with a model that captures far more than the targeted software system, and then gradually zoom in on the behaviour of the software system alone. Finally the above points also allow us to avoid an unbridgeable gap between a very abstract external model of global system behaviour, and the actual interactions between user interface objects and problem domain objects. Different viewpoints and abstraction levels in the analysis and the design model can be diagrammed without ending up with separate models or with no possibility of modelling the details of user interactions.

In order to arrive at a modelling technique allowing the points mentioned above, we analysed the various kinds of relationships between services and scenario types. These relationships are reflected in our approach:

- by hierarchies of services, namely aggregation, composition, specialisation and extension hierarchies (relationships between services within the same model),
- by transformations of services and scenario types (relationships between services or between scenario types belonging to different scenario models),
- by the possibility of detailing scenario types and thus of having several scenario types for the same service (relationships between scenario types of the same scenario model and of the same service).

Figure 67 gives an overview of all the possible relationships between services. Relationships between services within one model appear in the scenario model and are therefore also treated by the notation. The relationships between services of different models do not appear in the final documentation of the scenario model, but are important for the modelling process.

possible relationships between services			
within one model			...
external view of one object			internal view of an object
abstraction[a]	**Complete Aggregation** A complete aggregate service is an aggregation of several element (but not necessarily elementary) services.	**Specialisation** One generalized service is specialised into several specialised services.	**Composition** A (normally elementary) system service is decomposed into the services of the component objects of the object offering the system service (internal view of the system service)
reuse[b]	**Partial Aggregation** A partial aggregate service reuses one or more other services.	**Extension** An extended service extends another service.	

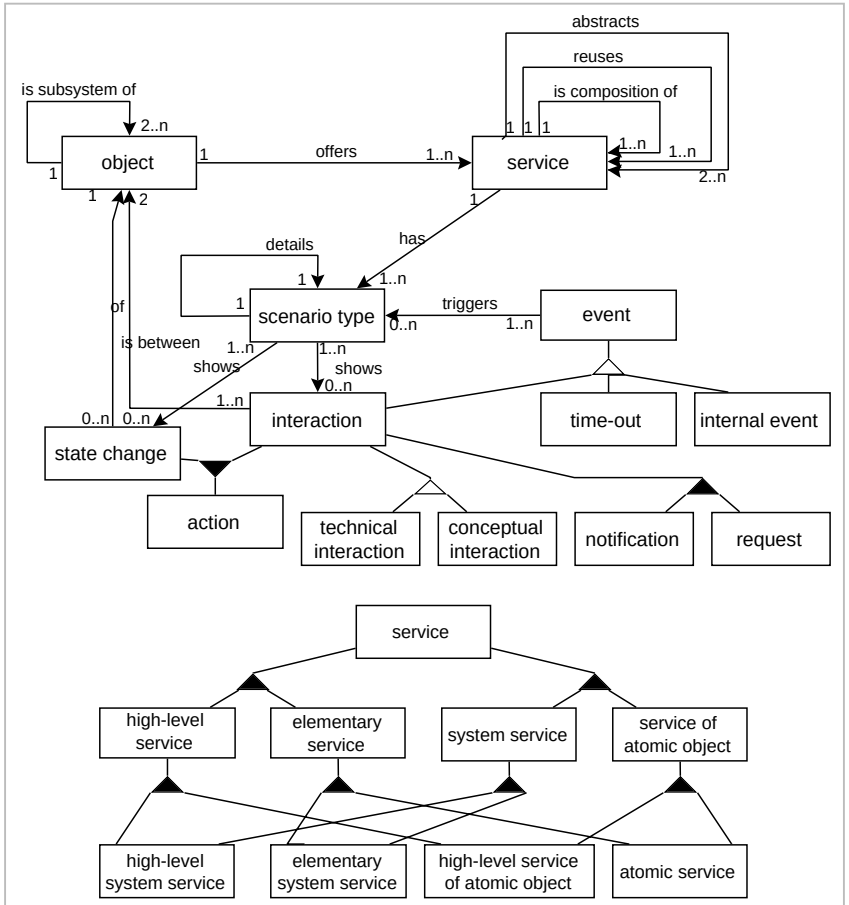
a. Abstraction: Complete aggregate and generalized services are abstractions of several other services. Generalized and complete aggregate services are always high-level services. Their specialised or element services may be high-level or elementary services.

b. Reuse: Extended and partial aggregate services reuse certain other services. Partial aggregate services are always elementary services. Extended services as well as the services that are extended may be both, elementary or high-level.

possible relationships between services	
...	**belonging to two different models**
	having same responsibility, same information flow and same server object
	The two services may be modelled equally, or their specifications my differ by varying degrees, e.g. they may have: - different names for the service, its interactions or their parameters - different position for a selection criterion - different algorithms (other interactions or other orders of interactions) - different internal composition (other component objects, other interactions among component objects, other component services) - one may be a simple or partial aggregate service, the other one a complete aggregate service - both are complete aggregate services, but their element services are not the same

Figure 67: Overview of the relationships among services

Our modelling approach has been based on the paradigm of systems of interacting objects. We use this paradigm on all the abstraction levels and for the internal view of systems as well as for the external view. As a consequence we introduced the distinction between the *services* and their *scenario types*, between *technical* and *conceptual* interactions, between *requests* and *notifications*, between *high-level* and *elementary* services and between *system services* and *services of atomic objects*. Figure 69 gives an overview



NOTE: - A service that is a complete aggregation or a generalisation is a high-level service.
- Aggregation, specialisation and extension hierarchies are mutually orthogonal. A service may be a specialised and/or extended service, independent of being a simple, partial aggregate or complete aggregate service.

Notation: Arrows show relations between terms, triangles show specialisations (solid:disjoint).

Figure 68: Metamodel for SEAM

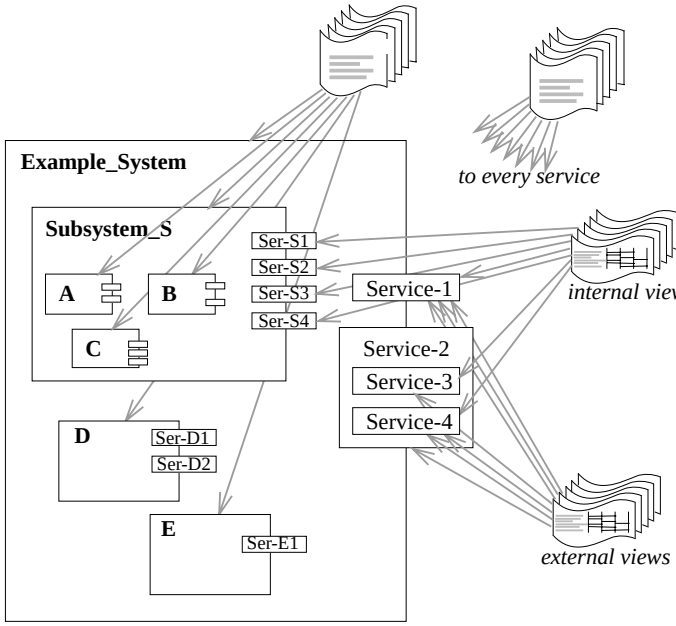
of these and of some other terms of SEAM. The metamodel in the figure 68 tries to express the most important relationships among the various terms.

Service	A **service** is a functionality offered by an object. The object may be a system, a subsystem or an atomic object. The properties of a service are given by its interactions and the possible orders of these interactions, the state changes it issues in the server object, the preconditions for the service, and the total information flow between the server object and its agents. See also page 95 and page 102.
State	Each object has a **state** which is given by the values of its internal components and its references to collaborating objects. We distinguish between the micro states, the macro states and the essential states of an object, definitions see page 156.
Scenario type	A **scenario type** specifies a set of possible sequences of interactions and/or state changes. In the case where a scenario type specifies a service offered by an object, the interactions are interactions between the server object and its agents and/or among the component objects of the server object, the state changes are state changes of the server object and of component objects of the server object, and the scenario type can be described by an interaction diagram for the internal and one for the external view of the service. But scenario types may also be used to describe arbitrary sequences of interactions, not only those of services. Interactions and state changes are also called **actions**. See also page 108.
Sever object - agents	The object offering a service is the **server object**. The objects the server object interacts with are its **collaborating objects** or its **agents**, they are also called **client objects** in the case of requests. See also page 102 and page 105.
Event	Services and scenario types are triggered by **events**. An event can be an interaction, a time-out, or any other kind of happening at a certain point of time and relevant to an object. See also page 103.
Interaction	An **interaction** is a message from one object to itself or to another object. It has a name and zero or more parameters. See also page 95.
Notification	A **notification** is an interaction after which the sending object resumes execution without waiting for any response. An interaction back to the sending object may or may not occur. See also page 105.
Request	A **request** is an interaction after which the sending object waits for a return interaction. A request and its answer are normally modelled by only one arrow. See also page 105

Technical event - conceptual event	When modelling **technical events**, we reflect the constructs of the program code. Technical events are always interactions. When modelling higher abstraction levels where we focus only on the concepts of the problem domain, we model **conceptual events**. Conceptual events can be arbitrarily high-level, and they can be interactions as well as other events. A conceptual event is either also a technical event, or it is a purely conceptual event. See also page 106.
Motive	The **motive** gives the reason or the motivation for initiating an interaction or another event. See also page 103.
Detailing scenario types	A scenario type may **be detailed** into another scenario type describing the same service by replacing some of its conceptual events by more detailed conceptual events or by technical events. These scenario types are said to be mutually **compatible**. See also page 129.
Composition of services	Services offered by systems or subsystems are **system services**. A system service is **composed of** services offered by component objects of this system. The composition of a service is shown in its internal view. See also page 134.
Elementary service - high-level service	**Elementary services** are those services of an object which are neither a generalisation nor a complete aggregation, i.e. from the external viewpoint they are no further specialised or completely decomposed into further services. Elementary services of atomic objects are **atomic services**. Non-elementary services are **high-level services**. See also page 137.
Complete aggregate service - partial aggregate service - simple service	A **complete aggregate** service is an aggregation of lower level services of the same object. A **partial aggregate** service is a service that at some point reuses another service of the same object. A **simple service** is a service that is neither a complete nor a partial aggregate service. The services which appear in a partial aggregate or complete aggregate service are called its **element services** (not to be confused with elementary services, a service can be an element service of another service without being an elementary service). See also page 136 ff.
Specialization of services	A service may be specialised into several **specialised services**. Their specifications contain more detailed scenario types, but each specialised service only covers part of the cases treated by the generalised service. See also page 142.
Extension of services	A service may have one or more **extended services**. The specification of an extended service reuses the specification of the original service, but changes one or more parts in the specification of its scenario types by replacing some or adding more actions. See also page 145.

Figure 69: Summary of some terms as used in SEAM

The scenario and object model are primarily documented by interaction diagrams and textual schemas. An example of how such a documentation set could look is shown in figure 70.



Each object (also the atomic objects *A*, *B*, *C*, *D* and *E*) is described by a schema which lists its services, references and internal components, and describes its life-cycle (either textually or by a separate diagram). Also each service is documented by a schema. Furthermore, the services of the object *Example_System* are each documented by one or more interaction diagrams for their external view (one for each scenario type). Also, all services in this examples with the exception of the atomic services have an interaction diagram showing their internal view.

In this example, the interaction diagrams are separate documents but logically they are still part of the schemas of the services, and these again are part of the schemas of the objects they belong to.

Interaction diagram: Schema of an object: Schema of a service:

Figure 70: Overview of the documentation of a scenario model

4.7.2 Reasons for the concepts as defined in SEAM

Why having services as well as events/interactions?

On the level of atomic objects and atomic services, where there is a 1:1 correspondence between triggering input interaction and service, we could just as well have eliminated the difference between events, interactions and services. Also when considering the external view of the system, we could ignore the differences between the interactions and the services by requiring that each triggering event on every abstraction level is an interaction to the server object, and by forcing a one-to-one correspondence between triggering event and service. This would also simplify the concepts. Each service would then consist of one triggering event that was an interaction, and zero or more output interactions. Such one-to-one correspondences can be found in the system operations of Fusion, the events in OORD [Umphress91] and the events and essential activities of MSA.

There are some good reasons why we have not chosen this approach:

- Not every service is triggered on every abstraction level by an interaction. On the abstraction level of the system as a whole there are services which are triggered by time-outs or other internal conditions[2]. We want to model these services without modelling any internal objects as artificial external agents (e.g. a clock), and without enforcing the use of interactions of the system with itself (e.g. a message *timeout()*).
- It is possible to build hierarchies of services by abstracting services. One service is detailed into several services. For interactions this is not possible. First, there are many cases where only a sequence of conceptual interactions can be detailed into (i.e. replaced by) a sequence of lower level interactions. Second, when detailing and abstracting interactions, this can change their direction. For instance, an input message from the agent to the system can become a request from the system to the agent, but we still model the service of the system and not the service of the agent.
- Especially when modelling the user interface, we have the advantage that interactions can be on a much finer level than services. Forcing a decomposition into services would not simplify the modelling (in fact, most methods do not offer the possibility of detailing their use case model until they actually model the user interface interactions). Moreover we could no more allow that non-atomic services may be triggered by more than one event type.

2. In MSA a difference is made between flow events and time events. But events are not equivalent to interactions and there is no clear mapping between these events and the interactions of an object-oriented system. Time events are used to capture services not triggered by any external agent. In OOSE only use cases are modelled that are caused by an input event from an external agent, a drawback that is also critiqued by [Firesmith95].

- We have adopted an all agent view, where we always consider all the agents of an object. We can abstract the interactions between one agent and the server object into a single input and a single output interaction. But we cannot include the interactions to other agents in this abstraction. The aggregation of services would thus stop as soon as a service would require inputs from several agents - a further abstraction of services would not be possible. We would end up at this level again with a flat list of services, determined by the criterion a) of chapter 2.1.3.3.

Interestingly enough, neither OORD, nor MSA, nor Fusion offer hierarchies of services. Also there is no transition defined from the conceptual events, used in for instance OORD, to the technical events as used in Fusion. In contrast, Graham [Graham93 and Graham94b] knows hierarchies of tasks (which correspond more or less to our services), but he does not model the actual interactions during the execution of a task. Tasks are just decomposed until the level of atomic services is reached: only there is the link made between tasks, the object offering the task and the message triggering the task.

Why having services as well as scenario types?

Why not say that a service is a scenario type? First, a service is a feature of an object, whereas a scenario type describes a sequence of interactions or actions. Second, the interactions involved in a service are just one property of the service, besides other properties we defined in chapter 4.2.1. Third, we allow detailing the scenario type without defining new services. One service can have several scenario types, which are mutually compatible and evolve from each other by detailing the information flow of the service. Fourth, in two different models we may have the same service, but with different and incompatible scenario types using different algorithms. Fifth, we allow services to be triggered by an interaction from an agent to the system (or server object), by a time-out or by an interaction form the system to an agent. We can allow this because we differentiate between the service offered by the server object and the interactions that take place between the server object and its agents.

Why making a distinction between conceptual and technical interactions and events?

As we have seen in chapter 2.1.2.3, most methods do not make this distinction. Some of them just leave it to the user to decide what kind of interactions they model with the notations they provide. Yet in most methods it is implicitly clear if the interactions and other events (or however they are called) signify technical events or conceptual events. Either these methods address only either requirements analysis or design, or they use different modelling techniques in analysis and design. In SEAM however, we want to use the same vocabulary and notations for the external high-level views of the system as a whole as well as for design details in any part of the system (problem domain subsystem as well as user interface subsystem). Thus, we need to make the distinction between conceptual and technical events explicit. Furthermore, making this distinction explicit also in the notation has the following advantages:

- We can show explicitly the transition from highly conceptual interactions between a user and a system down to single user inputs. This allows us to bridge the gap between an analysis model considering only higher level interactions and a design model using technical interactions.
- We can model the user interface on the level that is appropriate for this specific user interface. Within one diagram we can use technical interactions for the communication between the problem domain objects, and a mix of technical and conceptual interactions for the communication with the user interface objects and the user.[3]
- If necessary, we can choose to show the external view (interactions of the user with the system) and the internal view (interactions between subsystems) of a system on various abstraction levels.

Why having hierarchies of services?

We can summarize the reasons for having hierarchies in the following five points:

- We need hierarchies in the final model to represent systems subdivided into subsystems (a software system into several software components or an arbitrary system into software systems and manual systems) and to allow the change of system boundaries.
- We need hierarchies to represent system services that are inherently structured.
- We need hierarchies in the final model in order to be able to present various views of the system that are on several abstraction levels.
- We need hierarchies during the development process, because we understand the development process as an exploring and learning process. Many transient models are made, many intermediate abstraction levels are developed and later are neglected, many different hierarchies are tried out and changed into better ones. We want to support this process explicitly by our modelling technique.
- We need hierarchies because they match our perception of the services of a system and thus help to communicate our ideas concerning requirements and design among developers and users.

3. In the interaction diagrams of the design, Fusion does not model the user interface objects at all. OOSE shows the user interface objects, the interactions must be modelled on the technical level. BON uses a special arrow to show the interactions between the user and the problem domain objects, neglecting the effective interaction mechanisms and user interface objects.

Why having aggregation as well as composition hierarchies?

We have differentiated between the aggregation of services and the composition of services. In contrast to the aggregation hierarchy that aggregates services of one object into higher level services of the same object, the composition hierarchy composes services of different component objects and is linked to the composition of objects into systems. Other methods do not make this differentiation. For example the tasks of Graham [Graham94b] are decomposed independently of the objects, only on the level of atomic services is the link between a service and the object offering it made.

There are two reasons why we have chosen the approach of defining a service as being always offered by an object and never specify a service independent of any object:

- We adopted a pure and simple object- and component-oriented approach: objects offering and using services. Therefore we do not introduce different kinds of objects (e.g. event objects, service objects), and we do not define and refine objects and functions independently.
- We wanted to enable a high level decomposition of the system, dividing up large systems into subsystems. And we wanted to use for this only one single modelling approach, which works for dividing up a system into manual and software systems, for dividing up a software system into software components, for reusing existing components and for encapsulating sensible parts of a system.

As a consequence we had to differentiate between assembling services within the same object (aggregation) and assembling services and objects to a subsystem (composition).

Why having aggregation as well as inheritance hierarchies?

Other methods such as UML [Booch95] and Glinz [Glinz95] have aggregation hierarchies but not inheritance hierarchies. We have decided to provide both inheritance hierarchies and aggregation hierarchies, and also to distinguish between abstracting (complete aggregation and generalisation) and reuse (extending and partial aggregation). As we have seen in chapter 4.5, models using one technique can be transformed into models using another technique. It would not have been necessary to provide all different kinds of hierarchies, even more so as we also offer the possibility of detailing scenario types within one service. The reasons why we provide all of it are expressiveness and communication. We want to provide developers and users with those techniques that enable them to make a model that matches best how they perceive a problem or see a solution. This also allows us to support the presentation and communication of requirements and design ideas among all the project team members[4].

4. We therefore do not require that the services and scenario types have to be transformed into any special form before they can be modelled. This in contrast to e.g. [Glinz95], where any overlapping scenarios within a behavioural model first must be transformed into a set of disjoint scenarios.

As no language supports inheritance between services within the same object, using specialisation or extension of services is usually inadequate for lower level models. Exceptions are those cases where the specialisation of system services goes hand in hand with the specialisation of some component objects. For high level views, especially when eliciting requirements, the specialisation of services is a mighty technique whenever we first of all perceive services as generalized and specialised services. Extended services are especially helpful if further requirements are added to an existing model. It would limit the expressiveness if we forced users and developers to model the services without a specialisation hierarchy. Even worse, sometimes a decomposition into elementary services is not really feasible before all the different variants of the service have been explored, and this is best done in defining specializations. And it would lead to unnecessary reworking of the model if we did not offer extended services. Therefore we suggest using specialisation and extension hierarchies when finding and discussing the requirements. At a later stage of the project, they may be transformed into aggregation hierarchies and the final model may use only aggregation and composition hierarchies on all abstraction levels.

Why not just some predefined levels?

SEAM allows to model scenarios on various abstraction levels without changing the concepts and the notation and without having some predefined abstraction levels. This in contrast to other methods such as [Armour95] or [Regnell96] (for more details see chapter A.9.8), where some predefined abstraction levels are given, and where in the case of [Regnell96] even the concepts and notations change. These approaches do have the advantage that the user is no more forced to decide how many and which abstraction levels he needs and what the goals of his models really are. He must also no more decide how much he wants to model on which abstraction level and how complete and precise the descriptions should be. But having predefined abstraction levels has the disadvantage that the predefined levels are well suited for one specific application domain and development environment, but are counterproductive in another project. Therefore SEAM contains general enough concepts that can be adapted. Furthermore, having several notations is contrary to another goal of SEAM which is to have only one set of concepts and one notation which can be used on all abstraction levels.

Chapter 5
Case Studies

The following two case studies, a mail order firm (chapter 5.1) and a storage control system (chapter 5.2), show the use of the extended scenario modelling technique SEAM. We will try to mimic a possible development process showing the changes the model undergoes during the modelling process. Not every modelling element defined in chapter 4 will be used, and we only present a subset of the scenario models; other views of the object model are almost completely omitted.

5.1 Mail Order Firm

This case study shows the modelling of a software system for the handling of the orders in a mail order firm. We assume that there exists already a software system responsible for bookkeeping. We further assume that neither models showing the business processes of this mail order firm nor precise ideas concerning effective requirements and interfaces of the new software system exist. An iterative development process is chosen; we mimic this development process by subdividing the diagrams presented here into a first, second and third round.

First round: the external view of the whole company

Figures 71 and 72 show the context diagram and the object schema of the system *Mail_Order_Firm*.

Name of object type	Mail_Order_Firm Encompasses the whole company (also non-software components).
Services	• order • make_order_for_suppliers • delivery_of_supplies

Figure 71: Schema of the system *Mail_Order_Firm*

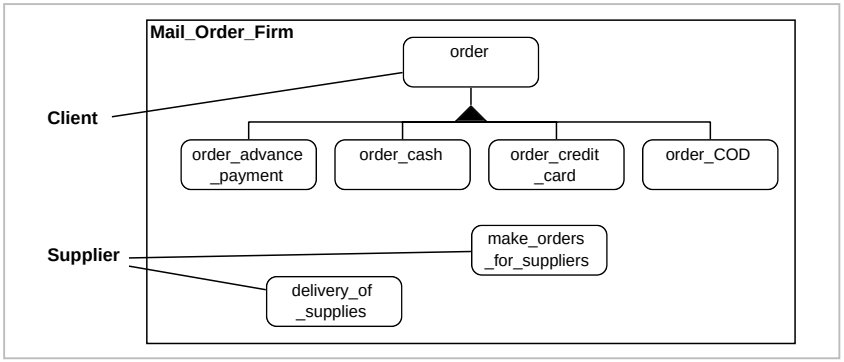


Figure 72: Context diagram of the system *Mail_Order_Firm*

All the services offered by the system *Mail_Order_Firm* are specified by schemas (figure 73). For each service one scenario type is modelled by an interaction diagram showing its external view (figures 74 and 75). The service *order* is modelled as a specialisation hierarchy. Because in this project the interaction diagrams for the external view are very trivial, these diagrams would probably be discarded later when the internal views have been modelled. Nevertheless they are very valuable for eliciting the requirements with the user. The provisional data model containing the specification of all the interaction parameters is not shown here.

Name of service	Mail_Order_Firm :: make_orders_for_suppliers • is elementary service Makes the orders for the deliveries of the suppliers.
Description	Each night, the system checks automatically the stock of the items. Whenever an item is no more available in the desired quantity, it is put on the order for the appropriate supplier. The orders are printed out, get signed by the responsible manager, and are sent out.

Name of service	Mail_Order_Firm :: order • has specialisations Handles the ordering and delivering of items ordered by clients.
Description	A client orders some items at the mail order firm. He states if he wants to pay by cash (i.e. collect the items at an agency of the firm), by credit_card, by collect on delivery (COD) or by advance payment. Depending on the kind of payment, the service is executed slightly differently.

Figure 73: Schemas of the services *make_orders_for_suppliers* and *order*

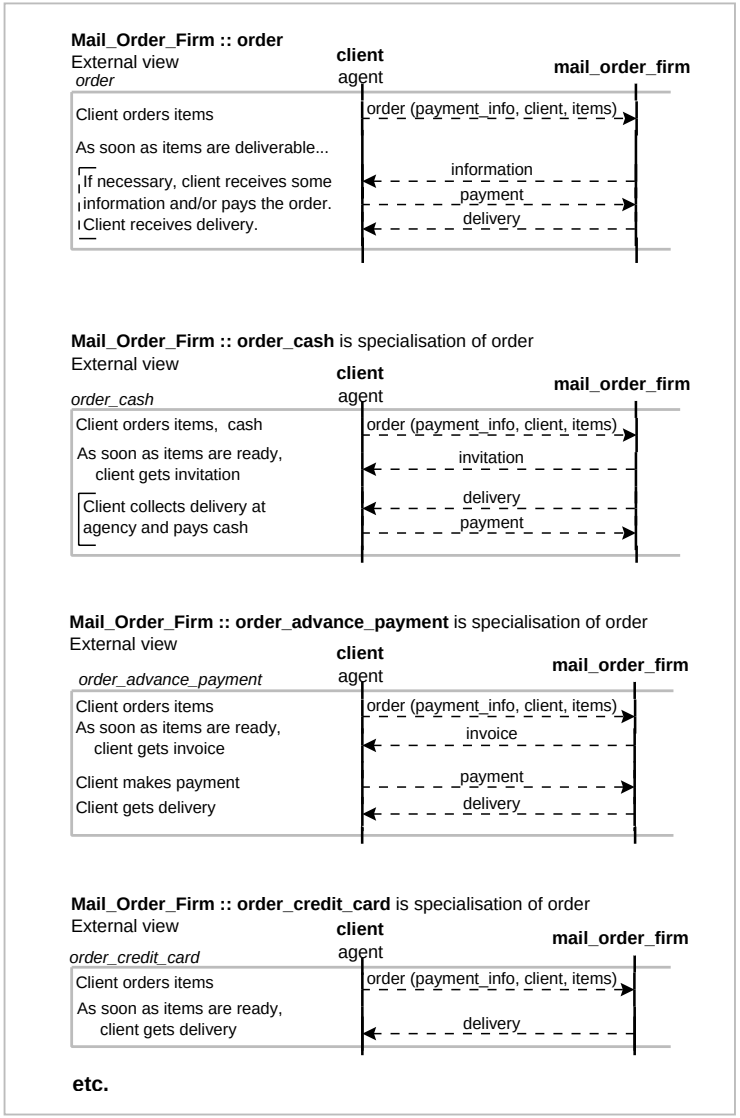


Figure 74: Specialisation hierarchy of the service *Mail_Order_Firm :: order* (external view)

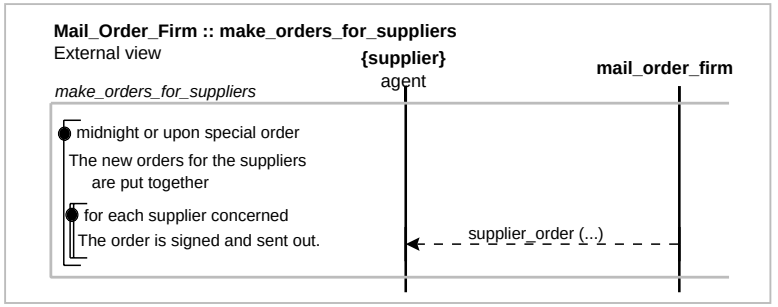


Figure 75: Service *Mail_Order_Firm :: make_orders_for_suppliers* (external view)

Second round: the internal view of the object *Mail_Order_Firm*

In the second round, we define the subsystems of the system *mail_order_firm*, specify its life-cycle, and refine its object schema (figure 76). Not all the subsystems are software systems. For each service of the system *mail_order_firm* we model the internal view by an interaction diagram; figures 77, 78 and 79 show the internal view of three of the four specialisations of the service *order*. These interaction diagrams also go into more detail concerning the interface between the system *mail_order_firm* and the agent *client*.

