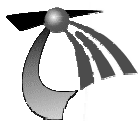


# Architectural Design of Groupware Systems: a Component-Based Approach

Cléver Ricardo Guareis de Farias



CTIT Ph.D.-thesis series, no. 01-38

CTIT, P.O. Box 217, 7500 AE Enschede, the Netherlands

ISBN 90-365-1680-3

Copyright © 2002 by C. R. Guareis de Farias, Enschede, The Netherlands.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the written permission of the author. No part of this publication may be adapted in whole or in part without the prior written permission of the author.

**ARCHITECTURAL DESIGN OF GROUPWARE SYSTEMS:  
A COMPONENT-BASED APPROACH**

DISSERTATION

to obtain  
the doctor's degree at the University of Twente,  
under the authority of the rector magnificus,  
prof.dr. F.A. van Vught,  
on account of the decision of the graduation committee,  
to be publicly defended  
on Thursday, May 30, 2002 at 15:00 hrs.

by  
Cléver Ricardo Guareis de Farias  
born on June 14, 1974  
in Santo Inácio, Brazil

This dissertation has been approved by:

promotor                      prof . dr. ir. C.A. Vissers

assistant-promotor        dr. ir. Marten van Sinderen

To my beloved wife Kelen,  
for all her love, care and understanding  
during this important period of my life.



---

# Preface

The technological advances of the last decade have brought many changes into our society. Among the developments that determined these changes, three are crucial for understanding this ongoing revolution: 1) the dissemination of personal computers at offices and at our homes; (2) the advent of the World Wide Web (WWW) and its rapid acceptance by the society; (3) the increasingly use of portable computers and devices.

The increasing dissemination of personal computers, not only at offices but also at our homes, brought us another perspective in the way we work. Tasks that were usually manually done, such as typing a text or filling a form, now can be automated.

The Internet in general and the WWW in particular also play an important role in this context by bringing people together in new ways. The exchange of information through the Internet creates new possibilities in all areas of our everyday lives. Considering only our working environment, the interaction between the different branches of a company spread all over a country or the world that previously had to be done by phone, fax or mail can now be done by electronic mails and documents, for example. People can now effectively work together in the same project no matter their physical location, either at home or across the corridor, either at another building or across the world.

The advent of portable computers and devices and their interconnection with the Internet through either wired or wireless technologies has caused even a greater impact in our lives, collaborating to create the so-called ubiquitous computing. On one hand, information can be gathered anywhere and instantly transmitted through the Internet. On the other hand, the same information can be available to anyone through a device with access to the Internet.

Computers have become essential working and entertainment tools of our lives. Yet, most of the computer systems are targeted to single users, although most of our working tasks is likely to involve groups of people. Computer systems that provide support for groups of people engaged in a common task are called groupware.

The development of groupware systems poses many different challenges. Apart from the social aspects of groupware, their developers are faced with problems typical of both distributed systems and cooperative work: problems pertaining to distributed systems are, amongst others, the need for adequate levels of transparency, reliability, security and heterogeneity support; problems related to cooperative work are mainly related to the need for tailorability in groupware systems.

We believe that the use of component-based technologies can greatly contribute to solve these problems. Component-based development aims at constructing software artefacts by assembling prefabricated, configurable and independently evolving building blocks called components. Components are deployed on top of distributed platforms, which contributes to solve many of the distribution-related problems of groupware systems. Components can also be configured, replaced and combined on-the-fly, which enhances the degree of tailorability pro-

vided by a system. In fact, most of the recent approaches towards the provision of groupware tailorability rely on the use of components.

This work focuses on how to systematically design component-based systems to support cooperative work. We provide a methodology based on the standard modelling notation Unified Modeling Language (UML) to design component-based groupware. This manuscript is structured as follows.

Chapter 1 further elaborates our motivation and objectives of this work and describes the approach used to accomplish the objectives of the work.

Chapter 2 presents some state-of-the-art information that is relevant to understand this work. This chapter defines some basic terminology and provides some background information related to component technologies, groupware systems and component-based groupware systems.

Chapter 3 introduces our design methodology, presents its scope and compares it with some related work.

Chapter 4, 5 and 6 detail our design methodology using UML as our design notation. Each of these chapters concentrates on a major milestone of the methodology.

Chapter 7 describes the extensions to UML that we provide in order to design groupware systems according to our methodology.

Chapter 8 presents a case study involving the design of an electronic meeting system to illustrate the main aspects of our design methodology.

Chapter 9 concludes this manuscript by drawing some conclusions and outlining some points for further research.

---

# Acknowledgements

*“And if you call for insight and cry aloud for understanding, and if you look for it as for silver and search for it as for hidden treasure, then you will understand the fear of the Lord and find the knowledge of God. For the Lord gives wisdom, and from his mouth come knowledge and understanding.” Proverbs 2:3-6*

My quest for knowledge brought me to the Netherlands four and a half years ago. However, this quest started years before with a dream, a dream given by the Lord to shape my life for years to come. During this period of time I have been blessed tremendously. So many blessings that I dare not count. From small to big things, the Lord has provided me with everything needed to fulfil my dream. So, I thank the Lord for his mercy and grace!

The Lord gave me Kelen, a companionship for this difficult period of time and beyond, a period that was particularly difficult for her. I thank her for postponing her own dreams to share with me good and bad moments. I thank her for staying at my side, giving me her love, care and understanding when most needed. Certainly, everything would have been more difficult if I were alone, if she were not there pushing me forward, encouraging me and cherishing me. This accomplishment is not only mine, but also yours. Thank you my love!

My special thanks go to Marten, Luís and Chris for the opportunity, guidance and friendship they gave me. I believe there were times when it was not easy to cope with my stubbornness. I learned a lot from you, and I hope you have learned something from me!

Besides the discussions I had with Marten and Luís, I also would like to thank for the fruitful discussions I had with the other members of the Architecture group, especially Giancarlo, Nikolay, Dick, Ciro and João Paulo. Your contributions were very important for my work!

I also appreciate the support provided by the other members of the TSS group. My gratitude goes particularly to the current and former secretaries of TSS, CTIT and TIOS. I will not mention names, so I will not forget anyone. However, be sure that your work made my work easier!

Many thanks to the other members of my Graduation Committee, namely Prof. dr. H. Wallinga, Prof. dr. ir. M. Aksit, Prof. dr. R.J. Wieringa, Prof. dr. ir. L.J.M. Nieuwenhuis, Prof. dr. W.L. de Souza and dr. ir. H. ter Hofte, for the time they invested in reading my work and gathering for the promotion ceremony.

I also would like to thank the financial support provided by CNPq, under the process number (200003/97-6 (NV)), as well as the support provided by the Faculty of Computer Science.

Perhaps the most difficult thing in doing a PhD is that most of the time you are working alone. However, when I was not working I was glad I had the support of so many good friends, particularly Ciro, Marcos, Ana and Pedro, João and Patrícia, Giancarlo and Renata,

Natasha Williams, Vlora, Glória, Alex, Luís, Marten, Dick, Nikolay, Helen, Diptish, Remco, Maarten and many others. Thank you guys!

I cannot forget to thank my good friend Samuel (Samuca) for helping me design the cover of this book!

Finally, I also would like to thank my family, relatives and friends in Brazil for their support and their prayers! I thank the Lord for you all. May God bless you!

Cléver Ricardo Guareis de Farias

Ribeirão Preto, April 26, 2002.

---

# Agradecimentos

*“E se clamares por entendimento, e por inteligência alçares a tua voz, se como a prata a buscares e como a tesouros escondidos a procurares, então entenderás o temor do Senhor, e acharás o conhecimento de Deus. Porque o Senhor dá a sabedoria: da sua boca vem o conhecimento e o entendimento.” Provérbios 2:3-6*

Minha busca por conhecimento me trouxe à Holanda há quatro anos e meio. Contudo, esta busca se iniciou anos antes através de um sonho, um sonho dado por Deus para me orientar na minha caminhada. Durante este período fui abençoado grandemente. Foram tantas bênçãos que não ousou contá-las. Desde pequenas até grandes cousas, o Senhor me deu tudo para que este sonho se tornasse realidade. Por tudo isso, agradeço ao Senhor por sua misericórdia e graça!

Por companhia neste período de dificuldades e incertezas, o Senhor me deu Kelen, período este que foi particularmente difícil para ela. Eu a agradeço por postergar seus próprios sonhos para compartilhar junto a mim momentos bons e ruins. Eu a agradeço por estar ao meu lado dando seu amor, carinho e compreensão quando mais precisei. Certamente, tudo teria sido muito mais difícil se eu estivesse sozinho, se ela não estivesse lá estimulando-me, encorajando-me e alegrando-me. Esta realização não é somente minha, mas também sua. Obrigado meu amor!

Eu também gostaria de agradecer de forma especial ao Marten, Luís e Chris pela oportunidade, orientação e amizade dadas a mim. Acredito que por diversas vezes não deve ter sido fácil agrentar a minha teimosia. Aprendi muito com vocês, e espero que tenham aprendido algo comigo!

Além das discussões que tive com o Marten e o Luís, também gostaria de agradecer as discussões proveitosas que tive com os outros membros do grupo de Arquitetura, especialmente Giancarlo, Nikolay, Dick, Ciro e João Paulo. Vocês contribuíram bastante para o meu trabalho!

Também agradeço o suporte dos demais membros do TSS. Minha gratidão especial vai para as secretárias e ex-secretárias do TSS, CTIT e TIOS. Não vou mencionar nomes, de forma a não me esquecer de ninguém. Contudo, estejam certas de que o trabalho de vocês facilitou o meu!

Muito obrigado aos membros da minha banca, Prof. dr. H. Wallinga, Prof. dr. ir. M. Aksit, Prof. dr. R.J. Wieringa, Prof. dr. ir. L.J.M. Nieuwenhuis, Prof. dr. W.L. de Souza e dr. ir. H. ter Hofte, pelo tempo investido na leitura do meu trabalho e por participarem da minha defesa.

Também gostaria de agradecer o apoio financeiro provido pelo CNPq, sob o processo nro (200003/97-6 (NV)), como também o apoio provido pela Faculdade de Informática.

Talvez a coisa mais difícil de se fazer um doutorado, seja trabalhar sozinho a maior parte do tempo. Contudo, quando não estava trabalhando, eu estava feliz por receber o apoio de tantos bons amigos, especialmente Ciro, Marcos, Ana e Pedro, João e Patrícia, Giancarlo e Renata, Natasha Williams, Vlora, Glória, Alex, Luís, Marten, Dick, Nikolay, Helen, Diptish, Remco, Maarten e muitos outros. Muito obrigado pessoal!

Ah, não posso me esquecer de agradecer ao meu amigão Samuel (Samuca) por me ajudar com o design da capa deste livro!

Finalmente, também gostaria de agradecer aos meus pais, irmão, cunhados, avós, sogros, tios, primos e amigos pelo apoio deles e por suas orações! Agradeço especialmente aos meus pais, Eva e Farias, e ao meu irmão, Kérlinton Vinícius, que tanto me incentivaram na realização deste sonho, sonho este que não era somente meu, mas também deles. Agradeço a Deus por todos vocês. Que Deus vos abençoe abundantemente!

Cléver Ricardo Guareis de Farias  
Ribeirão Preto, 26 de abril de 2002.

---

# Table of Contents

<b>Preface</b> .....	<b>i</b>
<b>Acknowledgements</b> .....	<b>iii</b>
<b>Agradecimentos</b> .....	<b>v</b>
<b>Table of Contents</b> .....	<b>vii</b>
<b>Chapter 1 Introduction</b> .....	<b>1</b>
1.1 Background.....	1
1.2 Motivation.....	2
1.2.1 Computer support for cooperative work .....	2
1.2.2 Functionality reuse and run-time support.....	3
1.2.3 Groupware tailorability .....	5
1.2.4 Component-based groupware.....	6
1.3 Design methodology .....	7
1.3.1 Elements of a methodology.....	7
1.3.2 Quality properties.....	7
1.3.3 The Unified Modeling Language .....	8
1.4 Objective .....	9
1.5 Approach.....	10
1.6 Structure of the thesis.....	10
<b>Chapter 2 Component-Based Groupware</b> .....	<b>11</b>
2.1 Component technology .....	11
2.1.1 Software component definition .....	11
2.1.2 Components versus objects.....	13
2.1.3 Discussion on software components .....	14
2.1.4 Component Object Model (COM) .....	15
2.1.5 Enterprise JavaBeans Component Model (EJB) .....	17
2.1.6 OMG CORBA standards.....	18
2.1.7 Component platforms.....	22
2.2 Groupware technology.....	23
2.2.1 Groupware-related definitions.....	24
2.2.2 Groupware systems .....	25
2.2.3 Groupware classification.....	33
2.3 Component-based groupware .....	36
2.3.1 Live .....	37
2.3.2 EVOLVE.....	38

---

2.3.3	DISCIPLE.....	39
2.3.4	CoCoWare.....	40
2.3.5	DACIA.....	41
2.3.6	DreamTeam.....	42
2.4	Conclusion .....	43
<b>Chapter 3 Component-Based Design .....</b>		<b>45</b>
3.1	Basic concepts.....	45
3.1.1	Abstraction.....	45
3.1.2	Abstraction levels.....	46
3.1.3	View.....	48
3.1.4	Concern levels.....	49
3.2	Software development process .....	50
3.3	Methodology overview .....	52
3.3.1	Methodology concern levels .....	52
3.3.2	Methodology views.....	54
3.4	Methodology scope.....	55
3.5	Related work .....	56
3.5.1	Unified Process .....	56
3.5.2	Catalysis.....	58
3.5.3	Pattern of Four Deliverables .....	59
<b>Chapter 4 Enterprise Level Modelling.....</b>		<b>61</b>
4.1	Enterprise systems and concepts.....	61
4.1.1	Enterprise systems.....	61
4.1.2	Enterprise concepts .....	63
4.2	Cooperative work models .....	65
4.2.1	Elaboration of cooperative work models.....	65
4.2.2	Coordination theory .....	66
4.2.3	Activity Theory .....	67
4.2.4	Action/Interaction Theory.....	68
4.2.5	Task Manager.....	70
4.2.6	Object-Oriented Activity Support .....	71
4.3	A cooperative work metamodel .....	72
4.3.1	Informal view of cooperative work.....	72
4.3.2	Analysis criteria .....	73
4.3.3	Analysis of cooperative models .....	73
4.3.4	An enterprise metamodel for cooperative work .....	76
4.3.5	Relationships between activity units .....	77
4.3.6	Coordination issues .....	78
4.3.7	Compliance to proposed criteria .....	79
4.4	Enterprise perspectives and abstraction levels.....	79
4.4.1	Enterprise perspectives.....	80
4.4.2	Refinement guidelines.....	84

---

4.5	Enterprise specification.....	87
4.5.1	Design trajectory .....	88
4.5.2	Modelling strategy .....	88
4.5.3	Concept diagram .....	90
4.5.4	Glossary of terms .....	96
4.5.5	Activity diagram.....	97
4.6	Specification example: travel cost claim process .....	102
4.6.1	Integrated perspective specification .....	102
4.6.2	Distributed perspective specification .....	103
4.6.3	Distributed perspective with role discrimination specification .....	105
4.7	Conclusion .....	107
<b>Chapter 5 System Level Modelling.....</b>		<b>109</b>
5.1	Enterprise to system transition.....	109
5.2	Architectural concepts .....	110
5.2.1	Functional entities .....	111
5.2.2	Functional behaviour.....	112
5.2.3	Interaction specialisation.....	112
5.2.4	Interaction point .....	113
5.2.5	Interaction patterns.....	114
5.3	System modelling.....	115
5.3.1	Behaviour perspectives .....	115
5.3.2	Design trajectory .....	117
5.3.3	Discussion on the design trajectory.....	118
5.4	Service specification .....	121
5.4.1	Modelling strategy .....	122
5.4.2	Use case diagram.....	125
5.4.3	Package diagram .....	128
5.4.4	System interaction diagram.....	129
5.4.5	System interface diagram.....	131
5.4.6	Interface operation specification .....	133
5.4.7	Statechart diagram.....	135
5.4.8	Activity diagram.....	139
5.4.9	Glossary of terms .....	142
5.5	Internal behaviour specification.....	142
5.5.1	Modelling strategy .....	142
5.5.2	Use case diagram.....	143
5.5.3	Glossary of terms .....	144
5.5.4	Activity diagram.....	144
5.6	Conclusion .....	145
<b>Chapter 6 Component Level Modelling.....</b>		<b>147</b>
6.1	System refinement .....	147
6.1.1	From system to components.....	147
6.1.2	Component classification schemes.....	149

---

6.1.3	Component decomposition approaches.....	150
6.1.4	Component categories.....	151
6.2	Collaboration concerns .....	152
6.2.1	Collaboration concern layers.....	152
6.2.2	Related work .....	155
6.3	Collaboration coupling and distribution issues.....	156
6.3.1	Coupling levels .....	156
6.3.2	Distribution issues of collaboration coupling.....	158
6.4	Component design trajectory .....	159
6.5	Component modelling.....	162
6.5.1	Modelling strategy .....	162
6.5.2	Techniques and road map.....	163
6.5.3	Component specification techniques.....	166
6.6	Refinement assessment .....	170
6.6.1	Refinement assessment in UML .....	171
6.6.2	Statechart diagram refinement assessment.....	172
6.6.3	Interaction diagram refinement assessment .....	174
6.6.4	Activity diagram refinement assessment.....	174
6.7	Remaining issues .....	177
6.7.1	System refinement guidelines .....	177
6.7.2	Component reuse.....	178
6.8	Conclusion .....	182
<b>Chapter 7 Component-Based Groupware Profile.....</b>		<b>185</b>
7.1	UML profiles .....	185
7.2	Profile specification .....	187
7.2.1	Design trajectory extensions .....	187
7.2.2	Structural extensions .....	190
7.2.3	Behavioural extensions .....	191
7.3	Summary of profile.....	195
<b>Chapter 8 Electronic Meeting System: a Case Study .....</b>		<b>197</b>
8.1	Collaboration context.....	197
8.1.1	Usage scenario .....	197
8.1.2	Collaboration features .....	198
8.2	Enterprise level specification.....	199
8.2.1	Integrated perspective specification .....	200
8.2.2	Decomposed perspective specification .....	202
8.2.3	Decomposed perspective with role discrimination specification .....	204
8.3	System level specification.....	204
8.3.1	Required service specification .....	205
8.3.2	Internal behaviour specification.....	210
8.4	Component level specification.....	211

---

---

8.4.1	System decomposition .....	211
8.4.2	Session management component specification .....	212
8.4.3	Session management component decomposition .....	217
8.5	Conclusion .....	220
<b>Chapter 9</b>	<b>Conclusion .....</b>	<b>221</b>
9.1	General considerations.....	221
9.2	Main contributions.....	222
9.2.1	Concern levels, perspectives and views revisited.....	222
9.2.2	Component types and collaboration concern layers revisited .....	223
9.2.3	UML shortcomings revisited.....	225
9.3	Directions for further research .....	226
<b>References</b> .....	<b>References .....</b>	<b>229</b>
<b>Index</b> .....	<b>Index.....</b>	<b>243</b>
<b>Summary</b> .....	<b>Summary.....</b>	<b>247</b>



---

# Chapter 1

## Introduction

This chapter presents background information and motivates our work, bringing it in line with the developments in Computer Supported Cooperative Work (CSCW). It also explains our main research objective and the approach used to achieve this objective.

This chapter is organised as follows: section 1.1 presents briefly the history of CSCW and introduces some general problems pertaining to this area; section 1.2 motivates the use of component technology in the development of groupware systems; section 1.3 introduces the concept of design methodology; section 1.4 presents the scope and main objective of our work; section 1.5 explains the approach used to accomplish our objective; finally, section 1.6 outlines the organisation of the rest of this thesis.

### 1.1 Background

Computer Supported Cooperative Work (CSCW) emerged as a research area in the mid 1980's [BaSc89]. CSCW focuses on suitable forms of cooperation between persons or groups of persons to perform a common task, and investigates the design, implementation and realisation of computer support for these forms of cooperation. Products that provide computer support for groups of persons engaged in a common task, as well as an interface to a shared environment, are called *groupware* [ElGR91].

In the early days of CSCW, much of the research was focused on the development of examples of what could be done to support cooperative work with computers, the so-called *point systems* [OCL+93]. At this stage, many different systems were developed, tested and evaluated, revealing the promises and difficulties lying ahead of this field.

In the late 1980's, computer supported collaboration was mainly restricted to universities and research centres. In the early 1990's, the first groupware systems became commercially available [SiJF99]. At the same time, the Internet was gaining momentum, and in the mid 1990's, the popularisation of the World Wide Web (WWW) fundamentally helped the dissemination of CSCW and groupware systems among computer users in general. Since then, people were able to actively work together in common projects, no matter their physical location, either within the same room or across the corridor, either in different buildings or across the world.

Nowadays, the advent of portable computers and devices, and their interconnection with the Internet through either wired or wireless technologies, are causing a great impact on our lives, contributing to realise the so-called "Ubiquitous Computing" or "Internet Everywhere" phenomenon. Personal Digital Assistants (PDAs), intelligent mobile phones (SmartPhones) and other handheld devices allied with new wireless technologies, such as Bluetooth [Blue99] and the Wireless Application Protocol (WAP) [WAP99], are promising to revolutionise the computing landscape.

The CSCW community is increasingly interested in the investigation of mobility in cooperation [BeBI96, BDL+99, KrLj98, LuHe98, PaSY00, LiPr00]. This new form of cooperation emphasizes some problems already common to CSCW, such as the lack of activity awareness, adequate communication and activity coordination support.

Much of the effort faced by groupware developers, especially in synchronous groupware, is related to the provision of *awareness* [Fuch99]. Group collaboration relies not only on explicit communication among the members of the group, but also on implicit availability of information of each other's presence and actions. In this way, awareness can be defined as the ability of the application to expose the activities of the people engaged in a common task.

Other problems that make groupware development a difficult task are *consistency management* and *latecoming* [Hoft98]. Possible inconsistencies may emerge whenever simultaneous user activities trigger concurrent actions on a shared workspace. Several approaches can be used to maintain consistency or to manage inconsistencies when they occur. Sometimes the emergence of inconsistencies is intentional, e.g., whenever multiple users work simultaneously to produce different versions of the same document.

In many conference-like applications, there is also a need to provide the current state of the conference to an attendee that arrives after the conference has started. Latecoming can be defined as the ability of the application to provide to a new conference user the description or the current status of a shared workspace.

To cope with these and other problems involved in groupware development, two basic approaches are normally considered: *ad hoc development* and use of *groupware platforms*. On one hand, in the ad hoc development, a groupware system is usually built from scratch, making little or no use of elaborate building blocks. On the other hand, groupware platforms, also called toolkits, provide a number of abstractions and facilities that allow one to build a groupware system based on a set of pre-defined building blocks that can be reused and combined in different ways. Examples of groupware platforms include Groupkit [RoGr92, RoGr96], COAST [SKS+96], COLA [TrRB93, TrRM94], Worlds [FiTK95], Prospero [Dour96] and TACTS [Teeg96].

Groupware development based on groupware platforms is becoming increasingly popular because it not only facilitates the development process itself, but also is more cost-effective if compared to ad-hoc development. As such, we believe it should be supported by a proper design methodology.

## 1.2 Motivation

### 1.2.1 Computer support for cooperative work

In order to build appropriate computer support for cooperative work we need to understand the context in which cooperative work takes place. A cooperative work context is characterised by the tasks involved, the people involved in the execution of these tasks, how one task affects another, etc.

Computer support should reflect the needs of people as they collaborate. We should be able to provide technological infrastructure in order to facilitate the execution of cooperative tasks, while maintaining the benefits of social interaction among the people involved. Computer

support for the execution of some tasks may be unnecessary and sometimes even undesirable, mainly when it inhibits social interactions.

Figure 1-1 illustrates the relationship between cooperative work and computer support. On one hand, cooperative work needs pose a set of requirements on the technological support that should be provided. On the other hand, the technological capabilities constrain the type of functionality that can be supported. As a result, groupware systems are developed as a compromise between what should be done and what can be done in order to adequately support cooperative work.

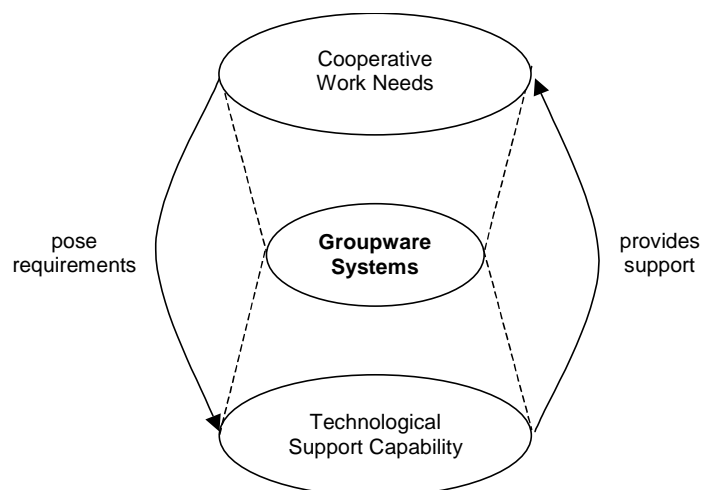


Figure 1-1: Cooperative work versus technological support

### 1.2.2 Functionality reuse and run-time support

The reuse of functionality and the provision of run-time support have been some of the main concerns associated with the development and deployment of groupware systems. In the early days of groupware development, groupware systems were developed by hand, usually extending the functionality of an originally developed single-user application for the needs of multiple distributed users [Dour96]. Eventually, multi-user aspects of groupware systems began to migrate into reusable blocks of code or libraries. These blocks of code and a suitable execution environment form the basis of groupware platforms.

A groupware platform, also known as groupware toolkit, usually supports the development of specific types of groupware systems. The generality of a groupware platform is determined by the abstractions and reuse support provided by the platform according to an application domain. In the sequel, we provide an overview of three of the most representative groupware platforms developed in the early and mid 1990's in order to illustrate some of the main characteristics of a groupware platform.

#### GroupKit

GroupKit [RoGr92, RoGr96] is a platform used in the development of real-time groupware systems. GroupKit uses a (semi-)replicated architecture approach containing three main components or processes, viz., a *registrar*, *session managers* and *conference applications*.

The registrar is the only centralised process in GroupKit that acts as a connection point for a community of conferences. A session manager is a replicated process that provides support

for the creation, deletion, joining or leaving of conferences. This process provides not only a user interface, but also a policy on how to manage conferences. A conference application is also a replicated process that provides functionality typically associated with single groupware applications, such as a shared whiteboard or a chat system. One or more conference applications can be invoked and managed by the user using the session manager, possibly at the same time.

GroupKit also contains components that provide support for event announcement and notification, access to shared elements, group awareness, etc.

## COAST

COAST [SKS+96] is an object-based platform used in the development of real-time groupware applications. COAST uses a replicated architecture approach, in which each local application operates on a copy of a shared document.

A shared document may consist of several parts. The access of an application user to a shared document is coordinated via a *session object*. The application users interact with the shared document via *view objects* and *controller objects*. A view object is used to visualise the document in a window, while a controller object is used to process the user input from the window. Both view objects and controller objects access the document using the associated session object.

A *session manager* allows users to create, join or leave sessions, while user objects are used to represent the concurrent users of the document. A *transaction manager* is used to ensure the integrity of shared objects, such as session, user and controller objects. A *replication manager* is used to synchronise the replicated objects.

## COLA

COLA [TrRB93,TrRM94] is another object-based platform used in the development of synchronous groupware systems. COLA is based on a distributed architecture, in which a lightweight activity model is used to provide mechanisms for describing cooperative contexts or situations based on object sharing. The main abstractions within this lightweight model are activities, roles, shared objects and events.

A lightweight *activity* defines a process in which users and objects interact and exhibit a public state. A number of users participate in an activity, which has a lifecycle subdivided into stages. Within COLA, activities are defined using an Activity Definition Language (ADL).

An activity is also associated with one or more *roles*. A role simply provides a common name to group one or more users in order to provide access control on shared objects.

A *shared object* encapsulates some information, which can be accessed and shared by multiple activities. The functionality associated with an object can be accessed via a set of operations. The operations provided by an object can be presented to other objects and activities in many different ways, based on who is accessing the object, what operations they wish to perform and when they access the object. COLA shared objects are defined using an Interface Definition Language (IDL).

The decoupling of a shared object from client objects and activities is achieved by means of an *object adapter*. An object adapter is an intermediary object between the object being pre-

sented and any clients of the object. An object adapter can be associated with one or more objects. A number of components are used to support object adapters within COLA, including a trader, an object factory, an adapter manager and a binder. COLA adapters are defined using an Adapter Definition Language (ADDL).

*Events* are used to propagate interesting information between activities and objects. Events can be propagated to roles, activities, users and objects. COLA events are described using an Event Description Language (EDL).

### 1.2.3 Groupware tailorability

A groupware system is usually developed to support a specific cooperative work context. In case there is a change in this context, we should be able to assess whether or not the existing computer support still provides the necessary support according to the new context. In case the available computer support no longer corresponds to the current needs, we should be able to adapt this support to reflect those changes.

It is unlikely that a system targeted to a group of people will satisfy everyone's preferences and needs equally. Some contingencies typical of cooperative work, such as a group's individual preferences, different modes of collaboration, the opportunistic nature of work and different contexts of use, make this situation even worse. The ability to adapt a groupware system within its context of use is generally regarded as *tailorability*.

Figure 1-2 illustrates the adaptation of a groupware system in order to cope with the changes in a cooperative work context. Given an original cooperative work context, a certain groupware system provides the necessary support for people engaged in a common task. However, as this context changes, the system users may realise that the system does not provide the necessary support anymore. In this case, the groupware system has to be adapted to cope with the new cooperative work context.

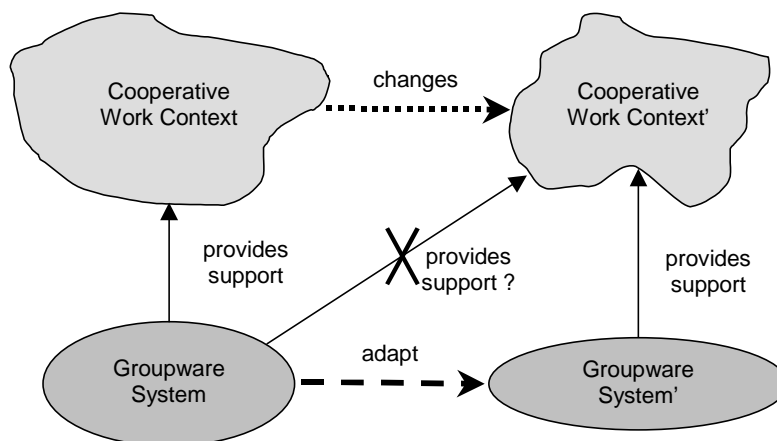


Figure 1-2: Adaptation of cooperative work support

Nowadays, tailorability is widely recognised as a central requirement in the design of groupware systems [Gree91, Hoft98, HuMe00, StHC99], especially with the constantly evolving work requirements that we face today.

*Tailoring* is defined as the activity of modifying a computer system within its context of use [KMS+00]. Tailoring takes place after the original design and implementation of the system; it can start during or right after the installation of the system.

Tailoring is usually carried out by individual users, local developers, helpdesk staff or groups of users. The level of tailorability varies a lot, from simple user interface tailoring to more sophisticated forms, such as the provision of a complete new set of functionality, which is commonly regarded as radical tailoring [MaLF95]. In the literature [Morc97], different levels of tailorability are defined:

- *customisation*, which deals with the selection of configuration options among a predefined set. Customisation is similar to what Greenberg [Gree91] defines as personalization. Examples of customisation in groupware include the selection of different floor control policies among the users of a shared whiteboard, or the selection of individual interface preferences;
- *integration*, which deals with the composition of a set of predefined components into one unified system. The composition of the components of a groupware platform is an example of integration;
- *extension*, which deals with the addition of functionality to a system and its components by adding new program code. Examples of extension include the implementation of a new floor control policy in a shared whiteboard or the addition of a new media in a multimedia conferencing system. Scripting languages are often used for both the integration of components and the extension of existing code without the need for recompilation.

#### 1.2.4 Component-based groupware

Component-based groupware development has gained increasing support in the past few years. This research area aims at constructing groupware systems by assembling prefabricated, configurable and independently evolving building blocks, the so-called *software components* (see Chapter 2, section 2.1.1 for a precise definition of software component). A component provides some functionality that can be used separately or in composition with the functionality provided by other components.

Components are blocks of code ready to be deployed on top of a suitable execution environment. Nowadays, such an execution environment, also known as *container*, provides a number of generic services for the execution of components, such as event notification, authentication, transaction and lifecycle management services. Additionally, containers and their support infrastructure provide the necessary transparency for the development of distributed systems.

It has also been argued that two basic properties that a groupware system must have in order to provide tailorability are extensibility and composability [Hoft98]. Extensibility corresponds to the ability to add new functions without interfering with existing ones, while composability corresponds to the ability to compose a function by selecting and combining more basic functions. These properties are closely related and usually associated with the use of components.

The use of components for the provision of groupware tailorability has received a lot of attention, c.f., [StHC99, HuMe00, Syri97, Teeg00]. Stiemerling and Cremers [StCr98] point out three different forms in which a groupware system can be tailored using components, viz.: changing the parameters of single components, changing the composition of components and

changing or extending the implementation of components. These forms of component-based tailorability roughly correspond to the levels defined in section 1.2.3, namely customisation, integration and extension, respectively.

Therefore, the development of component-based groupware systems not only benefits from functionality reuse but also benefits from deployment and tailorability facilities associated with components.

### 1.3 Design methodology

This section introduces the concept of *design methodology*. The section presents the elements of a design methodology, some quality properties and motivates the use of a standard design notation in this work.

#### 1.3.1 Elements of a methodology

A design methodology is defined as a collection of design methods based on design concepts and supported by design notations and tools [Quar98]. A design methodology is effective if it is capable of supporting the production of sound designs in a target application domain.

Design methods consist of systematic procedures, guidelines or techniques used to produce models of the elements in the application domain. A model is an artefact used to represent aspects of a problem under investigation, allowing one to abstract from unnecessary details and to provide a better understanding of the problem as a whole. A set of related models forms a specification.

Design concepts are basic constructs or elements used to represent characteristics of an element in the application domain. Design notations are textual or graphical representations of the design concepts and their possible compositions. A design notation may have an underlying abstract syntax and formal semantics in order to enforce the proper use of these concepts and to enable the composition of independent models to form a specification.

Tools provide (semi)-automated support for design, analysis (simulation, verification and validation), and code generation activities.

Figure 1-3 illustrates the elements of a design methodology and their relationships. Each box represents a different element, while an arrow connecting two boxes represents a relationship between these elements.

#### 1.3.2 Quality properties

There are a number of general *quality properties* that are desirable in a design methodology:

- *simplicity*: a methodology should use a minimal set of concepts, thus facilitating its learning and use as a whole;
- *systematicness*: a methodology should provide a stepwise process to guide the development of models, in which details are added systematically according to the objectives or intended use of the models;

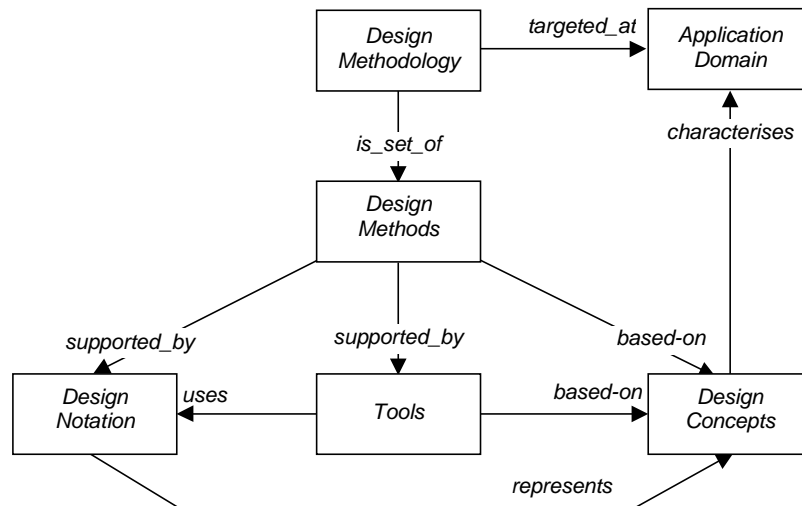


Figure 1-3: Elements of a design methodology

- *prescriptiveness*: a methodology should prescribe what should be done to produce models, rather than prescribe what may be done;
- *flexibility*: a methodology should be used in a variety of situations, without (major) changes or adaptations;
- *scalability*: a methodology should be applicable without modifications to the design of small and large systems.

### 1.3.3 The Unified Modeling Language

In the late 80's and early 90's the object-oriented community witnessed the appearance of a large number of design languages (see [Wier98] for a survey on these languages). Many of these languages were to some extent similar, but none of them appeared to be complete enough. By the mid 90's, these languages evolved and began to incorporate each other's techniques. In this way, some few prominent design languages appeared.

The Unified Modeling Language (UML) [OMG01] is a standard design notation for the specification, visualisation, construction and documentation of software systems artefacts. UML has been standardised by the UML Partners Consortium, led by Rational Software Corporation, under the auspices of the Object Management Group (OMG).

The development of UML started in 1994 when G. Booch and J. Rumbaugh joined forces to merge the Booch Method [Booc94] and Object Modelling Technique (OMT) [RBP+91]. One year later I. Jacobson joined the unification efforts, by merging the Object-Oriented Software Engineering (OOSE) [Jaco95] method with the other two.

In this work, we adopt UML as our specification language. The main reasons for adopting UML as the basis for our design methodology are the following:

- UML is a standard modelling language (OMG standard), which has facilitated its acceptance in industrial and academic settings;
- UML incorporates several useful techniques, which was one of the primary goals of the development of UML, i.e., to unify the different methodologies and to integrate the best practices present in industry;

- UML has an increasing number of supporting tools (see [FaDP99] for a report on some of the available UML supporting tools), which is a basic requirement for the dissemination and correct employment of any modelling language;
- UML is independent from any particular design methodology, which allows UML to be used by different design methodologies targeting specific problem domains;
- UML provides a number of built-in extensibility and specialisation mechanisms to extend its core concepts.

## 1.4 Objective

The design of groupware systems is a complex task, involving knowledge of both distributed systems and social sciences. In this work, we are not concerned with the problems pertaining to social sciences; but we rather focus on the technical aspects of developing groupware.

Our initial hypothesis is that the design of groupware systems could greatly benefit from a design methodology based on the use of component technology. This design methodology can help groupware designers to carry out the design process in a systematic way, such that they can focus on each different type of functionality that should be provided by the system separately from each other.

Groupware designers should have means to choose the best technological support for a certain cooperative work context and to assess whether or not the chosen set of components provide the desired support. In case there is a change in the cooperative work context and we have to adapt the support system, we should be able to analyse how we can better cope with these changes, e.g., by adding, removing or replacing components, and to assess whether or not the resulting composition still provides the required support.

To the best of our knowledge, there are not many works that concentrate on the design of component-based groupware systems. One of the main contributions in this area, proposed by H. ter Hofte [Hoft98], provides general guidelines and architectures for component-based design. However, this work does not provide a step-by-step methodology that relies on standard design notations, such as UML.

General-purpose UML-based methodologies are not generally suitable for the design of component-based groupware systems for a number of reasons. First, general-purpose methodologies tend to be complex in order to cope with a variety of situations and application domains. Second, most of existing UML-based methodologies are primarily targeted towards the design of object-oriented systems. Last but not least, existing methodologies that rely on UML may also suffer from the lack of support for component-based design associated with this notation. In this sense, the main objective of this work can be generally defined as follows:

*Provide a methodology for the design of component-based groupware systems.*

We believe that the guidelines and foundations proposed in this thesis could be used in combination with other works, like, e.g., Ter Hofte's work [Hoft98]. However, we do not intend to provide guidelines to incorporate tailorability in the design of groupware systems within the scope of this research.

UML is a design notation originally developed for the specification of object-oriented systems. Component support was added to the first UML standards as an afterthought. Therefore, the support provided by UML is likely to fall short of expectations and pose some drawbacks. A secondary objective of this work is to investigate the support provided by UML for the design of component-based systems, identifying some shortcomings and proposing adequate remedies as needed.

## 1.5 Approach

In order to accomplish our objectives, we adopt the following approach:

- investigate the state-of-the-art in groupware systems and component technology. This investigation provides a basis for understanding component technology and shows the diversity of existing groupware systems and component-based groupware platforms;
- introduce concern levels to structure the development of the application and design steps relating these levels. The concern levels help structuring and developing a groupware system systematically. Moreover, design steps and methods provide guidelines to structure the system in terms of a composition of software components, possibly reusing existing components;
- propose a conceptual model that can be used to create an abstract model of a cooperative work process. Having an abstract model of a cooperative work process, we are able to evaluate the needs for computer support and develop this support accordingly;
- identify requirements for component-based design and propose extensions to UML in order to cope with these requirements accordingly;
- demonstrate and evaluate the methodology by means of a comprehensive case study.

## 1.6 Structure of the thesis

The remaining of this thesis is structured as follows:

- Chapter 2 presents some state-of-the-art information that is relevant to position this work. This chapter defines some basic terminology and provides background information related to component technology, groupware systems and component-based groupware systems;
- Chapter 3 introduces our design methodology, presents its scope within the context of software development in general and provides an overview of some related work;
- Chapter 4, 5 and 6 discuss the different concern levels of our design methodology. Each of these chapters concentrates on a major milestone of the methodology;
- Chapter 7 describes extensions to UML that are necessary to design groupware systems according to our methodology;
- Chapter 8 presents a case study on the design of an electronic meeting system containing chat and voting functionality. This case study aims at illustrating the main aspects of our methodology;
- Chapter 9 concludes this manuscript by presenting the main conclusions and contributions of our work and identifying some issues for further research.

---

# Chapter 2

## Component-Based Groupware

This chapter presents an overview of two technologies, viz., component and groupware technologies. These technologies were brought together in recent years bringing new dynamics to groupware development. Component-based groupware development is a fast-growing discipline in CSCW. This chapter first presents separate overviews of these technologies and then points to the combined developments related to these technologies.

The structure of the chapter is organised as follows: section 2.1 focuses on component technology, viz., component definition, models and platforms; section 2.2 concentrates on groupware technology, describing some groupware applications in the context of groupware classification schemes; section 2.3 presents some component-based groupware architectures and platforms; section 2.4 wraps up this chapter presenting some conclusions.

### 2.1 Component technology

Component-based development has emerged in the recent years as a new paradigm for building software applications. The benefits of this technology include, a.o., a reduced time to market and development costs, and increased degree of interoperability, portability and maintainability.

This section provides an overview of several aspects of component technology. First, the section provides some component-related definitions, followed by a discussion on other definitions of components frequently found in the literature. In the sequel, the section introduces the major component models available nowadays as well as some component platforms that support these models.

#### 2.1.1 Software component definition

We adopt the following definition of *software component* in this work:

*“A component is a binary piece of software, self-contained, customisable and composable, with well-defined interfaces and dependencies.”*

Software components, or simply components, do not exist in isolation; they have some purpose. The purpose of a component is to provide some service that can be used by its environment. The environment of a component may consist of other components, applications, supporting platforms and so on.

The service provided by a component to its environment is accessible via one or more interfaces. An interface specification can be seen as a contract, which is established between a component that provides (implements) the specified services and the environment that uses (invokes) of these services.

A component may depend on the services provided by its environment to properly provide its own service. Therefore, the description of the component dependencies is as important as the description of the component interfaces. The component dependencies comprises not only the interfaces that are required by this component but also the events that can be either produced or consumed by this component. Further, the kind of deployment environment the component has been prepared for, i.e., a suitable execution platform, should also be described.

To be self-contained, a component needs to come with all the necessary resources to properly operate. These resources comprise any kind of data that is required by the component, such as image, audio, video, text, etc., provided that such data do not change over time and cannot be obtained from its environment.

A component being self-contained is also related to its binary representation, which allows the component to be directly executed or interpreted in a run-time environment. Together, binary representation and self-containment allow a component to be independently deployed in a variety of environments.

A component may have some properties that describe the characteristics of the component or expected (quality of) service. It should be possible to customise these properties to fit, for example, changes in the component environment at both deployment time and runtime.

Finally, in saying that a component should be composable, we emphasise that a component should not be a complete application, but instead a component should be a part in the whole. In other words, a component should provide some functionality that combined with the functionality provided by other components form a complete application. By no means, this implies a restriction to the size of a component. We may have a large component that can be considered a complete application in itself, but that will be combined with other (large) components to form a much larger application. Further, a component should be general enough to be reused in a number of applications. There will exist some specific components, which will hardly be reused. Nonetheless, a component should be created with reuse in mind.

Figure 2-1 depicts a component interacting with its environment.

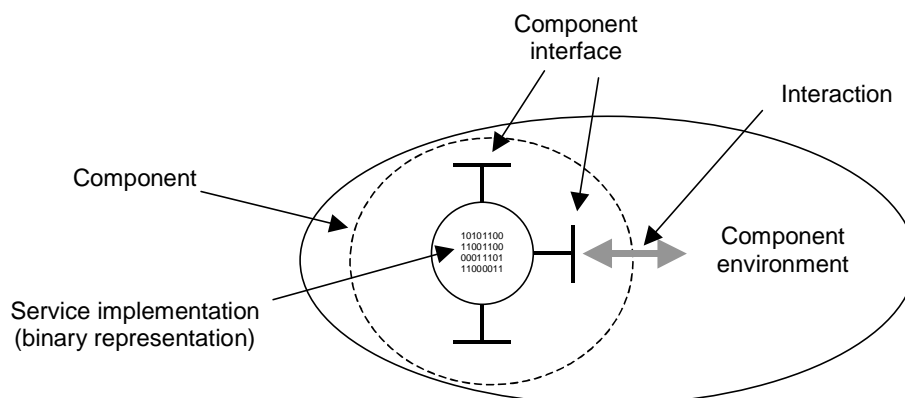


Figure 2-1: Component and its environment

We can classify a component in terms of the composition of its observable behaviour in two basic categories, viz., *simple* and *composite components*. If the behaviour of a component results from a single set of binary code, we call this component a simple component. However, if the behaviour of a component results from an aggregation of multiple components, such

that the combined external behaviour of this component corresponds to a unique component, we call this component a composite component.

Composite components are also known as compound components in the literature. Throughout this work, we simply use the term component to refer to a simple component and composite component otherwise.

Figure 2-2 illustrates the aggregation of simple components to form a composite component. Arrows connecting a component to the interface of other component represent component composition.

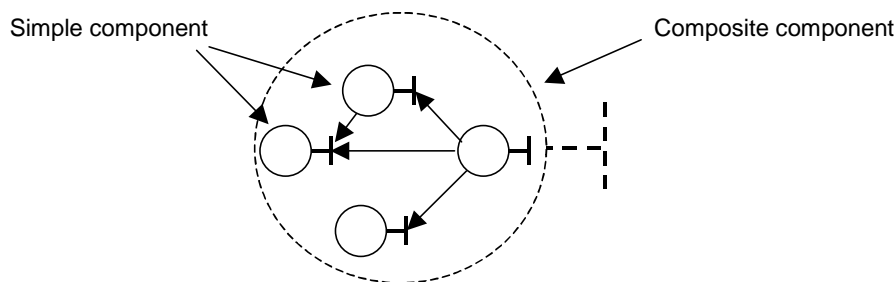


Figure 2-2: Composite component

A *component model* provides guidelines to create, implement and deploy components that can work together to form an application. It defines the basic architecture of a component, specifying the structure of its interfaces and the mechanisms by which its instances interact with their environment, including other component instances and the outside world.

### 2.1.2 Components versus objects

There is a consensus in the literature that components are not the same as objects. However, there is little agreement on the differences between them.

The internal behaviour of a component is usually described by a set of associated classes. When a component is deployed in an execution environment, such as an EJB container, the behaviour of the component will be provided by a set of associated objects that correspond to the classes that form a component. Actually, the behaviour of a component does not have to be necessarily realised using an object-oriented technology.

To deploy a component is to create an instance of the component (or activate a component) and to plug it to its execution environment. A component as a unit of deployment is not a collection of objects, but rather a collection of classes and interfaces.

We cannot compare components and objects directly because we are dealing with things that belong to different metalevels: components as units of deployment and objects as units of instantiation. Therefore, we should compare objects to component instances. A component instance is related to an object in the same way that a component is related to a class.

We should also distinguish between objects and distributed objects. Object-oriented technologies have been around for more than 20 years, usually associated with concepts, such as encapsulation, inheritance and polymorphism. However, in order to build distributed systems, the use of objects imposed more challenges than benefits. For example, objects can only call other objects within the same address space. Therefore, in order to solve this and other prob-

lems, distributed object technology emerged around 10 years ago. Components are usually implemented using distributed objects or objects with distributed-like capabilities. Nevertheless, these objects are not distributed, i.e., they run on a single processor.

A component is typically coarse-grained, while an object is typically fine-grained. Dozens or hundreds of objects are typically used to provide the behaviour of a component.

Table 2-1 summarises the differences between object, distributed object and component instance.

Object	Distributed Object	Component
unit of instantiation	unit of instantiation	unit of deployment
instantiation usually depends on the instantiation of other objects	instantiation usually depends on the instantiation of other objects	instantiation independes from other components (all required information is packaged together)
can only call other objects within the same address space	can call other objects across address space boundaries	can call other components across address space boundaries
runs inside a program	runs on top of an object execution environment (ORB)	runs inside a component execution environment (container)
fine-grained	fine-grained	coarse-grained

*Table 2-1: Comparison between object, distributed object and component*

### 2.1.3 Discussion on software components

There are many different definitions of software components in the literature. Based on our own definition, we can classify these definitions in three groups: equivalent, underspecified and overspecified definitions.

An equivalent definition has no substantial differences if compared to our definition. This notion of equivalency has nothing to do with the definition itself, but with the interpretation behind this definition.

D'Souza and Wills propose a definition for a component very similar to ours. According to them, a component is a package of software implementation that can be independently developed and delivered, has explicit and well-specified interfaces for the services it provides and the services it expects from other components, and can be composed with other components [DSW98].

Szyperski proposes another equivalent definition. According to him, a component is a unit of composition with contractually specified interfaces and explicit context dependencies, which can be deployed independently and is subject to composition by third parties [Szyp98].

On one hand, an underspecified definition of a component is usually too broad or vague if compared to our definition. Further, a definition can also be underspecified if it does not consider some of the important features of a component as discussed in the previous section. On

the other hand, an overspecified definition of a component adds too many features to a component that the development and usefulness of such component is jeopardised.

Booch et al. define a component as a physical and replaceable part of a system that conforms to and provides the realisation of a set of interfaces [BoRJ98]. This underspecified definition is too broad and allows the existence of different kinds of components other than software components, such as executables, libraries, tables, files and documents.

In a follow up definition, a component is defined as a physical piece of implementation of a system, including software code (source, binary or executable) or equivalents, such as scripts or command files, that is replaceable and provides the realisation of a set of interfaces [OMG01]. However, this definition does not contemplate some important features of a component, such as the required set of interfaces and the events produced and consumed by the component. So, this definition can also be considered underspecified.

Buschmann et al. defines a component as an encapsulated part of a software system that acts as a building block for the structure of the system and provides some interfaces. The internal structure of a component may be represented as modules, classes, objects or a set of related functions [BMR+96]. This definition can also be considered underspecified because it is too broad and fails to consider some basic features of our component definition.

Lewandowski proposes yet another component definition that can be considered underspecified since it is too vague. According to him, a component is defined as the smallest self-managing, independent, and useful part of a system that works in multiple environments [Lew98].

In another underspecified definition, Orfali et al. define a (minimalist) component as a self-contained, shrink-wrapped and binary part of an application that has a well-specified interface. Further, a component can be used in unpredictable combinations and it can be considered as an interoperable and extended object, supporting encapsulation, inheritance and polymorphism [OHE96]. The biggest problem faced by this definition is the lack of separation between the definitions of a component and an object.

Orfali et al. build on its definition of a minimalist component, adding support for security, licensing, versioning, life-cycle management, visual assembly, event notification, configuration and proper management, scripting, metadata and introspection, transaction control and locking, persistence, relationships, ease of use, self-testing, semantic messaging and self-installation [OHE96]. This definition of a (super) component is overspecified, given its complexity and lack of objectiveness.

#### **2.1.4 Component Object Model (COM)**

The Component Object Model (COM) is a language-independent, binary component standard introduced by Microsoft in 1993 [WiKi94]. COM core concepts include:

- a binary standard for function calling between components;
- the typed grouping of functions into interfaces;
- a base interface providing mechanisms for (1) other components to dynamically discover the interfaces implemented by a component and (2) a reference counter, allowing components to track their own lifetime and delete themselves when appropriate;

- a globally unique identifier mechanism (128-bit number) for components and their interfaces;
- a component loader to set up and manage component interactions.

COM also provides mechanisms for shared memory management between components and error and status reporting.

In COM, an interface is represented as a pointer to an interface node. The interface node contains a pointer to a table of operation variables. These variables point to the actual implementation of the operations. Interfaces are described in a language-independent way using the Microsoft Interface Definition Language (MIDL).

Figure 2-3 depicts a COM interface representation. A client holds a pointer to the interface node of another component. The interface node points to the virtual function table (v-table) of this component, which points to the memory slots where these functions are implemented.

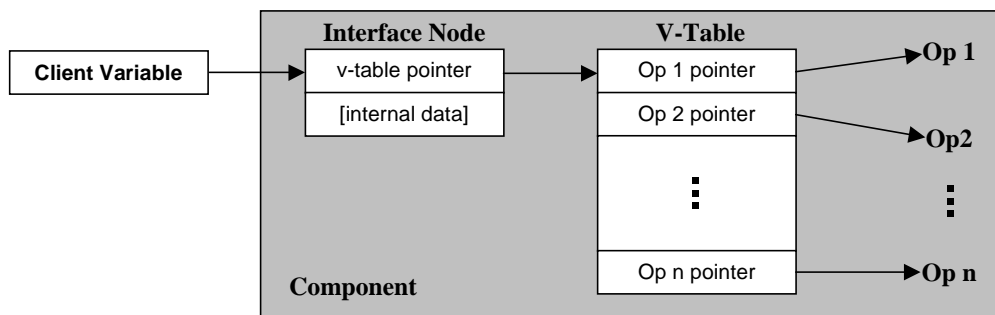


Figure 2-3: COM binary interface representation

COM was originally not a true component model. On its original version introduced in 1993 components were restricted to a single system. Nevertheless, COM is the foundation upon which other Microsoft component models are built.

In mid-1995 Microsoft introduced its distributed version of COM, called Distributed COM (DCOM) [MiCo96]. The introduction of DCOM allowed COM components to be physically separated from the clients that used them, making it possible to create networked applications built from components. At that time, COM already had some capabilities for interprocess communication across a network, which were further enhanced.

In 1996 Microsoft introduced the Microsoft Transaction Service (MTS) [MiCo98], one of the first attempts to combine transactions with components. MTS-based applications are built on top of DCOM/COM components and further introduce the following capabilities: transaction services, configurable security, resource pooling and easy component administration.

In 1998 Microsoft made its next move in the component technology market introducing COM+. This component technology coherently packages COM, DCOM and MTS into one product, enhancing existing services and further introducing new ones, such as asynchronous/queued components, loosely coupled event, component load balancing and in-memory database [Ses00].

Figure 2-4 shows the evolution from COM to COM+.

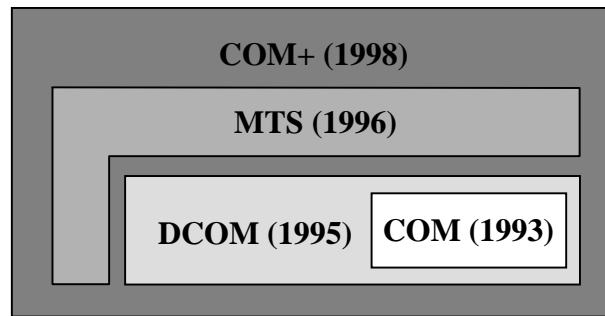


Figure 2-4: COM evolution

### 2.1.5 Enterprise JavaBeans Component Model (EJB)

The Enterprise JavaBeans Component Model (EJB) [SuMi00, Tho98] was introduced by Sun Microsystems in 1998 as an extension of its client-side component model JavaBeans. EJB is a server-side component model for the development of applications in the programming language Java.

According to the EJB, a component is called an enterprise bean. There are two kinds of enterprise beans, viz., a session enterprise bean and an entity enterprise bean. A session bean is a transient component that exists only during a single client/server session, while an entity bean is a persistent component that controls permanent data that is kept in permanent data stores, such as databases.

An enterprise bean resides inside a container. A container consists of a deployment environment for enterprise beans. Further, the container provides a number of services for each bean, such as lifecycle management, state management, security, transaction management and persistence management. In this way, the enterprise bean developer can focus on the application logic that should be provided. An EJB server provides a runtime environment for one or more containers. The server manages low-level system resources and allocates them to containers as needed.

The client application interacts with the enterprise bean by using two interfaces that are generated by the container, viz., the Home interface and the Object interface. When the client invokes an operation using these interfaces, the container intercepts each call and inserts management services.

The EJB Home interface provides access to the enterprise bean lifecycle services. Clients can use this interface to create, destroy or find an existing bean instance. The EJB Object interface provides access to the application methods of the enterprise bean. This interface represents the client's view of the bean and it exposes all the application-related interfaces for the client application, except those used by the EJB container to control and manage this object.

To access to an enterprise bean, one must define a remote interface using the Java Remote Method Invocation (RMI) Application Programming Interface (API). These definitions are used to generate the Object interface. If the client of the enterprise bean is not a Java-based application, the remote interface should be defined using Java IDL (CORBA integration).

Figure 2-5 depicts an enterprise bean inside an EJB container. The client application interacts with the enterprise bean via the EJB Home and Object interfaces. The EJB container provides

lifecycle management, state management, security, transaction and persistence management services for the enterprise bean.

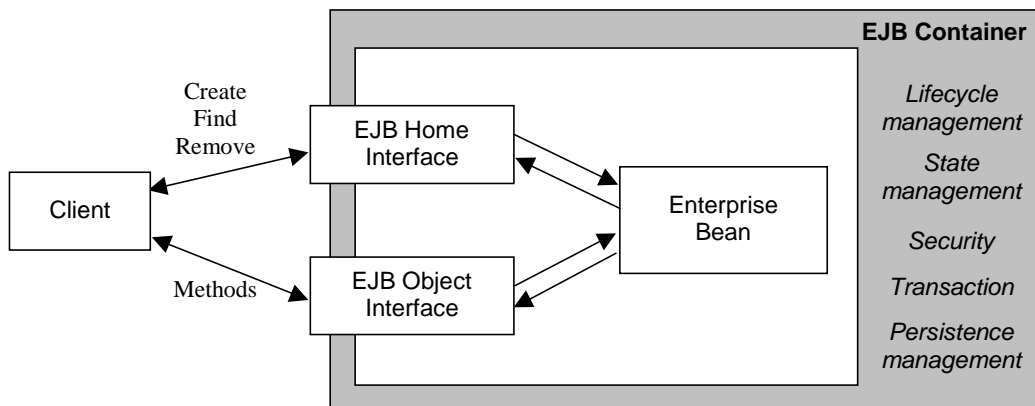


Figure 2-5: EJB container

### 2.1.6 OMG CORBA standards

The Object Management Group (OMG) is a non-profitable consortium formed by over 800 IT companies and research institutes worldwide. OMG was created to develop and promote the use of standards to achieve interoperability, portability and integration of software systems. Nowadays, OMG is focused on creating a component-based software marketplace by speeding up the introduction of standardised object software, with emphasis on reusability, interoperability and portability of components.

Initially, OMG concentrated its efforts in solving a fundamental problem on how to interoperate distributed object-oriented systems implemented in different languages across different platforms. As a result of this effort, OMG released in 1991 the Common Object Request Broker Architecture (CORBA). CORBA has developed based on the Object Management Architecture (OMA) [OMG97], which serves as a reference model for CORBA standards. OMA identifies five different constituents, viz., the Object Request Broker (ORB), Object Services, Common Facilities, Domain Interfaces and Application Interfaces.

Figure 2-6 depicts the OMA different constituents and their relationships. The elements of each different constituent (represented as circles) interact with each other and with the elements of other constituents by using the ORB as a software bus.

The ORB is the core of OMA, since it enables distributed objects to exchange requests and responses transparently across a network. CORBA defines the programming interfaces of the ORB component.

Object Services are general-purpose services (interfaces and objects) that are likely to be used by any CORBA-based application. OMA object services are collectively known as CORBAServices. The following services are currently specified: collection service, concurrency service, event service, externalisation service, licensing service, life cycle service, naming service, notification service, persistent object service, property service, query service, relationship service, security service, time service, trading object service and transaction service.

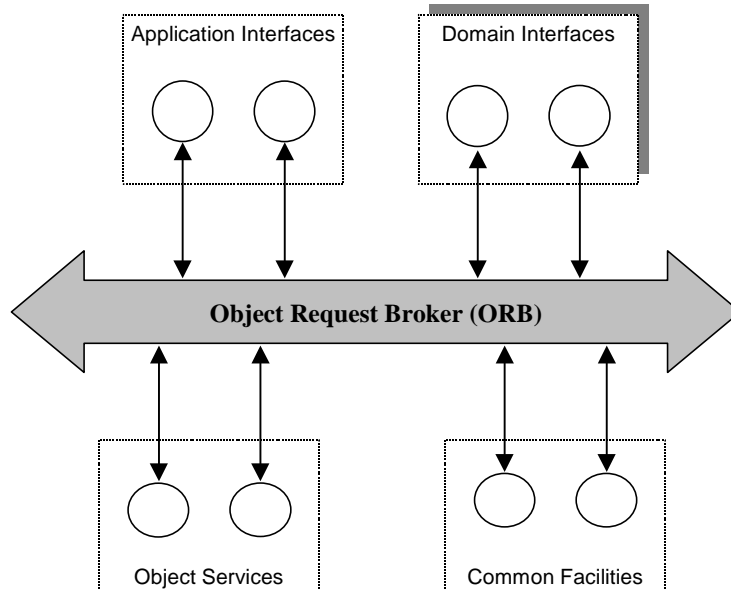


Figure 2-6: OMA reference model

Common Facilities are a collection of services that may be shared by a number of applications in different domains, such as user interfaces, and information, system and task management. These services, also known as CORBA facilities, are not as fundamental as the Object Services.

Domain Interfaces consist of domain-specific interfaces for different application domains, such as finance, healthcare, manufacturing, telecom, electronic commerce and transportation.

Application Interfaces are application-specific interfaces. These interfaces are developed separately by each vendor for each application and are not subject to standardisation by OMG.

CORBA 2.4 [OMG00a] does not support the notion of component explicitly. Instead, it provides the concept of CORBA object. A CORBA object is a 'virtual' entity capable of being located by an ORB and having client requests delivered to it. A CORBA object is reliably identified, located and addressed by its object reference. An object reference is not unique; instead, objects may be referred by multiple, distinct object references.

A CORBA object does not exist on its own. In order to perform client requests, there must be a concrete entity, called servant, that reifies a CORBA object and perform those requests. Each valid CORBA object is mapped to a servant. Therefore, an object reference actually identifies a servant. Servants can be created and destroyed transparently to the client as a result of issuing requests.

An interface is a static description of the possible operations that a client may request on an object that supports that interface. An operation is an identifiable entity that denotes the indivisible primitive of service provision that can be requested. Interfaces are described in a standard language called OMG Interface Definition Language (IDL). Although OMG IDL and Microsoft IDL have similar objectives, these are separate languages. Mappings from OMG IDL to specific implementation languages, such as C++ and Java, are available.

### CORBA Component Model (CCM)

Motivated by both the need for facilitating the building and deployment of component-based applications and the success of EJB, OMG is developing a new server-side component model as part of the new CORBA 3.0 specification, called CORBA Component Model (CCM) [OMG99b]. This component model aims at extending the CORBA core object model with a deployment model and providing a higher level of abstraction for CORBA and object services.

The two major advances introduced by the CCM are a component model and a runtime environment model. A component is an extension and specialisation of a CORBA object. Components are specified in an extension of OMG IDL, called Component Implementation Definition Language (CIDL).

A component is denoted by a component reference. A component also supports a variety of surface features, called ports, through which clients and other elements of an application environment may interact with this component. There are five different kinds of ports according to the component model:

- facets, which are the interfaces provided by the components for client interaction;
- receptacles, which are the connection points that describe the interfaces used by the component;
- event sources, which are the connection points that emit events of a specified type to interested event consumers;
- event sinks, which are the connection points into which events of a specified type are announced;
- attributes, which are named values primarily used for component configuration.

Figure 2-7 illustrates an abstract representation of a CORBA component and its ports. An equivalent interface is a special facet that allows clients to navigate through the component's interfaces and connect to its ports. A component has a single equivalent interface, but it may have multiple facets, receptacles, event sources, event sinks and attributes.

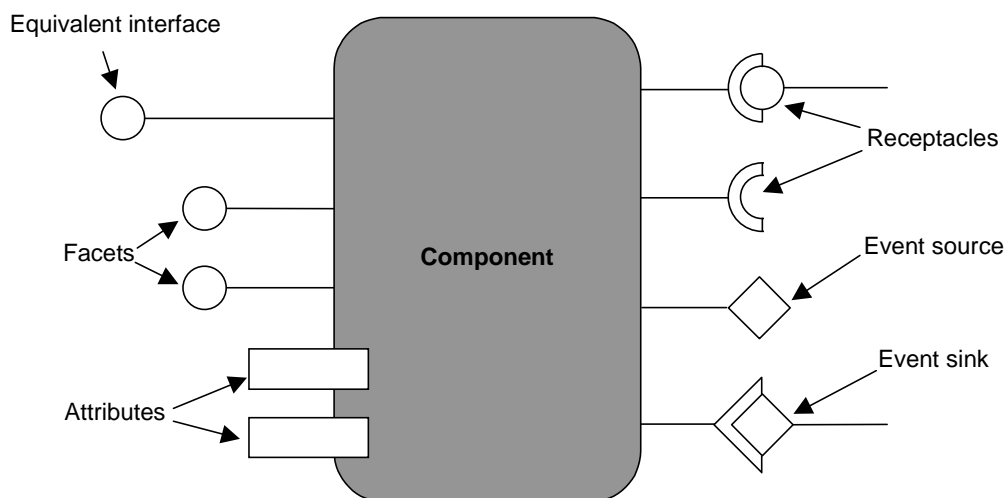


Figure 2-7: CORBA component

Similarly to EJB, the CCM defines different categories of components, viz., *service components*, *session components*, *process components* and *entity components*. A service component supports a single invocation per instance, is stateless and has no identity. A session component supports more than one invocation per instance, has a transient state and no persistent identity. A process component has a behaviour that may be transactional, has an explicitly declared state that is managed by the runtime environment and has an identity managed by the client. An entity component is similar to a process component, except for its identity, which is visible to the client but managed by the runtime environment.

A component instance is identified primarily by its component reference and secondarily by its set of facet references. Component instances are managed by component homes. A component home acts as a lifecycle manager for all the instances of a specified component type.

The CCM also introduces the concept of container. A container is a runtime environment consisting of a specialised POA plus a series of CORBA services to provide persistence, transaction, security and event notification capabilities to the components deployed inside the container. A container allows a client to navigate among the interfaces supported by all its components. A container is also responsible for connecting the provided and required interfaces of its components and connecting and transmitting the supplied and consumed events.

Since the CCM has been developed based on EJB, the integration between CCM and EJB components is facilitated. An EJB component can look like a CCM component to another CCM component and therefore it can be deployed in a CCM container. Similarly, a CCM component can look like an EJB component to another EJB component and therefore it can be deployed in an EJB container, respecting the constraint imposed by EJB that requires the use of Java as implementation language for the components embedded in the EJB container.