Name of object type	Mail_Order_Firm • is a white-box system Encompasses the whole company.
Services	• order • make_order_for_suppliers • delivery_of_supplies
Component objects	• order_system, is a software system • bookkeeping, is a software system • stock, is a manual process • several agencies, are manual processes
Life-cycle	(order ||*) || (make_order_for_suppliers*) || (delivery_of_supplies*)

Figure 76: Second schema for the system *Mail_Order_Firm*

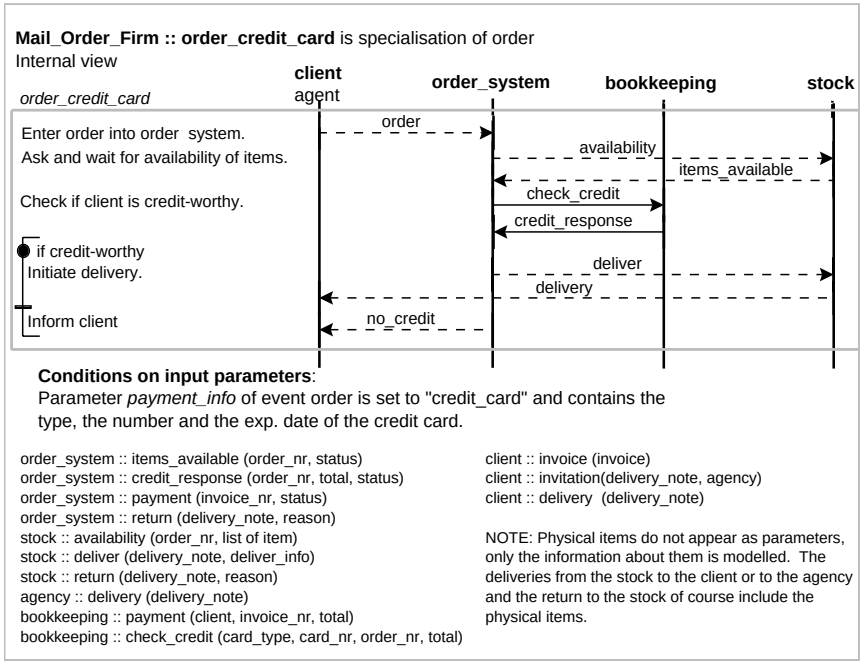


Figure 77: Internal view of the specialisation hierarchy for the service *order,* part 1

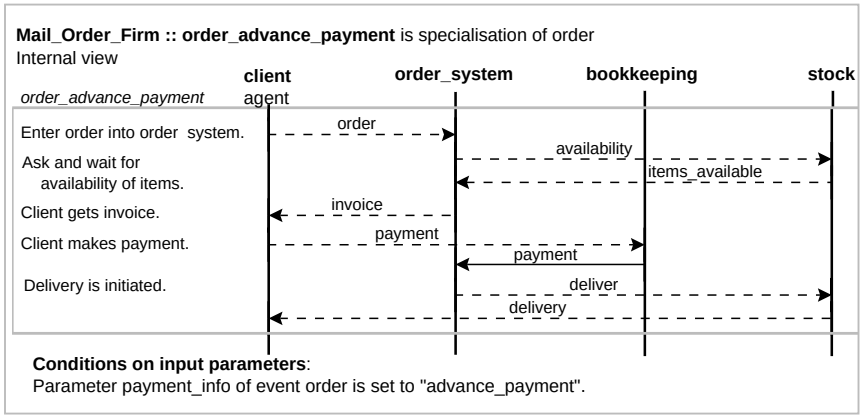


Figure 78: Internal view of the specialisation hierarchy for the service *order,* part 2

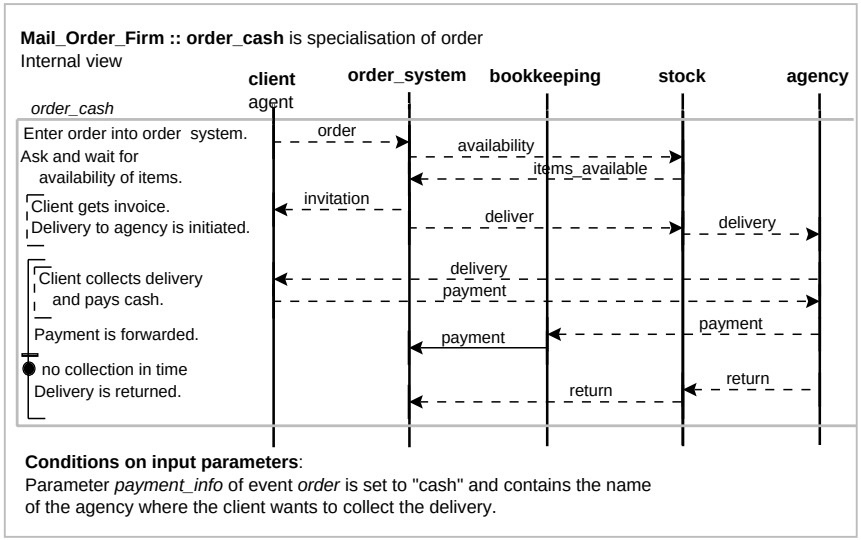


Figure 79: Internal view of the specialisation hierarchy for the service *order,* part 3

First alternative model

In the modelling workshops, alternative models are also proposed. In the first alternative model for the service *mail_order_firm :: order*, the interaction *order (payment_info, client, items)* of the generalised service is replaced in the specialised services by the interactions *order_cash (agency, client, items)*, *order_advance_payment (client, items)*, *order_COD (client, items)* and *order_credit_card (card_info, client, items)*.

Second alternative model

Another proposal is to transform the specialisation hierarchy into a single service that includes all different ways of payment. Figure 80 shows the corresponding interaction diagram.

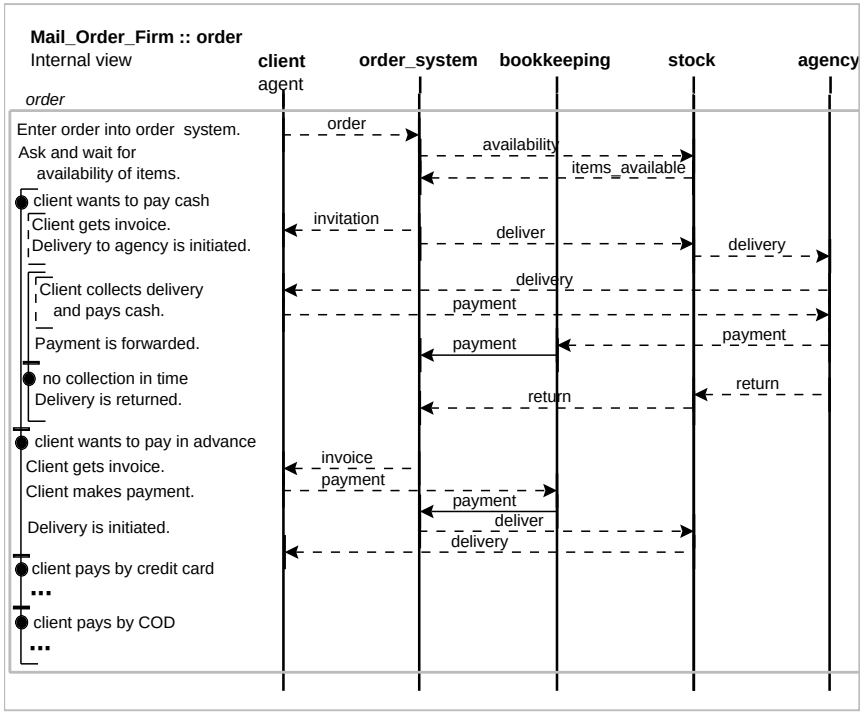


Figure 80: Internal view of the service *order* without specialisation

Third round: the external view of the subsystem *Order_System*

Before continuing modelling it must be decided which one of the alternatives for the service *order* is chosen. For this case study we choose the specialisation hierarchy. In the third round we focus on the external view of the services of the subsystem *Order_System*. A high-level specification of the external view of these services is already contained in the internal view of the services of *Mail_Order_Firm*. But in contrast to the interaction diagrams of the internal view of the services of *Mail_Order_Firm*, the interaction diagrams of the external view of the services of *Order_System* do not show any interactions between the objects *client*, *stock* and *bookkeeping*, because they are only agents. Also, we decompose the services of *Mail_Order_Firm* down to elementary services. Figure 81 shows the interaction diagram of the internal view of *Mail_Order_Firm :: order_advance_payment*, figure 82 contains the specification of the subsystem *Order_System*.

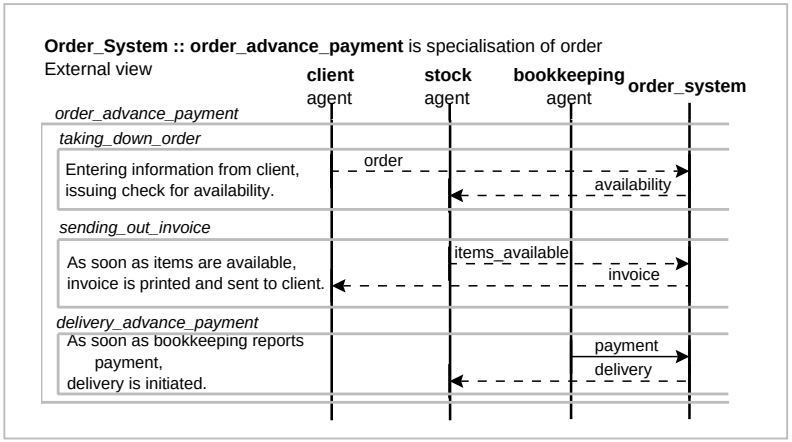


Figure 81: External view of the service *order_advance_payment* of the subsystem *Order_System*

Name of object type	Order_System • is a white-box subsystem Order_System is the software application that handles all the orders.
Services offered	High-level services: • order • order_advance_payment, is specialisation of order • ... Elementary services: • taking_down_order • sending_out_invoice • delivery_advance_payment • ....
Life-cycle	order ||*
Component objects	....

Figure 82: Schema of the subsystem *Order_System*

Fourth round: the internal view of the subsystem *Order_System*

Figure 82 shows part of the object model of the subsystem *Order_System*. Every component object (atomic objects as well as subsystems) of *Order_System* is also described by an object schema. Figure 84 contains the schema for the object *Order. Order* is an abstract object type and offers the atomic services *init_order*, *set_client*, *payment* etc. (we

assume that *Order* and its subtypes are atomic objects). The subtypes redefine these services and add further services, for instance the subtype *Order_Advance_Payment* adds the service *print_invoice.*

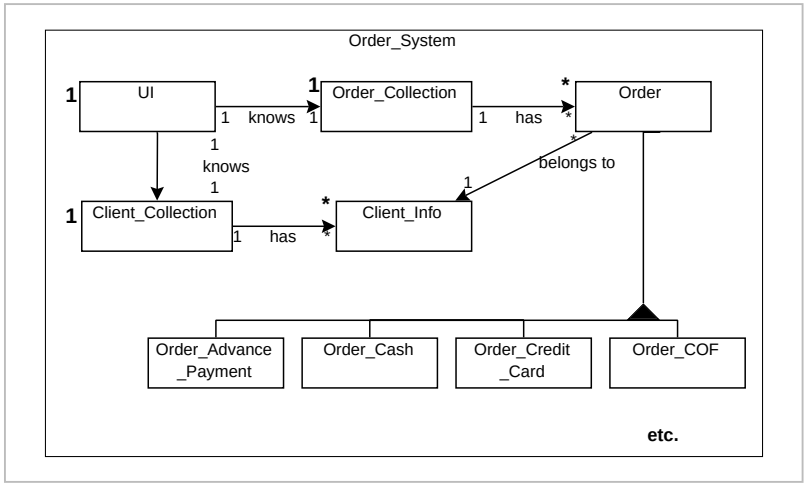


Figure 83: Object model of the subsystem *Order_System*

.

Name of object type	Order • is an atomic object • is abstract, has subtypes Handles all the information for one specific order, from the request from the client until delivery has taken place.
Services offered	Atomic services: • init_order • set_client • payment • ....
Life-cycle	see subtypes
Services used	• ....
References	• for_client : Client_Info, * -- 1
Attributes	• ....

Figure 84: Schema for the object *Order*

Finally, all the elementary services of the system *Order_System* are described by a schema and by at least one internal interaction diagram. Services of component objects may be further specified by schemas. We conclude here the case study with figures 85, 86 and 87 showing the schemas and part of the internal view of the elementary service *Order_System :: taking_down_order* and of the atomic service *Order :: set_client* (because *set_client* is an atomic service we can list the parameters with the name of the service in the schema and do not make an interaction diagram for the service).

Name of service	Order_System :: taking_down_order • elementary system service The order of the client is entered into *Order_System.*
Description	One order is entered for one client. If the client does not exist yet in the system, he is created.

Figure 85: Schema for the elementary service *taking_down_order*

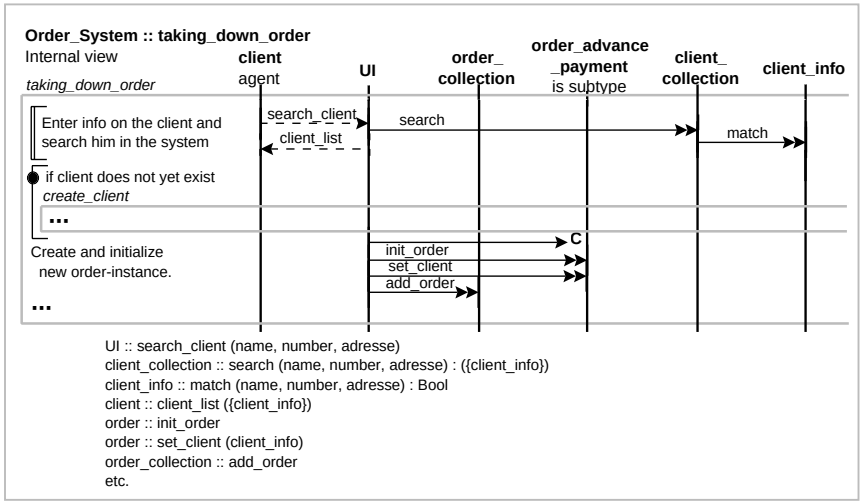


Figure 86: Internal view of the service *taking_down_order*

Name of service	Order :: set_client (c : Client) : () • atomic service Sets the reference *for_client* to the client that makes this order.
Description	The reference *for_client* is set to c. No preconditions.

Figure 87: Schema of the service *set_client*

5.2 ECO-System

The following example shows part of the scenario model of a system for the administration of a drum storage. Drums with environmentally damaging chemicals are to be stored in various storage buildings. The drums are delivered to a loading bay where a clerk enters a delivery manifest into the computer system. The transport system Drum_Storage takes the drums from the loading bay and puts them into the various buildings. The software system, we call it ECO-system, determines which drums are to be stored in which buildings and assigns a drum identifier to each drum. As the drums contain dangerous chemicals, special rules apply as to how the drums may be distributed onto the various storage buildings, for instance depending on the type of chemical contained in a drum. The example of the ECO-system is taken from [Coleman94], with slight changes in the problem statement and in the design.

First-cut requirements model

Figures 88 and 89 show the context diagram and the object schema of the whole system *ECO-System.* Figures 90, 91 and 92 contain some of the service schemas for the system services of *ECO-System.* Very abstract external views of the scenario types for the services *ECO-System :: deliver_drums* and *ECO-System :: get_status* are shown in figures 93 and 94.

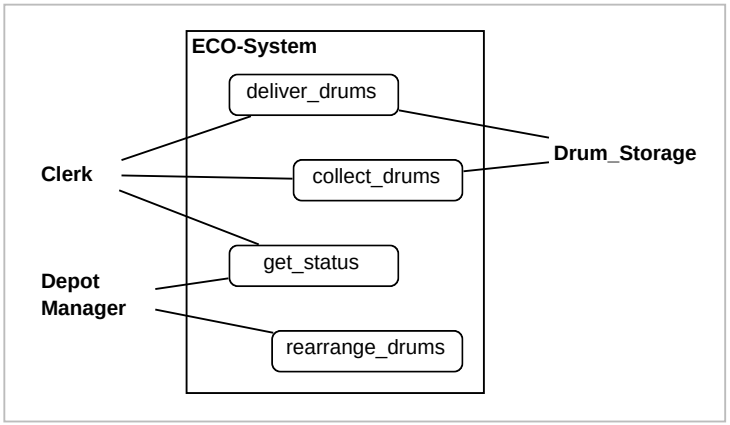


Figure 88: Context diagram of the system *ECO-System*

Name of object type	ECO-System • is a white-box subsystem The ECO-System controls the storage of environmentally damaging chemicals.
Services offered	High-level services: • deliver_drums • collect_drums • get_status • rearrange_drums
Life-cycle	(deliver_drums | collect_drums | rearrange_drums)* ‖ get_status*

Figure 89: Schema of the system *ECO-System*

Name of service	ECO-System :: rearrange_drums Rearrange drums in order to get a better distribution.
Description	The system checks the current status and tries to find a better distribution. The present free capacity, the free capacity after rearranging, and the number of drums to be moved are displayed to the user. The user decides if this is worth while. If the user wants to, the rearrangement takes place. A better distribution is one that allows that for each drum type a maximum of new drums can be stored. The best triple of maxima is determined based one statistics of past deliveries.

Figure 90: Schema of the service *rearrange_drums*

Name of service	ECO-System :: deliver_drums Check in a new delivery of drums.
Description	Checking in a delivery of drums includes the following steps: • If the load-bay is empty, the user can enter the contents of the manifest accompanying the delivery. • The user checks in each drum by entering its type and issues an identifier for the drum. • When all drums are checked in, then the system computes in which store buildings the drums are to be stored and sends the allocation to the system *Drum storage*. It may happen that not all drums can be stored. In this case, the system tells the user (by giving him the identifiers) which drums cannot be stored and must be returned from whence they came.

Figure 91: Schema of the service *deliver_drums*

Name of service	ECO-System :: get_status Get information on the status of the system.
Description	The following information can be asked for: • if the system is vulnerable or not (i.e. two neighbouring buildings contain the maximum permitted number of drums) • an overview of the drums that are stored in specific buildings

Figure 92: Schema of the service *get_status*

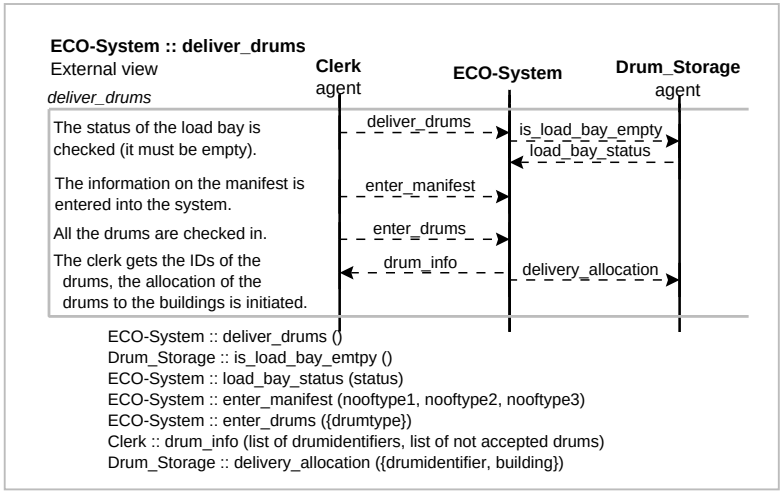


Figure 93: External view of the service *deliver_drums*

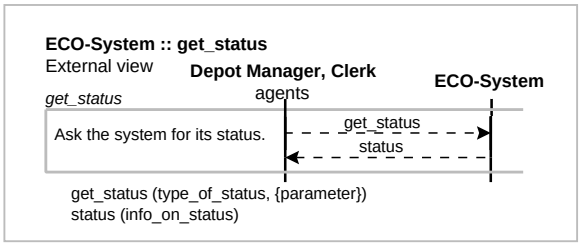


Figure 94: External view of the service *get_status*

Requirements analysis

The first-cut requirements model is further analysed, reworked and detailed. A provisional data dictionary is made with a description of all the parameters used in the interactions (not shown here). The services are elaborated by additional scenario types that are more detailed, some of the services also become partial or complete aggregate services instead of simple services. Such a more detailed scenario type for the service *ECO-system :: deliver_drums* is shown in figure 95.

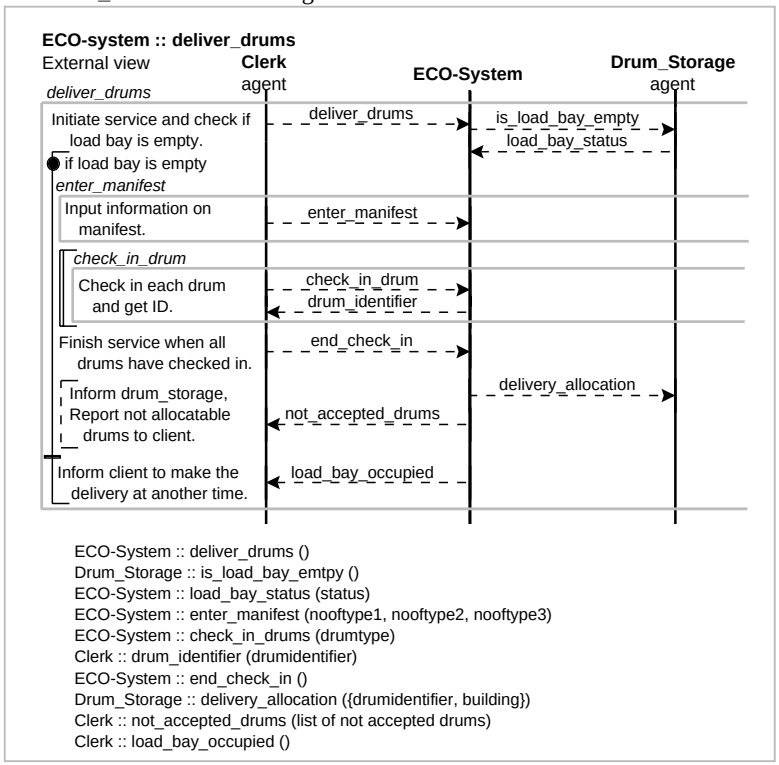


Figure 95: The service *deliver_drums* as a partial aggregation

In a further review, some users discover that the service *ECO-system :: deliver_drums* is not complete; a special case which the users consider as essential is missing. Therefore they propose an extended service which they call *deliver_drums_exception*. Figures 96 and 97 show its service schema and its interaction diagram. Furthermore, the line "has extended service deliver_drums_exception" is added to the schema of the service *deliver_drums*. In order to gain a better overview of the services defined so far, the users also make aggregation and inheritance graphs for the service *deliver_drums*; these service graphs are shown in figure 98.

Name of service	ECO-System :: deliver_drums_exception • is extended service of deliver_drums Checking in of a new delivery of drums, with handling of errors in the manifest.
Description	Conditions for deliver_drums_exception: The total number of checked in drums does not correspond to the total on the manifest. Consequences: After the user has checked in all drums as usual, he is informed about the discrepancy. He may choose to either abort or continue delivering. In any case, the divergence is stored by the system for later inquiries.

Figure 96: Schema for an extended service

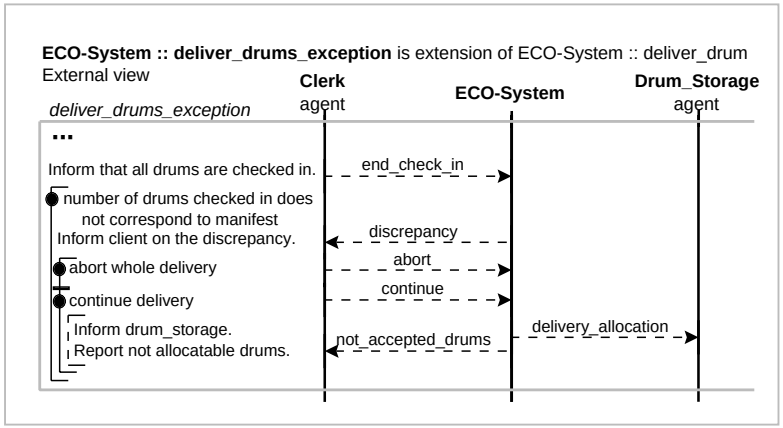


Figure 97: Scenario type for the extended service *deliver_drums_exception*

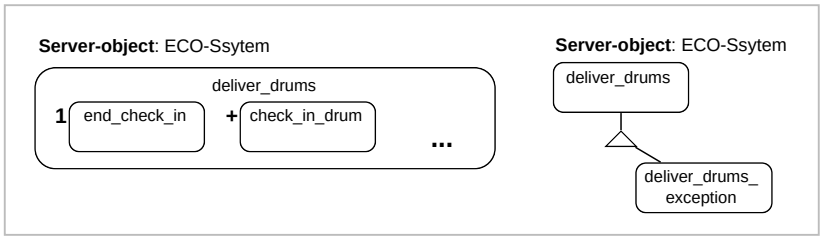


Figure 98: Aggregation and inheritance graphs of the service *deliver_drums*

When reviewing the models made so far, the system designer reworks the service *deliver_-drums* into a complete aggregation. He also integrates the extended service *deliver_-drums_exception* into the service *deliver_drums*. Figure 99 contains the interaction diagrams of the aggregate service and of some of its element services. Figure 100 shows the new service graph.

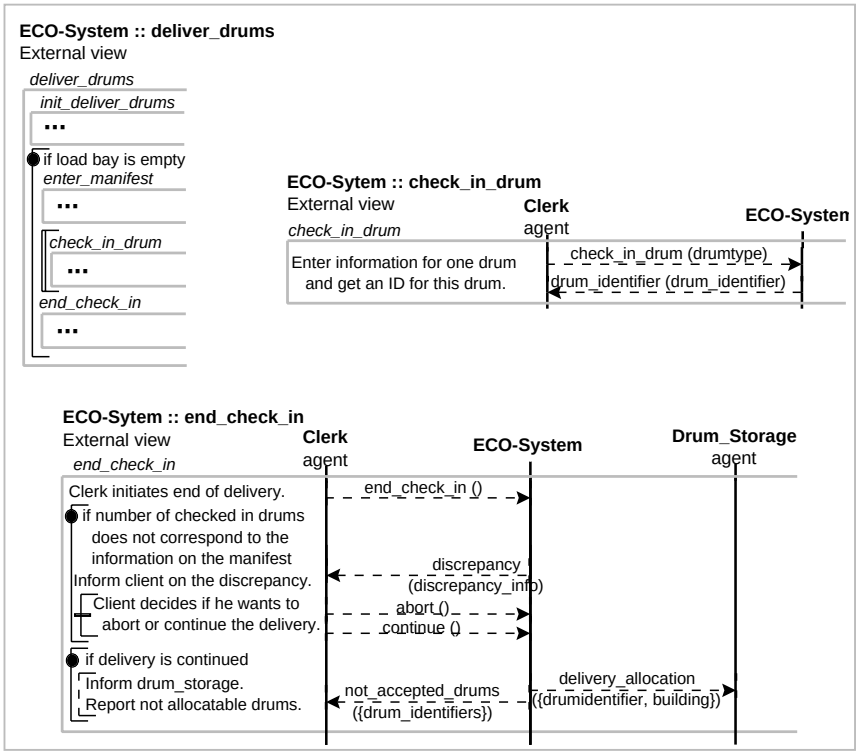


Figure 99: The service *deliver_drums* as a complete aggregation

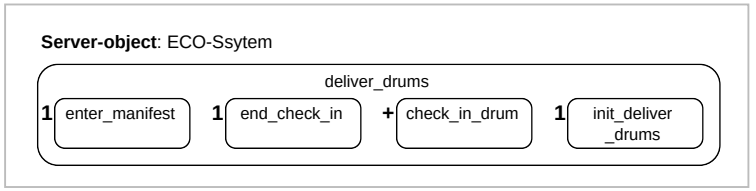


Figure 100: Aggregation graph of the service *deliver_drums*

The life-cycle of the object *ECO-System* was already specified in its schema (figure 89) by regular expressions. Another possibility would have been to specify the life-cycle by pseudo-code as in the left diagram in figure 101. Once the service *deliver_drums* is modelled as a complete aggregate service, the left diagram can of course be automatically extended into the right diagram.

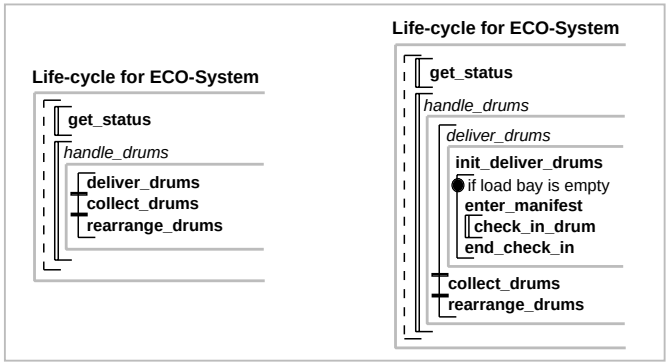


Figure 101: Life-cycle of *ECO-System*

System design

Besides other things it is decided that the system will be realised with a graphical user interface. After determining the user interface, the external view of the system is further detailed. In the case of the service *get_status* and *check_in_drum* further events are introduced which correspond to the actual interactions between the user and the system (figures 103 and 102). As soon as it will become clear that these interactions can be implemented like this, they could be transformed into technical interactions and the diagrams in figures 103 and 102 could be replaced by diagrams showing only technical interactions.