Figure 2-8 shows the interoperation of EJB components and CCM components in a mixed environment. A CCM client does not see the difference between a CCM component and an EJB component when both are deployed in a CCM container, and similarly an EJB client does not see the difference between an EJB component and a CCM component when both are deployed in an EJB container. This integration allows one to build an application using a combination of CCM and EJB components.

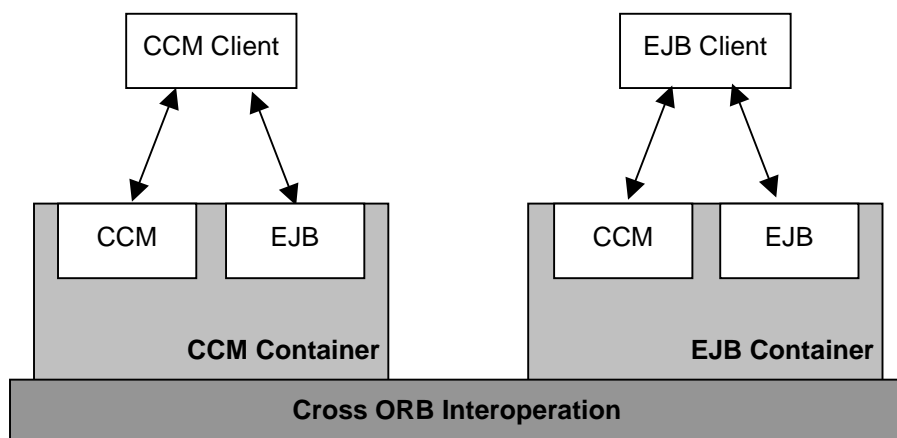


Figure 2-8: EJB and CORBA components integration

### 2.1.7 Component platforms

This section provides an overview of existing platforms for the deployment of component-based applications. These platforms are organised according to the component model presented in the previous sessions.

#### COM-based platforms

The platform support cornerstone of the COM family of component models is Microsoft's operating system Windows<sup>®</sup>. Since Windows 95, COM became an integral part of the Windows<sup>®</sup> operating system. In Windows 2000, COM+ became an integral part of the system as well. However, there is little support for COM-based platforms outside the Windows family of operating systems.

Further information on COM platforms can be found at <http://www.microsoft.com/com>.

#### EJB-based platforms

The major EJB-based platform is the Java<sup>™</sup> 2 Platform Enterprise Edition (J2EE) developed by Sun Microsystems. J2EE is a server-centric, multitier application architecture platform that provides full support for the EJB component model. This platform includes the Java core APIs and a number of enterprise Java APIs, such as Java Naming and Directory Interface, Java IDL, Java Messaging Service and Java Transaction Service.

J2EE is available in the following platforms, among others: Sun Solaris, Windows 2000/NT4 and Redhat Linux.

Other EJB-based implementations include Fujitsu Interstage, IBM Component Broker, Inprise Application Server, IONA iPortal Application Server and Oracle Application Server 4.0.

Further information on EJB platforms can be found at <http://www.javasoft.com>.

#### CORBA-based platforms

##### *Java IDL (JavaSoft)*

Java IDL is a very simple CORBA-enabled platform for Java. This platform includes an ORB that is available in every deployment of the Java 2 Platform Standard Edition (JDK<sup>™</sup> 1.2 platform or higher). The Java IDL ORB is compliant with the CORBA 2.3 specification but it supports only transient CORBA objects. Java IDL also provides a name server compliant with the CORBA Naming Service specification.

Currently, the JDK<sup>™</sup> 1.3 platform is available for the following operating systems, among others: Windows<sup>®</sup> 95/98/2000/NT 4.0, Sun Solaris and Linux.

Further information on Java IDL can be found at <http://www.javasoft.com>.

##### *Orbix 2000 (IONA Technologies)*

Orbix 2000 is one of the most commercially successful CORBA ORBs. This ORB provides full support for the CORBA 2.3 specification, including the portable object adaptor (POA), interface repository and object by value passing (dynamic type support). Orbix 2000 includes standard support for some of the CORBA services, such as the naming, security and transac-

tion services. Orbix 2000 is also one of the first ORBs to support some of the features of the CORBA 3.0 specification, such as portable interceptors, persistent state service (PSS) and asynchronous messaging interfaces (AMI).

Orbix 2000 is available in the following operating systems, among others: Compaq UNIX, Sun Solaris, RedHat Linux and Windows<sup>®</sup> 98/2000/NT4. Orbix 2000 also provides mappings for both C++ and Java.

Further information on Orbix can be found at <http://www.iona.com>.

#### *Visibroker 4.0 (Inprise/Borland)*

Visibroker is a comprehensive ORB. This ORB provides full support for CORBA 2.3 specification, including the POA, interface repository and objects by value passing (dynamic type support). Visibroker also includes support for the following CORBA services: event service, naming service, trading service, notification service, lifecycle service, property service, collection service, concurrency service, relationship service and time service.

Visibroker ships with different versions for the following operating systems, among others: Hewlett Packard HP-UX, Sun Solaris, RedHat Linux and Windows<sup>®</sup> 95/98/2000/NT4. Visibroker also provides mappings for both C++ and Java.

Further information on Visibroker can be found at <http://www.borland.com/visibroker>.

#### *Orbacus 4.0 (Object Oriented Concepts)*

Orbacus is another popular CORBA ORB. Orbacus is fully compliant with the CORBA 2.3 specification, including the POA, objects by value passing, interface repository and the new CORBA 3.0 portable interceptors. Orbacus 4.0 supports some of the CORBA services, such as the naming, event notification, property, time and trading services.

Orbacus 4.0 is available in different operating systems, including Sun Solaris, Hewlett Packard HP-UX, Linux and Windows<sup>®</sup> 95/98/NT/2000. Orbacus provides mappings for both C++ and Java.

Further information on Orbacus can be found at <http://www.ooc.com>.

## **2.2 Groupware technology**

Groupware technology is becoming increasingly popular. In the near future, it is expected that most software systems will provide some built-in groupware functionality. This section aims at presenting an overview of groupware technology. We concentrate on showing the diversity of currently available groupware systems and how these systems are usually classified.

The section is organised as follows. We first introduce some basic groupware terminology. Then, we describe a number of groupware systems and present some different groupware classification schemes.

### 2.2.1 Groupware-related definitions

This section provides a number of definitions related to groupware technology that will be used throughout this work. These definitions do not represent a consensus in the CSCW community, but rather they represent our personal view and understanding of the terminology that is usually found in the literature. We begin defining a *system* in generic terms:

*“A system is a set of interdependent interacting entities that form a unified whole.”*

Systems are found everywhere in our life, from natural systems, such as our body, a tree or a lake, to artificial systems such as a car, an airplane and a computer. Most of the time we are not concerned with the individual parts that form a system, but only with the outcome that these unified parts are able to produce. However, designers are more concerned with how the individual parts that form a system are structured and how they interact with each other.

*“A telematics system is a system that supports the interactions between people or automated processes or both via the application of tele-communication and information technologies.”*

The range of telematics systems is very broad by definition. We can find telematics systems everywhere, from banks to power supply companies, retailers, hospitals and so on and so forth. In this work we are interested in the study of a special type of telematics system, the so-called *groupware system*.

*“A groupware system is a telematics system that provides support for cooperative work.”*

This definition is quite generic and encompasses a lot of different systems that provide some support for cooperative work. Some of these systems support only one type of cooperative activity, while others support several types of activities. Some systems are focused on the development of new systems, while other are focused on the integration of existing ones. In order to distinguish one type of groupware system from another, some other definitions, such as groupware application, platform and environment, are necessary. A *groupware application* is defined as follows:

*“A groupware application is a groupware system that focuses on specific forms of collaboration.”*

There are many different ways of collaboration. For example, people may collaborate via the exchange of messages, sharing a common space, talking with each other via the phone or visualising each other in a face-to-face or in a computer-mediated basis. Whenever a groupware system provides some specific functionality that addresses one type of collaboration, we call this system a groupware application. However, in case support for multiple forms of collaboration is provided, we do not call this system a groupware application but a groupware system. For example, an audioconferencing application supports only one type of collaboration, a computer-mediated real-time conversation. Although some might see the integrated support of audio and video as a single form of collaboration, systems that support both forms of collaboration are usually called videoconferencing systems. Sometimes the term (groupware) tool is used as a replacement for the term application.

A *groupware environment* is another special kind of groupware system. A groupware environment is defined as follows:

*“A groupware environment is a groupware system that provides support for a range of collaboration forms through a set of integrated groupware applications and allows the integration of new applications.”*

The integration of individual groupware applications to support a number of collaboration forms is the main focus of a groupware environment. Further, a groupware environment also supports the integration of new applications, which add new forms of collaboration to the environment. For example, a videoconferencing system that also provides support for the exchange of messages and the manipulation of a shared workspace can only be called a groupware environment in case the integration of new forms of collaboration is supported, otherwise this system is simply regarded as a videoconferencing system.

Finally, a *groupware platform* is defined as follows:

*“A groupware platform is a groupware system that embeds a set of abstractions and facilities to support the development and/or deployment of groupware applications.”*

The provision of support for the development of new groupware applications is the main focus of a groupware platform. A groupware platform is usually based on a general architecture of groupware systems. The platform implements this architecture and provides some basic services for the development and deployment of new applications.

Figure 2-9 illustrates the relationship between groupware system, platform, environment and application. Groupware platforms are primarily focused on the development of groupware applications, while groupware environments are primarily concerned with the integration of groupware applications.

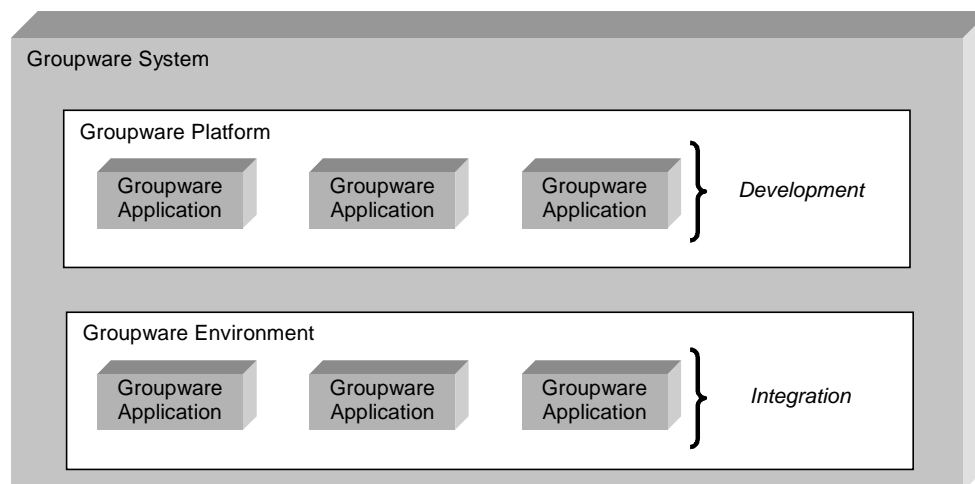


Figure 2-9: Groupware system, platform, environment and application

### 2.2.2 Groupware systems

Over the years, a number of different groupware systems were developed to support a specific work situation or a specific range of situations. The range of groupware systems available nowadays reflects the diversity of cooperative work tasks, duration, group size and location,

and organisational context. Some of these systems became commercially available, while most of them were just research prototypes.

In the sequel we identify and present a number of groupware systems that are frequently found in the literature. We do not intend to be complete neither in terms of the classes of systems covered nor in terms of the representatives of each class of application. Rather, we aim at giving the reader a rough idea of the diversity of the existing groupware applications and the functionality provided by them for its users. So, the reader can better picture the complexity involved in the development of applications with usually broad purposes and requirements. For a more comprehensive description of groupware systems we refer to [Hoft98].

### **Workflow Management Systems**

Workflow is defined as the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant of the business process to another for action, according to a set of procedural rules [WfMC99]. A business process consists of activities, and typically crosses boundaries within and between organisations. A *Workflow Management System* (WFMS) is defined as a system in which workflows are defined, performed, managed and monitored through the execution of software whose order of execution is driven by a computer representation of the workflow logic [WfMC99]. Examples of WFMS include Officetalk-D [ElBe82], Action Workflow [MWF+92], InConcert [McSa93] and Regatta [SMM+94].

Office procedure systems and image management systems formed the basis for workflow development [BrPe84, Elli79]. After the inclusion of routing and tracking capabilities in image management systems, the importance of a routing capability for the management of such systems as well as for the management of business processes in general was recognised [AbSa94, HsKl96]. Some general-purpose workflow management systems were developed, and in the late 1980's and early 1990's a large number of workflow management products became commercially available. However, the lack of standards for workflow systems made the interoperability between different products a difficult task.

To promote the development of workflow standards the *Workflow Management Coalition* (WfMC) was created in August 1993. This organisation defined a workflow reference model [Holl95, WfMC99] that concentrates mainly on interoperability and communication issues. This reference model basically defines the Workflow Enactment Services, a runtime environment consisting of one or more workflow engines, and five workflow APIs to provide interoperability with the enactment services. Some of the APIs are still in the process of being standardised, but currently a number of commercially available products, such as ActionWorks Metro [AcTe98] from Action Technologies Inc., MQSeries WorkFlow [IBCo99] from IBM, Staffware [Staf99] from Staffware Corporation and TBI/InConcert<sup>1</sup> from TBICO, conform to the published WfMC standards.

### **Electronic Mail Systems**

*Electronic mail (e-mails) systems* are by far the most used groupware systems. These systems allow the asynchronous exchange of electronic messages among users geographically dispersed around the world. Stegman distinguishes between e-mail applications and messaging

---

<sup>1</sup> <http://www.tibco.com>

systems [Steg97]. The former, also called e-mail clients, refers to the set of applications used by end-users to create, send and read messages, while the latter, also called e-mail servers, refers to the infrastructure used to store and transfer the messages, upon which e-mail applications operate.

E-mail clients are what most people associate with when they think in an electronic mail system. There are several e-mail clients commercially available, however Netscape Messenger<sup>2</sup> and Microsoft Outlook Express [Neib99] are among the most popular products. An e-mail client interacts with an e-mail server using standard APIs, such as POP3 and IMAP4. These standards allow e-mail clients to interoperate with any e-mail server that conforms to these standards. E-mail servers also use standard protocols to transfer messages among them, such as the Simple Mail Transport Protocol (SMTP). The Multipurpose Internet Mail Extensions (MIME) is another standard that is used in conjunction with SMTP for exchanging other types of messages rather than textual messages, such as applications, graphics, audio and video.

The increasing popularity of e-mail systems and, consequently, the exchange of messages in general created a phenomenon called information overload. Nowadays, it is not uncommon for someone to receive hundreds of e-mails from a variety of sources. Therefore, going through each e-mail to select only what can be of interest is a time consuming activity.

In the late 1980's the first applications to provide some sort of message filtering mechanism appeared, c.f. Information Lens [MGL+87] and ISCREEN [Poll88]. In Information Lens, messages were composed using semi-structured templates, which enabled a receiver to specify rules to filter and classify the messages sent. ISCREEN went further by leveraging its users with more capabilities to define rules to screen textual messages without adhering to the use of semi-structured templates.

Throughout the 1990's some applications were developed to provide more sophisticated filtering mechanisms that could be used not only to select e-mail messages but also web pages and voice messages, such as Tapestry [GNO+92], GroupLens [RIS+94, SKB+98] and CLUES [MaSc96]. Tapestry introduced the concept of collaborative filtering: users were encouraged to annotate messages and these annotations were then used for filtering purposes. GroupLens used a rating mechanism to evaluate different aspects of articles posted in newsgroups, such as interest in subject and quality of writing. Based on the ratings articles receive, GroupLens can filter articles out (the own with low rates), sort articles according to their rates or simply display the rates of the articles. CLUES is a dynamic mail filter that can be integrated into a visual mailer tool to automatically prioritise text and voice messages. CLUES uses different sources of information about working relationships to infer the relevancy of messages.

### **Chat Applications**

Besides e-mail, one of the most popular groupware applications is chat. *Chat applications* allow users to exchange textual messages, either synchronously or asynchronously. Whenever messages are exchanged asynchronously some sort of persistency mechanism is necessary to store them, usually in a server.

---

<sup>2</sup> <http://home.netscape.com>.

There are many variants of chat applications. Some of them allow only private or public conversation, while others allow a combination of both types of conversations. In a private conversation, a participant has to be explicitly invited, as opposed to a public conversation, where participants can engage in the conversation any time. Some applications rely only on social protocols to decide on who can engage in a conversation, such as, e.g., BABBLE [BrKE99].

Chat applications usually provide some sort of awareness functionality to its users, at least for the identification of the participants that are currently engaged in a chat conversation. BABBLE, for example, uses coloured dots, called marbles, to represent its users, and a large circle to represent the conversation. A marble inside the circle represents a participant that is currently engaged in the conversation, while a marble outside the circle represents a participant engaged in some other conversation.

Some chat systems also allow the exchange of pre-defined symbols, such as, e.g., the chat tool of ICQ<sup>3</sup>, and the exchange of images in general, such as e.g., Tickerchat [FMK+99] and the chat rooms found in most Internet portals.

### Shared Whiteboards Systems

*Shared whiteboards* are a particular kind of shared workspace in which the objects that populate the workspace are usually drawings. Thus, shared whiteboards are also known as shared drawing systems or collaborative drawing systems. Such systems are particularly useful to support meetings between distributed participants in which discussions and modifications in the drawings are needed. Modifications to the shared workspace are immediately visible to all participants of the workspace.

Shared whiteboards can be classified according to drawing capabilities that are provided. Peng [Peng93] identifies five different structures of graphics in shared whiteboard systems:

- video-captured images of freehand sketches, such as Videodraw [TaMi91], where freehand sketches are captured and transmitted to other participants using video cameras, monitors and projectors. Since there is no computational representation of the drawings involved one might question whether or not this kind of application should be considered a shared whiteboard;
- pixel-based graphics, such as GroupSketch [GRW+92], where drawings are simply defined as arrays of pixels (bit-map images);
- object-structured graphics, such as GroupDraw [GRW+92] and Conversation Board [BrGo92], where geometric structures, such as rectangles, circles and lines, are used to build the drawings. These geometric structures are usually programmed as objects and stored in a database to be addressed, manipulated and copied as individual entities;
- knowledge-based graphics, such as Network [FGL+92], where graphics primitives are programmed in terms of abstract construction and operation components that are specific to a particular design domain;
- semi-structured graphics, such as TeamWorkStation [IsMi91] and Tivoli [MoMC98], where graphics result from a combination of unstructured graphics (freehand or pixel based) with structured graphics (object-structured or knowledge-based).

---

<sup>3</sup> <http://www.icq.com>.

Shared whiteboards applications are rarely developed and used in isolation. These applications are usually part of a conferencing system, such as Microsoft's NetMeeting<sup>4</sup>, and they are used in combination with text, voice and video.

### **Co-authoring Systems**

*Co-authoring systems*, also called group authoring systems or shared editing systems, are used to improve the efficiency and quality of group writing as well as to support the cooperation between authors collaborating during the development of a document.

In the early days of groupware, co-authoring systems received a lot of attention due to the richness of cooperative features frequently required from those systems [GrSa87, FKL+88, ElGR91]. According to Michailidis and Rada [MiRa96] most of the features frequently found in co-authoring editors are related to concurrency control and data sharing, multi-user interfaces, auxiliary communication channels, information storage and retrieval, and the provision of awareness and notification services. Michailidis and Rada also identify the following possibly overlapping categories of co-authoring systems:

- conferencing and message-based systems, which include systems that take advantage of developments in computer-based conferencing and message systems to provide alternative communication channels for its users, e.g., Quilt [FKL+88] and MESSIE [SaHC93];
- group editors, which include systems that focus on the access and control to shared documents or information objects by its users, e.g., GROVE [ElGR91], CES [GrSa87] and Duplex [PaSS94];
- toolkit-based editors, which include systems that have been built based on a groupware toolkit, e.g., DistEdit [KnPr90];
- hypermedia editors, which include systems that support the organisation, structuring, representation and communication of hypermedia documents, e.g., SEPIA [HaWi92, SHH+92] and CoMedia [SaTr94];
- task-oriented editors, which include systems that have been designed to support a particular set of writing tasks, including brainstorming, drafting, revising and annotating, e.g., PREP [NKC+90];
- strategy-oriented editors, which include systems that have been designed to support particular forms of collaboration, including face-to-face and distributed meetings during the phases of collaboration that require intense communication and negotiation, e.g., DOLPHIN [SGH+94].

### **Group Calendaring and Scheduling Systems**

Probably, one of the toughest tasks in any medium to large size organisation is to schedule meetings involving a group of people. To cope with individuals time preferences and schedules when organising meetings is a time consuming activity, which can be significantly simplified by the use of *group calendaring and scheduling systems*.

Calendaring and scheduling systems are not the same thing, although they are complementary technologies. According to Knudsen and Wellington [KnWe97] a calendaring system aims at

---

<sup>4</sup> <http://www.microsoft.com/windows/netmeeting>.

placing and manipulating data on a calendar, while a scheduling system deals with the communication and negotiation between calendars for data placement. Knudsen and Wellington also propose a taxonomy for group calendaring and scheduling systems based on how these systems emphasize either calendaring or scheduling capabilities. Three classes of calendaring and scheduling systems are defined, viz., workgroup, departmental and enterprise.

Workgroup calendaring and scheduling systems, also called Personal Information Managers (PIMs), are targeted to individual users, who occasionally have to share information with other users. The calendaring-oriented features of these systems are richer than those equivalent features found in departmental and enterprise systems.

One of the precursors of modern workgroup calendaring and scheduling systems is the personal calendar system PCAL [Grei84]. PCAL offers some multi-user functionality, but it is mainly intended for personal use. Another workgroup calendaring and scheduling system is Visual Scheduler [BPH+90]. Visual Scheduler allows a user to manage both his/her calendar and somebody else's calendar. One of the most popular workgroup calendaring and scheduling systems nowadays is Lotus Organizer<sup>5</sup>. Lotus Organizer provides support for calendaring, scheduling, contact management and synchronisation with PDAs and cell phones.

Departmental calendaring and scheduling systems can be considered a compromise between workgroup and enterprise systems. The applications in this category do not focus specifically in either calendaring or scheduling features, but rather present a nice combination of these features with some degree of scalability. Many popular products that helped shape the calendaring and scheduling marketplace as we know it today can be included in this category, such as the multiperson calendar system MPCAL [GrSa87] and the meeting scheduler of Active Mail [GoSS92].

Outlook Calendar is the calendar and scheduling component of Microsoft Outlook [Neib99]. Outlook Calendar is probably one of the most popular departmental calendaring and scheduling products nowadays. This application provides a standard functionality for this category of products, including support for creating appointments and events, viewing and managing a calendar, organizing meetings and managing another user's calendar.

Enterprise calendaring and scheduling systems are targeted to large groups of users across local and wide area networks. Thus, these systems provide a highly scalable support for real-time scheduling and information sharing, at the expense of a more limited set of personal productivity and calendaring features for its users.

Examples of enterprise calendaring and scheduling solutions include Sun Microsystems Calendar Manager [Pale99] and Russel Information Sciences Calendar Manager<sup>6</sup>. Russell Calendar Manager is recognised as one of the most effective scheduling solutions for companies with mixed wide area network environments. This system assigns a calendar to each user and keeps calendars for resources and facilities in a distributed database. Users can access other users' schedules on a full or restricted basis. Built-in conflict resolution allows one to quickly determine the best time and place for meetings and to schedule several standby meetings for the same time slot.

---

<sup>5</sup> <http://www.lotus.com/organizer>.

<sup>6</sup> <http://www.russellinfo.com>.

Figure 2-10 shows the different classes of calendaring and scheduling systems based on the support provided and the number of users involved. Enterprise calendaring and scheduling systems provide a better support for scheduling, while workgroup systems emphasise the calendaring support. Departmental calendaring and scheduling systems sit in the middle, both in terms of the provided functionality and marketplace.

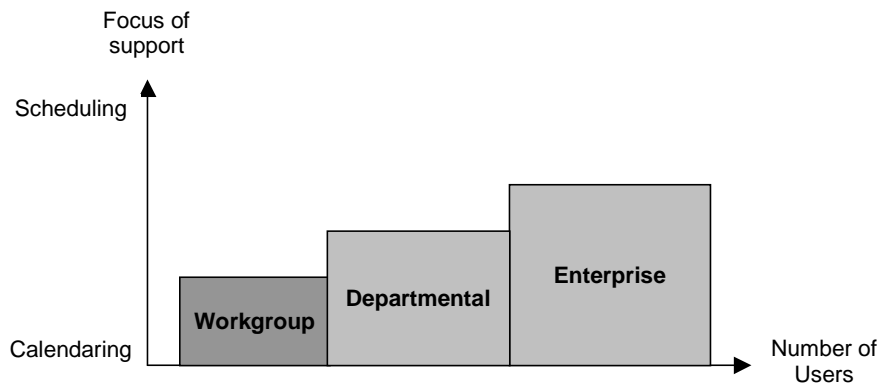


Figure 2-10: Calendaring versus scheduling support

### Collaborative Virtual Environments

*Collaborative Virtual Environments* (CVEs) comprise different types of systems, including Multi-User Dimension (MUD) environments and other place-based environments. These types of environments, particularly MUD-based environments, provide some additional communication support, which is not found in conventional groupware technologies [ChB199a].

MUD environments started as a complex variant of a computer game, in which players participate in a fictional world, moving through and acting in a textual virtual environment [Dour98]. In a MUD environment, activities take place within interconnected rooms, where other users and objects are located [ChB199a]. Users in a text-based MUD environment experience virtual rooms through textual descriptions about the objects and other users present in the room.

MUD environments have been used to provide collaboration in different contexts. For example, MOOSE Crossing [Bruc98] is a text-based MUD designed to support collaborative learning. Waterfall Glen [ChB199b] is also a text-based MUD that has been used to support work collaborations in a variety of situations across a period of several years. Collaborative Virtual Workspace (CVW) [SMD+97] is an object-oriented MUD with integrated functionality for audio, video, document sharing and whiteboard that provides support for geographically dispersed work teams.

Places are another popular metaphor used to build groupware applications. DIVA [SoCh94] is a place-based system representing a virtual office environment. DIVA unites in a single interface many of the key facilities needed to support distributed working groups, such as people, rooms, desks and documents. DIVA supports communication, cooperation and awareness, both synchronously and asynchronously.

TeamRooms [RoGr96] is another place-based system that provides support for synchronous and asynchronous collaboration, both co-located and distributed. TeamRooms adopts a metaphor of a room for the members of a team. A room serves both as a meeting place and as a

repository for the documents and the artefacts used to support the activities of the team members.

Worlds [FiTK95, FiKM96] is yet another place-based system. Worlds adopts the metaphor of locales to represent both users and tools. Audio and video functionality are also provided by this system.

Virtual reality has been increasingly used to build cooperative environments. VCS [ZWS+99] is a virtual environment that provides support for collaboration and awareness among multiple participants engaged in a conference. GCW [JaTW99] is another multi-participant virtual environment that provides support for collaborative learning.

### **Conferencing Systems**

*Conferencing systems* are among the most popular groupware systems in use nowadays. Conferencing systems can be roughly split into two categories, viz., *computer conferencing* and *multimedia conferencing*.

#### *Computer conferencing (asynchronous)*

Computer conferencing systems are a variant of electronic mail systems. Computer conferencing systems allow one to send messages to a uniquely identified place dedicated to the discussion of a particular subject. Messages posted there can be then retrieved and responded asynchronously over time.

The most well-known and largest conferencing application is Usenet, better known as “Internet news” or just “news”. Usenet uses the infrastructure of the Internet to provide thousands of dedicated places for discussion, which are called newsgroups. Some messaging systems, such as GroupLens, can be used to provide some mechanism to sort out the messages posted in the newsgroups according to various degrees of user interest.

Lotus Notes [GiDu97, HaFa99] is another groupware system that provides computer conferencing capabilities. Lotus Notes is by far the most successful groupware product to date, being used by millions of users worldwide. However, some authors argue that Lotus Notes is not only a groupware product, but also a groupware platform that provides the infrastructure for the deployment of new applications [Papo97].

#### *Multimedia conferencing*

Multimedia conferencing systems provide at least real-time audio and video conferencing support for remotely distributed participants. Some authors make a further distinction between audio conferencing systems (audio support only) and videoconferencing systems (audio and video support) [Hoft98], however this distinction is not significant within the context of this work. We use the term videoconferencing interchangeably with multimedia conferencing and the term audio conferencing to stress that only audio is supported. Some systems have also integrated support for the exchange of other types of media, such as messages and still images, and also provide a shared workspace.

Telephones can be considered as an early example of two-party audio conferencing systems. Modern (mobile) telephones provide support for multiparty audio conferencing and video is expected to be supported soon. However, due to the increasing availability of network band-

width in both Intranets and Internet worlds, videoconferencing systems are becoming increasingly popular in the groupware marketplace.

There are a number of benefits that contribute to the success and appeal of deploying a multimedia conferencing system in an organisation. Perey [Pere97] points out a number of tangible benefits of conferencing systems, including travel reduction and higher individual or group productivity, and intangible benefits, including the impression of sincerity and authority, the collection of accurate data for important decisions, the establishment of a community, and the promotion of creativity and consensus. However, the adoption of a multimedia conferencing system in a work environment is not always smooth as pointed out in [Sand92, MaGP99].

Several multimedia conferencing research prototypes have been developed since mid 1980's; most of these systems are quite generic. However, some of them were designed with a specific purpose in mind, such as RTCAL [GrSa87], which is a simple real-time conferencing system that supports the scheduling of meetings by its participants. RTCAL features audio support and a shared whiteboard.

VideoWindow [FiKC90] was one of the first videoconferencing systems to integrate high-quality video with full-duplex audio in the late 1980's. VideoWindow uses a large screen to produce life-size images of the participants.

MERMAID [WSM+90] is a distributed multiparty videoconferencing system developed in the early 1990's. Besides the support provided for audio and video, MERMAID supports information and document viewing and manipulation in shared windows.

DIVA [SoCh94] is a place-based system that supports both audio and video. This system is based on the concept of rooms representing virtual offices. DIVA maintains separate audio and videoconferences for each room available. Thus, in order to join a conference, a participant simply "enters" the desired room.

MAJIC [OMI+94] is a multiparty videoconferencing system that supports not only life-size images of the conference participants similarly to VideoWindow, but also multiple eye contact and gaze awareness. MAJIC also provides a shared workspace that can be used to share discussion materials and to "avoid" eye contact.

Montage [TaRu94, TaIR94] is a desktop videoconferencing system that supports lightweight interactions. This system supports two modes of interaction, viz., the glance and the visit modes. The glance mode concerns short-lived conversation in a small window intended, for example, to check someone's availability for a face-to-face conversation or for an extended interaction through the visit mode. The visit mode concerns a full-featured desktop videoconference with enlarged video window and access to a shared whiteboard.

NetMeeting is undoubtedly the most successful commercial multimedia conferencing product currently available in the market. Besides the audio and video exchange support, NetMeeting provides support for file transfer, chat, shared whiteboard and application sharing.

### **2.2.3 Groupware classification**

There are many different ways in which groupware applications can be classified: according to the functionality provided to its users, temporal and spatial forms of interaction, degree of

freedom provided for the application user and so on and so forth. Most of these classification schemes are incomplete and ambiguous; thus some overlapping is inevitable.

In the sequel we present some of groupware taxonomies. We believe that these taxonomies are among the most used and recognised among groupware researchers and practitioners.

### Product-based classification

One of the most intuitive ways to classify a groupware system is a product-based taxonomy. According to this taxonomy, systems with similar characteristics and functionality are grouped under the product name that most characterises them. The groupware systems described in section 2.2.2 are an example of this taxonomy.

One of the problems of this taxonomy is the intrinsic ambiguity that such a broad classification mechanism provides. Since some of the systems may have different sets of functionality, they could be placed in more than one category. In such cases those systems are primarily classified according to the functionality that characterises them most. For example, videoconferencing systems often have shared whiteboard and chat functionality available to their users. Still, these systems are largely known by its main functionality, i.e., videoconferencing.

Further, one of the tendencies in the CSCW arena is the development of groupware systems with multiple capabilities, instead of individual and specialised applications. So, as time goes by it will become harder and harder to distinguish the main functionality of an application and to classify it properly. Moreover, new classes of groupware applications, such as instant messaging systems [NaWB00], are constantly arising.

### Time-place matrix

The time-place matrix is probably one of the best known groupware classification schemes [Joha91]. The two dimensional matrix considers four classes of cooperative work interactions as a result of the temporal and spatial relations between the situations involved. Figure 2-11 shows the time-place groupware taxonomy.

	Same time	Different times
Same place	<i>synchronous local interaction (face-to-face)</i>	<i>asynchronous local interaction</i>
Different places	<i>synchronous distributed interaction</i>	<i>asynchronous distributed interaction</i>

Figure 2-11: Time-place matrix

The time dimension distinguishes between synchronous (same time) and asynchronous (different times) cooperative work. The space dimension distinguishes between local (same place) and remote or distributed (different places) cooperative work. Groupware systems can be classified by associating them with one or more types of interactions supported. For example, an electronic mail application is typically an asynchronous distributed application, while an electronic meeting room is typically a synchronous local application. Nonetheless, an electronic mail application that is used by two persons sharing the same office also provides support for asynchronous local interactions and, as argued by some [Hoft98], synchronous

local interactions, if used with high frequency. Most of the groupware applications to date fit into the synchronous distributed category, but only a few of them have been originally developed to support asynchronous local interactions, such as, e.g., electronic project rooms [Weis95].

The rigidity imposed by the time-place matrix has faced increasing criticism. Besides the controversy concerning the dichotomy between classification based on the functionality provided and classification based on the usage of this functionality, e.g., electronic mail used as (a)synchronous local application, there are two major criticisms [Hoft98]:

- the time-place classification scheme provides no intermediate stages between synchronous versus asynchronous work and local and distributed work. The sole definition of synchronisation and locality is controversial. For example, should a chat application that allows one to notice the text typed by another as soon as it is typed be considered synchronous or asynchronous? Should a groupware application used by co-workers located at adjacent offices be considered local or distributed?
- the time-place classification scheme also constrains the debate over situations that involve transitions between synchronous and asynchronous work and between local and distributed work.

### **Transportation and artificiality**

Benford et al. in [BBR+96] propose another taxonomy for groupware applications based on the degrees of transportation and artificiality provided by groupware systems in general.

Transportation is concerned with the extent to which the application users sense that they left their local space behind and have entered some new remote space. On one extreme, groupware systems with a small degree of transportation include face-to-face meeting supporting tools, in which the user's local space is preserved. On the other extreme, groupware systems with a high degree of transportation include collaborative virtual environments and telepresence systems, in which the application user perception is of total involvement and immersion into a remote space. Videoconferencing systems in general occupy an intermediate position in this sense.

Artificiality is concerned with the extent to which the application user-space is real (physical world) or synthetic (computer generated). On one extreme, there are face-to-face meeting supporting tools, telepresence systems and some videoconferencing systems, in which the degree of artificiality perceived by the application user is minimal, i.e., all the information is drawn from the physical world. On the other extreme, applications such as collaborative virtual environments and augmented reality systems have a high degree of artificiality since all the information is computer generated.

Figure 2-12 depicts a classification of groupware applications according to the transportation and artificiality dimensions. Videoconferencing systems with a shared data space, such as a shared whiteboard, are located in an intermediate position according to the artificiality dimension because they introduce some amount of synthetic information if compared with videoconferencing systems only.

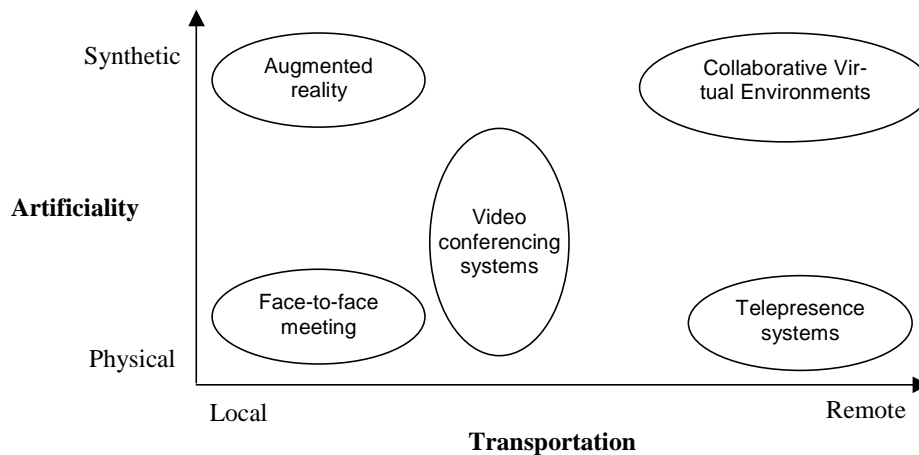


Figure 2-12: Taxonomy of shared space systems

### Restrictive versus Permissive applications

Galegher and Kraut propose yet another groupware classification scheme based on the degree of *prescriptiveness* or *permissiveness* embedded by the application [GaKr90]. Prescriptive systems have a model of intended courses or actions embedded in its structure to constrain and direct the behaviour of their users. Workflow management systems are a typical example of prescriptive systems.

Permissive systems do not attempt to constrain or direct the behaviour of their users, rather these systems seek to allow current practices to be extended into new practices not envisaged beforehand. The more permissive a system is, the more flexibility it allows. Examples of permissive application include electronic mail systems and videoconferencing applications.

Most groupware applications have a certain degree of both prescriptiveness and permissiveness embedded in their design. For example, in a permissive shared whiteboard in which no floor control policy is used or is available, everything is allowed. However, users often come up with social protocols to coordinate their actions, possibly making use of an audio channel or textual messages. In case some sort of floor control mechanism is employed, the user actions are constrained, so that the application acquires some level of prescriptiveness.

## 2.3 Component-based groupware

Component-based groupware is a new research area of CSCW that appeared in the late 1990's. Research on component-based groupware is mainly concentrated on the provision of platform support for the implementation and deployment of groupware systems. Component-based groupware platforms can be classified as *strictly component-based* and *loosely component-based* [SIHo99].

Strictly component-based platforms are formed by components that adhere to a (standard) component model. The use of standard components facilitates the integration and reuse of these components as well as the extension of the platform with third-party components. Loosely component-based platforms are formed by system parts that do not adhere to any component model. Although these system parts correspond to independent functional units, they cannot be classified as components.

In the sequel we present a number of component-based groupware platforms. We concentrate on strictly component-based platforms, since these developments are more relevant to our work. We do not intent to be complete, but rather we present the most significant groupware platforms to the best of our knowledge. For each platform, we present its architecture and main elements.

### 2.3.1 Live

Live [BDM+98] is a collaborative platform that provides support for the development and deployment of synchronous component-based groupware applications. This platform consists of a client-server infrastructure supporting a set of JavaBeans components. These components use this infrastructure to provide a high-level interface for groupware developers.

The main architectural concepts of Live are *named sessions* and *shared objects*. Object sharing is allowed within the context of a named session, which is supported by a server. Client applications can join a session to share objects on an object-by-object basis. Four different types of shared objects are available: *stateful objects*, which are high-level structures, e.g., strings, floats, arrays and vectors; *media stream objects*, which are used to provide audio and video support; *stateless objects*, which are used to provide event notification and synchronisation, e.g., channel objects and event streams; and *token objects*, which are used to provide concurrency control.

Figure 2-13 illustrates the main elements of the Live platform architecture. A dashed rectangle represents a host machine, a square represents an application, either client or server, an ellipse represents a named session inside the server, a circle represents objects and a dashed line connecting objects represents object sharing between client applications.

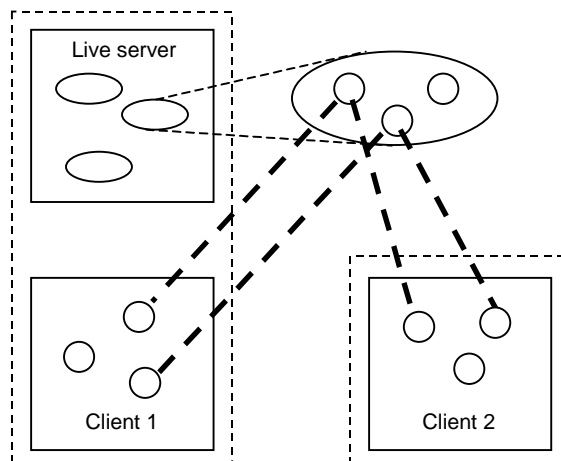


Figure 2-13: Live architecture

Live components are predominantly non-visual, i.e., a graphical user interface is not provided by these components. However, support is provided for the integration of third-party (visual) components into the platform. Live components include support for:

- *conference setup*, which enables an application to join and leave a session and keep track of other participants inside that session;
- *invitation*, which allows the invitation of new participants to take part in a shared activity;

- *data and event sharing*, which enables participants of a conference to share and unshare stateful and stateless objects;
- *access synchronisation*, which allows the use of floor control mechanisms to perform application level synchronisation;
- *media streaming*, which allows the use of media stream objects;
- *event streaming and temporal synchronisation*, which enables the temporal synchronisation of a stream object with another, and;
- *archiving*, which allows storing and retrieving of event and media stream contents to and from a local or a network file.

### 2.3.2 EVOLVE

EVOLVE [StHC99a, StHC99b, Stie00] is a component-based platform that aims at providing an Internet-based runtime environment for groupware deployment and supporting end-user tailorability. EVOLVE adopts a client-server architecture.

EVOLVE components are implemented using Flexibeans, an extension of the JavaBeans component model. Flexibeans extends JavaBeans with the concepts of *named ports*, *shared objects* and *remote interactions*.

The structure of a groupware system is described by five different types of files that are kept in the server: server Component Architecture for Tailorability (CAT) files, client CAT files, remote bind files, DCAT files and a user table.

A server CAT file describes the composition structure of the server side of a groupware system, while a client CAT file describes the composition structure of the client side of this system. A remote bind file describes the interconnection between a client CAT file and a server CAT file. A DCAT file describes a complete system by specifying the client and server components of this system. This file refers to a specific remote bind file to determine the connection between clients and servers. The user table relates DCAT files to users, specifying which applications are accessible to the users.

Figure 2-14 depicts the architecture of EVOLVE platform. A dashed rectangle represents a host machine, a C-square corresponds to the representation of a CAT file at runtime, a P-square corresponds to an instance of a component, a solid arrow corresponds to a 'refer to' relationship, a dashed arrow denotes the relationship between the representation of a CAT file at runtime and an instance of a component, and a dotted arrow associates a C-square or a P-square with their representation in a persistent environment. Tailoring is achieved through a set of operations offered by a tailoring API.

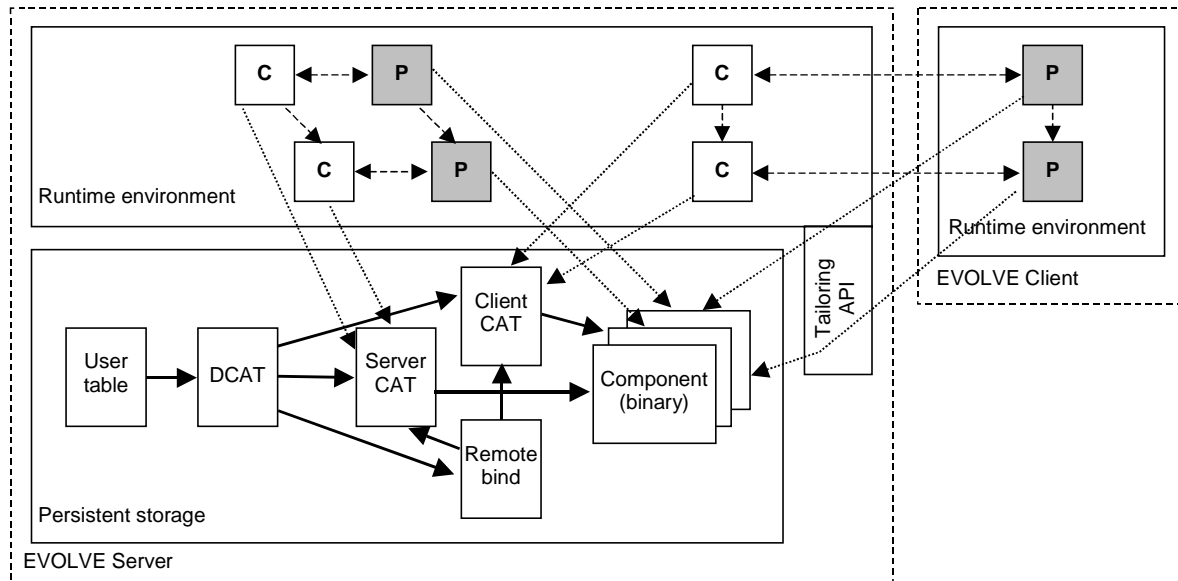


Figure 2-14: EVOLVE architecture

### 2.3.3 DISCIPLÉ

Distributed System for Collaborative Information Processing and Learning (DISCIPLÉ) [LiWM99,Mars99] is a real-time synchronous groupware platform. DISCIPLÉ provides support for the development and deployment of collaboration-aware applications and collaboration-transparent applications, i.e., applications originally developed for single users.

The architecture of DISCIPLÉ is hybrid, consisting of replicated client applications and centralised resource servers. Each application user runs a copy of the client application, containing all the components used in the collaboration. The main elements of this architecture are a collaboration bus, a multimodal human/machine interface, the application logic, resource servers and intelligent agents.

The collaboration bus is the most important element of the DISCIPLÉ platform since it brings the other elements together. The collaboration bus uses communication services to provide synchronous collaboration through real-time event delivery, event ordering and concurrency control. Further, the collaboration bus also provides some group awareness services.

A client application consists of a human/machine interface and the application logic. The multimodal human/machine interface provides several alternative ways for interaction between the application users and the application itself. This interface consists of facilities for speech recognition and synthesis, manual gesture sensing and gaze tracking.

The application logic consists of a number of JavaBeans components provided by the application developer. Thus, the application logic is not part of the DISCIPLÉ platform itself. An application user can add (import) new components to the application logic. These components become part of the application and consequently all application users can collaboratively interact with them.

Resource servers are points of contact for accessing common resources. The application client that first accesses a resource creates a resource server to manage that resource and publishes

its address on the collaboration bus to the other clients. Different servers are available for file access, database access and interprocess communication.

An intelligent agent layer spanning across all the other elements of the DISCIPLÉ platform is used to coordinate the work within each element and between elements. The knowledge captured at this layer is domain-based and application-independent.

Figure 2-15 illustrates the different elements of the DISCIPLÉ platform architecture and their relationship. Rectangles represent the elements of DISCIPLÉ architecture, while arrows represent the flow of information between these elements. There is an instance of the client application for each user of the system (replicated architecture) and a resource server for each different resource (centralised architecture).

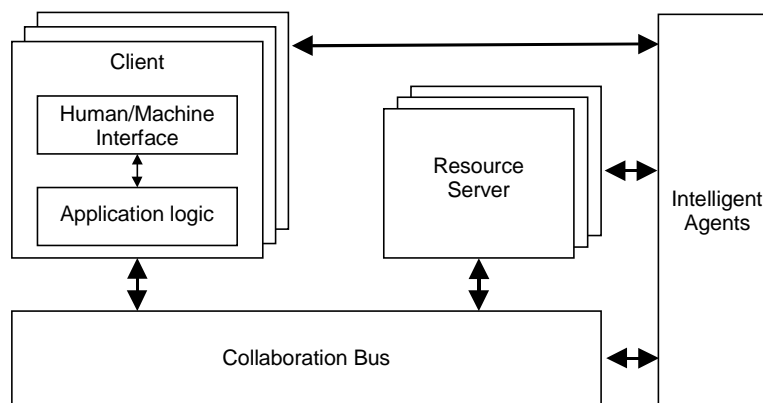


Figure 2-15: DISCIPLÉ architecture

### 2.3.4 CoCoWare

Collaborative Component softWare (CoCoWare) [S1Do01, S1Bi00] is a platform targeted at end-user composability and extensibility of component-based groupware. To the best of our knowledge, CoCoWare is the only component-based groupware platform based on CORBA.

CoCoWare implements a replicated component groupware architecture named Cooperative People and Systems (CooPS). CooPS is an extension of the model proposed by Ter Hofte in [Hoft98].

A groupware application according to CooPS consists of four types of components, viz., conference manager, conference tool, conference coordinator and conference enabler.

The conference manager component is the most important component of CooPS reference architecture. The conference manager is responsible for managing conference existence and participation. Sub-conferences of an existing conference are also managed by the conference manager. This component enables end-users to start and stop conferences, to join or leave conference and to invite or expel conference participants. There is one instance of this component for each conference the user participates.

Conference tool components are the actual means for users to communicate. Multiple tools can be used in each conference. CoCoWare contains a number of such components, including components for audio conferencing, video conferencing, chat, collaborative web browsing and application sharing.

The conference coordinator component maps users to user roles and enacts coordination policies on user actions based on these roles and on the state of the conference. The conference coordinator is an optional component. One coordination policy may be related to multiple conference tool components, but a component may be coordinated by at most one component.

The conference enabler component provides end-users and groupware components with information about users that can be invited to a conference or conferences that can be joined by users. The conference enabler is another optional component, since the information provided by this component can be obtained by other means, such as e-mail or telephone.

Figure 2-16 shows the main component types of CooPS architecture and their relationship. A dashed rectangle represents a host machine, while a double edge arrow represents the flow of information between different hosts. A grey rectangle and a white rectangle represent an optional and a mandatory component, respectively. Lines represent the interconnection or components.

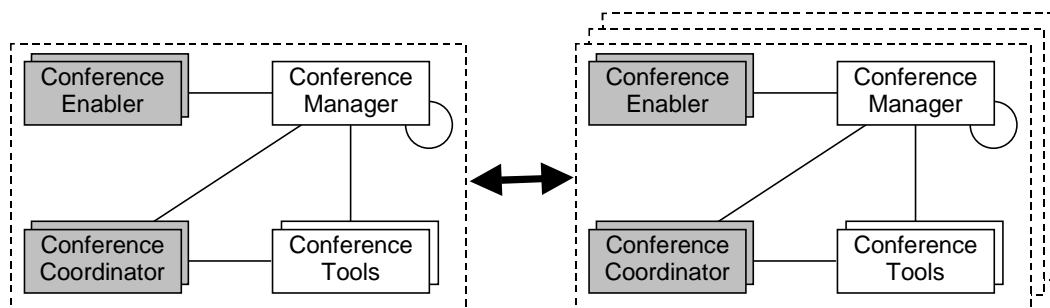


Figure 2-16: CooPS architecture

### 2.3.5 DACIA

Dynamic Adjustment of Component InterActions (DACIA) [LiPr99,LiPr00] is a platform for building mobile groupware applications. DACIA allows the construction of adaptable applications via the flexible composition of separate components.

DACIA components are not based on any of the component models presented in section 2.1. Instead, DACIA components communicate with each other via the exchange of messages through input and output ports, including stream data. A DACIA component is known as a Processing and ROuting Component (PROC). A PROC can apply some transformations to input data streams, such as the synchronisation of input data streams and the split of the items of an input data stream into multiple destinations.

An engine is another element of the DACIA architecture. An engine runs on each host and is responsible for administrative tasks, such as maintaining the list of PROCs and their connections, establishing and removing connections between PROCs, migrating PROCs from a host to another, and establishing and maintaining connections and communication between hosts. An engine has only partial knowledge about PROCs running on other hosts and the configuration of the application itself.

An engine usually works together with one or more monitors. A monitor represents the part of the application that gathers performance data and implements relocation and reconfiguration policies. However, monitors are optional, since applications can also be reconfigured

manually. While the engine and the PROCs are general-purpose, a monitor is usually application-specific.

Figure 2-17 illustrates the elements of DACIA platform architecture. A dashed rectangle represents a host machine, while an oval represents a component. The arrows between components represent the direction of data flow within the application.

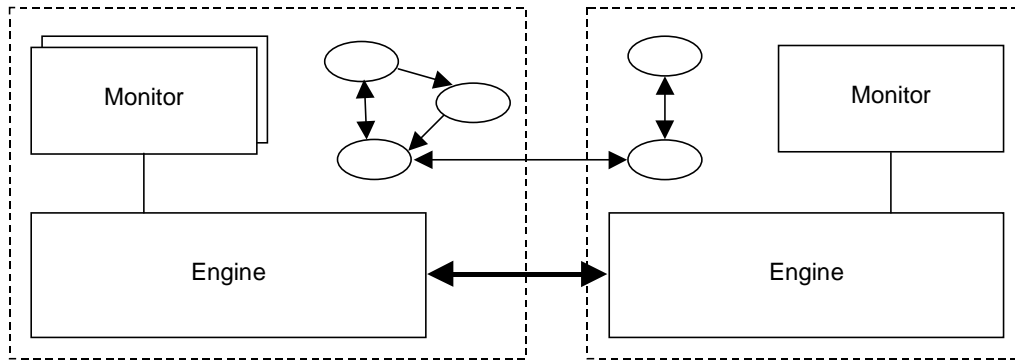


Figure 2-17: DACIA architecture

### 2.3.6 DreamTeam

DreamTeam [RoUn00] is a synchronous component-based groupware platform. This platform contains a development environment, consisting of groupware specific solutions, a runtime environment, providing special groupware facilities, and a simulation environment.

The architecture of DreamTeam is fully replicated. A DreamTeam application consists of multiple resources, each one containing a standard interface. Components are the most important type of resources. Other types of resources include user interface resources and data resources. DreamTeam components are called TeamComponents. TeamComponents are special resources that can aggregate resources themselves, except for other TeamComponents. Despite being Java-based, TeamComponents are not based on any Java component model.

Resources are not allowed to communicate with each other directly. Instead, messages from one resource to another are routed through the application interface. There are two communication modes between resources: intra-site communication, in which a resource communicates with its application (same host), and inter-site communication, in which a resource communicates with its peer resource in another host. Both communication modes are realised using events and callbacks. Intra-site communication uses standard Java method calls, while inter-site communication uses an underlying message-based communication system to distribute the events.

Figure 2-18 shows the main elements of the DreamTeam architecture. A dashed rectangle corresponds to a host machine, a small rectangle represents a resource, a solid arrow represents intra-site communication and a dotted arrow represents inter-site communication.

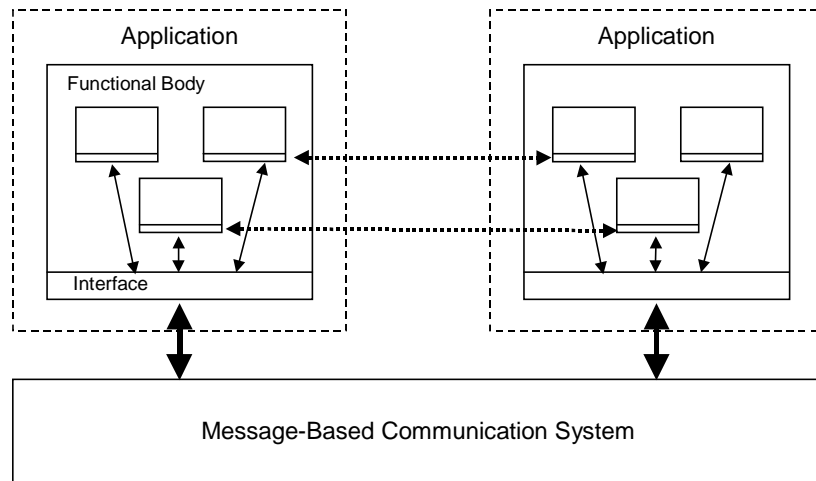


Figure 2-18: DreamTeam architecture

The standard interface of a component provides several method calls to access its attributes, such as profile data, integration mode, collaboration mode and state, and event notification.

A component can be integrated within its application seamlessly, i.e., the component user interface becomes part of an existing window, or as an anchor, i.e., the component user interface is a separate window. A component has also different levels of collaboration, such as private, exclusive and shared. A component running on a private collaboration mode has no shared data. A component running on an exclusive collaboration mode has some shared data that can be manipulated by only one user at a time. A component running on a shared collaboration mode has shared data that can be manipulated by several users simultaneously.

DreamTeam has several mechanisms for concurrency control, such as a transaction lock for the application, a monitor lock for each resource, and explicit locks for variables and method calls. DreamTeam also provides a number of basic components, including drawing components, user interface components, a browser component and an audio component.

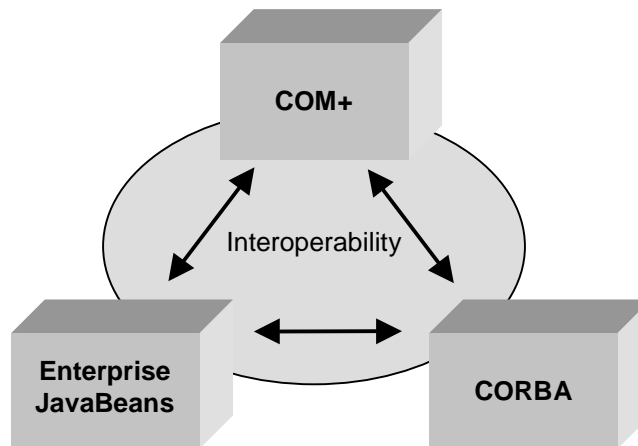
## 2.4 Conclusion

Component technology has been increasingly used to develop software applications. Despite being young, the component marketplace has grown steadily in the past few years. Still, recent estimates account for a small number of off-the-shelf components available in the market, approximately 4000 components [Will00b]. This number is five to ten times smaller than the number of components that are needed to achieve a critical mass on the selection and use of components in general.

A number of developments may serve as stimulus to the growth of the component market, but we particularly distinguish between two of them, viz., component model stability and interoperability. One of the major requirements needed to stimulate the growth of any market is stability, particularly in the component market the stability of existing component models. Component models are likely to evolve constantly, but major changes in these models hinder the market growth. Nevertheless, the major component models are expected to become stable in the near future, especially after the completion of the CORBA component model.

Another major development that may contribute to boost the component market is the so-called interoperability triumvirate [Fisc00]. Components developed according to one model should be able to seamlessly interoperate with other components independently of their component model. Nowadays, a number of products and standards can already be used to solve this problem and other improved solutions are on the way, particularly the alignment between the CORBA component model and EJB.

Figure 2-19 shows the component interoperability triumvirate among COM, CORBA and EJB.



*Figure 2-19: Component interoperability*

Groupware systems are also likely to evolve in the coming years. New applications will become available and traditional software systems will incorporate group features.

The development of groupware systems is an endeavour far from being trivial. To cope with different work situations, different applications are needed. Each application imposes its own particular requirements and the integration of these separate applications into one groupware system poses a new burden to the development process.

Component-based groupware tries to solve some of the problems faced by groupware developers by means of the provision of platform support and a set of components or services that can be reused and combined to form groupware applications and systems. Still, there is no specific methodology to help software developers, in general, and groupware developers, in particular, in choosing the appropriate components to build an application and to design new components accordingly.

As pointed out by Williams in [Will00a], one of the biggest challenges facing component-based development is the lack of a methodology for building components and component-based systems.

---

# Chapter 3

## Component-Based Design

The primary objective of this chapter is to provide an overview of our component-based groupware design methodology and to delimit the scope of this methodology. In order to accomplish this objective, the chapter first introduces some basic software-related design concepts and provides an overview of software development in general. Thereafter, the chapter outlines our design methodology and presents an overview of some UML-based methodologies.

The chapter is organised as follows: section 3.1 introduces some basic design concepts; section 3.2 discusses software development in general; section 3.3 introduces our component-based design methodology; section 3.4 presents the scope of our methodology; finally, section 3.5 provides an overview of related software development methodologies.

### 3.1 Basic concepts

#### 3.1.1 Abstraction

The process of developing products in general, and software in particular, has many things in common. Whether a piece of software, an automobile, a mobile telephone or a DVD player is being developed, it is possible to identify some basic design concepts in most situations. *Abstraction* is such a core concept. We define abstraction as follows:

*“An abstraction is a representation of only those aspects of an entity that are considered relevant at some stage during its development.”*

During the development of a complex product, it is unfeasible to tackle all the aspects involved at once. Instead, an abstraction is used to focus on the essentials of the product, while ignoring the other aspects for the time being.

Similarly to a design, an abstraction is something intangible, i.e., something that exists only in the mind of someone. However, in order to communicate the relevant aspects of the abstraction in concrete terms, we need a tangible representation. An accurate representation of an abstraction is called a *specification*. A specification can be developed according to a specification language or design notation.

For example, during the development of a car, the car designers may use drawings (specifications) to capture and reason about the intended features of the car and to communicate their design decisions.

Figure 3-1 illustrates the concept of abstraction. The board drawing represents as an abstraction of a car for the purpose of designing it.

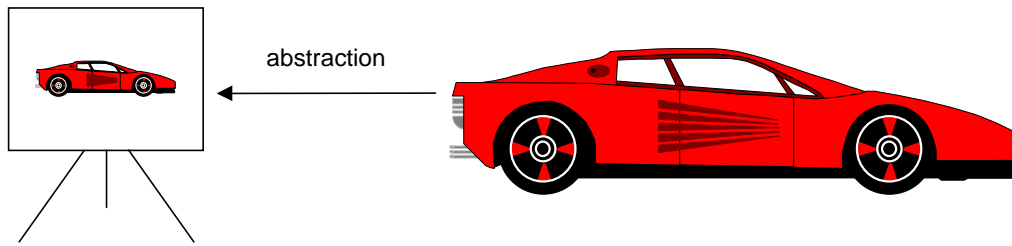


Figure 3-1: Abstraction illustrated

### 3.1.2 Abstraction levels

Any given specification should be produced to capture the requirements of the product to be developed and to serve as basis for developing this product. Nevertheless, despite the usefulness of the concepts of abstraction and specification, as means of representing abstractions, the realisation of a product based on a given specification may not be straightforward.

On one hand, based on the same set of requirements it is usually easier to produce an abstract or high-level specification, i.e., a specification whose abstraction is semantically closer to the requirements but far from the intended product, than to produce a concrete or low-level specification, i.e., a specification whose abstraction is semantically far from the requirements but closer to the product itself. On the other hand, due to the gap between the abstraction captured in a specification and the intended features of a product, the realisation of this product based on an abstract specification is usually more difficult than the realisation of the product based on a corresponding concrete specification.

Figure 3-2 illustrates the difference between the use of an abstract specification and a concrete specification to realise a corresponding product based on the same set of requirements. While the realisation of the product based on the abstract specification is much more difficult, the realisation of the product based on the concrete specification is easier. However, the development of an abstract specification is usually much easier than the development of a concrete specification.

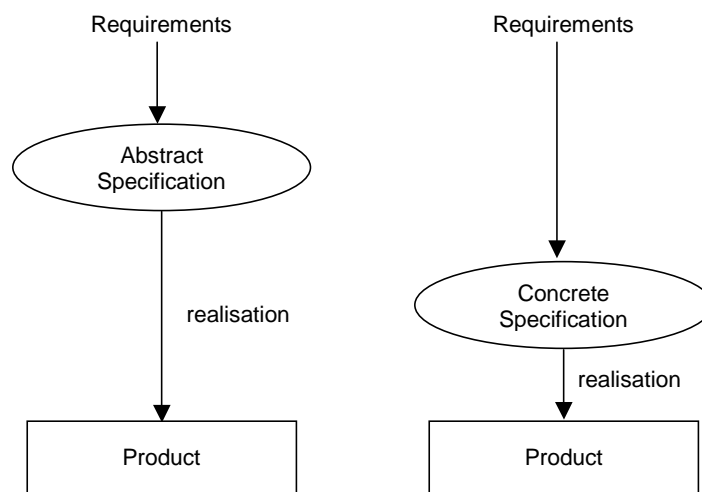


Figure 3-2: Abstract versus concrete specifications

In order to facilitate the development of a product we should be able to produce first an abstract specification of the product and incrementally produce new (more concrete) specifications with an increasing amount of details. Therefore, the development of a product should

comprise several design steps until a concrete realisation of the product can be achieved. Such an approach to development is known as *step-wise design* (see Figure 3-3).

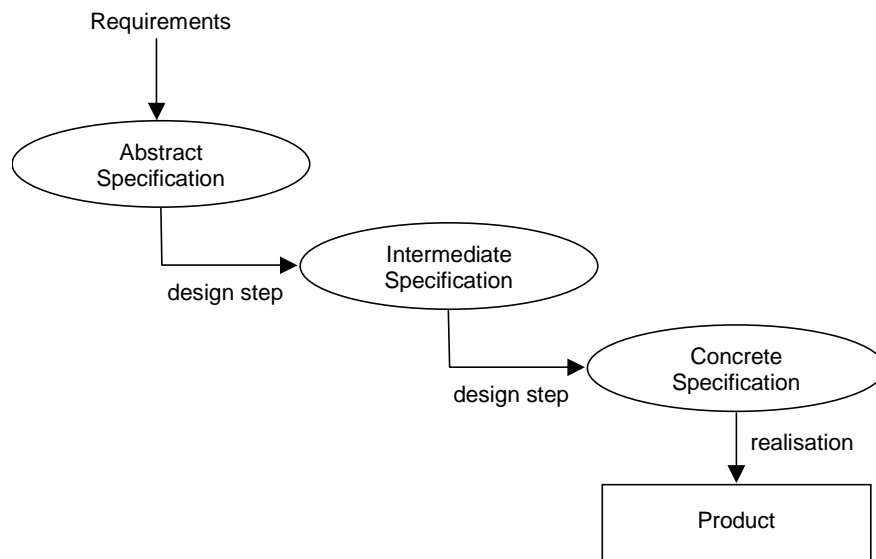


Figure 3-3: Step-wise design

Considering our car design example, a single drawing is not enough to guide the production of the car. Other drawings are necessary to represent each part of the car separately. For each part of the car, initial drawings have to be replaced by new drawings containing more detail. Still, the drawings for each part of the car must take into account the other parts, such that the individual parts of the car can be eventually assembled. This process continues until each and every part of the automobile has been properly designed and is ready to be realised using concrete materials, such as steel, rubber and plastic.

A step-wise development leads to the establishment of a number of *abstraction levels*. We identify abstraction levels as follows:

*“Given a series of abstractions ordered according to a refinement/abstraction relationship, we can identify for each abstraction a separate abstraction level.”*

*Refinement* and *abstraction* are opposite and complementary types of relationships or design activities. Through refinement, an abstraction is made more concrete through the introduction of details, while through abstraction, details of a more concrete abstraction are omitted. An important property of either refinement or abstraction is that the resulting abstraction/specification should conform to the original one.

Figure 3-4 illustrates a number of abstraction levels.

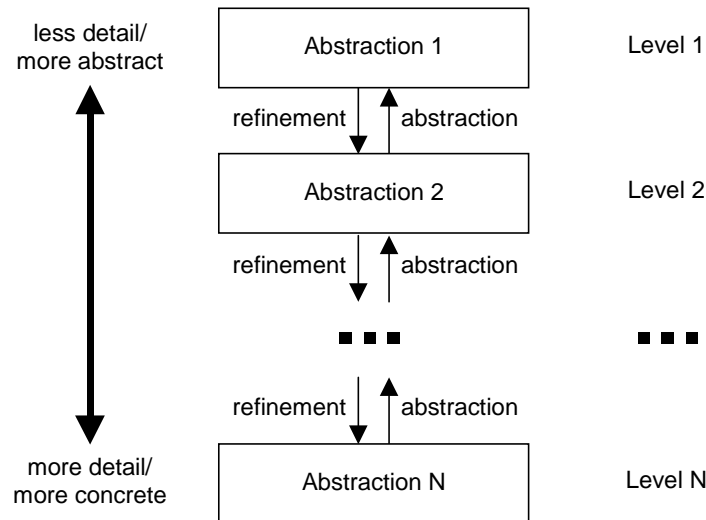


Figure 3-4: Consecutive abstraction levels

### 3.1.3 View

Another basic concept often associated with the design process is *view* or *perspective*. This concept is frequently used interchangeably with the concept of abstraction. However, in this work we adopt the following definition of view:

*“A view defines an abstraction that focuses on a specific set of concerns.”*

Our definition of view is closely related to the definition of viewpoint as defined in the ODP reference model [ISO95, Raym94]. Although some authors make a distinction between view and viewpoint [KaSt00], this distinction is not relevant in the context of this work.

Figure 3-5 illustrates two separate views of a car. The performance view focuses on performance aspects of the car, such as maximum speed and acceleration. The cost view focuses on cost aspects of the car, such as the costs for the standard vehicle, optional items and insurance. The set of concerns addressed by these views is completely independent from each other and thus cannot be compared. However, since they are applied to the same object, an implicit or indirect relationship is established between them.

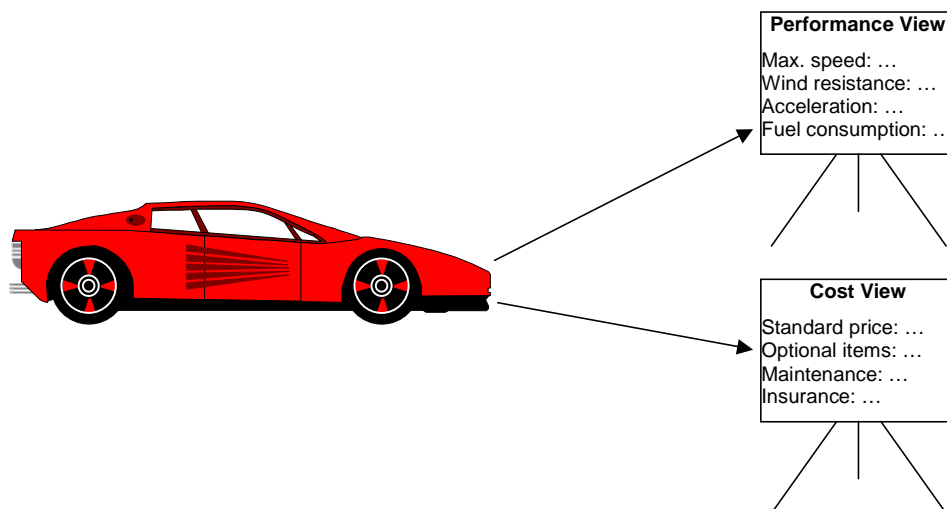


Figure 3-5: Different views of the same object

Views may complement each other to form a description of a whole (a single abstraction). In this case, the information contained in a view may overlap with the information contained in other views. For example, in our car example, we may have a view that focuses on the separate parts that form the car and another view that focuses on the costs and suppliers of each separate part. These two views share some information and consequently complement each other.

### 3.1.4 Concern levels

The development of a product according to different levels of abstraction facilitates the design process in general. However, in the design of complex product we should be able to focus on different sets of concerns at different times along the design trajectory. Each separate set of concerns could then be used to drive the development of a number of specifications, possibly across different abstraction levels.

In case different sets of concerns can be related along a design trajectory, different *concern levels* are identified. We define a concern level as follows:

*“A concern level comprises a number of concerns, defined according to an objective along a design trajectory, which guide the development of a number of related abstractions.”*

A concern level has a well-defined objective that distinguishes or characterises the abstractions that can be developed. Based on this objective, a number of specifications can be produced to capture these abstractions.

For example, in the design of a car two major concern levels that can be identified are the car as a whole and the car as an assemble of parts. The former concern level is concerned with the car as a whole and the issues associated with that, such as the model type, dimensions, weight, number of doors, price, fuel consumption and maximum speed. The latter concern level is concerned with the individual parts of the car and the issues associated with that, such as, for example for the engine, the horsepower, fuel capacity, torque, compression rate and weight.

Two concern levels can be related to each other if the concerns of one level can be related to the concerns of the other. Consequently, the abstractions that can be captured at one level can also be related to the abstractions that can be captured at the other level via the same kind of relationship.

Figure 3-6 shows the general relationship between two concern levels and between specifications developed within the same level or across different levels. Considering the relationship R1, established between Concern Level 1 and Concern Level 2, the same type of relationship can be observed between Specification 1 and Specification 2 and between Specification 1 and Specification 3. The relationship R2, established between Specification 2 and Specification 3, indicates a relationship common to specifications that belong to the same concern level, namely Concern Level 2.

Within a concern level, specifications can be produced according to different views or perspectives, which can be seen as subsets of the concerns defined within the level. These specifications can also be organised according to different abstraction levels, provided that a refinement/abstraction relationship can be established between these specifications. Therefore,

it is possible to produce multiple specifications according to different abstraction levels within a single concern level.

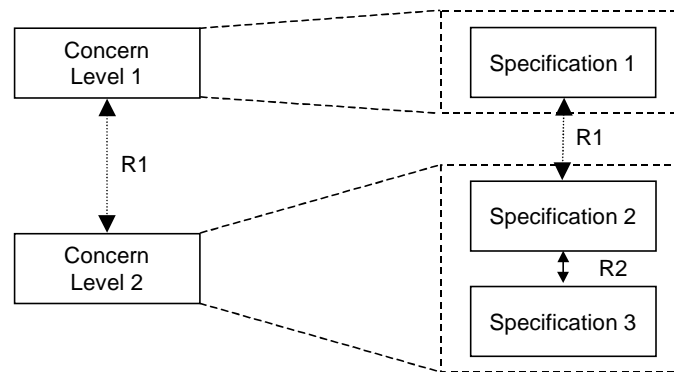


Figure 3-6: Relating different concern levels

## 3.2 Software development process

The development of complex software systems in general is difficult and challenging. Software developers usually face the task of delivering a product that satisfies the current user needs within strict time and budget constraints. Further, the required software product must also satisfy other requirements, such as quality, performance and reliability. Therefore, software development is usually tackled in a number of steps or phases.

A software system is traditionally developed according to the following large-grained phases: *requirements analysis*, *specification*, *design*, *implementation*, *test and integration*, *operation and maintenance*, and *retirement*.

Requirements analysis is the starting phase of the software development process. In this phase, the functional user requirements of a new software system are identified and documented. There is no distinction between the system and its environment because no one knows a priori which functionality should be included in the system and which functionality should not. Not only functional requirements are identified during requirements analysis, but also other requirements and constraints, such as cost and time to develop, reliability, availability and performance. In order to properly identify these requirements, software designers and end-users must interact.

Requirements analysis is considered the most critical phase in software development, since errors or misinterpretations made in this phase are propagated throughout the process. If these problems are not detected and fixed in the early stages of the development process, the costs of fixing them later grow tremendously [Faul97]. Occasionally, products that do not satisfy user needs may be developed if the user requirements were not properly identified.

The specification phase starts once the user requirements are clearly documented. This phase aims at providing a specification that describes what the system should do to fulfil the user requirements. Further, a line is drawn between the system and its environment, i.e., the system is separated from its environment. The specification of the system provides a description of the intended service for its users and it must include all the requirements previously identified. The system specification forms the basis for the remaining phases of the development process. The resulting software system is correct if it conforms to its specification.

The design phase aims at describing how to realise the service provided by the system as defined in the specification phase. The design phase is usually accomplished in two separate (sub-)phases: architectural design and detailed design. In the architectural design, a structure of modules is devised to support the service of the system, while in the detailed design each module is further refined and specified until a description that is ready for implementation is obtained. A module is an independent and interoperable part of a system. Several refinement steps may be needed to obtain a detailed design description.

The implementation phase aims at translating the design specification (of each module) to an implementation. An implementation is a description of a software system that can be interpreted or executed by a computing environment.

The test and integration phase aims at testing first each module individually, and then all the modules integrated as a whole. The purpose of testing is to evaluate if the resulting implementation complies with the specification developed in the specification phase. Once the system has been integrated and tested, it is ready for operation.

The operation and maintenance phase starts after the software system is delivered to its users. Any modification made to the system after it has been delivered is regarded as maintenance. Maintenance comprises different types of activities [Benn97]: *perfective*, *adaptive*, *corrective* and *preventive*. Perfective maintenance is performed as a result of user requests. Adaptive maintenance is needed as a consequence of changes in the software environment, such as changes in the operating system, hardware and other support services. Corrective maintenance is performed to remove faults in the system, while preventive maintenance is performed to facilitate future maintenance activities. A software system typically evolves through several years of maintenance. Therefore, the costs involved with this activity typically surpass the costs of developing the system itself [Scha93].

The retirement phase is the final phase of the software development process. At this stage, any further maintenance would not be cost-effective and therefore a new system has to be designed from scratch to replace the old one. Alternatively, the system has lost its usefulness, since its functionality is no longer needed, and should be completely abandoned.

The development of software according to the aforementioned phases corresponds roughly to the classic software life cycle, proposed in the early 1970's and known as the *waterfall model* [Royc70]. Figure 3-7 illustrates a variant of the waterfall model, incorporating the retirement phase. Boxes represent the different phases of the software life cycle, while arrows represent the changes in phases.

Although the phases of software development are important milestones, they are considered to be too abstract and coarse-grained to describe all the activities performed during software development and maintenance of complex software systems. Instead, finer-grained descriptions of the activities that should be carried out, such as those provided by a *process model*, are usually more appropriate for such purpose.

A process model is used to describe the artefacts, i.e., any kind of information created, used or changed in the development process, the activities (processes) that produce externally visible changes to these artefacts and their relationships. Process models are also known as *software development methodologies*.

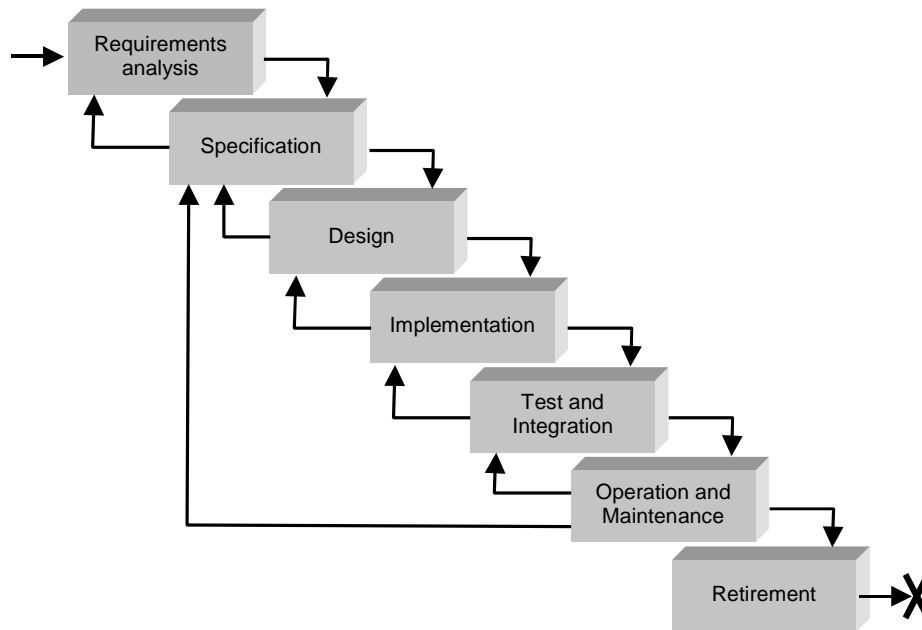


Figure 3-7: Waterfall model

### 3.3 Methodology overview

Our component-based design methodology is based on a number of concern levels and general views defined across these levels.

#### 3.3.1 Methodology concern levels

Our methodology proposes the design of a groupware system according to four concern levels, viz., the *enterprise level*, the *system level*, the *component level* and the *object level*.

The enterprise level is concerned with the enterprise context of the system under development. This level focuses on capturing an abstract representation of a cooperative work process, using a number of enterprise-related concepts and relationships between these concepts. This representation serves as basis for the design of an appropriate computer support for the process, to be provided by the system under development. Therefore, at the enterprise level there is no clear separation between the system under development and its environment.

The system level is concerned with the contribution provided by the system under development to the cooperative work process described at the enterprise level. At the system level the boundaries between the system and its environment are defined in terms of what is provided to and what is required from the system environment.

The component level is concerned with the decomposition of the support provided by the system under development in terms of the support provided by a number of interrelated components. Different compositions of components can be used to provide the same intended support. Components can be described according to different levels of granularity. At the highest granularity level, the system itself is seen as a large composite component, which corresponds to the system level. Composite components are then recursively decomposed into finer-grained components until only simple components are identified (see Chapter 2, section 2.1.1 for a definition of simple and composite components).

The object level is concerned with the internal behaviour description of simple components in terms of a number of related objects. Actually, the behaviour of a component does not have to be provided by objects, although object-oriented technology is generally recognised as the most convenient technique for developing components.

Figure 3-8 depicts the four concern levels of our component-based groupware design methodology. At the enterprise level, an oval represents an enterprise concept. At the component level, a grey circle represents a composite component, while a white circle represent a simple component. At the object level, a rectangle represents an object.

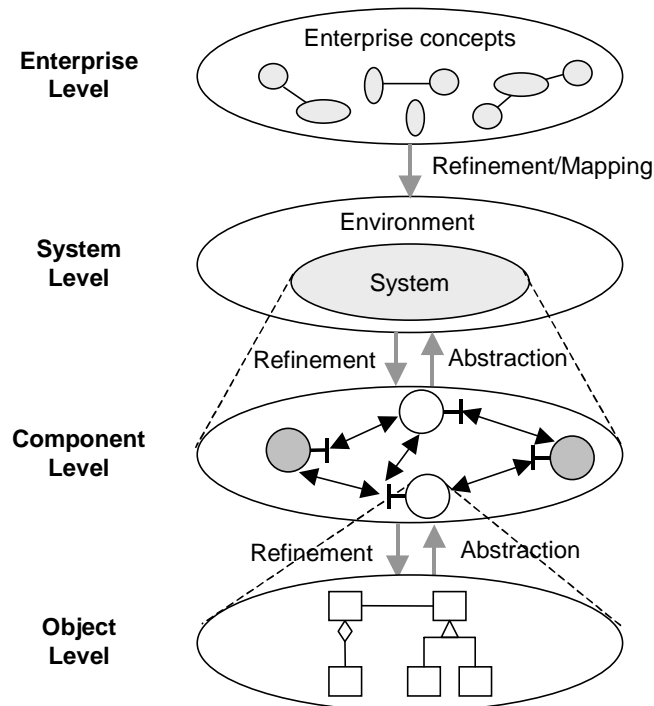


Figure 3-8: Concern levels of our component-based design methodology

The system, component and object levels are related to each other via simple refinement and abstraction relationships. However, the enterprise level is related to the system level through a combination of mapping and refinement relationships.

The relationship between the enterprise and system levels is a mapping because since the boundary between the system and its environment is not defined at the enterprise level. The concepts captured at the enterprise level have to be mapped to the context of the system, its environment or the boundary between both at the system level. A concept mapped to the boundary between a system and its environment indicates that this concept is related to both. For example, the concept may represent some common behaviour to be performed by the system in cooperation with its environment.

The relationship between the enterprise and system levels is also a refinement because abstract concepts can be refined into concrete concepts that make up the system description as a whole. Therefore, it may not be possible to abstract from a system level specification and obtain the original enterprise specification.

Figure 3-9 shows different possibilities for a transition from an enterprise level specification to a system level specification. A different system specification can be obtained based on different mappings/refinements of the concepts defined in the enterprise specification.

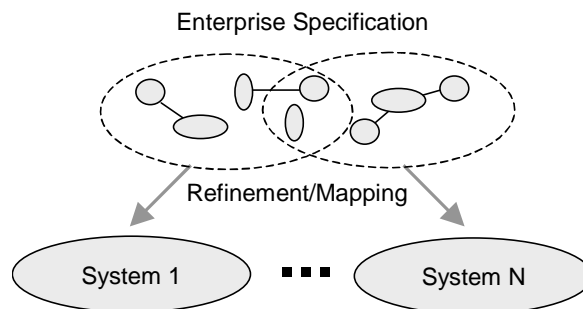


Figure 3-9: Different enterprise to system mappings/refinements

### 3.3.2 Methodology views

A behaviour description is of no practical use if it cannot be associated to an entity that either exists or will exist in the foreseeable future. So, we concentrate our research on modelling entities capable of carrying out behaviour, the so-called functional entities. In order to model a functional entity we have to delimit its behaviour to describe it properly. When we delimit some behaviour we are constraining the behaviour that is of concern to us.

In case we were interested in describing a closed or self-sufficient entity, we only would need to describe its structure and its internal behaviour. However, since the entities that of interest are open entities, i.e., entities that interact with other entities, interactional aspects should also be taken into account while describing a functional entity.

In this sense, besides structuring the development process according to concern levels, we also consider different views at each one of these levels. Each view focuses on a specific set of concerns that are common to all concern levels. We identify three basic views that capture the main set of concerns related to the architectural design of groupware systems:

- *structural view*, which is concerned with the logical delimitation (structure) of entities and their interconnection, i.e., how entities are logically interrelated;
- *behavioural view*, which is concerned with the behaviour of each entity in isolation, i.e., how a single entity behaves in time.
- *interactional view*, which is concerned with the cooperative behaviour of multiple entities, i.e., how the behaviour of two or more entities relates to each other. The behavioural and the interactional views can be seen as dual perspectives of the same aspect, viz., the behaviour of an entity, which complement each other.

Figure 3-10 illustrates how the different views span across the concern levels.

	Structural View	Behavioural View	Interactional View
Enterprise Level			
System Level			
Component Level			
Object Level			

Figure 3-10: Basic modelling views

Views are not limited to structural, behavioural and interactional, although these views are the most relevant ones for functional design. Other views, such as *information* and *testing*, may also be used.

The information view is concerned with the information needs of entities and how this information is structured. At the enterprise level, the information view is concerned with the information items produced or consumed in the enterprise. At the system level, the information view is concerned with the information associated with the system, i.e., information that is used, changed, created or destroyed by the system behaviour. At the component level, the information view is concerned with the associated with individual components. Finally, at the object level, the information view is concerned with the information associated with individual objects.

The testing view is concerned with the testing process of entities. At the enterprise level, the testing view is concerned with the description of general testing requirements. At the system level, the testing is concerned with the description of procedures to test the system as a whole. At the component level, the testing view is concerned with the description of procedures to test individual components. Finally, at the object level the testing view is concerned with the description of procedures to test individual objects.

The information and testing views are also called extended views because, although they can be used to capture valuable information regarding the development of a groupware system in general, they are not the most fundamental views for behavioural modelling. Therefore, we do not provide guidelines to create specifications based on these views in the context of this work.

### 3.4 Methodology scope

Our methodology focuses on the high-level architectural design of groupware systems. Our methodology partially supports the requirements analysis phase. We do not focus on user requirements capturing or gathering, but rather on the organisation and understanding of these requirements. Requirements organisation is accomplished at the enterprise level, although an enterprise specification may also serve other purposes.

Our methodology fully supports the specification phase and the architectural design sub-phase, which can be compared to the system and component levels of our design methodology, respectively. The detailed design sub-phase is similar to our object level, and since the

development process of a component at the object level corresponds to traditional object-oriented software development, this phase is outside the scope of this research.

The phases of implementation, test and integration, operation and maintenance and retirement are outside the context of our research. Therefore, we cover only some of the phases of the software development process presented in section 3.2.

Figure 3-11 indicates the scope of our design methodology with respect to the software development process phases of the waterfall model.

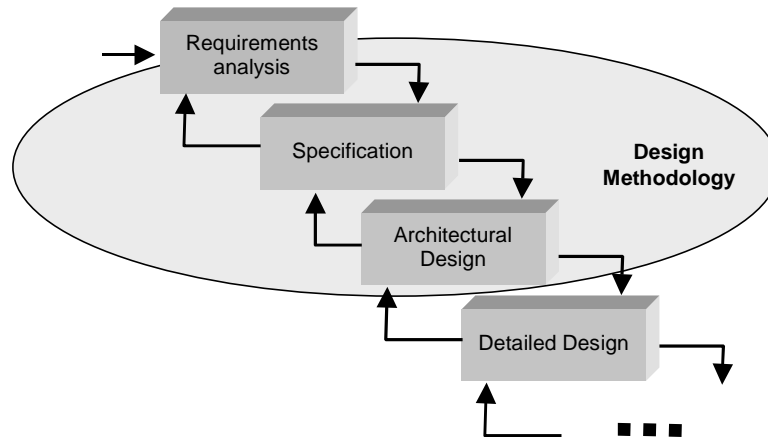


Figure 3-11: Scope of the design methodology

### 3.5 Related work

There are a number of software development methodologies available nowadays and a number of tools and environments that support these methodologies. Software development methodologies are usually organised around a modelling technique. General purpose modelling techniques can be used in different methodologies.

UML has been widely used in both academic and industrial settings to model software systems. This broad acceptance can be partially credited to its independence from a particular software development process. Since UML development has started, a number UML-based software development processes have emerged in the software development marketplace [JaBR99, DSWi99, Hrub98, ABL+00, AtBM00, ChDa00].

Below, we discuss three processes that are particularly interesting for our work due to their use of abstraction levels and views: Unified Process [JaBR99], Catalysis [DSWi99] and Pattern of Four Deliverables [Hrub98].

#### 3.5.1 Unified Process

The Unified Process [JaBR99] is a software development process developed hand in hand with UML. However, the basic principles of the Unified Process have been developed in the late 1980's, when I. Jacobson conceived the Unified Process predecessor, the so-called Objectory Process. Ever since, the Unified Process has evolved over the years until reaching its current form.

The Unified Process is use-case driven. A use case describes a piece of the user's functional requirements, while the set of all use cases describes all the user's requirements, and thus the complete functionality of the system. A series of design activities, which are derived from the use cases, drive the development process. Therefore, use cases are the basis for the system specification, design, implementation and test.

The Unified Process is architecture-centric. The Unified Process focuses on the development of the system architecture, i.e., the blueprint of the software system, since the early phases of development. On one hand, the development of the system architecture is influenced by the use cases. On the other hand, the system architecture influences the selection and allocation of use cases to 'system parts'.

The Unified Process is also interactive and incremental. A software system is developed incrementally in a number of cycles. Each cycle corresponds to a project, which ends with a new release of the software system, ready for delivery for its users. Each cycle of the Unified Process consists of four phases, viz., inception, elaboration, construction and transition, where each phase may be further subdivided into iterations.

Figure 3-12 illustrates the life cycle of a software system according to the Unified Process. A software system is developed incrementally in cycles. A cycle is divided in phases, which are iteratively developed.

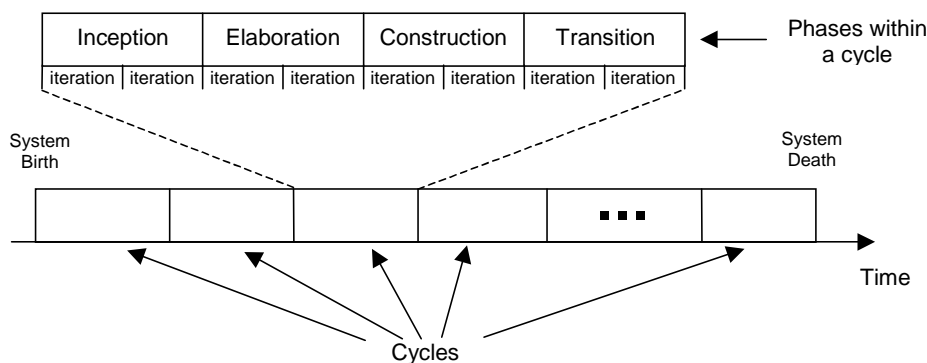


Figure 3-12: The life cycle of a software system according to the Unified Process

A number of models are produced within each cycle through a series of related activities, generally known as core workflows. There are five core workflows, viz., requirements, analysis, design, implementation and test. These workflows produce, respectively, the use-case model, the analysis model, the design and deployment models, the implementation model and the test model. These models are related to each other through trace dependencies.

Figure 3-13 depicts the Unified Process core workflows and their resulting models. In this figure, the dependencies between the use-case model and the other models are depicted as well.

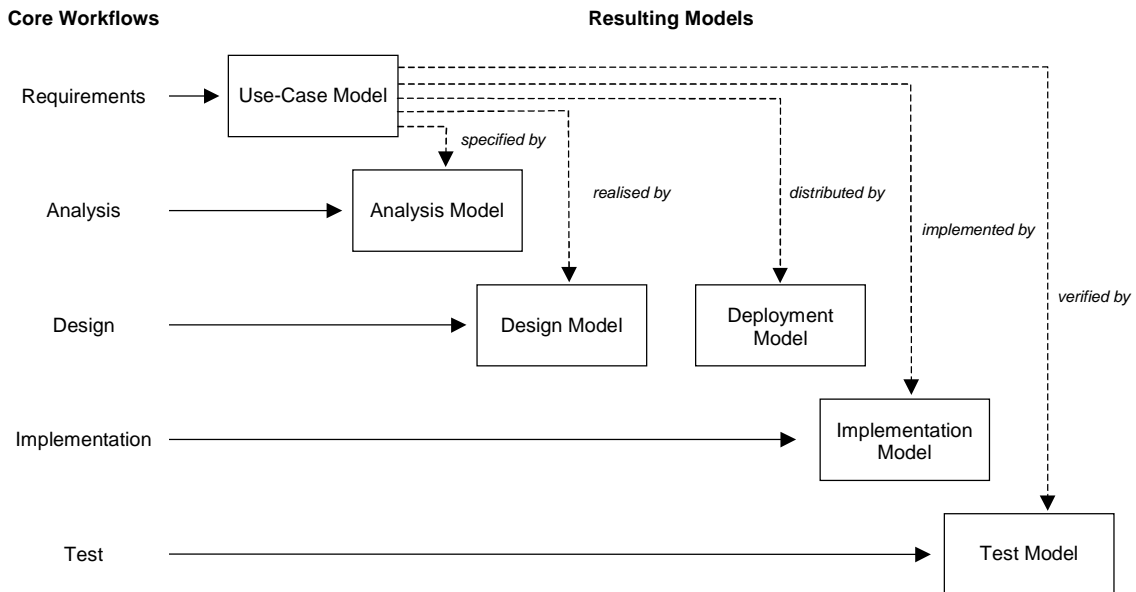


Figure 3-13: Core workflows and resulting models

### 3.5.2 Catalysis

Catalysis [DSWi99] is another software development process based on UML. Catalysis is based on three modelling concepts, viz., type, collaboration and refinement, and frameworks. A type specifies the external behaviour of an object. A collaboration specifies the behaviour of a group of objects, while a refinement relates different levels of behaviour description. Frameworks describe recurring patterns of these three modelling concepts.

Catalysis splits the development process in three abstraction levels: the domain/business, the component or system specification and the internal design of the component or system. The domain or business level is used to establish the problem domain terminology and to understand business processes, roles and collaborations. The component specification level describes the externally visible behaviour of components, the system or both, while the internal design level defines the internal architecture and behaviour the components, the system or both.

Figure 3-14 shows the abstraction levels adopted by Catalysis.

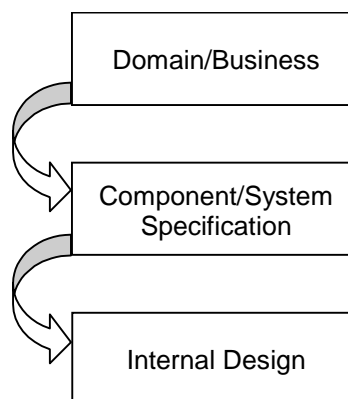


Figure 3-14: Catalysis abstraction levels

Similarly to the Unified Process, Catalysis does not have a unique sequence of activities. Different sequences of activities and deliverables can be used according to the project at hand. However, the development of a typical (large) business system involves the activities of requirement capturing, system specification, architectural design and component internal design. These activities are performed in accordance with the abstraction levels shown in Figure 3-14.

Figure 3-15 depicts the software development process according to Catalysis.

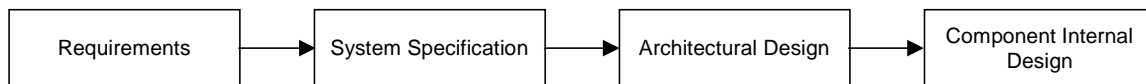


Figure 3-15: Software development according to Catalysis

### 3.5.3 Pattern of Four Deliverables

Pattern of Four Deliverables [Hrub98] is yet another software development process based on UML. According to this process, a software system is developed according to different abstraction levels and from several views. This process basically defines four levels of abstraction, viz., system, architectural, class and procedural, and four views, viz., use case, logical, component and deployment.

At each abstraction level and according to each view, the system can be described by a set of related design deliverables. A design deliverable consists of a piece of information about a software system. A deliverable has a representation, properties, responsibilities, attributes, methods and relationships to other deliverables.

Typically there are four types of deliverables: *classifier model*, which specifies static relationships between classifiers; *classifier interaction model*, which specifies interactions between classifiers; *classifier*, which specifies the classifier responsibilities, roles and static properties of classifier interfaces; and *classifier state model*, which specifies the classifier state machine and dynamic properties of classifier interfaces. A classifier represents a static entity in a system model. Typical UML classifiers are class, interface, use case, node, subsystem and component.

Figure 3-16 shows an example of the design deliverables used at the system level and according to logical view. The following design deliverables are identified: system model, system interaction model, system and system state model.

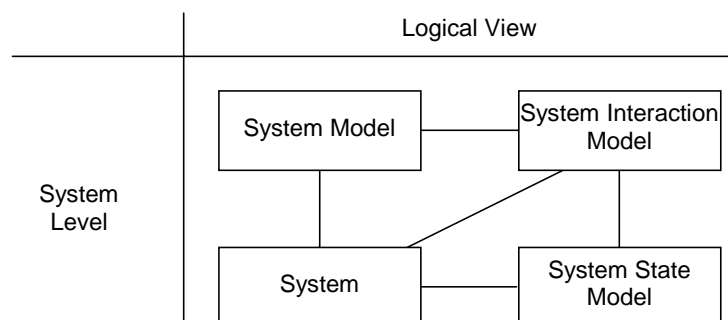


Figure 3-16: Design deliverables



---

# Chapter 4

## Enterprise Level Modelling

This chapter discusses the modelling of a system at the enterprise level. The chapter introduces some enterprise-related concepts and motivates the development of a cooperative work metamodel to serve as basis for the enterprise modelling of cooperative work processes. The chapter also describes and analyses a number of cooperative work models that have been proposed for the modelling of groupware systems, and proposes a new cooperative work metamodel based on a number of identified criteria. Finally, the chapter presents the techniques that can be used for modelling a system at the enterprise level and illustrates the use of some of these techniques by means of a simple case study.

The structure of the chapter is organised as follows: section 4.1 introduces some basic enterprise definitions and motivates the need for enterprise modelling; section 4.2 presents a number of cooperative work models; section 4.3 identifies the concepts that are relevant for the enterprise modelling of groupware systems and proposes a cooperative work metamodel; section 4.4 discusses the development of enterprise level specifications according to multiple abstraction levels; section 4.5 presents the techniques that can be used for enterprise modelling; section 4.6 illustrates the concepts, perspectives and techniques applied at the enterprise level by means of a simple case study involving a travel cost claim process; finally, section 4.7 draws some conclusions.

### 4.1 Enterprise systems and concepts

This section presents some basic definitions and explains the importance of enterprise modelling.

#### 4.1.1 Enterprise systems

An *enterprise* is typically a complex system that comprises groups of entities and is directed towards specific business goals or purposes. An *enterprise entity* is an individual person, group of persons or computer system, which is responsible for the execution of the processes and activities within the enterprise. The goals of an enterprise are achieved through the coordinated execution of activities by its entities using available resources. A resource is anything that is used, produced, or transformed by an activity.

The combined effect of the activities within the enterprise on its environment can be expressed as a service. The environment of an enterprise may be formed by other enterprises and/or stakeholders. While the other enterprises are interested in the service(s) the enterprise can provide, stakeholders are the people and/or organisations that benefit from the fulfilment of the enterprise goals.

Figure 4-1 depicts an enterprise as a goal-oriented system. Two-sided arrows represent common activities between the enterprise and its environment.

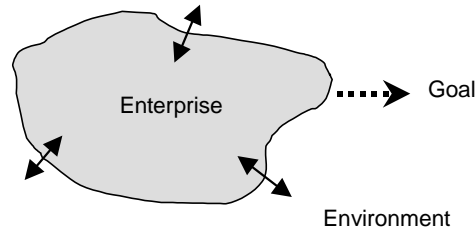


Figure 4-1: Enterprise as a goal-oriented system

An enterprise model is a simplified view of an enterprise. In general, the production of different enterprise models may serve different purposes [ErPe00, Ould95], such as to provide a better understanding of the key business mechanisms of an existing enterprise, to act as the basis for improving current business practices, to show the structure and experiment with innovative business practices and to identify outsourcing opportunities.

Most importantly in the context of this research, the elaboration of enterprise models serve as the basis for creating suitable telematics systems to support the enterprise. Once there is a model of the enterprise, one can decide on the requirements of the telematics systems, what functionality these systems should provide and how these systems should be developed. Multiple alternative solutions can be used in the design of technological support for an enterprise, such that different types of telematics systems can be derived to properly support the activities performed in an enterprise.

The main benefit of developing enterprise models to serve as basis for further development is that if we have a better understanding of an enterprise and its goals we can produce support systems that integrate easily with each other, such that information sharing and exchange can be accomplished efficiently.

Figure 4-2 illustrates different telematics systems that can be used to support the same enterprise. Each telematics system is represented as an oval. Subsystems of telematics systems are represented as circles, while behavioural dependencies between subsystems are represented as arrows connecting the circles.

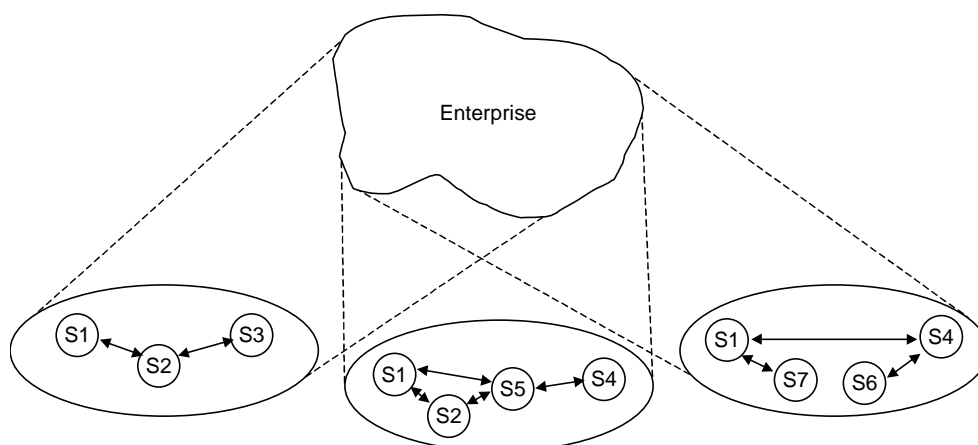


Figure 4-2: Telematics systems support to an enterprise

The elaboration of enterprise models for the purpose of creating a technological infrastructure can be approached in two different ways, viz., to serve as basis for the design of the whole infrastructure or to serve as basis for the design of a specific (collaborative) system. In the former case the whole enterprise has to be modelled, while in the latter case only part of the

enterprise directly related to the system being developed has to be modelled. In this work we only consider the latter case because we focus on the design of individual groupware systems. Although the techniques presented in this chapter can be extrapolated to cover both cases, such extrapolation is outside the scope of this research.

Figure 4-3 shows in grey a partial enterprise specification. This specification serves as the basis for the development of a specific application, S2. In this case, only the aspects related to the scope of S2, including its environment, are to be dealt with in the development of S2.

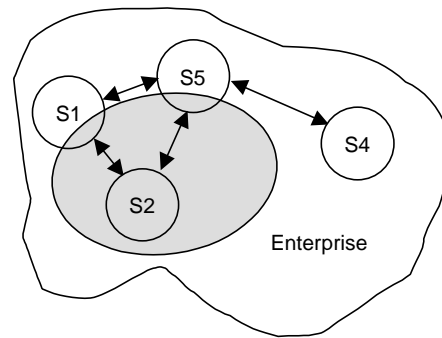


Figure 4-3: Partial enterprise specification

#### 4.1.2 Enterprise concepts

In order to produce models at the enterprise level we should know what concepts are relevant. We create enterprise models by identifying the relevant concepts and their relationships according to some objective.

There are many different types of concepts that can be used to model an enterprise. However, based on the purpose of the models, some concepts can be more relevant than others. In case we are interested in modelling the structure of an enterprise according to a human resources perspective, the concept of worker and organisation is more relevant than, for example, the concept of business goal. Still, all these concepts would be relevant in an overall model of the enterprise.

An *enterprise metamodel* is a model formed by a set of concepts and their relationships that can be used to produce enterprise models according to some objective.

Figure 4-4 illustrates an enterprise metamodel and an enterprise model. The enterprise model was created based on the set of concepts and their relationships defined in the enterprise metamodel. Concepts are represented by different geometrical shapes, while a relationship between concepts is represented by a line connecting two concepts.

For the purpose of developing telematics systems in general and groupware systems in particular, we have to devise a set of enterprise concepts that is relevant for this area. Still, the set of concepts that could be used may be changed to suit a specific type of target system. For example, suppose we can use the set of concepts formed by C1, C2, C3 and C4 to create enterprise models of groupware applications in general. Also suppose we are producing enterprise models of a specific groupware application, such as a videoconferencing system or a shared document repository. In this specific domain, we could have some other concepts, such as C5 and C6, which either specialises a previous concept, possibly replacing it, or is specific to this domain and should be taken into account as well.

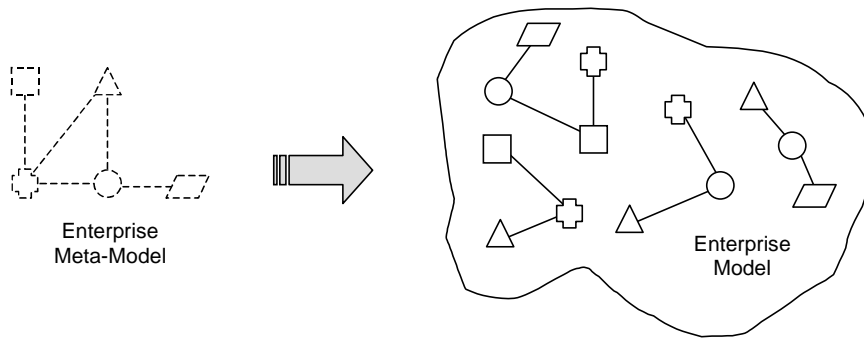


Figure 4-4: Enterprise metamodel and enterprise model

We define a *cooperative work metamodel* as an enterprise metamodel that focuses on the set of concepts related to the representation of cooperative work and their relationships. A cooperative work metamodel is used to produce enterprise models of a cooperative work context or process.

Two alternative approaches can be used to create a cooperative work metamodel:

- 1) to perform domain analysis on different groupware systems, such as shared whiteboards, chats, workflow management systems and co-authoring systems, in order to identify the concepts that are common to most of these systems and the relationships between these concepts, or;
- 2) to analyse a set of cooperative work (meta)models and extract the concepts that are common to most of these models and the relationships between these concepts.

The main difference between both approaches is the following: while groupware systems can be built according to the concepts present in a cooperative work model, domain analysis models are extracted from available groupware systems. The former approach would be particularly suitable if we were interested in capturing the concepts that concerns a specific domain. An example of this approach can be found in [SiJF99], where a conceptual model of a workflow process is presented. We did not adopt this approach since we did not want to restrict ourselves to one or more specific (sub)domains. We adopted the latter approach instead because we believe that the equivalent results could be achieved in a shorter period of time.

Figure 4-5 shows the relationship between both approaches.

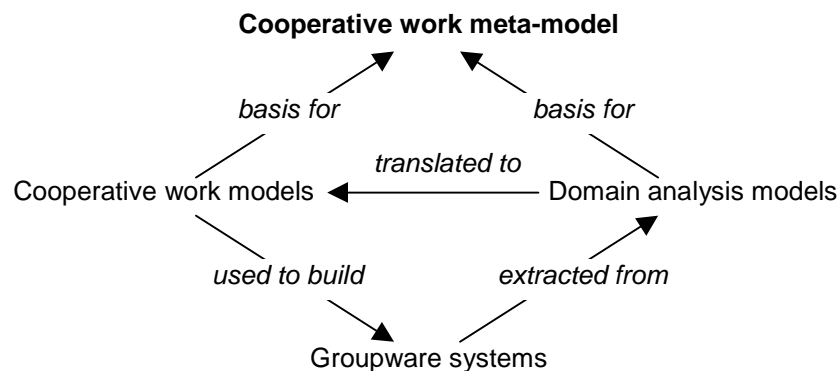


Figure 4-5: Alternative approaches to create a cooperative work metamodel

## 4.2 Cooperative work models

Cooperative work models have been successfully used to represent application domain knowledge in the development of groupware systems. We observed in the literature that many alternative models have been developed, such that we decided to study and compare these models to: (1) identify a number of general concepts present in these models; (2) help identifying a number of criteria that can be used to check whether or not the concepts present in a particular model are suitable for the abstract modelling of cooperative work, and; (3) obtain enough information for developing our own model in case no model is found adequate.

A cooperative work model can be originated from different sources. For example, a cooperative model can be: (1) the result of a particular or general understanding of cooperative work; (2) an interpretation of the cooperative work support provided by a groupware system; or (3) derived from a social view or theory of cooperative work.

In this work, we have considered all these sources for cooperative work models. The models we have selected are representative of these cases and the information contained in the models was sufficient to reach our objective.

In the following sections, we present for each model a brief informal description as found in the literature and a representation of its concepts and their relationships (according to our interpretation of the model). But first, we describe the approach used to capture the concepts present in a model.

### 4.2.1 Elaboration of cooperative work models

In order to systematically capture and analyse the concepts underlying a number of cooperative models, we develop concept diagrams for these models. A concept diagram consists of a UML class diagram in which a class represents a concept and an association between classes represents a relationship between their corresponding concepts. Such a high level representation is commonly called *conceptual perspective* in the literature [FoSc97].

The development of a concept diagram aims at capturing the concepts present in the application domain under investigation. Therefore, for each cooperative model presented in the following sections we develop one or more concept diagrams to help understanding its main concepts.

The steps towards the representation of a cooperative work model are the following:

- *identification of classes*. An object is a concept, an abstraction or thing with crisp boundaries and meaning for the problem at hand [RBP+91]. A class is a representation of a group of objects with similar structure (attributes), common behaviour (operations), common relationships to other objects (links), and common semantics. An object in a particular model can represent either a tangible entity, such as an actor, a document or a tool, or a non-tangible entity, such as an activity or a task. A class should be used to represent each distinct group of similar objects;
- *identification of associations between classes*. An association represents a group of links, usually between two classes, such that these links have a common structure and a common semantics. Any dependency between two or more classes is represented as an association;

- *identification of possible association classes.* An association class is an association with class properties, such as attributes and operations;
- *identification of the class attributes.* The identification of attributes is not mandatory in our approach. Attributes are identified only if they are important for the comprehension of a model.

Concept diagrams are mainly concerned with structural aspects of the information being captured, i.e., how concepts are structured and related to each other. Behavioural aspects are not relevant at this abstraction level. Consequently, we do not consider operations in our diagrams at this level.

## 4.2.2 Coordination theory

Coordination Theory consists of a set of principles that allows one to manage interdependencies between activities performed to achieve a goal [MaCr90]. This theory concentrates on coordination problems. Typical examples of coordination problems are the identification of goals, the mapping of goals to activities, the ordering of activities, the selection of actors to perform an activity, the management of interdependencies between activities and the allocation of resources for an activity.

Several disciplines provide different and complementary insights for understanding coordination. A conceptual framework to better understand and analyse coordination issues has been proposed in [MaCr90]. This framework consists of some concepts, viz. goals, activities, actors, resources and interdependencies.

One or more actors perform one or more activities, which are determined by some goals. Activities depend on each other and their interdependence can be characterised in terms of objects common to the activities. These common objects, such as shared resources, constrain how each activity is performed. Three generic kinds of interdependence are possible: (1) prerequisite, i.e., the output from an activity is required by another, (2) common resource, i.e., a resource is required by multiple activities, and (3) simultaneity, i.e., two or more activities must occur at the same time (time is the common resource between the activities).

Figure 4-6 presents our concept diagram for the Coordination Theory.

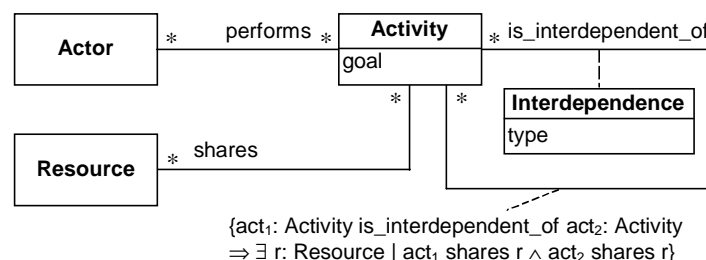


Figure 4-6: Coordination theory class diagram

The identification of the key concepts of the Coordination theory is straightforward. The concepts of actor, activity and resource are mapped onto the classes Actor, Activity and Resource, respectively. The concept of goal is mapped onto a class attribute, while the concept of interdependence is mapped onto an association class. In the associations between two classes rep-

resented in a class diagram, the multiplicity symbol ‘\*’ represents that zero or more instances of a class may be contained in a given association at a time.

A goal is represented as an attribute of an activity because a goal is only meaningful in the context of an activity and does not depend on other concepts. Similarly, the type of interdependence is represented as an attribute because it can be seen as a property of the association between two activities. Furthermore, one activity can only be interdependent of another if there is a resource that is shared by both activities (see constraint in Figure 4-6).

### 4.2.3 Activity Theory

Activity Theory [Kuut91, Nard96] is one of the most popular cooperative work theories in the CSCW community. According to this theory, an activity is the basic unit of analysis. An activity is a collective phenomenon that transforms a material object. An activity is realised through a series of actions, each action being carried out by a number of operations. The objective of an activity is accomplished through the transformation of the associate object towards some desired state.

An activity involves a community of participants. However, only part of this community understands the purpose of the activity, the so-called active participants. The active participants, also called active subjects, can be either individual or collective. The participants of an activity perform this activity through conscious and purposeful actions.

The theory also considers the cultural mediation of the relationships within an activity. Tools mediate the relationship between the active subject and the material object, rules mediate the relationship between the active subject and the others participants, and division of labour mediates the relationship between the object and the community.

Figure 4-7 depicts our concept diagram for the Activity Theory.

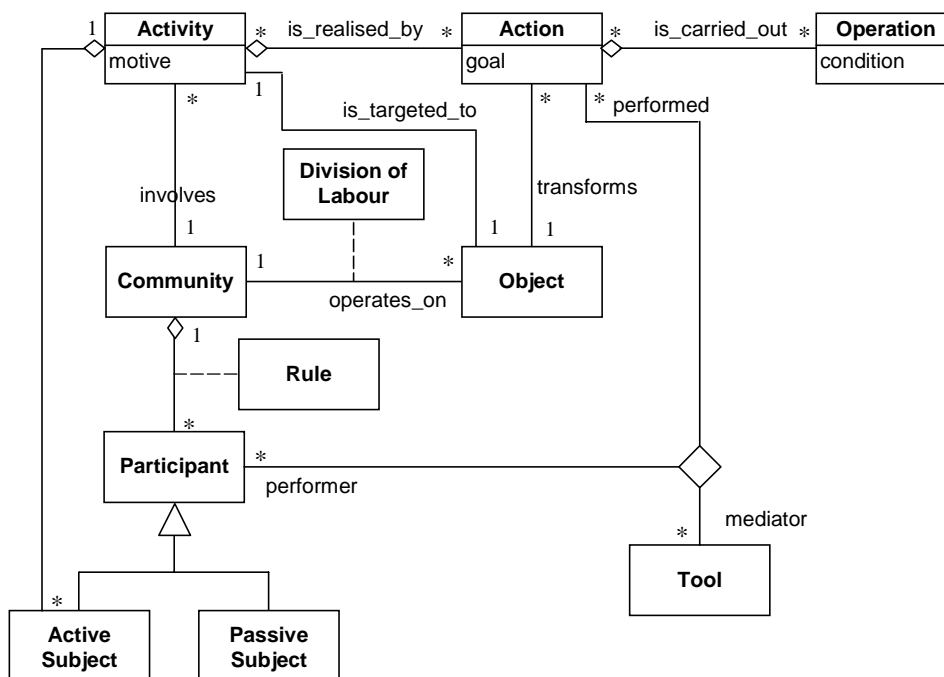


Figure 4-7: Activity Theory class diagram

The concepts of community, active subject, material object, activity, action, operation and tool have been identified in a first step and modelled as classes Community, Active Subject, Object, Activity, Action, Operation and Tool, respectively. The class Passive Subject has been defined to represent all the participants involved in an activity that are not active subjects. The classes Active Subject and Passive Subject have been generalised into the class Participant. Each instance of the class Participant is part of a Community. In the associations between two classes represented in a class diagram, the multiplicity symbol '1' represents that exactly one instance of a class participates in a given association at a time.

While analysing the Activity Theory we noticed that the relationship between the active subject and the activity was stronger than the relationship between the community and the activity. In this way, we model an Active Subject as part of an Activity, i.e., as an aggregation association, while a Community is only associated with an Activity. An aggregation association is graphically represented by a diamond. The community, by means of its participants, operates on the material object and realises an activity performing a set of purposeful actions, mediated by a tool. These actions transform the material object according to the motive of the activity. An action can be carried out by means of several non-conscious operations. The relationship between the classes Participant, Action and Tool is a ternary relationship represented by a hollow diamond attached to each one of these classes.

Rules regulate the relationship between the community and the active subject and, therefore, are modelled as the association class Rule, which is established between the classes Community and Participant. The division of labour mediates the relationship between a community and a material object. Thus, the division of labour is also modelled as an association class, Division of Labour, which is established between the classes Community and Object.

#### 4.2.4 Action/Interaction Theory

The Action/Interaction Theory has been the basis for the development of a groupware environment called WORLDS [FiTK95, FiKM96]. In particular, two aspects of this theory have been emphasised in the development of this system, viz. the notions of action and interaction and the notion of social world.

The concepts of action and interaction are the main concepts in this theory. Actions are embedded within courses of interaction and occur within the context of dynamic structural conditions. Courses of interaction are composed by sequences of connected actions. Structural conditions may either restrain or facilitate courses of interactions. A trajectory consists of a course of actions over time, and the actions and interactions that contribute to the evolution of the trajectory.

Actions are carried out by interactants. An interactant controls a trajectory and its course of interactions through their actions. An interactant can be either individual or collective. Articulation or alignment of actions is necessary when multiple interactants are involved.

Social worlds are interactive units that arise when a group of individuals acts in a collective way. A social world has activities, sites where activities occur and technology as means to carry out the activities. A social world can also have sub-worlds and individuals can be members of several social worlds simultaneously. The work undertaken by the members of a social world can be described using the action/interaction notions.

Figure 4-8 shows our concept diagram for the Action/Interaction Theory.

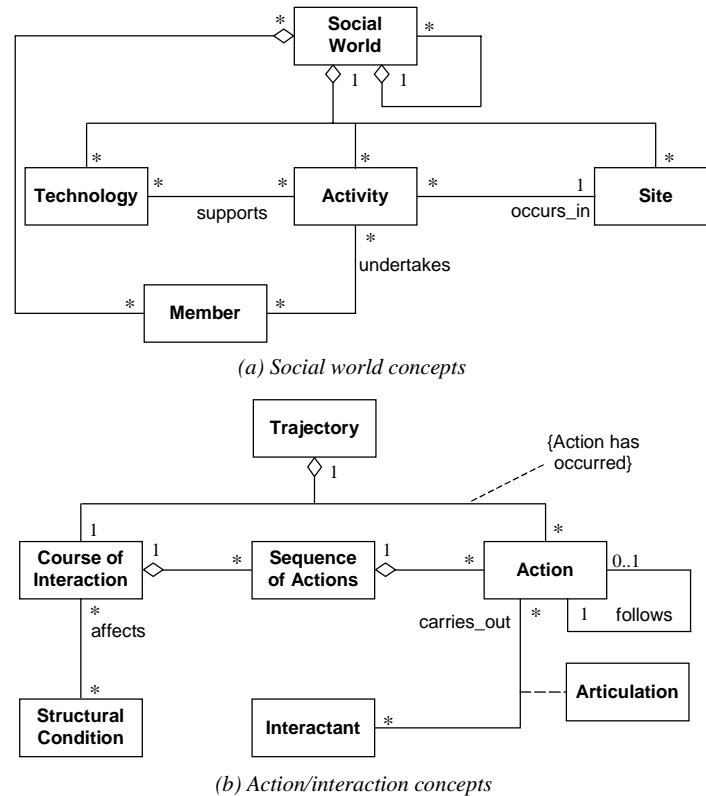


Figure 4-8: Action/Interaction theory class diagram

Due to the conciseness of the presentation of the main concepts of Action/Interaction Theory in the literature, we found it difficult to understand and precisely represent some relationships between concepts. So, for convenience purposes, we decided to split the object representation of the theory into two classes diagrams, viz., one describing the action/interaction concepts (see Figure 4-8b) and another representing the social world concepts (see Figure 4-8a).

Concerning the social world, we have identified the following concepts: social world, activity, technology, site and member. Each one of these concepts is represented as a class, viz., Social World, Activity, Technology, Site and Member, respectively.

Concerning actions and interactions, we have identified the following concepts: trajectory, course of interaction, structural condition, sequence of actions, action and interactant. Each one of these concepts was mapped onto a corresponding class, viz., Trajectory, Course of Interaction, Structural Condition, Sequence of Actions, Action and Interactant, respectively.

A Trajectory consists of a Course of Interaction and a set of Actions. However, only the actions that have been performed belong to a trajectory. This property is represented by a constraint in the association between Trajectory and Action. A Sequence of Actions consists of a set of Action objects connected to each other. The association follows represents the connection between two Action objects. This association implies that each Action may be associated with at most one other Action, which is represented by the '0..1' multiplicity in the class diagram.

Another important feature of this model is the need for articulation or alignment of actions when multiple interactants are involved. Because an action can be performed by several interactants, articulation is necessary among the actions that can be performed by a certain inter-

actant. This requirement is implied in the definition of the association class Articulation between the classes Interactant and Action.

#### 4.2.5 Task Manager

The Task Manager [KrHW93] is a system developed for the specification and management of cooperative work. Task Manager has been developed around the concept of task. People perform a common task by making use of shared documents or services, and communicate with each other by exchanging messages. A task can be seen from different points of view: (1) as a project, (2) as a set of subtasks with the establishment of dependencies between them or (3) as a folder, used only as a container of shared objects, such as subtasks, documents and messages.

The resources needed to achieve the goal of a task can be either computerised, such as documents, or non-computerised, such as rooms, budgets and machinery. The latter kind of resources is handled by services implemented outside the scope of Task Manager.

People can have different levels of participation in a task. Some people, called participants, have the same access rights to the attributes of a task, its documents and services and its messages, while other people, called observers, are only interested in the completion of a task. Observers only have read access to the information related to the task. A special type of participant, namely the one responsible for the execution of the task, has more access rights than the other participants. All the people involved in a task can exchange electronic mail messages within the context of the task.

Figure 4-9 shows our concept diagram for Task Manager.

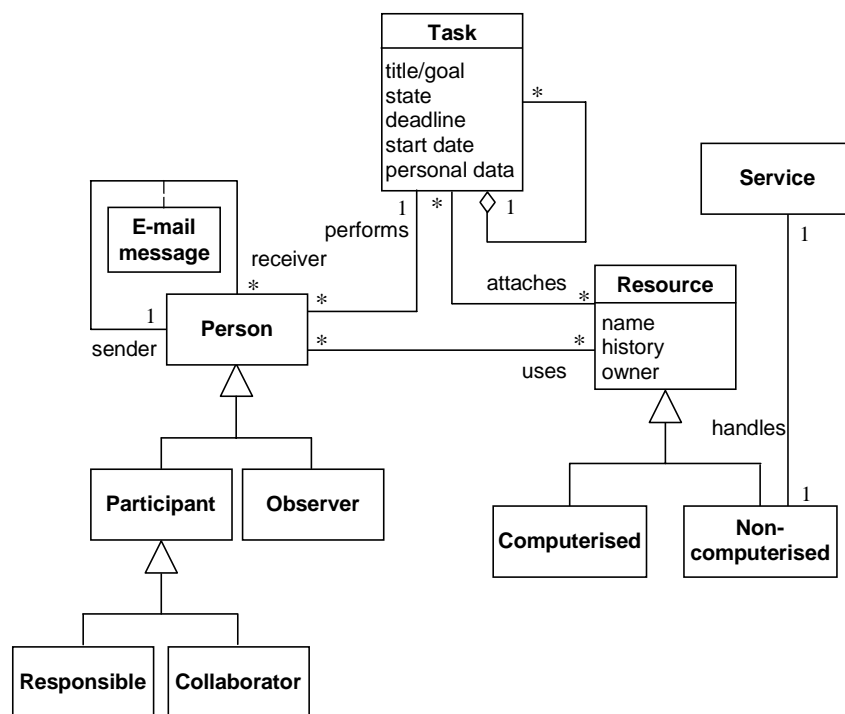


Figure 4-9: Task Manager class diagram

Within Task Manager, the main concepts we have identified are tasks, people and resources, which are represented by the classes Task, Person and Resource, respectively. A task may be

decomposed into multiple subtasks; this is represented by the aggregation association between the class Task and itself.

The class Person has been specialised into the classes Participant and Observer. The class Participant has been specialised into the classes Responsible and Collaborator. These classes represent all possible participation levels in a task. Messages are represented as an association class between persons with sender and receiver roles, respectively.

The class Resource has been specialised into the classes Computerised and Non-computerised resources. Non-computerised resources are handled by external services, represented by the class Service.

#### 4.2.6 Object-Oriented Activity Support

The Object-Oriented Activity Support Model (OOActSM) [Teeg96] aims at providing an integrated framework for groupware systems.

The concept of activity is the basic abstract concept of OOActSM. An activity is a structured object possibly containing an arbitrary number of sub-activities. An activity is executed by one actor, which can be a single person, a group of persons or a programmed autonomous agent in a computer.

An activity also has a context, representing elements created or manipulated by the activity (e.g., documents), elements used to perform an activity (e.g., tools or documents), other people involved in the activity (e.g., an organisation or a department) and information items required by the activity (e.g., goals or policies). The objects in the context of an activity can have a context themselves.

An activity has a state, an execution history and an execution procedure. The state and the operations of an activity determine what really happens.

Figure 4-10 depicts our concept diagram for OOActSM.

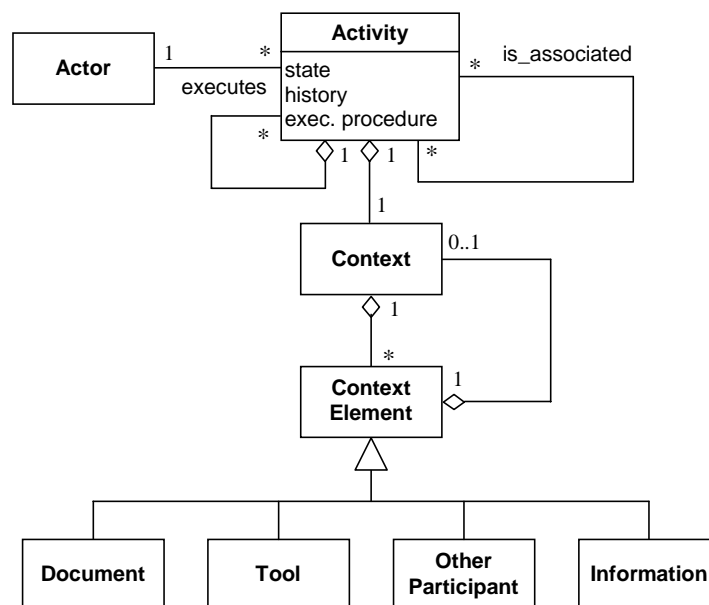


Figure 4-10: OOActSM class diagram

During the analysis of OOActSM we identified the concepts of activity, actor, context, document, tool, other participant and information, which are modelled as the classes Activity, Actor, Context, Document, Tool, Other Participant and Information, respectively.

An Activity object can aggregate many Activity objects, each one representing a sub-activity. Additionally, an Activity can be associated with multiple Activity objects through the association `is_associated`. Each Activity has one Context, which can include elements such as Document, Tool, Other Participant and Information. Each element in the context of an activity may have its own context. To fulfil these requirements we define the class Context Element, which generalises all the classes that can be in the context of an activity. We also define that multiple instances of Context Element can be contained in an instance of the class Context.

### 4.3 A cooperative work metamodel

#### 4.3.1 Informal view of cooperative work

Cooperative work takes place whenever two or more persons are engaged in the execution of a common task within a particular context. This common task defines a cooperative process. The persons engaged in a cooperative process not only work together, but also share a common goal or objective, which is often accomplished through the transformation, creation or manipulation of some common artefacts or resources.

A *cooperative process* defines some behaviour that should be carried out by the persons engaged in the process accordingly, so that the process goal or objective can be accomplished. In this sense, the execution of the associated behaviour should be coordinated such that the necessary work is properly done, i.e., work is done in a proper order, all the work is done and no redundant or conflicting work is done.

In case computer support is available for the (partial) execution of the cooperative process, computer supported cooperative work is achieved.

Figure 4-11 depicts our informal view of (computer supported) cooperative work.

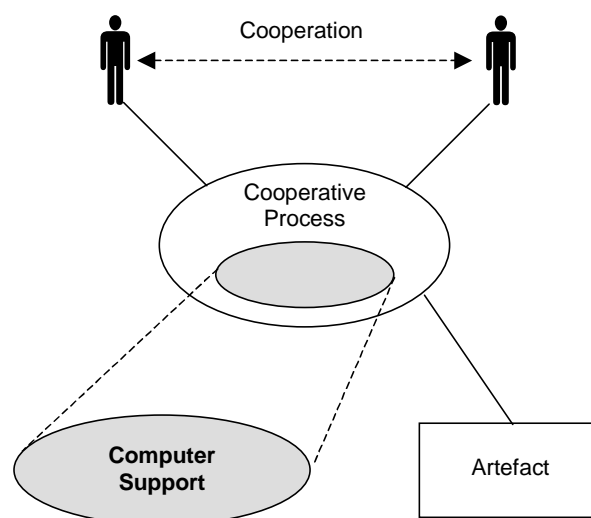


Figure 4-11: Informal view of cooperative work

### 4.3.2 Analysis criteria

In order to analyse the models presented in section 4.2, we need to establish some basic criteria. These criteria refer to the selection of an appropriate set of concepts to represent a cooperative work model and their relationships. These criteria were established based on the studies carried out on the cooperative work models and groupware systems (see Chapter 2) and our informal view of cooperative work.

Additionally, the selection of an appropriate set of concepts should adhere to the following *quality principles* [Sind95]:

- *consistency*, which requires that concepts should be consistent in their representation of the aspects in the real world;
- *orthogonality*, which requires that distinct concepts should be used to represent different aspects;
- *propriety*, which requires that concepts should be proper to the modelling needs;
- *generality*, which requires that concepts should be of general purpose in a given domain and the complete set of concepts be sufficient to cover the needs of the domain.

Therefore, the criteria used to analyse and evaluate the identified models are the following:

1. It should be possible to represent an entity capable of executing behaviour using a single concept. This entity may correspond to a person, group of persons or a computer system.
2. It should be possible to dissociate the behaviour executed by an entity from the entity itself, i.e., the same entity should (potentially) be able to execute multiple behaviours.
3. It should be possible to structure behaviour into behaviour units and relationships between these units.
4. It should be possible to represent anything both used in and established during the execution of a behaviour unit using a single concept.
5. It should be possible to represent behaviour according to different, but related, levels of abstraction.
6. It should be possible to represent the contribution of an entity in the execution of a behaviour unit, where different types of contribution may exist.

### 4.3.3 Analysis of cooperative models

Each aforementioned criterion can be independently satisfied ( $\checkmark$ ), partially satisfied (+/-) or not satisfied (-) by a particular model according to our interpretation. Further, in case not enough information about the model is available to make an assessment, the satisfaction of the criterion is undefined (?).

Table 4-1 summarises the analysis performed on all models. According to this analysis, no particular model completely satisfies all the proposed criteria. Particularly, no model managed to satisfy the second criterion because in general entities were directly associated with behavioural units. In the sequel, we analyse each model separately.

<b>Model \ Criterion</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>
<i>Coordination Theory</i>	√	-	+/-	+/-	?	-
<i>Activity Theory</i>	+/-	-	+/-	√	√	√
<i>Action/Interaction Theory</i>	+/-	-	?	-	?	√
<i>Task Manager</i>	+/-	-	+/-	√	√	+/-
<i>OOActSM</i>	+/-	-	√	+/-	√	+/-

Table 4-1: General criteria evaluation

### Coordination Theory

The Coordination Theory satisfies the first criterion because it uses a single concept to represent an entity capable of performing behaviour (the concept of actor). However, the second criterion is not satisfied because in this model there is not a concept representing the behaviour that may be performed by an actor, i.e., an actor is directly associated with behaviour units (the concept of activity). In this sense, the third criterion is partially satisfied based on the same reason, i.e., the Coordination Theory only has concepts representing behaviour units and the relationships between them.

The fourth criterion is partially satisfied because although this theory introduces the concept of resource, the use of this concept is only meaningful when activities share a common resource. The satisfaction of the fifth criterion is undefined because although the Coordination Theory considers only an activity as its behaviour unit, there is not enough information to say whether or not activities can be modelled according to different levels of abstraction. Specifically, it is unclear whether or not an activity can be refined into a number of more elementary activities. Finally, the sixth criterion is not satisfied because according to the Coordination Theory there is no possibility to distinguish between roles played by actors to perform activities.

### Activity Theory

The Activity Theory partially satisfies the first criterion because it uses two concepts in order to represent an entity capable of performing behaviour (the concepts of participant and tool). However, the second criterion is not satisfied because, similarly to the Coordination Theory, the Activity Theory does not have a concept representing the behaviour that may be performed by a participant/tool. The third criterion is partially satisfied because there is no explicit representation of the relationships between behaviour units, except for the different levels of granularity.

The fourth criterion is satisfied because the Activity Theory has a concept (the concept of object) that can be used to represent something used or rather transformed by behaviour units. The fifth criterion is satisfied because the different types of behavioural units define each a different abstraction or granularity level. Finally, the sixth criterion is also satisfied because the Activity Theory supports two different roles with respect to the execution of a behaviour unit (activity), viz., performer and mediator.

### Action/Interaction Theory

The Action/Interaction Theory partially satisfies the first criterion because it uses two concepts in order to represent an entity capable of performing behaviour (the concepts of technology and member/participant). The second criterion is not satisfied because this theory does

not have a concept representing the behaviour that may be performed by an entity. Actually, it is not clear how behaviour is structured into behaviour units (activities and actions). As far as we could understand the theory, behaviour can only be represented as sequences of actions, but the relationship between the concepts of activity and action is not clearly defined. As a consequence, the satisfaction of the third criterion is undefined.

The fourth criterion is not satisfied because this theory does not have a concept that can be used to represent something used by its behaviour units. The satisfaction of the fifth criterion is also undefined because although this theory considers actions as its primary behaviour unit, there is not enough information in the documentation on whether or not actions can be modelled according to different levels of abstraction. Finally, the sixth criterion is satisfied because the Action/Interaction Theory supports two different roles with respect to the execution of a behaviour unit, viz., an activity undertaker (role played by member) and an activity supporter (role played by technology).

### **Task Manager**

The Task Manager model partially satisfies the first criterion because although it uses a single concept in order to represent an entity capable of performing behaviour (the concept of person), this concept only includes human entities. The second criterion is not satisfied because this model does not have a concept to represent the behaviour that may be performed by an entity. The third criterion is partially satisfied because although the concept of task corresponds to a behavioural unit, there is no explicit representation of the relationships between tasks, except for the task/subtask relationship.

The fourth criterion is satisfied because the Task Manager model has a concept (the concept of resource) that can be used to represent something used by behaviour units in order to properly execute behaviour. The fifth criterion is satisfied because behaviour units can be represented according to different levels of abstraction or granularity. Finally, the sixth criterion is also partially satisfied because this model only supports different roles with respect to the execution of a task by a human entity, viz., participant and observer.