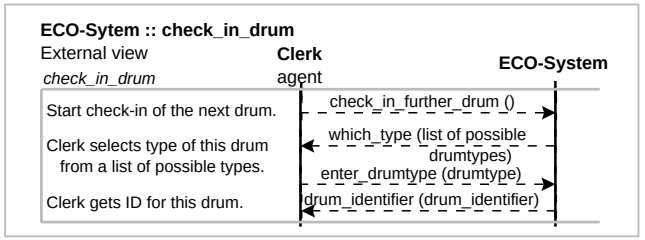


Figure 102: Service *check_in_drum*: user interface design

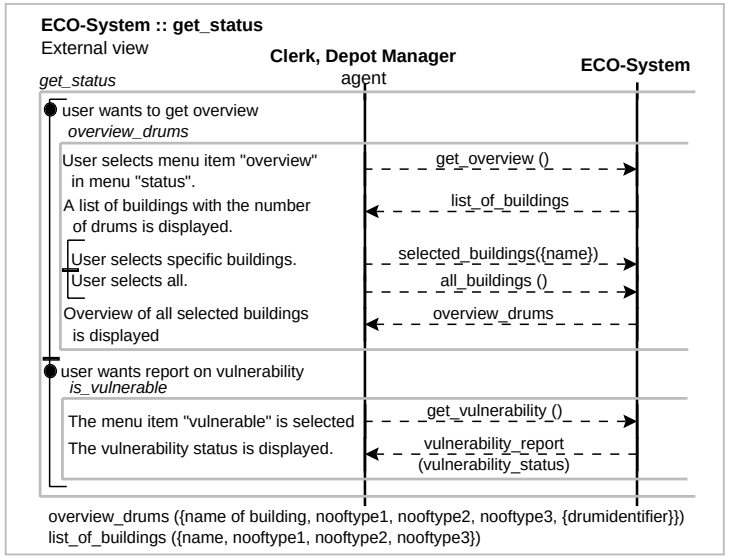


Figure 103: Service *get_status*: user interface design

High-level internal design

In the high-level internal design of ECO-system, all objects responsible for handling user inputs and outputs (i.e. all the pairs of controller and view objects) are assembled into a group *UI*. This group and its services will be further detailed in the detailed design, which is not shown in this case study. In the interaction diagram of figure 104 we show the high-level design of the internal view of the elementary system service *ECO-System :: overview_drums*. Finally, in figures 105 and 106 we give the schemas of two of the atomic component services of the system service *overview_drums*.

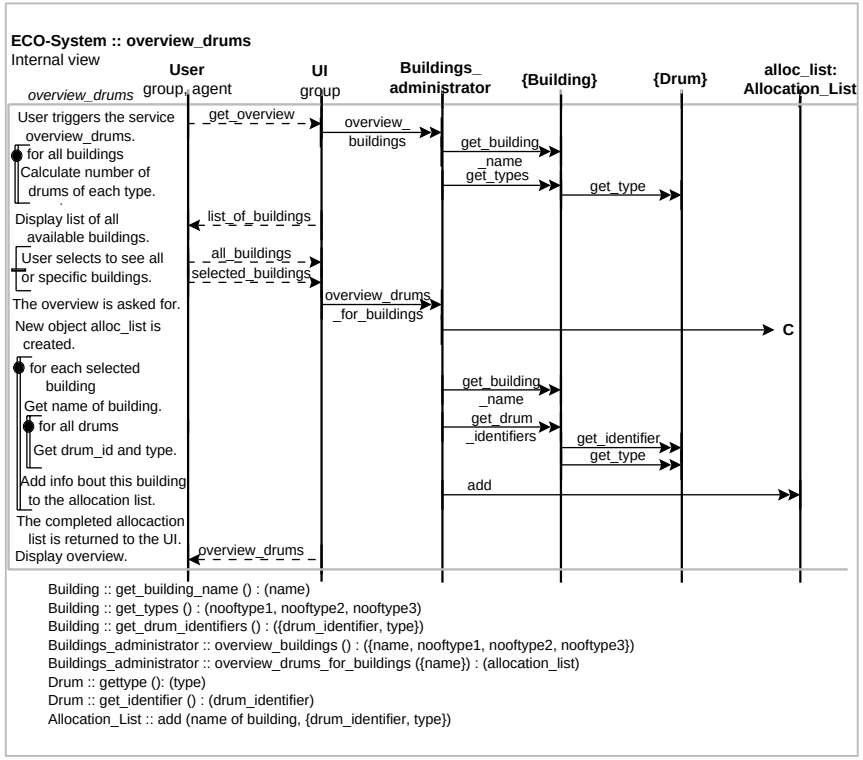


Figure 104: Internal view of the service *overview_drums*

Name of service	Buildings_administrator :: overview_drums_for_building ({Building}) : Allocation_List - is an atomic service Returns a list of drums of the designated buildings.
Description	Input: - name of one or more buildings Output: - allocation list, which contains the identifiers and drum types of all the designated buildings, sorted according to building and drum type For the designated buildings all the drums are searched and asked for their identifiers and types. A list of them is returned. In case that one of the provided names of buildings is not a valid name of an existing store building, an empty list of drums is returned for this building.

Figure 105: Schema for the service *overview_drums_for_building*

Name of service	Building :: get_types (): (nooftype1, nooftype2, nooftype3) - is an atomic service Returns the number of stored drums.
Description	Output: - nooftype1: number of drums of type 1 stored in this building - nooftype2: ditto - nooftype3: ditto All drums contained in the building are asked for the drum type they have. The numbers of these types are added up and returned.

Figure 106: Schema for the service *get_types*

Chapter 6
Summary and Outlook

In this thesis we started our discussion of modelling global behaviour by asking if and how the weaknesses of the matrix approach can be overcome (chapter 1). Following an overview of the concepts used in modelling global behaviour, we described the characteristics summarized under the term matrix approach (chapter 2). We identified three main sources for difficulties that may arise with the matrix approach: *i)* the limited expressiveness of the scenario model because of its restriction to one abstraction level and its lack of modelling elements for the relationships among scenario types; *ii)* the direct derivation of the object model from an initial data model; *iii)* the pursuit of contradicting goals for the analysis model and the modelling process. As a consequence we proposed an enhanced scenario modelling technique which we call SEAM (chapter 4). This modelling technique overcomes some of the weaknesses of the matrix approach. In the following we summarize the strengths of SEAM with respect to the criticisms of the matrix approach as discussed in chapter 2.2.2:

The first criticism of methods based on the matrix approach was the flat list of scenario types. SEAM allows us handling redundancy across different scenario types by having partial and complete aggregate services as well as enhanced services. It allows the modelling of specialisation hierarchies as such, and it shows dependencies among services by specifying complete aggregate services and by modelling also the life-cycle of the system, the subsystems or the atomic objects.

The second criticism was that the matrix approach supports the modelling of only one abstraction level. SEAM has no such limitation. Scenarios and services can be modelled on an arbitrary number of abstraction levels which are well interconnected, and the scenario model can be gradually refined.

The third criticism was that the matrix approach supports only the modelling of monolithic systems. In SEAM we have overcome this limitation by introducing the notion of subsystems.

The fourth criticism referred to difficulties arising in the transition from the analysis to the design model. Even when using SEAM, one could end up with a design object model

that is data driven and/or function driven. But SEAM helps to avoid this situation by clearly distinguishing between the provisional data model and the object model, by emphasizing the difference between the initial analysis model and the final high-level view, and by encouraging a one-model approach where the final model is achieved in an iterative process. This allows changes to the object structure until a high-quality design model is reached. While the perception of the system changes during this process, the initial scenario model is adapted in step. Thus the discrepancy that would otherwise appear between the scenario models of analysis and design, and the difficulties in traceability between the two, are avoided. Because in SEAM system services may be modelled on various abstraction levels and from both an internal and external viewpoint using the same notations and concepts, the gap between the specification of the external high-level view of the system services and the specification of the internal object operations can be avoided. This is achieved by specifying also scenario types on the level of the actual user interactions.

The last criticism in chapter 2.2.2 was that the matrix approach models a matrix between functions and data elements, and not a network of interacting objects that are encapsulations of functions and data. In contrast, SEAM uses the paradigm of interacting objects which are encapsulations of functions and data for the scenario model as well. Each service is offered by an object which may be a white-box or a black-box encapsulation. There is a clear distinction between the external view of an object and its internal view, and this distinction is preserved throughout all the concepts and notations. Of course, in SEAM we can also establish a matrix between the services and the object types that are changed by them or are used as parameters. But in contrast to the matrix approach, neither the services nor the objects are necessarily modelled in a flat list (hierarchies of services, systems of objects) and the services are always linked to the objects that offer them.

Certain weaknesses of the matrix approach cannot be overcome just by offering an enhanced modelling technique. Contradicting analysis modelling goals need to be avoided by an adequate project planning process. Decisions concerning the optimum abstraction levels for a given model and the level of completeness to be achieved are still up to the people involved in the modelling process. And some conflicts, for example the desire for an easy controllable process versus the non-existence of stable models, or between the advantages of informal modelling techniques and the advantages of formal languages, cannot be resolved, since they are inherent to software development. They can only be made known to developers instead of just being ignored. For their successful handling as well as for many other stumbling blocks mentioned in this thesis, no universally valid remedies can be given. This necessitates deploying software developers who are aware of the potential difficulties and know enough about software engineering in order to be able to react properly when problems and risks arise.

Contributions

I consider the following items as my main contributions to the field of object-oriented software development:

- an ***overview of the basic concepts*** of modelling global behaviour with:
  - the analysis and description of *classification schemes* for scenario types, including a set of possible criteria for grouping scenario instances into scenario types and a set of possible criteria for determining the end of a scenario
  - the differentiation between modelling *technical and conceptual events*
  - a *summary of various methods* which are chosen and described from the point of view of modelling global behaviour
- the analysis and description of the characteristics and drawbacks of the ***matrix approach***
- the description of two kinds of ***intent clashes in the goals of analysis modelling***, and the classification of software development methods into three categories concerning their approach to these intent clashes
- an ***enhanced scenario modelling technique*** that includes concepts and notations for:
  - *systems, subsystems, object groups* and *atomic objects* offering services
  - the distinction between the *services* and their *scenario types*
  - *technical* and *conceptual* interactions, *requests* and *notifications*, *internal* and *external views* of services
  - the *detailing of scenario types* (reflecting relationships between scenario types of the same service)
  - *composition, aggregation, specialisation* and *extension hierarchies of services* (reflecting relationships between services within the same scenario model)
  - *transforming services or scenario types* (reflecting relationships between services or scenario types of different scenario models)

Outlook

Based on this thesis, research could continue in the following areas:

- **Industrial pilot project**: So far, the proposed enhanced scenario modelling technique has only been used for small case studies. The next step would be to evaluate this approach in industrial projects, i.e. in projects with a higher degree of complexity, with real users, where the result of the modelling effort is not known from the beginning, and where changes in the perception of the system can appear. For such a pilot project, SEAM as well as a prototype CASE-tool would need to be integrated into the methods and tools that would be already in use in that particular project.
- **Hyperlink CASE-tool**: Classical CASE-tools consist mainly of various diagrams editors and of a data dictionary containing the definitions of all model elements. The tools are targeted at paper output, i.e. the resulting models and diagrams are not viewed primarily on-line but are printed out. In contrast to this, newer research efforts on CASE-tools make heavy use of hyperlinks (see e.g. [Alvarez95]). The resulting tools are browsers that support on-line viewing of the models, allow complex hierarchies of diagrams and specifications, support zooming and folding techniques as well as other automatic changes to the appearance of models (e.g. grouping objects, hiding objects or interactions, expanding interaction diagrams of aggregate services), and make a clear distinction between entering, viewing and printing information. Therefore it would be very interesting and productive to collaborate on current research on hyperlink CASE-tools when seeking to implement a CASE-tool for SEAM.
- **Scenario model simulation**: Further research in the areas of simulating scenario models would be very interesting. So far, we only propose walk-throughs. However, the interaction diagrams could be simulated automatically, even those of higher level scenario types. This simulation could be interactive, with a human providing all the decisions and data elements that are left open, or it could be carried on with scripts. Some open questions are: What are the goals of such simulations? Which part or parts of a scenario model (only external view of the whole system or also internal views) should be simulated? What should such a simulation tool look like?
- **Verification and validation methods for scenario models**: We discussed completeness and consistency for SEAM, but we did not provide any sophisticated verification and validation techniques and processes for SEAM. The investigation of such techniques and processes, in the context of SEAM as well as for scenario models in general, would be another important research issue.
- **Integrating DBMSs**: So far we have not discussed complications which might arise in projects dominated by a database system. Such a database system could be a new object-oriented database, a new relational database, or a legacy system. In all three cases, it might become necessary to adapt our modelling concepts to the concepts used in the database system. For the cases where the coexistence of two different models is chosen, i.e. one for the database and one for the application, the consequences for the modelling of global behaviour by using scenarios need to be investigated further.
- **Viewpoints**: When discussing the possible transitions between scenario models and the need for a high expressiveness of the modelling technique, we mentioned the modelling and the integration of different viewpoints in requirements determination. Further research should clarify in more detail how this should be done, in particular concerning the process and tool-support (see e.g. [Finkelstein96]).

Appendix A
Overview of some methods

A.1 Introduction

The following overview summarizes various methods that provide modelling techniques for describing global behaviour. To make such an overview is not a trivial task, as the essential differences often lie in the details. Especially the definitions of terms such as trigger, motive, event, message, stimulus, action, service and operation are different from method to method. Also, the mapping between these terms, i.e. how they relate to each other, is not the same at all. One consequence of this are similar looking diagrams not modelling the same facts and similar notations not having the same expressiveness and precision. Another consequence is that the overview suffers from the following weaknesses:

- No simple classification or direct comparison as is it is often done in overviews of static models has been possible. Concerning the static aspects of object modelling their is much more common agreement than concerning global behaviour and dynamic aspects. Here, we not only have differences in the symbols (as different symbols for modelling inheritance), in the terms (such as using class instead of object type), and in the semantic elements offered (such as offering link-classes or not). Much worse, the basic concepts of the different methods are often so distinct that they cannot be compared directly and that the terms cannot be mapped onto each other.
- No general metamodel and no metamodels of the individual methods are provided. Most publications of methods do not offer a metamodel of the basic modelling concepts. Not that meta-modelling would not have been known for quite awhile[1], but most methods[2] are only described quite vaguely by text, some definitions and examples. Due to the missing metamodels, there are often inconsistencies and inexactitudes in the concepts and terms; these then necessitate personal interpretations and therefore any metamodels created by someone else than by the authors of the method only reflect these personal interpretations. Having no metamodels of the individual methods, we also did not make a generic metamodel for the overview. An attempt for such a generic metamodel can be found in [OMG92], though not many methods have been described based on this metamodel[3].
- No complete description of the goals of the different models has been included in the overview. Though the goals of the different models were of great importance, not all the methods mention them explicitly, or they describe them only very vaguely or even contradictory. Most often, the methods mainly offer a set of notations useable for various purposes.

1. Many companies have made metamodels of their modelling techniques and of published well known methods in order to get a common agreement on their interpretation of the method, as e.g. [Briod93].
2. An exception will be the unified modelling language (UML) of Booch et. al. Its notation will be based on a metamodel (see [Booch95]).

In the first chapters we summarize some of the well-known methods that have found acceptance also in industry. To round up the overview, we add some further interesting notations not mentioned in chapter 2.1. Not all of the following methods claim to be object-oriented, yet they all offer modelling techniques that are of great interest in the context of object-oriented analysis. This is an overview, not a reference manual. Therefore certain details are omitted or simplified.

A.2 OOSE

The following summary is based on [Jacobson92] and [Jacobson95], though these sources are in their description of use cases and interaction diagrams quite vague and in many aspects incomplete.

Terms

Actors: Actors are objects in the environment of the system which interact with the system. They define the roles users can play.

Stimuli: The communication between objects is modelled by stimuli. A stimulus (equivalent to what in many methods is called an event) is sent by one object to another object in order to stimulate some behaviour in this object. It is either a **signal** (interprocess) or a **message** (intraprocess).

Use case instances: A use case instance is a sequence of related transactions performed by an actor and the system in a dialogue. A use case thus concentrates on the interactions observed at the user interface. It is an external model of the system as a whole. Though internal objects may be mentioned, the use case does not model or elicit the internal interaction structure.

3. An exception is FORAM as described in [Graham93].

Transactions: A transaction starts with a stimulus from an actor and finishes when its use case instance waits for the next stimulus from the actor. Transactions in OOSE could be compared to the system operations in Fusion.

Use case classes: A use case class consists of similar use case instances and contains a specification of the possible transactions of the use case. The whole set of all use case classes specifies the complete functionality of the system and gives a dynamic and black box view of the system.

Mappings and classifications

Grouping of use case instances into use case classes: Several use case classes may start with the same stimulus. As a consequence, it may happen that the class of a use case instance can only be determined after its complete execution. Which execution paths are grouped together into one use case class is up to the developer.

End of a use case: There are no general rules when a use case ends. If a sequence of transactions is modelled as one or as several use cases, is decided by the developer. Complexity is one possible criterion.

Common parts of use cases: Common parts of use cases can be extracted into **abstract** use cases.

Aggregation of use cases: In order to avoid any bias towards functional decomposition, OOSE does not offer any aggregation of use cases (in contrast to OMT that knows an add-relationship between use case instances).

Use cases and subsystems: Objects can be grouped into a hierarchy of subsystems, yet in the references mentioned above no precise description is given how the use case model is used together with subsystems.

More abstract use case models: OOSE suggests making for each abstraction level a complete use case model and to have traceability between these levels. A table is used that shows which use case from one level corresponds to which use cases of the other level. However, the different use case models can not be integrated into one model.

Transactions versus system state transitions: A use case can also be looked at as an incomplete state transition graph where each stimuli performs a state change of the system. The transactions of the use case correspond to the transitions of the state transition graph. These state transition graphs are implemented by controller objects.

Use cases and the user interface: It is assumed that the sequence of the external stimuli and the classification of the use cases does not change between requirements analysis and design. The initial use case model is already supplemented by a UI-prototype. The interaction diagrams and all other models of OOSE are based on the use cases of the requirements model.

Notations

The use case model

The use case model consist of a narrative description of each use case class and of a diagram that shows the relationships between the use cases. Possible relationships between use cases are:

- **Extends**: for modelling rare variants, subcourses or optional parts of a course. The extend-use case inserts itself into the original use case at instantiation time when certain conditions are fulfilled.[4]
- **Uses** (see example in figure 109): for extracting parts that are similar into **abstract** use cases. These cannot be triggered directly from the user, they are only executed as part of a concrete use case. The **concrete** use cases insert abstract use cases at instantiation time.

The narrative description contains the **basic course** of transactions, and in several additional paragraphs the **alternative courses** of transactions (see example in figure 108). Alternative courses show variants and contain error handling. The basic course contains more than only one possible sequence of transactions; some of the transactions may well be conditional or repetitive. What is expressed in the basic course and what is considered as alternative course is up to the developer. For each transaction, the user actions (stimuli) as well as the reactions of the system on these stimuli are recorded. Often the transactions are numbered. In the description of the transactions the same terms may be used as in the object model. Object names may be emphasised; but this is not a necessity. In the case of abstract or extending use cases, the details concerning where and how this use case may be inserted into another use case may also described in a narrative way.

Interaction diagrams

Each use case is refined with interaction diagrams. These show the object and object services involved in that use case as well as the detailed user interactions with the interface objects. The objects need not correspond to the final implementation classes, most often they are larger blocks. Each stimulus in the diagram corresponds to exactly one operation of a block, even if internal to the block the operation may be dispatched onto different object operations in the case where the object is state controlled and not stimuli controlled.

There is one diagram for the basic course and one for each of the important alternative courses. Not all possible courses are modelled. Thus one single diagram only has a small amount of optional or repetitive parts. The interaction diagram is a time-line diagram with the following elements (examples see figures 110 and 111):

4. Some critics doubt the usefulness of this modelling construct and of the distinction between extends and uses, see [Firesmith95], [Rumbaugh94b] and [Booch95].

- *Blocks or objects*: These can be entity, interface or controller objects, yet no low-level objects. Objects of the same type can be drawn with several lines or collapsed into one line.
- *System border*: All external actors are collapsed into one special line. The communication with the actors is only by signals. Most interaction diagrams start with a stimulus from an external actor. There may be an arbitrary number of signals crossing the system boundary on one interaction diagram.
- *Stimuli*: Messages (requests) and signals (notifications) are distinguished by different symbols. Each stimulus is labelled with its name.
- *Parameters:* The parameters of the stimuli are normally listed together with the name of the stimuli.
- *Operations:* Rectangles on the lines denote the operation that is triggered by the stimuli and show the duration of the operation. Return messages may (e.g. to emphasize a return parameter) but need not be explicitly shown.
- *Textual description*: A short description (structured english or pseudo-code) of the algorithm is provided to the left of the diagram. This description also contains pseudo-code constructs for *repetition* and *condition.*
- *Probes*: Probes are special symbols in the interaction diagram that denote where another use case can be inserted (for the extend-relationship between use cases).

State transition diagrams

State transition diagrams are recommended to help implementing the controller objects. Any notation may be used.

The use of the models in the development process

To define the requirements and to discuss them with the customer, a use case model is made together with a domain object model and a first interface prototype. For finding the use cases, also storyboarding techniques may be used. These do not concentrate on general use case classes but on individual instances. The use cases are also used to define the responsibilities of the individual objects in the analysis object model. This is derived from the problem domain object model and extended by interface and controller objects. During design, the objects or blocks are then further specified by the definition of the operation signatures which are determined by the interaction diagrams. The goal of the interaction diagrams is to define the protocol of the blocks and the parameters for each stimulus. The interaction diagrams are also an important tool to stabilize and rework the architecture of the system before any detailed internal design of the blocks and their classes is done. The use case model is also reworked during design because the interaction diagrams lead to a homogenization of the original description of the use cases. It is thus not assumed that the use case model is stable at the end of requirements analysis.

Use cases reflect the user's view, whereas the objects reflect the developer's view and are derived from the use cases. Though the identification and definition of the objects is said to be straight forward, it is also mentioned that during robustness analysis and interaction diagram design the objects have to be harmonized and homogenized so that they support all and not only some use cases in a reusable way (reusable objects are the *entity* and *interface objects*, use case specific objects are the *controller objects*). The use cases drive the development activities, from delimiting the system to testing the application. Use cases are used for finding the initial requirements as well as for defining the user interface, they neatly fit into the user-centred design. Yet for all this, the underlying assumption is that those use cases defined at the beginning of the project will be the same as those of the final high-level view, and that already the initial use case model contains a high-level specification of the user interface.

A.3 Fusion

The following summary is based on [Coleman94].

Terms

Events: "An *event* is an instantaneous and atomic unit of communication between the system and its environment. An *input event* is sent by an agent to the system; an *output event* is sent by the system to an agent."

Scenario: "Sequence of events flowing between agents and the system for some purpose".

Agents: Agents are active entities in the environment with which the system communicates. Agents model human users, or other hardware or software systems. The system itself is nothing else than the agent that is being analysed.

System operations: "An input event and the effect it can have are called a system operation". In other words, these are the services offered by the system as a whole. They are atomic. Because only sequential systems are taken into account, at any point in time only one system operation can be active. A system operation is always triggered by an input event from an agent. It can neither be triggered by another system operation, nor by any active object internal to the system (e.g. a timer).

Methods: These are the services offered by individual objects.

States: The term state is used in two different contexts: for all possible system states (values of all variables) and for the states used in the state transition diagram that controls the correct sequences of the events (see also chapter 4.4.1).

Mappings and classifications

There is a 1:1-mapping between input events and system operations, and between the system operations and the methods of the controller objects. The parameters of the input events are attributes and objects of the object model or data elements only defined in the data dictionary. Each system operation is described by one operation schema (in order to simplify the postconditions these can be broken up into several schemas with differing preconditions). Each system operation becomes a method of an object (which serves as controller object) in the design and has one interaction graph which can be decomposed.

Output events are mentioned in the operation schemas of the corresponding input events, but they need not correspond directly to any methods or return values in the interaction graphs.

One system operation includes all those sequences of events that start with the same input event type. A system operation contains only one input event, the triggering event, and ends as soon as all internal state changes and corresponding output events have occurred. Longer sequences of events are described in the system life-cycle. The regular expression used for its specification can be decomposed into several expressions.

Notations

The interface model of analysis consists of the life-cycle model specified by regular expressions and the operation model specified by operation schemas. Furthermore event trace diagrams are used to sketch certain scenarios in order to help finding the system operations. The event trace diagrams are not suited to model the whole system life-cycle and do not belong to the final system documentation. In the design, global behaviour is modelled by the interaction graphs. In the implementation, a state machine is used to implement the system life-cycle as specified in analysis.

Event trace diagrams for scenarios

These diagrams show one possible sequence of input and output events between the system and its agents. Iterations can be represented (see figure 112), but not interleaving or alternative courses.

Regular expressions for the system life-cycle

The life-cycle model is a regular expression (example see figure 113), consisting of:

- *input event names*, *output event names*, *local names* of regular expressions (the expressions can be decomposed into arbitrary many regular expressions, making the specification more readable),
- *operators* for concatenation x.y, alternation x|y, repetition x* or x+, optional [x], interleaving x||y and grouping (x).

The most important rules are that output event must not be interleaved with input events and that interleaved regular expressions must not start with the same event.

Operation schemas for the operation model

One operation schema describes one system operation and contains the following information:

- *operation name* and textual *description*
- list of all those objects, attributes and relationships of the object model that are *read* or *changed*
- list of all those objects that are *created*
- list of all those objects, attributes and data elements that are *supplied* as parameters of the system operation
- list of all *output events* including the names of the receiving agents
- *preconditions* and *postconditions*

All the objects, attributes and relationships mentioned in the operation schema must be defined in the object model of analysis.

Common parts of system operations cannot be factored out, they are repeated in each system operation. System operations that can be triggered by an agent as well as by another system operation (e.g. producing a status report), are modelled twice.

Interaction graphs for the design model

One interaction graph represents all possible sequences of method calls necessary for the execution of one system operation. The graph contains the following information (examples see figures 118 and 117):

- *objects* and *collections of objects* (a message to a collection means that the method is invoked for each object of the collection, a collection object is modelled as an ordinary object),
- *messages*: method calls with parameters and return value,
- *sequence numbers*: show the flow of control and give a vague idea on possible sequences (the precise information on the order of method calls is found in the description of the methods),
- *method descriptions*: for each method the algorithm is described including all method calls to other objects.

The interface objects are modelled for the output events yet not for the input event. The detailed user interactions as well as the mechanisms to check the life-cycle are not part of the design model.

State transition diagrams for the implementation

The state machine that controls the sequence of the events and decides whether an input event is accepted or rejected is already specified by the life-cycle expressions. These expressions are transformed into a state transition diagram which is implemented by specific controller or interface objects.

The use of the models in the development process, weaknesses

The analysis model turns an initial requirements document into an unambiguous and precise set of models and defines the intended behaviour of the system. Requirements determination itself is not part of the Fusion process. The analysis model is expected to abstract away implementation details and to focus on the problem and not on the software solution. All external behaviour is modelled exhaustively. This enables consistency and completeness checks, yet makes latter changes that exceed minor details impossible and presupposes that the first choice of system operations is the best and final one.

The interaction graphs are used in the design to help allocating the methods to the classes and to document the implementation of each system operation. Basically it is assumed that the "external" view of the analysis model can directly be enhanced by the internal view of the design and that only minor changes in the external view are necessary. This gives a very straight forward development process with a high degree of seamlessness. Yet the method ignores some inherent limits to seamlessness (see also [Beringer94] for a deeper discussion of the limits of seamlessness).

In the following we list some weaknesses of the Fusion method. Most of them are due to the fact that the analysis model predetermines many important design decisions concerning the internal software structure. Yet these decisions cannot be taken properly when modelling the external behaviour on a conceptual level.