### **OOActSM**

The OOActSM model partially satisfies the first criterion because it has two concepts to represent an entity capable of performing behaviour (the concepts of actor and tool). The second criterion is not satisfied because similarly to the other models, the OOActSM model does not have a concept to represent the behaviour that may be performed by an entity. The third criterion is satisfied because this model has concepts representing behaviour units (the concept of activity) and the relationships between them.

The fourth criterion is partially satisfied because this model has two concepts (the concepts of document and information) that can be used to represent something created or manipulated by behaviour units. The fifth criterion is satisfied because behaviour units can be represented according to different levels of abstraction or granularity. Finally, the sixth criterion is also partially satisfied because this model only explicitly supports a single role with respect to the execution a behaviour unit, viz., executer (role played by actor).

#### 4.3.4 An enterprise metamodel for cooperative work

Because none of the models analysed completely satisfied our criteria, we propose a new metamodel. The main concepts of this metamodel are *functional entity*, *functional role*, *functional behaviour*, *activity unit* and *resource*.

A functional entity represents an entity capable of executing behaviour in the real world. A functional entity executes behaviour by itself or in cooperation with other functional entities. Therefore, a functional entity represents a, possibly future, telematics system or subsystem, or a person or group of persons in the real world.

A functional role describes a functional behaviour. A functional entity should play a certain functional role in order to execute the behaviour associated with the role. A functional role can be played by more than one functional entity at the same time. Sometimes a functional entity can play more than one functional role at the same time, while at other times the same functional entity may not be allowed to play different functional roles at the same time.

A functional role could in principle also be used to describe rights, permissions and responsibilities not captured in behaviour definitions. However, in the context of this work these issues are not relevant and therefore not described.

A functional behaviour is a (partial) description of some cooperative work process. A functional behaviour defines the behaviour that may be performed by a functional entity, playing the corresponding functional role. In the context of this work, a functional role is related to a functional behaviour on a one-to-one basis. A functional behaviour is defined in terms of activity units and relationships between these units.

An activity unit represents an atomic unit of work or behaviour within a cooperative work process at a given abstraction level. An activity unit can be executed entirely by a single functional entity or by two or more functional entities in cooperation.

The execution of an activity produces some result that can be used, changed or destroyed by other activities. A resource, also called *artifact*, represents anything used, created, destroyed or changed by an activity. There is no behaviour associated with a resource.

A relationship can be defined between two or more activity units. A relationship is used to constrain the occurrence or execution of the related activity units based on the occurrence or not of other activity units. We do not constrain the possible types of relationships that can be established between activity units. However, we suggest a desirable set of relationships (see section 4.3.5).

Figure 4-12 shows a conceptual representation of our cooperative work metamodel.

A functional entity should play a functional role in order to perform an instance of the activity units defined by the corresponding functional behaviour. Because a functional role describes a single functional behaviour and a functional behaviour is described by a single functional role, these concepts actually represent the same thing in this specific use of these concepts.

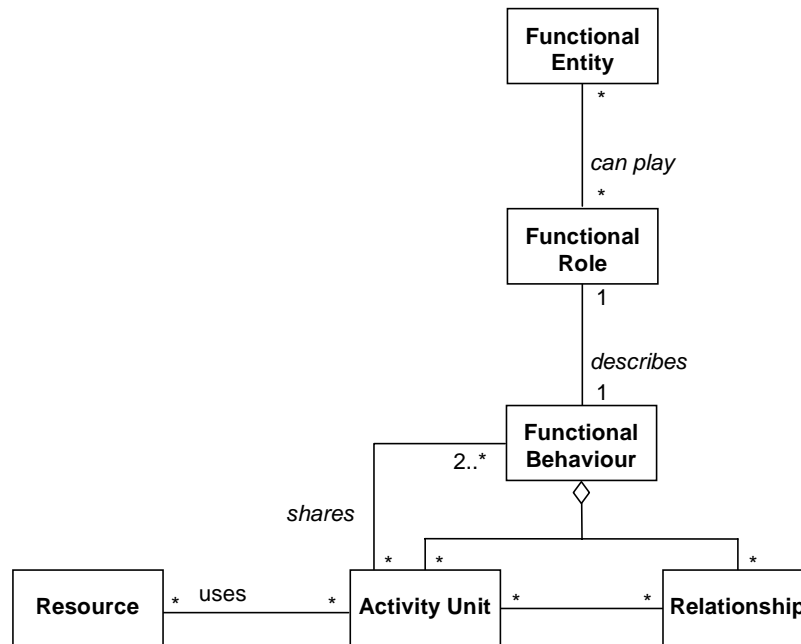


Figure 4-12: A cooperative work metamodel

Within a functional role/behaviour, activity units and relationships between these units are defined. Two or more functional behaviours may share an activity unit provided that the activity unit is defined in these functional behaviours. A shared activity unit represents an activity unit whose execution is carried out in cooperation by all functional behaviours that share the activity unit. However, in the context of this work we do not distinguish individual interaction contributions to the execution of an activity unit at the enterprise level, i.e., a shared activity unit is only seen as an integrated activity. Finally, the execution of an activity unit may require the use of some (common) resource.

#### 4.3.5 Relationships between activity units

Different types of relationships can be established between two or more activity units. In the sequel we present some basic types of relationships, called causality relations. The motivation and reasoning behind the identification of these different types of causality relations can be found elsewhere [Ferre94, Quar98, Sind95, VFQ+00, Wege98, FeFa01]. These different types of causality relations were chosen because they have been proved useful to describe the relationship between behaviour units of distributed system in general.

Basically, the occurrence of an activity unit may depend on the occurrence or non-occurrence of other activity units and on the results established by the occurrence of preceding activity units. Considering only the occurrence and non-occurrence of other activity units, we have the following basic types of causality relations between two activity units  $a$  and  $b$ :

- *enabling* ( $a$  enables  $b$ ), in which the occurrence of activity unit  $a$  enables the occurrence of activity unit  $b$ , i.e., activity unit  $b$  is allowed to occur provided that activity unit  $a$  has occurred;
- *disabling* ( $a$  disables  $b$ ), in which the occurrence of activity unit  $a$  disables the occurrence of activity unit  $b$ , provided that  $b$  has not occurred yet nor  $b$  occurs at the same time as  $a$ ;

- synchronisation (*a synchronises with b*), in which the occurrence of activity unit *a* depends on the occurrence of activity unit *b* (and vice-versa), such that both activity units must occur at the same time.

Further, we could also consider the absence of a causality relation between activity units as a special case of relationship in which the occurrence of activity unit *a* does not depend on the occurrence or non-occurrence of activity unit *b* (and vice-versa). In other words, *a* and *b* are both enabled and do not constrain each other's occurrence. Such special kind of relationship can be used to represent concurrency.

Two or more basic causality relations can be composed using boolean operators to create more elaborated types of relationship, such as non-deterministic choice based on the execution of activity units. A non-deterministic choice between two activity units *a* and *b* (*a disables b* and *b disables a*) models a relationship in which the occurrence of activity unit *a* disables the occurrence of activity unit *b*, provided that *b* has not occurred yet (and vice-versa) and both do not occur simultaneously.

#### 4.3.6 Coordination issues

Coordination is usually needed whenever multiple functional entities are involved in the execution of a number of common activity units. Since the execution of an activity unit is only possible in our metamodel whenever a functional entity plays a functional role where the activity unit is defined, coordination is in one way or another related to a functional role.

A description of an activity unit defined in a functional role is actually a description of (possibly) an infinite number of activity units that can be performed, all sharing some common characteristics. Thus, when a functional entity plays a role, this entity is actually able to execute instances of the activity units defined in the corresponding functional role.

Coordination between the instances of activity units that can be performed by a functional entity playing a certain role is determined by the relationships established between the activity units themselves. Since these relationships are defined as part of the functional role, the need for coordination is more notorious whenever two or more functional entities can play the same functional role at the same time.

Consider, for example, the enterprise specification illustrated in Figure 4-13. The functional entities E1 and E3 can both play the functional role FR1 at the same time, while the functional entity E2 can only play the functional role FR2. The functional role FR1 defines only the activity unit I1, which is a two-party activity unit shared by the functional role FR2. Whenever I1 is executed by either E1 in cooperation with E2 or E3 in cooperation with E2, an instance of the activity described by I1 is executed. Thus we have two instances of I1, potentially executed at the same time.

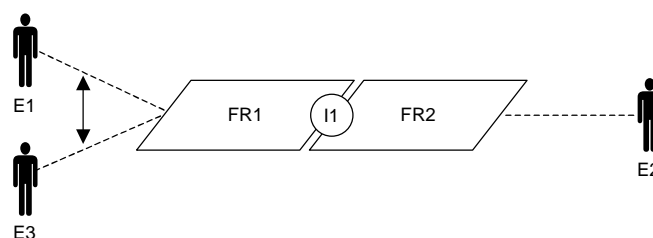


Figure 4-13: Coordination between functional entities playing the same role

In case I1 represents a shared activity unit whose concurrent execution would generate conflicting results for the functional entity E2, any realisation of the enterprise specification illustrated in Figure 4-13 should guarantee that the instances of I1 are distinct, i.e., they do not interfere with each other.

Different solutions can be used to solve such problems. For example, the functional entities E1 and E3 can informally communicate with each other and agree on whom should perform the activity unit first (informal coordination). The functional entity E2 could use a lock mechanism to prevent the potential concurrent execution of I1. Additionally, some policy involving for example the use of a floor control mechanism could be applied to decide on which functional entity should be able to perform I1 at a time. In all these cases, we should indicate by means of constraints on the association between the functional role in which the activity unit is defined and the activity unit itself that some mechanism should be provided at some later development stage.

#### **4.3.7 Compliance to proposed criteria**

If we compare the metamodel proposed in section 4.3.4 with the criteria enumerated in section 4.3.2, we observe that the first criterion is satisfied through the concept of functional entity, the second criterion is satisfied through the concept of functional role, the third criterion is satisfied through the concepts of functional behaviour, activity unit and relationship and the fourth criterion is satisfied through the concept of resource.

Further, although not addressed so far, the activity units of a functional behaviour can be modelled according to different abstraction levels and the execution of an activity unit can be carried out according to different role types (both issues are discussed in section 4.4). Therefore, the proposed metamodel satisfies all the identified criteria.

### **4.4 Enterprise perspectives and abstraction levels**

Enterprise level specifications can be developed according to different abstraction levels. Starting with an abstract specification, we can refine this specification to produce a more concrete and detailed specification, using the same set of concepts or specialisations of these concepts.

The refinement/mapping onto the system level can be carried out based on any given enterprise level specification. The specification used as basis for developing a system level specification is called *reference specification*. On one hand, the more abstract the reference specification, the bigger the gap that must be bridged towards producing a system level specification. Therefore, a more detailed reference specification would be preferred to help in this transition. On the other hand, the main drawback of developing detailed enterprise specifications lies in the risk of doing a lot of work that will not be used after all. In this sense a proper balance must be found.

Figure 4-14 illustrates the development of enterprise level specifications according to different abstraction levels. Each enterprise level specification corresponds to a different abstraction level.

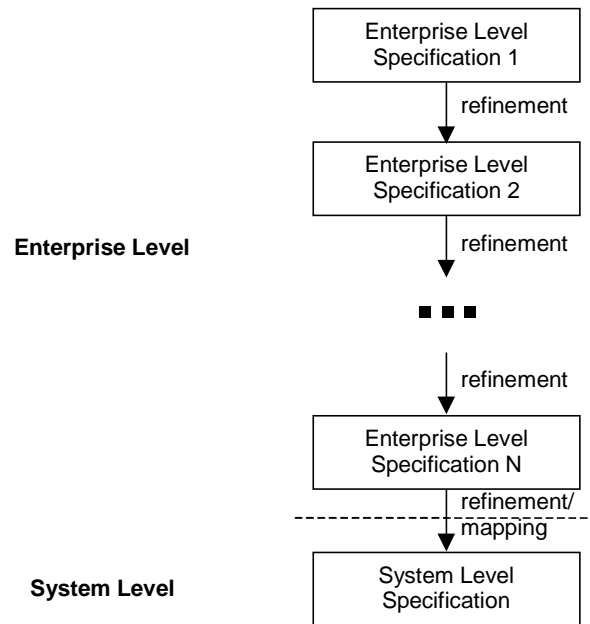


Figure 4-14: Enterprise abstraction levels

#### 4.4.1 Enterprise perspectives

Each enterprise level specification should be developed according to an enterprise perspective. We identify three different perspectives at the enterprise level: *integrated*, *distributed* and *distributed with role-discrimination*. Each of these perspectives can be seen as a subset of the concerns addressed at the enterprise level. Additionally, these perspectives are hierarchically related; a more concrete perspective will consider a super set of the concerns considered by a more abstract perspective.

##### Integrated perspective

The integrated perspective provides an integrated view of the enterprise and its functional behaviour as a whole. According to this perspective, the enterprise is usually seen as a single functional entity type that abstractly represents all the functional entities that will be part of the enterprise in a lower abstraction level. The functional entity representing the enterprise can play a single functional role describing the abstract functional behaviour of the enterprise. Alternatively, it can be permitted to associate a single functional behaviour to multiple functional entities, in case, for example, we were interested in describing the service provided by the enterprise to its environment.

The integrated perspective identifies a single type of activity unit, namely an action. An *action* models an activity unit whose execution is carried out exclusively by a single functional entity [QFS+97, VFQ+00]. One of the main characteristics of an activity in general and an action in particular is atomicity. An action is indivisible at the given abstraction level. Further, we abstract from the action duration, i.e., we only consider the moment of time at which the action is finished. Therefore, an action has either occurred and we can refer to its occurrence, or it has not occurred and we cannot refer to its occurrence. Actions have no intermediate steps or states that can be referred to.

Figure 4-15 depicts the specialisation of our cooperative work metamodel according to the integrated perspective. For conciseness purposes, we concentrate only on the concepts of

functional entity, function role, activity unit (action) and relationship, and we do not show the concepts of functional behaviour and resource.

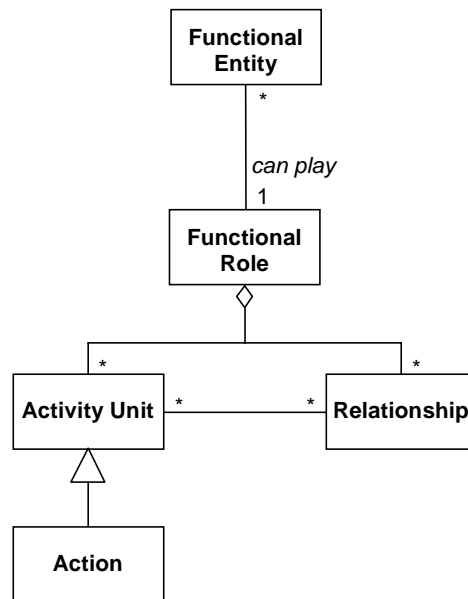


Figure 4-15: Specialisation of metamodel according to integrated perspective

The functional entity representing the enterprise can play a single functional role describing the functional behaviour under consideration. This behaviour is defined in terms of actions and relationships between these actions. The execution of an action is usually carried out by a single functional entity, playing the functional role in which the action is defined. However, two or more functional entities are also entitled to execute a single functional role describing their combined abstract behaviours.

### Distributed perspective

The distributed perspective provides a distributed view of the enterprise and its functional behaviour. According to this perspective, the enterprise is seen as a set of multiple functional entities capable of executing parts of the functional role or behaviour under consideration in isolation or in cooperation with one another.

The distributed perspective distinguishes not only an action but also an *interaction* as a possible unit of behaviour. An *interaction* models an activity unit whose execution is carried out by two or more functional entities in cooperation [QFS+97, VFQ+00]. In the context of this work we do not distinguish individual contributions of the functional entities to the execution of an interaction at the enterprise level, i.e., an interaction is seen as an integrated action involving two or more functional entities. Therefore, similarly to the concept of action, an interaction either occurs or does not occur. If the interaction occurs, all the entities involved in the interaction can refer to its occurrence. Otherwise, none of the involved entities can make such a reference.

Figure 4-16 shows the specialisation of our cooperative work metamodel according to the distributed perspective. Once more, for conciseness purposes, we concentrate only on the concepts of functional entity, function role, activity unit (action and interaction) and relationship, and we do not show the concepts of functional behaviour and resource.

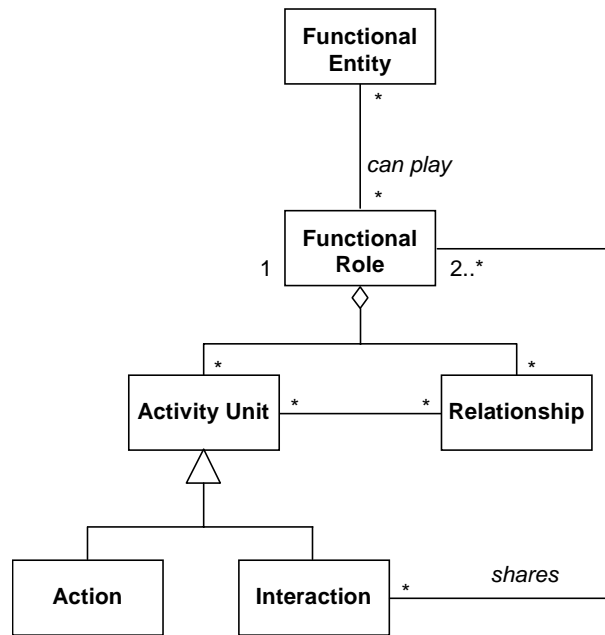


Figure 4-16: Specialisation of metamodel according to distributed perspective

Different functional entities can play different functional roles describing parts of the functional behaviour under consideration. Each functional role/behaviour is defined in terms of actions and (shared) interactions and relationships between them. The execution of an action is carried out exclusively by a single functional entity, playing the functional role in which the action is defined, whereas the execution of an interaction is carried out by two or more functional entities in cooperation, playing the functional roles that share the interaction.

### Distributed perspective with role discrimination

The concept of interaction as introduced in the previous section is generic because an interaction may involve the participation of more than two entities, which is regarded as multi-party interaction. However, as the design trajectory at the enterprise level leads to the system level we specialise an interaction to the participation of two entities (two-party interaction). This specialisation is needed because the realisation of multi-party interactions using current software technologies is less straightforward, often resulting in less efficient systems, than the realisation of two-party interactions. Usually, a construct involving multiple two-party interactions is needed to “implement” a multi-party interaction. A more detailed discussion on the needs for the specialisation of the concept of interaction at the system level is presented in Chapter 5.

Besides considering only two-party interactions, the development of a specification according to the distributed perspective with role discrimination also aims at describing the part or role type taken by a functional behaviour and consequently functional role in the execution of an interaction. This is the only type of “interaction contribution”, which defines the responsibility for the execution of the interaction, identified at the enterprise level in the context of this work. We distinguish between two different and complementary parts or role types, viz., *actor* and *reactor*.

The role of *actor* indicates that the functional entity playing the corresponding functional behaviour where the associated interaction is defined is ultimately responsible for the execution of the interaction, such that it can be considered the initiator of the interaction.

The role of *reactor* indicates that the functional entity playing the corresponding functional behaviour where the associated interaction is defined supports the execution of the interaction, such that it can be considered a contributor to the execution of the interaction that is initiated by another entity. Therefore, this entity should be willing to execute the interaction in the first place. Any two entities performing an interaction should always participate in its execution as an actor and as a reactor, respectively.

Figure 4-17 shows the specialisation of our cooperative work metamodel according to the distributed perspective with role discrimination. Again we do not show the concepts of functional behaviour and resource for simplification purposes.

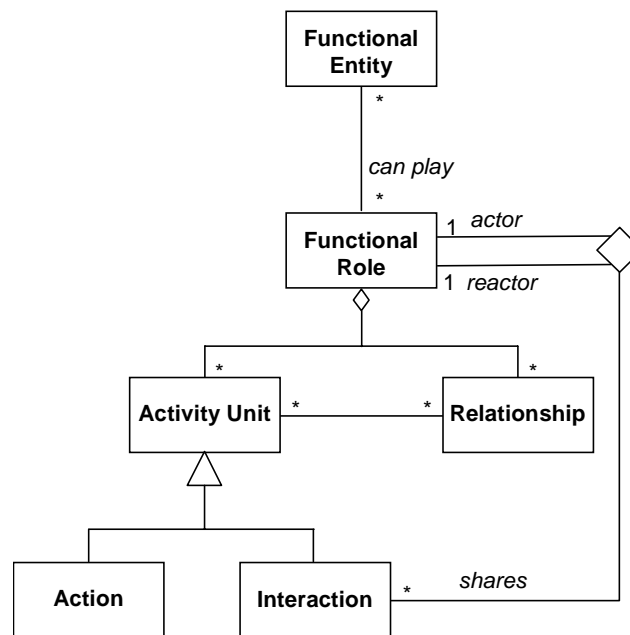


Figure 4-17: Specialisation of metamodel with role-discrimination

Different functional entities can play multiple functional roles and consequently perform the functional behaviour associated with these roles. A functional entity playing a functional behaviour may play different role types with respect to different interactions defined within the same functional behaviour. Each functional behaviour is defined in terms of actions, interactions and relationships between them. However, an interaction is shared by exactly two functional behaviours, each one of them acting either as actor or reactor in the execution of the interaction.

### Summary of enterprise perspectives

An enterprise level specification should be developed according to any of the abovementioned perspectives. The choice for a particular perspective depends on how abstract the intended enterprise specification should be, which in turn may depend on the starting point of the design process, e.g., complete (re-)design or modifications of existing enterprise specifications. However, in a design trajectory specifications are usually developed in sequence according to each of the perspectives.

The integrated perspective is best suited to develop high-level enterprise specifications, because it abstracts from the functional entities and functional roles of the enterprise, considering the behaviour of the enterprise as a whole.

The distributed perspective is best suited to develop more concrete enterprise specifications, because it considers an enterprise as a set of functional entities and functional behaviours that can be played or performed by these entities. Further, the distributed perspective also takes into account behaviour that is shared by multiple functional roles and consequently should be executed in cooperation by the enterprise entities.

The distributed perspective with role discrimination is best suited to develop low-level enterprise specifications, because it takes into account the needs of system level, i.e., a unit of common behaviour is shared by two functional roles, each one playing a different role with respect to the execution of this behaviour.

Figure 4-18 illustrates three different enterprise specifications developed according to each of our enterprise perspectives. A functional entity is represented by a human icon, whereas a functional role/behaviour is represented by a parallelogram. Functional entities are associated with functional roles via a dashed line connecting a human icon to a parallelogram. An action is represented by a circle comprised within the boundaries of a functional role, while an interaction is represented by a circle crossing the boundaries of two or more functional roles. The roles of actor and reactor are represented by respectively an *a* and an *r* attached to each of the functional roles that share an interaction. Finally, a relationship between two activity units is represented by a solid line or curve connecting two circles.

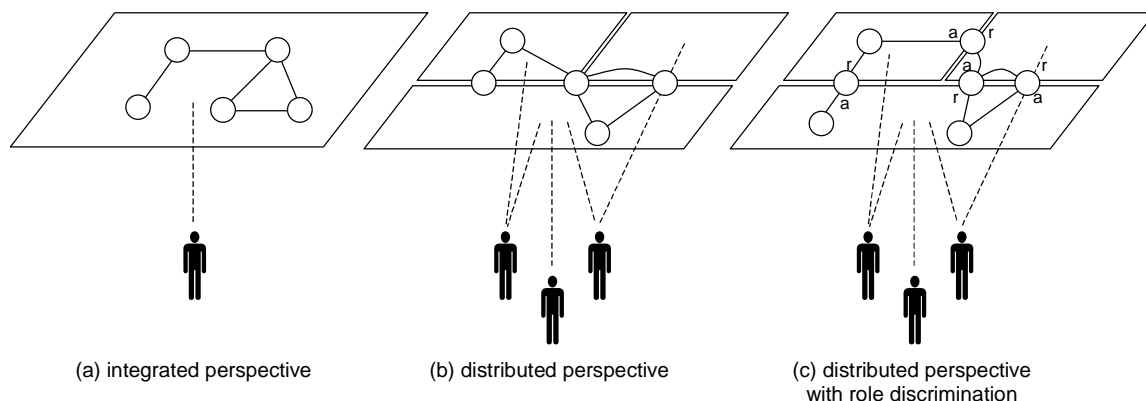


Figure 4-18: Enterprise perspectives illustrated

Figure 4-18a shows a specification developed according to the integrated perspective and it is the most abstract specification. Figure 4-18b shows a refinement of the Figure 4-18a specification, developed according to the distributed perspective, whereas Figure 4-18c shows a refinement of the Figure 4-18b specification, developed according to the distributed perspective with role discrimination.

#### 4.4.2 Refinement guidelines

This section presents some practical guidelines for behavioural refinement in the context of this work. Some of these guidelines refer to basic refinement types, while others can be seen as a combination of the basic refinement types presented in [Quar98]. We do not discuss conformance assessment now, but rather we defer this discussion to Chapter 6.

Before presenting these guidelines, we need to introduce some basic terminology. We call the functional role being refined abstract functional role, whereas we call the resulting functional

role(s) concrete functional role. We also call the activity unit being refined abstract activity unit, whereas we call the resulting activity unit(s) concrete activity unit.

### Functional role refinement

We identify two different types of functional role refinement:

- an abstract functional role can be refined through its replacement with two or more concrete functional roles with some common behaviour (interaction). The concrete functional roles comprise the abstract functional role and a common unit of behaviour is identified between the concrete functional roles. This type of refinement is only applicable whenever the functional entities that can play the resulting concrete functional roles at a given time are different, i.e., the same functional entity cannot play the resulting concrete functional roles at the same time, otherwise there would be no interaction. This type of refinement can also be seen as one of the types of action refinement (see section Action refinement below);
- an abstract functional role can be refined through its replacement with two or more concrete functional roles without some common behaviour. The concrete functional roles comprise the abstract functional role but no common units of behaviour are identified between the concrete functional roles. This type of refinement is only applicable whenever the functional entities that can play the resulting concrete functional roles at a given time are the same.

Figure 4-19 the different types of action refinement. A functional role is represented by a parallelogram. An action is represented by a circle comprised within the boundaries of a functional role, while an interaction is represented by a circle crossing the boundaries of two functional roles.

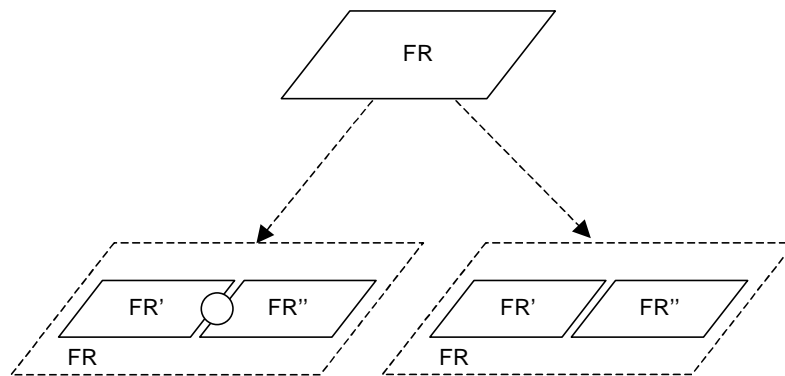


Figure 4-19: Functional role refinement

### Action refinement

We identify two different types of action refinement:

- an abstract action can be refined into a number of related concrete actions. Thus, refinement takes place within the same functional role;
- an abstract action can be refined into a concrete interaction, shared by two or more concrete functional roles. Thus, not only the abstract action is refined but also the functional role where the abstract action is defined is refined as well (see section Functional role refinement above).

Figure 4-20 depicts the different types of action refinement. A functional role is represented by a parallelogram. An action is represented by a circle comprised within the boundaries of a functional role, while an interaction is represented by a circle crossing the boundaries of two functional roles. A relationship between two actions is represented by a solid line connecting these actions.

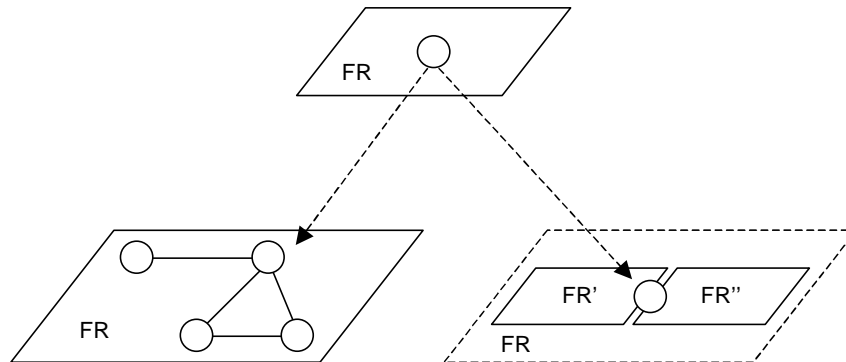


Figure 4-20: Action refinement

### Interaction refinement

We identify two different types of interaction refinement:

- an abstract interaction can be refined into two or more, possibly related, concrete interactions shared by the same functional roles. This type of refinement is used to detail the structure of an activity unit modelled as an interaction;
- an abstract interaction can be refined into two or more related concrete interactions such that the functional roles that share a concrete interaction is a proper subset of the functional roles that share the abstract interaction and the union of all the subsets containing the functional roles that share the concrete interactions forms the set of functional roles that share the abstract interaction. This type of refinement is used, for example, to refine a three-party interaction into two or more two-party interactions and their relationship.

Figure 4-21 depicts the different types of interaction refinement. A functional role is represented by a parallelogram. An interaction is represented by a circle crossing the boundaries of two or more functional roles. A relationship between two interactions is represented by a solid line or curve connecting these interactions.

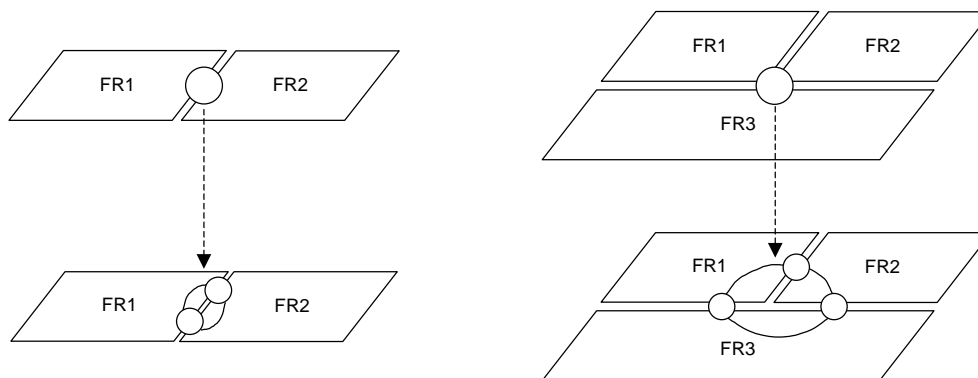


Figure 4-21: Interaction refinement

In case the role a functional role has a certain role type in the execution of a two-party abstract interaction, this functional role should hold this role type for at least one concrete interaction.

Figure 4-22 depicts three different refinements of an interaction with role discrimination. The roles of actor and reactor are represented by respectively an a and an r attached to each of the functional roles that share an interaction. The first two refinements from left to right are valid whereas the third refinement is not valid according to our refinement guidelines.

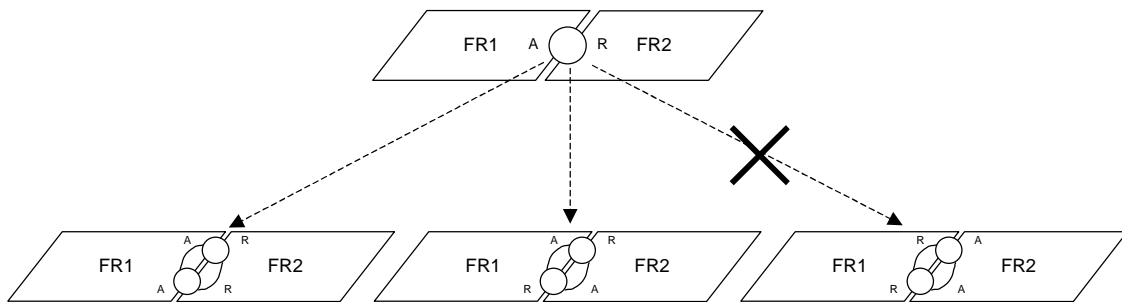


Figure 4-22: Interaction with role discrimination refinement

### Causality refinement

Causality refinement takes place whenever we want to refine the relationship between two activity units in more detail through the introduction of inserted actions. These actions are used to model activities in the concrete behaviour that were not relevant in the specification of the abstract functional behaviour.

Figure 4-23 shows two examples of causality refinement. A functional role is represented by a parallelogram. An action is represented by a circle comprised within the boundaries of a functional role, while an interaction is represented by a circle crossing the boundaries of two functional roles. A relationship between two activity units is represented by a solid line or curve connecting these activity units.

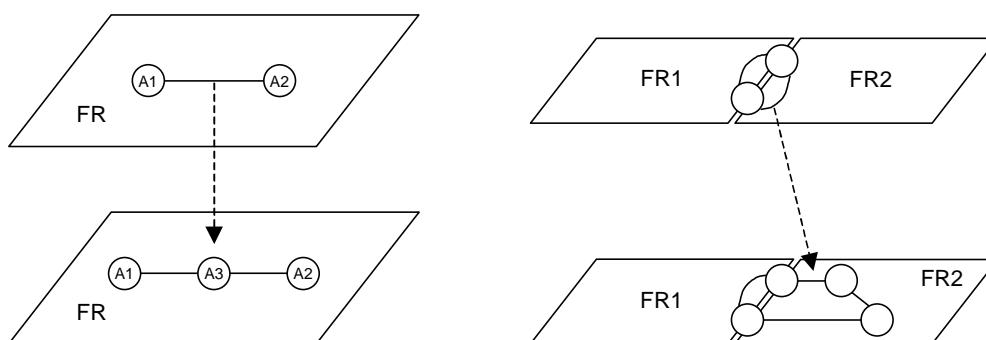


Figure 4-23: Causality refinement

## 4.5 Enterprise specification

The specification of a groupware system at the enterprise level aims at providing a conceptual and integrated description from this system and its environment. The description is conceptual because it models instances of concepts of an application domain, such as those concepts identified in our cooperative work metamodel, and integrated because no division of respon-

sibility with respect to the execution of behaviour is made between the system and its environment, especially in the more abstract specifications.

We consider different sources of information for developing enterprise level specifications, such as domain knowledge information, problem description (usage scenario), user requirements, interviews with experts from the enterprise and other sources. The actual identification and capturing of these sources of information, as well as the format in which these sources are presented to the enterprise modeller are outside the scope of this research. We assume that the information used to create an enterprise level specification is accurate.

#### 4.5.1 Design trajectory

An enterprise level specification should be developed according to the perspective that best suits the objectives of the designer and the available sources of information. In general we model a cooperative work process in three major steps, each one corresponding to the development of an enterprise level specification according to one of the perspectives defined in section 4.4.1.

In the first step, we develop an enterprise level specification according to the integrated perspective, the so-called integrated perspective specification. In the sequel, we refine the integrated perspective specification developing another specification according to the distributed perspective, the so-called distributed perspective specification. In a third step, we refine the distributed perspective specification according to the distributed perspective with role discrimination, creating the so-called distributed perspective specification with role discrimination.

Figure 4-24 illustrates the design trajectory at the enterprise level.

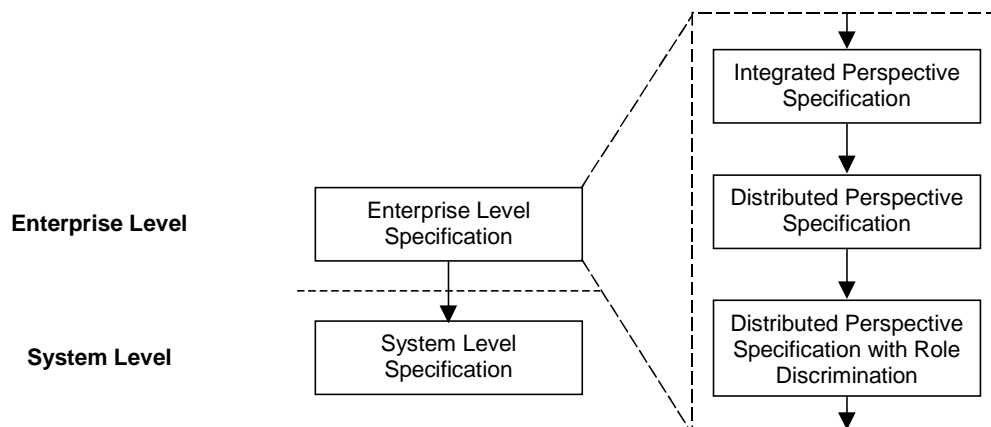


Figure 4-24: Design trajectory at the enterprise level

#### 4.5.2 Modelling strategy

The development of an enterprise level specification, irrespective of the perspective being adopted, is also carried out based on the structural, behavioural and interactional views. These views help us focus on the different subsets of concerns defined by the perspective at a time.

Except for the integrated perspective, which encompasses no interactional concerns, the development of an enterprise level specification according to both the distributed perspective

and the distributed perspective with role discrimination encompasses concerns related to the structural, behavioural and interactional views.

Figure 4-25 illustrates abstractly the intersection of the concerns defined by the different views versus the concerns defined by the different enterprise perspectives.

	Structural View	Behavioural View	Interactional View
Integrated Perspective			
Distributed Perspective			
Distributed Perspective with Role Discrimination			

Figure 4-25: Intersection of views and perspectives concerns

Different specification techniques can be used to create an enterprise level specification according to a given perspective. Each technique may be related to a particular set of concerns, and therefore may or may not be (to some extent) applicable to a certain view. In order to be applied to capture a certain view within the scope of a given perspective, the set of concerns underlying a technique should match the concerns defined by the view.

The structural view at the enterprise level is captured using concept diagrams in combination with a glossary. A concept diagram is used to capture instances of concepts of our cooperative metamodel and their relationships, such as functional entities, functional roles and activity units, while a glossary is used to textually document these concepts.

The behavioural view at the enterprise level is captured using activity diagrams. An activity diagram is used to describe the relationships between the activity units defined in a functional role.

The interactional view at the enterprise level is also captured using activity diagrams. An activity diagram is used to capture the relationships between the interactions identified in an integrated perspective, i.e., we do not consider the contributions of the different functional roles to the execution of the interaction.

A concept diagram could be used to capture the relationships between activity units as well according to both the behavioural and the interactional views. However, the use of this technique for this purpose is not recommended since these concepts diagrams tend to become unreadable when compared to their corresponding activity diagrams (see, for example, FaFS99).

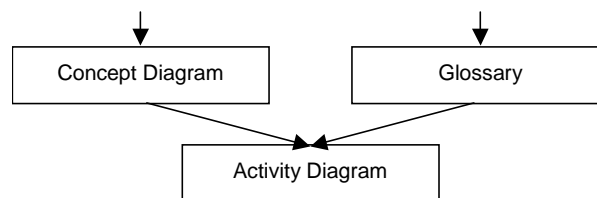
Table 4-2 summarises the techniques used in the development of an enterprise level specification and their adequacy to capture the concerns defined by each view.

Technique	Structural View	Behavioural View	Interactional View
<i>Concept Diagram</i>	applicable	applicable	applicable
<i>Glossary of Terms</i>	applicable	not applicable	not applicable
<i>Activity Diagram</i>	not applicable	applicable	applicable

*Table 4-2: Modelling techniques at the enterprise level*

These techniques are applied in a straightforward way to produce an enterprise level specification, starting with the elaboration of concept diagrams and culminating with the elaboration of activity diagrams.

Figure 4-26 depicts the suggested road map for the development of a given enterprise level specification. Each box corresponds to the elaboration of part of the enterprise level specification based on the corresponding technique. Arrows indicate the order in which the design activities should be carried out. Boxes horizontally placed besides each other indicate that the associated design activity can be carried out in parallel. In the sequel we discuss each one of these techniques separately.



*Figure 4-26: Road map for developing an enterprise level specification*

### 4.5.3 Concept diagram

In most realistic enterprise specifications, it is not practical to capture all the instances of enterprise concepts in a single diagram because this diagram would be too complex to be useful. Instead, we suggest to produce a collection of related concept diagrams, each of them based on a specific requirement or set of related requirements.

A concept diagram is used to capture instances of the concepts and their relationships that are relevant for an enterprise. Therefore, based on our cooperative work metamodel we look for instances of the following concepts: functional entity, functional role, activity unit and resource.

An instance of a concept is represented as a class stereotyped with the type of the concept. For example, suppose we have a requirement saying that a conference controller should be able to create new sub-conferences in cooperation with a conference manager. According to this requirement, relevant concepts are conference controller and conference manager, which can be seen as instances of the functional role concept, and create sub-conference, which can be seen as an instance of the interaction concept.

Figure 4-27 depicts the corresponding concept diagram for the given requirement. A concept is represented as a rectangle, while its associated stereotype is represented between “<< >>”. A line connecting the two concepts represents an association between them, in which a functional role shares the interaction.

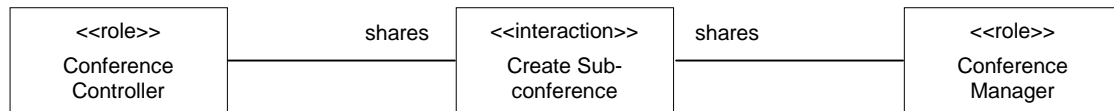


Figure 4-27: Concept diagram for simple requirement

One can decide to capture instances of all types of concepts in the same diagram. However, we suggest the elaboration of diagrams that concentrate on different subsets of concepts. A subset should be organised on a two by two basis, according to the relationship between concepts. By producing diagrams containing only two types of concepts, we focus our attention on these concepts and their relationships. Further, we guarantee that the set of all subsets covers all the concepts and their relationships as identified in section 4.3.4.

Figure 4-28 depicts a simplified version of our cooperative work metamodel, containing only its main concepts. An oval represents a subset of concepts. The separation between subsets is emphasised by using different line patterns for each oval.

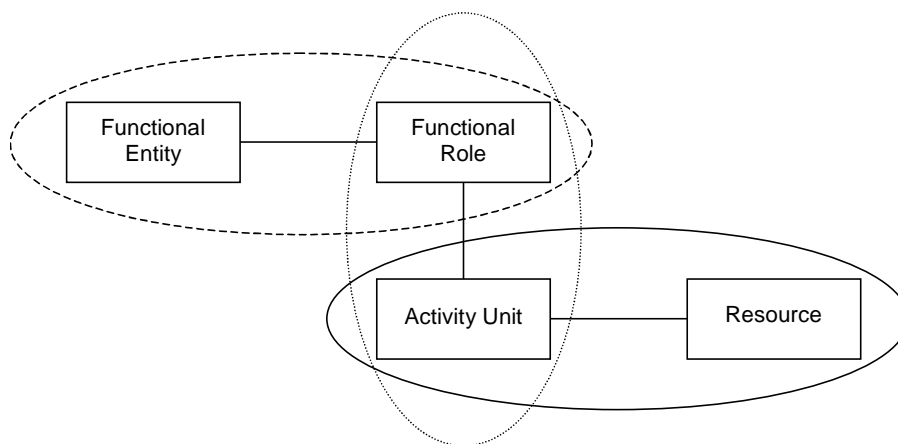


Figure 4-28: Simplified representation of the cooperative work metamodel

By identifying groups of two related concepts we end up with three different subsets of concepts. Therefore, we suggest the elaboration of three different types of concept diagrams:

- *entity-role diagram*, which is used to capture possible relationships between functional entities and functional roles;
- *role-activity diagram*, which is used to capture the activity units defined in a functional role;
- *activity-resource diagram*, which is used to capture any resource that may be used by an activity unit.

There is no specific order in which these diagrams should be developed. In fact, they can be developed in parallel with each other.

### Entity-role diagram

The identification of entities capable of executing behaviour and the identification of possible behaviours that may be associated with these entities is essential within any enterprise specification. In this sense, an entity-role diagram is elaborated in three different steps: (1) the identification of functional entities; (2) the identification of functional roles, and; (3) the establishment of relationships between functional entities and functional roles.

Functional entities and functional roles are two distinct concepts, and their separation is essential in most enterprise specifications. For example, suppose that the individual Richard Pereira currently holds a PhD student position at the Architecture of Distributed Systems (Arch) group of the University of Twente. In the university context, Richard can play the role of PhD student, and consequently Richard is able to conduct some research at the University, help teaching a course offered by his group, or attend a course of interest. Eventually, Richard will probably finish his PhD work and he will not be able to play the PhD student role anymore (unless, of course, he starts a new PhD).

A functional entity is a concrete entity capable of carrying out the execution of some behaviour, such as a person, a group of persons, or a groupware system. Functional entities are key elements, since they are responsible for running or performing the activities within an enterprise. For each functional entity identified, a corresponding class, stereotyped as `<<entity>>`, should be added to the entity-role diagram.

A functional role is a description of some behaviour that can be played by a functional entity. A functional entity should play the functional role in order to perform the activity units defined by the corresponding functional behaviour. A description of all functional roles that can be played by a functional entity in a certain context describes the complete behaviour of the functional entity in that context. For each functional role identified, a corresponding class, stereotyped as `<<role>>`, should be added to the entity-role diagram. The development of enterprise specifications centred on the concept of functional role is similar to what De Weger describes as modular specification style for business process specification [Wege98].

Functional entities can be organised according to the organisational structures and hierarchies of the enterprise. In UML, we model such kind of relationships using an aggregate association between classes. For example, Richard Pereira belongs to the Arch group, which is part of the Telematics Systems and Services (TSS) group. TSS is part of the Faculty of Computer Science, one of the faculties of the University of Twente (see Figure 4-29).

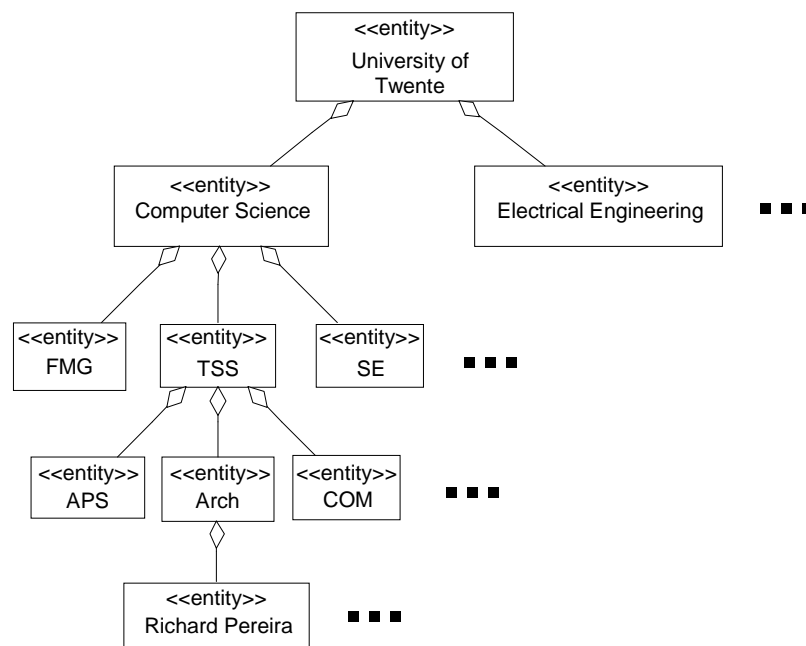


Figure 4-29: Modelling functional entity hierarchies

Functional roles can also be organised according to an aggregation relationship. In this case, a functional role consists of a number of other functional roles, which are in principle independent from each other. For example, it is common in many companies to have the functional role of Chairman-CEO. This role consists of an aggregation of two distinct functional roles, viz., the company chairman and the company CEO. Such a distinction is particularly true when the same functional entity plays two independent functional roles at the same time.

A functional role can also be specialised into one or more specific functional roles, which describe a particular behaviour that the functional entity playing the more generic role may assume provided that some conditions are met. For example, in the Netherlands the functional role PhD student can be specialised into basically two functional roles, viz., AIO and MOZ. The difference between the two specialisations is roughly the following: while an AIO has to provide educational support for the university, an MOZ has to do some work on projects, where he is allocated. Nevertheless, both have to conduct research and write a PhD thesis.

The specialisation relationship between functional roles can also be seen as a representation of a state, or rather a sub-state of the generic functional role, such that the change of states is dynamic and occurs based on the satisfaction of conditions, such as the execution of certain activities.

Similarly to functional entities, the aggregation of functional roles is modelled in UML using an aggregate association between the classes corresponding to the functional roles. The specialisation of functional roles is modelled using generalisation/specialisation association between the classes corresponding to the functional roles. An abstract functional role can be added to the diagram to help structuring the relationships between functional roles. An abstract functional role is similar to the concept of abstract classes in object-oriented languages. An abstract functional role has no corresponding functional role or functional behaviour in the enterprise.

Last but not least, in the elaboration of entity-role diagrams we should associate functional entities with the functional roles they can play. A functional entity can play multiple functional roles, possibly at the same time. In case the functional entity is not allowed to play two or more functional roles at the same time due to, for example, some conflict of interest, this constraint has to be explicit in the associations between the functional entity and functional roles.

If a functional entity is allowed to play only one functional role, there is actually no separation between the entity and the role. In this case, the representation of the association between an entity and the roles this entity is allowed to play is optional, but still desirable. However, if the same entity is allowed to play several roles, these roles should be identified and their association with the entity should be defined accordingly.

A functional role can be played by more than one functional entity, possibly at the same time. In case two or more functional entities are not allowed to play the functional role at the same time, this constraint has also to be explicit in the associations between the functional entities and the functional role.

Since enterprise level specifications can be developed according to different levels of abstraction, the association of functional entities with functional roles may not be straightforward at an early stage of development. Additionally, some functional entities may only be identified in the development of the system and component level specifications. However, this associa-

tion should be eventually established. In case of the existence of restrictions in the associations between functional entities and functional roles, these restrictions can be represented as constraints that are attached to the corresponding associations in the concept diagram.

Figure 4-30 depicts a simple entity-role diagram. A restriction on the associations between the entity and the functional roles is shown as a constraint between the different associations. The XOR constraint specifies that Richard Pereira can play only one role at a time. He can either play the role of AIO or MOZ.

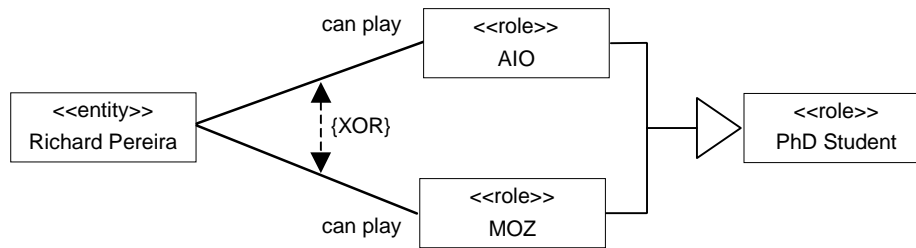


Figure 4-30: Entity-role diagram

### Role-activity diagram

Besides identifying the functional entities and functional roles in the enterprise, we should identify for each role the activity units that can be performed by the entities playing the role, i.e., the activity units defined by the corresponding functional behaviour. The elaboration of a role-activity diagram is carried out in two steps: (1) the identification of activity units, and; (2) the association of these activity units with functional roles.

We identify only those activities representing some coherent piece of functionality that provide some result for a functional entity, represented by a functional role, at a given abstraction level. An activity unit actually represents an activity type, in the sense that a particular functional entity playing the functional role in which the activity unit is defined can perform multiple instances of the corresponding activity unit.

According to our cooperative metamodel, an activity unit can represent either an action or an interaction. The decision on whether or not to represent the activity unit as an action or as an interaction depends exclusively on whether or not at the abstraction level under consideration the execution of the activity unit is carried out entirely by a single functional entity or by two or more functional entities in cooperation. In the former case the activity unit (action) should be defined as part of a single functional role, while in the latter case the activity unit (interaction) should be defined as part of multiple functional roles.

For each action identified, a corresponding class, stereotyped as <<action>>, should be added to the role-activity diagram. Similarly, for each interaction identified, a corresponding class, stereotyped as <<interaction>>, should be added to the diagram.

After identifying all the relevant activity units, we associate those activities with a single functional role (action) or to two or more functional roles (interaction). If an activity unit cannot be associated with a previously identified functional role, a new functional role should be defined and added to the entity-role diagram. Based on the perspective being used to elaborate the enterprise level specification, we should also identify the part taken by the functional role with respect to the execution of an interaction: the functional entity playing the functional role executes the interaction either as an actor or as a reactor.

Figure 4-31 shows a role-activity diagram. According to this diagram three different functional roles share a three-party interaction, Invite to Conference, viz., Conference Manager, Messenger and Conferee.

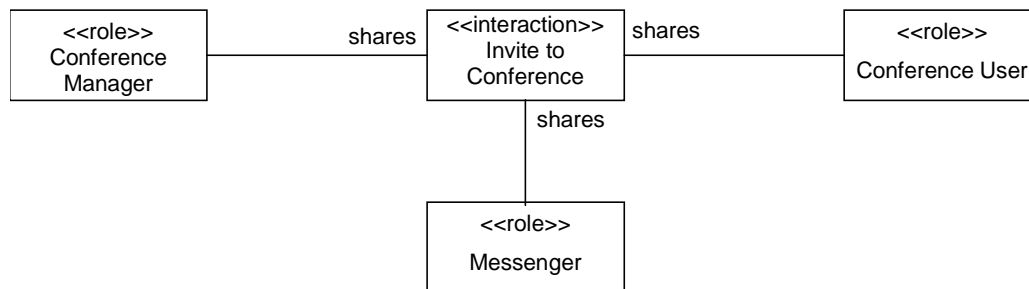


Figure 4-31: Role-activity diagram

Since Invite to Conference is a multi-party interaction at the given abstraction level, the part that each functional role takes with respect to the execution of the interaction is not described.

The example of Figure 4-31 also illustrates the benefit of using a single concept to represent an entity capable of performing behaviour. At this point, we only indicate that a given activity unit requires the participation of three different functional roles in its execution. Further, given a particular context, e.g., modelling of a videoconferencing system, we can reasonably assume that the functional roles of Conference Manager and Conference User can only be played by two different human beings. Nonetheless, we can have many different possibilities with respect to the choice of functional entities that can play the functional role Messenger. For example, we may choose another human being to play the functional role of Messenger, in such case there is no computer support for the execution of this particular unit of work, or we may choose different computing entities to play this functional role, e.g., an e-mail system or the videoconferencing system itself.

The possibility of making such choices increases the flexibility in the development of groupware systems, which is not only desirable but also necessary. The existence of generic computer systems and services in general and groupware systems in particular that are able to carry out the execution of an activity unit or group of related activity units should also be taken into account in the development of enterprise level specifications.

### Activity-resource diagram

An activity unit usually needs some resources in order to be executed properly or produce some resource as a result of its execution. These resources can be computerized, such as documents, charts, records and data, or non-computerized, such as hardcopy documents, contracts, conference rooms and orders. For non-computerized resources we can always have a computerized representation that functions as a surrogate for the resource itself. Therefore, the concept of resource represents all sorts of things that can be represented as a piece of information, which can be used, created, destroyed or changed by an activity unit.

An activity-resource diagram aims at capturing the relationship between an activity unit and any resource associated with this activity. On one hand, an activity may use several resources. On the other hand, a resource can be used by several activities. In case we need to represent that the same resource cannot be used by two or more activity units at the same time, we could add such a constraint in the association between the resource and the activity units.

A resource as a piece of information can be structured. Although not required at this abstraction level, it is possible to represent the properties of a resource as attributes. For example, a resource representing a registration form may have as attributes name, address, age, occupation, etc. A resource may also be related to other resources via aggregation and specialisation relationships.

The identification of the resources associated with an activity unit may not be straightforward at an early stage of the development process. Further, the focus of our research is not information modelling, but rather behaviour modelling. Therefore, we should not be thorough at this point, especially regarding the identification of resource attributes, and try to capture resources that are relevant for the understanding of the behaviour being specified. Whenever possible we should attempt to establish relationships among the identified resources. The UML specialisation and aggregation association types should be used for such purposes. For each resource identified, a corresponding class, stereotyped as <<resource>>, should be added to the activity-resource diagram.

Figure 4-32 depicts a simple activity-resource diagram. The action Create Shared Document makes use of a resource, named Document Info, which represents some information about a shared document being created. This resource is further specialised into Word Document and PowerPoint Document. We could also have represented some attributes of the resources, such as author, version and location.

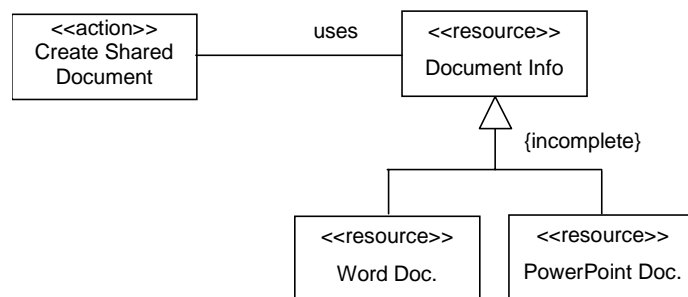


Figure 4-32: Activity-resource diagram

#### 4.5.4 Glossary of terms

A glossary, also known as model dictionary or data dictionary, is a document that defines terms. A glossary lists and defines all the terms that require clarification in order to improve communication and reduce the risk of misunderstanding [Larm97].

The elaboration of a glossary aims at creating and maintaining a standard documentation of the concepts identified at the enterprise level. The glossary is usually elaborated in parallel with development of concept diagrams. However, maintaining and updating the glossary should be an ongoing activity performed throughout the design trajectory.

For each term in the glossary, we must provide the category of the term, the context at which the term has been identified (concern level + perspective) and a short description of its meaning. The category of the term specifies whether the term refers to a functional entity, a functional role, an action, an interaction or a resource.

Since a glossary is not an UML diagram, there is no standard format for a glossary. Two alternative approaches are a textual format, in which each term is described using separate

paragraphs to document the term, and a table format, in which each term is documented in a separate table row with columns for the category, context and description. Figure 4-33 illustrates these alternative formats for a glossary.

*Term:* Richard Pereira  
*Category:* Functional entity  
*Context:* Distributed perspective specification/Enterprise  
*Description:* A member of the Arch group.

*Term:* AIO  
*Category:* Functional role  
*Context:* Distributed perspective specification/Enterprise  
*Description:* A special type of PhD student position.

<i>Term</i>	<i>Category</i>	<i>Context</i>	<i>Description</i>
Richard Pereira	Functional Entity	Distributed perspective specification/Enterprise	A member of the Arch group.
AIO	Functional Role	Distributed perspective specification/Enterprise	A special type of PhD student position.

Figure 4-33: Different formats for a glossary

### 4.5.5 Activity diagram

The elaboration of activity diagrams aims at capturing the relationships between the activity units identified in the role-activity diagrams. Activity diagrams can be used to capture both the behavioural view and the interactional view. In case we consider only the modelling of the relationships between the activity units defined within a single functional role (actions and interactions), we capture the behavioural view. Otherwise, in case we consider only the modelling of the relationships between the activity units shared by two or more functional roles (interactions), we capture the interactional view.

Each activity unit identified in the role-activity diagrams at a given abstraction level should be mapped onto a corresponding activity in an UML activity diagram, which is represented by a state. There are basically four types of states in an UML activity diagram: initial state, final state, action state and activity state.

The initial state indicates the default starting place for the behaviour execution and it is unique within an activity diagram. A final state indicates that the behaviour execution has finished and it can appear multiple times within an activity diagram. Both an initial state and a final state do not represent the execution of an activity. The execution of an activity can only be represented by an action state or an activity state. While an action state cannot be decomposed (action state is atomic), an activity state can be further decomposed into a collection of action and activity states (activity state is non-atomic). Since an activity unit is atomic at a given abstraction level, any activity unit identified in a role-activity diagram should be mapped onto an action state in an activity diagram. So, from now on unless otherwise mentioned (see discussion in section Limitations of modelling with activity diagrams), an activity in a UML activity diagram refers to an action state.

#### Behavioural view modelling

The behavioural view is captured through the elaboration of a number of activity diagrams focusing on capturing the relationships between the activity units defined in a single functional role at a given abstraction level.

For each activity unit representing an action identified in the corresponding role-activity diagram, a corresponding activity, stereotyped as <<action>>, should be added to the activity diagram. Similarly, for each activity unit representing an interaction identified in the corresponding role-activity diagram, a corresponding activity, stereotyped as <<interaction>>, should be added to the activity diagram.

An enabling relationship is established between two activities in an activity diagram by means of a transition connecting these activities. A transition is a directed relationship between a source activity (source state) and a target activity (target state). Upon the completion of the source activity the transition is fired and the execution of the target activity is initiated. It is possible to attach a boolean predicate to a transition, known as guard condition, to provide a fine-grained control over the firing of a transition. In such a case, the transition is fired whenever the source activity is completed and the guard condition evaluates to true.

Figure 4-34 illustrates the enabling relationship in a simple activity diagram. A solid filled circle represents the diagram initial state, while a circle surrounding a solid filled circle represents the diagram final state. A circle represents an action state, while an arrow represents a transition. In this example, we do not distinguish between actions and interactions by means of a stereotype for simplification purposes.

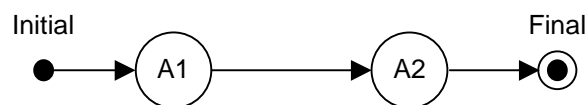


Figure 4-34: Activity diagram with enabling relationship

A choice relationship in the execution of two activity units in general is established by means of two transitions connecting the source activity to the target activities. The choice is not made on the execution of the target activities per se, i.e., on the completion of one of the target activities, but rather on which transition will fire after the source activity is completed. In case we use mutually exclusive guards the transition whose guard satisfies its predicate will fire. In case no guards are used or the guards are non-mutually exclusive, i.e., the predicate of more than one transition is satisfied, a non-deterministic choice is made on which transition will fire and consequently which target activity will be executed.

Figure 4-35 illustrates the choice relationship between activities A2 and A3. A diamond represents a shorthand notation for a choice or decision point. Because there is no guard condition associated with either activity A2 or activity A3, this diagram represents a non-deterministic choice. In this example, we also do not distinguish between actions and interactions for simplification purposes.

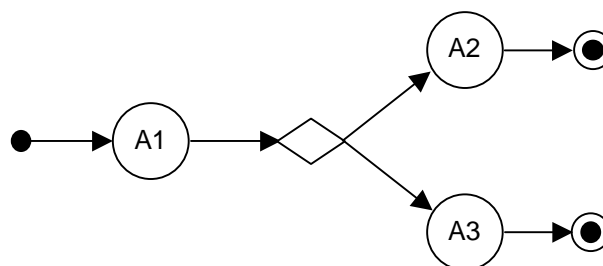


Figure 4-35: Activity diagram containing choice relationship

A concurrency relationship in the execution of two activities in general is established by means of a “fork transition” connecting a source activity to the concurrent activities. Once the transition is fired, the transition is “split” into two or more transitions, such that the execution of two or more activities is initiated. Conversely, to merge the concurrent execution of two or more activities a “join transition” connecting multiple source activities to a single target activity should be used. In such case, once all source transitions are fired the transition to the

target activity is fired as well. Concurrent transitions can be combined with guarding conditions to provide conditional concurrent relationships.

Figure 4-36 illustrates the concurrency relationship between activities A2 and A3. Both a fork transition and a join transition are represented by a synchronisation bar. Once activity A1 is executed, a fork transition is fired and activities A2 and A3 are executed concurrently. Once both activities A2 and A3 are completed, i.e., the transitions from these activities to the synchronisation bar are fired, the join transition is fired and the behaviour comes to an end. In this example, we also do not distinguish between actions and interactions for simplification purposes.

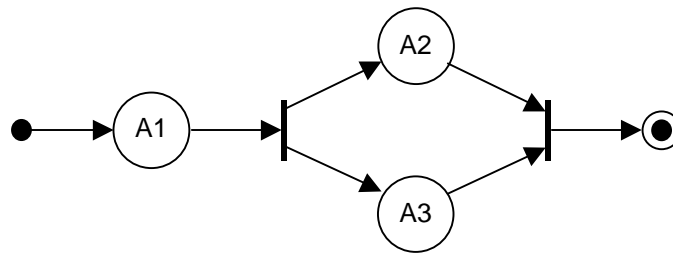


Figure 4-36: Activity diagram containing a concurrent relationship

In case there is a relationship between activity units belonging to different functional roles that can be played by the same functional entity at the same time, we should use swimlanes to capture such relationships<sup>7</sup>. A swimlane represents a partition on an activity diagram used for organising responsibilities for the execution of activities. The activities performed by each functional role should be captured in a separate partition of the diagram.

Figure 4-37 depicts an activity diagram capturing the relationship between activities defined in different functional roles. For each functional role, a separate partition is defined. Partitions are separated by a swimlane, which is represented by a dashed line.

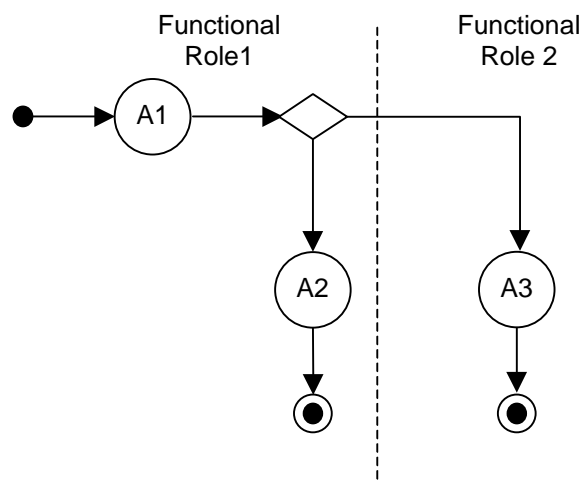


Figure 4-37: Activity diagram relating two functional roles

<sup>7</sup> Such types of relationship are possible in our model because one could imagine that exists a more comprehensive or coarse-grained functional role that contains all functional roles that can be played by a functional entity at the same time, but was not explicitly captured in the diagrams for convenience purposes.

### Interactional view modelling

The interactional view is captured through the elaboration of a number of activity diagrams focusing on capturing the relationships between the activity units that are shared by the functional roles defined at a given abstraction level. For each activity unit representing an interaction identified in the role-activity diagrams, a corresponding activity, stereotyped as <<interaction>>, should be added to the activity diagram.

We capture the interactional view in an integrated way, similar to the development of a specification according to the integrated perspective. However, while in the development of a specification according to the integrated perspective the activity units captured represent actions, in the interactional view modelling the activity units captured represent (integrated) interactions.

In some sense the interactional view complements the behavioural view, or rather, the behavioural view is a decomposition of the interactional view with respect to the modelling of the relationships between interactions. Each functional role has its own behavioural description, which encompasses participations in the execution of interactions. The description of the relationships between the interactions supported by a functional role (abstracting from the actions defined in the role) must be a valid decomposition of the relationships between the interactions as a whole. Otherwise, the integrated behaviour described by the set of interactions and their relationships will not take place.

Figure 4-38 illustrates a valid decomposition of an interactional view description into two behavioural view descriptions. The behaviour described in Figure 4-38a is the common behaviour of the entities that play complementary functional roles, Functional Role A and Functional Role B. The behaviours described in Figure 4-38b and Figure 4-38c are a possible decomposition of the integrated behaviour described Figure 4-38a. Even though the activity unit A1 can start its execution in parallel with the activity unit A2 according to the description in Figure 4-38b, A1 will actually finish before A2 because of the corresponding behaviour description captured in Figure 4-38c.

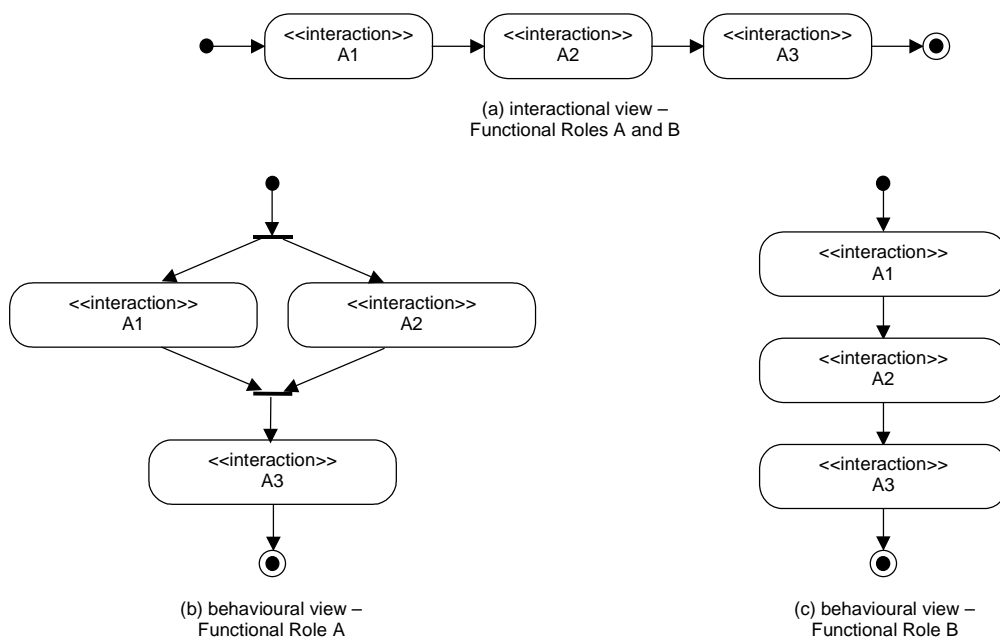


Figure 4-38: Interactional view versus behavioural view

### Limitations of modelling with activity diagrams

Although activity diagrams are the UML technique best suited to model activity units and their relationships, its use poses some limitations towards functional behaviour modelling.

The first major limitation has to do with the absence of appropriate behaviour structuring constructs. The only mechanisms for behavioural structuring are swimlanes and subactivity states.

By using swimlanes we can split activity units according to the responsibility for their execution. Nonetheless, it is not possible to adequately model interactions using swimlanes because there is no way to represent an activity unit whose execution is carried out by two or more functional entities in cooperation (according to complementary functional roles).

Aside from the limitations imposed by the use of swimlanes, the need for behaviour structuring constructs is best experienced whenever we need to model medium to complex behaviours. In order to manage the increasing complexity the only modelling construct available is a subactivity state. By using a subactivity state we can “assign” some closely related behaviour to an activity state, hiding this complexity as needed. However, by using such artifices we unintentionally mix activity units that belong to different abstraction levels in the same diagram.

The use of behaviour blocks, such as proposed in [QFS+97, Quar98, EJL+99], could be considered a more interesting and elegant solution to replace the use of both swimlanes and subactivity states. A behaviour block is a placeholder for behaviour that allows the representation of interactions, as well as, the structuring of complex behaviours without interfering with the abstraction level in which the activity units are being modelled.

The second major limitation for behaviour modelling using activity diagrams is related to the different types of relationships that can be established between activity units. The different possibilities offered by UML can be seen as a subset of the different types of relationship described in section 4.3.5 with some additional constraints.

The enabling relationship described in section 4.3.5 is comparable to a transition between two activity units. However, due to the strong ties between UML activity diagram and UML state-chart diagram, actually the former is a specialisation of the latter, the difference in semantics between a transition and an enabling relationship is perceptible.

In a UML activity diagram an activity is modelled as an action state. An action state represents an atomic activity, whose execution is triggered by a fired transition leading to the state. Therefore, if we consider the behaviour description of Figure 4-34, once the execution of activity unit A1 is completed, the transition will fire and the execution of activity unit A2 is initiated.

An enabling relationship between two activity units, e.g., A1 enables A2, is more abstract than a transition connecting these activity units because it simply prescribes that the execution of A2 can only complete after the execution of A1 is completed. The enabling relationship does not prescribe by any means that A2 has to start only after A1 has finished, although in practical terms it can be interpreted as so, i.e., to enforce that the execution of A2 has to start after A1 is finished is a valid implementation of the enabling relationship.

The semantics of a transition allied with the semantics of an action state also make it impossible to create a disabling-like type of relationship in UML. Since the execution of the activity is atomic, once it has started, i.e., the transition has been triggered, it cannot be stopped and consequently the execution of an activity unit cannot be disabled.

The disabling relationship is very important because it allows to model, for example, the choice between the two activity units based on the completion of these activities rather than based on the triggering of these activities, as supported by UML.

Although, perhaps, not as important as the disabling relationship, the synchronisation relationship is also not supported by UML. There is no way we can synchronise the completion of the execution of activity units. By means of a synchronisation bar, we can only either synchronise the beginning of execution of two or more activities or wait for the completion of two or more activities before continue with the execution of the behaviour, which in both cases does not imply that the results will be available at the same time.

## 4.6 Specification example: travel cost claim process

This section illustrates the enterprise concepts, perspectives and modelling techniques presented along this chapter using a simple case study. In this case study, we model a simple travel cost claim process, in which the employees of the Faculty of Computer Science request a reimbursement for travel expenses. Basically, once a travel cost claim is submitted it has to be evaluated and in case the claim is approved the corresponding reimbursement is made.

The claim process was specified according to three different abstraction levels, each one developed according to a different behavioural perspective (see section 4.4.1). However, on this case study we concentrate on basically three types of diagram, viz., entity-role, role-activity and activity diagrams. In the sequel we present the main aspects of each one of these abstraction levels.

### 4.6.1 Integrated perspective specification

The enterprise specification was developed according to the integrated perspective. According to this perspective we represent the part of the enterprise being modelled as a single functional entity and the corresponding cooperative process as part of a single functional role.

Figure 4-39 depicts the entity-role diagram developed according to the integrated perspective. The functional entity Computer Science can play only a single functional role, Travel Cost Claim, which models our claim process.

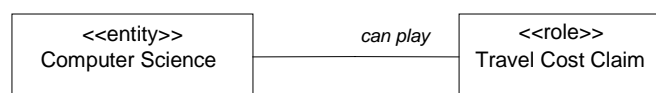


Figure 4-39: Integrated entity-role diagram

The functional role Travel Cost Claim defines a number of activity units, which are performed exclusively by the functional entity Computer Science. Therefore, these activity units are modelled as actions according to this perspective. The following actions have been identified at this level (see Figure 4-40): Submit Travel Claim, Assess Claim, Inform Claim Assessment and Reimburse Expenses.

The action `Submit Travel Claim` models the submission of a cost claim describing a number of travel expenses. The action `Assess Claim` models the assessment of expenses declared on the cost claim. The action `Inform Claim Assessment` models the notification of the assessment made on the travel claim. The action `Reimburse Expenses` models the operation of reimbursement of the approved expenses.

Figure 4-40 depicts the role-activity diagram developed according to the integrated perspective.

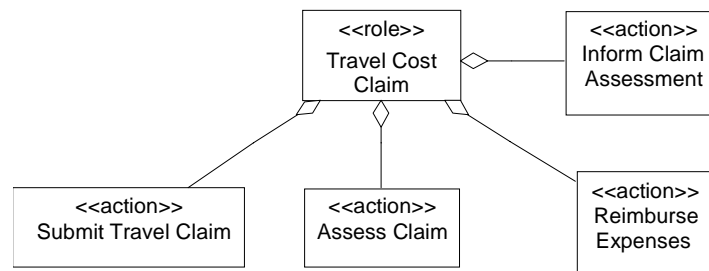


Figure 4-40: Integrated role-activity diagram

Once the activity units that form a functional role have been identified we have to model the relationship between these units. The relationships between the identified activity units are captured by means of an activity diagram, as shown in Figure 4-41.

Once a travel claim is submitted, it must be evaluated and the result of this evaluation should be informed. In case the claim is approved the expenses are reimbursed, otherwise no reimbursement takes place.

Figure 4-41 depicts the activity diagram developed according to the integrated perspective.

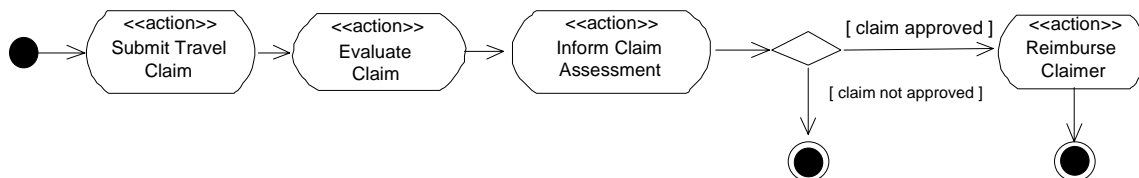


Figure 4-41: Integrated activity diagram

## 4.6.2 Distributed perspective specification

The next step after specifying the travel cost claim process according to the integrated perspective is to refine this specification, producing a new specification according to the distributed perspective.

In this case study, the functional entity `Computer Science` is refined into a number of functional entities represented by its collective, called `Faculty Personnel`. This functional entity represents any personnel of the `Computer Science Faculty`. The functional role `Travel Claim` is also refined into four different functional roles, viz., `Claimant`, `Manager`, `Financial Clerk` and `Workflow Controller`.

In this simplified case study, any faculty member can play any of the aforementioned functional roles without restrictions. However, due to some conflict of interest, in a more elaborated case study we would imagine that at least there would be a restriction stating that a fac-

ulty member playing the functional role Claimant should not play the functional role Manager at the same time and/or we could have further refined faculty personnel into management personnel, financial personnel, etc and we could have added some extra constraints in the association between these functional entities and functional roles.

Figure 4-42 depicts an entity-role diagram developed according to the distributed perspective.

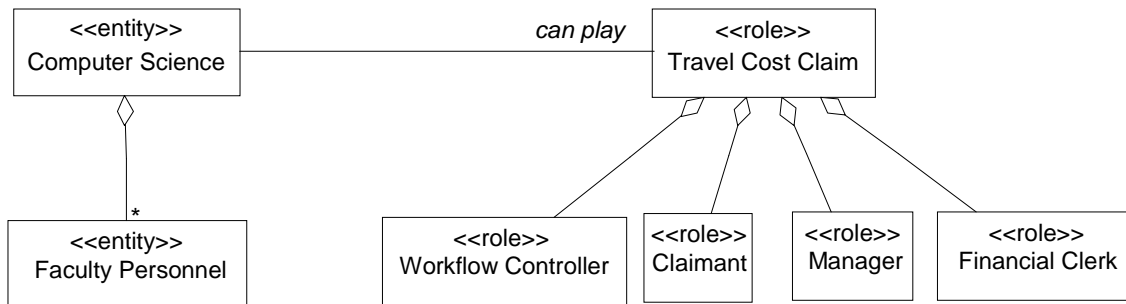


Figure 4-42: Distributed entity-role diagram

Not only the functional role Travel Cost Claim is refined at this level, but also each of the previously defined activity units. The action Submit Travel Claim is refined into the interaction \_Submit Travel Claim, which is shared by the functional roles Claimant and Workflow Controller. The action Assess Claim is refined into the interaction \_Assess Claim, which is shared by the functional roles Manager and Workflow Controller. The action Inform Claim Assessment is refined into the interaction \_Inform Claim Assessment, which is shared by the functional roles Claimant and Workflow Controller. Finally, the action Reimburse Expenses is refined into the interaction \_Reimburse Expenses, which is shared by functional roles Claimant, Financial Clerk and Workflow Controller.

Figure 4-43 shows a role-activity diagram developed according to the distributed perspective

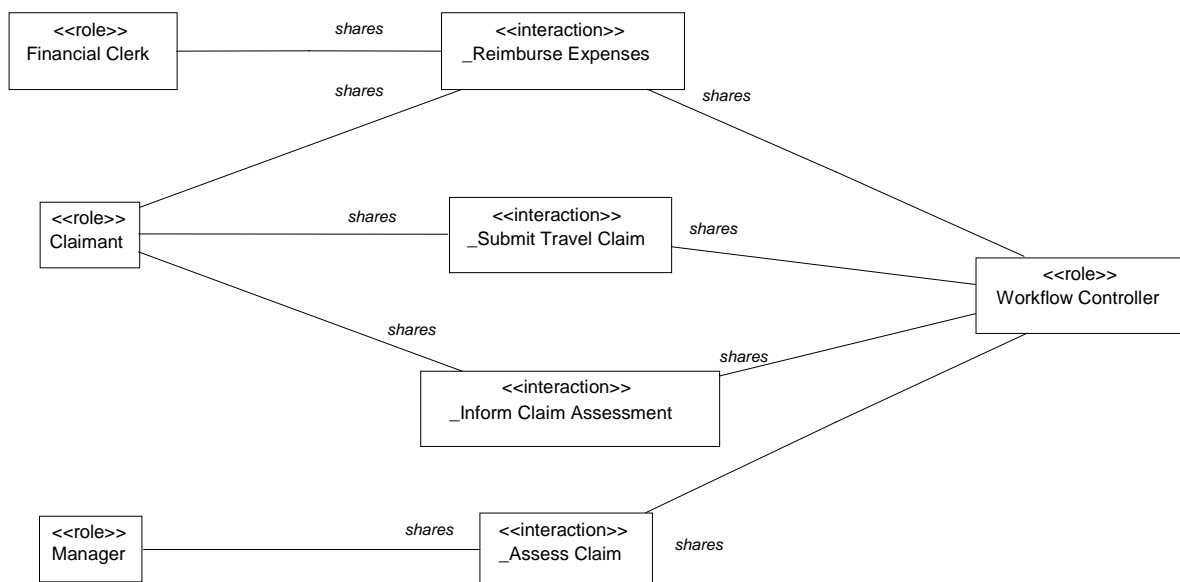


Figure 4-43: Distributed role-activity diagram

For each separate functional role identified at this level, a separate activity diagram is developed to capture the relationship between the interactions shared by each functional role. These diagrams together with the role-activity diagram form the behavioural view description.

Figure 4-44 shows the activity diagram for the functional role Claimant.

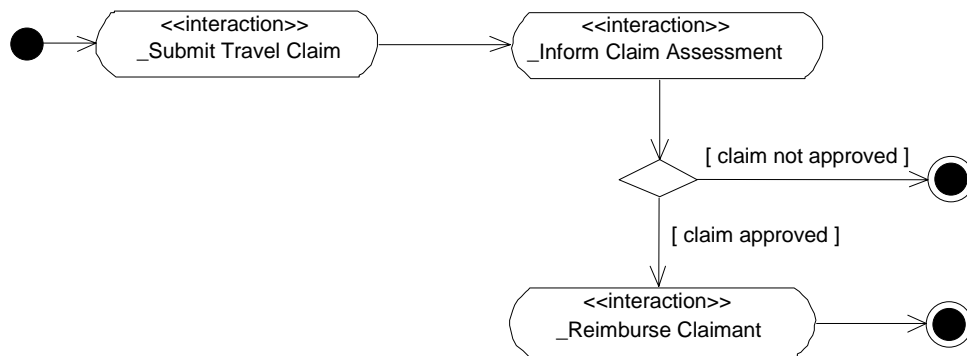


Figure 4-44: Claimant activity diagram

The interactional view is captured according to the distributed perspective by the development of an activity diagram describing integrated interactions, which in this case is similar to the activity diagram described by Figure 4-41.

### 4.6.3 Distributed perspective with role discrimination specification

The next step after specifying the travel claim process according to the distributed perspective is to refine this specification, producing a new specification according to the distributed perspective with role discrimination.

In this case study, there are no changes on the functional entities identified at this level if compared to the functional entities identified at the previous abstraction level. The different functional roles also remain the same. However, we do refine some of the interactions previously identified as follows. The interaction `_Assess Claim` is refined into the interactions `_Travel Claim Notification` and `_Assess Claim`, both shared by the functional roles `Manager` and `Workflow Controller`. The interaction `_Reimburse Expenses` is refined into the interaction `_Reimbursement Request Notification`, which is shared by functional roles `Financial Clerk` and `Workflow Controller`, and the interaction `_Transfer Money`, which is shared by the functional roles `Financial Clerk` and `Claimant`.

Further, we also identify the role played by the functional entities while executing each interaction at this level. For example, the functional entity playing the functional role `Claimant` performs the interaction `_Submit Travel Claim` as actor, whereas the functional entity playing the functional role `Workflow Controller` performs this interaction as reactor.

Figure 4-45 depicts a role-activity diagram developed according to the distributed perspective with role discrimination.

Similarly to the previous abstraction level, a separate activity diagram is developed for each functional role to capture the relationship between the interaction shared by each functional role in separate. These diagrams together with the role-activity diagram form the behavioural view at this abstraction level.

An activity diagram describing integrated interactions is also developed. This diagram, which captures the interactional view, is shown in Figure 4-46.

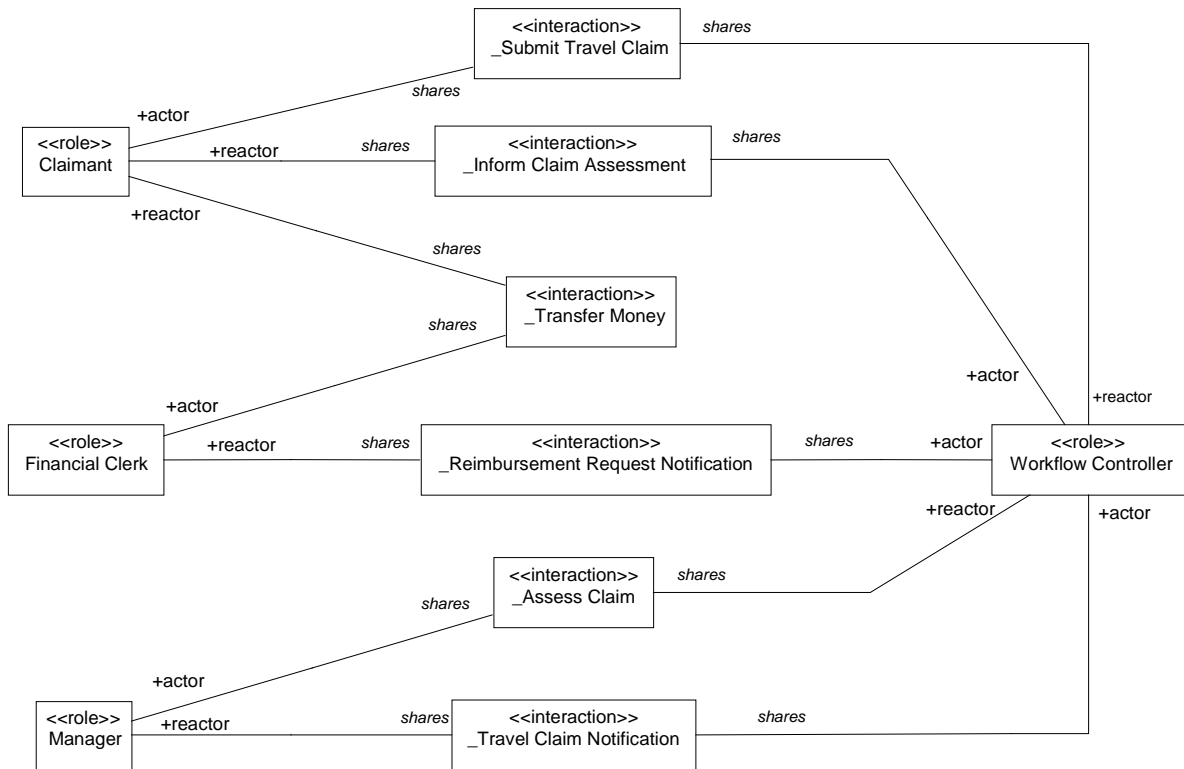


Figure 4-45: Distributed role-activity diagram with role discrimination

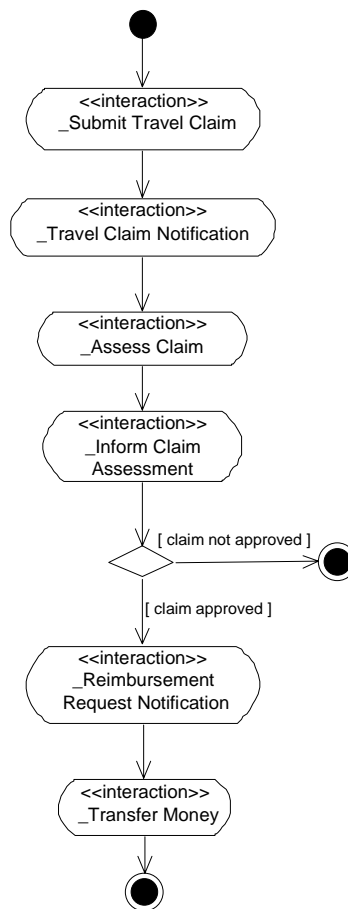


Figure 4-46: Interactional view of enterprise specification

## 4.7 Conclusion

This chapter concentrates on the first concern level of our design methodology, the enterprise level. The chapter explores the notion of enterprise and enterprise support systems. This chapter also analyses a number of cooperative work models and proposes a new cooperative work metamodel to serve as basis for the development of enterprise level specifications targeted at cooperative work support.

The cooperative work models analysed are represented using UML class diagrams. Since we are only interested in capturing the concepts and their relationships present in a model, the use of class diagrams suffices and poses no threat of improperly representing the models. Based on the study carried out on these models, our informal interpretation of cooperative work and a number of quality principles, a number of criteria are drawn to enable us to compare those models and to propose a new metamodel, which emphasises behaviour modelling. We believe that representing the behaviour of enterprise entities is the best approach towards effectively capturing the functional requirements of an automated support for some cooperative work process.

Four basic concepts have been identified, and their relationships have been established: functional entity, functional role, activity unit and resource. These concepts are generic and subjected to specialisation according to some purpose. For example, an activity unit can be specialised into an action or interaction depending on whether or not the execution of the activity unit is shared by multiple functional roles. Further, in case of two-party interactions, the part taken by the functional role in the execution of an interaction can be described as either actor or reactor, which indicates the primarily responsibility for the execution of the interaction.

The chapter also suggests that the modelling of a cooperative process at the enterprise level should be carried out according to different abstraction levels, such that the transition to the system level afterwards is facilitated. We have identified three perspectives that can be used to produce enterprise level specifications: the integrated perspective, the distributed perspective and the distributed perspective with role discrimination.

Based on the objectives of a perspective and how the basic views defined in our methodology apply to these perspectives, a number of modelling techniques can be used to produce an enterprise level specification. We suggest a simple road map to describe the order in which descriptions based on these techniques can be developed and for each technique we provide a number of guidelines for its application.

Not only we provide guidelines for developing enterprise level specifications based on UML, but also we identify some pitfalls regarding the use of UML for developing such specifications. In order to couple with these pitfalls in some extreme cases the use of the current UML standard could be complemented or replaced by some other design notation.

There are a number of factors that may influence the development of enterprise level specifications, a.o., the domain of the cooperative process being modelled and available telematics systems support.

The application domain is the first factor to influence the modelling of a cooperative work context at this level. Depending on the domain, a different set of concepts can be applied both to facilitate the communication between the enterprise designers and enterprise stakeholders,

and to focus on the relevant aspects of that particular domain. Nevertheless, we believe that the proposed set of concepts is quite generic and can be successfully used in several cooperative work domains without significant changes. A previous version of the metamodel proposed here [FaFS00a], which was not as generic as this one, has been successfully used in the abstract modelling of some cooperative work context, such as a travel claim process, a paper reviewing process and brainstorming conferencing. These examples are reported in [FaFS99].

The knowledge of available telematics systems and services also helps an enterprise modeller on deciding whether or not to use these systems and services as functional entities capable of carrying out the execution of some of the identified activity units according to some functional role. If no functional entity is available to play a certain functional role, such functional entity has to be developed.

---

# Chapter 5

## System Level Modelling

This chapter concentrates on the system concern level of our design methodology. The chapter first discusses in general terms the relationship between the enterprise level and the system level, introducing specialisations for a number of enterprise concepts and introducing new concepts as necessary. These specialisations form the basis for modelling a groupware system at the system level. The chapter also presents a number of perspectives that can be used to model the behavioural aspects of a groupware system, viz., required service, decomposed required service, and internal integrated perspectives. Afterwards, the chapter presents the modelling techniques used at the system level according to a design trajectory that takes into account these behavioural perspectives.

This chapter is structured as follows: section 5.1 discusses the relationship between the enterprise and system levels; section 5.2 presents the specialisation of a number of enterprise concepts into system concepts; section 5.3 introduces a number of behaviour perspectives and describes the system design trajectory based on the development of specifications according to these perspectives; section 5.4 concentrates on developing the system service specifications; section 5.5 concentrates on developing the system internal behaviour specification; finally, section 5.6 presents some conclusions.

### 5.1 Enterprise to system transition

According to our component-based groupware design methodology, enterprise level specifications are used to help us understand the context in which groupware systems should operate, by capturing the relevant aspects of a cooperative work process at a high abstraction level. Additionally, as the design trajectory points towards the system level, enterprise level specifications can also help us better understand the system requirements.

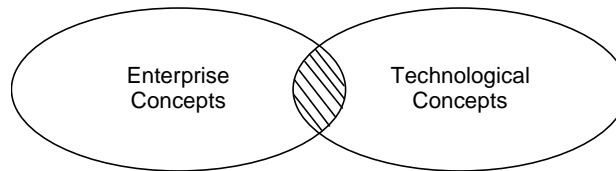
At the system level of our design methodology, the enterprise reference specification is used as basis for the decision on which functionality should be provided by a groupware system and which functionality should be part of its environment, regardless of whether or not there is some computer supported involved.

An enterprise and the technological support for its processes or activities are at different concern levels, i.e., the set of concerns that drive them are usually different from each other. This difference is reflected by the different sets of concepts that are usually used to produce enterprise level specifications and system level specifications, respectively.

Nonetheless, in order for someone to properly and systematically develop telematics systems, in general, and groupware systems, in particular, that can effectively support an enterprise there should be some relationship between enterprise level concepts and system level concepts. Otherwise, one could argue that there would not be a systematic way to design telematics systems based on enterprise specifications. Further, one could imagine that in case

telematics systems properly support an enterprise, this support is more coincidental rather than engineered.

Figure 5-1 illustrates the intersection between two different sets of concepts representing enterprise driven concerns and technological driven concerns.

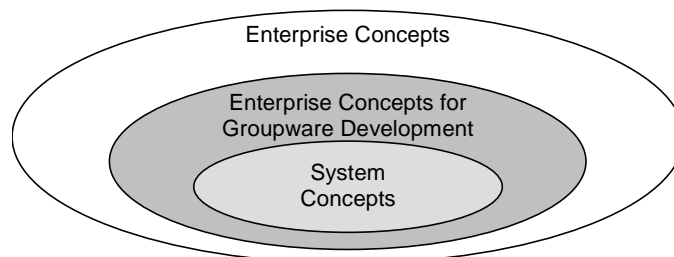


*Figure 5-1: General enterprise versus technological intersection of concepts*

The stronger the correspondence between the sets of concepts concerning the enterprise concerns and the set of concepts concerning the system, the easier the transition from one to another. So, ideally, the set of system concepts should be a subset of the set of enterprise concepts.

Since our set of enterprise concepts were identified having in mind the abstract representation of computer support for cooperative work, the concepts used at the system level of our design methodology are basically the same as the ones used at the enterprise level. However, since the system level represents a more concrete concern level, some of those concepts have to be specialised.

Figure 5-2 represents the relationship between different sets of concepts. The set of concepts used at the system level is basically a specialisation or subset of the set of concepts used at the enterprise level. The set of concepts used at the enterprise level is a subset of the set of concepts that can be used for the development of enterprise specifications in general.



*Figure 5-2: Abstract relationship between enterprise and system concepts*

## 5.2 Architectural concepts

This section discusses the specialisation of some enterprise concepts into a number of architectural concepts that are used both at the system and component levels.

Table 5-1 shows the main correspondences between concepts at the enterprise and system levels. These sets contain basically the same concepts.

Enterprise Level Concepts	System Level Concepts
functional entity	functional entity
functional role/behaviour	functional behaviour
action	action
interaction	interaction
resource	information

*Table 5-1: Correspondence between concepts*

### 5.2.1 Functional entities

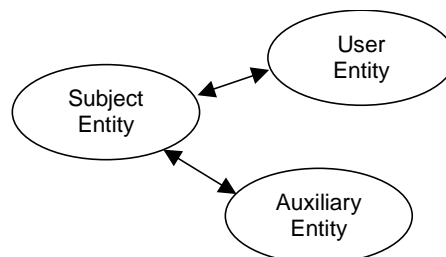
One of the primary concepts at the system level is functional entity, which is the same as defined at the enterprise level. Each functional entity has a single (functional) behaviour, which is determined by a composition of the functional behaviours described by a number of functional roles that the functional entity can play at the enterprise level. Therefore, one or more enterprise functional roles (and their corresponding functional behaviours) are mapped onto a single functional behaviour at the system level.

Functional entities do not exist in isolation, but rather they are interconnected. A functional entity has some capability (functionality) that can be accessed by its environment, which comprises other functional entities. The functionality of an entity represents what it can do or provide to other entities.

From the point of view of an entity under development, the so-called subject entity, its environment usually consists of a number of other entities, which can be classified as user entities or auxiliary entities. User entities consist of human beings or telematics systems that use the functionality provided by the subject entity. Auxiliary entities consist of other telematics systems whose functionality is used by the subject entity to provide its own functionality. For an entity to use the functionality provided by another entity, both entities need to interact with each other.

At the system level, the subject entity, its user entities and its auxiliary entities correspond to those functional entities identified at the enterprise level.

Figure 5-3 depicts the different types of functional entities as they interact with each other. A functional entity is represented by an oval, while an interaction between two entities is represented by a double edge arrow connecting two entities.



*Figure 5-3: Different types of functional entities*

### 5.2.2 Functional behaviour

The behaviour of a functional entity is defined by a set of related activity units whose execution yields some desired functionality. An activity unit is modelled either as an action or an interaction. The concept of action remains the same as defined at the enterprise level, however the concept of interaction is further specialised to fit the characteristics of the system level (see section 5.2.3).

The transition from the enterprise level to the system level usually involves the refinement of the abstract activity units into concrete activity units. The abstract activity units are defined as part of the functional behaviour described by each of the functional roles mapped onto the functional entities at the system level. However, since our focus is on the design of a subject entity, only those activity units associated with its functional behaviour are refined.

The refinement guidelines used in this process are the same as those used at the enterprise level and presented in Chapter 4 (see section 4.4.2). However, regardless of the abstraction level in which the reference enterprise level specification is developed, the interactions obtained should always conform to the interaction characteristics as defined at the system level.

The execution of an activity produces some result, which is represented as information that can be used by other activities. The concept of information corresponds to the enterprise concept of resource, which is used during the execution of an action and exchanged during interactions.

### 5.2.3 Interaction specialisation

The concept of interaction as originally introduced in Chapter 4 is highly abstract and generic. On one hand, this concept is abstract because it allows complex activities to be described in a concise form. On the other hand, this concept is generic because an interaction may involve the participation of more than two functional entities.

In principle, the generality and abstraction inherent to an interaction pose no problem if we produce models at a high abstraction level, such as those developed at the enterprise level. In fact, these capabilities are useful at high abstraction levels. However, the realisation of these models may not be straightforward at a certain moment during the design process.

Therefore, we need to specialise the concept of interaction, such that: (1) it can be used across the system and component levels of our design methodology; (2) it facilitates the mapping onto implementations afterwards; and (3) it can be represented using UML or an extension thereof.

As a first step towards this specialisation, an interaction was restricted in Chapter 4 to the participation of two entities according to the distributed perspective with role discrimination. Further, two different role types were associated with an interaction (actor and reactor), which determine the responsibility taken by a functional entity in the execution of the interaction.

At the system level, we further specialise an interaction by identifying the contributions of the functional entities to the execution of the interaction and to the establishment of information values. In the context of this work, we consider a single type of interaction contribution, viz., value passing [VFQ+00]. Under these circumstances, an interaction is always asymmetrical,

i.e., the entities involved in the interaction play different and complementary roles and information values can be passed in only one direction according to these roles.

Since in the context of this work interactions are the means used by an entity to use (provide) the functionality provided (used) by another entity, we distinguish between the roles of *interaction initiator* and *interaction reactor*, which correspond to the enterprise roles of actor and reactor, respectively. An interaction initiator corresponds to the entity that initiates the interaction to request the execution of some functionality by another entity, while an interaction reactor corresponds to the entity that reacts upon the request of the interaction initiator to provide the requested functionality. We say that an entity supports an interaction if this entity plays the role of interaction reactor with respect to this interaction.

Figure 5-4 illustrates the representation of an interaction at the system level. A white oval is used to represent an entity, while a solid arrow originating in the interaction initiator entity and pointing towards the interaction reactor entity represents the interaction.

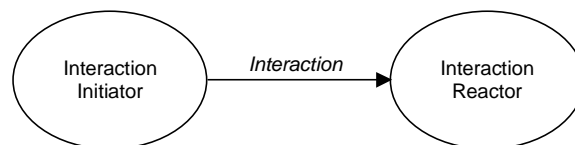


Figure 5-4: Interaction at the system level

In practical terms, the occurrence of an interaction can be interpreted as the generation of a stimulus by the interaction initiator followed by the reception of this stimulus by the interaction reactor. A stimulus represents an instance of communication between two entities that may convey some information. We abstract from the mechanisms used to transmit the stimulus from the interaction initiator to the interaction reactor (see section 5.2.4).

An interaction stimulus is generated after the occurrence of an activity in the interaction initiator that enables the occurrence of the interaction. The reception of the interaction stimulus enables the occurrence of other (inter)actions in the interaction reactor.

#### 5.2.4 Interaction point

An *interaction point* models the mechanisms through which entities can interact with each other [VFQ+00]. Such mechanisms can have different forms: the entities involved in the interaction can be in direct contact with each other, e.g., a (human) user interacting with a telematics system via a graphical user interface (GUI) provided by the system; or the entities involved in the interaction may be separated from each other by a physical distance, e.g., two applications running at different computing platforms and interacting with each other using the capabilities of a middleware platform.

The concept of interaction point abstracts from the actual mechanisms, but allows the designer to represent that interactions share a mechanism. Therefore, an interaction point can be seen as a (logical) location. That is why we say that interactions occur at interaction points. In the context of this work we assume that interactions in the real world take place via interfaces, such as a GUI or an OMG IDL interface [OMG00b].

Figure 5-5 illustrates a number of interactions between functional entities. A solid double edge arrow is used to model a group of interactions. In a detailed view of one of these groups,

an interface is graphically represented using a horizontal T-bar attached to one of the entities, while an interaction is represented as a dashed arrow connecting one entity to the interface of another entity.

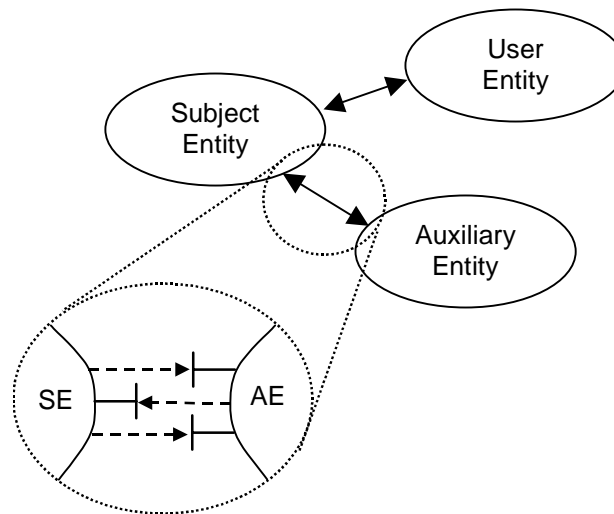


Figure 5-5: Interfaces as interaction points

### 5.2.5 Interaction patterns

We distinguish between two different interaction patterns based on the purpose of the interactions and the causality relations between interactions, viz., *invocation pattern* and *notification pattern*.

The invocation pattern consists of two causality-related interactions that are used by, e.g., a user entity to request the execution of some functionality provided by the subject entity, and to inform the user entity of the result of this execution, respectively. The occurrence of the first interaction, also called *request interaction*, is followed by the occurrence of a second interaction, also called *response interaction*. Conversely, the occurrence of a response interaction is always preceded by the occurrence of a request interaction.

The occurrence of a response interaction does not guarantee that the functionality has been executed accordingly, since it may also indicate the subject entity has failed to provide the requested functionality. In such case, the response interaction indicates the occurrence of an exception during the execution of the requested functionality.

The notification pattern consists of a single interaction, called event notification interaction, which is used by, e.g., the subject entity to announce or notify the occurrence of noteworthy situations to a user entity. There is no response interaction associated with an event notification interaction.

In principle, the notification pattern could also be used to request the execution of some functionality without expecting some response in return. In this case, there is no guarantee whatsoever that the requested functionality has been properly executed.

Figure 5-6 illustrates the aforementioned interaction patterns. Figure 5-6a shows the invocation interaction pattern, while Figure 5-6b shows the notification interaction pattern. In the

invocation interaction pattern, the response interaction is shown as a dashed arrow to distinguish it from the request interaction.

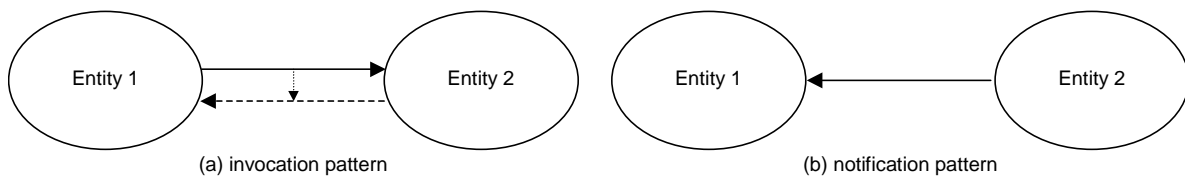


Figure 5-6: Interaction patterns

In practice, interaction patterns are mapped onto operation specifications, in case of IDL-like interfaces, or to event descriptions, in case of GUIs.

An operation specification basically consists of the operation name, a return clause, parameters and return values. In the case of the invocation pattern, the operation name represents the request interaction, and the return clause (including exceptions) represents the response interaction, parameters and return values represent the information exchanged during the request and response interactions. In the case of the notification pattern, the operation name represents the event notification interaction and the return clause is empty. The operation parameters represent the information exchanged during the interaction. There are no return values associated with the interaction.

Event descriptions are similar to operation specifications. However, different events are used to represent different interactions, i.e., we suggest the use of separate events to represent a request interaction or an event notification interaction and a response interaction. Each event has its own parameters that represent the information exchanged during the interaction.

To avoid confusion, we use the term interface to refer to any type of interface. We also say that an entity provides an interface in case this entity supports the interactions that are associated with the interface specification.

## 5.3 System modelling

### 5.3.1 Behaviour perspectives

The behaviour of a subject entity can be modelled according to different perspectives. We distinguish three different perspectives, viz., the *required service perspective*, the *decomposed required service perspective* and the *integrated internal perspective*.

#### Required service perspective

The required service perspective provides the most abstract description of the behaviour of a subject entity. The required service of a subject entity consists of the description of the entity behaviour according to point of view of its user entities, henceforth known as service users.

Therefore, the required service of an entity can be described in terms of the interactions between the subject entity and its service users and their ordering and information value dependencies. This description abstracts from the interactions between the subject entity and its auxiliary entities. The subject entity and its auxiliary entities are considered as a single entity.

Figure 5-7 illustrates the required service perspective of a subject entity. The subject entity, represented as an oval, and its auxiliary entity, represented as a weight-lifter icon, are seen as a single entity, represented as a dashed oval. A service user is represented as a human icon. Interactions are represented as double edge arrows connecting the subject entity with its service users or auxiliary entity.

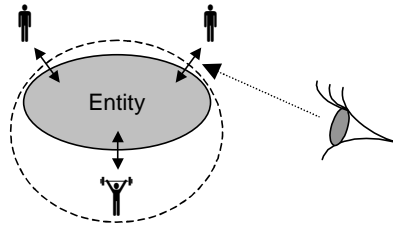


Figure 5-7: Required service perspective

The combined behaviour of the subject entity and its auxiliary entities is described using a black box metaphor. In a black box metaphor, the description is provided only in terms of the behaviour that is observed from the outside by the service users. The observable behaviour includes what happens, when it happens and what it generates as result.

### Decomposed required service perspective

The decomposed required service perspective also enables an abstract description of the behaviour of a subject entity. However, the description provided by this perspective takes into account not only the entity service users but also its auxiliary entities.

Therefore, the decomposed required service of an entity can be described in terms of the interactions between the subject entity and its service users and between the subject entity and its auxiliary entities, and their ordering and information value dependencies. This description also takes into account the services that may be used by the subject entity in order to provide its own service. Consequently, the subject entity and its auxiliary entities are seen as separate entities.

Figure 5-8 illustrates the decomposed required service perspective of a subject entity. The behaviour of the subject entity, represented as an oval, is described using a black box metaphor. A service user is represented as a human icon, while an auxiliary entity is represented as a weight-lifter icon. Interactions are represented as double edge arrows connecting the subject entity with its service users or auxiliary entity.

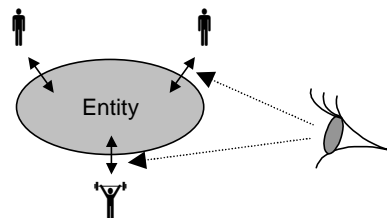


Figure 5-8: Decomposed required service perspective

### Integrated internal perspective

The integrated internal perspective provides a more concrete description of the behaviour of a subject entity compared to the required service and decomposed required service perspectives.

From the integrated perspective, the behaviour of the subject entity is described not only in terms of the behaviour as observed from the outside, but also in terms of the behaviour as observed from the inside in an integrated and monolithic way.

Therefore, the integrated behaviour of an entity is described not only in terms of the interaction contributions in which the entity participates and their ordering and information value dependencies, but also in terms of a set of internal actions and their relationships and the relationship between actions and interaction contributions. There is no disclosure of the entity internal structure, i.e., actions are not assigned to sub-entities, hence the name integrated internal perspective.

Figure 5-9 illustrates the integrated internal perspective of a subject entity. The subject entity is represented as an oval. A service user is represented as a human icon, while an auxiliary entity is represented as weight-lifter icon. An action is represented as small circle, while a relationship between two actions is represented as a line connecting these actions. Interactions are represented as double edge arrows connecting the entity with its service users or auxiliary entity.

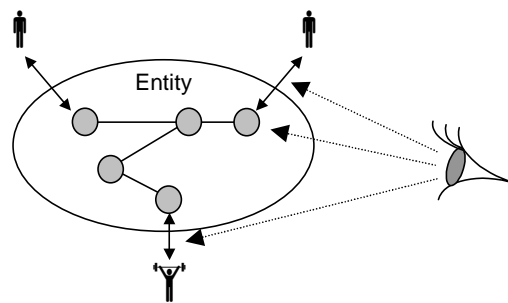


Figure 5-9: Integrated internal perspective

The behaviour of the subject entity is described using a grey box metaphor. In a grey box metaphor, the description is provided not only in terms of the behaviour that is observed from the outside, but also in terms of the behaviour observed from the inside in an integrated way.

### 5.3.2 Design trajectory

At the system level, a subject entity corresponds to the groupware system under development, while user entities of the subject entity correspond to the system users. Auxiliary entities correspond to other systems whose functionality is used by the groupware system under development.

We model a groupware system at the system level in two major steps. In a first step, we specify the system service, which is also usually carried out in two steps, viz., the elaboration of a specification according to the required service perspective, the so-called required service specification, followed by the elaboration of a specification according to the decomposed required service perspective, the so-called decomposed service specification.

The development of the required service specification constitutes a desirable milestone along the design trajectory at the system level because it facilitates the development of the decomposed service specification afterwards. Nevertheless, the development of the required service specification is not mandatory. Further, in case a groupware system has no auxiliary entity

associated with it, the required service specification corresponds to the decomposed service specification.

In our design trajectory, the specification of the system service is not enough to refine the system into components afterwards. We also need to specify the system according to the internal integrated perspective to help us decide on how to decompose the system properly. The development of such a specification is therefore recommended.

So, in a second step, we have to specify the internal integrated behaviour of the system, the so-called internal behaviour specification. The internal system behaviour specification is an intermediary design step to arrive at the component level.

Figure 5-10 illustrates the design trajectory at the system level.

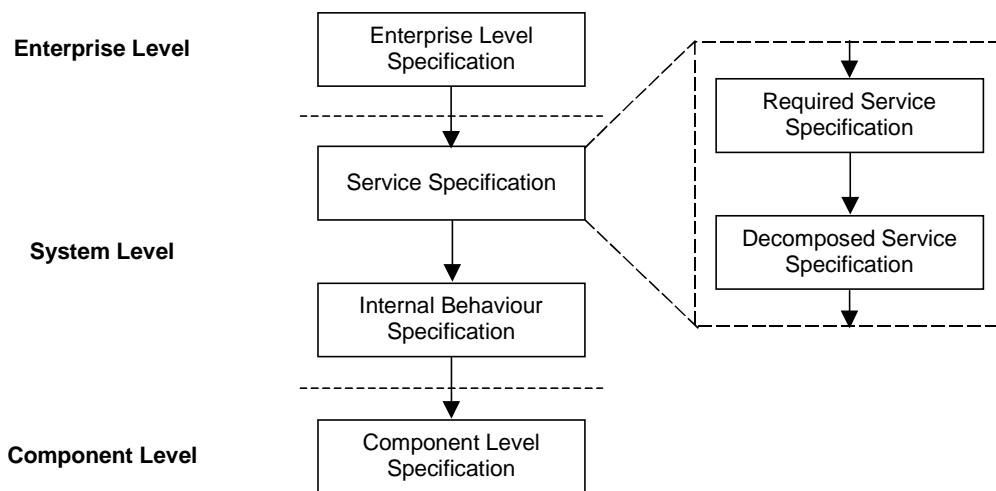


Figure 5-10: Design trajectory at the system level

### 5.3.3 Discussion on the design trajectory

According to [QFS+97, Quar98], the required service specification can be developed based on two consecutive steps or milestones, viz., the service definition and the definition of service provider and service users.

The service definition milestone aims at defining the shared boundary between the system and its environment through the definition of the interactions relating these entities and their ordering in an integrated way, i.e., this description abstracts from the different ways in which the responsibilities and constraints for executing the interactions are distributed over the service users and service provider.

The definition of service provider and service users milestone aims at defining the individual interaction contributions of the system (service provider) and its service users for the execution of the interactions previously identified in the service definition milestone.

Given both the characteristics of interactions as defined at the system level and the set of concerns defined by our behavioural and interactional views, we argue that the description of the system required service according to the interactional view is comparable to the service definition milestone, whereas the description of the system according to the behavioural view is comparable to the definition of service provider and service users milestone.

Interactions as defined at the system level have always the same characteristics, involving the (optional) value passing of information from one entity to another. Further, one of the entities participating in the execution of an interaction, the namely entity playing the interaction reactor role, is always willing to execute the interaction, which means that once the entity playing the interaction initiator role starts the execution of the interaction, the interaction will take place.

Since during interactions information can always be passed from the interaction initiator entity to the interaction reactor entity, and the interaction reactor does not restrict the values that can be established during the interaction, an abstract representation of such interactions as provided by the service definition milestone will only omit the roles of interaction initiator and interaction reactor, i.e., we only abstract from which entity is responsible for initiating the interaction.

Additionally, since these responsibilities are partially identified at the enterprise level with the development of an enterprise level specification according to the distributed perspective with role discrimination, it makes little sense to abstract from these responsibilities at the system level as prescribed by the service definition milestone.

Therefore, since the interactional view of the required service specification provides a description of the cooperative behaviour of the system and its service users, in which an interaction is seen as an integrated action between these entities, this description can be compared to the service definition milestone, except for interactions being asymmetrical at the system level of our design methodology.

Figure 5-11 illustrates the difference between the service definition milestone and the interactional view description involving a system and its service user. Both descriptions see interactions as integrated actions. Still, the service definition is more abstract than the interactional view since these integrated actions are modelled as actions by the former, while they are modelled as “directed actions” by the latter.

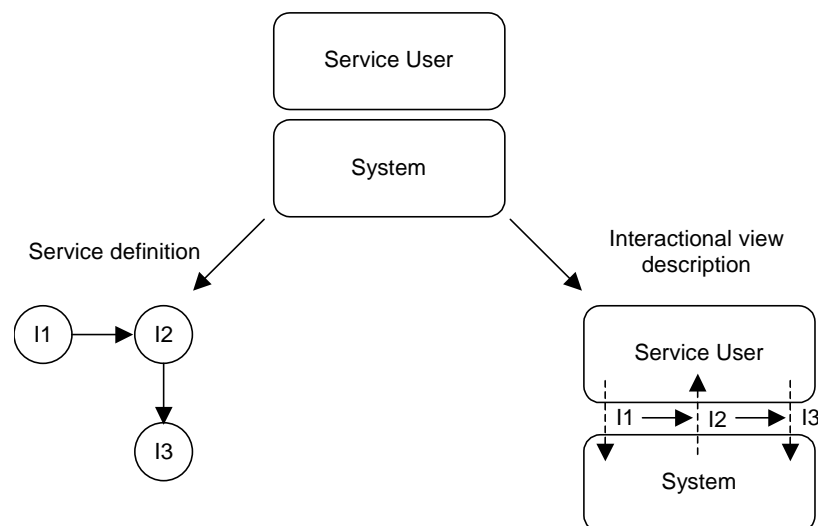


Figure 5-11: Service definition versus interactional view description

Similarly, the definition of service provider and service users milestone can also be compared to the behavioural view description of the required service specification of both the system and its service users since both concentrate on describing the interaction contributions and

their relationships. Nevertheless, the interaction contributions captured in our behavioural view follow the characteristics of interactions as defined at the system level and therefore they are more concrete than the interaction contributions described in the definition of service provider and service users milestone, which permits value passing as well as value matching and value generation [VFQ+00].

Figure 5-12 illustrates the difference between the definition of service provider and service users milestone and the behavioural view description involving a system and its service user. Both descriptions are refinements of the previously defined service definition and interactional view description, respectively. Still, the interaction contributions defined along the behavioural view description follows the interaction characteristics as defined at the system level.

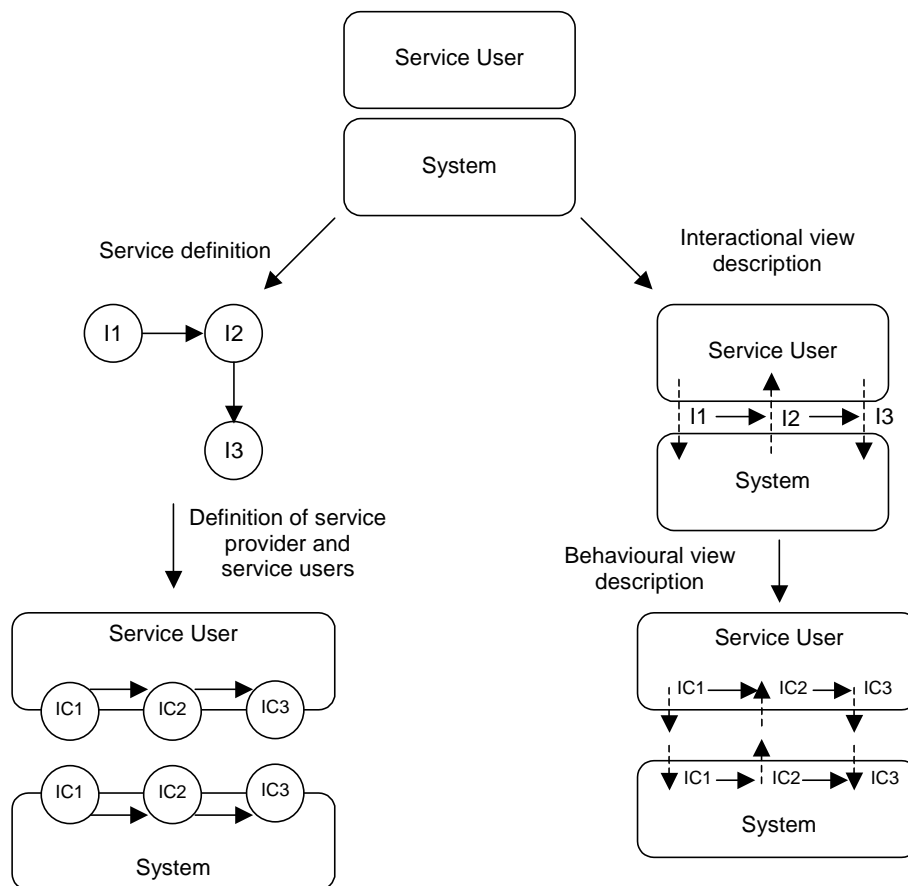


Figure 5-12: Definition of service provider and service users versus behavioural view description

Given the characteristics of interactions at the system level, we also argue that based on the interactional view description of the required service of the system and on the behavioural view description of the system, we can derive the corresponding behavioural view description of the service users.

For example, given two interactions I1 and I2 between two functional entities, viz., a system and its service user, in which according to the interactional view the execution of I1 enables the execution of I2, and both interactions are initiated by the same entity, e.g., the service user. We have in principle two different alternatives for distributing this causality relation between the interaction contributions of these two entities according to the behavioural view: (1) the

system poses no restriction to the execution of the behaviour associated with the interactions, while the service user restricts the execution of the behaviour associated with I2 based on the occurrence of I1 and (2) both the system and the service user restrict the execution of the behaviour associated with I2 based on the occurrence of I1 (see Figure 5-13 for illustration).

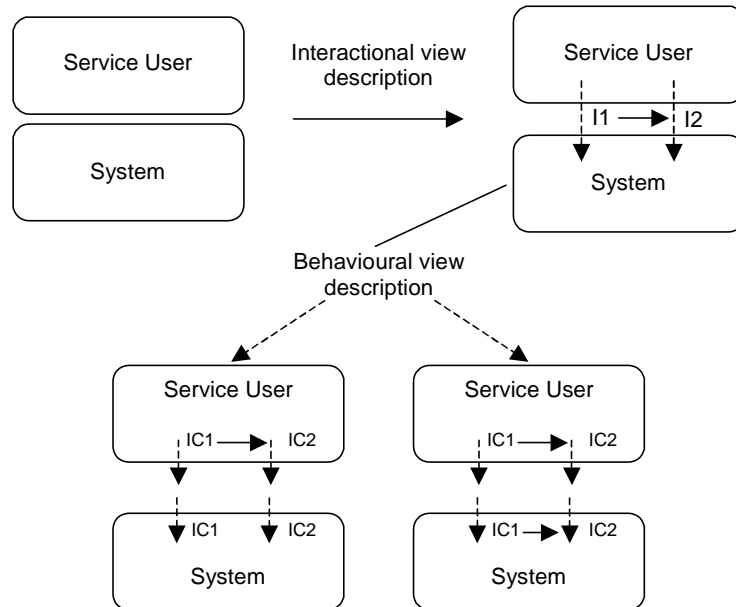


Figure 5-13: Distribution of causality relation

However, given that the functional entity playing the role of interaction reactor, in this case the system, is always ready to interact, the existence of a relationship between the execution of I1 and I2 actually constrains the order in which the entities playing the role of interaction initiator can execute the interaction (alternative 1). Conversely, from the point of view of the entities playing the role of interaction reactor, the existence of such relationship actually prescribes the desirable or expected order in which the interactions should take place. Although in practice the system would be ready to execute both interactions in any order.

Thus, given that an interaction is asymmetric and one entity cannot prevent an interaction from being executed once the entity playing the role of interaction initiator starts it, and given an interactional view description involving two entities, we can mirror the behavioural view description of the system, i.e., the definition of the interaction contributions and their ordering as provided by the system, and come up with the definition of the interaction contributions and their ordering as provided by its service users.

## 5.4 Service specification

In order to specify the system service we need to identify the interactions in which the system is involved and the relationships between these interactions. The difference between developing the required service specification and the decomposed service specification is that in the former case we abstract from the interactions between the system and its auxiliary entities and in the latter case we do not

Consider, for example, the sequence of interactions as depicted in Figure 5-14, in which the occurrence of interaction Int A is followed by the occurrence of Int B, which is followed by the occurrence of Int C. This description corresponds to the specification of the system decom-

posed service. However, in order to specify the required service of the system we would capture the relationship between the occurrences of interactions Int A and Int C, in which the occurrence of Int A is followed by the occurrence of Int C.

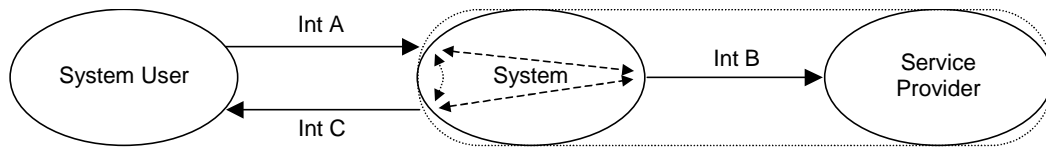


Figure 5-14: Required service versus decomposed required service perspectives

Therefore, the techniques used to model a system according to both perspectives are the same. The only difference resides on the application of these techniques according to the perspective being specified. Having this difference in mind, we discuss only how to model the system according to the decomposed required service perspective, since this perspective is potentially more complex than the required service perspective because it usually involves a larger number of entities and their interactions.

### 5.4.1 Modelling strategy

#### General issues

The development of the decomposed service specification is carried out based on the structural, behavioural and interactional views. These views help us focus on the different subsets of concerns defined by the applied perspective, i.e., the decomposed required service perspective.

Figure 5-15 illustrates abstractly the intersection of the concerns defined by the different views versus the concerns defined by the decomposed required service perspective.

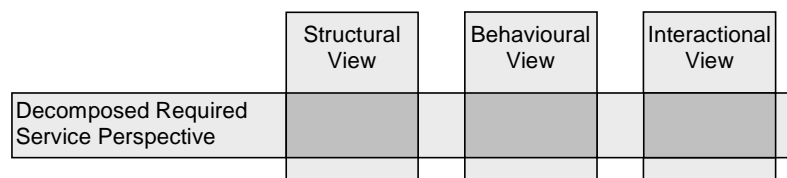


Figure 5-15: Intersection of views and the decomposed required service perspective concerns

The set of interactions in which the system is involved defines the system structure. We identify both the interactions supported by the system, i.e., interactions in which the system plays the reactor role, and the interactions initiated by the system, i.e., interactions in which the system plays the initiator role.

The identification of interactions corresponds to the identification of interface operations. In order to capture the structural view, we basically have to identify operations, group these operations into interfaces and establish dependency relationships from the system environment to the interfaces supported by the system and vice-versa.

The identification of relationships between interactions corresponds to a description of the system according to the behavioural and interactional views. When we take into account the interaction contributions of a single entity, we have a description according to the behavioural view. However, when we take into account interactions as a whole, i.e., interactions as integrated activities between entities, we have a description according to the interactional view.

To identify relationships between interactions, we basically have to capture the order in which interactions can take place. Possibly, we also have to identify relationships between the information exchanged during these interactions. The modelling of such ordering could in principle be carried out using different specification styles [Sind95]. However, due to the usual complexity involved and due to the modelling capabilities of UML we believe that a state-oriented specification style is the most appropriate specification style to this concern level. Therefore, we prescribe the order in which interactions should take place according to the state of the system.

### Techniques and road map

We prescribe the development of the decomposed service specification in three major steps. In the first step, the required service specification is refined and a number of interactions between the system and its auxiliary entities are identified (structural view refinement). In the second step, the ordering between the interactions previously identified in the service specification and the interactions newly identified is established (interactional view refinement). In this step, the resulting interactional view description includes the required service specification provided by the auxiliary entities to the system. In the third step, the interaction contributions of the system and their ordering are identified. Additionally, the interaction contributions of the auxiliary entities and their ordering are identified as well (behavioural view refinement). In the second and third steps, value dependencies are identified as well.

Figure 5-16 lists the steps towards the development of the decomposed service specification.

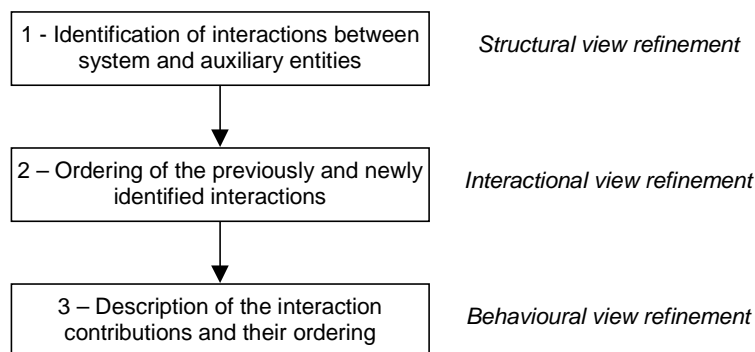


Figure 5-16: Steps for the development of the decomposed service specification

The structural view is captured using use case, package and system interface diagrams. Use case diagrams are used to capture the system requirements; package diagrams are used to capture the static relationship between the system and its environment; and system interface diagrams are used to define the interfaces of the system and its environment and to capture interface dependencies. Additionally, a glossary is used to document the terms identified in the use case, package and interface diagrams.

The behavioural view of an entity (either the system or its auxiliary entities) is captured using primarily a mix of statechart diagrams and activity diagrams. Statechart diagrams are used to capture the order in which the interactions (interaction contributions) supported by the entity should occur. Activity diagrams are used to relate these interactions to the interactions (interaction contributions) initiated by the entity in response. Additionally, these diagrams can be used to capture the relationship between the information exchanged during these interactions.

The behavioural view of an entity is complemented through the specification of the interface operations supported by this entity. Interface operation specifications are used to capture the relationship between the information exchanged between two interactions that form an invocation interaction pattern. Further, such specifications are also used to document how these interactions affect the state of an entity.

The interactional view is captured primarily using system interaction diagrams. Interaction diagrams are used to capture the ordering of the interactions. The textual description of use cases can also be used to capture the interactional view. Therefore, despite that a use case diagram is considered by many a UML behavioural diagram, we consider its graphical form as a representation of the structural view of a system, whereas we consider other forms of representation, such as a textual form, as a representation of the interactional view and possibly the behavioural view (see section 5.5.2).

Table 5-2 summarises the techniques used in the development of the system decomposed service specification.

Technique	Structural View	Behavioural View	Interactional View
<i>Use Case Diagram</i>	applicable	not applicable	applicable
<i>Package Diagram</i>	applicable	not applicable	not applicable
<i>Interface Diagram</i>	applicable	not applicable	not applicable
<i>Glossary of Terms</i>	applicable	not applicable	not applicable
<i>Statechart Diagram</i>	not applicable	applicable	not applicable
<i>Activity Diagram</i>	not applicable	applicable	not applicable
<i>Interface Operation Specification</i>	not applicable	applicable	not applicable
<i>System Interaction Diagram</i>	not applicable	not applicable	applicable

*Table 5-2: System service specification techniques*

There are several alternative ways in which we can apply these techniques to produce a service specification. However, we suggest a straightforward road map based on the steps previously identified, starting with the elaboration of use case diagrams and culminating with the elaboration of the glossary.

Figure 5-17 depicts our suggested road map for specifying the decomposed required system service. Each box corresponds to a certain type of diagram or specification. Arrows indicate the order in which the diagrams should be developed. Boxes horizontally placed besides each other indicate that the associated design activity should be carried out in parallel. Alternatively the elaboration of a glossary could be carried out in parallel with the whole design process, instead of at the end of this process. In the sequel we discuss the application of each of these techniques separately.

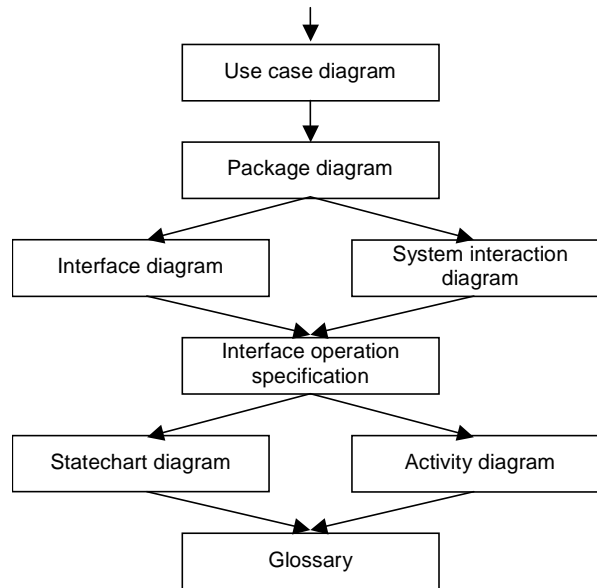


Figure 5-17: System decomposed service specification road map

### Running example

We illustrate the use of each technique using a simple running example. We present the required service specification of a question and answer collaboration support system, QA-Service for short. The QA-Service is quite simple: once users have logged into the system they can ask and answer questions to and from one another. A question is specific to a user, called responder. Similarly, an answer is specific to the user that asked the question, called inquirer. Both a question and an answer can only be delivered to a responder and an inquirer, respectively, in case they are logged. A question cannot be answered twice. However, an inquirer can ask the same question or another question multiple times to the same responder, until a response is provided.

#### 5.4.2 Use case diagram

The elaboration of UML use case diagrams serves two main purposes, viz., to establish the context of the subject entity and to capture its requirements. The context of the subject entity is established through the identification of the entities that belong to its environment and interact with the subject entity. The requirements are captured through the description of these interactions and their associated behaviour.

The aforementioned purposes are directly related to the main constructs in a use case diagram, viz., actor and use case. An actor represents a coherent set of roles that an associated entity can play while interacting with the system. A use case is a unit of behaviour comprising sequences of activities that the system performs to produce an observable result to one or more actors.

The elaboration of a use case diagram can be carried out in four steps: (1) identification of actors; (2) identification of use cases; (3) establishment of relationships between actors and use cases; and (4) textual description of the use cases. There is no need to capture all the information in a single diagram. One may use separate use case diagrams to avoid too much detail in a single diagram.

At the system level, an actor represents either a service user or an auxiliary entity. Actors are identified based on the functional roles played by the functional entities that interact with the functional entity representing the subject entity.

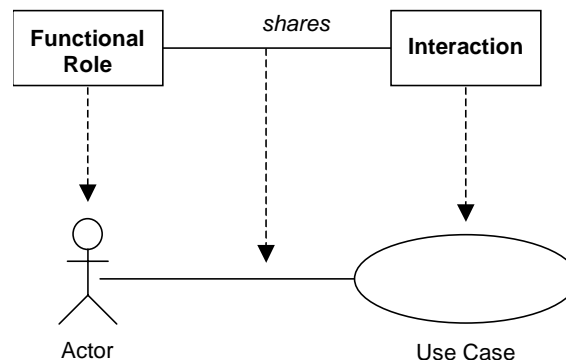
An actor can correspond to a single functional role or to several closely related functional roles. Generalisation/specialisation relationships should be established between similar actors. The relationships established between functional roles at the reference enterprise level specification should be used as indications of how to establish the relationships between the identified actors.

Use cases are primarily identified based on activity units, primarily interactions, identified in the reference enterprise level specification. A use case defines a scenario in terms of interactions between the system and one or more actors. A use case may correspond to a single interaction or to multiple interactions. If an interaction is likely to have a correspondence in more than one use case, this may serve as an indication that this interaction is still quite abstract and therefore should be refined.

In the development of a decomposed service specification, we should only identify use cases that can be directly associated with actors, minimising relationships between use cases. The establishment of relationships between use cases is discussed in section 5.5.2.

The establishment of associations between actors and use cases is straightforward. We only associate an actor with a use case in case one of the functional roles corresponding to the actor shares at least one of the interactions corresponding to the use case.

Figure 5-18 shows the relationship between enterprise concepts and their corresponding representation in a use case diagram. Basically, a functional role corresponds to an actor and an activity unit to a use case. The relationship in which the functional role shares an interaction with the functional role(s) corresponding to the system corresponds to an association between the corresponding actor and use case.