Internal state information in the operation schemas

In order to define the pre- and postconditions, internal state information needs to be modelled in the object model that is never used as input or output information and is hidden from the user of the system. This state information appears also in the read and change clauses of the operation schema and must be assigned to classes. This leads to complications in the transition to the design for two reasons:

- The final object model should satisfy the quality criterias for good encapsulation of operation and data (as they are even mentioned by [Coleman94] itself), yet the assignment of the internal state information to classes is made before the responsibilities and methods of these classes are determined. Where which state information is stored influences significantly the possible structure of the interaction graphs and limits their optimization. Of course, it is possible to rework the whole object structure during the design and to redo the analysis model, yet this is not very feasible.

- The postconditions of the operation schemas describe the reactions in a declarative way. When moving to a procedural specification, not necessarily the same attributes are optimal in the algorithms of the methods as have been used in the declarative descriptions.

Moreover, there is an imbalance between the treating of the system states used in the life-cycle model and of the system states used in the pre- and postconditions. Whereas the first ones are not defined and considered in the object model until implementation, the internal information necessary to define the system state used in the operation schemas must already be modelled in the object model of analysis.

No further user interactions during one system operation

Fusion assumes that the input events defined in analysis (conceptual events) correspond exactly to procedure calls (technical events) invoked by a user interface which is not further modelled in analysis. When the design models already exist or some people have already experiences with many similar systems, it is of course possible to make an appropriate analysis model with input events that correspond to object methods in the design. Yet in normal forward engineering, this normally is not the case; as a consequence, decisions that influence significantly the possible software solutions and the system architecture are taken when considering only an external high-level view of the system.

Most often we want to design user interfaces where the user can enter all the parameters of a system operations interactively. Yet for this the following problems arise:

- At any time only one system operation can be active. For multi-user systems (such as information systems) this would mean that one user blocks the whole system until he has entered all parameters.
- If a wrong value of a parameter should not lead to the rejection of the whole input event, then already the interface objects need to check them. For this they either have to interact with the corresponding problem domain objects, or they need a copy of their content, before they send a message to the controller object that itself interacts again with the corresponding problem domain objects. Both solutions do not give an optimal system architecture, and worse, do not appear in the models of the design.

Life-cycle model

The recommended transformations of the system life-cycle into a state machine targets only at sequential systems. Distributed and concurrent systems cannot be implemented by the suggested approach. System operations may never execute in parallel and only one single life-cycle for the whole system is provided.

Those cases where the acceptance of an event is determined by both, the event types and their parameters (such as the name of the agent that sent it or the object instances on which the events will have an effect), are very difficult to model appropriately by the sys-

tem life-cycle and the preconditions. A certain amount of redundancy is unavoidable. This redundancy is carried on into the implementation, because:

- different objects are responsible for checking the preconditions and the values of the input parameters and for controlling the state machine,
- the states mentioned explicitly in the preconditions and the states derived from the life-cycle expressions are treated in different phases of the development process and are never homogenized.

No symmetry between input and output events

Output events most often correspond to individual objects or attributes without any parameters. In the operation schema there is no corresponding keyword to 'supplied'. This inconsistency between input and output events continues in the interaction graphs where output events do not become the return values of the system operations. They are only parameters of some new methods of some interface objects. This lacking symmetry is due to the fact that the system life-cycle of analysis views the input and output events as notifications, whereas in the design the input events become requests.

Extended Fusion for requirements modelling

In [Coleman95] use cases as introduced by OOSE are integrated into Fusion. The use cases are modelled by

- a context diagram showing also the relationships (extend, uses) among the use cases,
- a textual description for each use case class defining its name, stating its goal and listing all the transactions involved in the use case,
- scenario diagrams showing use case instances.

The uses relationships allows us factoring out common parts of use cases into full fledge sub use cases. In the scenario diagram, sub use cases appear as shaded boxes (see figure 114). The extend relationships allows us weakening assumptions made for a use case by treating special cases. The inserted transactions are no full fledge use case. [Coleman95] recommends the usage of the extend relationship for defining system increments for the EVO-Fusion development process [Cotton95]; first only the basic use cases are implemented, later on the system is enhanced to treat also the various extends.

The use case description does not specify the exact ordering of the transactions. This is done during analysis, when use cases are subdivided into system operations, and the system life cycle gets defined.

A.4 Modern Structured Analysis

The following summary is based on [Yourdan89] and [Brantschen91].

Terms

Terminators: objects in the environment of the system that interact with the system.

Input and output dataflows: interactions between the system and its terminators.

Business transactions, business functions, essential activities (in the domain of banks also called **banking operations):** abstractions of processes triggered by a specific event.

Events: trigger the business functions. Different types of events are distinguished:

- *flow event*: the business function is triggered by a dataflow coming from a terminator.
- *time event*: the business function is triggered by some time constraints (e.g. "once a week").

Mappings and classifications

There is a 1:1-mapping between event types and business transaction. A business transaction ends when the reaction of the system is completed. Yet there are two different criteria that can be applied to determine when the reaction of the system is completed:

- The business transaction ends as soon as the next input from a terminator is awaited. Thus every input dataflow is a new event. The concept of having perfect technology and business transactions with zero duration can be applied to such an analysis model.
- Especially in the realm of information systems, the first classification results in a huge amount of different business transactions and in cutting apart business procedures into independent business transactions which belong together from the viewpoint of the user and sometimes even cannot be executed independently. Therefore all reactions of the system can be said to belong to one business transaction until a more conceptual business procedure is ended or the system is in a "stable" state. If intermediate interactions with terminators are necessary, these is done by input and output dataflows that are not considered as events. Though this second classification results in simpler models, there are many unresolved details around it (such as vagueness, inconsistencies, concept of perfect technology).

Modern structured analysis does not provide any further structuring or grouping of events or business transactions. Some adaptations of the method have introduced a third type of events, the *internal events*. These help to extract common parts into a separate business transaction. This business transaction is not triggered by a terminator but by some other business transaction.

Notations

Environmental model

The **event list** is a table with the following rows:

- name of the *terminators* causing the business transaction
- name (and parameters) of the *input dataflow*
- *type of the event* (flow, time, internal)
- *conditions* for the different reactions
- *reactions* (name and parameters of the output dataflows)
- name of the *terminators* receiving the output dataflows

The table contains one entry for each business procedure (example see figure 122).

The **context diagram** (see figure 123) is a dataflow diagram that shows the system, the terminators and all input and output dataflows. A textual description of the dataflows specifies all the data entities, attributes and relationships transmitted by this dataflow. All the data that flows into or out of the system is modelled in an **entity relationship diagram**.

Behavioural model

For each business transaction a **dataflow diagram** is made. These are further decomposed according to the rules of dataflow modelling. The lowest level processes are described by **minispecs**. These describe textually or with pseudo-code the algorithm for processing the input into the output data. Furthermore the dataflow diagrams can be enhanced by control flows and by control processes specified by state transition diagrams.

The correlations between the data entities and the business transactions are shown in a **CRUD-matrix**, which can be automatically derived from the minispecs and the entity relationship diagram.

The use of the models in the development process

Modern Structured Analysis is used at the beginning of a project to get a technology independent, essential model of the problem. The main goal is defining the problem, and thus having an aid in discussing and documenting it. This is mainly achieved by the environmental model. A complete behavioural model as described by the method is often omitted in practice, because it does not add any new insights into the problem and the structure given by low-level dataflow diagrams often cannot be used for designing the software system. The gap between the analysis model and the design model is quite large, and the two goals, having a technology independent model and having a smooth transition to the design, often proof to be irreconcilable.

A.5 OBA

OBA stands for Object Behaviour Analysis. The method focuses mainly at determining the scope and the functional requirements of the system and at finding suitable objects for an object-oriented analysis model. OBA can be used in combination with other methods such as OMT and Booch, which then provide the syntax for the analysis model and also a process for the design. OBA is described in [Rubin92].

Terms, mappings and classifications

An **event** occurs whenever an object invokes a service in another object. Yet during analysis, only those events are taken into account that cause one or more objects to experience a state change. Also, only those state changes are considered that consequently affect the behaviour of the system[5]. Events thus reflect essential occurrences in the system or in its environment.

A **scenario** (also called use scenario) is a sequence of service requests and activities in order to accomplish some overall task. The developer is quite free which sequence of service requests he considers to be one scenario. For event-driven systems, a good starting point is to consider each external event type as one scenario. The only restriction is that there may be no noteworthy or essential state changes within one scenario, i.e. after each such state change a new scenario starts. Essential states are all those states that are modelled in the dynamic model. Thus each scenario equals to one transition or event in the state transition diagrams of the dynamic model.

Each scenario is documented by one **script** and contains a reference to a core activity area; these areas can be used to group scenarios. The possible sequences of scenarios are first defined by pre- and postconditions (added to all the scripts) and then also shown in the dynamic models of the objects (state transition diagrams).

The participants and initiators in the scenarios are called **parties**. These are either outside of the system or correspond to one or several objects within the system. A party may correspond to a group of objects. Due to the differentiation between parties and objects, it is possible to keep these groups of objects even after the definition of a finer and more complex object model.

The behaviour of the parties that can be contracted for use by other parties are the **services**. There are four main categories of services: accepting notification, accepting information, providing information, providing a service. For providing a service a party performs several **actions**; these may invoke services of other parties.

5. A **state** represents a condition of an object during which certain physical laws, rules, and policies apply. Only those states that influence the behaviour of the system differently (i.e. not the same scripts are applicable in these states) are considered to be different in analysis.

Notations

Scripts

A script is a table with four columns:

- the party that is the *initiator*,
- the initiator's *action* that necessitates a service from another party,
- the *participant* that provides the service,
- the *service* (its name or a brief declarative description).

The reaction of the participant on a service invocation is expressed on the next line of the script if it is considered as important enough to be listed at all. There exist two versions of scripts: scenario instances with concrete values for parties and objects as in figure 124, and scenario types annotated by *control flow annotations*. These are either textual annotations to the left of the scripts or are expressed by additional diagrams. They may show concurrency, repetition, selection and options.

Preconditions express what must be true in order that the script is applicable. *Postconditions* denote those state changes that have an influence of the applicability of future scripts. Pre- and postconditions are expressed in terms of state descriptions of objects, for instance "overdrawn (account)".

References to goals, objectives, core activities and other scripts are listed as *traces*.

Glossaries

Glossaries are used for the description of *parties* (they specify also the objects that a party consists of), of *alias names*, of *attributes* (used for the definition of states and the description of services), of *services*, of *states*, of *reorganizations* and so on. They evolve during the scripting process and during object modelling. They enable to find gradually the optimal object structure without redoing all other scripts. Together with the scenario scripts and the object definitions, glossaries are an integral part of the analysis documentation of OBA.

Dynamic model

The dynamic model contains for each object a *glossary* of the essential states of this object and, if applicable, an *object life-cycle*. The glossary lists for all states their names, their definitions (boolean function over attributes and values), their textual description and their traces to the scripts. The definitions are derived from the pre- and postconditions and are expressed in terms of logical expressions over tuples of object and state description. An object may well have several states in parallel. The Object life-cycles are modelled by petri nets or Harel state charts.

The use of the models in the development process

The goal of analysis is to construct a model of the problem domain, i.e. a static and a dynamic model of the domain objects with traceability back to goals and objectives of the system. First the analysis context is set by identifying business goals, objectives and the core activity areas of the system. Typical scenario instances are chosen and scripted. Glossaries of parties, services and attributes evolve. Only in a second step are the objects determined and specified by using object modelling cards and recording the results in the glossaries. Building inheritance hierarchies and identifying relationships involves also the reorganisation of the object definitions which also include a list of all contracted services. The last step is modelling the system dynamics: state glossaries are derived from the pre- and postconditions of the scripts, state charts or petri nets are made for all important objects, state glossaries are updated. If necessary, the scenario scripts are restructured so that each scenario corresponds to one event in the dynamic model. Missing states and missing scripts are added. Also, the scripts may now be enhanced by control flow information.

The scripts have thus a twofold purpose during the development process: they support the learning process at the beginning of the project and help to find the problem domain objects, and they document a high-level view of the behaviour of the final system.

A.6 FORAM

FORAM stands for Financial Object-Oriented Rapid Analysis Method. The strength of FORAM lies in its process for determining the systems goals and in its approach to domain and system analysis. It uses task scripts developed in a workshop to capture requirements, and it integrates very well the users and managers during the whole analysis process. For the object model, the SOMA notation is used. FORAM as summarized here is described in [Graham93] and [Graham94b], SOMA in [Graham94], business process modelling using basically the same modelling techniques is described in [Graham95].

Terms and notations

FORAM distinguishes the following three models:

- **Task object model** (TOM): *context model*, *message table*, and *task cards* (see figure 119). The objects are tasks and not business objects! At the beginning, the task object model is an external view of the system. But after the decomposition process, it also shows an internal view, yet not based on objects but on internal tasks. These are then used to find the objects and their methods. It is assumed that the task model reflects both, the business and the software system alike.
- **Business object model** (BOM): *object model*, *class cards* (including rules and collaborators), *event traces* and *state diagrams*.

- **Implementation object model**: detailed object model of the code. It is programming language dependant and includes also low-level objects.

Whereas the task object model is mainly for capturing requirements and analysis, the business object model incorporates also the first steps of logical design.

Context model

The context model shows the system object, the external objects and the message flows between them; it may also show part of the task model. External objects are either actors (users adopting a particular role to interact with the system for some purpose), external systems, or something else that interacts with the system. The system itself can be considered as an actor being under closer consideration.

Message table

The message table lists the external messages of the system. The table has one message row for each interaction on the context diagram. Each row lists the *message name*, the name of the *trigger event* (may be a temporal or an external event), the *source* of the message (an external object), the *target*, the *information* sent or received (parameters of the message), the *expected result* (what the initiator expects to get or to happen), and the *goal* of the message.

Messages are dataflows exchanged between the system and the external objects (more precisely messages are semiotic acts). The message is from the initiator to its recipient, even if the actual dataflow is the other way round. The mapping between events, messages and tasks is very vague; messages appear only in the message table and not in the task specifications[6]. The **goal** gives the characteristics of a message. It specifies the desirable states to be achieved and shows why the message has been sent. Goals complete the description of the system's scope and are the starting point to discover the tasks.

Task scripts

The task scripts (described by **task cards**) specify the individual task types. They include the following items: *name* of the task, *supertasks*, *component tasks*, *task body*, *associated tasks*, *exceptions*, *side-scripts*, *rulesets*. Task scripts are decomposed into **component scripts**. The order of the component scripts is described by the rules of the ruleset. **Sub-scripts** show the specialisation of tasks. Exception handling and special cases are treated by **side-scripts**.

Tasks are first class objects in the task domain. Tasks are carried out in order to achieve the goals of the messages. They are considered as objects due to their prototypical character and because they can participate in specialisation, composition and usage structures. For the external view of the system, there is at least one task script for each message in the message table. Tasks are decomposed until **atomic tasks** are reached. Which tasks are considered as atomic tasks is determined by the developers and may change during the process[7]. The process of task decomposition is called **task analysis**. Task decomposition is considered as being object-oriented, in contrast to functional decomposition combined with ERD-modelling. It is well suited for RAD-workshops, and it mirrors well business processes, based on communicating actors. The decomposition activity is purely in the task domain, which is orthogonal to operations and objects.

A **task** is an equivalence class of use cases, a **use case** is an equivalence class of **scenarios**. Task scripts, use cases and scenarios stand for three different abstraction levels.

Class cards and object diagrams

Class cards specify the interface of the objects and contain *attributes* (responsibilities for knowing), *operations* with *assertions* (responsibilities for doing), *relationships* (only one-way pointers out of the object!), *server objects*, and a *ruleset*. The object model models interface, domain and application objects. Any notation is possible, though the SOMA-notation for the object diagram is preferred, because it also shows rules and allows us modelling composite objects, so called layers.

For each object type, **rules** are defined which may be inherited like other features of an object. Rules need not be as precise as the **assertions** of the operations; these are derived during the design from the rules. Rules have the advantage that they model in a very understandable and readable way the business rules and dynamic behaviour of an information system and that they reflect human reasoning. Furthermore, they are not bound to a particular logic and to the restrictions of this logic, and they can be fuzzy whenever the state of knowledge acquisition does not yet allow more details. Yet they are not suited to express the dynamics when formal correctness is already an issue in analysis. Rules can show global control, business rules (relating several attributes), dependencies between attributes and operations (triggers) and integrity rules, i.e. they can contain any kind of second order information.

Event traces and state charts

The dynamic aspects are completely specified by the rules, assertions and constraints on the class cards. Event trace diagrams (example see figure 120) and state charts are only used as a help to identify and specify the objects and their operations. Event traces are also used in walk-throughs in order to verify the completeness of the models and the consistency between the TOM and the BOM. Furthermore, they are used as the basis for the system test scripts. Event trace diagrams may be annotated with comments concerning temporal constraints, decision processing etc.

6. [Graham96] clarifies the relation between tasks and message: messages: the description of messages also contains the name of the task that is necessary to fulfil the goal of the message.

7. In [Graham96] an atomic task is defined as a task that cannot be further decomposed without introducing terms foreign to the domain. Furthermore, an atomic task is described by one sentence.

The use of the models in the development process

In contrast to many other methods, FORAM includes also object-oriented requirements determination. This is mainly done in workshops at the beginning of the project where the *task object model* and the *preliminary business object model* are developed. The following steps are carried out in the workshops:

- **Defining primary business objectives:** determining and prioritizing the primary business objectives, defining measures for each objective. The objectives are needed to determine the system boundaries and later on to decide on the time-boxes for the RAD-development cycles.
- **Context modelling**: defining the system boundaries and developing the context model and message table. The goals and expected results become clear.
- **Task analysis**: decomposing and describing the tasks that lie behind the goals.
- **Building the object model**: retrieving candidate objects and methods from the task-scripts using text analysis. The task-scripts may be further refined, until no further objects fall out. Furthermore, the tasks-scripts may be decomposed until the level of the effective user interface is reached. The objects are recorded in class cards and in an object diagram. To verify the object model and to find the collaborators for each class, event traces are modelled and role-plays are carried through.

All the models developed during the workshop are further revised and refined when developing the implementation object models and the prototypes in the time-box cycles.

A.7 BON

The method BON is documented in [Walden95].

Terms

Events: An event or system event is “a stimulus that can change the course of action of a system and make it react”, “something to which a system will respond with a certain behaviour”. An **external event** “is triggered by something in the external world over which the system has no control”. An **internal event** “is triggered by the system itself as part of its reaction to one ore more external events”.

Scenarios: A scenario is “a script of a possible system execution showing the objects involved, which other objects they call, and the temporal order of these calls”.

States: The **system state** is the sum of all information stored in a system, it reflects the history of events. An **object state** is that part of the system state that has an effect on the future behaviour of one particular object. It may but needs not correspond to some attributes of the object, the data making up the object state needs not be stored in the object itself.

Messages: A message is an invocation of an operation on an object (feature call, message passing).

Mappings and classifications

Events

The system events of the final system are very low-level (e.g. command selection, sensor inputs, commits of transactions). As scenarios and event charts are only used to give a high-level overview of the system and to detect problem domain classes, only those events that represent a more abstract stimuli are considered in the models of analysis. Yet there exists no precise concept of event abstraction.

Scenarios

A scenario is stimulated by either an external or an internal event, normally such an event is listed in one of the event charts. There may be further inputs from the user during a scenario not mentioned in any event chart. The name of a scenario may or may not be similar to the name of the event by which it is stimulated. There is also no direct correspondence between the events in the event chart and the message relations in the diagrams. Theoretically there would exist a client relation in the static model for any two classes that have a message relation between any of their instances in the dynamic model. Yet for the sake of simplifying the static model, not all client relations are shown, the less important ones only appear in the class interface specifications.

Because the dynamic model only contains diagrams for a few examples of scenario instances, their are no rules concerning the classification of scenario instances into scenario types. Also, no guidelines are given when a scenario should end and the next one begin. A decomposition of scenario diagrams is possible by using object groups.

Object groups

Objects can be compressed into groups. An object group is defined as “a set of objects treated as a unit in some message passing context”. It may group objects

- that receive all the same or a similar message (mirroring an inheritance structure in the system and incorporating polymorphism in the dynamic model),
- which are called in a sequence to do a more abstract step (see figure 128),
- of which all than one serve only as data containers.

An object may belong to different group hierarchies. The rules for the compression of message relations (omitting message relations, applying one message relation to a whole group) are analogous to the rules for client relations in the static model. Though BON emphasizes the possibilities of zooming and compressing, these concepts are only applied for classes, objects and relations, but not for scenarios and events, because the dy-

namic diagrams are only considered as auxiliary models which are neither exhaustive nor complete and only help to develop the class specifications.

Notations

Charts

The **event chart** is a list of important system events. For each event, its name (which may well be a sentence) and a list of possibly involved object types are recorded. Only a subset of all system events is considered, namely:

- *input events*: external or internal events which trigger essential types of system behaviour,
- *output events:* internal events that are triggered by special system states and are not directly related to an external event; other internal events of interest.

In the **scenario chart** all those scenarios are listed that illustrate important aspects of the overall system behaviour. For each scenario its name and a short textual description (only a few sentences) are listed.

For an example of the above charts see figure 126. There exist also charts for classes, clusters, systems and object creation. For more complex systems, separate event and scenario charts may be made for different subsystems or tasks.

Dynamic diagrams

Only a representative set of scenarios is further detailed in dynamic diagrams, these are called object scenarios (see figures 127 and 128). One object scenario describes the call structure of only a single execution path. Alternative courses are not included and are deliberately neglected in the dynamic model.

A dynamic diagram is the graphical representation of one object scenario. It contains:

- *Objects*, *sets of objects* and *object groups*.
- *Message relations*: no message names, no parameters or return values. Message relations signify a potential call from a client object to an operation of some server object.
- *Sequence numbers*: represent time (only one possible sequence of potential message relations) and are at the same time the references to the entries in the scenario box.
- *Scenario box*: contains the name of the scenario plus a list of the most important steps of the scenario execution. One entry summarizes several message relations, no details are included.

The dynamic diagrams of BON are on a very high abstraction level. For the reason of simplicity, neither the diagram nor the scenario box show any conditional control or event type information (parameters, return messages).

External events as the stimuli of the scenario and other user inputs are denoted by special arrows and do not signify a call to an operation. Output events (or the invocation of interface operations) are not included in the diagram. Also, return messages are normally not shown. If and when a return value or the control is passed back is too low a detail for these diagrams. Interface objects are not included in the diagrams, therefore the user interactions are not modelled into details. User inputs are notifications, all other message relations are requests.

An arrow to a set of objects means sending a message to one object of this set (in contrast to the semantic in Fusion). The intermediate steps of getting the object handle from the set object and the set object itself are not shown on the dynamic diagram. They are represented on the static diagram as (compressed) client relations and in the class interface specification as references to sets.

The use of the models in the development process

BON divides up the whole development process into 9 *tasks*. The three tasks for gathering the analysis information are: *delineate system borderline*, *list candidate classes*, *select classes and group into clusters*. The three tasks for describing the gathered structure are: *define classes*, *sketch system behaviours*, *define public features*. The three tasks for designing a computational model are: *refine system*, *generalize*, *complete and review system*. For each task the input sources, the deliverables and the acceptance criteria are defined. The order of the tasks may change, depending on the concrete circumstances of the project. Also, not in every project all the tasks are needed. Furthermore, various *activities* are distinguished: *finding classes, classifying, clustering, defining class features, selecting and describing object scenarios, working out contracting conditions, assessing reuse, indexing and documenting, evolving the system architecture*. These activities appear in all the tasks though with differing weight.

For finding scenarios, BON recommends to look for *user tasks* (not to be confused with the tasks of the development process). A user task may then be broken down into *user actions* which become the scenarios. Yet this hierarchical structuring is not reflected in the dynamic models. The scenarios are used for the following purposes:

- delineating system borderline,
- finding classes and basic concepts, determining object operations, refining the system structure,
- documenting the final system.

Reversibility and seamlessness are leading principles in BON. Furthermore, constructing metaphors and contracting are emphasised in order to get a high quality software system that suits the needs of the user. The class interface specifications are the basic model,

all other charts and diagrams are only additional aids to develop and to understand the contracts of the individual classes. The high-level abstractions of analysis are chosen in such a way that they can be used all the way down to the design and the code, thus simplifying the propagation of changes throughout the different models. In order that the dynamic model always represents the actual system, it is subject to continuous improvements until the final user metaphors and the best abstractions are found. Therefore, at no point in the development all the dynamic characteristics of the system will be modelled, the effort is concentrated on those aspects that are felt to be the most important ones and are needed in order to evolve the contracts of the classes. The documentation always reflects the actual system and not historical aspects of the different iterations in the development process (faking the ideal process). Thus most charts and scenarios produced during development are discarded at the end; the final documentation contains only those few scenarios that are considered as essential for helping other people to understand the implementation.

A.8 Booch and OMT

Booch and OMT both have a vague distinction between analysis and design, their notations can be used throughout the development process. The main focus of the methods lies on the static object model. Modelling the global behaviour is only done as far as it is necessary to find the static model. Both methods offer a large kit of notations with many optional details. It is up to the developer which notations he uses to what extent and for which purposes, and how he integrates the resulting models. The following overview does not go into the various details of the methods concerning modelling the dynamics and the global behaviour, we only mention the most important notations. The references are [Booch94], [Rumbaugh91] and [Rumbaugh94]. For experience reports on the use cases as defined by OMT see also [Loenvig95] and [Hansen95].

Notations

State transition diagrams of Booch

The diagrams are derived from Harel state charts and from the dynamic model of OMT. Each diagram describes the behaviour of one object type by specifying its essential state changes (see figure 129).

Object diagrams of Booch

Object diagrams contain information concerning associations, visibility and messages between objects. They represent a snapshot in time of an otherwise transitory stream of events. They show all messages and objects needed in a certain context, for instance for a complicated interaction mechanism or for a system function. In analysis they are used to indicate the semantics of primary and secondary function points, in the logical design to enlighten certain key mechanisms. It is up to the developer what he models with the

object diagrams and how many different static and dynamic elements he mixes into one diagram. An example is found in figure 131.

Interaction diagrams of Booch

The interacting diagrams do not contain additional information to the object diagrams, yet they represent the execution of a scenario as a time-line diagram. These may be used to express one single scenario instance or a whole scenario type. Further details such as scripts to express different execution courses or symbols for showing the focus of control are optional (see figure 130).

Use cases in OMT

“A use case describes the possible sequences of interaction among the system and one or more actors in response to some initial stimulus by one of the actors. It is not a single scenario (a specific history of specific events exchanged among system and actors) but rather a description of a set of potential scenarios, each starting with some initial event from an actor to the system and following the ensuing transaction to its logical conclusion” [Rumbaugh94b]. Use cases describe the system as a black box and focus on the externally visible behaviour. In one use case all those transactions are grouped together that are similar in nature. Within one use case arbitrary many interactions with several actors may occur.

The use cases are described in natural language (see figure 133). The description contains a *main case* and various alternatives or *subcases*. Common parts of use cases may be extracted by the *add-relationship* (see figure 132). The relationships are annotated with their cardinality (one, optional, many). The add-relationship replaces the uses- and the extends-relationships of OOSE.

The functional model of OMT

Any *operation* can be looked at from a black-box (external) viewpoint, or from an internal viewpoint. The external viewpoint is modelled in a declarative way by operation specifications which are similar to the operation schemas of Fusion (see figure 138). The internal view is documented by object-oriented data flow diagrams and object interaction diagrams (see figures 137 and 136).

In the system analysis model, only the top-level *system operations* are considered; these are invoked by interactions with outside actors.

The dynamic model of OMT

The complete dynamic model is specified by state charts of all those objects that undergo different essential states, and by tables for modeless objects. In the design the events correspond to procedure calls. In the analysis they can be more abstract.

Other diagrams such as the event trace diagrams (see figure 135) for showing a particular sequence of interactions among objects in a single execution (scenario), or the event flow diagrams (see figure 134) only support the better understanding of the dynamic model.

The use of the models in the development process

After the identification of the core requirements of the future software system and of the system boundaries, the desired system behaviour is modelled by use cases or business functions (also called function points). Booch differentiates the behaviour into a primary behaviour (also called *key* or *fundamental behaviour*) and a secondary behaviour (behaviour under exceptional conditions). Whereas during analysis all primary behaviour is modelled by scenarios, only some of the secondary behaviour is looked at in order to insure that no essential patterns of behaviour are missed. These use cases or business functions are then used together with other sources such as domain models to define the responsibilities[8] of the individual objects, i.e. all the services an object provides.