*Figure 5-18: Correspondence between concepts*

Figure 5-19 shows the use case diagram for the QA-Service. Three use cases, viz., Control Presence, Pose Question and Answer Question, as well as a single actor, viz., Question Answer User, are identified.

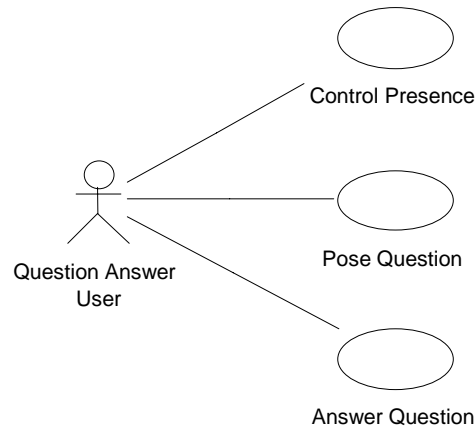


Figure 5-19: Use case diagram for QA-Service

Finally, each actor and each use case portrayed in the use case diagram under development have to be described. The description of an actor is straightforward and involves just the description of the functional roles that correspond to the actor.

UML does not prescribe how a use case should be described. Therefore, we suggest the following convention inspired by a proposal presented in [Larm97]: use cases should be described informally using plain text and for each use case we should provide an use case identification, the corresponding enterprise activity units, the use case purpose, the associated actors and the detailed description.

Figure 5-20 presents the suggested convention for the textual description of a use case in a table-like format.

<b>Use Case:</b>	Name of the use case.				
<b>Enterprise Activities:</b>	Name of the corresponding activity units at the enterprise level.				
<b>Purpose:</b>	Short description of the purpose of the use case.				
<b>Associated Actors:</b>	List of actors indicating who interacts with the use case.				
<b>Detailed description:</b>	Detailed account of the interactions initiated by the actors and the corresponding system responses.				
<table style="width: 100%; border: none;"> <tr> <td style="text-align: center;"><b>External action</b></td> <td style="text-align: center;"><b>Entity response</b></td> </tr> <tr> <td style="text-align: center;">Numbered interactions initiated by the actors.</td> <td style="text-align: center;">Numbered description of entity responses.</td> </tr> </table>		<b>External action</b>	<b>Entity response</b>	Numbered interactions initiated by the actors.	Numbered description of entity responses.
<b>External action</b>	<b>Entity response</b>				
Numbered interactions initiated by the actors.	Numbered description of entity responses.				

Figure 5-20: Use case description format

The use case detailed description should provide a numbered account of the interactions initiated by the associated actors and the corresponding subject entity response. This suggests a refinement of an enterprise interaction into multiple system level interactions and possibly actions that are carried out by the subject entity. Thus, the subject entity response can be provided in terms of both internal actions performed by the subject entity exclusively and interactions that are initiated by the subject entity. However, in the development of the decomposed service specification, we abstract from internal actions.

There can be independent interactions in one use case, i.e., the behaviour of a use case can be initiated in many different ways. Further, each interaction should be described in a typical or

normal course of events, which does not preclude the description of alternative courses of event either.

Each interaction initiated by an actor can be independently subjected to preconditions and postconditions. A precondition defines a condition or constraint that must be true before the interaction is executed, guaranteeing in this way the successful execution of the corresponding behaviour. A postcondition defines a condition or constraint that must be true after the interaction is executed. The definition of preconditions and postconditions is optional for use case diagrams.

Figure 5-21 depicts the description using our suggested notation for the Answer Question use case from the QA-Service example. This example contains a normal course of action, in which an answer to a previously asked question is provided, and an alternative course of action, in which an answer is provided to a non-existing question.

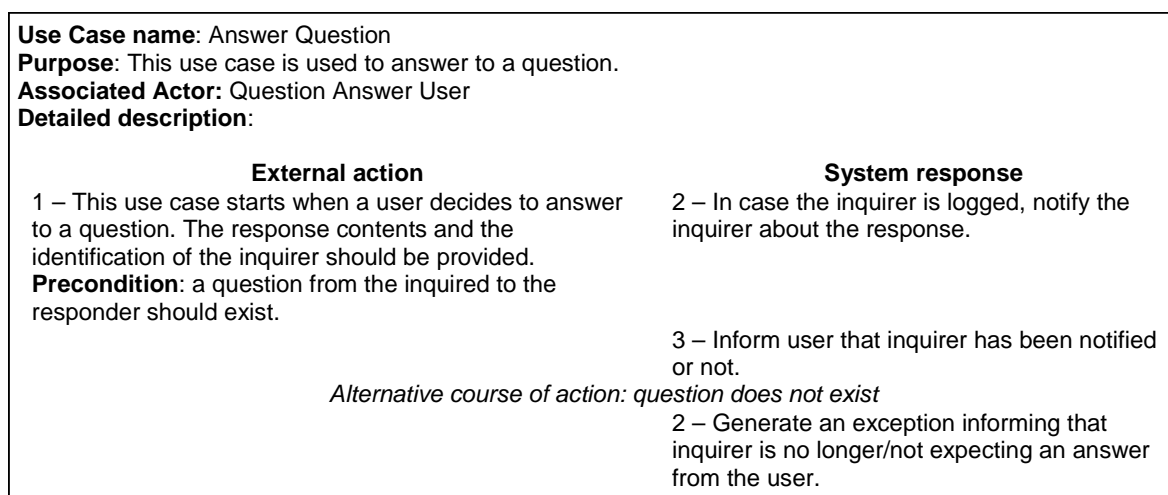


Figure 5-21: QA-Service use case description

### 5.4.3 Package diagram

The elaboration of a package diagram aims at capturing the static relationship between the subject entity and its environment, and helping structuring the design or specification. A package diagram consists of a UML class diagram containing packages and dependency relationships between packages.

The elaboration of a package diagram can be carried out in two steps: (1) the identification of entities; and (2) the establishment of dependency relationships between these entities.

A package in a package diagram represents a functional entity or group of functional entities at the system level. Each actor identified at the use case diagram usually corresponds to a separate functional entity. However, two or more actors can also correspond to the same functional entity, provided that these actors are related to each other via a generalisation/specialisation relationship. The functional entity representing the system itself should also be portrayed as a separate package in the diagram. Each identified package should be labelled with the system structure stereotype, <<system>>, to indicate that the element being stereotyped represents a separate functional entity at the system level.

Dependency relationships should be established between the identified packages. In this context, a dependency relationship between two packages indicates that the functional entities they represent interact with each other.

Figure 5-22 illustrates the package diagram for the QA-Service. Two packages, viz., QA User and QA Collaboration Support System, are identified. These packages correspond to the functional entities representing the system (service) users and the system (service) itself, respectively.

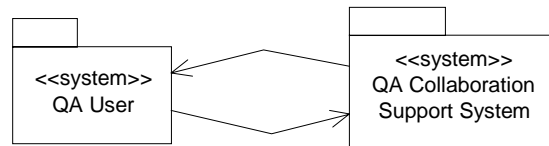


Figure 5-22: Package diagram for QA-Service

#### 5.4.4 System interaction diagram

The first step towards modelling the behaviour of the subject entity is to capture the interactions with its environment. This task is accomplished through the elaboration of a series of system interaction diagrams (either system sequence or system collaboration diagram).

A system interaction diagram consists of a UML interaction diagram in which the interacting elements represent functional entities at the system level. This diagram is similar to what Hruby describes as a package interaction diagram in [Hrub98].

It is up to the designer to choose between the elaboration of either system sequence diagrams or system collaboration diagrams. The difference between these diagrams resides on the emphasis they put on two different aspects of an interaction. Whilst a sequence diagram emphasises the lifelines of the entities participating in an interaction, a collaboration diagram emphasises how these entities are related to each other.

The elaboration of a (series of) system interaction diagram can be carried out in two major steps: (1) definition of usage scenarios; and (2) identification and ordering of interactions.

A usage scenario describes different situations in which the subject entity is used, including possibly configuration situations. The simplest forms of a usage scenario are the use cases themselves. In this case, a scenario involves the execution of a single use case. However, we suggest the definition of scenarios involving the execution of several use cases simultaneously in order to capture the relationship between interactions defined in different use cases. The more use cases involved, the more complex the scenario becomes. Therefore, we suggest a combination of simple scenarios with some more complex ones.

Examples of scenarios extracted from the QA-Service include:

- 1) User A logs in and asks question to User B, who has not logged in yet.
- 2) User B has logged in. User A asks question to User B and logs off. User B answers the question.
- 3) User A has logged in again and User B answers question.

For each scenario described, we should identify the interactions that take place between the subject entity and its environment and their ordering. Special care should be given to the identification of instances of the interaction patterns. Each instance should be mapped onto a corresponding interface operation afterwards (see section 5.4.5).

Interactions, as well as the information exchanged during an interaction, are identified based on the use case descriptions. These interactions correspond to the interactions identified in the reference enterprise level specification or to refinements thereof.

Each interaction is modelled as a message that is sent from one or more functional entities representing the subject entity environment to the functional entity representing the subject entity itself (or vice-versa). In an interaction diagram, a functional entity is modelled by an object, which actually represents an instance of the functional entity.

A message should contain the identification of the interaction stimulus and optionally some parameters representing the information exchanged. Usually, in order to establish the relationship between the information values exchanged an informational model of the system is necessary (see section 5.4.6).

A message is modelled as an arrow connecting the subject entity with its environment. In case of an invocation interaction pattern, different types of arrows can be used to distinguish between a request interaction and a response interaction. Further, for the sake of convenience, the representation of a response interaction can be omitted in case no information is conveyed together with the associated stimulus. However, this is not possible whenever the response interaction is used for synchronisation.

Figure 5-23 shows an interaction diagram developed for the QA-Service. This diagram represents scenario 3, i.e., the successful response of a question. A constraint specified in OCL (see section 5.4.6) attached to the interaction `answerNotification` is used to relate the information values established during this interaction with previously established information values. In this example, `answer.contents` and `answer.inquirer` refers to the information values established during the interaction `answer`. This notation can be generalised to any activity unit (*activity\_unit.information\_value*).

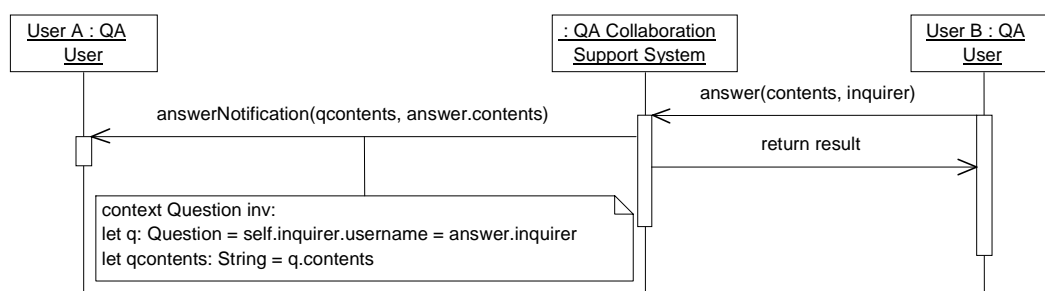


Figure 5-23: Interaction diagram for QA-Service

### Limitations of interaction diagrams

An interaction diagram is the UML diagram most suitable for representing behaviour according to the interactional view of our methodology. Nevertheless, the capabilities offered by this technique are beyond our actual needs for a number of reasons.

Because one of the early motivations for the use of UML interaction diagrams was the representation of a use case [SiGr99], this indirectly created a limitation on the modelling requirements or capabilities of this technique. Interaction diagrams are used basically to represent a scenario involving a limited number of entities exchanging messages according to a possible alternative behaviour that these entities are able to perform. Additionally, we can neither compose nor decompose interaction diagrams, leading to the development of very complex diagrams whenever more complex scenarios are described.

Further, because such diagrams concentrate on the specification of an orderly sequence of messages, it is neither possible to specify alternative sequences nor parallel sequences of messages in a single diagram.

The identification of these shortcomings regarding the modelling of the interactional view using UML interaction diagrams has been corroborated by similar findings elsewhere. In [Wier99], Wieringa proposes the design of software systems based on the identification of the messages that cross the software boundary, the identification of the behavioural properties of these messages (ordering in time) and the identification of the communication properties of these messages (ordering in space), which is roughly equivalent to our interactional view. Wieringa acknowledges the usefulness of UML to represent scenarios as well as its limitations to describe communication properties as a whole, suggesting the use of other non-UML techniques for such purposes.

The limitations intrinsic to interaction diagrams could be softened in case some concepts and constructs similar to those found in another specification technique called Message Sequence Charts (MSC) [RuGG96] were incorporated to this technique. MSC is a scenario description language standardised by ITU-T to describe the order in which interactions between a number of independent message-passing instances described in SDL take place.

Among the facilities provided by MSC to structure behaviour we emphasise the concepts of *coregion* and *submsc*. Along an MSC diagram a total ordering of message events is normally assumed. In this sense, a coregion is used to denote a region within the diagram in which the specified events are not ordered. An MSC diagram can be refined into another MSC diagram, called *submsc*, which represents a decomposition of the former diagram without changes in its observable behaviour. Additionally to the use of coregions and *submscs* to structure behaviour, this specification language allows the composition of simple MSC diagrams to form more complex behaviours.

#### 5.4.5 System interface diagram

The elaboration of a system interface diagram aims at defining the interfaces of the subject entity and its environment, and to capture interface dependencies. An interface diagram consists of a UML class diagram containing functional entities, associated interfaces and dependency relationships between entities and interfaces.

The elaboration of an interface diagram can be carried out in four steps: (1) the representation of functional entities; (2) identification of interfaces; (3) association of interfaces with entities; and (4) establishment of dependency associations between entities and interfaces.

The representation of functional entities is straightforward. The functional entities to be depicted in an interface diagram correspond to the functional entities identified in the reference enterprise level specification and captured in the UML package diagram. Each functional en-

tity or group of functional entities captured in the package diagram should have a corresponding functional entity in the interface diagram. A functional entity in an interface diagram is represented by a class stereotyped with the system stereotype, <<system>>. This stereotype indicates that the associated class represents a functional entity at the system level.

The identification of interfaces can be carried out in two steps: the identification of operations and the grouping of operations into interfaces.

An operation is directly associated with an instance of either the interaction invocation pattern or the notification invocation pattern. For each identified instance of the interaction pattern that a functional entity supports, i.e., support for either a request interaction or a notification interaction, a corresponding operation should be identified. However, the identification of the supported interactions may not be straightforward at this point because we may not know them all beforehand. Instead, this activity should be carried out in parallel with the modelling of the system interaction diagram or afterwards.

After all operations have been identified, they should be grouped into one or more interfaces and described properly (see section 5.4.6). We do not constrain how the operations should be grouped into interfaces. As a rule of good practice, operations representing the invocation pattern and operations representing the notification pattern are usually grouped into separate interfaces because the existing component models already distinguish between different interface “types”, such as operation, signal and stream, according to their purpose. Nevertheless, such a distinction is not relevant in the context of this work.

Ultimately, the designer is responsible for deciding to create a separate interface to hold a subset of operations that have some characteristics in common or not. Nevertheless, an interface should be uniquely identified within its context of use. Generalisation/specialisation relationships can also be established between the identified interfaces to facilitate the distribution and specification of operations.

The next step after describing interfaces is to establish realisation relationships between the interfaces and their associated functional entities. A realisation relationship between a functional entity *F* and an interface *I* indicates that *F* supports the operations described in *I*, i.e., the entity provides some behaviour associated with the operations defined in the interface.

The final step in the elaboration of a system interface diagram is to establish dependency relationships between functional entities and interfaces. A dependency relationship between a functional entity and an interface indicates that this entity either invokes the operations defined by the interface or notifies the occurrence of events via this interface. For each dependency association identified between two packages *A* and *B* (package *A* is dependent on package *B*) in the package diagram, there should be at least a corresponding dependency association between an entity *A* and an interface that is realised by an entity *B*.

Figure 5-24 illustrates the system interface diagram developed for the QA-Service. The QA Collaboration Support System entity realises two functionality interfaces, viz., Logging Interface and QA Interface, while the QA User entity realises a single notification interface, viz., QA Notification Interface.

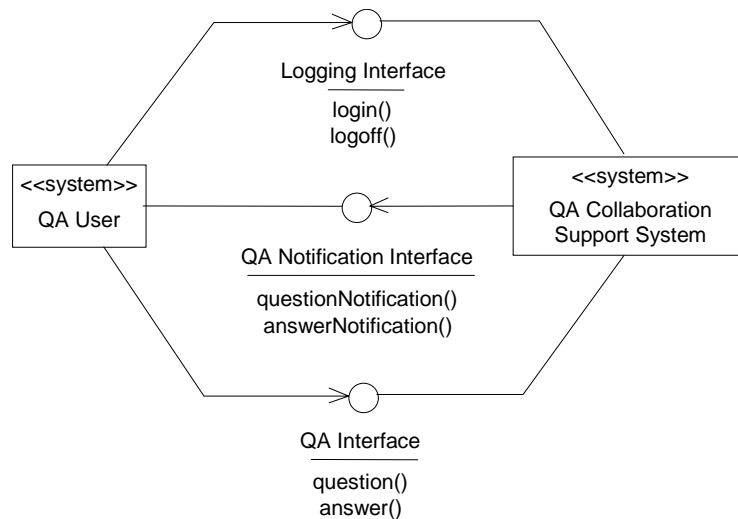


Figure 5-24: System interface diagram for QA-Service

It is also possible to elaborate a variant of the system interface diagram in which we capture only the interfaces associated with a single functional entity. In this variant we include, besides the functional entity, the interfaces that are either realised by this entity or associated with the functional entity in a dependency relationship. In this case, we have to elaborate a separate interface diagram for each functional entity captured in the package diagram.

In principle we could have prescribed the use of UML component diagrams to capture functional entities and their interfaces and interface dependencies. However, the use of stereotyped classes to model a functional entity instead of a component allows us to create interaction system diagrams based on these classes, which is not possible using UML components depicted in component diagrams.

#### 5.4.6 Interface operation specification

All the identified interfaces have to be properly specified through the description of its operations. Similarly to use cases, UML does not prescribe how operations should be described. Therefore, we suggest a convention using plain text, which is described in Figure 5-25.

<b>Operation:</b>	Signature of the operation (operation name, parameters and return type)
<b>Interface:</b>	Name of the interface that the operation belongs to
<b>Purpose:</b>	Short description of the purpose of the operation
<b>Cross references:</b>	Name of the use case(s) associated with this operation
<b>Preconditions:</b>	Description of the preconditions that are associated with the operation
<b>Postconditions:</b>	Description of the postconditions that are associated with the operation
<b>Exceptions:</b>	List of exceptions that may be raised by this operation
<b>Behaviour description:</b>	Name of one or more activity diagrams that describe the actions taken in response to the operation

Figure 5-25: Operation description format

In the context of an operation specification, a precondition represents a constraint that if true guarantees the successful execution of the operation. Consequently, the satisfaction of all preconditions of an operation guarantees that the constraints established by the corresponding postconditions are met. So, a precondition is used to associate an interaction with the current state of the subject entity.

A postcondition is a constraint that must be true after an operation is successfully executed. So, a postcondition is used to describe the changes in the state of the subject entity as a result of the execution of the interaction and possibly a number of actions that follows the occurrence of the interaction.

Preconditions and postconditions can be described informally using natural language or formally using the Object Constraint Language (OCL) [WaK199, OMG01]. OCL is an expression language defined as part of UML to describe constraints on object-oriented models.

The main problem regarding the documentation of an operation in general, and the documentation of pre and postconditions in particular, is the need for an information model for the system. An information model is equivalent to a description of a functional entity according to the information view (see Chapter 3, section 3.3.2). Since the development of models according to this view is not the focus of our research, we do not provide any specific guidelines for undertaking such a task, except for the comments in the sequel.

UML offers a lot of support for the modelling of structural and behavioural aspects. However, UML does not provide explicit support for data modelling. One can model datatypes using a class diagram, in which classes represent datatypes and associations between classes represent relationships between these datatypes. Such a diagram can be regarded as a sort of concept diagram, in which we do not have an explicit distinction between behaviour and information. In order to create such a distinction, we attach the stereotype `<<datatype>>` to each class captured in the diagram to indicate that the concept represents a datatype.

Figure 5-26 shows the informational model developed for the QA-Service. Three datatypes, viz., QAService, User and Question, are identified and their relationship established. These datatypes represent information related to the QA-Service, QA-Service users and questions, respectively.

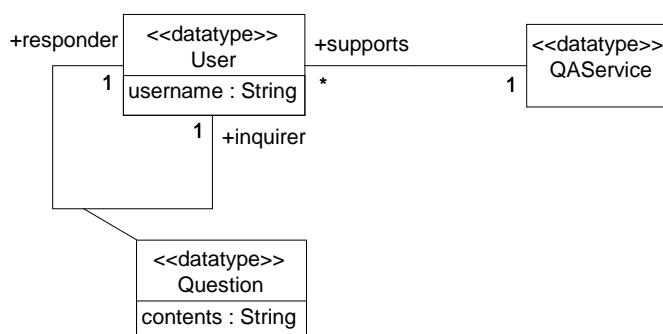


Figure 5-26: Informational model for QA-Service

Other non-UML languages, more suitable for data modelling, could also be used to produce information models of the subject entity. However, the biggest advantage of using UML for modelling data lies on the possibility of using OCL to describe pre and postconditions formally.

OCL is a typed language and provides a number of pre-defined datatypes, such as integer, boolean, set, sequence and bag, as well as a number of operations that are associated with these types. Further, since OCL models are written in the context of a UML model, the types of the classifiers defined in this model can be used as types in OCL as well.

To a lesser extent, the need for an information model persists even if we model only the operation parameter types. For the purpose of specifying the parameter type, we only need to describe the data carrier associated with the type. The description of operations on the datatype is not necessary, which enables us to use a standard interface description language, such as OMG IDL for such purpose.

Because of the characteristics of interactions, in most cases it is impossible to prevent entities from invoking an operation, even though the preconditions for the invocation of the operation may not be true. In such cases, the execution of the associated system functionality is likely to raise an exception. Therefore, the specification of an operation should also include a list of exceptions that may be raised as a result of the invocation of the operation.

The behaviour associated with the invocation of an operation is described by different diagrams, according to the perspective being specified. So, the identification of the diagrams that describe this behaviour should be added accordingly.

Figure 5-27 shows the main elements of the specification of the operation answer. Preconditions and postconditions are described informally as comments (--) and formally using OCL.

<b>Operation:</b>	boolean answer(in contents: String, in inquirer: String)
<b>Interface:</b>	QA Interface
<b>Preconditions:</b>	-- question from inquirer to responder must exist if (QAService.supports->exists(u : User   u.username = login.username)) u.question[responder]->exists(q: Question   q.inquirer = inquirer) endif
<b>Postconditions:</b>	-- if inquirer is logged result = true if (QAService.supports->exists(u : User   u.username = inquirer)) result = true -- and question no longer exists not u.question[responder]->exists(q: Question   q.responder = login.username and q.inquirer = inquirer) -- else result = false else result = false endif
<b>Exceptions:</b>	QuestionNotDefined

Figure 5-27: QA Interface operation specification

### 5.4.7 Statechart diagram

The elaboration of UML statechart diagrams aims at capturing the order in which the interactions both supported and initiated by the subject entity can take place.

### Modelling requirements and approach

In order to model the order in which interactions can take place, we need to be able to relate two or more interactions in various ways. For example, consider the system and its associated

interactions as shown in Figure 5-28, which consists of four interactions: I1, which is supported by the subject entity and I2, I3 and I4, which are initiated by the subject entity (the direction of the arrows indicate the entity responsible for initiating the interaction). If interactions I2, I3 and I4 take place as a result of the occurrence of I1, we have many different possibilities for the ordering of these interactions, e.g., I3 followed by I4 followed by I2, I3 followed by I2 or I4 followed by I2, I3 in parallel with I4, both followed by I2, to name just a few possibilities.

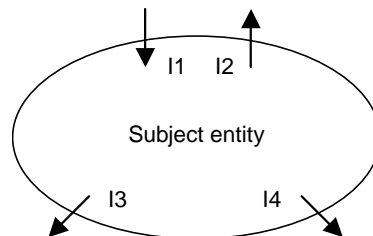


Figure 5-28: Subject entity and associated interactions

The modelling of the ordering of interactions is best accomplished in UML using statechart diagrams. A UML statechart diagram allows us to model that the ordering of the interactions supported and initiated by the subject entity is dependent on the state or states of the subject entity and on a number of transitions connecting these states.

Typically, a statechart diagram consists of a number of states and transitions connecting these states. A transition, which may be guarded by a specific condition, is triggered by the occurrence of an event, which in the context of this work may indicate either the satisfaction of a condition or the elapse of some period of time, which from the point of view of an external observer can be seen as an spontaneous transition, or the reception of an interaction stimulus, either a request interaction or a notification interaction.

According to the UML semantics, during a transition, a number of actions can be executed by the subject entity atomically, i.e., all the actions specified should be executed sequentially within a single thread of execution, i.e., without interruption, in order to complete the transition. In the context of this work and purpose of this diagram, an action may represent the generation of a stimulus representing a request interaction, a notification interaction, a response interaction (including exception) and an instance of the invocation interaction pattern.

Figure 5-29 illustrates the basic elements of a statechart diagram. After the occurrence of the event  $Event_1$ , actions  $Act_1$ ,  $Act_2$ , up to  $Act_N$  are sequentially executed during the transition as a single action.

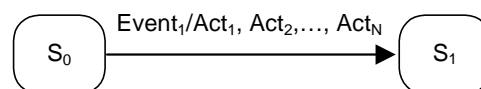


Figure 5-29: UML statechart diagram basic elements

The basic problem with the semantics of a UML statechart diagram is the low-level granularity of behaviour assumed with respect to the intended use of such a diagram. According to UML, a statechart diagram is used to model the behaviour of simple modelling elements, such as an object, which typically run in a single processor.

The only form of concurrency allowed is through the use of a composite state containing concurrent sub regions. This form of concurrency enables us to model at most that the occurrence of an event enables a transition in one or more concurrent regions at the same time. However, the semantics associated with the transition itself remains the same.

Considering the subject entity as a large unit of behaviour where typically multiple threads can execute in parallel with each other, possibly in different processing units, though this distribution of behaviour is transparent, it is fair to assume that the actions, representing interactions, that follow the occurrence of an event may actually be performed in parallel. This assumption is corroborated by the elaboration of even a simple UML activity diagram in which the occurrence of an action may be followed by the parallel occurrence of two or more actions (behaviour forking).

The semantics of a transition does not allow us either to specify conditions on the occurrence of individual actions or sequences of actions, which would enable us to specify alternative sequences of actions that can all be executed as consequence of the occurrence of the same event. This assumption is also corroborated by the elaboration of a simple UML activity diagram in which the occurrence of an action may be followed by the conditional occurrence of one action (sequence of actions) among multiple alternative actions.

We can work around this restriction quite easily by specifying alternative transitions of the same event, simply guarded by different conditions, which may lead to the execution of different sequences of actions. However, this typically increases the complexity of a diagram.

Based on these requirements and constraints, we propose a combined approach to capture the order in which interactions take place. First, we capture the order in which the interactions supported by the subject entity can occur, based on its state, using a statechart diagram. Then, we complement this diagram with a number of activity diagrams to capture the order in which the interactions initiated by the subject entity during a state transition can occur.

Figure 5-30 illustrates our combined approach for modelling the order in which the interactions both supported and initiated by the subject entity can take place. The reception of an interaction stimulus is modelled as an event in a statechart diagram, whose occurrence triggers a state transition ( $Int_1$ ). During this transition a number of interactions are executed ( $Int_2$  and  $Int_3$ ). The interaction that triggers a state transition and the interactions that are executed during the transition itself are modelled as activities in an activity diagram and their ordering established. In the example illustrated in Figure 5-30, the interactions  $Int_2$  and  $Int_3$  can occur in parallel.

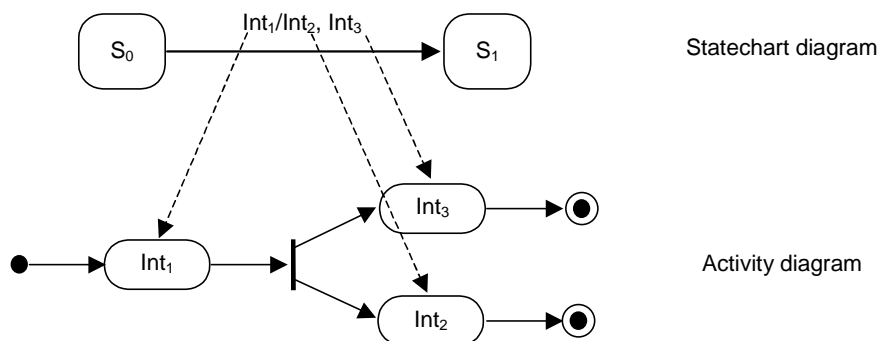


Figure 5-30: Combined approach for interaction order modelling

In principle, if the interactions initiated by the subject entity during a state transition are executed sequentially, we can capture the order of these interactions simply using a statechart diagram; there is no need to develop a separate activity diagram to capture the order in which the interactions take place. However, since these activity diagrams form the basis for capturing the subject entity internal behaviour (see section 5.5.4), we prescribe their use always.

### Modelling strategy

The strategy for elaborating a statechart diagram is the following: to model each interaction supported by the system, as well as each internal event that leads to a change in the state of the subject entity, as an event that triggers a transition in a statechart diagram.

Because the elaboration of a single statechart diagram for a system that exhibits a complex behaviour may result in a diagram that is too complex and possibly unintelligible, multiple complementary statechart diagrams can be developed alternatively.

In this sense, the elaboration of a statechart diagram can be carried out in three steps: (1) identification of states; (2) identification of the events associated with each state; and (3) association of events with state transitions.

A state is a situation during the lifetime of the subject entity, in which the subject entity satisfies some conditions, executes some activities and waits for an operation invocation or an event notification. Similarly to an activity diagram, a statechart diagram has also initial and final states. In order to identify the states of a subject entity, we should identify what conditions must be satisfied so that the subject entity can support certain events and group these conditions in a state. These conditions can be identified based on use case descriptions or on interface operation specifications.

For each state identified, we should identify the events that trigger a transition from one state to another or possibly to the same state, and add a transition for each of these events. Events can be identified based on use case descriptions (interactions and internal events) or on interaction diagrams (interactions only). Preconditions involving the occurrence of an event can be modelled as guard conditions in the transition. Each transition should be properly documented using an activity diagram (see section 5.4.8).

Figure 5-31 shows the statechart diagram developed for the QA-Service. Only two states are identified, viz., Inactive and Active. Transitions representing the interactions supported by the QA-Service connect these states. Guard conditions are not explicitly depicted because they are already described in the corresponding operation specification.

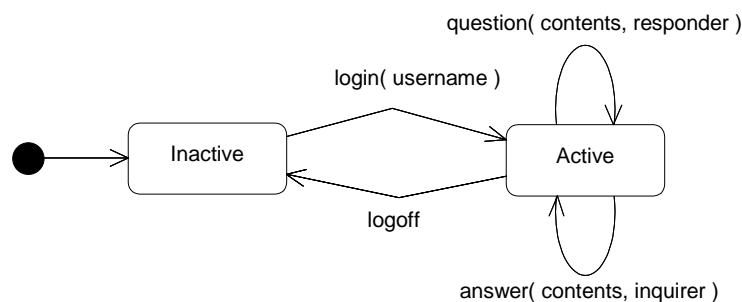


Figure 5-31: QA-Service statechart diagram

### 5.4.8 Activity diagram

The elaboration of a UML activity diagram aims at capturing, for each interaction that the subject entity supports, the behaviour associated with this interaction in terms of the order in which other interactions are executed as a result. Therefore, the elaboration of an activity diagram aims at capturing the order in which the interactions executed during a state transition take place.

For each interaction that the subject entity supports, captured in the statechart diagram(s), a separate activity diagram should be provided to model the subject entity's reaction to its associated operation invocation or event notification.

#### Modelling requirements and conventions

Because we are modelling the reaction of the subject entity in terms of its external behaviour, whenever a given interaction occurs, we are actually modelling the contribution provided by the subject entity to the occurrence of this interaction and possibly other related interactions in which the subject entity participates (either as an actor or as a reactor) during the course of behaviour that follows the occurrence of the interaction.

In an activity diagram, the execution of an activity unit can be represented either as an action state or as an activity state. Whilst an action state cannot be decomposed (an action state is atomic), an activity state can be further decomposed in a collection of action and activity states. Since an activity unit is atomic in our model, any interaction identified should be modelled as an action state, or simply as an action, in an activity diagram.

In order to model an interaction contribution in an activity diagram, we basically have to be able to model either the interaction stimulus generation or the interaction stimulus reception. The modelling of the interaction stimulus generation is straightforward using an UML activity diagram. The UML semantics for an action distinguishes between different kinds of actions, a.o., a send action, a return action and a call action.

A send action is an action that results in the sending of a signal, i.e., communication of a stimulus between two instances. Therefore, a send action is appropriate to model a request interaction, a notification and a response interaction whenever the response represents the occurrence of an exception.

A return action is an action that results in returning a value to a caller instance. Consequently, a return action is also appropriate to model a response interaction.

A call action is an action that results in the invocation of an operation (request and response) on an instance. So, a call action is appropriate to model an instance of the invocation interaction pattern as a whole, i.e., abstracting from the individual request and response interactions.

The UML semantics for an action does not consider explicitly an action that represents the reception of a stimulus. However, a special type of action is offered, viz., an uninterpreted action, whose semantics is not explicitly defined in UML. Hence, an uninterpreted action is appropriated to model the reception of a request interaction, a notification interaction and the reception plus response of an instance of the invocation interaction pattern. Optionally, the reception of such a stimulus can be replaced by an event connecting two actions, instead of using an uninterpreted action.

Figure 5-32 shows the difference between the use of an uninterpreted action and an event in the modelling of the reception of a request interaction called  $Int_1$ . In Figure 5-32a,  $Int_1$  is modelled as an uninterpreted action, whilst in Figure 5-32b,  $Int_1$  is modelled as an event.

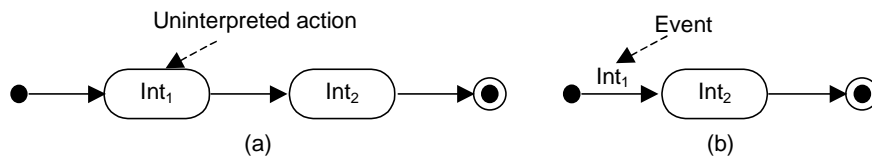


Figure 5-32: Uninterpreted action versus event

Graphically, UML makes no distinction between the different kinds of actions. Therefore, stereotypes should be used to make this distinction explicit as follows: an action representing a request interaction should be labelled with the stereotype `<<requestInteraction>>`; an action representing a response interaction should be labelled with the stereotype `<<responseInteraction>>`; an action representing an exception should be labelled with the stereotype `<<exception>>`; an action representing an instance of the invocation interaction pattern as a whole should be labelled with the stereotype `<<invocationInteractionPattern>>`; finally, an action representing a notification interaction pattern should be labelled with either the stereotype `<<notificationInteractionPattern>>` or with the stereotype `<<notificationInteraction>>`.

### Modelling strategy

The elaboration of an activity diagram can be carried out in two major steps: (1) identification of interaction contributions; and (2) establishment of relationships between the identified interaction contributions.

The first step towards elaborating an activity diagram is to identify all the interaction contributions of the interactions that are both supported and initiated during a state transition. In order to identify the interaction contributions, we have first to identify the corresponding interactions and derive the interaction contributions.

These interactions, especially those performed by the subject entity during the transition being modelled, can be identified based on the use case descriptions and on the scenarios described in the interaction diagrams. An interaction contribution is identified based on the information values passed during the interaction and on the corresponding type of the interaction, e.g., request interaction, notification interaction, etc. We also identify whether or not the subject entity initiates or supports the interaction, although this can be inferred from the context. Each interaction contribution identified should be stereotyped according to the aforementioned conventions.

The first activity of the diagram always corresponds to the event that triggers the state transition being modelled in case this event refers to the occurrence of an interaction. Otherwise, a (guard) condition should connect the diagram initial state to the first activity(ies) representing the corresponding interactions in the diagram.

The execution of the behaviour associated with an instance of the invocation interaction pattern may also raise a number of exceptions, which are not usually captured in the interaction diagrams. An exception consists of an abnormal condition that occurs during the execution of some behaviour. There are several kinds of situations that may lead to the occurrence of an exception, such as the disrespect for preconditions, invalid arguments, etc. The identification of exceptions should take into account what can go wrong during the execution of the transi-

tion, such that the successful completion of the transition is jeopardised. The occurrence of an exception indicates that the transition has not taken place.

The generation of an exception is usually modelled as a stereotyped action, while the reception of an exception is usually modelled as a stereotyped event. Exceptions, however, should be documented in a separate class diagram. We should try to organise the exceptions in a hierarchy, establishing generalisation/specialisation relationships between them and introducing new exceptions as needed. Each exception should be modelled as a separate class, with the exception definition stereotype, <<exception>>, attached to it.

Relationships should then be established between the interaction contributions identified according to the execution logic of the behaviour. The types of relationships that can be established between activity units as a consequence of the use of UML are discussed in Chapter 4.

While relating the interaction contributions associated with a subject entity's state transition, we possibly have to specify the relationships between the values of information exchanged during these interactions. Once again, an information model of the subject entity may be necessary to carry out this task.

Figure 5-33 shows the description of the interaction contributions triggered by the occurrence of the interaction answer and their relationship. A separate activity diagram is developed for each on the transitions depicted in Figure 5-31.

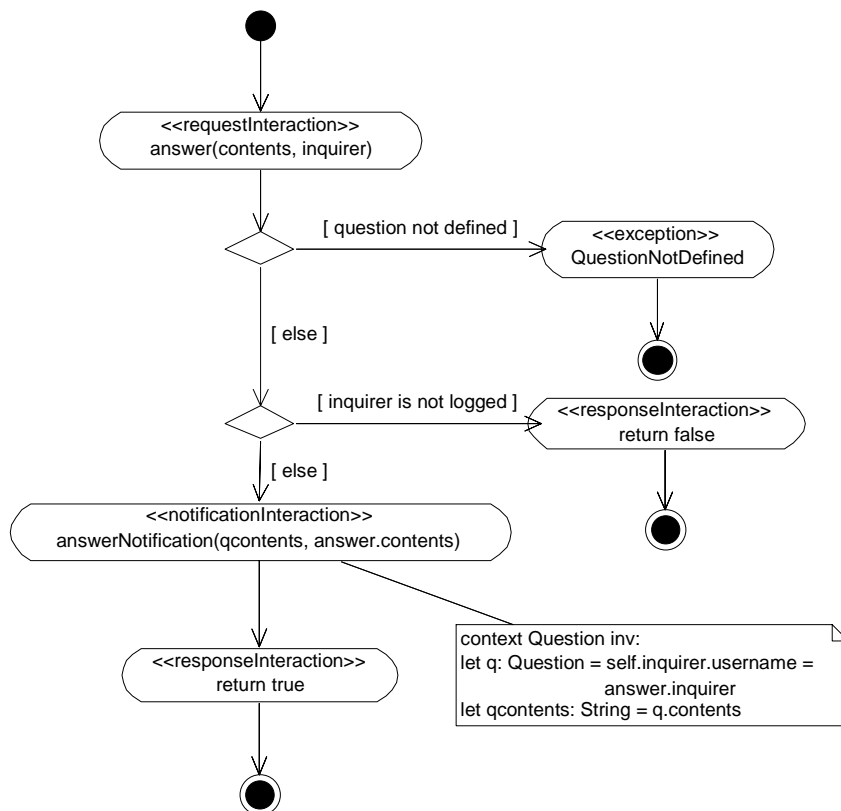


Figure 5-33: QA-Service statechart transition specification

### 5.4.9 Glossary of terms

At the system level, we should document the terms related to use cases, actors, functional entities and interfaces. We do not need to document the instances of the interaction patterns in the glossary because they are best described elsewhere (see section 5.4.6).

Figure 5-34 shows two entries of the QA-Service glossary describing a use case and an interface respectively.

<i>Name</i>	<i>Type</i>	<i>Description</i>	<i>Level</i>
<i>Answer Question</i>	Use Case	Use case used to answer a previously asked question.	System
<i>QAInterface</i>	Interface	Interface used by the QA-Service users to ask questions and answer them.	System

Figure 5-34: QA-Service glossary specification

## 5.5 Internal behaviour specification

In order to model the behaviour of the system according to the integrated internal perspective, we have to refine the behaviour described in the decomposed service specification by inserting a number of actions and their relationships in the original behaviour. These actions are executed together with other interactions during a system state transition.

### 5.5.1 Modelling strategy

#### General issues

The development of the internal behaviour specification is carried out through the refinement of the behavioural view description developed in the decomposed required service specification. There are no changes in the structural view and interactional view descriptions previously developed. Actually, the interactional view remains the same, while some changes that do not affect the structural view description for an external observer can be carried out on the structural view description itself.

Figure 5-35 depicts the intersection of the concerns defined by the different views versus the concerns defined by the decomposed required service and internal integrated perspectives.

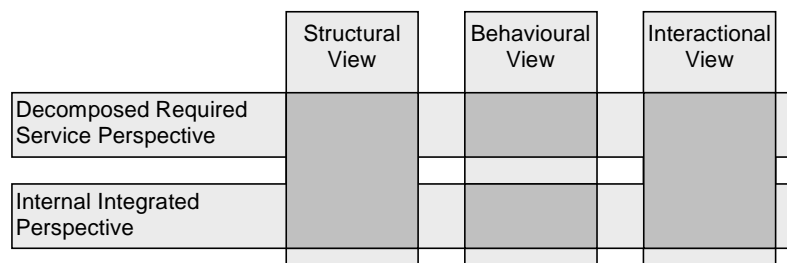


Figure 5-35: Intersection of views and perspectives concerns

#### Techniques and road map

Although there are no changes in the structural view for an external observer, one may need to change how units of behaviour are organised when internal activities are modelled. These

changes are only visible in use case diagrams, through the introduction of new use cases as necessary. Further, any new use case identified should be documented as a new term in the glossary.

The behavioural view is captured primarily through the refinement of the activity diagrams previously identified in the decomposed service specification. Activity diagrams are used to capture the behaviour of the system associated with the system state transitions, triggered mainly by the occurrence of interactions supported by the system. The additional use case descriptions are also part of the behavioural view.

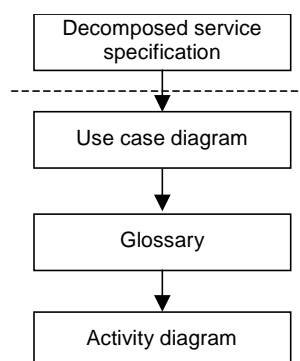
Table 5-3 summarises the techniques used in the development of the system internal behaviour specification.

Technique	Structural View	Behavioural View	Interactional View
<i>Use Case Diagram</i>	applicable	applicable	applicable
<i>Glossary of Terms</i>	applicable	not applicable	not applicable
<i>Activity Diagram</i>	not applicable	applicable	not applicable

*Table 5-3: System internal integrated behaviour specification techniques*

Modelling the internal integrated behaviour of a system is straightforward. Starting with the decomposed service specification, we augment the existing use case diagram to include actions, adding new use cases as necessary. Then, each new use case should be documented and the glossary should be updated. Finally, new activity diagrams are created to capture the relationship between actions and interactions.

Figure 5-36 depicts our suggested road map for specifying the system internal integrated behaviour. Each box corresponds to the elaboration of a certain type of diagram or specification. Arrows indicate the order in which the design activities should be carried out. In the sequel, we discuss the application of each of these techniques separately.



*Figure 5-36: System internal integrated behaviour specification road map*

### 5.5.2 Use case diagram

The elaboration of a use case diagram to model the subject entity according to the integrated internal perspective aims at showing a more detailed description of the subject entity internal behaviour as far as the support of interactions is concerned.

The use case diagram(s) used in the description of the subject entity decomposed service serves as our starting point. In this sense, such diagram(s) can be further elaborated according to two basic steps: (5) abstract description of internal activities; and (6) factoring of behaviour.

To continue the elaboration of a use case diagram, we first need to add additional information about the subject entity behaviour. At this point, we should not only reveal the end result that the execution of the use case functionality provides, but also abstractly describe a number of actions that the subject entity carries out in order to achieve this result. One should not be concerned about describing these actions thoroughly because such textual descriptions may not always be appropriate.

While describing a use case in further detail, we should try to identify new use cases via two factoring techniques [BoRJ98]. First, we should try to extract some common behaviour from one or more use cases and allocate this behaviour into new use cases, which can be used by the other use cases (establishment of an <<include>> relationship). Then, we should try to extract some variant behaviour from a use case and create a new use case that incorporates the original behaviour and extends this behaviour to meet specific conditions (establishment of an <<extend>> relationship).

Although it looks trivial, the decision on whether or not to establish one of these types of relationship between two use cases is one of the main sources of errors and confusion while modelling use case diagrams [Vaug01]. Such confusion steams from a lack of precise semantics for these relationships, as argued in, e.g., [SiGr99].

### 5.5.3 Glossary of terms

The glossary should be updated whenever a new use case is identified.

### 5.5.4 Activity diagram

The elaboration of UML activity diagrams aims at detailing the behaviour associated with state transitions. For each activity diagram developed at the decomposed required service specification, a new refined activity diagram should be developed to replace the previous description.

The refinement of an activity diagram can be carried out in two additional steps: (3) identification of actions; and (4) establishment of relationships between the previously identified interaction contributions and the newly identified actions.

The identification of the internal activity units, i.e., actions, is based on the use case descriptions and on the subject entity requirements as a whole. These actions are inserted as a result of the refinement of the existing causality relations between the different interaction contributions. These actions also appear in the detailed description of use cases at this level. If the behaviour description of a use case includes the behaviour of another use case (include relationship), the actions of the included use case should also be taken into account accordingly.

Relationships should be established between the previously identified activity units and the newly identified activity units according to the execution logic of the behaviour. However, the

new behaviour description should preserve the order in which the previously described interactions take place (refinement conformance).

## 5.6 Conclusion

This chapter focuses on the system level of our design methodology. The system level determines the scope of the groupware system being developed with respect to its support for a cooperative work process. This scope is defined through the definition of its required service and external service dependencies. The system level is also concerned with the description of the system internal behaviour according to an integrated perspective, which forms the basis for refining the system into components afterwards.

The chapter presents the specialisation of some basic design concepts, previously presented at the enterprise level. The need for such specialisation lies on the modelling requirements and capabilities of the system and component levels and expressiveness of UML.

The transition from the enterprise level to the system level requires a number of design decisions. These decisions directly impact the functionality of the system. For example, one should determine which functional roles and, consequently, which activity units require some computer support for its execution and how these activity units are refined into more concrete activity units.

The chapter also proposes three different perspectives for the development of models or specifications of a groupware system. Each one of these perspectives defines a major milestone along a design trajectory. The first milestone comprises the required service specification, the second milestone comprises the decomposed service specification and third milestone comprises the internal behaviour specification. For each of these milestones, the chapter discusses a number of issues that have to be addressed in order to model the system accordingly.

The development of each separate specification is carried out based on the structural, behavioural and interactional views. For each view, a number of techniques have been presented to address the modelling characteristics and requirements of the view. A road map to use these techniques has also been proposed according to the characteristics of each milestone.

The presentation of these techniques exposes both the modelling capabilities and the modelling shortcomings of the current UML version (UML 1.4). Three of the most notorious shortcomings identified are the following: the impossibility of creating interaction diagrams for entities modelled as components, which will become more evident at the component level; the limited capabilities of interaction diagrams for modelling interaction patterns, instead of a particular scenario; and the limited capabilities of statechart diagrams for modelling the state-dependent behaviour of complex systems.

Another shortcoming, which was already pointed out in Chapter 4, is the absence of behaviour structuring constructs in an activity diagram. The absence of such constructs is one of the reasons why we propose a combined approach using both statechart and activity diagrams to capture the order in which interaction contributions of an entity can be executed.

There are two major factors that influence the development of models of a groupware system at the system level, viz., the presence of systems with similar functionality and the environment in which the system is to be deployed.

In case there are other groupware systems or groupware system designs with functionality that is similar to the functionality of the system under development, we should take the interfaces of these systems into account. We should look at those interfaces, i.e., defined set of operations, trying to identify features or operations that are common to most systems in order to provide the system under development with similar or standard interfaces.

If the system has standard interfaces, it is much easier for its users to interact with the system because the user may already be familiar to the interface. Further, the use of standard interfaces also facilitates the integration of the system with other systems. Therefore, the use of standard interfaces should be stimulated.

The environment in which the system is to be deployed should also be taken into account. In case the environment provides a number of (standard) services, these services can be considered as part of the system environment. The more functionality can be shifted to the system environment, the less complex becomes the system design. The system can still provide the required functionality but without the additional implementation overhead that would be normally required.

---

# Chapter 6

## Component Level Modelling

This chapter presents the component level of our design methodology. It first discusses the refinement of a system into components based on pre-defined component types. The use of pre-defined component types allows for a better control of the decomposition process, by the identification of sets of common concerns that are addressed by specific component types.

Then, the chapter discusses a design trajectory similar to the design trajectory used in the specification of a groupware system at the system level that is used in the refinement of abstract components into finer-grained concrete components. Additionally, the chapter also presents a number of techniques to model a groupware system at the component level. Most of these techniques have, however, been introduced at the system level. Finally, the chapter also provides guidelines for refinement assessment and comments on issues related to component reuse.

The chapter is structured as follows: section 6.1 discusses different strategies for the refinement of a system into components and presents our decomposition approach based on three different categories of components, viz., application, groupware and simple components; section 6.2 discusses the logical distribution of collaboration concerns into concern layers within the scope of groupware components; section 6.3 presents the concept of collaboration coupling, which can be established based on the different concern layers, and relates collaboration coupling with the physical distribution of components; section 6.4 presents our design trajectory at the component level, while section 6.5 presents the techniques used in the specification of components according to the views and perspectives adopted in this work; section 6.6 provides some informal guidelines for refinement assessment using UML; section 6.7 discusses some other issues relevant for component-based development; finally, section 6.8 presents some conclusions.

### 6.1 System refinement

#### 6.1.1 From system to components

There are in principle two main approaches to tackle system refinement: we can either refine the interactions between the system and its environment without changes in the granularity of the system, i.e., without decomposing the system into smaller parts, or decompose the system into smaller parts and allocate the system interactions to these parts without changes in these interactions, except for the introduction of new (internal) interactions between the smaller parts. The former type of refinement is called *interaction refinement*, while the latter is called *interaction allocation and flowdown* [Wier98].

These approaches can be combined through multiple refinement steps, in which an interaction refinement is followed by an interaction allocation and flowdown, or vice-versa. Sometimes, these two types of refinement have to be applied at the same time, i.e., within the same re-

finement step, which can be seen as a third type of refinement. For example, suppose a login interaction (login(user)) that only happens for a user that is not logged in. In this example, the need to maintain state synchronisation and the application of distributed constraint on the occurrence of the interaction impose both the decomposition of the system into distributed components and the decomposition of the login interaction into a request interaction and a response interaction.

Figure 6-1 illustrates the difference between the interaction refinement and interaction allocation and flowdown. A system is represented as a large oval, while a system part is represented as a small oval. Interactions are presented as double-edged arrows. Figure 6-1 also shows that these approaches can be combined in a single refinement step or in successive refinement steps to produce some design, in which both the system is decomposed into smaller parts and the interactions are refined into more detailed interactions.

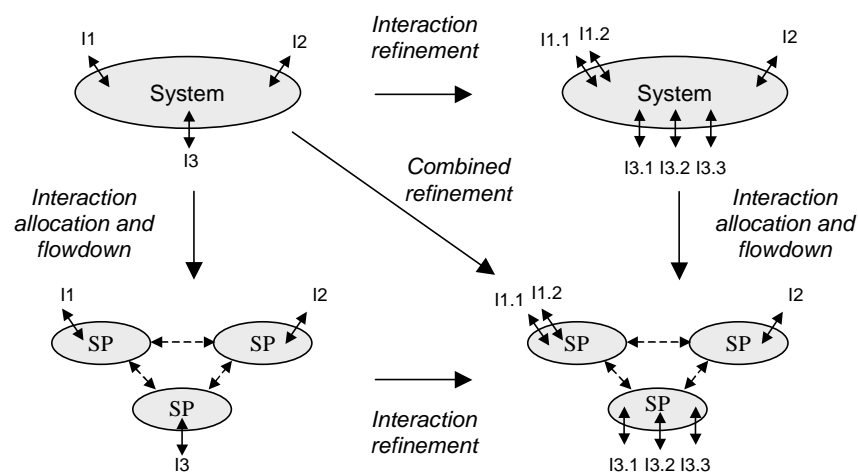


Figure 6-1: Alternative refinement approaches

At the component level, the system is refined into a set of related components, such that the system integrated behaviour is replaced by the composed behaviour of these components. The composed behaviour of the resulting components should conform to the previously specified system behaviour.

We assume here that the refinement process is carried out exclusively according to the interaction allocation and flowdown type of refinement, which characterises the use of a decomposition approach at component level. Since the interactions between the system and its environment are preserved as we refine the system into a set of interrelated components, any interaction refinement should be carried out at the enterprise level and in the transition from the enterprise level to the system level. Therefore, unless explicitly mentioned, we use the term refinement or decomposition as shorthand for interaction allocation and flowdown at the component level.

Figure 6-2 illustrates the refinement of a system into a set of interrelated components. The system is represented as a grey oval, while components are represented as grey circles. At the system level, we abstract from the interactions by using a double edge arrow to represent an interaction point. At the component level, an arrow is used to represent an interaction, while a T-bar is used to represent an interface. A dashed T-bar represents an internal interface, i.e., an interface whose operations are not invoked externally by the system environment, but rather

internally by the components that form the system decomposition. A dashed arrow connecting a component to an internal interface represents an internal interaction.

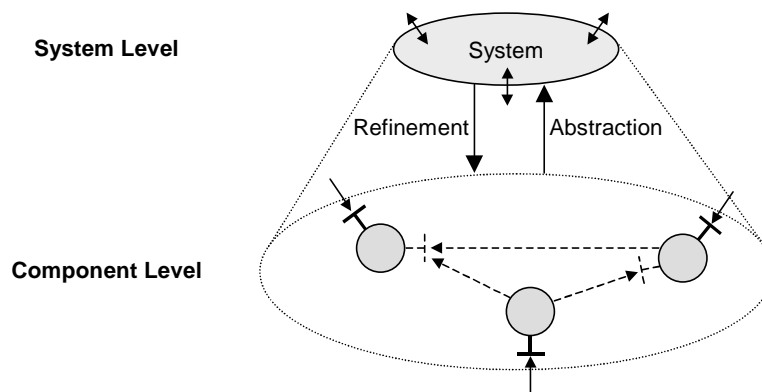


Figure 6-2: System refinement

### 6.1.2 Component classification schemes

Components can be classified in different ways and for different reasons. For example, component classification schemes can be used in a component repository to facilitate the search for components. Further, classification schemes can also facilitate the development of components since more specific development guidelines can be provided to certain types of components.

The most basic component taxonomy refers to whether or not a component is simple or composite (see definition of simple versus composite components in Chapter 2, section 2.1.1). Although effective, this taxonomy scheme is too restrictive because we can only classify a component after its development. Additionally, no distinction is made between composite components containing multiple aggregation levels.

Components can also be classified according to their granularity (level). The granularity of a component defines the number of refinement steps or decompositions that were carried out in a composite component until only simple components were obtained. For example, a component with granularity 3 indicates that 3 decomposition steps were necessary until the behaviour of the component was represented as a composition of simple components only.

The granularity of a component can be homogeneous or heterogeneous. A component with a homogeneous granularity is based on finer-grained components of the same granularity, whereas a component with a heterogeneous granularity is based on finer-grained components of mixed granularity.

The main drawback of classifying components according to their granularity is similar to drawback found in the simple versus composite classification scheme. In a top-down refinement approach, we usually do not know in advance the granularity of a component. Therefore, we can only classify the component afterwards.

In a top-down refinement approach, a better strategy could be the classification of a component according to its purpose in a given reference architecture, which can be regarded as vertical classification scheme. Therefore, different component types could be potentially identified according to the types of concerns addressed by these components along a design trajectory.

Another general approach that can be used to classify components is domain-related, in which a component is primarily classified according to its target application domain, which can be regarded as horizontal classification scheme.

### 6.1.3 Component decomposition approaches

We can identify two slightly different approaches regarding the decomposition of a system into components, viz., a *continuous recursion approach* [DSWi99] and a *discrete recursion approach* [HeSi00]. These approaches are not fundamentally different, since the latter can be seen as a specialisation of the former.

In a continuous recursion approach, the system is continuously refined into finer-grained components until components of a desired granularity level or complexity, e.g., simple components, are identified. Since there are no specific component types, this approach provides mainly general-purpose guidelines for reducing the complexity of a component.

In a discrete recursion approach, the system is systematically refined into components of pre-defined categories along a design trajectory. A component category defines a number of characteristics common to a number of components. Different component categories types correspond usually, but not necessarily, to different component granularities. This decomposition approach usually provides both general-purpose refinement guidelines and specific refinement guidelines for each component category.

Figure 6-3 illustrates the difference between the two approaches. Components are represented as circles, where white and grey circles represent components of different categories. Component composition is represented by lines interconnecting two or more components, whereas refinement is represented by an arrow pointing from a single component to a set of components. Figure 6-3a represents the continuous recursion refinement approach, in which a single component category is identified (grey components). Figure 6-3b represents the discrete recursion refinement approach, in which two component categories (grey and white components) are identified.

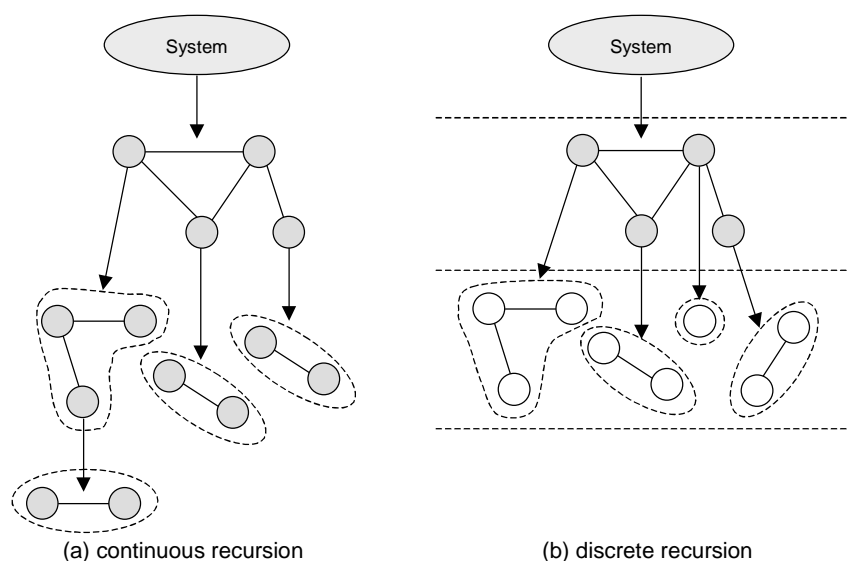


Figure 6-3: Alternative decomposition approaches

#### 6.1.4 Component categories

In this work we use a discrete recursion approach for component decomposition, because this allow us to tailor the component categories according to different set of concerns at the component level. Therefore, we identify three different categories of components inspired by [HeSi00], viz., *simple components*, *groupware components* and *application components*.

A simple component is the most basic unit of design, implementation and deployment. At the component level, a simple component is not further refined. An instance of a simple component runs on a single machine, which is usually part of a distributed environment. Our simple components correspond to the distributed components of [HeSi00], but we avoid this term here to prevent confusion. Simple components are developed according to specific subsets of concerns (see section 6.2).

A groupware component consists of a set of simple components that cooperate in order to provide a mostly self-contained set of groupware functionality. A groupware component corresponds to the behaviour of an independent cooperative concept or feature that can be reused to build larger groupware components and systems.

Self-containment of a groupware component does not imply that this component is isolated from other components. On the contrary, a groupware component should be composable, i.e., one should be able to compose different groupware components, and these components should be able to interact with each other. Nevertheless, a groupware component should have minimal dependencies in order to maximise its reuse.

A groupware component also encapsulates distribution aspects. Since a groupware component is formed by simple components and simple components can be distributed individually across a network, the distribution aspects normally required by a groupware component are consequently addressed by the composition of (distributed) simple components. However, simple components can be used to address not only the physical distribution aspects but also the distribution of concerns and responsibilities that form a groupware component.

An application component corresponds to a groupware application, i.e., an independent application that can be used separately or integrated into another groupware system. The groupware system under development is a typical example of an application component. However, groupware components can also be used as building blocks for larger application components. In most cases, an application component consists of a set of interrelated groupware components that cooperate in order to provide some application-level functionality.

In order to illustrate this component hierarchy, we take a videoconferencing system as an example. This system can be seen as a composition of individual applications, such as videoconferencing, chat and shared whiteboard applications. A videoconferencing application can be decomposed into separate groupware components, which provide, e.g., audio support, video support, attendance support, etc. An audio support component can be decomposed into separate simple components to handle the connection establishment, coding, decoding, transmission, and so on.

Nevertheless, this classification scheme is flexible and subject to the designer's choice and interpretation. For example, in our videoconferencing system example, one could see videoconferencing, chat and shared whiteboard functionality as a separate groupware component

each. Consequently, this system is seen as a composition of individual groupware components, instead of application components.

Figure 6-4 shows the different categories of components used in the context of our refinement approach. A groupware system, which corresponds to an application component, can be refined into a number of individual groupware applications, each one also corresponding to an application component, or into a number of groupware components. A groupware component can be refined into a number of individual groupware components or into a number of simple components, which cannot be further refined at the component level.

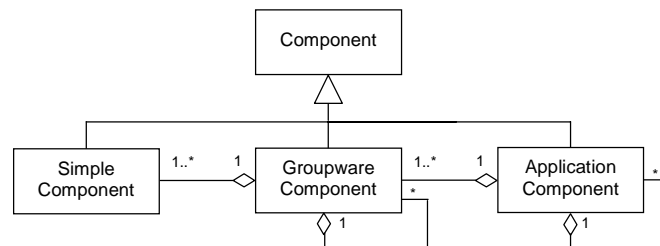


Figure 6-4: Adopted component categories

The component categories adopted in this work are not necessarily in a one-to-one correspondence to granularity levels. The granularity of a simple component is always 0, however although typically a groupware component will have granularity 1, it may not necessarily be true because the groupware feature represented by the groupware component may actually be implemented by a single simple component (see section 6.2).

Informally, each component category defines a sort of concern sub-level of the component level. In each one of these sub-levels, components are identified according to a different set of concerns, which characterise the components defined in the sub-level.

## 6.2 Collaboration concerns

This section discusses the logical distribution of collaboration concerns in a groupware component, such as how to cope with the different levels of freedom that a particular cooperative activity may require.

### 6.2.1 Collaboration concern layers

Suppose a particular groupware component manages the editing of a shared document. This component is responsible for maintaining the consistency of the document, allowing at the same time changes by multiple users. Initially, a user may choose to have a different view of the document, e.g., an “outline” view, as opposed to the view currently used by another user, e.g., a “normal” view. Should this particular choice affect how the second user views the document or should different users be allowed to have different views of the document?

Considering that this particular component is also responsible for keeping the users informed about changes in the document, should a user be notified of every change in the document or another user’s action, or should the user be only notified whenever a change affects the part of the document he or she is currently working on?

These are examples of issues that have to be dealt with by a groupware component. In order to provide flexibility to deal with these and other issues, we identify a number of layers, so-called collaboration concern layers, in which different aspects of the functionality of a groupware component can be structured.

We identify four separate layers, viz., *interface*, *user*, *collaboration* and *resource*. Each layer uses the functionality provided by the layer below in order to provide some functionality that is used by the layer above. The interface layer is the highest layer, while the resource layer is the lowest layer.

The interface layer is concerned with providing a suitable interface between a human user and the groupware component. Considering a drawing component of a shared whiteboard application, the interface layer should enable a user to create new drawings and change or delete existing drawings by means of graphical commands. The interface layer should also enable the user to visualise the drawing itself through the interpretation of drawing commands. Therefore, the interface layer handles all the communication with the users via a graphical user interface.

The user layer is concerned with the local support for the activities performed by a single user. This logic is local to the user and does not affect the collaboration as a whole. For example, suppose that our drawing component enables a user to make changes to a local copy of a shared drawing, without changing the shared drawing immediately. Only when the user is satisfied with these changes, the user incorporates these changes to the shared drawing. Therefore, the user layer maintains a user's perception of the collaboration. The user layer also supports the interface layer, relating it with the collaboration layer.

The collaboration layer is concerned with the collaboration logic of multiple users. Considering our drawing component, the collaboration layer should be able to handle the drawing contributions of multiple users, relating them as necessary. Therefore, this layer is responsible for the implementation of the core aspects of the collaboration and for relating the user layer to the resource layer.

The resource layer is concerned with the access to shared collaboration information (resources), which is, for example, kept persistently in a database. In our drawing component, the resource layer should be able to store the drawing and/or drawing commands, saving and loading them as necessary. The resource layer is only accessible through the collaboration layer.

Figure 6-5 depicts the collaboration concern layers identified in this work and their relationship.

Each collaboration concern layer can be implemented by one or more simple components. An interface component is a component that implements the interface collaboration layer. Similarly, user, collaboration and resource components implement the user, the collaboration and the resource collaboration layers, respectively. Nevertheless, it is common to implement both the interface and user layers in a single (user) component.

We distinguish between three different types of functionality interfaces that a component can support based on the purpose and visibility (scope) of the interface, viz., *graphical user interfaces*, *internal interfaces* and *external interfaces*.

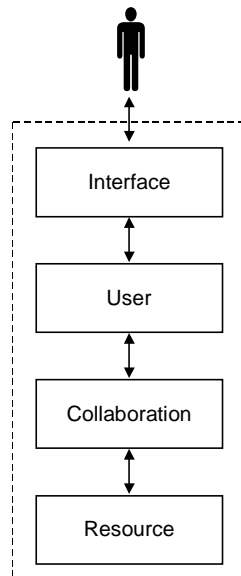


Figure 6-5: Collaboration concern layers

A graphical user interface supports the interactions between a human user and an interface component. An internal interface supports the interactions between the components of a single groupware component. Such an interface has internal visibility, i.e., a groupware component cannot interact with another groupware component using internal interfaces. An external interface supports the interactions between groupware components. Such an interface has external visibility.

Interface components and resource components usually do not have external interfaces. Therefore, interactions between groupware components are only achieved via the user and collaboration components.

Figure 6-6 illustrates an implementation of the collaboration concern layers in which there is a single simple component for each layer. The groupware component is represented by a dashed rectangle, while a single component is represented as a solid rectangle. An interface is represented by a T-bar attached to a component, while an operation invocation is represented by an arrow leading to an interface. External interfaces are represented by a solid T-bar that crosses the boundary of the groupware component, while internal interfaces are represented by dashed T-bars inside the limits of the groupware component. Graphical interfaces are not explicitly represented.

A groupware component does not need to have all four layers. For example, it is only meaningful to have a resource layer if some shared information has to be stored persistently. Similarly, an interface layer is only meaningful if the component interacts with a human user.

Nevertheless, if a groupware component has more than one layer, these layers are strictly hierarchically related. For example, the interface layer cannot access the collaboration layer or the resource layer directly, nor can the user layer access the resource layer directly. Thus, a single groupware component consists of at least one and up to four ordered collaboration concern layers.

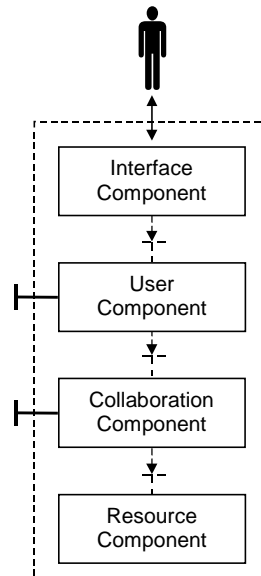


Figure 6-6: Implementation of concern layers

Figure 6-7 illustrates some examples of groupware components containing valid sets of collaboration concern layers. Each layer is represented by a corresponding simple component.

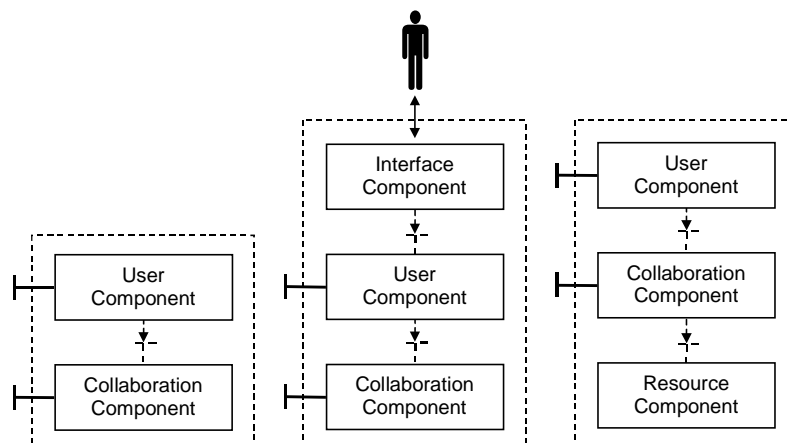


Figure 6-7: Examples of valid layering distribution

### 6.2.2 Related work

Our collaboration concern layers have been identified based on a number of developments, viz., Patterson's state levels [Patt95], Herzum and Sims' distribution tiers [HeSi00] and Wilson's architectural layers [Wils99]. These developments address similar issues although with different terminology.

Patterson [Patt95] identifies four state levels in which a synchronous groupware application can be structured, viz., display, view, model and file. The display state contains the information that drives the user display. The view state contains the information that relates the user display to the underlying information in the application, the model state. The file state consists of a persistent representation of the application information. Based on Patterson's state levels, Ter Hofté proposes a four-level collaborative architecture known as the zipper architecture [Hoft98].

Herzum and Sims [HeSi00] propose a distribution architecture for business components that consists of four tiers, viz., user, workspace, enterprise and resource. These tiers are roughly equivalent to our collaboration concern layers. However, this architecture emphasises distribution aspects, instead of the collaboration aspects that we emphasise in our work.

Wilson [Wils99] proposes the development of a distributed application according to four architectural layers, viz., view, application-model, domain and persistence. The view layer deals with user interface issues, the application-model layer deals with application-specific logic, the domain layer deals with domain-specific logic, and the persistence layer deals with the storage of information in a persistent format.

Table 6-1 shows the correspondence between our collaboration concern layers, Patterson's state levels, Herzum and Sims distribution tiers and Wilson's architectural layers.

Collaboration Concern Layers	Patterson's State Levels	Herzum and Sims' Distribution Tiers	Wilson's Architectural Layers
Interface	Display	User	View
User	View	Workspace	Application-Model
Collaboration	Model	Enterprise	Domain
Resource	File	Resource	Persistence

Table 6-1: Collaboration concern layers and related approaches

## 6.3 Collaboration coupling and distribution issues

### 6.3.1 Coupling levels

So far we have discussed the collaboration concern layers of a groupware component in the context of a single user. However, collaboration concerns usually involves multiple users.

In the context of a groupware system, and particularly of a groupware component, two or more users may or may not share the same perception of the ongoing collaboration supported by the system. For example, in the case of the groupware component that manages the editing of a shared document, if one user chooses an outline view of the document instead of a normal view, this decision could possibly affect another user's view of the document. In case all the component users share the same perception, the outline view would replace the normal view for all users. Otherwise, the other users do not share the same perception of the collaboration, and only that particular user would have an outline view of the document.

We can apply the same reasoning to each collaboration concern layer of a groupware component. As a consequence, collaboration concern layers can be coupled or uncoupled to one another. *Coupling* was introduced in [Hoft98] as a general mechanism for uniting the interaction contributions of different users, such that users might share the same view or state of a collaboration.

A collaboration concern layer of a groupware component is coupled if all users have the same perception of the information present in the layer and how this information changes. Therefore, a collaboration concern layer across multiple users can be coupled or uncoupled. An im-

important property of coupling is downwards transitivity, which means that if a layer is coupled, the layers below, from the interface layer down to the resource layer, must be coupled as well in order to ensure consistency.

Four levels of coupling can be established based on the collaboration layers defined in this work, viz., *interface coupling*, *user coupling*, *collaboration coupling*, and *resource coupling*.

The interface coupling level represents the tightest coupling level. All the component users have the same perception of the collaboration, starting at the user interface layer. This level corresponds to the collaboration style known as What You See Is What I See (WYSIWIS) [SBF+87].

The user coupling level offers more freedom (independence of use) to the component user than the interface coupling level. All the component users have the same perception of the collaboration starting at the user layer, i.e., the information at the user layer is shared by all users, but their interface layers are kept apart.

The collaboration coupling level goes a step further and offers more freedom than the user coupling level. All the component users have the same perception of the collaboration starting at the collaboration layer, i.e., the information at the collaboration layer is shared by all users, but their interface and user layers are kept apart.

The resource coupling level offers the loosest coupling level. All component users have the same perception of the collaboration only at the resource layer, i.e., the information at the resource layer is shared by all users, but their interface, user and collaboration layers are kept apart.

Figure 6-8 depicts the collaboration coupling levels defined in this work for two users. A large rectangle labelled with the layer initial indicates that the layer it represents is coupled, while a small rectangle also labelled with the layer initial indicates that the layer it represents is uncoupled. Figure 6-8a depicts the interface coupling level, Figure 6-8b depicts the user coupling level; Figure 6-8c depicts the collaboration coupling level; and Figure 6-8d depicts the resource coupling level. Figure 6-8e depicts the absence of coupling at all levels, i.e., the two instances of the groupware component operate independently from each other.

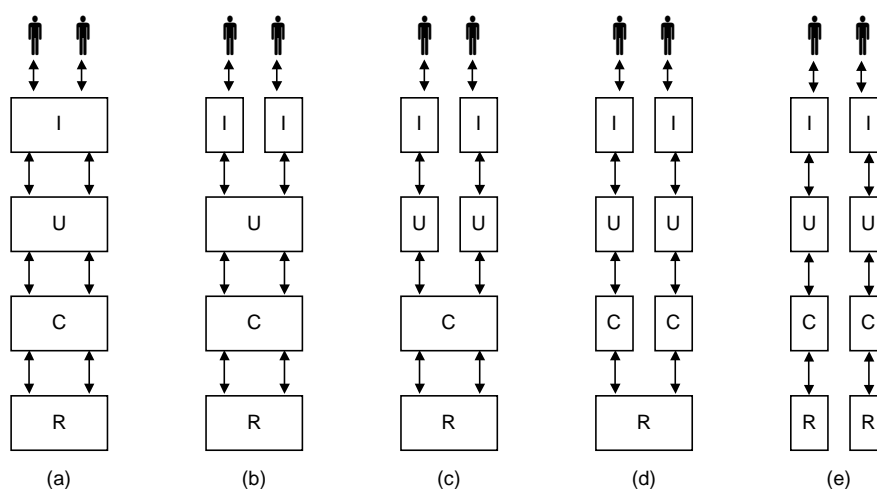


Figure 6-8: Collaboration coupling levels

In a truly flexible groupware system, the users of this system should be able to choose between the different levels of coupling, changing it at run-time based on the characteristics and requirements of the task at hand. They should also be able to choose a temporary absence of coupling, i.e., the users may decide to work independently for a while, resuming their coupling status sometime later. In this case, additional mechanisms to assure the consistency of the collaboration afterwards have to be implemented.

### 6.3.2 Distribution issues of collaboration coupling

There are basically two ways to achieve collaboration coupling in a given layer, viz., using a centralised architecture or using a replicated architecture with synchronisation mechanisms.

In a centralised architecture, a single copy of the collaboration state, i.e., all the information contained in a layer, is maintained and shared by the users of the layer. Concurrency control mechanisms should be used to avoid inconsistencies if necessary.

In a replicated architecture, multiple copies of the collaboration state are maintained, one for each user of the layer. In this case, synchronisation mechanisms (protocols) are used to maintain the consistency across the replicated copies of the collaboration.

The collaboration state of an uncoupled layer is non-centralised by definition, i.e., each user maintain its own copy of the collaboration state. However, the collaboration state of a given coupled layer can be either centralised or replicated. Therefore, we can possibly have different architectures for each coupling level defined in this work.

Except for the resource coupling level, which can only be either centralised or replicated since the other layers are uncoupled, we may have for each coupling level a fully centralised architecture, a fully replicated architecture or a hybrid architecture. In a fully centralised architecture, coupling across all coupled layers is achieved using a centralised architecture. Similarly, in a fully replicated architecture, coupling across all coupled layers is achieved using a replicated architecture. A hybrid architecture combines both architectural styles. However, because of implementation-related reasons, once a coupled layer is implemented using a centralised architecture, all layers below should also be implemented using this architectural style.

Although possible in principle, it is very unlikely that interface coupling is achieved using a centralised architecture because of the complexity involved and response time requirements. Interface coupling is usually implemented based on shared window systems (see [SFB+87, AhEL90, LJJ+90]).

Figure 6-9 illustrates three possible combinations of centralised and replicated architectures to achieve coupling at the collaboration layer for two users. Figure 6-9a depicts a fully centralised architecture, Figure 6-9b depicts a hybrid architecture, and Figure 6-9c depicts a fully replicated architecture. A rectangle represents an instance of a simple component, which is labelled with the initial of the layer it implements. A large rectangle indicates a component in a centralised architecture, while two small rectangles connected by a synchronisation bar, a double edge horizontal arrow, indicates a component in a replicated architecture. Small rectangles without a synchronisation bar indicates that the layer they represent are uncoupled.

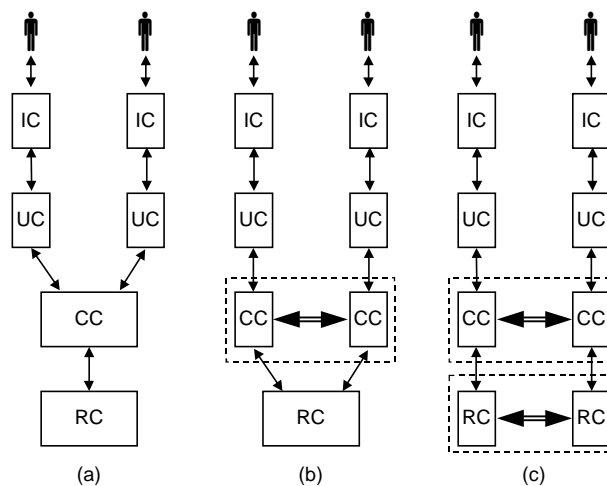


Figure 6-9: Centralised versus replicated architectures

The discussion on the benefits and drawbacks of centralised versus replicated architectures, as well as the mechanisms used to achieve consistency in both architectures, is beyond the scope of this work and we refrain from discussing these issues further. For a detailed discussion on these issues we refer to [AhEL90, LJL+90, PaDK96, SKS+96, Hoft98, RoUn00].

## 6.4 Component design trajectory

In order to model a groupware system at the component level we need to decompose the system recursively into finer-grained components, according to the component categories defined in section 6.1.4. Each component category can be seen as a sub-level of the component concern level, in which the design of component of the same category is carried out following the same trajectory, techniques and associated guidelines.

Initially the system is seen as an application component. The refinement of an application component into fine-grained application components is not mandatory. This refinement step will take place whenever the abstract application component actually corresponds to the specification of two or more groupware systems, which are mostly independent from each other, i.e., we can identify two independent service descriptions. For example, the groupware system under development represents a groupware environment comprising a number of groupware applications. In such case, further refinements of the application components may or may not reveal some common behaviour between the application components, such as common groupware components.

We can address the existence of a groupware component common to more than one application component in two separate ways at deployment time: there are separate instances of the common component, one for each application component, or there is a common instance shared by all the application components. In the former situation, the service provided by each application component is completely independent from the other components, while in the latter situation the service provided by the application components are dependent on one another.

In case an application component cannot be refined into finer-grained application components, this application component should be refined into a set of related groupware components. Likewise an application component, a groupware component can be refined into finer-

grained groupware components, in which the service provided by each of these components is independent from one another. Once again, further refinements of these groupware components may or may not reveal some common behaviour between the groupware components, such as common simple components.

Once again, we can address the existence of a simple component common to more than one groupware component in two separate ways at the deployment time: there are separate instances of the common component, one for each groupware component, or there is a common instance shared by all the groupware components. In the former situation, the service provided by each groupware component is completely independent from the other components, while in the latter situation the service provided by the groupware components are dependent on one another.

In case a groupware component cannot be refined into finer-grained groupware components, this groupware component should be refined into a set of related simple components. Once a groupware component is refined into a set of related simple components, these simple components cannot be further refined at the component level.

At each refinement step we should specify the identified components according to the behavioural perspectives proposed at the system level. After the specification of each component according to the required service and decomposed required service perspectives, a search should be carried out in order to identify existing components that, at least partially, provide the required functionality (see section 6.7.2 for more on component reuse). If a suitable component is identified, the design process for this component is basically over. However, if a suitable component is not identified, the design process continues and the component should be specified according to the integrated internal perspective in order to facilitate further refinements.

Figure 6-10 illustrates our design trajectory at the component level. In this example, we consider that the system is directly refined into a number of groupware components and these groupware components are directly refined into a number of simple components. Simple components can only be refined into objects at the object level. For simplification purposes, in Figure 6-10 we abstract from the required service specification and the decomposed service specification, representing both only as service specification.

The starting point for the refinement process at the component level is the specification of the system developed according to the integrated internal perspective. Therefore, the system internal behaviour specification corresponds to an internal behaviour specification of an application component.

The identification of concrete components based on an internal behaviour specification takes into account both the interactions supported and initiated by the abstract component and its internal actions.

In general, concrete components are initially identified based on different subsets of the required service specification of the abstract component, i.e., we first take into account part of the interactions involving the abstract component and its service users. Different subsets of these interactions correspond to separate concrete components. The specification of these interactions and their relationships usually provides, for each concrete component, a partial specification of the component required service. At this point, the boundaries of the concrete components are not precisely defined yet.

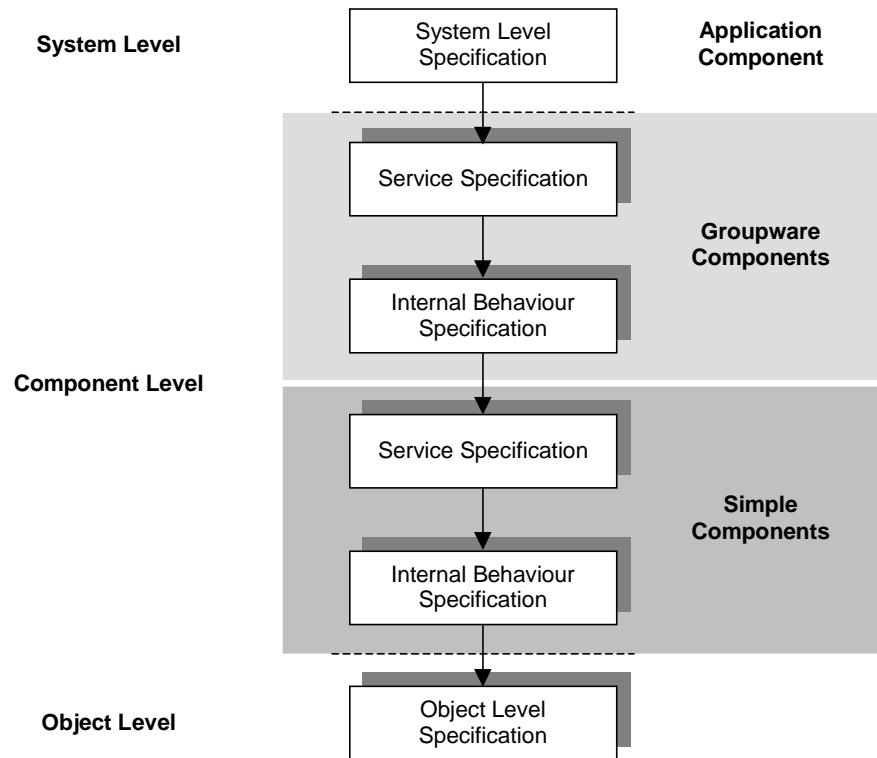


Figure 6-10: Design trajectory at the component level

The initial required service specification of each concrete component is usually a partial description because it abstracts from both interactions between the abstract component and its auxiliary entities and the internal actions of the abstract component.

In a second design step, we take into account both the interactions between the abstract component and its auxiliary entities and the abstract component internal actions. This allows us to draw the boundary of the concrete components by specifying their decomposed service specification. In this design step, we possibly identify new concrete components, which neither support nor initiate any of the interactions between the abstract component and its service users.

Whilst the specification of the concrete components according to the required service perspective can be seen primarily as a refinement of the abstract component required service, the specification of the concrete components according to the decomposed required service perspective consists of a refinement of the abstract component internal behaviour specification. Therefore, this design step is strongly influenced by the abstract component internal actions, the so-called abstract actions, and their relationships.

Whenever we establish the boundaries of the concrete components, we can either refine the relationship between abstract actions through the insertion of a number of concrete actions, which can subsequently be refined into interactions between the concrete components or refine abstract actions directly into interactions between the concrete components.

Both refinement techniques can be applied separately or in combination. However, the decision on whether we choose action refinement or causality refinement followed by action refinement should be taken in an early specification stage, possibly while these actions are being identified because this choice may affect the set of concrete components obtained.

## 6.5 Component modelling

This section presents the modelling strategy, road map and specification techniques used in the specification of components. The strategy described in this section can be applied for each concrete component identified as a result of the decomposition of an abstract component. We use the term subject component to refer to any given concrete component being specified.

### 6.5.1 Modelling strategy

For each component category, the development of the subject component required service specification, decomposed service specification and internal behaviour specification is carried out basically in the same way as the development of their corresponding specifications at the system concern level. Therefore, the specification of the subject component according to each perspective is carried out based on the structural, behavioural and interactional views.

Figure 6-11 depicts the intersection of the concerns defined by the different views versus the concerns defined by the different perspectives.

	Structural View	Behavioural View	Interactional View
Required Service Perspective			
Decomposed Required Service Perspective			
Internal Integrated Perspective			

Figure 6-11: Intersection of views and perspectives concerns

The set of interactions in which the subject component is involved defines the component structure. Initially in the development of the component required service specification, we consider only the interactions between the subject component and the abstract component service users that are both supported and initiated by the subject component. These interactions correspond to a subset of the interactions that are common to both the abstract component and its service users (see section 6.4). At this point, we abstract from other service users of the subject component, i.e., other concrete components that may use the services provided by the subject component as a result of the refinement of the abstract component. Consequently, we abstract from the interactions between the subject component and other concrete components.

The identification of the interactions between the subject component and its service users determines the interface(s) supported and associated operations. In order to capture the structural view, we identify operations, group these operations into interfaces and establish dependency relationships between the subject component service users and the interfaces supported by the subject component (and vice-versa).

The relationships between these interactions are described in the subject component specifications according to the behavioural and interactional views. When we take into account the interaction contributions of a single entity, we have a description according to the behavioural

view. However, when we take into account interactions as a whole, i.e., interactions as integrated actions between entities, we have a description according to the interactional view.

To identify relationships between interactions, we basically have to capture the order in which interactions can take place and information value dependencies. Similarly to the system level, we capture the order in which interactions can take place according to the state of the subject component.

At the required service perspective, the resulting specification of a subject component corresponds to a subset of the required service specification of the abstract component, since only a subset of the abstract component interactions with its service users are considered. In case there is a relationship between some of these interactions and the interactions assigned to another concrete component, this relationship should be taken into account and specified indirectly by means of conditions and constraints.

During the development of the subject component decomposed required service, we refine its structural, behavioural and interactional views descriptions, possibly incorporating the interactions with the abstract component auxiliary entities and other internal interactions, i.e., interactions with other concrete components. Internal interactions are identified as a result of the refinement of an abstract action or a relationship between abstract actions assigned to different concrete components. During this refinement step, we can possibly identify concrete components not yet identified during the previous refinement step, since this step considers only those concrete components that interact with the abstract component service users.

From the point of view of the subject component, in this refinement step we identify only those interactions related to its internal auxiliary entities, i.e., other concrete components that provide some service to the subject component. However, since from the point of view of another concrete component our subject component may be seen as an auxiliary entity itself, we augment the subject component required service specification to include the description of the service provided to other components, which were not originally considered during the specification of the subject component required service.

The development of the internal behaviour specification of a subject component is also carried out based on the structural, behavioural and interactional views. However, the structural and interactional views described in the decomposed service specification are almost unchanged during this refinement step. Actually, the interactional view remains the same, while some changes that do not affect the structural view description for an external observer can be carried out on the structural view description itself. Nevertheless, once more the focus is on the refinement of the behavioural view description.

During the development of the internal behaviour specification, we can refine an abstract action, the relationship between two interactions or both, through the identification of a number of concrete actions and their relationships.

### **6.5.2 Techniques and road map**

The structural view of a subject component according to the required service and decomposed required service perspectives is captured using functionality decomposition table, use case, package and system interface diagrams. A decomposition table is used to assign functionality from an abstract component to finer-grained concrete components. Use case diagrams are used to capture the subject component requirements. Package diagrams are used to capture the

static relationship between the subject component and its environment. Finally, component interface diagrams are used to define the interfaces of both the subject component and its environment, and to capture interface dependencies. Additionally, the glossary is used to document new terms identified in the use case, package and interface diagrams.

Similarly to the system level, the specification of the subject component according to the internal integrated perspective may require some changes on how units of behaviour are organised after the introduction of internal actions. These changes are only visible in use case diagrams and they are captured augmenting the existing use case diagrams as necessary. Further, any new use case identified should be documented in the glossary.

During the specification of the required service and decomposed required service perspectives, the behavioural view of a subject component is captured using primarily a mix of statechart diagrams and activity diagrams. Statechart diagrams are used to capture the order in which the interactions (interaction contributions) supported by the component should occur. Activity diagrams are used to relate these interactions to the interactions (interaction contributions) initiated by the component in response. Additionally, these diagrams can be used to capture the relationship between the information exchanged during these interactions.

The behavioural view description of the subject component developed according to the service perspectives is complemented through the specification of the interface operations supported by this component. Interface operation specifications are used to capture the relationship between the information exchanged between two interactions that form an instance of the invocation interaction pattern. Further, such specifications are also used to document how interactions affect the state of the component.

During the specification of the internal integrated perspective, the behavioural view of the subject component is captured primarily through the refinement of the activity diagrams previously identified in its corresponding decomposed service specification. Further, the corresponding activity diagrams developed to capture internal integrated perspective of the abstract component may be partially refined as well. Additional use case descriptions are also considered at this point.

The interactional view is captured primarily using component interaction diagrams. A component interaction diagram is similar to a system interaction diagram at the system level. However, instead of capturing the ordering of interactions between system level entities, a component interaction diagram captures the ordering of interactions between component level entities. Use cases can also be used to capture the interactional view in their textual form.

Table 6-2 summarises the techniques that can be used in the development of service and internal behaviour specifications of a subject component.

There are several alternative ways in which we can apply these techniques to produce an intended specification of a component. However, the suggested road map is similar to the road map used at the system level for specifying the system required service, decomposed required service and internal integrated behaviour, except for the elaboration of a decomposition table in the beginning of this process.

Technique	Structural View	Behavioural View	Interactional View
<i>Decomposition table</i>	applicable	not applicable	not applicable
<i>Use case diagram</i>	applicable	applicable	applicable
<i>Package diagram</i>	applicable	not applicable	not applicable
<i>Interface diagram</i>	applicable	not applicable	not applicable
<i>Glossary of terms</i>	applicable	not applicable	not applicable
<i>Statechart Diagram</i>	not applicable	applicable	not applicable
<i>Activity Diagram</i>	not applicable	applicable	not applicable
<i>Interface operation specification</i>	not applicable	applicable	not applicable
<i>Component interaction diagram</i>	not applicable	not applicable	applicable

Table 6-2: Component specification techniques

Figure 6-12 depicts our suggested road map for specifying the decomposed required service (Figure 6-12a) and internal integrated behaviour (Figure 6-12b) of a subject component. We do not show the road map for the specification of the required service of a component because this road map is similar to the road map shown in Figure 6-12a.

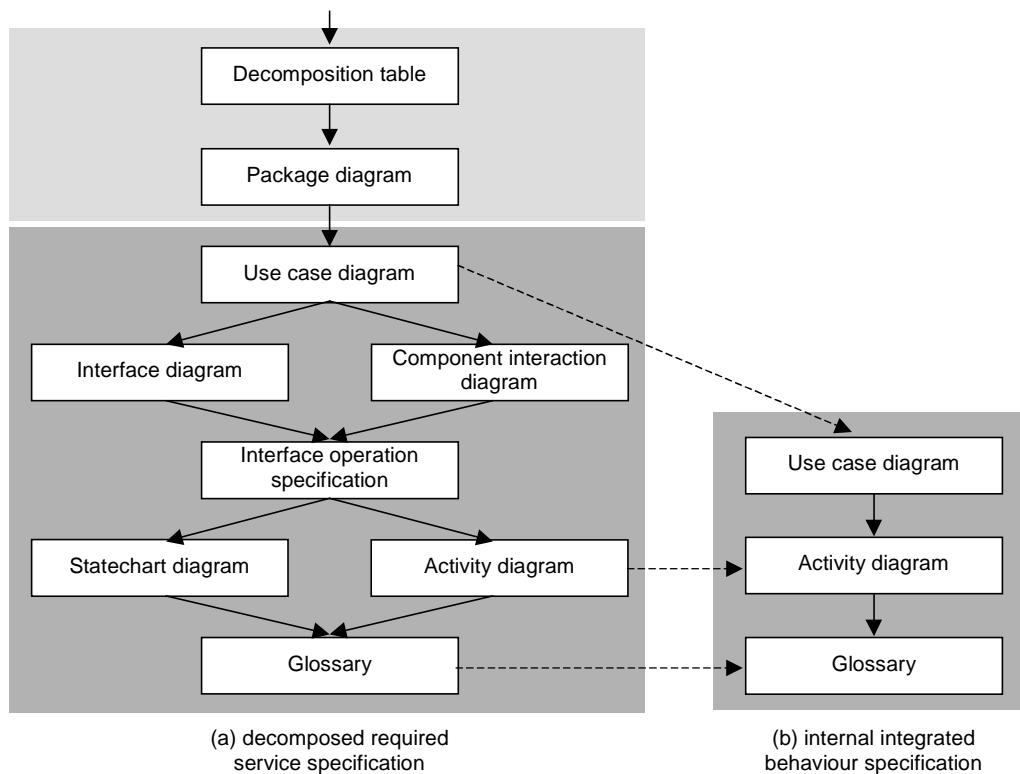


Figure 6-12: Component level modelling road map

In Figure 6-12a, each box corresponds to the elaboration of a certain type of diagram or specification. Solid arrows indicate the order in which the design activities should be carried out, whereas dashed arrows connecting the same type of diagram or technique across different specification milestones indicate that the diagram or technique is augmented or refined in a

subsequent refinement step. Boxes horizontally placed beside each other across the same design milestone indicate that the associated design activity should be carried out in parallel. The area in light grey represents the design activities that concern multiple components, while the area in dark grey represents the design activities that concern a single component. In the sequel, we briefly discuss the application of each of these techniques at the component level.

### 6.5.3 Component specification techniques

#### Functionality decomposition table

The elaboration of a decomposition table aims at assigning functionality from an abstract component into finer-grained concrete components. An abstract component can represent either a groupware system, i.e., an application component, or a groupware component. The use of decomposition table in this work was inspired by the use of a traceability table with similar purposes in [Wier98].

Since use cases capture pieces of functionality, we elaborate a decomposition table by assigning use cases to components. Normally different alternative sets of components may all correctly support the functionality represented by use cases being assigned, i.e., the different sets of components produce all equivalent results.

A functionality decomposition table can be elaborated in three basic steps: (1) identification of use cases; (2) identification of potential components; and (3) assignment of use cases to components.

The format of a functionality decomposition table is simple. In one dimension of the table, we list the use cases associated with the abstract component being refined. In the other dimension of the table, we list concrete components as potential candidates for assignment. Therefore, the elaboration of a functionality decomposition table is not a design activity specific for the specification of a concrete component, but rather shared by all concrete components identified.

Figure 6-13 shows the generic format of a functionality decomposition table.

<i>Component</i> <i>Use case</i>	<b>Component 1</b>	<b>Component 2</b>	<b>...</b>	<b>Component N</b>
<b>Use case 1</b>		X		
<b>Use case 2</b>	X			
<b>Use case 3</b>		X		
<b>...</b>				
<b>Use case N</b>				X

Figure 6-13: Functionality decomposition table format

The identification of the use cases to be assigned is straightforward. It consists of all the use cases defined or previously assigned to the abstract component being refined. However, we elaborate the decomposition table incrementally, according to the behavioural perspective being specified.

In the specification of the concrete components according to the required service perspective, we list only those use cases directly associated with actors that represent service users of the abstract component. In the specification of the concrete components according to the decomposed required service perspective, we list other use cases that are associated with actors that represent auxiliary entities. Finally, in the specification of the concrete components according to the internal integrated perspective, we list use cases that are not directly associated with actors.

The identification of potential concrete components is more subject to personal judgement. A potential component is a component that can potentially provide the functionality described by a use case. We identify a potential component only if there is a clear indication that at least one use case can be assigned to this component.

In general, there is no rule of thumb on how to assign use cases to components. A good practice is to keep similar functionality in the same component in order to reduce dependencies, and to place distinct functionality in separate components. Although similarity and distinction are subjective terms, sometimes it suffices to rely on the individual judgement and experience of the application designer. In section 6.7.1, a number of guidelines are provided in order to help the application designer identifying potential components.

When assigning use cases to components, it is possible that no other previously identified component is likely to provide the use case functionality. In such case, a new component has to be added to the list of components and the use case is assigned to this new component.

A use case should not be assigned to more than one component. In case a use case is likely to be assigned to two or more components, it is possible that this use case is too complex and that it could be factored into simpler use cases. In this case, these use cases would replace the original use case in the resulting use case diagram for the abstract component being refined.

### **Package diagram**

The elaboration of a package diagram aims at capturing the static relationship between the concrete components and their environment. Similarly to the functionality decomposition table, the elaboration of a package diagram concerns multiple components.

Each component identified at the functionality decomposition table should be modelled as a separate package. Each package should be labelled with the component structure stereotype, <<component>>, to indicate that the element being stereotyped represents a separate functional entity at the component level. Further, dependency relationships should be established between the identified concrete components and between these concrete components and the entities belonging to the environment of the abstract component (service users and auxiliary entities).

### **Use case diagram**

The elaboration of a use case diagram at the component level aims at capturing the functionality of the concrete components and establishing their context.

For each component identified in the refinement process, a separate use case diagram should be elaborated. This use case diagram should include the use cases assigned to this component and their relationships. Similarly to the development of the functionality decomposition table,

a use case diagram is usually developed incrementally according to the milestones defined by the different behavioural perspectives.

New use cases can also be added to the diagram, by applying the factoring techniques described at the system level. In case new use cases are identified, they should be described according to the convention suggested at the system level. The description of the existing use cases should be updated to reflect any changes in the diagram. The actors that should be identified in this diagram include not only the actors previously associated with these use cases, but also actors that represent other components that are part of the component environment.

### **Component interface diagram**

The elaboration of a component interface diagram aims at defining the concrete component interfaces and capturing interface dependencies. This diagram is also elaborated incrementally according to the required service and decomposed required service perspectives.

For each component identified in the functionality decomposition table and depicted in the package diagram we should include this component into the diagram and identify its supported interfaces and interface dependencies. A component is represented in the diagram by a class labelled with the component stereotype, <<component>>.

Further, for each interface we should identify its operations based on instances of the interactions patterns associated with the component. These interactions are a subset of the interactions supported and initiated by the abstract component, augmented by the interactions supported and initiated by the concrete component in order to interact with other concrete components.

Interface dependencies are established according to the guidelines provided at the system level for the elaboration of the system interface diagram.

Similarly to the system interface diagram, it is also possible to elaborate a variant of the component interface diagram in which we capture only the interfaces and interface dependencies associated with a single component. In this variant, we include, besides the component, the interfaces that are either realised by this component or associated with the component in a dependency relationship. In this case, we have to elaborate a separate interface diagram for each concrete component.

### **Interface operation specification**

The elaboration of an interface operation specification aims at describing the operations of an interface.

The description of an operation specified in the context of an abstract component is mostly kept unchanged. Small changes, though, are possible, concerning for example, the name of the interface where the operation is defined. This occurs because although the operation may refer to an interaction associated with the abstract component, the interface to which the operation was assigned may change.

Operations associated with interactions introduced during component decomposition should be properly described using the guidelines provided for this technique at the system level.

**Glossary of terms**

At the component level we should document the terms related to use cases, components and interfaces introduced during the refinement process.

**Component interaction diagram**

The elaboration of a component interaction diagram aims at capturing the order in which interactions between a concrete component and its environment takes place. In general we represent a number of concrete components in a same diagram. Thus, interactions between concrete components are only considered at the decomposed required service perspective.

Similarly to the elaboration of the system interaction diagram, it is the designer prerogative to choose between the elaboration of either sequence interaction diagram and collaboration interaction diagram. However, to facilitate the correctness assessment, the designer should use the same type of diagram as the one he used to capture the interactions involving the abstract component.

The scenarios used to capture the interactions involving the abstract component should also be used to capture the interactions at this level. At the component level, interactions are also identified based on the detailed descriptions of the use cases captured in the corresponding use case diagrams developed for each concrete component.

Additional interactions are only identified during the specification of the component decomposed required service. These interactions correspond to the refinement of an abstract action or relationship between two abstract actions assigned to different components.

**Statechart diagram**

The elaboration of a statechart diagram aims at capturing the order in which the interactions supported by a concrete component can take place according to the state of the component. For each concrete component, a separate statechart diagram should be elaborated according to the guidelines described at the system level.

This diagram is also developed incrementally according to the required service and decomposed service perspectives. In the development of the diagram according to the required service perspective, we consider only a subset of the interactions and states (or possibly some refinement of these states) defined in the statechart diagram of the abstract component. In the development of the diagram according to the decomposed required service perspective, we also consider the interactions between a concrete component and other concrete components as a result of the abstract component internal integrated behaviour decomposition. In this case, we can introduce additional states as necessary.

**Activity diagram**

The elaboration of an activity diagram aims at capturing for each interaction that a concrete component supports the behaviour associated with this interaction in terms of the order in which other (inter)actions are executed as a result of the occurrence of this interaction. These (inter)actions are executed during a state transition.

For each state transition identified, a separate activity diagram should be incrementally developed according to our behavioural perspectives. This diagram initially captures only the interactions between the concrete component and the service users of the abstract component. In a

subsequent refinement step, we possibly capture the interactions between the concrete component and the abstract component auxiliary entities, and the interactions between the concrete component and other concrete components. Finally, as a result of yet another refinement step, we also capture the actions performed during the state transition.

These actions usually consist of a refinement of the actions previously identified as part of the abstract component internal integrated behaviour and the relationships between these actions. Similarly to the elaboration of an activity diagram at the system level, actions can be identified based on use case detailed descriptions. However, as we refine abstract components into concrete components successively, these activity units become more fine-grained and they are usually not captured at this level of detail in a use case diagram.

## 6.6 Refinement assessment

The refinement of an abstract behaviour should result in a concrete behaviour that conforms to the abstract one. At the component level, the refinement of an abstract component into a number of interrelated fine-grained concrete components should conform to the behaviour provided by the abstract component.

An abstract behaviour can be refined into a more concrete behaviour in many different ways. In order to assess whether or not the concrete behaviour conforms to the abstract behaviour, we have to abstract from the concrete behaviour and compare both behaviours. Both behaviours should be equivalent from the point of view of an external observer (*observable equivalence*).

Figure 6-14 illustrates the conformance assessment process applied in the refinement of an abstract component into a number of concrete components. An abstract component is represented as large oval, while a concrete component is represented as a small circle. Interrelationship between two concrete components is represented as a line connecting these components. The abstraction of the behaviour provided by concrete components conforms to the behaviour of an abstract component in case both behaviours provide the same service to an external observer.

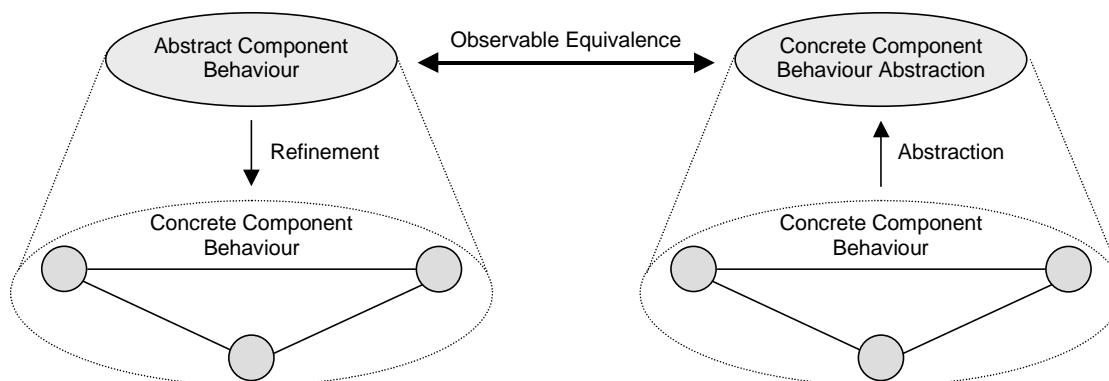


Figure 6-14: Conformity assessment process

In order to assess the refinement of an abstract component into a decomposition of a number of interrelated concrete behaviours, each one assigned to a separate concrete component, we can use the following strategy: (1) abstract from the integrated internal specification of each concrete component to obtain the decomposed required service specification; (2) combine the

decomposed required service of each concrete component, abstracting from their internal interactions, to obtain an integrated internal behaviour specification of the abstraction of the concrete components; (3) abstract from the integrated internal behaviour specification of the abstraction of the concrete components to obtain a decomposed required service specification; and (4) abstract from the decomposed required service specification of the abstraction of the concrete components to obtain a required service specification, so we can compare the resulting specification with the required service specification of the abstract component.

### 6.6.1 Refinement assessment in UML

The provision of formal guidelines for refinement assessment using UML is outside the scope of this research. Since multiple diagrams determine the behaviour of a functional entity, instead of a single diagram, this further complicates conformance assessment because it is more difficult to compare two behaviours.

Therefore, we provide some general guidelines that can be applied to the different types of diagrams for an informal assessment. Some of these guidelines are inspired in the refinement assessment process proposed in [Quar98]. Although these guidelines can be mostly applied to assess the refinement at any abstraction level, including refinement at the enterprise level, we discuss their application only at the component level.

The requirements that must be fulfilled during the refinement process are:

- the interactions between concrete components and the abstract component environment should form disjoint subsets of the corresponding interactions between the abstract component and its environment. The same (concrete) interaction cannot be associated with more than one concrete component;
- each concrete component contains a subset of the abstract component states. Additional states and transitions can be identified, but no original state or transition can be removed. Therefore, a transition connecting two states of the abstract component, should in a concrete component either connect the same corresponding states or connect substates of the original corresponding states;
- additional actions cannot be introduced into the concrete behaviour specification, unless these actions represent a refinement of an abstract action or a refinement of a relationship between two abstract activity units.

Besides these requirements we also need to provide some basic definitions to help us understand the refinement assessment guidelines.

The activity units (actions and interactions) performed by an abstract component are called abstract activity units (abstract action and abstract interaction), while the activity units performed by a concrete component are called concrete activity units (concrete action and concrete interaction).

We assume that each abstract activity unit has at least one corresponding concrete activity unit. In this way, it is possible to compare the behaviour executed during a transition in the abstract component, with the behaviour executed during a corresponding transition in a concrete component by comparing the abstract activity units with their corresponding concrete activity units.

Concrete activity units that correspond to abstract activity units are called reference activity units, since they are considered as reference points in the concrete behaviour for conformance assessment, while the remaining concrete activity units are called inserted activity units, since they were inserted during the refinement.

### 6.6.2 Statechart diagram refinement assessment

A statechart diagram is used to capture the order in which the interactions involving a component can take place. A statechart diagram captures this order by associating the possible occurrences of these interactions with the component state.

If the order in which interactions take place in the abstract component should be maintained in the concrete components, we have to decompose the abstract component state such that the association between interactions and the component state is preserved.

Consider for example, the statechart diagram depicted in Figure 6-15. According to this diagram, the state of an abstract component is formed by two states, viz., State A and State B. Further, in State A two interactions can take place, represented by the events Int1 and Int2, while in State B only one interaction, represented by event Int3, can take place.

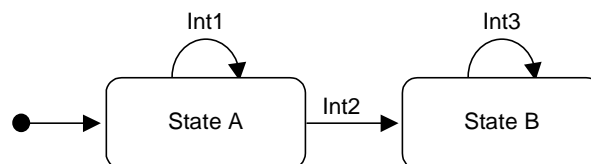


Figure 6-15: Statechart diagram for abstract component

In the refinement of the abstract component into two concrete components, C1 and C2, we have two alternative scenarios for the assignment of the interactions, considering that Int1 should be supported by component C1 and Int3 should be supported by component C2. In the first scenario, Int1 and Int2 are assigned to C1, while Int3 is assigned to C2. In the second scenario, Int1 is assigned to C1, while Int2 and Int3 are assigned to C2.

Figure 6-16 shows a possible refinement of the abstract component statechart diagram for the first assignment scenario. Figure 6-16a shows the statechart diagram for component C1, while Figure 6-16b shows the corresponding statechart diagram for component C2. State A' and State A'' are corresponding states of State A, while State B' and State B'' are corresponding states of State B. Int1', Int2' and Int3' are reference interactions corresponding to Int1, Int2 and Int3. Int4 is an inserted interaction.

Although not supported by the concrete component C2, the interaction Int2' is important for the C2 behaviour. Only after the occurrence of Int2', C2 should change its state to State B'', allowing Int3' to take place. Therefore, there should be a relationship between the occurrence of Int2 and a state change in C2. This relationship is accomplished through the introduction of an interaction between C1 and C2 (see Figure 6-16c), modelled as Int4, which is executed during the transition triggered by the occurrence of Int2'.

Other interactions can also be inserted because the concrete components have to exchange information that previously was centralised. During the refinement, other states can also be inserted for completeness purposes. These states are called inserted states since they were inserted during the refinement process.

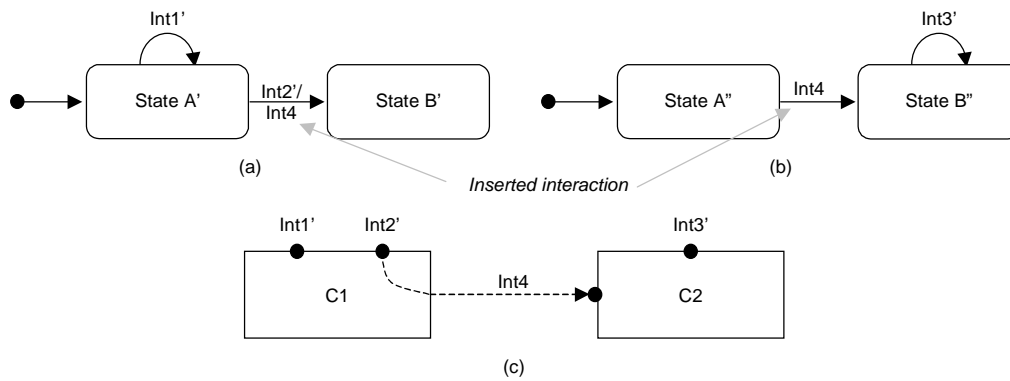


Figure 6-16: Refinement of statechart diagram for abstract component

The representation of the refinement of the statechart diagram for the abstract component, considering the second scenario is shown in Figure 6-17. In this case, C2 would support Int2' and an inserted interaction Int4 would then be initiated by C2, such that the possible occurrence of Int1' in C1 is disabled.

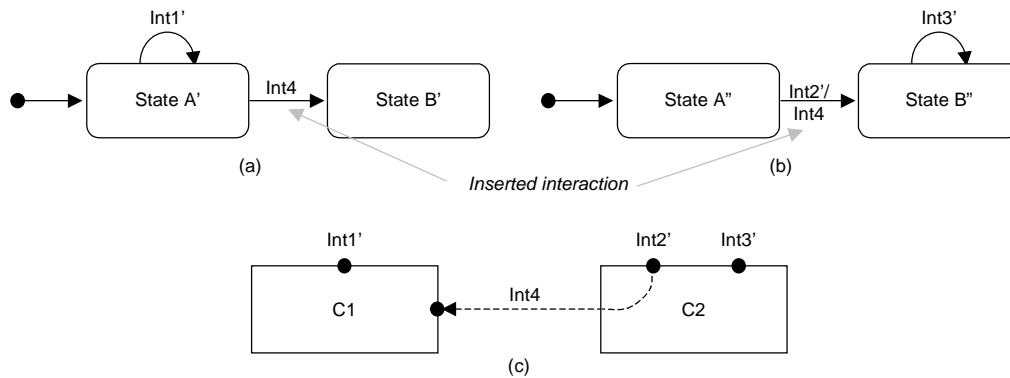


Figure 6-17: Alternative refinement of statechart diagram for abstract component

In order to assess whether or not the statechart diagram of an abstract component was correctly refined, we have to merge the statechart diagrams of the concrete components, abstracting from inserted interactions and states, and compare both diagrams. When merging different statechart diagrams, the following points have to be observed: (1) transitions associated with inserted interactions should be removed; (2) states that have the same corresponding state in the abstract component should be merged; and (3) inserted states should be removed.

Figure 6-18 illustrates the abstraction of two concrete components' statechart diagrams. The resulting abstraction should be compared to the statechart diagram of the abstract component (see Figure 6-15).

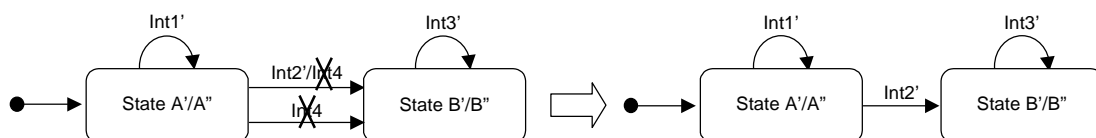


Figure 6-18: Abstraction of concrete components' statechart diagrams

### 6.6.3 Interaction diagram refinement assessment

Similarly to a statechart diagram, an interaction diagram is also used to capture the order in which the interactions involving a component can take place. However, this type of diagram does not represent the state of the component.

Because we only look at the component from the outside, it is easier to assess whether or not the order in which the abstract interactions occur is indeed preserved and respected by the order in which the concrete interactions occur.

Figure 6-19 illustrates two interaction diagrams. Figure 6-19a illustrates a diagram in which an abstract component interacts with its environment. In Figure 6-19b, the abstract component was refined into two concrete components, which interact with each other and with the abstract component environment. The interactions Int1', Int2' and Int3' are reference interactions in the concrete behaviour that correspond respectively to the interactions Int1, Int2 and Int3 in the abstract behaviour. The interaction Int4 is an inserted interaction in the concrete behaviour.

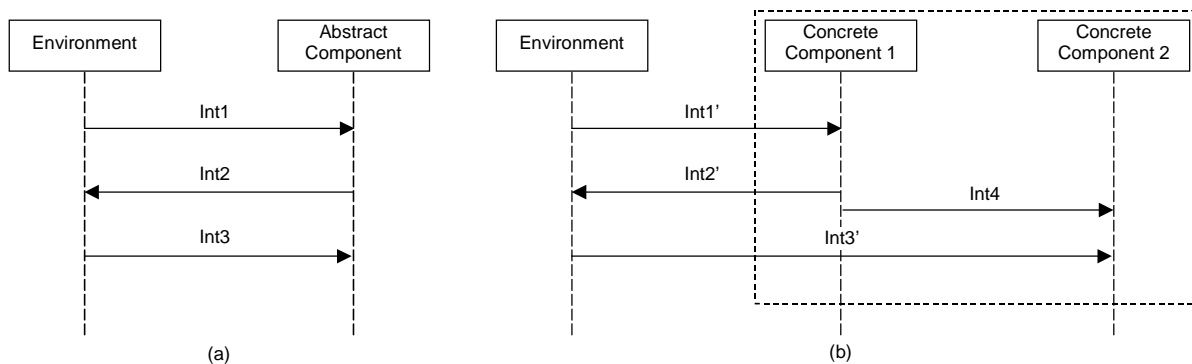


Figure 6-19: Interaction diagram refinement assessment

In order to assess whether or not the interaction diagram of an abstract component was correctly refined, we have to abstract from the independent behaviour of these concrete components, by considering them as a single (abstract) component. In this way, we abstract from (inserted) interactions that take place between the concrete components and compare the reference interactions and their relationships with their corresponding abstract interactions and their relationships.

This process is usually straightforward and in Figure 6-19b we just have to make the interaction that cause Int4 a condition for the interaction caused by Int4 (interaction Int3') and consider Concrete Component 1 and Concrete Component 2 as a single component to obtain a diagram equivalent to the diagram depicts in Figure 6-19a.

### 6.6.4 Activity diagram refinement assessment

An activity diagram is used to capture the order in which the interaction contributions of a component are executed during a state transition, and possibly the order in which actions are executed during this transition. In an activity diagram, interaction contributions are modelled as activities. Therefore, they can be considered as integrated interactions for refinement assessment purposes.

We should be able to abstract from specifications of the same component provided according to the different behaviour perspectives, i.e., abstract from and internal integrated behaviour

specification to obtain a decomposed required service specification and abstract from a decomposed required service specification to obtain a required service specification. Additionally, we should be able to abstract from the behaviour provided by multiple components separately by composing these behaviours into an abstraction of concrete behaviours.

In the refinement of a required service specification into a decomposed required service specification, the abstract interaction contributions are maintained, i.e., each abstract activity corresponds to a unique reference activity, and additional interaction contributions are inserted as a result of the refinement of the relationship between interaction contributions.

In the refinement of a decomposed required service specification into an internal integrated behaviour specification, the abstract interaction contributions are maintained, and additional actions are inserted as a result of the refinement of the relationship between interaction contributions, possibly followed by the refinement of newly inserted actions.

Figure 6-20 depicts the successive refinement of a simple required service specification (Figure 6-20a) into a decomposed required service specification (Figure 6-20b), followed by its refinement into an internal integrated behaviour specification (Figure 6-20c). As a result of each refinement step, reference activities are represented as a solid circle, whereas inserted activities are represented as a dashed circle. For example, in Figure 6-20b A1' and A3' are reference activity units in the refinement of the required service specification, whereas A2 is an inserted activity unit.

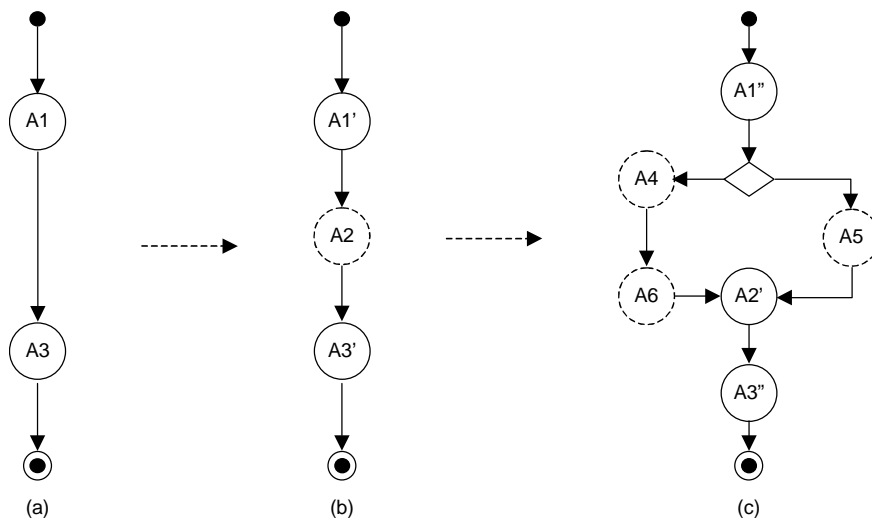


Figure 6-20: Successive refinements

To abstract from a concrete behaviour the following steps should be carried out: (1) identification of reference activities and inserted activities in the concrete behaviour; (2) abstraction from inserted activities; and (3) replacement of reference activities by their abstract activity. These steps should be carried out recursively for each refinement step until an abstraction is obtained that can be compared with the original abstract behaviour.

We can only abstract from an inserted activity unit in case the execution of this activity unit enables another inserted activity unit or the reference activity unit within the same refinement step. However, if the execution of an inserted activity unit enables a concrete activity unit corresponding to another refinement step, we can only abstract from the inserted activity unit if the abstract activity also enables the same concrete activity.

Thus, the basic criterion for comparing an abstract behaviour with the abstraction of a concrete behaviour is the following: the relationships between abstract activity units should be maintained by their corresponding reference activity units.

Figure 6-21 illustrates the abstraction of the behaviour described by Figure 6-20c in two abstraction steps, according to our abstraction guidelines. In the first abstraction step, we abstract from A4, which is an inserted activity that enables another inserted activity, A6. In the second abstraction step, we abstract from A6 and A5, which are inserted activities that enable the reference activity, A2'. As a result, we obtain a behaviour corresponding to the behaviour described by Figure 6-20b.

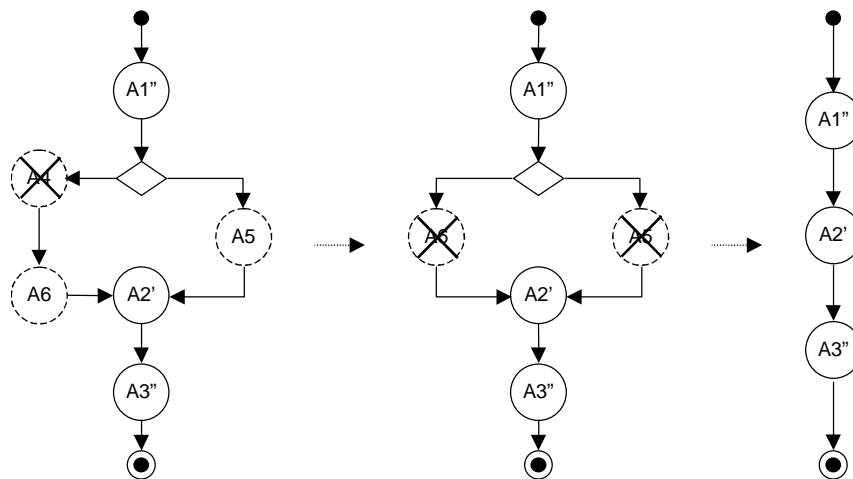


Figure 6-21: Refinement abstraction

The decomposition of the internal integrated behaviour specification of an abstract component into a number of concrete components is usually requires the introduction of a number of internal interactions between the concrete components in order to perform the decomposition properly. These inserted interactions are a result of the refinement of actions, either directly or indirectly.

An inserted interaction is a direct refinement of an action whenever the action is part of the abstract behaviour specification. An inserted interaction is an indirect refinement of an action whenever such an action is not part of the abstract behaviour specification, but rather it is inserted as a result of the refinement of a relationship between abstract actions. In this case, the inserted action is subsequently refined into an interaction.

Figure 6-22 illustrates the difference between the two refinement approaches. An interaction is graphically represented by a dashed arrow, while an action is graphically represented by a circle. A relationship between two activity units is represented by a solid arrow.

In Figure 6-22, an abstract component internal integrated behaviour specification is been refined into two concrete components, C1 and C2. Figure 6-22a shows the direct refinement of action A2 into an interaction, followed by its refinement into two separate interactions, A2' and A2''. Figure 6-22b shows the indirect refinement of the relationship between actions A1 and A2 into the inserted interaction I12 and the indirect refinement of the relationship between actions A2 and A3 into the inserted interaction I23.

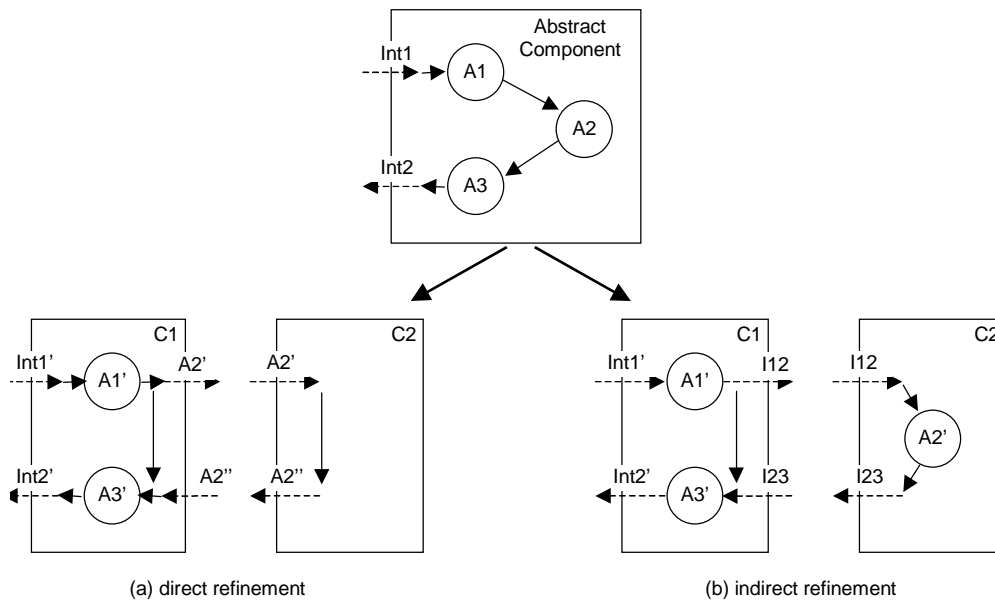


Figure 6-22: Direct versus indirect interaction refinement

In order to assess whether or not an abstract behaviour has been properly decomposed, we have to compose the resulting concrete behaviours. We abstract from the resulting decomposition by abstracting from the inserted interactions and replacing each inserted interaction by a corresponding inserted action. Then, we can abstract from the resulting composed behaviour following the aforementioned guidelines and compare both the original abstract behaviour with the resulting abstraction of the concrete behaviours.

## 6.7 Remaining issues

This section briefly discusses some other issues that are relevant in the overall context of this work, but have not been the main focus of our research.

### 6.7.1 System refinement guidelines

In many cases, a groupware system, or application component according to our component taxonomy, is directly refined into a number of groupware components, instead of into other application components. In the sequel, we describe four general guidelines that can be used in the refinement of most groupware systems into groupware components. The application of these guidelines helps not only the refinement and structuring of these components but also their potential reuse. For further guidelines, we refer to [Hoft98].

#### Avoid unnecessary dependencies between components

This is a quite generic design principle, also known as *orthogonality* [Sind95]. According to this principle we should refine an application component into groupware components whose services are as independent as possible from each other. Therefore, we avoid mixing them up.

If we reduce the external dependencies of a component we also increase its reuse capacity. The less dependent is a component from other components, the easier this component is (independently) reused. Additionally, the component becomes less susceptible to changes, i.e.,

changes in the environment or internal design have minimal impact on the component boundary (service provided).

### **Separate general groupware services from specific services**

Besides refining an application component into a number of application components with minimal dependencies between them, we should allocated general groupware services into separate components. In this way, we separate the specific functionality of an application component from its generic functionality.

There are a number of groupware services that are common to several applications, and therefore are considered general services. Examples of general groupware services include conference management services, coordination services and communication services.

Conference management services refer to the set of services that allow people to manage conference abstractions. A conference is an abstraction of a cooperative work context. In the literature we find this abstraction associated with different terms, such as session, locales, room, place, etc.

Coordination services refer to the set of services that allow people to coordinate their collaboration efforts, avoiding any possible misdoing that may result from the inadequate execution of some cooperative work. One of the most common coordination mechanisms is the use of a token or floor, which must be hold before one is able to execute some work.

Communication services refer to set of services that allow people involved in the collaboration to communicate to each other. Examples of communication services include audio, video, chat, application sharing, etc.

### **Separate communication services into specific components**

Besides separating communication services from other specific and general groupware services, these services should be separated from each other and refined into specific components. Thus, communication services should be decomposed into more elementary building blocks.

For example, an audio communication service should be provided by a specific component, whereas another component can provide a video communication service or a combination of both. A chat communication service should also be provided by a separate component.

### **Allocate general services to the support environment**

Component support environments usually offer a wide range of general-purpose services, such as the event service, the persistency service, the concurrency control service, the transaction service and the security service. Groupware components should not redo these services. Instead, groupware components should in general be concerned with the collaboration logic, leaving the provision of general services to the support environment.

## **6.7.2 Component reuse**

One of the main aspects of component reuse is the integration of reuse into the software development process. In the sequel, we discuss two approaches for component reuse and two of the major aspects associated with it.

### Design with reuse versus design for reuse

Software reuse in general and component reuse in particular has a direct effect in the design process itself. In the literature, we identify two different approaches for reuse [Same97, Vlie00]: *development with reuse* and *development for reuse*. The latter is also known as reuse-driven development.

In the development with reuse approach, component reuse opportunities are identified based on the architectural design of a given software system. After the system specification is refined into a number of components, and the required service of these components is described, a search is carried out for the identification of suitable components, which can be reused with or without modifications. Therefore, this development approach can be considered opportunistic.

Figure 6-23 illustrates the design with reuse approach. After the component architecture of a given system has been specified, a search for suitable components is carried out. In case some candidate components are identified, these components are evaluated and, in case they fit the required service specification, these components are incorporated into the development process.

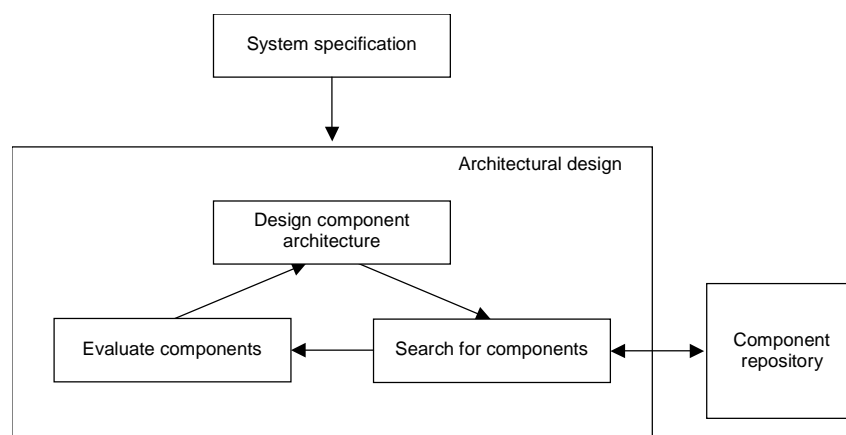


Figure 6-23: Component-based design with reuse

In the development for reuse approach, not only the requirements of a single system are taken into account during the design of the component architecture, but also the requirements of a family of systems belonging to the same domain. Therefore, components are designed for “optimal” reusability within that domain. Sometimes, components are made reusable after the system has been developed, while in other situations components are made reusable beforehand, so that the development of the system specification is already influenced by these components.

Figure 6-24 illustrates the design for reuse approach. After the component architecture of a given system has been specified, a search for suitable components is carried out. In case some candidate components are identified, these components are made reusable according to the requirements of the system domain. The reusable components obtained are evaluated and incorporated into the development process.

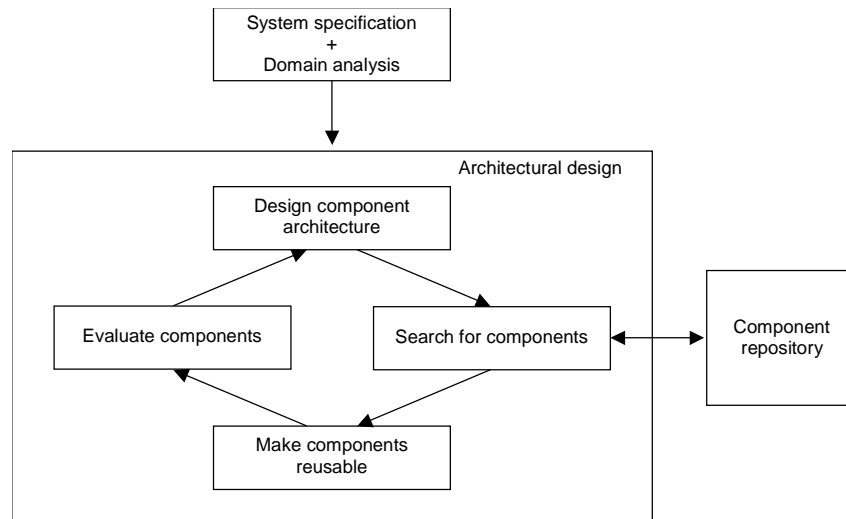


Figure 6-24: Component-based design for reuse

### Reusing components

After the refinement of an application component into groupware components, a major concern is the reuse of existing components.

The identification of reuse opportunities is not as straightforward as it may appear. In the reuse process, some of the issues that must be dealt with include [Vayd99] how to document the functionality of a component, how to find the required component, and how to cope with the need for component modification and adaptation.

The use of UML in the design and development of components facilitates their documentation. A proper documentation of the service provided by a component should include at least a description of supported interfaces, a description of required interfaces, a description of the required events, a description of the provided events, and a description of the component service as a whole. The component internal integrated behaviour specification is desirable as well. Some authors also suggest the inclusion of examples on how to use the component in its documentation [Will00b]. Any sort of UML and non-UML diagrams can be used for such purposes.

Components should be stored in (component) repositories. A *component repository* is a database for the storage and retrieval of components. Index and retrieval techniques can be used to catalogue and find a suitable component in a repository. Repositories can be classified as *local*, *domain-specific* and *reference* [Same97]. A local repository contains general-purpose components. A domain-specific repository contains special-purpose components within an application domain, whereas a reference repository contains references to components in other repositories.

To the best of our knowledge, there is no component repository specialised for groupware components, unless we consider some of the component-based groupware platforms as repositories as well (see Chapter 2, section 2.3).

In an ideal scenario, when searching for a suitable component for reuse we would find a component that provides the same required service. However, such a situation usually does not correspond to the reality.

Frequently, when reusing a component, we may not want to make any changes to this component because these changes may have undesired and unforeseen side-effects. For example, these changes could have an effect on other components that rely on the interfaces and services that are currently supported and provided by this component. In most cases, however, changes are not even possible since we may not have access to the component source code.

To cope with such a situation, we can make use of an adapter or wrapper component. The use of adapter components has been inspired by the adapter design patterns [GHJ+95]. An adapter component works as an intermediary between two incompatible components: a component A, which requires a certain interface, and a component B, which provides an incompatible interface. An adapter component would then provide the interface required by component A, while accessing the interface provided by component B.

Figure 6-25 depicts the use of an adapter component to cope with interface mismatches. A circle represents a component, while an ellipse represents an adapter component. An arrow connecting two components indicates an operation invocation, while a crosscut arrow indicates interface incompatibility.

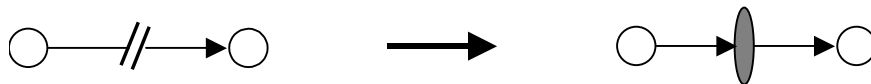


Figure 6-25: Adapter component

Sometimes, simple adaptation does not solve the problem and new functionality has to be added or superfluous functionality removed. If the application developer has access to the component source code, this may be feasible. However, in case the application developer has no access to the component source code, alternative solutions should be used. For example, an approach to provide extensibility for legacy components using a monitoring framework is outlined in [FaDi00].