Use case descriptions and scenario diagrams are not expected to make up a complete model, they are only aids to determine the object structure. The complete behavioural model is given by the class specifications and the state diagrams of the individual objects.

Though the processes described by Booch and OMT are not identical, they both target at developing models that show the static and the dynamic aspects of the individual objects. Booch uses for this the class diagrams, the state transition diagrams, the object diagrams, the interaction diagrams, the module diagrams and the process diagrams. OMT subdivides the model into an object model, functional model and dynamic model. All these models evolve during analysis and design, getting more and more detailed and precise.

A.9 Further notations for modelling global behaviour

A.9.1 Path Expression Groups

Path Expressions Groups (PEGs), introduced in [Adam94] and [Adam92], are used for the description of essential interaction patterns within a system and between a system and its environment.

The interactions of a group of cooperating objects are described by path expressions. A **group** contains several object instances that cooperate to accomplish a task. Groups are orthogonal to classes. A **path** defines a set of permissible sequences of such a group, i.e.

8. The responsibility of an object includes two key items: the knowledge an object maintains and the actions an object can perform [Wirfs90, page 61] and [Booch94, page90].

it captures the essential interaction patterns within the group and specifies its global behaviour.

Notation

Paths are described by regular expressions over object operations and other paths (see figure 152). The possible orders are defined by the symbols ";" for sequence, "|" for alternation, "*" and "+" for repetition and "[ ]" for optional paths. Interleaving or parallel object operations or paths cannot be modelled. The typical initiator of a path (most often external to the group for which the path is specified) is modelled by the symbol ">>" in front of the path expression. Arguments passed to the operations are not specified in the paths for the reason of simplicity.

Groups are considered as first class objects. Their declarations contain the following elements:

- *name* and *textual description,*
- *inheritsFrom:* a group inherits the path expressions from its super-group,
- *primary participants*: the objects which are components of this group,
- *secondary participants*: other objects that interact with this group as clients or as servers,
- *invariants*: important dependencies between the objects,
- *behaviour*: high level behaviour defined by a path expression composed of detailed behaviours; thus any client of the group can quickly grasp the typical behaviour,
- *paths*: typical/essential interaction patterns, detailed behaviour,
- *exception paths*: atypical interaction patterns.

Each group may contain other groups, and thus its behaviour is a composition of the behaviour of its components. Also, each path expression can be composed by other path expressions which may also be path expressions over component groups. Thus composition of groups and composition of path expressions goes hand in hand. When taking all those detailed behaviours that are typically initiated by the user, the external visible functionality of the system can easily be derived.

Using the PEG notation

PEG is a notation for capturing and describing object interaction patterns. It is not a system development method but may be used as enhancement to traditional notations[9]. Adams considers path expressions as being far more usable for documenting object

9. BON recommends to use PEGs for complex dynamic systems where a full dynamic model is necessary and thus object scenario diagrams are not sufficient [Walden95].

interaction than state charts, which focus only on one object type, or event traces, which capture only one single sequence of messages, though latter can be easily deducted from path expressions.

A.9.2 Scenario trees

Scenario trees are used for modelling user views in a formal way. They are described in [Hsia94].

Scenarios and scenario schemas: A scenario is a "possible way to use the system to accomplish some function the user desires", a "sequence of event types that accomplishes a functional requirement". Scenario schemas are the descriptions of scenario types, scenario instances are instantiations of scenario schemas. A scenario schema corresponds to exactly one path in a scenario tree, each alternative course of event types results in a different scenario schema. Events that are optional or iterated are not possible within one scenario schema, several scenarios are necessary to model them. One of the possible scenario schemas of the scenario in tree in figure 140 is the following list of input and output events:

```
Off_H, Not9, digit, digit, digit, Busy, On_H
```

Each scenario starts in the initial state of the system and ends, when this initial state is reached again. The states are modelled in such a manner that one scenario corresponds to one functional requirement.

User views and scenario trees: A user view is "a set of specific scenarios as seen by a certain user group or by users that employ the system in a like manner". A scenario tree "describes and represents all the scenarios for a particular user view". Examples for scenario trees are found in figure 140 and 141. A scenario tree consists of states (nodes) and events (transitions). Only those input and output events are mentioned, that cause a state transition. The attributes of the events are not specified in the scenario trees. The root of the tree is the initial state, each leaf is again the initial state. The graph has the form of a tree because the same states can be mentioned several times.

Event and event types: Events are "specific stimuli that change the system state, trigger another event, or do both. An event... can be both an input and response, internal and external to the system." An event can have attributes (parameters). An event type "defines all possible events with similar attributes".

Assumptions for the use of scenario trees:

There are several assumptions that limit the use of scenario trees and of this kind of scenario analysis:

- The system must be sequential, no concurrent stimuli (input events) and responses (output events) are allowed. Only a single response can be taken into consideration for each stimulus.

- Each scenario is described individually for each agent, i.e. one scenario does not show the interactions with several agents but only between one single agent and the system. Thus the scenarios are strictly modelled for one single user's viewpoint. In this aspect, the scenario trees of Hsia differ from most other methods (usually a scenario contains input and output events from and to different agents).
- The starting and ending state of the system must be the same for each scenario. Scenarios for system initialisation cannot be modelled. Further the behaviour of the system under consideration ideally consists of a flat structure of independent scenarios, each scenario having a well defined sequence of events without any alternative courses.
- Concurrent or interacting user views cannot yet be checked against each other for deadlocks.

The goal of using scenario trees in the requirements specification process is to *generate, analyse and validate dynamic requirements in a systematic and formal way.* It is only applicable to requirements that can be easily expressed as a sequence of fine-grained events normally on a technical level (such as input of individual digits), where inconsistencies and modelling errors are a great risk for the project (the damage caused by errors justifies high prevention costs), where the sequences of events are very complex (as it is typically the case in transmission systems such as PBX or network protocols) and where the requirements are known at the time of requirements determination and will only change in minor details.

Process for developing scenarios:

The process for the identification, formalization, verification and validation of scenarios contains the following steps:

- *Scenario elicitation:* Building for each user group a scenario tree. All scenarios for one single agent are grouped into one user view and a scenario tree is created.
- *Scenario formalization:* Converting each scenario tree into a *regular grammar* and a *conceptual state machine*, one per user view (see figures 140 and 141). A conceptual state machine is a deterministic finite state machine that has exactly one initial and one terminating state which are identical.
- *Scenario verification*: Automated checking the grammar for internal incompleteness, redundancies and inconsistencies. If errors are detected, the previous steps are repeated.
- *Scenario generation, prototype generation and scenario validation*: All possible scenario schemas are extracted, a prototype is built automatically from the conceptual state machine, and the user validates the scenarios with the aid of the schemas and of the prototype.

The user view is first written down in the form of a scenario tree and not directly in the form of a state chart because scenario trees match better the way how a user sees its interactions with the system, and because the sequence of the events is directly visible. The scenario trees are not necessarily consistent; but possible inconsistencies are detected with the regular grammars and the conceptual state machines. The user is involved significantly in the scenario elicitation and scenario validation, and also to some part in the formalisation and verification.

A.9.3 Composition of scenarios based on statecharts

In [Glinz95] a behavioural model showing the external view of a system is made. A formal notation is used in order to enable various consistency checks and to allow simulation or automatic prototyping of the behavioural model.

A **scenario** is defined as a sequence of interactions between a user and a system. All the scenarios in a scenario model must be disjoint. In case some scenarios are overlapping, they must first be transformed into several disjoint scenarios or into one single scenario. Each scenario is modelled by a state chart, this state chart is closed, i.e. it has exactly one entry and one exit state (see figure 148). The transitions are triggered by external or internal events and have actions. These can again be interpreted as internal events and can trigger other transitions. Events are broadcast to all state transitions. A transition takes an arbitrarily small time interval, thus the model is quasi-synchronous. This avoids certain nondeterministic behaviour that could otherwise occur.

Disjoint scenarios can be composed and integrated into a complete model of the system behaviour (example see figure 147). The following four compositions are possible: **sequence**, **alternative**, **iteration** and **concurrency**.

Various consistency checks are possible, even if some scenarios are not yet specified by a decomposition or a closed state chart (these scenarios are replaced by dummy scenarios). Deadlocks can be detected, the reachability of states can be checked, the required mutual exclusions of scenarios can be verified, inconsistencies in the names of events and actions can be detected (by executing the model) and the model can be checked for completeness (missing specifications of certain parts, missing behaviour when executing the model).

A.9.4 CRC cards

CRC cards, introduced by Cunningham [Beck89] in order to help students to learn the essentials of object-oriented programming, have become widely used in many methods (e.g. also by Wirfs-Brock, who has made them widely known by her book on responsibility driven design [Wirfs90]). Here we base our summary on the description of CRC-card by [Wilkinson95]. In this book not only the cards themselves are introduced, but also an informal development process that uses the cards for analysis and design is described.

Basically, a CRC card is an index card. There exists one card for each class. On its front side the card has the name of the class as well as a table of responsibilities (left) and collaborators (right). In each row, one responsibility offered by this class is listed, together with the names of the collaborating classes this class has to contact in order to fulfil this responsibility. Classes, responsibilities and collaborators are considered as the most essential concepts of an object-oriented model. During analysis, only the aggregate responsibilities are listed; these are the responsibilities as they are needed by a user. Also, only the name of the collaborating classes are listed. A short textual description of the class is added to the back of the card. In the design, subresponsibilities (corresponding to the various steps the object has to take to provide the desired results) are added. The collaborators are annotated with the name of the responsibility that is needed (see figure 143). Further details such as class attributes are added to the back of the card.

The model is developed in CRC card sessions. There a small group of people (developers and users) develop the model. The sessions are often driven by scenarios. These are used in walk-throughs to detect new classes and to verify the model. A **scenario** is defined as a detailed example of a function of the system. A function is considered as a single, visible, testable behaviour. In analysis, these functions are analogous to system requirements, seen from a high level user point of view. For each function, a set of scenario exists that explores what would happen given different parameters."

The static view of the resulting model is documented by a class model that shows the collaborations and subtyping relationships between the classes (see figure 145), and class descriptions that correspond to the information on the class cards. The dynamic view is given by scenario descriptions. These are tables that show for each step of the scenario the client, the server, and the responsibility demanded of the server[10] (see figure 144). A scenario can also call another scenario. In this case, instead of the responsibility name, the scenario name is entered in the table. There is no rule as to what makes up one scenario, any activity of the application that requires collaboration can be considered as being a scenario. Yet in order to avoid redundancy, short scenarios are preferred. Whenever a scenario may be initiated by several client objects, the name of the client object is left open.

A.9.5 Requirements scripts in OSMOSIS

OSMOSIS is an experimental CASE-tool that supports also the modelling of scenarios in the form of requirements scripts [Berteaud95]. It is assumed that requirements are gathered in the form of manuals, work descriptions, workshop notes or interviews. These are called sections and are together the document of informal requirements. When building the requirements model, in a first step **requirements scripts** (RS) are derived from the sections. In a second step, the scripts are then reworked in order to avoid redundancies and incompatibilities. Also, the actors for the major scripts (not all scripts have an external actor) are identified, the relations between these scripts are defined, the values

10. These tables are very similar to the scripts of OBA, though begin and end of scenarios are defined differently.

of the attributes (characteristics or aspects of the RS) are defined, the objects (data entities) used by the scripts are identified, and where necessary the scripts are decomposed into further scripts.

The OSMOSIS CASE-tool provides browsers to view the resulting semantic network. This network has the nodes document, section, requirements script, actor and object as well as links among these concepts and between the concepts and textual attributes (which are of type String). The following list shows all the possible links, a * denotes a set of links:

writer [RS, String]	performance [RS, String]	partOf [RS, RS]
date [RS, String]	security [RS, String]	next* [RS, RS]
status [RS, String]		previous* [RS, RS]
	what[RS, String]	inheritFrom [RS, RS]
identifiedBy [RS, String]	preConditon[RS, String]	composedOf* [RS, RS]
fromDocument [RS, Document]	postCondition[RS, String]	
fromSection [RS, Section]	who*[RS, Actor]	etc.
	toScripts*[Object, Script]	

The link “what” describes the objective of an RS (normally just one sentence). The detailed description of the behaviour is given by the decomposition of the RS into further RS’ and not by the link “what”. The focus of the modelling process is always “on what the system realizes for its users rather than how it does it.”

A.9.6 Storyboarding

Storyboarding proposed by [Umphress91]

OORD (Object-oriented requirements determination) [Umphress91] uses storyboarding, together with concepts maps and event-response lists, to determine the requirements. The notations are deliberately vague, soft and fuzzy, their strength lies in their intuitiveness, understandability and simpleness. OORD knows the following notations:

- **Concept maps**: semantic layered network of information. All concepts (not necessarily data elements or objects) and links (not only static relationships) that make sense to the user are noted; there are no further modelling rules.
- **Event-response list**: a list of events, responses, constraints and maybe some conditions for the events (see figure 149). One event with its responses makes up one scenario. The event-response list shows all those interactions of the system with its environment that are visible to the user. For technical systems the events may be very low-level, yet the developer is free to use the list for any appropriate abstraction level.
- **Storyboard**: like a prototype on paper (see figure 149). The pictures show how the user interface or system state could look like and change its appearance during the scenarios. The event-response list describes the behaviour from one frame or picture of the storyboard to the next one. Whereas the event-response list should be determined as complete as possible, storyboards are only drawn for some selected scenario instances.

Object-oriented requirements determination is a front-end process to traditional object-oriented development methods. The only goal of OORD is to find and establish candidate needs and to provide a conceptual model of the problem in a manner that is closer to the object-oriented paradigm than a functional description. The notations make the communication between developers and clients easy and provide also the basis for the decisions concerning the automation boundary of the target system.

The reconciliation of the candidate needs and their checking for completeness and consistency is left to the process of requirements analysis, which will then result in the requirements specification document. In the requirements analysis the concepts of the concept maps become potential classes. The storyboard gives hints for the allocation of operations to the classes. The events and responses help to validate the completeness of the behaviour.

A very similar approach is proposed in [Duffy95]. The object-oriented requirements analysis model is also documented by concept maps and event-response list. The model is used as a front end to OMT.

Storyboarding proposed by [Zorman95]

[Zorman95] describes the tool REBUS that allows us capturing and documenting scenarios in a storyboard-like representation. Each scenario consists of various pictures (called frames), and for each scenario various concepts and relations can be specified. These so called *within* scenario relations are objects, measures, spatial elements, temporal elements and behavioural elements. The scenarios are considered as being a natural mean of representing and capturing domain knowledge during requirements envisaging. The definition for scenarios is taken from [Benner93]: “Scenarios are partial descriptions of system and environment *behaviour* arising in restricted *situations*." Partial description means that they need not completely specify all the states that comprise a behaviour, nor need they completely specify all the attributes of any given state, only what is relevant for the chosen representation. The concepts behaviour and situation are clearly distinguished, but both, situation and behaviour are a partial ordering of states or transitions. The focus can be either on the sequence of states, or on the sequence of transitions.

A.9.7 Business processes (“Geschäftsvorfälle”)

In certain problem domains, especially banks and insurance companies, the global behaviour is modelled by business processes (business transactions, banking operations, “Geschäftsvorfälle”). These models are well suited to define how the enterprise works and what services it offers, but the question arises, how these models are combined with object-oriented software development. In [Mueller93] it is discussed how these business processes can be mapped onto an object-oriented analysis model.

Characteristics of business processes in [Mueller93]

A “Geschäftsvorfall”, we translate it here by business process, is an autonomous process or activity within the enterprise. It may be triggered internally or externally. A business process consists of certain building blocks, also called actions or business processes (“Geschäftsprozesse” in contrast to “Geschäftsvorfälle”). These may be manual or automated. The business processes are modelled on a conceptual level that does not consider any software system related aspects. The building blocks are logically and temporally related, but they cannot be triggered on their own. The duration of a business process is not limited in time, it may also be interrupted to call other business processes, or to get further information. In a software development project, the model of the business processes is made together with a semantic data model as part of the first step of analysis. They show the functionality of the system from the user’s viewpoint. But models of business processes may also be used outside of software development. An example for a graphical description of a business process is shown in figure 150.

Mapping business processes onto classes of an object-oriented system

There are two possibilities of mapping business processes onto classes. It is assumed that all the persistent information is modelled by entity classes. The first possibility is to allocate each business process as an operation to a specific entity class. The second possibility is to model each business process as a class itself (so called process classes). The first mapping handles well all those business processes that are very short, cannot be interrupted and have no persistent data that concerns the business process itself. But if for a certain business process there may exist at the same time several instances that concern the same entity instance, the business process must be modelled as a process class. If it were allocated as an operation to the entity class, then the operations would have to be re-entrant. Furthermore, whenever several business processes have the same operations and attributes, the business processes could be modelled by an inheritance hierarchy in order to allow reuse. Because inheritance is only possible between classes and not between the operations of a class, also in this case the business processes are modelled as business classes.

A.9.8 Extending OOSE by use case levels

The approach of [Armour95]:

In [Armour95], use cases are modelled on several abstraction levels. The highest level consists of the **high level use cases** that are grouped into **functional areas**. These functional areas organize the system behaviour by functionality. A high level use case is described by its business event, which is initiated by an actor and reflects a responsibility of the system. Each high level use case is detailed by an **expanded use case**. Expanded use cases contain a complete use case descriptions as known from OOSE, with the exception that normally only the basic course is shown. Also pre- and postconditions are added. Each expanded use case is once again detailed into one or several **detailed use cases**. There may also be several levels of detailed use cases. On the level of detailed use cases also alternative and exceptional courses are specified, redundant parts of the use cases are factored out into abstract use cases, and conditional logic is used in the description. Furthermore, **use case dependency diagrams** show the dependencies between the use cases. They are used for verifying the pre- and postconditions, for deriving work flow models, or for giving a better understandability for newcomers to the use case documentation.

Use case modelling is advocated in [Armour95] for software system development as well as for business modelling and business reengineering. The process of use case modelling is subdivided into various steps, starting with the context diagram and the high level use cases, ending with the refinement of the detailed use cases. When modelling the software system, then the high level use cases are meant to be understandable to the users, whereas the most detailed use cases may contain the minute details of a user interaction by a GUI interface.

The approach of [Regnell96]:

In [Regnell96], a hierarchical use case model is suggested providing three levels of use cases, each level having its own notations. The highest level is the **environment level**. Use cases are identified and associated with services, actors and goals. “A service is a package of functional entities (features) offered to the users in order to satisfy one or more goals that the users have. ... A use case models a usage situation where one or more services of the target system are used by one or more users with the aim to accomplish one or more goals. A use case may either model a successful or an unsuccessful accomplishment of goals." Furthermore, use cases can be organized into packages according to the services. The use cases are described by context diagrams, which show use cases, actors, and packages.

At the **structure level**, use cases are divided up into episodes, using sequencing, alternatives, repetitions, exceptions and interrupts. The diagram is like a flow chart, showing for each use case its pre- and postconditions as well as its episodes. Repetition, exception and interrupts are shown by annotations to the episode symbol. Each episode can again be decomposed into further episodes, for this a Jackson-like structure chart is used.

Events and scenarios only come in at the **event level**. For each use case an interaction diagram in time-line notation is made. It shows the events between the system and its actors. Repetition etc. are shown by rectangles drawn around the whole part of the diagram that is repeated. There exists also a rich set of symbols for activating and deactivating timers. And **event** is defined as being either a stimuli (a message from a user to the target system), a **response** (a message from the target system to a user), or an action (an event inside the system). A **scenario** is defined as being “a realisation of a use case described as a sequence of a limited number of events with linear time order." Two different grades of scenario instantiation are distinguished: scenarios with formal parameters and scenarios with specific parameter values.

The proposed notation is an extension to the telecommunication norm ITU-T MSC for message sequence charts.

A.9.9 Use case maps

In order to bridge the gap between the use cases of the requirements level and the interaction and visibility graphs of the detailed design, Buhr proposes use case maps [Buhr95]. Use case maps model large-grained behaviour patterns. They can be used for detailing use cases as well as for abstracting design patterns. Use case maps target at a system view and are a tool for reasoning and documenting the high-level design. They show chains of causally-related responsibilities. Details such as the interfaces of components or effective interactions among components cannot be shown.

The graphical part of a use case map (example see figure 121) contains one or several paths through a use case. These paths are cause effect chains through the system. Along the path responsibilities are shown. These are coarse-grained units of activity. The responsibilities may be bound to components. Components are only characterized by the responsibilities, no interfaces are defined. In an unbound use case map, responsibilities are not yet allocated to any components, and components are not yet shown.

The textual part of a use case map contains among other things the names of the responsibilities, the names of the input parameters of the stimulus of the use case, and pre- and postconditions for interpath coupling.

A.9.10 M.E.R.o.DE

The goal of M.E.R.o.DE (Model-driven Entity Relationship object-oriented DEvelopment) [Dedene94] is to provide a formal object-oriented notation for requirements specification in order that the specification models can be verified concerning their consistency and that the implementation can be validated in respect to its specification.

Models

The **business model**, which models the exact functioning of the business by business entities, business constraints and business rules, contains the following schemas:

- for the static aspects: *entity relationship models*, *dependency graphs* and *abstract data types specifications*,
- for the dynamic aspects (see figure 139): one *object event table* for the whole system and a *structure diagram* for each object.

The object event table shows for each relevant event type (common events which may act on several objects), which object types it may create (C), modify (M) or destroy (D). The structure diagrams, which are JSD-diagrams, describe the sequence restrictions each object type imposes onto the event types which trigger methods of this object.

Structure diagrams specify the life-cycles of object instances. Based on the dynamic models, the object types are then specified as abstract data types by their state vector (containing all attributes) and their event-vector (one method for each event in which the object participates).

Further models of M.E.R.o.DE are the **scope model**, the **design model** and the **technology model**.

Consistency checking

Because the schemas of the business model can be represented by formal languages and because the schemas are strongly interconnected, consistency checking is possible. The models are checked for deadlock by deriving the global behaviour from the individual schemas: the structured diagrams are transformed into regular expressions over common event types, including operators for sequence, iteration and selection. In order to guarantee consistency, object types that have common event types need to allow the same sequence of event types in their life-cycle.

A.9.11 Behavioural models of Kowal

The behavioural models of Kowal are described in [Kowal92].

Terms

A **motive** is the cause or reason why a source sends one or several **stimuli** to the system. Stimuli are input dataflows. An **external event** is the action of presenting a stimulus to the system, the system reacts with a response, i.e. output data flows and/or internal activities. A **scenario** consists of a set of input and output dataflows, control flows and behaviours. A **behaviour** consists of one or more units of activity. Behaviours are for instance inputs, outputs, verifications, activation of another behaviour, calculations, deductions etc. Behaviours use stored data and input data flows and they achieve some inspectable results, either by internal state changes or by output dataflows. Beside the external events there exist are also **temporal events** and **anomalous events** (an occurrence of an anomaly that is recognized by the system and that requires action to preserve or restore the system to a normal condition). To show the possible orders of events, the following relationships between events can be specified:

- *Administrative relationship*: All those events that serve the administrative purposes of the same data stores (creation, maintaining, securing...) are mutually dependent.
- *Aggregate relationship:* The result from one event provides the data for another event. The two events accomplish a single objective.
- *Directive relationship*: An event issues a control flow and/or data flow that initiates an otherwise independent event.

Notations

The system behaviour is modelled on three different abstraction levels:

- **Operational view:** A *context diagram* shows the dataflows between the system and the agents and defines the problem boundaries. *Event diagrams* show for each event the information flows, the event process and the data stores. Each event process may have several input dataflows. Also the relationships between events are modelled. All the data of the dataflows is modelled in an *entity relationship diagram*.
- **Architectural view:** *Scenario diagrams* are data flow diagrams that show the processors, controlflows, dataflows and data stores involved in a scenario; there may be one or several scenario diagrams for one event. They are annotated with additional descriptions of constraints. *Scenario specifications* describe the interface and the technical requirements for each processor of a scenario. The interface definitions are done by action diagrams that show the sequence of behaviour patterns, i.e. list all inputs and outputs for each processor.
- **Behaviour view:** *Behaviour diagram***s** are dataflow diagrams that show for each behaviour pattern the processes involved. They are supplemented by the *behaviour specifications* which use action diagrams with pseudo-code to give a low-level specification of the behaviour patterns. Moreover for reactive systems *state transition diagrams* are used to specify finite state processors.

In the action diagrams graphical brackets and pseudo-code are used to model the structure of the logic which is used to transform input data into output data. The brackets can express sequence, selection, repetition and concurrency (see figure 146).

Behavioural models of Kowal and structured analysis

The method of Kowal is a further development of structured analysis overcoming many of its weaknesses. Kowal introduces dependency between events, traceability throughout all the abstraction levels, flexibility for the divergencies between the views of the problem and the structure of the final software solution, and allows a complete and consistent specification of the software system. The handling of the many different diagrams and concepts is possible, because the relations between them are well defined.

A.9.12 Further approaches

The methods mentioned above are by no means all methods having modelling techniques for showing the global behaviour. A more informal use of scenarios is also found in [Reisin90] or [Holbrook90]. Syntropy [Cook94] also models the system as a stimulus response system and shows the responses of the individual objects onto the external events which are interpreted as being broadcasted across the system. [Martin95] explicitly models the specialisation hierarchies of event types and their connection with the object operations by event diagrams. Scenario models showing the external and internal view of global behaviour are also proposed in [Cartiant95]. A more formal approach to modelling the behaviour of composite systems is proposed in [Dubois93].

Appendix B
Examples of diagrams from various methods

This appendix contains examples of diagrams from various methods. These diagrams are primarily referenced in chapter 2.1 and in appendix A. We only have included them here in order to facilitate the reading of these chapters. The references of the publications from which the diagrams have been scanned in are given in the figure captions. Additional explanations to the diagrams are also found in these references. A list of all the diagrams in this appendix is found in the list of figures at the beginning of this thesis.

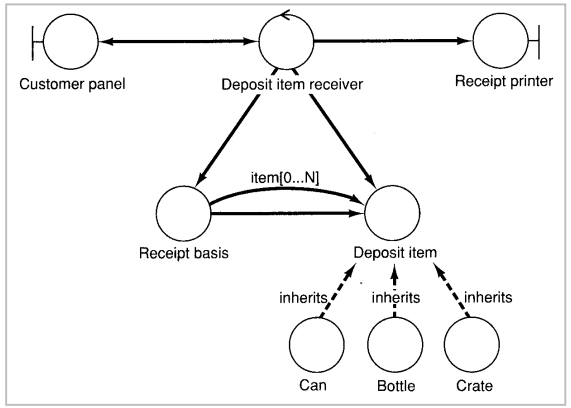


Figure 107: Object model with communication associations (arrows with no labels) in OOSE [Jacobson92, page 189]

Use case "customer withdrawal":

(1) The office personnel insert a request for a customer withdrawal at a certain date and a certain warehouse. The window in Figure 13.7 is shown to the office personnel when giving the command issuing a customer withdrawal.

(2) The information filled by the office personnel is Customer, delivery date and delivery place. The name of the office personnel person is filled from the beginning but it is possible to change it. The office personnel person selects items from the browser and adds the quantity to be withdrawn from the warehouses. The browser can here show only the items of the current customer.

(3) The following criteria are checked instantly:

(*a*) The customer is registered,

(*b*) The number of items ordered exist in any warehouse.

(*c*) The customer has the right to withdraw the items.

(4) The system initiates a plan to have the items at the appropriate warehouse at the given date. If necessary transport requests are issued. The items are reserved three days in advance. (The longest possible time for a warehouse redistribution has been measured to take three days.)

(5) At the date of delivery a warehouse worker is notified of the withdrawal.

(6) The warehouse worker issues requests to the forklift operators to get the items to the loading platform. The forklift operator executes the transportation.

(7) When the customer has fetched his items the warehouse workers marks the withdrawal as ready. The items are removed (decreased) from the system.

Alternative courses

There are not enough items in the warehouses

The office personnel are notified and the withdrawal cannot be executed.

Customer has no right to withdraw an item or Customer is not registered.

Notify the office personnel. The withdrawal cannot be executed.

Figure 108: Description of a use case in OOSE [Jacobson92, page 349]

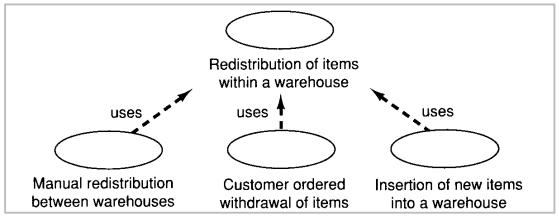


Figure 109: Modelling the relationship between abstract and concrete use cases in OOSE [Jacobson92, page 343]

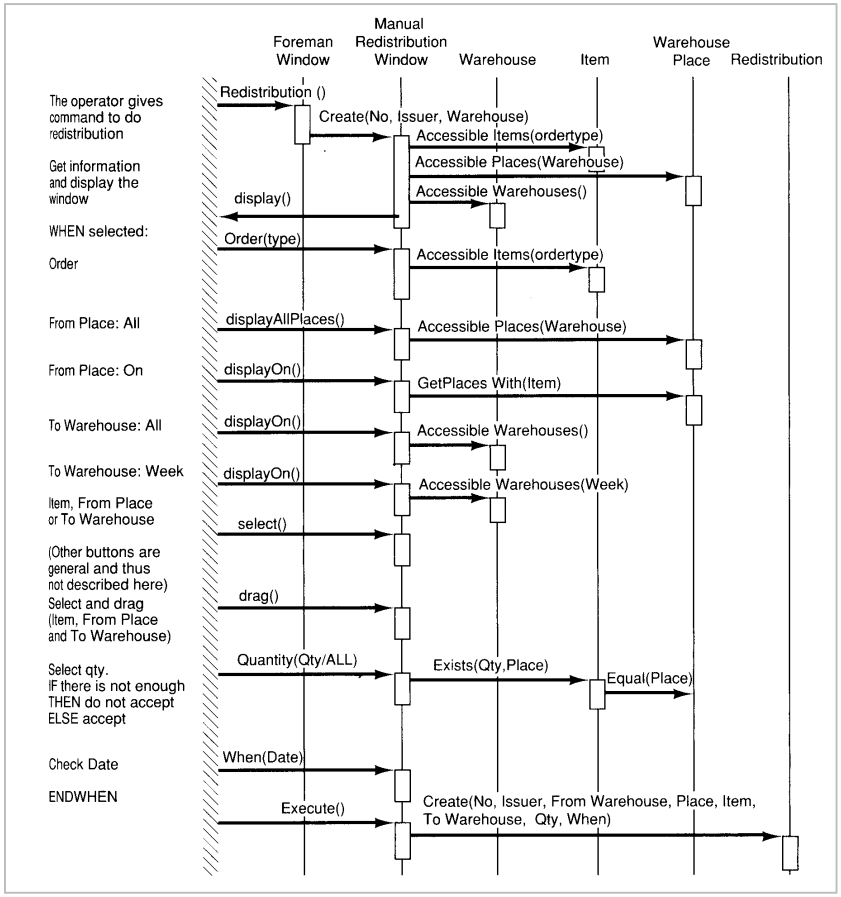


Figure 110: Interaction diagram in OOSE (example with two interface objects) [Jacobson92, page 381]

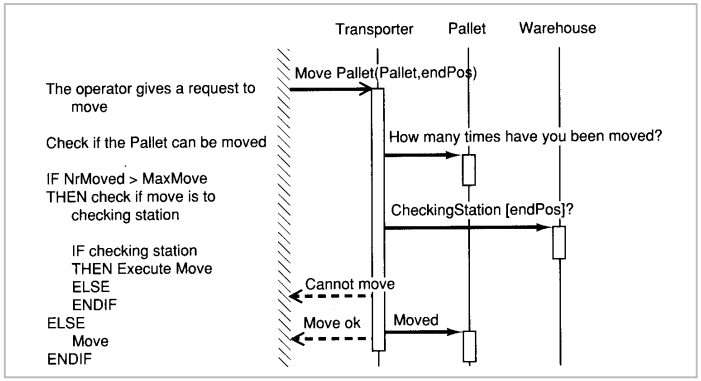


Figure 111: Interaction diagram in OOSE (output signals) [Jacobson92, page 218]

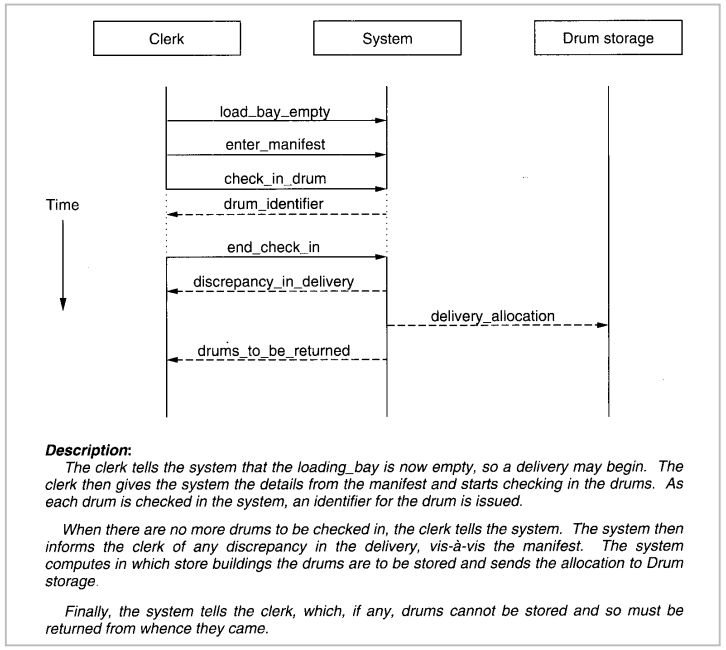


Figure 112: Event trace diagram in Fusion (analysis model) [Coleman94, page 47]

```
lifecycle Banking_system: Initialization . ((Transaction | Enquiry)* || PrivEnquiry*)

Initialization = open_account. #account_number . Enquiry* . deposit_money
Transaction    = withdraw_cash . (# dispense_cash | # insufficient_funds) |
                 deposit_money
Enquiry        = check_balance . #current_balance
PrivEnquiry    = authorize. (#confirm | #deny).
                 (inquire. #amount)* . finish
```

Figure 113: Life-cycle expressions in Fusion (analysis model) [Coleman94, page 33]

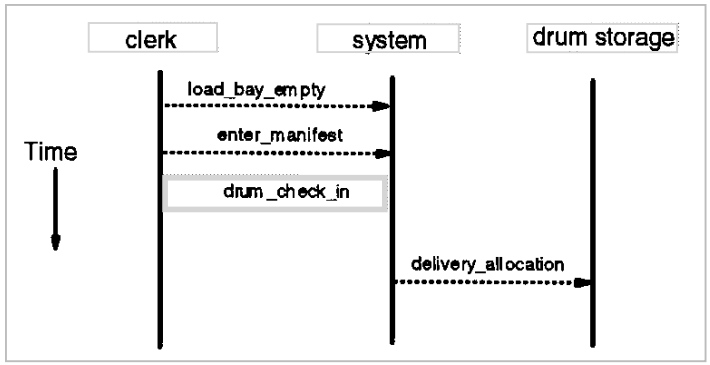


Figure 114: Scenario diagram of extended Fusion showing a use case instance with a sub use case [Coleman95]

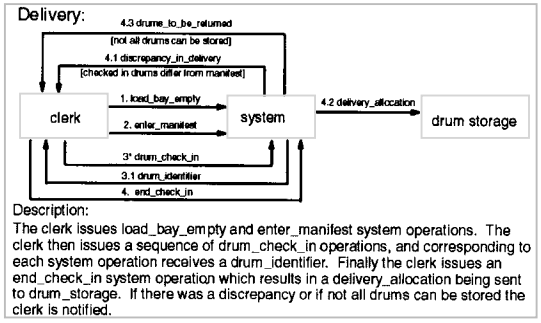


Figure 115: System interaction graph in extended Fusion [Coleman95]

Operation:	withdraw_cash
Description:	Requests an amount of cash to be taken from a given account. Cash is dispensed only if account has sufficient funds.
Reads:	**supplied** acc_number:accountnumber, **supplied** request:money
Changes:	account with account.number equal to acc_number (*i*)
Sends:	customer: {insufficient_funds}, cash_dispenser: {dispense_cash}
Assumes:	acc_number is a valid account number, and the account is not overdrawn.
Result:	If (Initial account.balance ≥ request) then account.balance has been reduced by request, and dispense_cash amount has been sent to cash_dispenser. Otherwise, insufficient_funds has been sent to customer. The account is not overdrawn. (*ii*)

Figure 116: Schema of a system operation in Fusion (analysis model)[Coleman94, page 31]

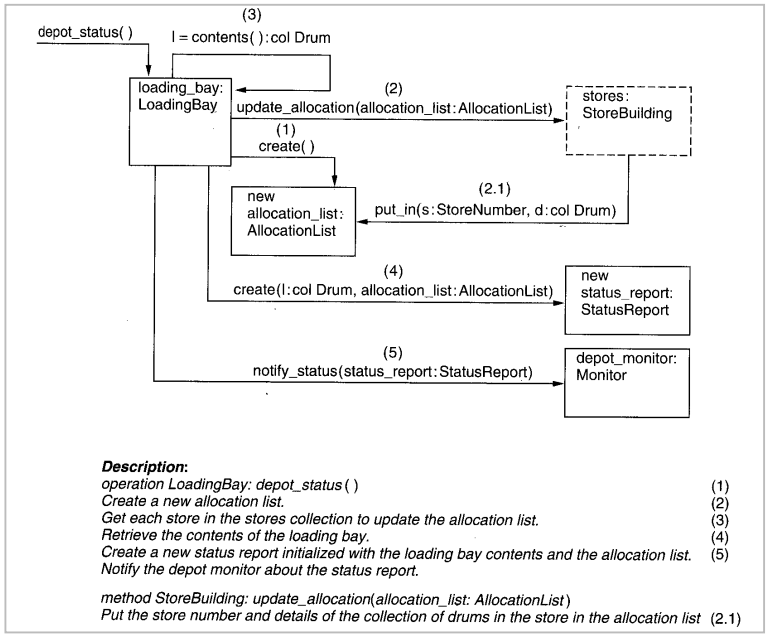


Figure 117: Object interaction graph in Fusion (design model) [Coleman94, page 75]

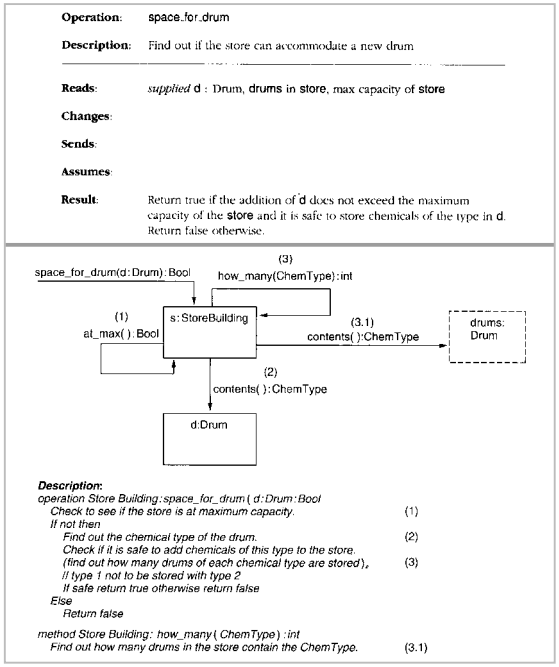


Figure 118: Decomposition of object interaction graphs in Fusion [Coleman94, page 79]

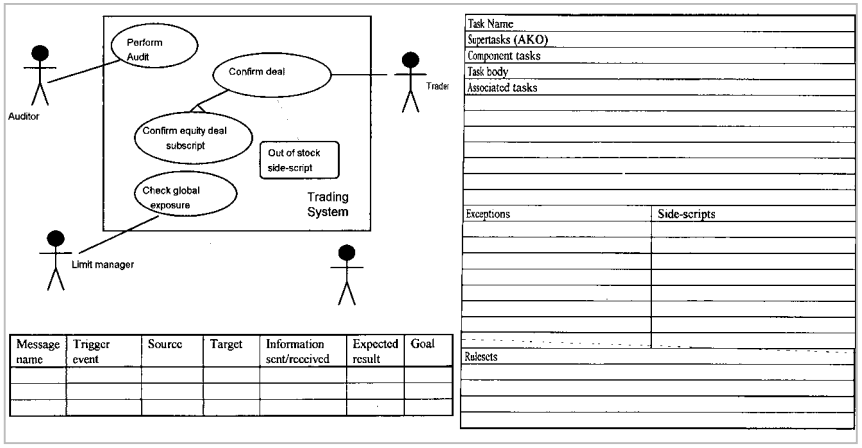


Figure 119: Context model, message table and task cards in FORAM [Graham94b]

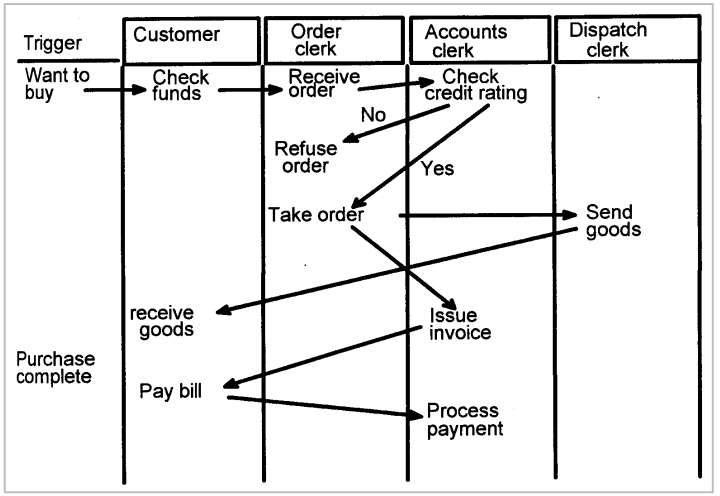


Figure 120: Event trace in FORAM [Graham94b]

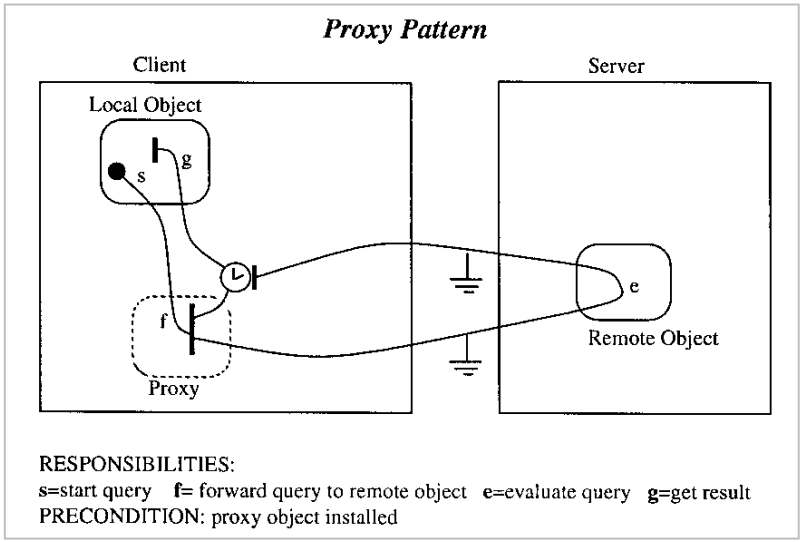


Figure 121: Use case map for a design pattern [Buhr96]

N	TYP	EREIGNIS	BEDINGUNG	ANTWORT
1	(F)	Kunde verlangt Auskunft über Hotels, Flüge, Destinationen		Auskunft über Destinationen, Flüge, Hotels, Hotelbelegungen
2	(F)	Kunde bucht Angebot		Bestätigung an Kunde u. Buchhaltung Flug buchen, Hotel reservieren
3	(F)	Zahlungsavis trifft ein	Ticket vorhanden	Reiseunterlagen u. Ticket an Kunde
			Ticket nicht vorh.	intern
4	(F)	Kunde annuliert Reise		Flug u. Hotel annulieren, Annulation an Buchhaltung
5	(F)	Adressänderung eines Kunden		intern
6	(F)	neue Angebote, Adressaenderungen von Hotels, Fluggesellschaften		intern
7	(F)	Ticket trifft ein	Zahlung eingegangen	Reiseunterlagen u. Ticket an Kunde
			Zahlung nicht eing.	intern
8	(T)	Kunde zahlt nicht		Flug u. Hotel annulieren Annulation an Buchhaltung
9	(T)	alte Buchungen, Kundendaten		intern

Figure 122: Event list in Modern Structured Analysis [Beringer92c]

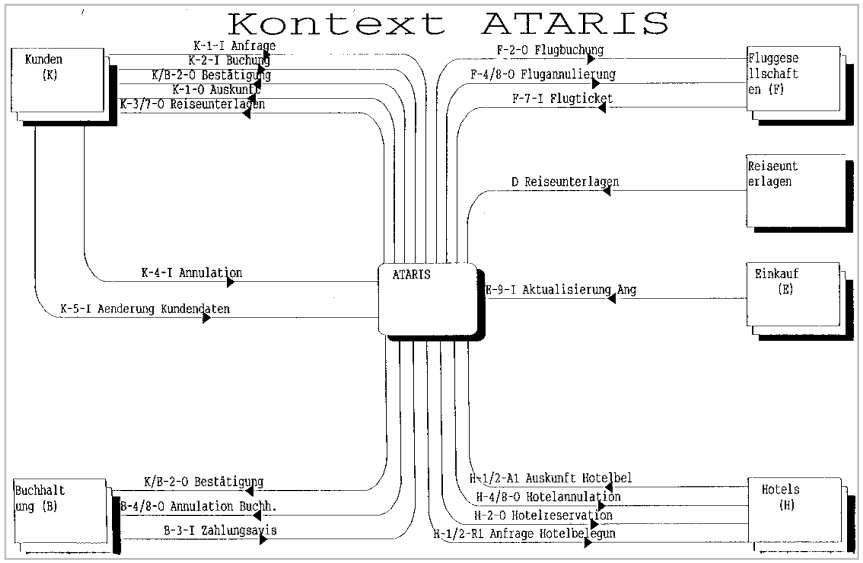


Figure 123: Context diagram in Modern Structured Analysis [Beringer92c]

Script notation:

Initiator	Action	Participant	Service
thing1	notifies	thing2	thing2 can be notified
thing1	provides info to	thing2	thing2 can accept info
thing1	requests info from	thing2	thing2 can provide info
thing1	requests service from	thing2	thing2 can provide service

Example for a user script:

Script Name Modification.1.example
Author Donna
Version 1.0
Precondition exists (Spreadsheet), displayed (Spreadsheet)
Postcondition modified(Spreadsheet)
Trace Core Activity—Modification

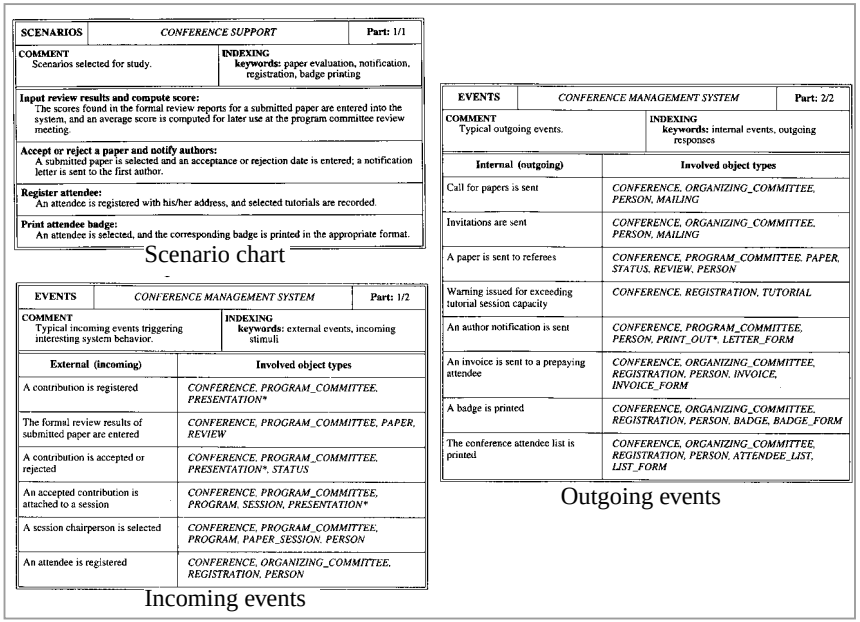
INITIATOR	ACTION	PARTICIPANT	SERVICE
User	select D1	Spreadsheet	select a cell
User	type text NEW	D1	set content to text
User	set text style to bold	D1	set text style to bold
User	select A2	Spreadsheet	select a cell
User	type text NAME	A2	set content to text
	(repeated select and type text for example)	B2, C2, D2, A3 through A10	
User	select Row 2	Spreadsheet	select a row
User	set text style to bold	Row 2	set text style to bold
User	extend row height to 34 pixels	Row 2	resize height
User	select A12	Spreadsheet	select a cell
User	type text TOTALS	A12	set content to text
	(repeat select and type)	A13, A14	
User	select A12:A14	Spreadsheet	select vertical collection of cells
User	set text style to bold	A12:A14	set text style to bold
User	select B3	Spreadsheet	select a cell
User	type number 55000	B3	set content to number
User	choose format $xx,xxx	B3	set format to currency
User	select B3:B10	Spreadsheet	select vertical collection of cells
User	copy first cell into rest of cells	B3:B10	fill down
User	select B4	Spreadsheet	select a cell
User	replace 55 by 60	B4	set content to text
	(repeat rest of changes)	B5:B10	
User	select Column A	Spreadsheet	select a column
User	contract column width to 30 pixels	Column A	resize width

Figure 124: Scripts in OBA [Rubin92]

> " Der Berater holt dann seinen *Beratungsordner*, *sucht* das *Produkt* über ein *Register* und *öffnet* den Ordner an der *entsprechenden Stelle*. Zum Beratungsordner gibt es noch einen *Formularordner*, in dem *Standardformulare* (z.B. *Vertrag zugunsten Dritter*) *abgelegt* sind und einen *Musterordner*, in dem *Ausfüllhilfen* für Verträge und *Schlüsselblätter* *abgelegt* sind ."

Ausschnitt aus einem Szenario

Figure 125: Textual description of a scenario [Kilberth93, page 99]



SCENARIOS	CONFERENCE SUPPORT	Part: 1/1
COMMENT Scenarios selected for study.	INDEXING keywords: paper evaluation, notification, registration, badge printing	

Input review results and compute score:
The scores found in the formal review reports for a submitted paper are entered into the system, and an average score is computed for later use at the program committee review meeting.

Accept or reject a paper and notify authors:
A submitted paper is selected and an acceptance or rejection date is entered; a notification letter is sent to the first author.

Register attendee:
An attendee is registered with his/her address, and selected tutorials are recorded.

Print attendee badge:
An attendee is selected, and the corresponding badge is printed in the appropriate format.

Scenario chart

EVENTS	CONFERENCE MANAGEMENT SYSTEM	Part: 1/2
COMMENT Typical incoming events triggering interesting system behavior.	INDEXING keywords: external events, incoming stimuli	

External (incoming)	Involved object types
A contribution is registered	CONFERENCE, PROGRAM_COMMITTEE, PRESENTATION*
The formal review results of submitted paper are entered	CONFERENCE, PROGRAM_COMMITTEE, PAPER, REVIEW
A contribution is accepted or rejected	CONFERENCE, PROGRAM_COMMITTEE, PRESENTATION*, STATUS
An accepted contribution is attached to a session	CONFERENCE, PROGRAM_COMMITTEE, PROGRAM, SESSION, PRESENTATION*
A session chairperson is selected	CONFERENCE, PROGRAM_COMMITTEE, PROGRAM, PAPER_SESSION, PERSON
An attendee is registered	CONFERENCE, ORGANIZING_COMMITTEE, REGISTRATION, PERSON

Incoming events

EVENTS	CONFERENCE MANAGEMENT SYSTEM	Part: 2/2
COMMENT Typical outgoing events.	INDEXING keywords: internal events, outgoing responses	

Internal (outgoing)	Involved object types
Call for papers is sent	CONFERENCE, ORGANIZING_COMMITTEE, PERSON, MAILING
Invitations are sent	CONFERENCE, ORGANIZING_COMMITTEE, PERSON, MAILING
A paper is sent to referees	CONFERENCE, PROGRAM_COMMITTEE, PAPER, STATUS, REVIEW, PERSON
Warning issued for exceeding tutorial session capacity	CONFERENCE, REGISTRATION, TUTORIAL
An author notification is sent	CONFERENCE, PROGRAM_COMMITTEE, PERSON, PRINT_OUT*, LETTER_FORM
An invoice is sent to a prepaying attendee	CONFERENCE, ORGANIZING_COMMITTEE, REGISTRATION, PERSON, INVOICE, INVOICE_FORM
A badge is printed	CONFERENCE, ORGANIZING_COMMITTEE, REGISTRATION, PERSON, BADGE, BADGE_FORM
The conference attendee list is printed	CONFERENCE, ORGANIZING_COMMITTEE, REGISTRATION, PERSON, ATTENDEE_LIST, LIST_FORM

Outgoing events

Figure 126: Event charts and scenario charts in BON [Walden95, pages 253-255]

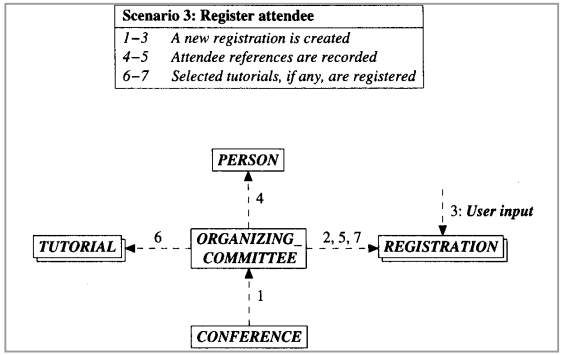


Figure 127: Object Scenario in BON [Walden95, page 259]

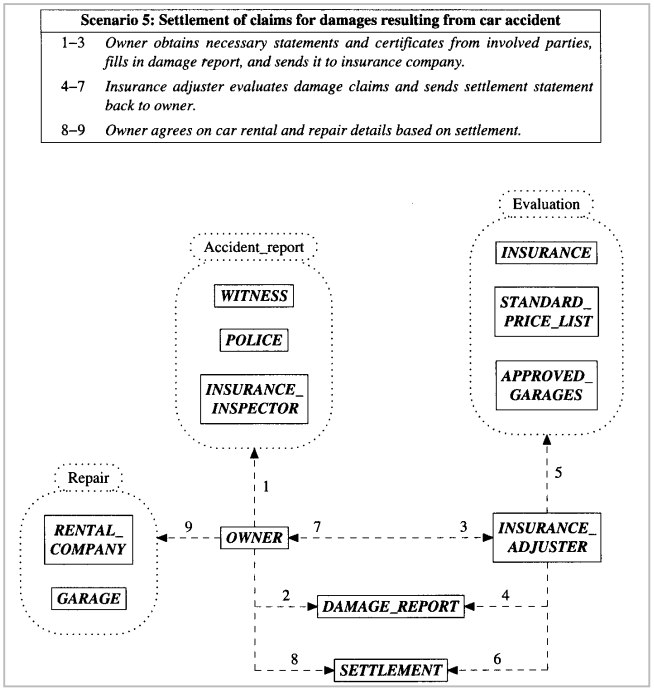


Figure 128: Object Scenario in BON (with grouping of objects into several subtasks) [Walden95, page 114]

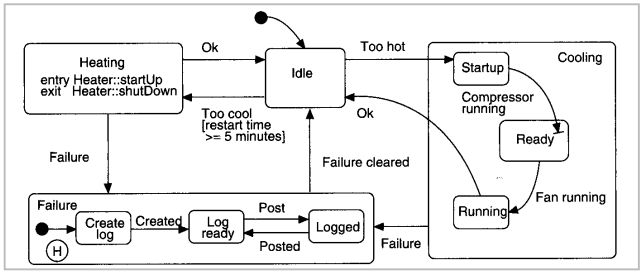


Figure 129: State chart as used in Booch [Booch94, page 207]

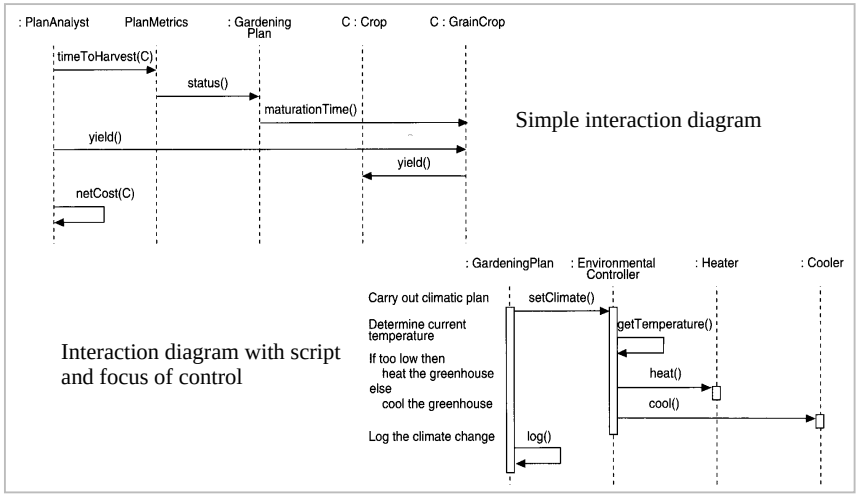


Figure 130: Interaction diagrams in Booch [Booch94, page 218]

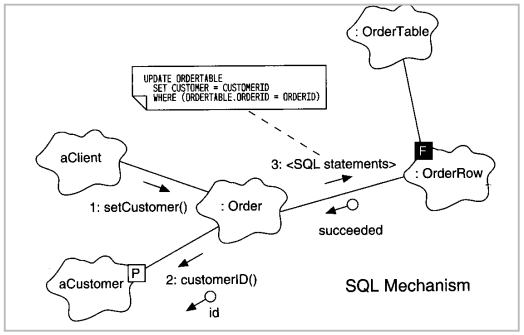


Figure 131: Object diagram for one mechanism [Booch94, page 402]

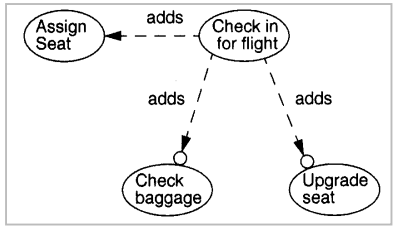


Figure 132: Add-relationships between use cases in OMT [Rumbaugh94b]

Use Case:	Assign Seat
Summary:	A passenger with a reservation on a flight requests a seat assignment. The system obtains information from the passenger and then attempts to make an assignment. The assignment is given to the passenger or the passenger is told that no assignment is presently available.
Actors:	Passenger
Preconditions:	The passenger has a reservation on a flight by the given airline
Description:	A passenger requests a seat assignment on a flight. (This may be implicit as part of checking in or may be an explicit request by the passenger.) The system (in the form of the agent) asks for the date of the flight, the flight number, departure airport, and passenger name. The passenger supplies the information. Instead of name, the passenger can supply frequent flyer number with the airline. [Exception: Too early to make assignment.] The system finds the reservation. [Exception: No reservation found.] If the reservation already has a seat assignment, it is given to the passenger who is offered the opportunity of changing the assignment. If there is no assignment or the passenger wants to change, the system requests seat preference (window, aisle) and smoking preference (yes, no). The system uses the information, including frequent flyer level, to try to find a suitable seat assignment subject to previous assignments and the policies of the airline. If necessary, the system asks additional questions: would the passenger accept a bulkhead seat, would the passenger accept a seat in an emergency exit row? The system proposes a seat assignment. If all the passenger's preferences cannot be satisified, then the system proposes the best match it can find. [Exception: No assignment possible.] The passenger can accept the assignment or ask for changes, in which case another assignment is attempted. The use case concludes when an assignment is made and accepted.
Exceptions:	Too early: Raised if the current date is too early, based on an airline algorithm that includes fare category and frequent flyer level. The passenger is advised when seat assignments will be possible and the use case terminates. No reservation found: Raised if no reservation can be found. The information is rechecked and searched if necessary for a partial match. If still no reservation can be found, then the passenger is advised to obtain one and the use case terminates. No assignment possible: Raised if no assignment is possible based on an airline-dependent formula that includes number of unassigned seats, days until departure, fare category, and frequent flyer level. The passenger is advised to obtain an assignment on check in. If this is part of check in, then the passenger is placed on standby in the order of arrival. The use case terminates.
Postconditions:	If this is part of check in, then the passenger either has a seat assignment or is on the standby queue in order of arrival. Otherwise none (the attempt may fail).

Figure 133: Description of a use case in OMT [Rumbaugh94b]

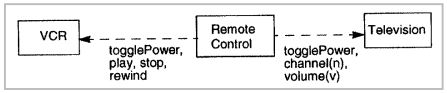
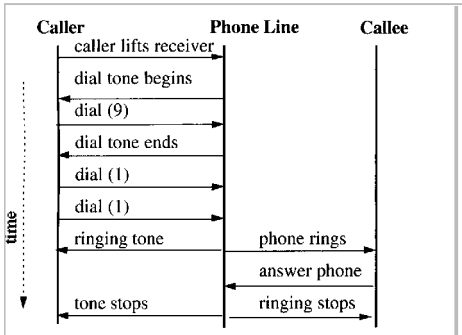
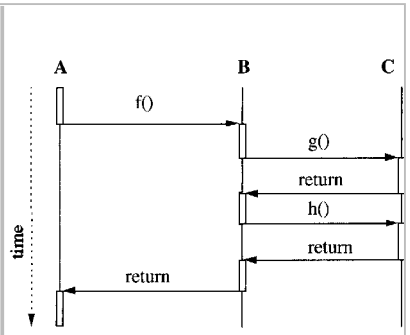


Figure 134: Event flow diagram in OMT [Rumbaugh95]



Event trace for 'phone call' with arrows that denote signals and with concrete values instead of parameter types.



Event trace with arrows that denote procedure calls.

Figure 135: Two different kinds of event trace diagrams in OMT [Rumbaugh95]

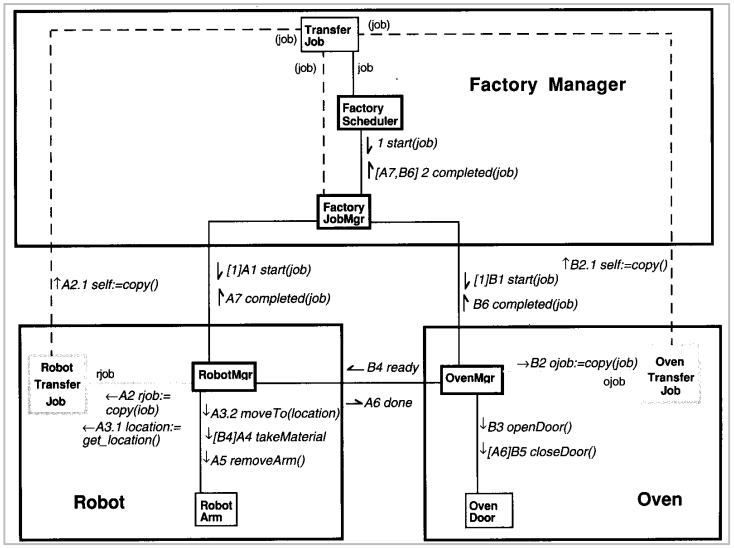


Figure 136: Concurrent object interaction diagram in OMT [Rumbaugh95b]

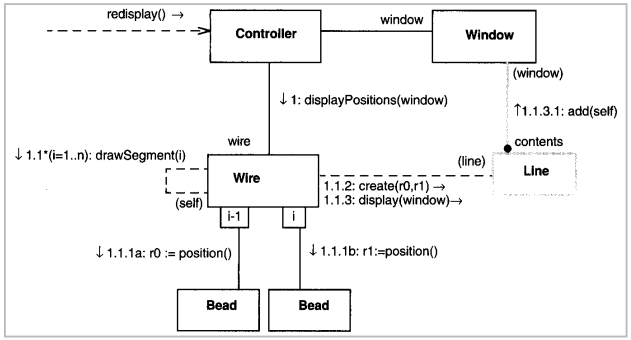


Figure 137: Object interaction diagram showing the design of the operation 'redisplay' [Rumbaugh95b]

Operation:	sort (elements: Array of T)
Responsibilities:	organizes the elements of the array (in place) so that they are in increasing order
Inputs:	elements – an array of objects of type T
Returns:	none
Modified objects:	elements
Preconditions:	All of the elements must be of type T or a subtype. Type T has an operation compare(t1:T) returning {LT, EQ, GT} which compares two elements and returns whether the first is less than, equal to, or greater than the second. The operation must define a total order on the values of T. Duplicate values of T are allowed.
Postconditions:	The elements are ordered according to the comparison function. The array contains exactly the same number of occurrences of each value of T as before. If two elements compare as equal, then they have the same relative order before and after the operation.

Figure 138: Operation specification in OMT [Rumbaugh95b]

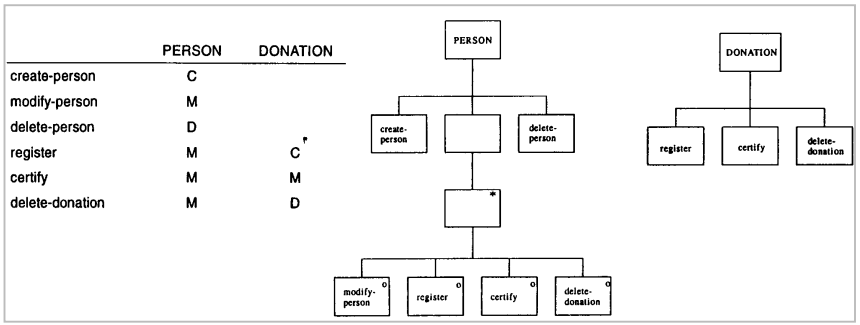
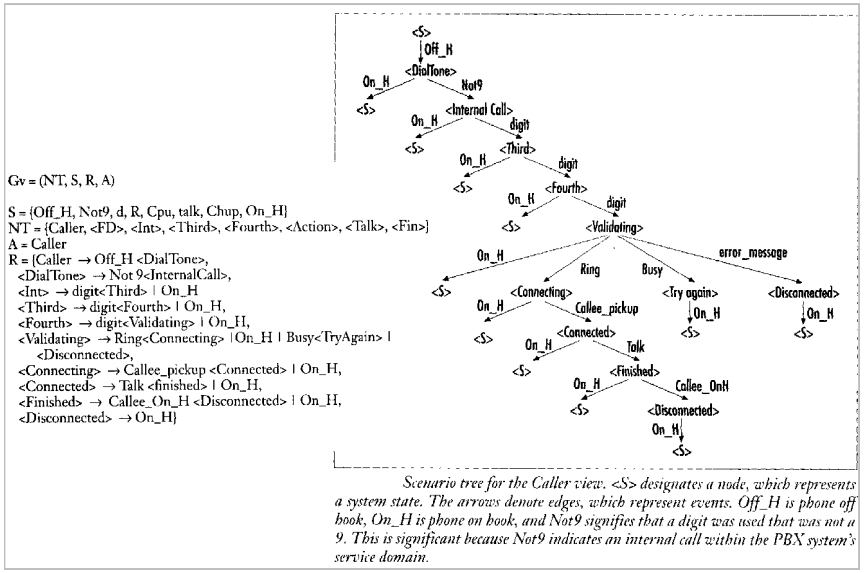
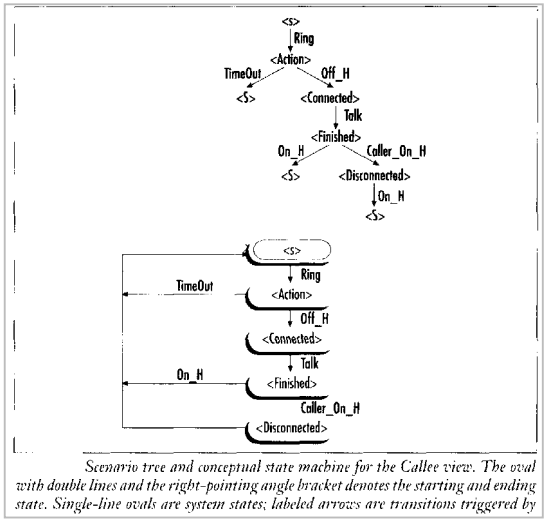


Figure 139: Object-event table and structure diagrams in M.E.R.O.DE [Dedene94]



Scenario tree for the Caller view. <S> designates a node, which represents a system state. The arrows denote edges, which represent events. Off_H is phone off hook, On_H is phone on hook, and Not9 signifies that a digit was used that was not a 9. This is significant because Not9 indicates an internal call within the PBX system's service domain.

Figure 140: Scenario tree and grammar of the user-view of the 'caller' [Hsia94]



Scenario tree and conceptual state machine for the Callee view. The oval with double lines and the right-pointing angle bracket denotes the starting and ending state. Single-line ovals are system states; labeled arrows are transitions triggered by

Figure 141: Scenario tree and conceptual state machine of the user-view of the 'callee' [Hsia94]

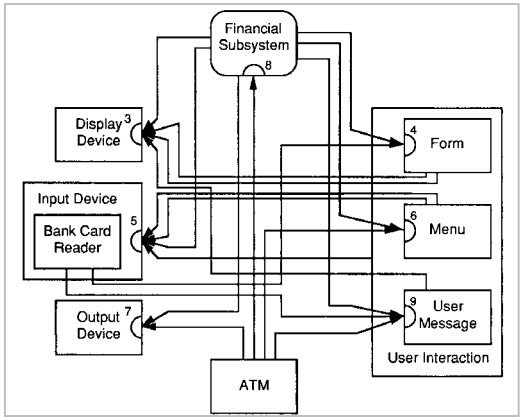


Figure 142: Collaboration graph in Responsibility Driven Design [Wirfs90, page153]

Librarian	
check out Lendable	
check Borrower status	Borrower: can borrow
check it out	
update Borrower	Borrower: know Lend.
check in Lendable	Lendable: check out
check if overdue	Lendable: know overdue
check it in	Lendable: check in
update Borrower	Borrower: know Lend.
record fine	Borrower: know fine.
search for Lendable	Collection: retrieve
display message	UI Subsystem
get info from user	UI Subsystem

Figure 143: Class card in [Wilkinson95, page115]

Scenario 3: Someone Gets a Borrower Object.

Client	Server	Responsibility	Comment
someone	DB	get Borrower	
DB	DB++ classes	get info from DBMS	get info from Borrower table
DB	Borrower	create	
Borrower	List	create	create empty set of lendables
DB	DB	Scenario 2	get lendables for Borrower's set
DB	Borrower	add lendable to set	add each lendable

Figure 144: Scenario description in [Wilkinson95, page146]

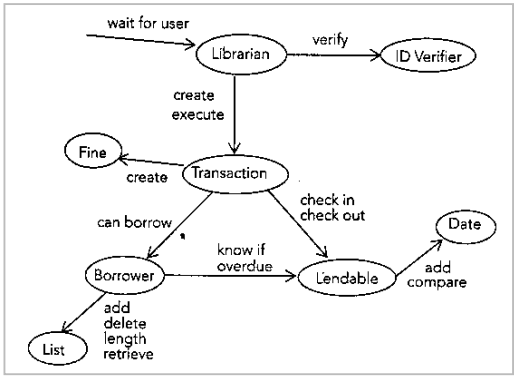


Figure 145: Class diagram with main collaboration in [Wilkinson95, page143]

SCENARIO SPECIFICATION

Scenario 6.3 - Shopper wants information about the merchants who sell a brand-name product.

Input REQUEST-B from shopper.
Search INDEX for BRAND-NAME.
If PRODUCT-TYPE is found,
  Input INDEX entry for PRODUCT-TYPE,
  Output INFO to shopper,
Else Search PRODUCT for similar PRODUCT-TYPEs,
  Output ALTERNATES of likely PRODUCT-TYPEs to shopper,
  Input CHOICE from shopper,
  Input PRODUCT-TYPE entry for PRODUCT indicated by CHOICE.
  Output INFO to shopper.
Input CHOICE of BUSINESS-NAME from shopper.
  Input MERCHANT entry for BUSINESS-NAME.
  Output INFO to shopper.
  If number of MERCHANT entries > 1,
    Output ALTERNATES to shopper.
    If shopper Input CHOICE,
      continue loop,
    Else Wait 30 seconds.
      Output WELCOME to shopper.
End.

Processor Requirements: See Technology section of System Description.
Processing Priorities: Shoppers' requests are high priority.
Performance Requirements: Some type of unique screen display must be made within 3 seconds. This includes messages to shoppers such as "Searching for MERCHANT," "Still searching for MERCHANT," etc.
Capacity Requirements: System must be able to perform all requested activities and provide unique screen displays to a maximum of six shoppers at a time.
Time Relationships: N/A.
Interface Standards: N/A.
Security Requirements: N/A.
Fail-Safe Requirements: N/A.
Safety Requirements: N/A.
Other Requirements: Similar PRODUCT-TYPE, BUSINESS-NAME are names that have reasonable likeness to the input provided by the shopper.

Figure 146: Action diagram with brackets, used for the specification of a scenario [Kowal92, page292]

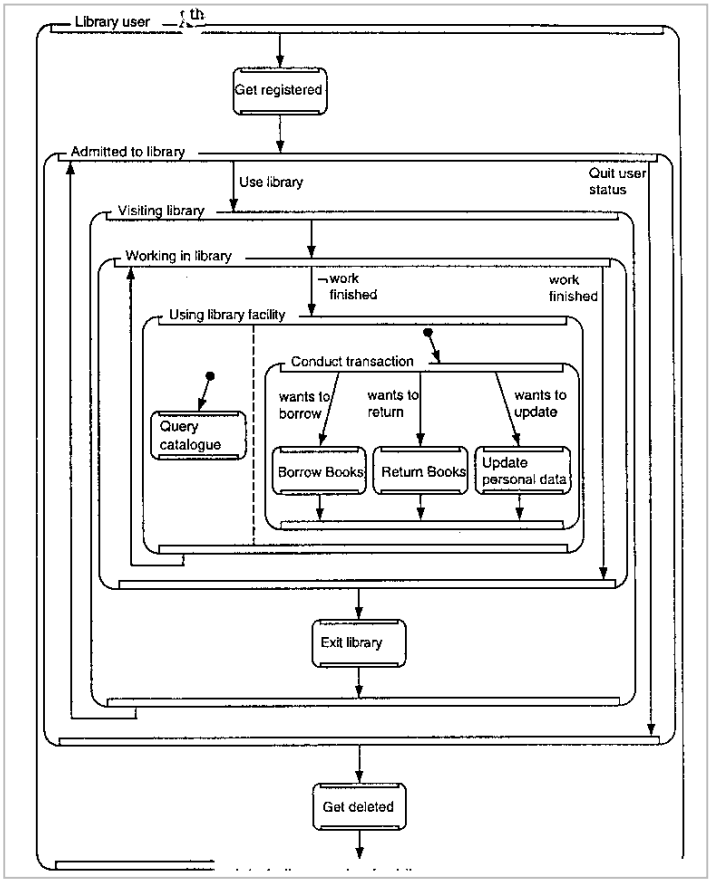


Figure 147: Composition of scenarios in [Glinz95]

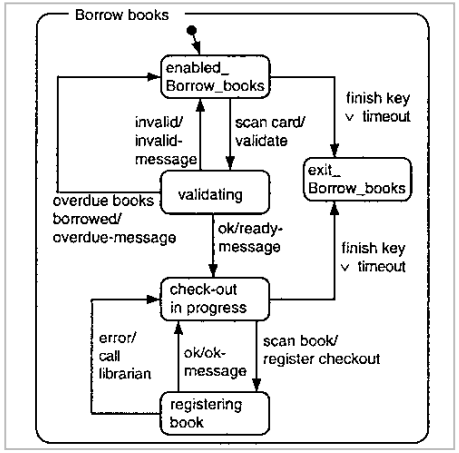


Figure 148: State chart for one scenario in [Glinz95]

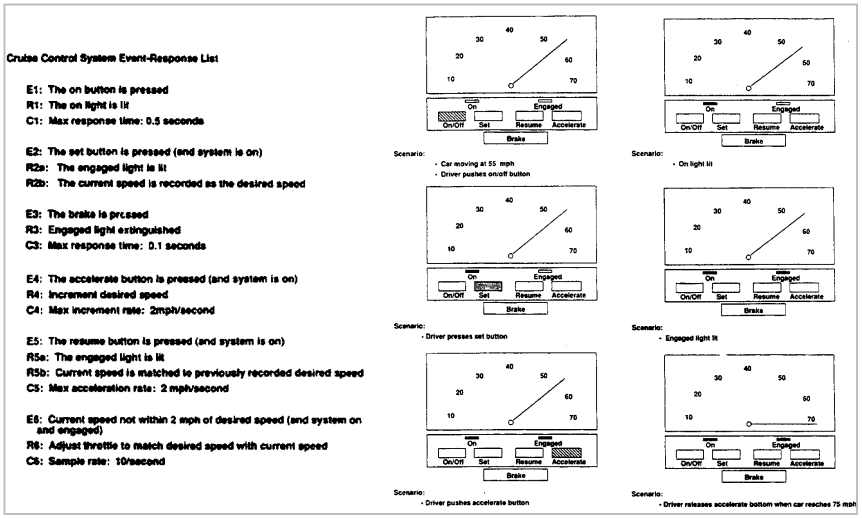
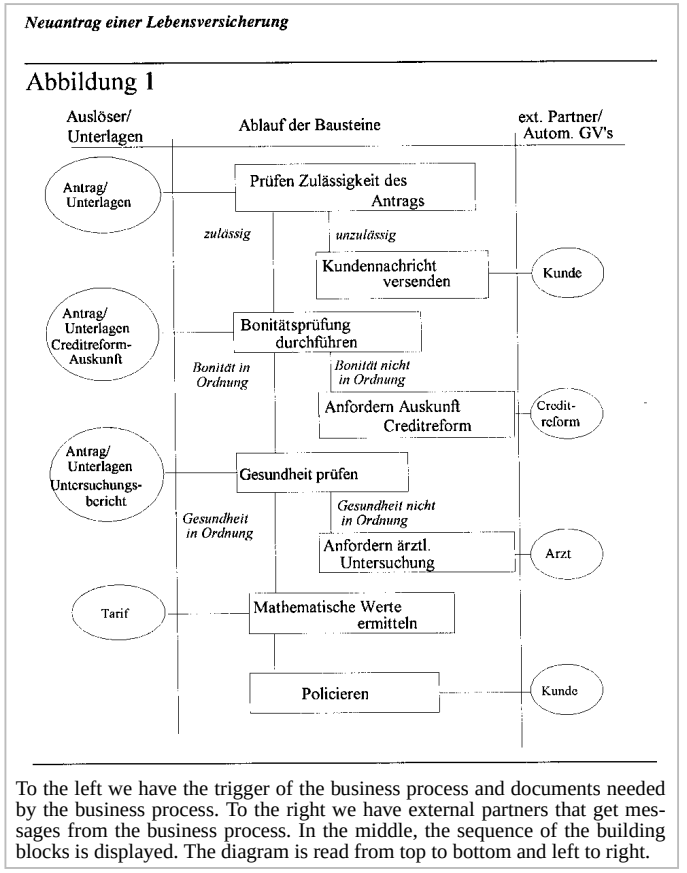
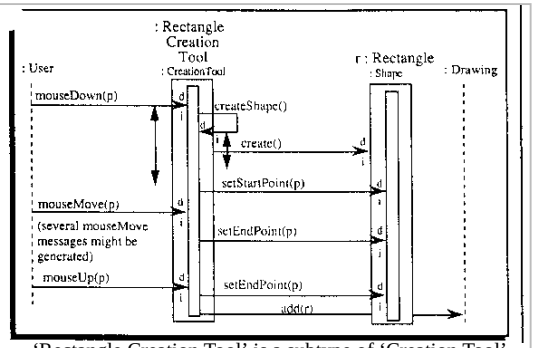


Figure 149: Event-response list and sample storyboard in OORD [Umphress91]



To the left we have the trigger of the business process and documents needed by the business process. To the right we have external partners that get messages from the business process. In the middle, the sequence of the building blocks is displayed. The diagram is read from top to bottom and left to right.

Figure 150: Graphical representation of a “Geschaeftsvorfall” in [Mueller93]



- 'Rectangle Creation Tool' is a subtype of 'Creation Tool'
- the horizontal arrows point to the objects which are referenced in the operation call
- 'd' shows in which type the operation is declared
- 'i' shows in which type the operation is implemented
- the vertical arrows indicate the durations of the operations

Figure 151: Bergen interaction diagram, captures also the inheritance hierarchy of the objects [Baklund95]

Part of the description of the framework Dispatcher/Pane:

```
behaviour:
    create ;
    open ;
    (windowActivating;
       (update*;
        cursorEntersPane;
          (update | userInput)*;
        cursorLeavesPane
       )*;
     windowDeactivating
    )*
    terminate.

 paths:
    create:
"path to create the group"
        model  >>
           pane model:;
"specifies the model"
           [pane name:];
"specifies the pane's name"
           [pane paneMenuSelector:].
"specifies a message that can be sent to the
model to obtain the menu"

    terminate:
"path to terminate the group"
        close.

    open:
       topPane  >>
          pane reframe:;
"determine rectangle for the pane's area"
          pane open.

    close:
       topPane >>
          pane close ;
          model release.
"break connections between the model and the
pane"


update:
        model >>
"update the displayed information"
           refresh
           | (pane update: ; refresh)
           | (pane update:with: ; pane perform:)
           | (pane update:with:with: ;
                  pane perform:with:).
    cursorLeavesPane:
       USER >>
          dispatcher processInput ;
          dispatcher deactivate .
    cursorEntersPane:
       topDispatcher >>
          dispatcher processInput ; dispatcher

activate ;
    windowActivating:
       topPane >>
          pane displayWindow .
    windowDeactivating:
       topPane >>
          pane grayselection .
    userInput:
"handle user input"
        USER >>
          dispatcher processInput ; dispatcher
processKey ;
          (menuRequest | keyInput | ctrlKeyInput |
             functionKeyInput).
    menuRequest:
          dispatcher processFunctionKey ;
          (topPane performMenu
           | (pane performMenu;
                  (topPane performMenu
             | model perform:))).
    functionKeyInput:
          (dispatcher select
          | (dispatcher processKey: ;
              dispatcher processFunctionKey:)).

exception paths:
    none
```

Figure 152: Description of a group of interacting objects using path expressions (PEG) [Adam94]

Appendix C
Enhancing Fusion

In this appendix we describe one possible way to integrate some features of our enhanced scenario modelling technique (SEAM) into the Fusion method.

The analysis model

Hierarchies of services

Fusion, as described by [Coleman94], assumes that the requirements are already determined and specified in some requirements specification document. During analysis, these requirements are modelled in the analysis model and the external view of the global behaviour of the system is specified by system operations. These system operations are very fine grained. They describe the reactions of the system and map them onto exactly one input event. Event trace diagrams (called scenarios) which show sequences of input and output events are used to help determine the system operations, but they are not really part of the analysis model and do not give a more abstract view of the global behaviour.

By introducing higher level system services, hierarchies of services and conceptual interactions, we can enhance Fusion so that it also supports requirements determination and offers the capability to model higher abstraction levels of the external view of global behaviour. The lowest level services - elementary system services in SEAM - correspond to the system operations of Fusion, and the interactions of SEAM correspond to events in Fusion. One of the method-specific characteristics of Fusion is its technique of modelling system operations. Therefore we do not use the syntax and semantics of the elementary system services of SEAM as described in chapter 4, but we keep almost completely to the syntax and semantic of the system operations of Fusion. We propose the following semantic for the system operations in enhanced Fusion:

- Whereas in SEAM elementary system services may have several interactions with the agents, in enhanced Fusion only one input event is allowed. This input event is a notification. In most cases it is a purely conceptual interaction. It models the total input data flow necessary for this system operation from the triggering agent to the system.[1]

- There is a one-to-one correspondence between system operations and triggering events, but in contrast to Fusion they do not necessarily have the same name.
- A system operation is always triggered by an event coming from an external agent. For time-outs, an artificial external agent must be introduced. System operations that are triggered internally, either by another system operation or by consistency triggers, are not allowed.
- System operations may be modelled as specialisations or extensions of other system operations.
- Common parts of system operations cannot be modelled by separate services. A system operation is always a simple service and cannot be modelled as a partial aggregate service. The only way to factor out common parts in the description of system operations is to use functions and predicates which are defined in the data dictionary (see [Coleman94, chapter C-4.2]).

Describing higher level system services

The higher level system services are described by the notation given in chapter 4. All the interactions are notifications. Only the external view of the services is modelled. For the description of high-level services we may use interaction diagrams as well as context diagrams and service diagrams. The scenarios of Fusion are not used.

Whether higher level system services are defined at all, and to which degree they are reflected in the analysis model, depends on the goal of the analysis model. It is also feasible to have two analysis models. The first is developed when the basic functional requirements are determined and before the exact user interface and user interactions are known, and only contains higher level system services. The second is developed after the effective user interface has been decided upon. It contains all the elementary system services, and maybe some higher level system services as well.

Describing elementary system services

Elementary system services are described by the operation schemas of Fusion which are slightly modified and enhanced:

- A section “Receives” is added. This section contains the name and the parameters of the input event (only one), and the name of the agent that sends this event. The parameters are no longer listed in the section “Reads”, and the keyword “supplied” is no longer necessary. The name of the input event is not necessarily identical to the name of the system operation.

1. We allow that the input event may also be a technical interaction. Though this is the default in the Fusion method as described by [Coleman94], it should be used with care. Such a technical interaction must appear as a method call in the design. Examples of the rare situations where this is the case are command line inputs or signals from hardware devices.

- In the case of elementary system services that are specialisations or extended services, a section “Specialisation of” or “Extension of” is added. This section contains the name of the generalized or original service, and a textual description of the conditions under which this extended or specialised service is triggered. In the case of a specialised service, the operation schema contains the whole specification of the specialised service. For an extended service, the operation schema describes what differs from the original service (e.g. instead of the message *a*, the messages *b* and *c* are sent).

In contrast to the high-level services which are described by the interaction diagrams in an algorithmic way, the description of the elementary system services is declarative. No interaction diagrams are made for elementary system services. The interactions are only mentioned in the sections “Receives”, “Sends” and “Results”.

The object model of analysis

We assimilate the analysis object model as it is specified by the Fusion method but we consider it as being only a provisional external data model. It describes the system state as used in the operation schemas, but does not describe the internal structure of the system. The internal structure is determined in the design. The operation schemas of analysis refer to the data items of the object model. Event parameters may also be typed values that are only defined in the data dictionary.

The system life-cycle

In Fusion the system life-cycle is specified by regular expressions showing the possible orders of input and output events. As there does not exist any concept of higher level services, the names of these regular expressions have no further meaning. [Coleman94] recommends enforcing that each input event is followed by an expression containing all possible output events to this input event, and not containing any further input events.[2] This is the case in the life-cycle expressions in figure 113, where *PrivEnquiry* is defined as:

PrivEnquiry = authorize.(#confirm | #deny).(inquire.#amount)*.finish

In enhanced Fusion we simply replace the strings of input and output events belonging to the same system operation by the name of the system operation. Furthermore, the regular expressions specify the complete aggregate services of the system. For above example, we get the following specification of the complete aggregate service *PrivEnquiry*:

2. Another possibility is to mix output events with input events. For above example this would give the following regular expression:
PrivEnquiry = authorize . ((#confirm. (inquire. #amount). finish) | #deny)*
This has the advantage that we can show directly in the life-cycle model that the input event *inquire* can only be accepted after the output event *confirm* and must be rejected after the output event *deny.* The disadvantage of this variant is that the regular expressions get very complex (that is why this variant is not recommended in Fusion [Coleman94, chapter 2-5.2]), and that regular expressions cannot be mapped to system operations or complete aggregate services.

PrivEnquiry = authorize . inquire* . finish

Of course, complete aggregate services may be further described by service schemas and/or interaction diagrams.

The design model

In Fusion messages are always method calls. In enhanced Fusion this is no longer the case. We enlarge the meaning of the term message to include all different kinds of interactions (technical interactions, conceptual interactions, notifications, requests).

Visibility graphs and class descriptions

For enhanced Fusion we change nothing in the visibility graphs, inheritance graphs and class descriptions of Fusion. The class descriptions correspond to the object schemas of SEAM, but we specify neither the services that an object uses nor the life-cycle and essential states of the object. The services that an object uses can only be seen in the interaction graphs. They are recapitulated nowhere else. As in Fusion, the life-cycles of the atomic objects are not specified at all, and only the life-cycle of the system as a whole is modelled (in the analysis model). The methods of Fusion correspond to the atomic services of SEAM.

System operations

In Fusion, each analysis input event involved in a system operation corresponds in the design to a method of the controller object responsible for controlling this system operation. Operation schema data items which are marked as supplied become the parameters of this method. In enhanced Fusion we do not enforce this 1:1 correspondence; on the contrary, analysis input events may even be purely conceptual events, and the corresponding interaction graphs may have several technical input events from the user. If we do not show the interface objects responsible for the user inputs, then we get several interaction graphs, one (or more) for each technical input event. If we show all user interface objects, then one interaction graph is the result (if it is not further decomposed). This one graph shows all user interactions for the whole system operation.

If a system operation is described by several operation schemas, if a generalized system operation has several specialised system operations, or if a system operation has extended system operations, then all these operations are described by the same interaction graph in the design. This interaction graph has to treat all the special cases which may be described by more than one operation schema.

Though we have allowed only one scenario type for the external view of the system operations, we allow several scenario types on several abstraction levels for the internal view of the system operations. This is helpful whenever complex design patterns are used, where we would like to distinguish between the conceptual flow of events and the effective interaction mechanisms.

Changes and enhancements in the syntax of the interaction graphs

Fusion uses two-dimensional object diagrams for the interaction graphs. As we have shown in chapter 4.2.5.5, instead of time-line interaction diagrams two-dimensional object diagrams can also be used for modelling scenario types. We suggest using these diagrams for enhanced Fusion, but we propose some changes and some enhancements compared to the syntax given by Fusion.

- *Messages*: In the interaction graphs we allow notifications as well as requests, and conceptual as well as technical interactions.
- *Create-messages*: Fusion shows both, the creation and the initialisation of objects with a message *create*. This leads easily to confusion, especially when an object is not initialised by the same object as created it. We therefore suggest using the notation proposed in chapter 4.2.5.5. For the creation a special arrow is used which denotes only the creation of the object and is not a method call to the created object.
- *User interface*: We suggest modelling the interface objects for both, the handling of the input events and the handling of the output events. The user interface objects can either be modelled in detail, or summarized by an object group.
- *Decomposing graphs*: The decomposition is not down the hierarchy of method calls but along groups. If a graph should be decomposed, then several objects are put together in a group. The internal views of services offered by this group are modelled by separate interaction graphs. The external view of these services is not modelled or specified separately.
  Groups of objects are also used whenever certain details are not yet known, or when details are deliberately abstracted away (e.g. one group for the whole user interface, one group for each pair of view and controller, or one group for the network communication). Groups are modelled by a dotted rectangle.
- *Description of methods*: Instead of one description for each method, we make one description per interaction graph. This description is labelled with the name of the system or of the group that offers the service that is described by this specific interaction graph, and by the name of the service.
- *Hiding objects*: We also allow object hiding and the replacement of several messages by an indirect message as shown in chapter 4.2.5.4. Yet message hiding is not possible in two-dimensional object diagrams, because the pseudo-code annotation is disconnected from the diagram.

Differences between enhanced Fusion and SEAM

In order to keep the most typical features of Fusion, we have not integrated all concepts of our enhanced scenario modelling technique into Fusion. There remain therefore some substantial differences between enhanced Fusion and SEAM:

- In enhanced Fusion, analysis and design are defined by the viewpoint they adopt towards the system. Analysis models the global behaviour of the software system from an external point of view and uses a provisional data model for the data view. Design determines the objects that make up the software system and shows the internal view of the global behaviour. The external behaviour of a system that is larger than the future software system cannot be modelled, just as it cannot be modelled in Fusion.
- No subsystems[3] are modelled. Analysis is only concerned with the external behaviour of the whole system. In the design, only atomic objects are used. We introduced the notion of object groups to abstract away complex behaviour and to decompose object interaction graphs. But these groups only facilitate the modelling; they have no significance in the final software system.
- The life-cycle is only considered for the system as a whole, not for subsystems or for atomic objects.
- The lowest levels of system services (i.e. the system operations) are modelled differently to the higher level system services. Also various constraints apply to the system operations that we do not have for the elementary system services in SEAM.[4] Enhanced Fusion is more precise than SEAM concerning what is modelled as an elementary service and how it is modelled.
- A system service cannot be modelled as a partial aggregate service.
- We use different kinds of interaction diagrams: For the external view of the high-level system services, time-line diagrams are used. For the internal view of the elementary system services, two-dimensional object diagrams are used.
- We left the class descriptions of Fusion as they are. In contrast to SEAM they do not show which services a class uses from other classes.

Of course, an integration of SEAM into Fusion could go further or less far than is proposed in this appendix. To what degree the Fusion method as described in [Coleman94] is followed and to what degree our SEAM approach is used must be decided according to the specific circumstances of a given project. Factors which must be taken into account are project domain and complexity, tools used, the knowledge and experience of the people involved, and other project specific constraints.

3. Introducing subsystems into Fusion would necessitate to rework concepts and notations for aggregation in the object model of analysis and for exclusive references in the visibility graphs of the design.
4. We could also use the semantic and syntax of the elementary service as described in chapter 4. The system operations would then be described like all other system services, i.e. in an algorithmic way, by one or more scenario types, having zero, one or more input interactions, and being triggered by an external agent or internally.

References

[Adams92] G. Adams; *Describing Groups of Interacting Objects Using Path Expressions*; Master Thesis, School of Computer Science, Carleton University, Ottawa, Ontario, 1992

[Adams94] G. Adams, J.P. Corriveau; *Capturing Object Interactions*; Proceedings of TOOLS Europe '94, Versailles, 1994

[Alvarez95] X. Alvarez et. al; *Use-Cases, Interaction Diagrams, Hypermedia & Visualization*; Workshop on "Requirements Engineering: Use Cases and More", OOPSLA 95, 1995

[Armour95] F. Armour et. al; *Use Case Modeling Concepts for Large Business System Development*; Workshop on "Requirements Engineering: Use Cases and More", OOPSLA 95, 1995

[Atkinson95] C. Atkinson; *A Comparison of Object-Oriented Methods*; Seminar given at the EPFL in Lausanne, May 1995

[Bailer93] B. Bailer; *Geschäftsmodelle, Verständnis und Folgen der Technologieneutralität*; Institutsbericht Nr. 93.43, Institut für Informatik der Universität Zürich, 1993

[Baklund95] E. Baklund et. al; *Extending Interaction Diagrams*; ROAD July-August 1995

[Beck89] K. Beck and W. Cunningham; *A Laboratory for Teaching Object-Oriented Thinking*; Proceedings of OOPSLA 89, 1989

[Benner93] K. M. Benner et. al; *Utilizing Scenarios in the Software Development Process*; Information Systems Development Process, edited by N.Prakash et. al, IFIP Transactions A-30, 1993

[Berard93] E. V. Berard; *Essays on Object-Oriented Software Engineering, Volume1*; Prentice Hall, 1993

[Beringer92] D. Beringer; *Qualitätssicherung*; Seminar of the GfAI Gruppe für angewandte Informatik AG, 1992

[Beringer92b] D. Beringer et al.; *SEUEDA, Phasenkapitel Konzept*; Internal document of the GfAI Gruppe für angewandte Informatik AG, 1992

[Beringer92c] D. Beringer; *Uebungen zur essentiellen Systemanalyse*; Seminar of the GfAI Gruppe für angewandte Informatik AG, 1992

[Beringer93] D. Beringer; *Der Weg zum Objektmodell*; Output, Sonderausgabe über Objektorientierte Systeme, November 1993

[Beringer94] D. Beringer; *Limits of Seamlessness in Object-Oriented Software Development*; Proceedings of TOOLS Europe '94, Versailles, 1994

[Beringer95] D. Beringer; *The Model Architecture Frame: Quality Management in a Multi Method Environment*; Proceedings of SQM'95, Seville, 1995

[Beringer95b] D. Beringer; *Modelling Global Behaviour in Object-Oriented Analysis: Overview of the Usage of Scenarios, Use Cases and Interaction Diagrams*; Project of the postgraduated course on software engineering of 1995, EPFL, Lausanne, Switzerland, 1995

[Beringer96] D. Beringer; *Modelling Global Behaviour in Object-Oriented Analysis: Scenarios, Use Cases and Interaction Diagrams*; Technical Report No. 96/215, Software Engineering Laboratory, EPFL, Lausanne, Switzerland, 1996

[Beringer96b] D. Beringer; *The Goals of the Analysis Model*; Technical Report No. 96/216, Software Engineering Laboratory, EPFL, Lausanne, Switzerland, 1996.

[Berteaud95] R. Berteaud, J. Bezivin; *Requirements Modeling in the OSMOSIS Workbench*; Workshop on "Requirements Engineering: Use Cases and More", OOPSLA 95, 1995

[Blaha93] M. Blaha; *Aggregation of parts of parts of parts*; JOOP September 1993

[Booch91] G. Booch; *Object Oriented Design with Applications*; Benjamin/Cummings, 1991

[Booch94] G. Booch; *Object Oriented Analysis and Design*; Benjamin/Cummings, 1994

[Booch95] G. Booch, J. Rumbaugh; *Unified Method for Object-Oriented Development*; Documentation Set Version 0.8, Rational Software Corporation, 1995

[Bowen95] J. P. Bowen, M. G. Hinchey; *Ten Commandments of Formal Methods*; IEEE Computer, April 1995

[Brantschen91] S. Brantschen; *Essentielle Systemanalyse, Moderne Strukturierte Analyse*; Seminar of the GfAI Gruppe für angewandte Informatik AG, 1991

[Briod93] P. A. Briod, S. Moser, G. Wanner; *BI-CASE/OBJECT V1.1 - A Modern Systems Development Methodology*; Bedag Informatik, Berne, Switzerland, 1993

[Buhr95] R. J. A. Buhr; *Use Case Maps: A New Model to Bridge the Gap Between Requirements and Design*; Workshop on "Requirements Engineering: Use Cases and More", OOPSLA 95, 1995; see also *Use Case Maps for Object Oriented Systems*; Prentice Hall, 1995

[Buhr96] R. J. A.Buhr; *High-Level Design with Use Case Maps*; Tutorial OOPSLA 96, 1996

[Cartiant95] I. Cartiant; *MARCO: Object Architecture Method*; Report on Object Analysis and Design, May-June 1995

[Chang93] C. K. Chang; *Is existing software engineering obsolete?*; IEEE Software, September 1993

[Coad91] P. Coad and E. Yourdon; *Object-Oriented Analysis*; Prentice-Hall, 1991

[Coleman94] D. Coleman et al.; *Object-Oriented Development: the Fusion Method*; Prentice-Hall, 1994

[Coleman95] D. Coleman; *Fusion with Use Cases - Extending Fusion for Requirements Modelling*; Presentation at the Fusion Users Meeting of OOPSLA 95, 1995

[Cook94] S. Cook, J. Daniels; *Designing Object Systems, Object-Oriented Modelling with Syntropy*; Prentice Hall, 1994

[Cotton95] T. Cotton; *Evolutionary Fusion: A Customer-oriented Incremental Life Cycle for Fusion*; Fusion in the Real World, edited by R. Malan, R. Letsinger and D. Coleman, Prentice-Hall, 1995

[Davis90] A. Davis; *Software Requirements - Analysis and Specification*; Prentice Hall, 1990

[Davis93] A. Davis; *Software Requirements - Objects, Functions and States*; Prentice Hall, 1993

[Dedene94] G. Dedene, M. Snoeck; *M.E.R.o.DE: A Model-driven Entity-Relationship object-oriented DEvelopment method*; ACM SIGSOFT, vol 19 no 3, 1994

[DeMarco79] T. DeMarco; *Structured Analysis and System Specification*; Yourdon Press, 1979

[D'Souza93] D. D'Souza; *Comparing OO Analysis and Design Methods*; Proceedings of the OOP 93 in Muenchen

[Dubois93] E. Dubois et al.; *O-O Requirements Analysis: an Agent Perspective*; Proceedings of the ECOOP 93 in Kaiserslautern, 1993

[Duffy95] D. J. Duffy; *Object-Oriented Requirements Analysis*; ROAD, July-August 1995

[Eckert93] G. Eckert, P. Golder; *Improving Object-Oriented Analysis*; Technical Report No. 93/32, Software Engineering Laboratory, EPFL, Switzerland, 1993

[Eckert95] G. Eckert; *Improving the Analysis Stage of the Fusion Method*; Fusion in the Real World, edited by R. Malan, R. Letsinger and D. Coleman, Prentice-Hall, 1995

[Eckert95b] G. Eckert and M. Kempe; *Modeling with Objects and Values: Issues and Perspectives*; ROAD vol 1 no 5, 1995

[Firesmith95] D. G. Firesmith; *Use Cases: The Pros and Cons*; ROAD July-August 1995

[Finkelstein96] A. Finkelstein, I. Sommerville; *Viewpoints in Requirements Engineering*; Special issue of the Software Engineering Journal, vol 11 no 1, January 1996

[Floyd89] C. Floyd; *Softwareentwicklung als Realitaetskonstruktion*; Proceedings of the Fachtagung Software-Entwicklung, of the GI, Marburg, 1989

[Gilb88] T. Gilb; *Principles of Software Engineering Management*; Addison Wesley, 1988

[Glinz95] M. Glinz; *An Integrated Formal Model of Scenarios Based on Statecharts*; Proceedings of the 5th European Software Engineering Conference ESEC '95, Sitges, Spain, 1995

[Graham93] I. Graham; *Financial Object-Oriented Rapid Analysis Method (FORAM), Training Notes 1 to 4*; Seminar given at the SBV in Basel, 1993

[Graham94] I. Graham; *Object-Oriented Methods, 2nd Edition*; Addison-Wesley, 1994

[Graham94b] I. Graham; *Beyond the Use Case: Combining task analysis and scripts in object-oriented requirements capture and business process re-engineering*; Proceedings of TOOLS Europe' 94, Versailles, 1994

[Graham95] I. Graham; *Business Process Modelling*; Object Expert, November-December 1995 and January-February 1996

[Graham96] I. Graham; *Task scripts, use cases and scenarios in object oriented analysis*; Object Oriented Systems, vol 3 no 3, September 1996

[Gryczan92] G. Gryczan, H.Züllighoven; *Objektorientierte Systementwicklung, Leitbild und Entwicklungsdokumente*; Informatik-Spektrum Nr. 15, 1992

[Hansen95] F. Hansen; *Experiences with OMT Analysis: "From the Real World to an Application"*; Workshop on "Requirements Engineering: Use Cases and More", OOPSLA 95, 1995

[Harel87] D. Harel; *Statecharts: a visual formalism for complex systems*; Science of Computer Programming 8, 231-274, 1987

[Heeg94] G. Heeg; *Object Behavior Analysis - Von der Suche nach Objekten*; Talk given at a meeting of Choose and INTEC-OO in Bern, Switzerland, 1994

[Hofmann93] H. F. Hofmann; *Requirements Engineering, A Survey of Methods and Tools*; Institutsbericht Nr. 93.05, Institut für Informatik der Universität Zürich, 1993

[Holbrook90] H. Holbrook; *A Scenario-Based Methodology for Conducting Requirements Elicitation*; ACM SIGSOFT, vol 15 no 1, 1990

[Høydalsvik93] G. H. Høydalsvik, G. Sindre; *On the Purpose of Object-Oriented Analysis*; Proceedings of OOPSLA 93, 1993

[Huber90] G. P. Huber; *A Theory of the Effects of Advanced Information Technologies on Organizational Design, Intelligence, and Decision Making*; Academy of Management Review, vol 15 no 1, 1990

[Hsia94] P. Hsia et al.; *Formal Approach to Scenario Analysis*; IEEE Software, March 1994

[Jacobson92] I. Jacobson; *Object-Oriented Software Engineering*; Addison Wesley, 1992

[Jacobson95] I. Jacobson; *A growing consensus on use cases*; JOOP March-April 1995

[Jacobson95b] I. Jacobson et. al; *Using contracts and use cases to build plugable architectures*; JOOP May 1995

[Jarke94] M. Jarke, K. Pohl; *Requirements engineering in 2001: (virtually) managing a changing reality*; Software Engineering Journal, November 1994

[Jufer92] R. Jufer; *Strukturierte Spezifikation*; Seminar of the GfAI Gruppe für angewandte Informatik AG, 1992

[Kilberth93] K. Kilberth, G. Gryczan, H. Zuelligkoven; *Objektorientierte Anwendungsentwicklung: Konzepte, Strategien, Erfahrungen*; Verlag Vieweg, 1993

[Kilov94] H. Kilov and J. Ross; *Information Modeling, an object-oriented approach*; Prentice Hall, 1994

[Kolbe95] K. Kolbe; *How do you know you are building the "right" software? Experiences with Use Cases*; Workshop on "Requirements Engineering: Use Cases and More", OOPSLA 95, 1995

[Kotonya96] G. Kotonya, I. Sommerville; *Requirements engineering with viewpoints*; Software Engineering Journal, January 1996

[Kowal92] J. A. Kowal; *Behavior Models, Specifying User's Expectations*; Prentice Hall, 1992

[Kruchten95] P. B. Kruchten; *The 4+1 View Model of Architecture*; IEEE Software, November 1995

[Kruchten96] P. B. Kruchten, C. J. Thompson; *Iterative Software Development for Large Ada Programs*; Proceedings of Ada-Europe'96, Montreux, June 1996

[Loenvig95] B. Loenvig; *Experiences with Requirement Capturing, Specification and OO Requirements Analysis*; Workshop on "Requirements Engineering: Use Cases and More", OOPSLA 95, 1995

[Martin85] J. Martin and C. McClure; *Diagramming Techniques for Analysts and Programmers*; Prentice Hall, 1985

[Martin95] J. Martin and J. Odell; *Object-Oriented methods: a Foundation*; Prentice Hall, 1995

[McGinnes92] S. McGinnes; *How Objective is Object-Oriented Analysis?*; Proceedings of CAiSE'92, 1992

[McMenamin84] M. McMenamin, J.F. Palmer; *Essential Systems Analysis*; Yourdan Press, 1984

[Meyers92] S. Meyers and S. P. Reiss; *An Empirical Study of Multiple_View Software Development*; Proceedings of SIGSOFT'92, 1992

[Meyer93] B. Meyer and J. M. Nerson; *Design by Contract*; Proceedings of the Summer School of EDBT in Leysin, 1993

[Moser92] S. Moser; *Metrics and Estimation, Aufwandschätzungen in Informatik-Projekten*; Seminar of the GfAI Gruppe für angewandte Informatik AG and of IBM Switzerland, 1992

[Mössenböck96] H. Mössenböck and K. Koskimies; *Visualisierung objektorientierter Software durch Ereignisdiagramme*; SI-INFORMATIK, no3 ,1996; see also K. Koskimies and H. Mössenböck; *Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs*; Proceedings of ICSE-18, Berlin, 1996

[Mueller93] G. Mueller-Luschnat et al.; *Objektorientierte Analyse und Geschäftsvorfall-modellierung*; Proceedings of the EMISA-Fachtagung über objektorientierte Methoden für Informationssysteme, Universität Klagenfurt, 1993

[Nierstrasz95] O. Nierstrasz and L. Dami; *Object-Oriented Software Technology*; in Object-Oriented Software Composition, edited by O. Nierstrasz and D. Tsichritzis, Prentice Hall, 1995

[Odell93] J. Odell; *Using business rules with diagrams*; JOOP July-August 1993

[ODMG93] *The Object DBMS Standard*; Th. Atwood, Object Magazine, September-October 1993, see also *The ODMG Object Model*; M. Loomis, JOOP June 1993

[OMG92] *Object Analysis and Design, Volume 1: Reference model, Draft 7.0*; OMG, 1992

[Parnas86] D. L. Parnas and P. C. Clements; *A Rational Design Process: How and Why to Fake It*; IEEE Trans. on Soft. Eng., vol 12 no 2, February 1986

[Rawsthorne95] D. A. Rawsthorne; *Transaction Based Analysis*; Workshop on "Requirements Engineering: Use Cases and More", OOPSLA 95, 1995

[Regnell96] B. Regnell et. al; *A Hierarchical Use Case Model with Graphical Representation*; Proceedings of ECBS'96, March 1996

[Reisin90] F. Reisin; *Kooperative Gestaltung in partizipativen Softwareprojekten*; PhD Thesis, Technische Universität Berlin, 1990

[Rubin92] K. S. Rubin, A. Goldberg; *Object Behavior Analysis*; Communications of the ACM, September 1992

[Rumbaugh91] J. Rumbaugh et al.; *Object Oriented Modelling and Design*; Prentice-Hall, 1991

[Rumbaugh93] J. Rumbaugh; *Controlling code - How to implement dynamic models*; JOOP May 1993

[Rumbaugh94] J. Rumbaugh; *Modeling & Design*; JOOP, issues of June 1994, September 1994, November-December 1994, February 1995, March-April 1995 and May 1995

[Rumbaugh94b] J. Rumbaugh; *Getting started, Using use cases to capture requirements*; JOOP September 1994

[Rumbaugh94c] J. Rumbaugh; *Building boxes: Composite objects*; JOOP November-December 1994

[Rumbaugh94d] J. Rumbaugh; *The life of an object model*; JOOP March-April 1994

[Rumbaugh95] J. Rumbaugh; *OMT: The dynamic model*; JOOP February 1995

[Rumbaugh95b] J. Rumbaugh; *OMT: The functional model*; JOOP March-April 1995

[Sadr96] B. Sadr, P. J. Dousette; *An OO Project Management Strategy*; IEEE Computer, vol29 no9, 1996

[Schauer93] H. Schauer, B. Kuhnt; *Partizipative Software-Entwicklung*; Institutsbericht Nr. 93.35, Institut für Informatik der Universität Zürich, 1993

[Sharble93] R. Sharble and S. Cohen; *The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods*; ACM SIGSOFT, April 1993

[Shlaer88] S. Shlaer, S. J. Mellor; *Object-Oriented Systems Analysis, Modeling the World in Data*; Yourdon Press, 1988

[Shlaer92] S. Shlaer, S. J. Mellor; *Object Lifecycles, Modeling the World in States*; Yourdon Press, 1992

[Siddiqi94] J. Siddiqi; *Challenging Universal Truths of Requirements Engineering*; IEEE Software, March 1994

[Stein93] W. Stein; *Objektorientierte Analysemethoden - ein Vergleich*, Informatik-Spektrum (1993) 16:317-332, Springer-Verlag, 1993

[Stein94] W. Stein; *Objektorientierte Analysemethoden: Vergleich, Bewertung, Auswahl*, B.I. Wissenschaftsverlag, Mannheim, 1994

[Umphress91] D. A. Umphress, S. G. March; *Object-oriented requirements determination*; in JOOP Focus on A&D, edited by R. S. Wiener, SIGS Publications, 1991

[Ungar91] D. Ungar, R. B. Smith; *SELF: The Power of Simplicity*; Lisp and Symbolic Computation, vol4 no3, 1991

[Waldén95] K. Waldén, J.M. Nerson; *Seamless Object-Oriented Software Architecture*; Prentice Hall, 1995

[Ward94] R. Ward, J. Stevens; *Object Orientation Is Not Always Best!*; Proceedings of Ada in Europe, Copenhagen, 1994

[Wassermann92] A. Wassermann; *Behaviour and scenarios in object-oriented development*; JOOP February 1992

[Webster] *Webster's Ninth New Collegiate Dictionary and Webster's Collegiate Thesaurus*; NeXT Digital Version, 1988/1992

[Wilkinson95] N. M. Wilkinson; *Using CRC Cards: an informal approach to object-oriented development*; Prentice Hall, 1990

[Wirfs90] R. Wirfs-Brock, B. Wilkerson.; *Designing Object-Oriented Software*; Prentice Hall, 1990

[Yourdon89] E. Yourdon; *Modern Structured Analysis*; Prentice Hall, 1989

[Zorman95] L. A. Zorman; *Requirements Envisaging by Utilizing Scenarios (REBUS)*; PhD Thesis, USC/ISI USA, 1995

Curriculum Vitae

Name:	Isolde Dorothea Beringer - Bärtschi
Date of birth:	May 15, 1964
Nationality:	Swiss

1970 -1978	Primary and secondary school, Untergymnasium
1978 -1982	Gymnasium Bern-Neufeld
1983 -1989	Student of computer science, mathematics and micro-electronics at the Universities of Bern and Neuchatel
1987	Trainee at SAS Institute, Cary NC, USA
1988 -1989	Diploma student at the ABB Research Centre in Dättwil
1989	Diploma in computer science (lic. phil. nat.) from the faculty of Science at the University of Bern
1990 -1991	Software engineer at Ascom Gfeller AG in Bern
1991 -1994	Methodologist at GfAI, Gruppe für Angewandte Informatik AG in Herrenschwanden
1994 -1997	Research assistant and PhD student at the Software Engineering Laboratory of the Swiss Federal Institute of Technology in Lausanne (EPFL)