### Developing reusable components

Any component that is developed can be potentially reused to a certain extent. However, some components are better suited for reuse than others. In principle, there is no guarantee that the application of our methodology in the design of component-based groupware results in a set of components that are optimal for reuse.

The reusability of a component is determined by a number of factors, such as its generality, minimal dependencies, use of standard interfaces and naming conventions, and use of parameters for configuration [Same97]. Some of these issues are discussed elsewhere in this chapter.

A component should be made as independent as possible from the specific application for which it was originally developed. The component should also be general enough to be reused in the design of other applications in the same domain and possibly in other domains.

Some of the techniques that can be used to create general components include *widening* and *narrowing* [Same97]. A component can be generalised by widening its scope, i.e., extending its requirements. Although the component may provide the required functionality for a certain application, we can extend its functionality to cover other applications in the same domain. The drawback of widening is an increase in complexity and development costs. Widening could be used in combination with parameterisation to select a subset of the component func-

tionality that is more appropriate for the specific problem at hand. Narrowing is the opposite of widening. We can narrow the scope of a component to some basic functionality that can be reused by several applications or components.

The main problem associated with the development of reusable components is cost. Creating a reusable component takes longer and therefore is more expensive than making specific components. Estimates of the extra cost associated with developing a reusable component instead of a non-reusable component providing the same required functionality account from 50% to at least 100% [Vlie00]. However, the development costs of individual components are compensated by the increase of speed in developing other applications. Thus, in the long term, the gains stemmed from reusing components surpass the costs of developing these components in the first place [Vayd99].

## 6.8 Conclusion

This chapter focuses on the component level of our design methodology. The component level defines a decomposition of a groupware system into a number of interrelated components, such that the service provided by the groupware system is provided by the composed individual services of these components.

The chapter initially discusses system to component refinements and refinement approaches. Two basic types of refinements are identified: (1) interaction refinement and (2) interaction allocation and flowdown. The latter type of refinement is used at the component level because we consider that the set of interactions between the system and its environment already identified at the system level is maintained at the component level.

The chapter also discusses two alternative component decomposition approaches that can be used at the component level: (1) continuous recursion and (2) discrete recursion. Then, the chapter motivates our choice for a discrete recursion refinement approach based on predefined component categories. We believe that a discrete refinement approach is more suitable than a continuous refinement approach in the context of this work, because it provides a higher degree of certainty and direction during the refinement process. In this sense, three different component categories are identified: (1) simple components, (2) groupware components and (3) application components.

A number of guidelines for refining an application component into groupware components are introduced. Moreover, within the scope of a groupware component, several issues are discussed, including the presence of collaboration concern layers and the implementation of these layers using simple components. We have identified four different layers: (1) interface layer, (2) user layer, (3) collaboration layer and (4) resource layer.

The chapter presents the concept of collaboration coupling between two or more users of a groupware component. We have identified four different levels of coupling based on the concern layers identified: (1) interface coupling, (2) user coupling, (3) collaboration coupling and (4) resource coupling. Each coupling level can be achieved based on centralised, replicated or hybrid architectures.

The use of different component categories at the component level conveys a layering approach similar to the use of concern levels. Each component type addresses a subset of the concerns associated with the component concern level in general. Within each refinement

step, an abstract component is refined into a number of finer-grained concrete components, whose specifications are developed based on the same perspectives defined at the system level.

Since we specify a component based on the required service, decomposed required service and internal integrated perspectives, the techniques used at the system level to model a groupware system according to these perspectives are also used at the component level.

The chapter also introduces the concept of refinement assessment and a number of informal guidelines for assessing the refinement of UML models. These guidelines can be used for refinement assessment not only at the component level, but also at the enterprise and system levels.

Finally, the chapter discusses a number of issues related to component reuse. Two different approaches for component-based development are outlined: (1) development with reuse and (2) development for reuse. In the former approach, first a high-level design and specifications of the required components are produced, followed by a search for suitable components. In the latter approach, the design and specification of a system are already influenced by available components and changes in the original system specification can take place afterwards.

The approach used in our design methodology can be primarily classified as design with reuse. Nevertheless, it also incorporates some of the characteristics of a design for reuse approach, since one of the most important external factors that influence the specification of a groupware system at the component level is the availability of components and component-based platforms.

The availability of existing components is crucial in our refinement approach. Once we identify the service provided by a component, a search for suitable components should be carried out. If a component cannot be reused, the integrated behaviour of this component has to be modelled and, depending on the component category, another level of refinement is required.

The availability of component-based platforms is another important factor that influences the design process at the component level. These platforms serve not only as component repositories, but also as execution environments, providing a number of services that can be used by the components being designed.



---

# Chapter 7

## Component-Based Groupware Profile

This chapter presents a UML profile for component-based groupware design. The chapter provides some background information on the UML profile mechanism, before presenting the profile itself. The profile contains lightweight extensions to the UML metamodel to support the design of groupware systems according to the methodology presented in this work.

This chapter is structured as follows: section 7.1 presents the concept of profile; section 7.2 presents the component-based groupware profile developed in this work; finally section 7.3 presents a summary of the profile.

### 7.1 UML profiles

UML is a modelling language with some built-in extensibility mechanisms that allow one to create new modelling elements to suite certain modelling needs. A coherent set of such extensions, defined according to a specific purpose or domain, constitutes a *profile* [OMG01].

In order to understand the possible extensions that can be made to UML and thus added to a profile, we need to understand the structure of UML itself. UML is defined according to a four-layer architecture, in which each layer is used to produce different kinds of models.

The first layer, the *user object layer*, comprises the actual information that we want to describe. This information corresponds to objects in our daily life, such as a BMW Z8 convertible, Jules Verne's Mysterious Island and a Seiko watch.

The second layer, the *model layer*, defines a language that describes an information domain. The model layer represents a model of a particular domain. For example, we could have a modelling element describing a book in terms of its author, year of publication, ISBN, etc. The user object layer consists of an instance of the model layer.

The third layer, the *metamodel layer*, defines a language to describe models of an information domain. The metamodel layer represents UML itself and therefore contains all UML modelling elements, such as a class, an attribute and a package. The model layer consists of an instance of the metamodel layer.

The forth and topmost layer, called *meta-metamodel layer* or *UML metamodel layer*, defines a language for specifying metamodels. This layer contains modelling elements, such as meta-class and meta-attribute. The metamodel layer consists of an instance of the meta-metamodel layer.

The UML metamodel layer is itself can be seen as an instance of another metamodelling architecture, the Meta-Object Facility (MOF) [OMG00b]. The MOF architecture is also a four-layer architecture that can be used to define description languages in the scope of OMG OMA architecture.

Given this metamodeling architecture, we can extend UML modelling capabilities in two different ways [OMG99b], viz., either specialising the UML metamodel to add new semantics to existing UML modelling elements or changing the MOF model to add new modelling elements to the UML metamodel. The former type of extension is called *lightweight extension* and the latter type of extension is called *heavyweight extension*.

Figure 7-1 illustrates the UML metamodel architecture and how it can be extended. Figure 7-1 also shows some examples of elements belonging to each layer.

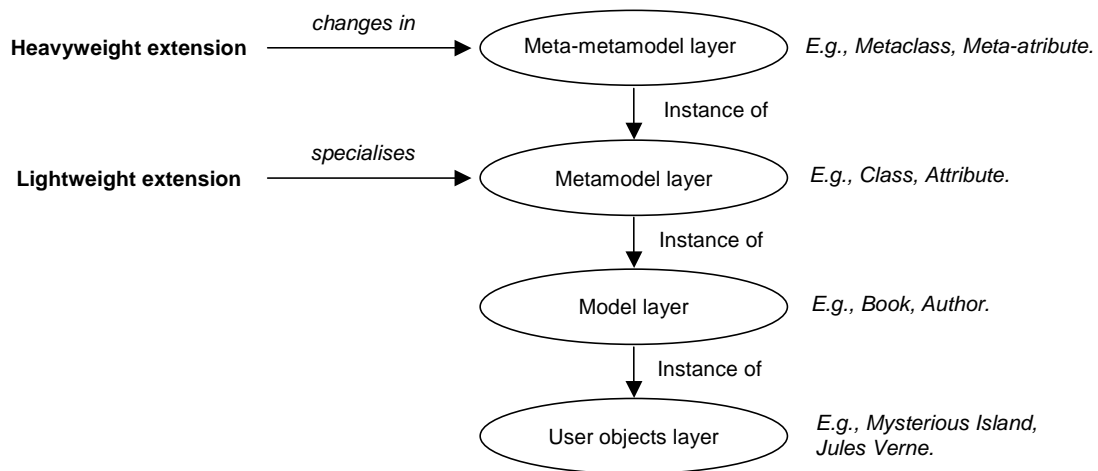


Figure 7-1: UML metamodeling architecture

The mechanisms used to provide lightweight extensions to UML are *stereotypes*, *tag definitions* and *tagged values*, and *constraints*. These mechanisms are generally regarded as lightweight extensibility mechanisms, since they are capable of adapting the UML semantics without changing the UML metamodel.

A stereotype is an extensibility mechanism that allows us to define additional values, constraints and sometimes a different graphical notation by labelling modelling elements, so that they behave in some aspects as if they were instances of new virtual metamodel elements. A stereotype can also be used to indicate a difference in meaning or usage between modelling elements with a similar structure. A stereotype is always applied to specific modelling elements known as base class. A base class represents the name of a UML metamodel class, rather than a user defined class.

A tag definition is an extensibility mechanism that allows us to specify new kinds of properties that can be attached to modelling elements. These properties correspond to tagged values. Thus, a tagged value allows information to be attached to a modelling element in accordance with its tag definition.

A constraint is an extensibility mechanism that allows us to refine the semantics of a modelling element or add new semantics to this element. The semantics of an element is changed via the specification of an expression written in a constraint language, such as OCL, a mathematical notation or plain natural language.

A profile is the proper place to define lightweight as well as heavyweight extensions according to the profile target purpose or domain.

## 7.2 Profile specification

This section presents the profile containing UML extensions to support the design of component-based groupware applications according to the methodology presented in this work. This profile only contains lightweight extensions, which are presented using the UML standard profile notation.

We can group the different types of extensions roughly in three categories: extensions to cope with the design trajectory, extensions to cope with the design of the static structure of entities and extensions to cope with the design of the behaviour aspects of entities.

### 7.2.1 Design trajectory extensions

This section defines a number of stereotypes used to structure the design of a groupware system along the design trajectory of our methodology.

#### Enterprise Model

Stereotype	Base Class	Description
EnterpriseModel <<enterpriseModel>>	Model	Represents the specification of a cooperative work process at the enterprise level.

The notation used for EnterpriseModel is a package stereotyped as <<enterpriseModel>>.

#### System Model

Stereotype	Base Class	Description
SystemModel <<systemModel>>	Model	Represents the specification of a groupware system at the system level.

The notation used for SystemModel is a package stereotyped as <<systemModel >>.

#### Component Model

Stereotype	Base Class	Description
ComponentModel <<componentModel>>	Model	Represents the specification of a groupware system at the component level.

The notation used for ComponentModel is a package stereotyped as <<componentModel >>.

#### Object Model

Stereotype	Base Class	Description
ObjectModel <<objectModel>>	Model	Represents the specification of a groupware system at the object level.

The notation used for ObjectModel is a package stereotyped as <<objectModel >>.

Since the scope of our design methodology comprises the enterprise, system and component levels only, we do not provide guidelines to produce models of a groupware system at the object level. However, we add this stereotype to this profile for the sake of completeness.

### Integrated Perspective

Stereotype	Base Class	Description
IntegratedPerspective <<integratedPerspective>>	Model	Represents an enterprise level specification according to the integrated perspective.

The notation used for IntegratedPerspective is a package stereotyped as <<integratedPerspective>>.

### Distributed Perspective

Stereotype	Base Class	Description
DistributedPerspective <<distributedPerspective>>	Model	Represents an enterprise level specification according to the distributed perspective.

The notation used for DistributedPerspective is a package stereotyped as <<distributedPerspective>>.

### Distributed Perspective with Role Discrimination

Stereotype	Base Class	Description
<i>DistributedPerspectiveRoleDiscrimination</i> <<distributedPerspectiveRoleDiscrimination>>	Model	Represents an enterprise level specification according to the distributed perspective with role discrimination.

The notation used for DistributedPerspectiveRoleDiscrimination is a package stereotyped as <<distributedPerspectiveRoleDiscrimination>>.

### Required Service Perspective

Stereotype	Base Class	Description
<i>RequiredServicePerspective</i> <<requiredServicePerspective>>	Model	Represents a system level or component level specification according to the required service perspective.

The notation used for RequiredServicePerspective is a package stereotyped as <<requiredServicePerspective>>.

### Decomposed Required Service Perspective

Stereotype	Base Class	Description
<i>DecomposedRequiredServicePerspective</i> <code>&lt;&lt;decomposedRequiredServicePerspective&gt;&gt;</code>	Model	Represents a system level or component level specification according to the decomposed required service perspective.

The notation used for `DecomposedRequiredServicePerspective` is a package stereotyped as `<<decomposedRequiredServicePerspective>>`.

### Integrated Internal Perspective

Stereotype	Base Class	Description
<i>IntegratedInternalPerspective</i> <code>&lt;&lt;integratedInternalPerspective&gt;&gt;</code>	Model	Represents a system level or component level specification according to the integrated internal perspective.

The notation used for `IntegratedInternalPerspective` is a package stereotyped as `<<integratedInternalPerspective>>`.

### Structural View

Stereotype	Base Class	Description
<i>StructuralView</i> <code>&lt;&lt;structuralView&gt;&gt;</code>	Model	Represents a model that contains information about the static structure of entities and their interconnection.

The notation used for `StructuralView` is a package stereotyped as `<<structuralView>>`.

### Behavioural View

Stereotype	Base Class	Description
<i>BehaviouralView</i> <code>&lt;&lt;behaviouralView&gt;&gt;</code>	Model	Represents a model that contains information about the behaviour of an entity in isolation.

The notation used for `BehaviouralView` is a package stereotyped as `<<behaviouralView>>`.

### Interactional View

Stereotype	Base Class	Description
<i>InteractionalView</i> <code>&lt;&lt;interactionalView&gt;&gt;</code>	Model	Represents a model that contains information about the behaviour of multiple entities as they interact with each other.

The notation used for `InteractionalView` is a package stereotyped as `<<interactionalView>>`.

### 7.2.2 Structural extensions

This section defines a number of stereotypes used to model the static structure of functional entities along the design trajectory of our methodology.

#### Functional Entity

Stereotype	Base Class	Description
<i>FunctionalEntity</i> «entity»	Class	Represents an entity capable of executing behaviour.

The notation used for FunctionalEntity is a class stereotyped as «entity».

#### Functional Role

Stereotype	Base Class	Description
<i>FunctionalRole</i> «role»	Class	Represents a description of some functional behaviour that can be performed by a functional entity.

The notation used for FunctionalRole is a class stereotyped as «role».

#### Action Definition

Stereotype	Base Class	Description
<i>ActionDefinition</i> «action»	Class	Represents an activity unit whose execution is carried out exclusively by a single functional entity.

The notation used for ActionDefinition is a class stereotyped as «action».

#### Interaction Definition

Stereotype	Base Class	Description
<i>InteractionDefinition</i> «interaction»	Class	Represents an activity unit whose execution is carried out by two or more functional units in cooperation.

The notation used for InteractionDefinition is a class stereotyped as «interaction».

#### Exception Definition

Stereotype	Base Class	Description
<i>ExceptionDefinition</i> «exception»	Class	Represents a possible abnormal situation that may arise during the execution of some behaviour. This stereotype should be used to organize exceptions in a hierarchy.

The notation used for ExceptionDefinition is a class stereotyped as «exception».

**Resource**

Stereotype	Base Class	Description
<i>Resource</i> «resource»	Class	Represents all sort of things that can be represented as a piece of information and are either used, created, destroyed or changed by an activity unit.

The notation used for Resource is a class stereotyped as «resource».

**Datatype**

Stereotype	Base Class	Description
<i>Datatype</i> «datatype»	Class	Represents an abstract datatype definition. This stereotype should be used in the elaboration of an information model.

The notation used for Datatype is a class stereotyped as «datatype».

**System Structure**

Stereotype	Base Class	Description
<i>SystemStructure</i> «system»	Package	Container for models of a functional entity at the system level.

The notation used for SystemStructure is a package stereotyped as «system».

**Component Structure**

Stereotype	Base Class	Description
<i>ComponentStructure</i> «component»	Package	Container for models of a functional entity at the component level.

The notation used for ComponentStructure is a package stereotyped as «component».

**7.2.3 Behavioural extensions**

This section defines a number of stereotypes used to model the behaviour of functional entities along the design trajectory of our methodology.

**Action**

Stereotype	Base Class	Description
<i>Action</i> «action»	Action	Represents the execution of an activity unit by a single functional entity.

The notation used for Action is an action state stereotyped as «action».

**Interaction**

Stereotype	Base Class	Description
<i>Interaction</i> <<interaction>>	Action	Represents the integrated execution of an activity unit by two or more functional entities in cooperation.

The notation used for Interaction is an action state stereotyped as <<interaction>>.

**Interaction Contribution**

Stereotype	Base Class	Tags	Description
<i>InteractionContribution</i>	Action	isInitiator, isReactor	Stereotype used to define different types of interaction contributions based on the roles of interaction initiator and interaction reactor.

This stereotype has no visual notation.

**Is Initiator**

Tag	Stereotype	Type	Multiplicity	Description
isInitiator	<i>InteractionContribution</i>	UML::Datatypes::Boolean	1	Tag used to identify the interaction initiator role.

**Is Reactor**

Tag	Stereotype	Type	Multiplicity	Description
isReactor	<i>InteractionContribution</i>	UML::Datatypes::Boolean	1	Tag used to identify the interaction reactor role.

**Request Interaction**

Stereotype	Base Class	Parent	Description
<i>RequestInteraction</i> <<requestInteraction>>	Action state, Event	Interaction- Contribution	Represents an interaction contribution in which the execution of some functionality is requested.
Constraints			
If isInitiator = true than stereotype can only be applied to an action state base class. If isReactor = true than stereotype can also be applied to an event base class. isInitiator <> isReactor			

The notation used for RequestInteraction is either an action state or an event stereotyped as <<requestInteraction>>. In case the tagged value is omitted, this information can be inferred by the context of use.

### Response Interaction

Stereotype	Base Class	Parent	Description
<i>ResponseInteraction</i> <<responseInteraction>>	Action state, Event	Interaction- Contribution	Represents an interaction contribution in which the result of the execution of a previously requested functionality is informed.
<b>Constraints</b>			
If isInitiator = true than stereotype can only be applied to an action state base class. If isReactor = true than stereotype can also be applied to an event base class. isInitiator <> isReactor			

The notation used for ResponseInteraction is either an action state or an event stereotyped as <<responseInteraction>>. In case the tagged value is omitted, this information can be inferred by the context of use.

### Exception

Stereotype	Base Class	Parent	Description
<i>Exception</i> <<exception>>	Action state, Event	ResponseIn- teraction	Represents a special type of response interaction contribution that indicates the occurrence of an abnormal situation during the execution of the requested functionality.
<b>Constraints</b>			
Same as parent.			

The notation used for Exception is either an action state or an event stereotyped as <<exception>>. In case the tagged value is omitted, this information can be inferred by the context of use.

### Invocation Interaction Pattern

Stereotype	Base Class	Parent	Description
<i>InvocationInteractionPattern</i> <<invocationInteractionPattern>>	Action state, Event	Interaction- Contribution	Represents two interaction contributions of an instance of the invocation interaction pattern: the interaction contributions of a request interaction and the interaction contribution of a response interaction.
<b>Constraints</b>			
isInitiator = true and isReactor = true.			

The notation used for InvocationInteractionPattern is either an action state or an event stereotyped as <<invocationInteractionPattern>>.

**Notification Interaction Pattern**

Stereotype	Base Class	Parent
<i>NotificationInteractionPattern</i>  <<notificationInteractionPattern>>, <<notificationInteraction>>	Action state, Event	InteractionContribution
Description		
Represents the interaction contribution of an instance of the notification interaction pattern regarding a single event notification interaction. It can also represent the interaction contribution of the corresponding event notification interaction, which is used to notify the occurrence of noteworthy situations.		
Constraints		
If isInitiator = true than stereotype can only be applied to an action state base class. If isReactor = true than stereotype can also be applied to an event base class. isInitiator <> isReactor		

The notation used for NotificationInteractionPattern is either an action state or an event stereotyped as either <<notificationInteractionPattern>> or <<notificationInteraction>>. In case the tagged value is omitted, this information can be inferred by the context of use.

**System**

Stereotype	Base Class	Parent	Description
<i>System</i>  <<system>>	Class	FunctionalEntity	Represents the behaviour of a functional entity at the system level. This behaviour is the result of the composed behaviour of a set of fine-grained components.
Constraints			
All the operations associated with an entity should be described by an interface. An entity cannot have operations of its own.			

The notation used for System is a class stereotyped as <<system>>.

**Component**

Stereotype	Base Class	Parent	Description
<i>Component</i>  <<component>>	Class	FunctionalEntity	Represents the behaviour a functional entity at the component level. This behaviour may be the result of the composed behaviour of a set of fine-grained components.
Constraints			
All the operations associated with an entity should be described by an interface. An entity cannot have operations of its own.			

The notation used for Component is a class stereotyped as <<component>>.

### 7.3 Summary of profile

This profile contains a number of lightweight extensibility mechanisms that can be used to design a groupware application according to our design methodology. This profile introduces stereotypes and tag definitions. Constraints are not defined.

We introduce only lightweight extensions for a number of reasons:

- most of the existing UML support tools can better cope with lightweight extensions than with heavyweight extensions. In this way, we can use functionality already available in these tools to create and manipulate lightweight extensions instead of implementing additional functionality to cope with heavyweight extensions;
- there is an ongoing effort under the auspices of OMG to produce a new version of UML (UML 2.0) with, amongst others, enhanced capability for component-based development [Kobr99]. We believe that it is easier to adapt lightweight extensions to cope with the forthcoming UML standard;
- finally, we believe that the extensions introduced here improve sufficiently the UML capability to model component-based groupware in the context of this work. However, this does not exclude by any means the need for general improvements in UML in order to support component-based development and consequently the ongoing OMG standardisation efforts towards the development of UML 2.0.

Table 7-1 summarises the tags defined by this profile.

Tag	Stereotype
isInitiator	InteractionContribution
isReactor	InteractionContribution

*Table 7-1: Summary of tags*

Table 7-2 summarises the stereotypes defined by this profile.

Stereotype	Base Class	Notation
Action	Action	<<action>>
Action Definition	Class	<<action>>
BehaviouralView	Model	<<behaviouralView>>
Component	Class	<<component>>
ComponentModel	Model	<<componentModel>>
ComponentStructure	Package	<<component>>
Datatype	Class	<<datatype>>
DecomposedRequiredServicePerspective	Model	<<decomposedRequiredServicePerspective>>
DistributedPerspective	Model	<<distributedPerspective>>
DistributedPerspectiveRoleDiscrimination	Model	<<distributedPerspectiveRoleDiscrimination>>
EnterpriseModel	Model	<<enterpriseModel>>
Exception	Action state, Event	<<exception>>
ExceptionDefinition	Class	<<exception>>
FunctionalEntity	Class	<<entity>>
FunctionalRole	Class	<<role>>
IntegratedPerspective	Model	<<integratedPerspective>>
Interaction	Action	<<interaction>>
InteractionDefinition	Class	<<interaction>>
InteractionContribution	Action	N/A
InteractionalView	Model	<<interactionalView>>
InterfaceOperationSpecification	Model	<<interfaceOperationSpecification>>
InternalIntegratedPerspective	Model	<<internalIntegratedPerspective>>
InvocationInteractionPattern	Action state, Event	<<invocationInteractionPattern>>
NotificationInteractionPattern	Action state, Event	<<notificationInteractionPattern>> or <<notificationInteraction>>
ObjectModel	Model	<<objectModel>>
RequestInteraction	Action state, Event	<<requestInteraction>>
RequiredServicePerspective	Model	<<requiredServicePerspective>>
Resource	Class	<<resource>>
ResponseInteraction	Action state, Event	<<responseInteraction>>
StructuralView	Model	<<structuralView>>
System	Class	<<system>>
SystemModel	Model	<<systemModel>>
SystemStructure	Package	<<system>>

Table 7-2: Summary of stereotypes

---

# Chapter 8

## Electronic Meeting System: a Case Study

This chapter presents a case study consisting of the architectural modelling of an electronic meeting system (EMS) according to our methodology. This system provides both chat and voting functionality to a number of users engaged in a common task. The purpose of this case study is to illustrate all the steps defined in our methodology.

This chapter is structured as follows: section 8.1 presents the collaboration context that serves as basis for the design process; section 8.2 presents the specification of the EMS at the enterprise level; section 8.3 presents the specification of the EMS at the system level; section 8.4 presents the specification of the EMS at the component level; finally, section 8.5 draws some conclusions. Due to the complexity and length of the case study, only some (small) parts of these specifications are discussed.

### 8.1 Collaboration context

An electronic meeting system (EMS) should be designed to allow a number of distributed users to exchange messages asynchronously and to vote on a number of issues. Because our methodology does not support requirements gathering explicitly, our starting point is a description of the context of collaboration. This context is informally described first as a usage scenario that succinctly describes the intended or possible use of the EMS, and then as separate sets of collaboration features in a table-like format.

#### 8.1.1 Usage scenario

The case study we consider, computational support should be designed for a group of people engaged in a common task. Let's assume, for example, the Architecture of Distributed Systems (ADS) group of the University of Twente, which has several teaching and research responsibilities.

Marten van Sinderen, a senior researcher of the ADS group, is co-chairing the 2000 International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS). In order to get the group actively involved with the organisation of this event Marten creates an electronic meeting session called IDMS and invites the other members of the ADS group to participate in the arrangements of this event.

The IDMS session works primarily as a discussion forum, in which the participants make suggestions for the arrangements, discuss important dates, make commitments and reminders and inform what has been done via message exchange. The IDMS session works also as a decision-making forum, in which a poll can be opened and participants can vote anonymously while the poll is open. After the poll is closed, participants can check and discuss its results.

The session is created some months before the conference and lasts until the conference is over. Further, whenever a participant joins the IDMS session, he can be updated with the contents of the discussion and existing polls.

After the conference has taken place, Marten terminates the IDMS session and creates a new electronic meeting session to discuss the organisation of a new event, the 2001 International Conference on Protocols for Multimedia Systems (PROMS).

### 8.1.2 Collaboration features

The EMS collaboration features have been grouped in a number of sets, which are described in the sequel.

#### Feature 01: Collaboration should be restricted to registered personnel

<i>Feature</i>	<i>Description</i>
User registration	Only registered users are allowed to collaborate.
User logging	A registered user has to log on using a password in order to start collaborating. A logged user stops collaborating when he logs off.
Password changes	A logged user should be able to change his password.
Registration removal	A registered user should be able to remove his registration. A user should only be allowed to remove his registration, if the user is not logged. Registration removal should be password protected.

#### Feature 02: Collaboration should take place in the context of a cooperative session

<i>Feature</i>	<i>Description</i>
Session creation	A logged user should be able to create a cooperative session. A session should have a name that indicates the purpose of the session. A user should be able to create multiple sessions, with distinct name. The user that creates a session becomes the session controller.
Participation through invitation	The session controller is also a participant in the session. However, the participation of other users should be restricted by invitation. A participant should be able to invite other users to a session. A logged user should be notified whenever he has been invited to the session. The user only becomes a participant in that session if he accepts the invitation. A user can participate in multiple sessions.
Session attendance	A participant should join the session to start collaborating. When a participant joins the session, he becomes active. A participant remains active until he leaves the session. Only an active participant can invite other users to the session. However, an active participant cannot create another session. Further, the participant must first leave the session before logging off.
Participant exclusion	The session controller should be able to exclude a participant from the session, provided the controller is active.
Session awareness	Once a participant joins a session, he should be informed of the other participants in the session and those who are currently active. All active participants should be notified whenever a new participant is added to or excluded from the session. All active participants should be notified whenever a participant joins or leaves the session.
Session termination	The session controller should be able to terminate the session, provided there are no active participants in the session.

**Feature 03: Collaboration can be achieved through the exchange of messages**

<i>Feature</i>	<i>Description</i>
Message exchange	An active participant should be able to send a message to the other participants of a session. Other active participants should receive the message as soon as possible. The sender of a message should be identified. Messages should be organised according to the session in which they were sent.
Message update	Once a participant joins a session, he should be able to retrieve all the messages that were exchanged during the collaboration.
Message persistence	All messages exchanged should be kept until the session is terminated.

**Feature 04: Poll creation**

<i>Feature</i>	<i>Description</i>
Opening polls	The session controller should be able to open a poll, provided he is active. It should be possible to open multiple polls in the lifetime of a session, however only one poll can be open at a time, i.e., if there is an open poll, this poll has to be closed first before a new poll can be opened.
Poll format	A poll should have a question and a number of alternatives. The question is the issue to be decided while the alternatives are all the possible answers. A voter should choose only one of the presented alternatives.
Closing polls	The session controller should be able to close a poll, provided he is active. Once a poll is closed, participants are not able to vote on that poll anymore. Once a poll is closed it cannot be reopened.
Poll awareness	Active participants should be notified when polls are opened or closed. Once a participant joins a session, he should be informed of an open poll.
Poll persistence	The results of all closed polls should be available for consultation by the participants of the session until the session is terminated.

**Feature 05: Collaboration can be achieved through the cast of votes**

<i>Feature</i>	<i>Description</i>
Casting votes	Any active participant should be able to cast a vote on an open poll. A vote should contain the voter's choice according to the alternatives.
Vote validation	A vote has to be valid to be counted. A valid vote contains one and only one of the alternatives of the poll.
Vote accuracy	The vote of a participant should be counted only once per poll. A participant should not be allowed to change his vote once it has been cast. It should not be possible for a cast vote to be altered and it should not be possible for a cast vote to be eliminated from the final tally.
Vote privacy	It should not be possible to link a cast vote to a participant.
Pool update	Once a participant joins a session, he should be able to check the status of polls.
Poll results	Any active participant should be able to check the results of a closed poll.

**8.2 Enterprise level specification**

We developed three different enterprise level specifications of the EMS, each one according to a separate perspective as defined at the enterprise level. This section presents an overview

of these specifications, with emphasis on the integrated and decomposed perspective specifications.

### 8.2.1 Integrated perspective specification

The enterprise level specification of EMS according to the integrated perspective was developed based on the collaboration features described in section 8.1. For each set of features, the structural view was initially described in a series of concept diagrams, covering the different types of concept diagram, i.e., entity-role, role-activity and activity-resource diagrams.

According to this perspective there is a single functional entity, EMS Enterprise, representing the enterprise itself. This entity is associated with a single functional behaviour, EMS Enterprise Role, describing all the activity units and their relationship that are relevant for the collaboration context being described. A single entity-role diagram, common to all five different sets of features was developed (see Figure 8-1).

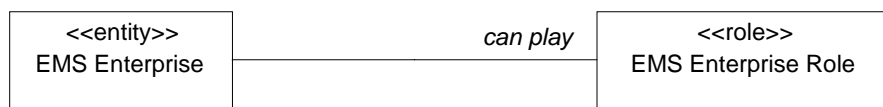


Figure 8-1: Entity-role diagram at the integrated perspective specification

EMS Enterprise is a functional entity that abstractly represents all the functional entities that are will be part of the enterprise in a lower abstraction level. Similarly, EMS Enterprise Role describes the integrated behaviour that can be played by the EMS Enterprise.

For each set of features, at least one role-activity diagram was developed to capture the activity units that are part of the EMS Enterprise Role. These activity units are modelled as actions at this abstraction level.

Figure 8-2 shows a role-activity diagram that captures some of the actions identified in the context of the second set of collaboration features. These actions refer to the creation and termination of a collaboration context, which is represented by a session, and to the activities used to engage and disengage in a collaboration context.

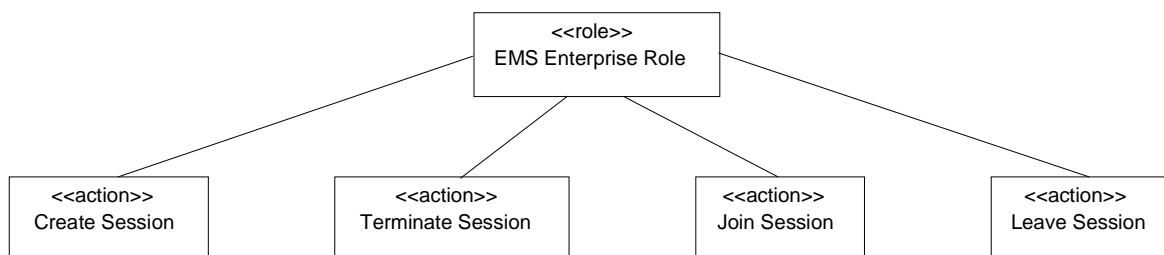


Figure 8-2: Role-activity diagram at the integrated perspective specification

Other actions identified as part of the EMS Enterprise Role include Register, Log On, Invite User, Exclude Participant, Send Message, Open Poll, Cast Vote, Close Poll, etc.

For each role-activity diagram developed, one or more activity-resource diagrams were developed to capture the information used or established during the execution of an activity unit. For example, Figure 8-3 illustrates the activity-resource diagram developed for action Create Session. The execution of this activity unit uses a single resource, Basic Session Info, repre-

senting the information used or established or both. Basic Session Info is structured into three separate resources, viz., Session Subject Info, User Identification Info and Session Id Info. These resources represent the information concerning the session subject, controller and identification, respectively.

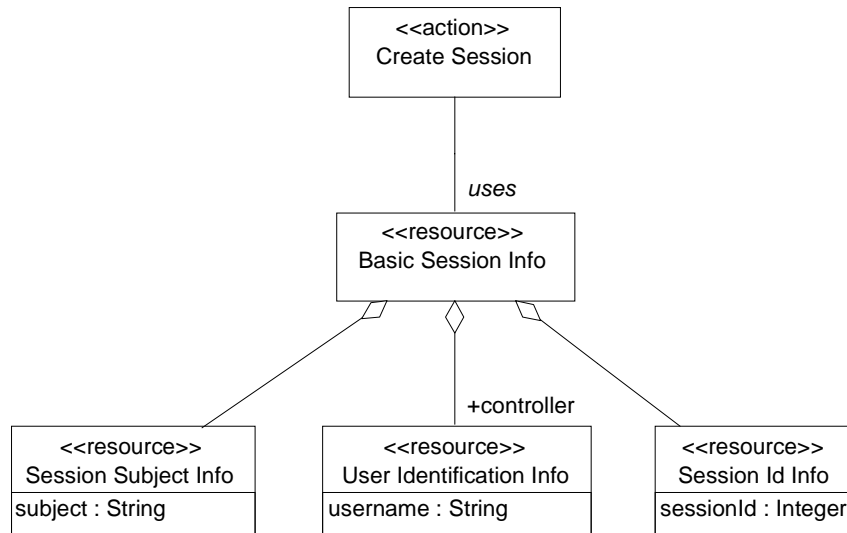


Figure 8-3: Activity-resource diagram at the integrated perspective specification

Finally, the structural view description was completed with the documentation of the concepts identified in the diagram using a glossary. For each concept captured in the concept diagrams, a separate entry was added to the glossary.

Figure 8-4 shows the entries describing the activity units depicted in Figure 8-2.

Name	Type	Description	Context
<i>Create Session</i>	Action	Activity performed in order to create a context of collaboration to host some cooperative activity.	Integrated perspective specification/Enterprise
<i>Join Session</i>	Action	Activity performed in order to engage in an existing context of collaboration.	Integrated perspective specification/Enterprise
<i>Leave Session</i>	Action	Activity performed in order to stop the participation in a cooperative activity.	Integrated perspective specification/Enterprise
<i>Terminate Session</i>	Action	Activity performed in order to bring a context of collaboration to an end.	Integrated perspective specification/Enterprise

Figure 8-4: Glossary of terms at the integrated perspective specification

The behavioural view was captured in a number of activity diagrams. For each set of features, a separate activity diagram was developed to capture the relationship between the activity units captured in the corresponding role-activity diagrams. These diagrams were not only used to capture the relationship between these activity units, but also used to relate the activity units captured in the different sets of features. For example, Figure 8-5 illustrates the activity diagram used to capture the relationship between the activity units depicted in Figure 8-2. Nevertheless, this diagram also captures the relationship between these activity units and two other activity units identified in the context of the first set of collaboration features, viz., actions Log On and Log Off. According to this diagram, only after the execution of the action Log On and before the execution of the action Log Off, the actions Create Session, Terminate Session, and Join Session can be executed. Further, the action Leave Session should follow the execution of the action Join Session.

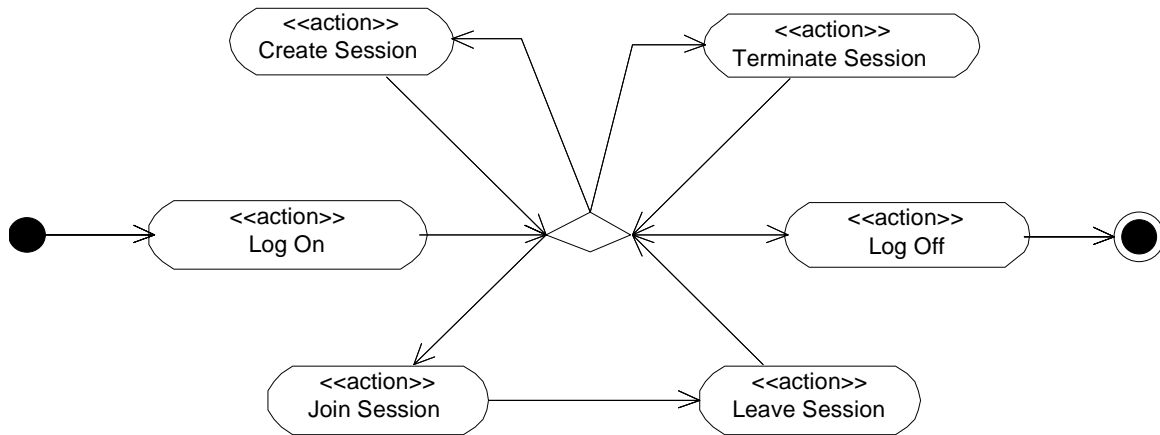


Figure 8-5: Activity diagram at the integrated perspective specification

The diagram in Figure 8-5 uses the same pseudo-state to represent both decision and merge elements in a non-deterministic choice between the execution of the various activities at a high abstraction level. A transition is triggered by the completion of an activity. However, since there are multiple transitions that are not labelled, these transitions are all enabled at the same time as a result. These transitions are said to be in conflict with one another, but only one of them must fire. The choice of which transition should fire is non-deterministic and indicates the activity to be executed.

### 8.2.2 Decomposed perspective specification

The enterprise level specification of EMS according to the decomposed perspective consists of a refinement of the integrated perspective specification. During this refinement step, both the functional entity EMS Enterprise and functional role EMS Enterprise Role were refined.

The functional entity EMS Enterprise was refined into two separate functional entities, viz., Architecture Group and EMS. EMS is a single entity that represents the system being designed, while Architecture Group is a collective entity that represents all the persons capable of interacting with the system according to our usage scenario.

Figure 8-6 shows the refinement of the entity EMS Enterprise into the entities Architecture Group and EMS.

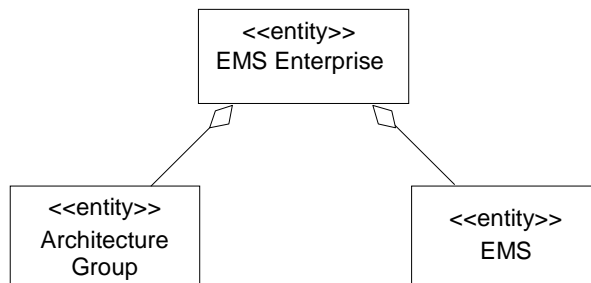


Figure 8-6: Functional entities at the decomposed perspective specification

The functional role EMS Enterprise Role was refined into a number of functional roles, viz., User, Access Manager, Session Manager, Invitation Manager, Chat Manager and Voting Manager. The associations between the functional entities and functional roles were established by a number of entity-role diagrams developed in the context of each set of collaboration features.

Basically, the functional entity Architecture Group can play the functional role User, while the functional entity EMS can play the remaining functional roles, because later they will relate to system responsibilities.

The functional role User, which describes the behaviour of any potential user of the EMS at this abstraction level, was then specialised to distinguish between the behaviours of registered and unregistered users, logged and unlogged users, session participants and controller, and so on. The remaining functional roles describe the behaviour that can be automated to provide the required computer support for the execution of the identified activity units. At this abstraction level, it would suffice to have a single functional role describing all the activity units that the EMS can perform. Nevertheless, an early identification of sub-roles or sub-behaviours can be useful in the decomposition of the system into components (see section 8.4). Particularly, such a refinement facilitates the structuring and specification of behaviour at the enterprise level, given the limitations of UML activity diagrams in the specification of complex behaviours.

The structural view was complemented with the role-activity and activity resource diagrams for each set of collaboration features and the addition of new terms to the glossary. In the role-activity diagrams, all the actions identified in the integrated perspective specification were refined into interactions. Some actions were refined directly into interactions, while other actions were first refined into more concrete actions, which were then refined into interactions. For example, action Create Session was refined into interaction `_Create Session`, which is shared by the functional roles Inactive User and Session Manager, while action Join Session was refined into interactions `_Join Session` and `_Join Session Notification`, which are shared by the functional roles Inactive Participant and Session Manager, and Active Participant and Session Manager, respectively (see Figure 8-7).

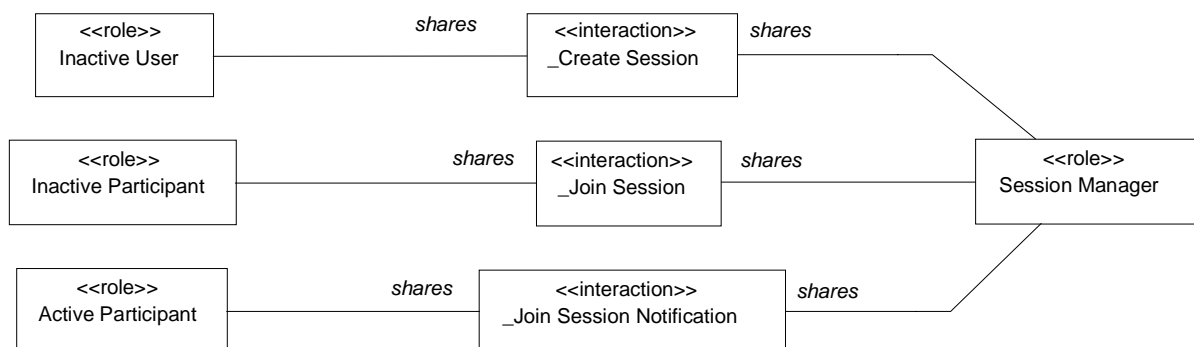


Figure 8-7: Role-activity diagram at the decomposed perspective specification

In the activity-resource diagrams, the same informational resources used by an action at the integrated perspective are used by its corresponding interaction(s).

The behavioural view was captured by one or more activity diagrams for each functional role. Sometimes, a single activity diagram was developed for two or more roles played by the same functional entity. In those cases, the behaviour associated with each separate role is distinguished from another via a swimlane.

Figure 8-8 shows an activity diagram developed for the functional role Session Manager. This diagram describes that either the interaction `_Join Session` or the interaction `_Leave Session` can be executed. In case the former is executed, the interactions `_Participation Notification` and

\_Join Session Notification are executed as a result, while in case the latter is executed, the interaction \_Leave Session Notification is executed as a result.

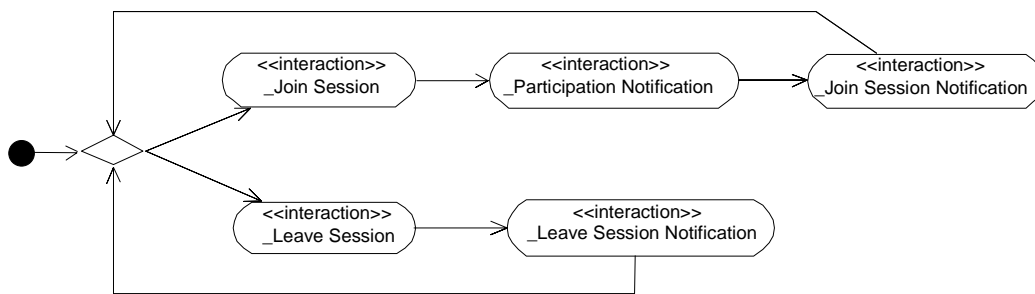


Figure 8-8: Activity diagram at the decomposed perspective specification

We do not identify interaction contributions at the enterprise level, and we only identified interactions in the development of the decomposed perspective specification. Therefore, the interaction view specification corresponds to the integrated description that would be obtained combining all activity diagrams of the behavioural view, abstracting from the different functional roles.

### 8.2.3 Decomposed perspective with role discrimination specification

The enterprise level specification of EMS according to the decomposed perspective with role discrimination consists of a refinement of the integrated perspective specification. The decomposed perspective with role discrimination specification is similar to the decomposed perspective specification. Since all the interactions identified in the latter were already two-party interactions, we simply identified the roles taken by the functional roles that share these interactions in their execution. The behavioural and interactional views descriptions were unchanged.

Figure 8-9 depicts a role-activity diagram at the decomposed perspective with role discrimination specification. This diagram shows two interactions, \_Open Poll and \_Close Poll, shared by the same functional roles, Active Controller and Poll Manager. The diagram also shows that in both cases the functional role Active Controller plays the role of actor in the execution of these interactions, while the functional role Poll Manager plays the role of reactor.

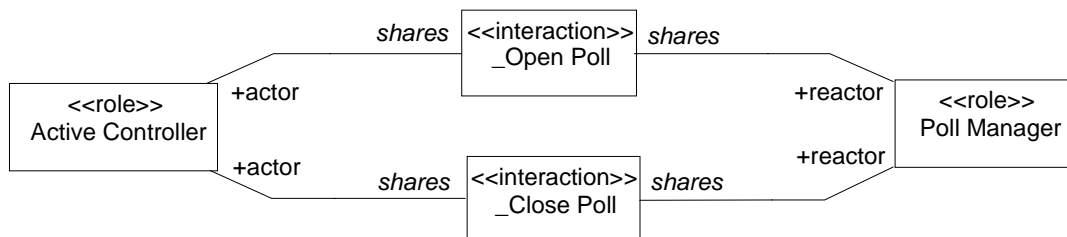


Figure 8-9: Diagram at the decomposed perspective with role discrimination specification

## 8.3 System level specification

The system level specification was developed in two steps. First, we developed the system required service specification and then, since no external support was required, we developed the system internal behaviour specification.

### 8.3.1 Required service specification

The development of the system level specification according to the required service perspective was based on the enterprise level specification according to the decomposed perspective with role discrimination. Since all the functional entities and functional roles identified in the reference enterprise specification are related with the EMS, the system level specification is a complete refinement of the reference enterprise level specification.

The structural view was initially captured by use case diagrams. In these diagrams, actors and use cases were identified based on the functional roles and interactions defined in the reference enterprise specification. The functional roles that can be played by the functional entity Architecture Group were mapped onto three different actors, viz., EMS User, EMS Participant and EMS Controller. Based on the generalisation/specialisation relationships established at the enterprise level specification, generalisation/specialisation relationships were also established at the system level specification as follows: EMS Controller is a specialisation of EMS Participant, which is a specialisation of EMS User.

Since the functional roles that can be played by the functional entity EMS describe the behaviour of the EMS itself, these functional roles were not mapped onto actors. However, the activity units defined in the context of these functional roles were mapped onto use cases. Since only interactions were identified, these interactions were mapped to use cases on a one-to-one basis or a many-to-one basis. For example, the interaction `_Cast Vote` was mapped onto the use case `Control Voting`, while the interactions `_Retrieve Messages` and `_Send Message` were both mapped onto the use case `Exchange Message`. Some relationships were established between the identified use cases, and between use cases and associated actors.

Figure 8-10 shows a use case diagram representing use cases related to the exchange of messages and the opening and closing of polls.

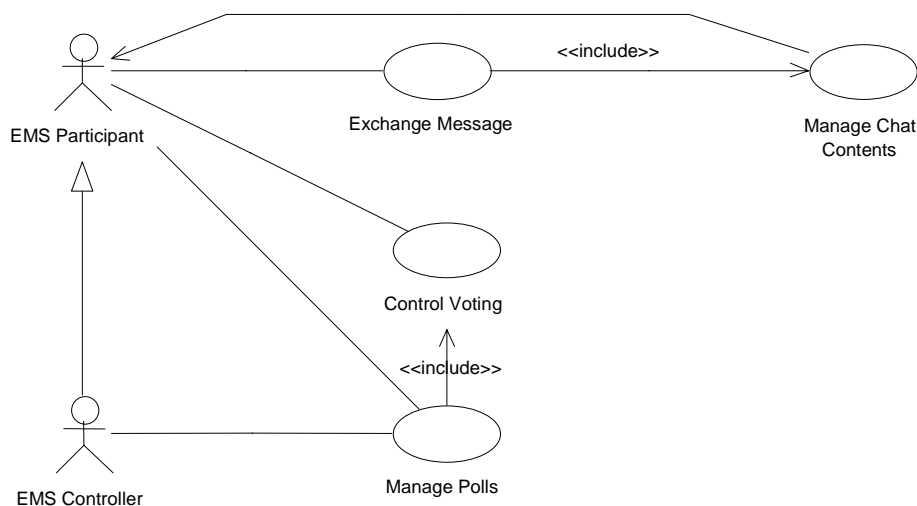


Figure 8-10: Use case diagram at the system required service specification

The use cases identified were textually described using our suggested convention in terms of static interaction contributions and their relationships, i.e., no internal action was described during the development of the system required service specification.

Based on the use cases diagrams, a package diagram was developed containing two functional entities and their relationship, viz., EMS and EMS User. These functional entities represent the system being designed and its users, respectively.

At this point we develop an informational model of the system in order to describe the relationship between the information exchanged during interactions.

Figure 8-11 shows a class diagram describing part of the information model developed at the system level. This diagram shows all the datatypes we identified and their relationships, except for the datatypes related to polls.

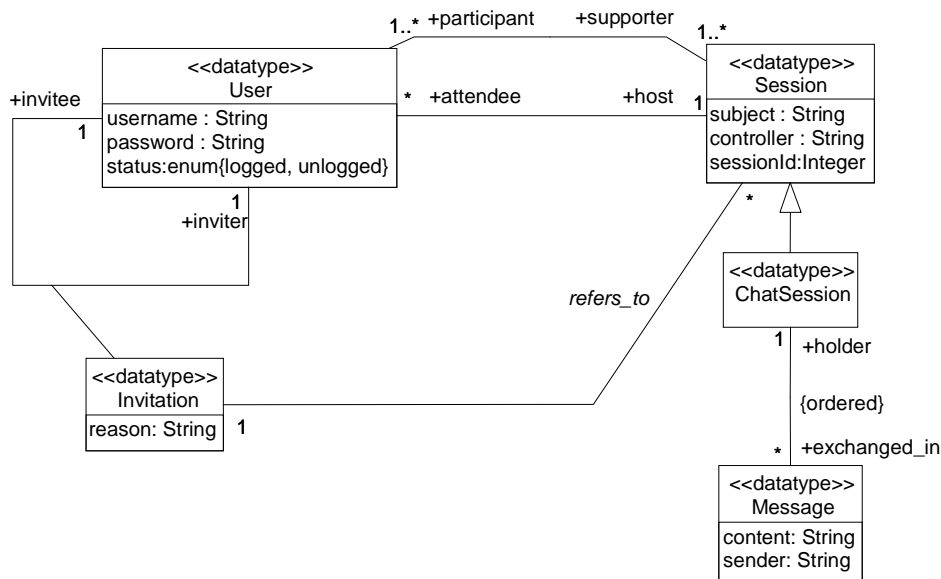


Figure 8-11: Informational model at the system level

The interactional view was captured by a number of interaction sequence diagrams. In order to develop these diagrams, we identified a number of usage scenarios. These scenarios are not extensive, i.e., they do not cover all the alternative ways in which the system can be used. However, they do cover the normal or expected use of the service provided by the system.

Six different scenarios were identified varying in complexity. These scenarios complement each other, in the sense that together they can form a bigger scenario. For example, our second scenario starts with a logged user, which then creates a session, joins the session, invites new participants and sends a message. In our third scenario, one of the invited users logs on and receives an invitation to attend the session. This user accepts the invitation and joins the session.

Figure 8-12 shows the sequence interaction diagram developed for our fourth scenario. According to this scenario, the session controller (User A) checks for polls and opens a new poll. Subsequently, the session controller excludes a participant (User B) from the session. In this diagram, the relationship between the information exchanged in notification interactions is described using a note attached to one of the interactions. The relationship between interaction of an invocation interaction pattern is described elsewhere.

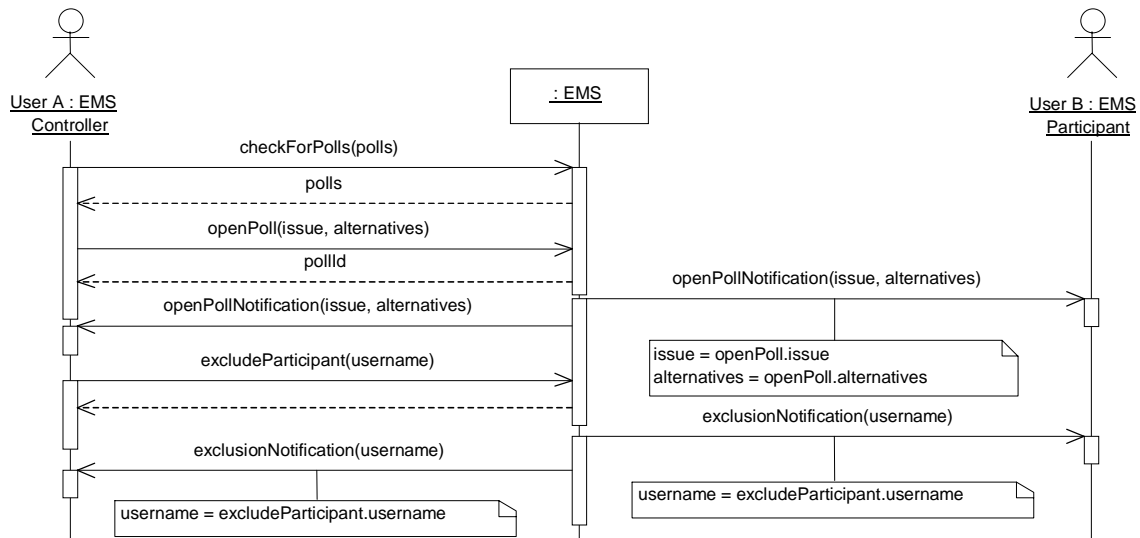


Figure 8-12: Sequence interaction diagram at the system required service specification

The interactions identified in the interaction diagrams were mapped onto operations according to the interaction patterns. Each pair of request/response interactions was mapped onto a corresponding operation and grouped into a functionality interface, EMS Interface, which is realized by the system. Each notification interaction was mapped onto a corresponding operation and grouped into a notification interface, User Interface, which is realized by the system users. We could have grouped the operations into separate functionality and notification interfaces, but such a division is irrelevant for the purpose of our case study.

Figure 8-13 shows the interface diagram developed for the functional entity EMS. This diagram shows that EMS realizes the EMS Interface, while invoking the operations on the User Interface.

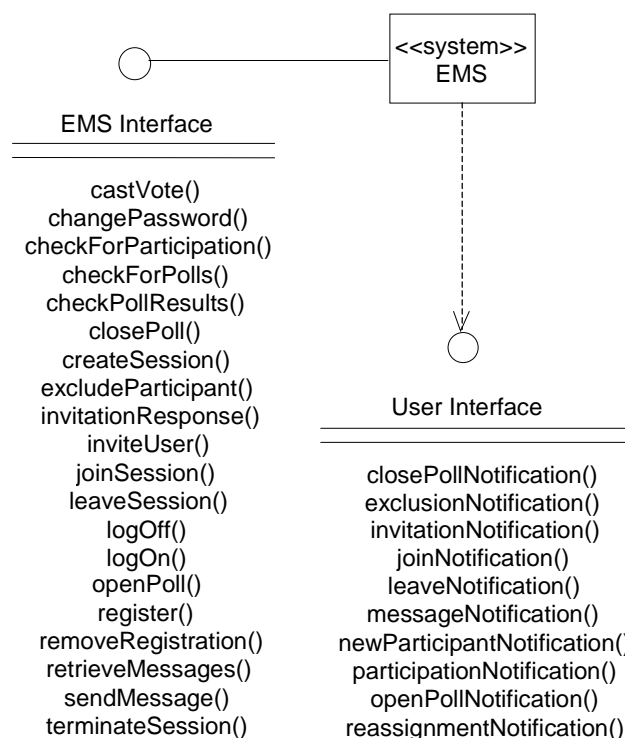


Figure 8-13: Interface diagram for the functional entity EMS

The structural view description was completed with the addition of the entries in the glossary for the use cases, actors, functional entities and interfaces identified in the system required service specification.

The behavioural view was initially captured in the specification of the operations defined in the EMS Interface. For each operation, we specified, a.o., its preconditions and its postconditions, both informally using text and formally using OCL. In this way, we have been able to relate the information exchanged during a request and a response interactions.

We used a notation similar to CORBA IDL to describe the information values passed in a request interaction and the information values passed in a response interaction. In the operation signature, the keyword in indicates that the information value is passed during a request interaction, while the keyword out indicates that the information value is passed during the response interaction. Although not used in this case study, an inout keyword could be used to indicate that the information value is passed during both the request and response interactions.

Figure 8-14 shows the specification of the operation sendMessage.

<b>Operation:</b>	void sendMessage (in content: String)
<b>Interface:</b>	EMSInterface
<b>Purpose:</b>	To send a message in the context of a collaboration.
<b>Cross references:</b>	Exchange Message
<b>Preconditions:</b>	-- none
<b>Postconditions:</b>	-- message is added to the set of messages exchanged in the session let cs: ChatSession = ChatSession.sessionId = joinSession.sessionId cs.exchanged_in->includes(m: Message   m.content = content and m.sender = logOn.username)
<b>Behaviour description:</b>	send message
<b>Exceptions:</b>	--

Figure 8-14: Interface operation specification at the system required service specification

In order to capture the order in which the interactions supported by the system should take place we developed a statechart diagram. The states of this diagram were identified mainly based on the different specialisations of the functional role User as identified at the reference enterprise specification. We identified the following states: Unregistered, Registered, Unlogged, Logged, Inactive and Active.

For each request interaction supported by the system, a separate transition connecting these states was added to the diagram. In this case study, there was no spontaneous transition, i.e., all identified transitions were a result of the occurrence of a corresponding interaction.

Figure 8-15 shows part of the statechart diagram developed at the system level. This diagram shows mainly the ordering of the interactions related to the exchange of messages and voting collaboration features, mainly. During the development of the statechart diagram we avoided specifying guarding conditions on the diagram itself so that its readability is not jeopardised. Preconditions for the execution of an interaction are described in the corresponding interface operation specification.

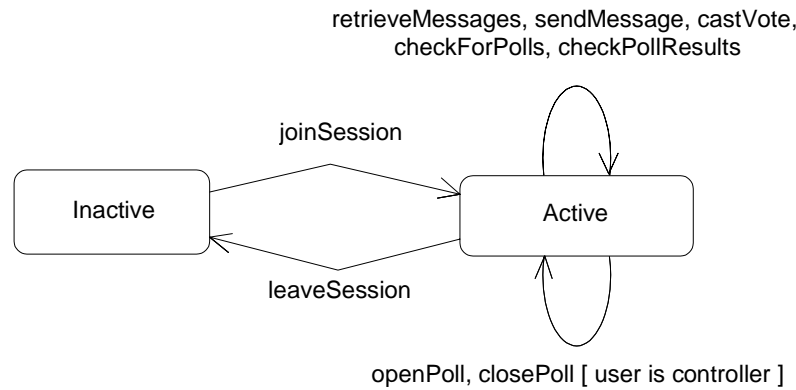


Figure 8-15: Statechart diagram at the system required service specification

For each transition described in the statechart diagram, a corresponding activity diagram was developed to capture the order in which interactions are executed by the system in response to the event that triggered the transition. Since in this case study all state transitions were added as a result of the occurrence of a request interaction, the corresponding activity diagrams were used to relate the request interaction with the response interactions and other interactions that may be executed in between.

Figure 8-16 depicts the activity diagram developed for the transition triggered by the occurrence of the interaction `openPoll`. According to this diagram, the occurrence of this interaction may be followed by the occurrence of three different response interactions, in which two of them refer to the occurrence of an exception. In case the normal response interaction should take place, its occurrence should be followed by the concurrent occurrence of notification interaction `OpenPollNotification`. A separate notification interaction should be received by each active participant in the corresponding session.

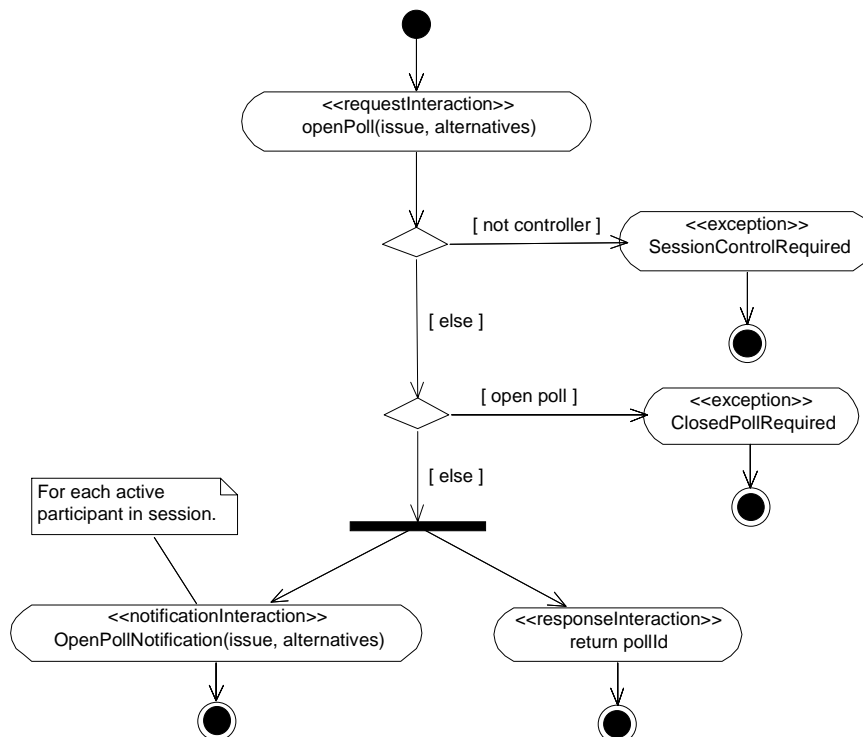


Figure 8-16: Activity diagram at the system required service specification

### 8.3.2 Internal behaviour specification

The development of the system level specification according to the internal integrated perspective was carried out based on the system required service specification. Initially, the description of the use cases identified in the required service specification was complemented with the description of some of the actions that are executed by the system in response to an interaction. Subsequently, we applied a factoring technique to extract some common behaviour from some use cases and we created new use cases (see Chapter 5). These new use cases were properly described and documented in the glossary.

After refining the use case diagrams and use case descriptions, we refined each activity diagram defined in the system required service specification inserting a number of relevant actions that are executed by the system as a result of the occurrence of an interaction and its associated state transition. Some of these actions were already described in the use case diagrams. However, some actions, such as actions describing conditions, are not part of a typical use case description for convenience purposes, but have to be included for completeness.

Figure 8-17 shows the refinement of the activity diagram shown in Figure 8-16. According to this diagram, the relationships between interactions are refined through the insertion of a number of actions. Nevertheless, the ordering in which interactions should be executed is preserved.

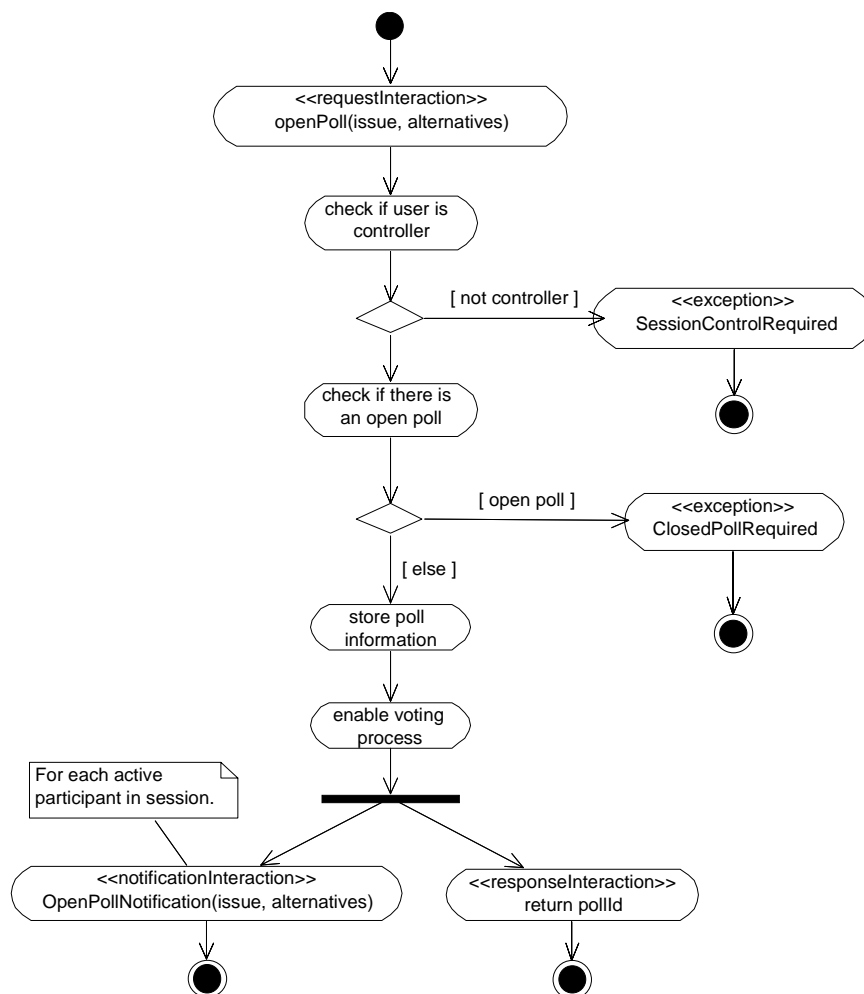


Figure 8-17: Activity diagram at the internal behaviour specification

## 8.4 Component level specification

The component level specification was developed in two steps. In the first step, we refined the system level specification and identified a number of groupware components. Each concrete component was then specified according to the decomposed required service perspective and internal integrated perspective. In the second step, each groupware component identified was refined into a number of simple components. Once again, each concrete simple component was then specified according to the decomposed required service perspective and internal integrated perspective.

### 8.4.1 System decomposition

Our methodology prescribes that we start system decomposition by elaborating a functionality decomposition table. In the decomposition table we listed all use cases identified in the use case diagram developed at the internal integrated specification. We also identified five potential candidate components, corresponding to the five functional roles that could be played by the EMS at the enterprise level.

The assignment of use cases to components followed a basic guideline. We tried to avoid unnecessary dependencies between components, by selecting related pieces of functionality and assigning them to the same component.

Figure 8-18 shows the functionality decomposition table elaborated during the refinement of the system into a number of groupware components.

<i>Component</i> <i>Use case</i>	<b>Access Management</b>	<b>Chat Management</b>	<b>Invitation Management</b>	<b>Session Management</b>	<b>Voting Management</b>
Access System	X				
Control Participant Presence				X	
Control Session Navigation				X	
Control Session Participation				X	
Control Voting					X
Create/Terminate Session				X	
Exchange Message		X			
Invite User			X		
Manage Chat Contents		X			
Manage Pending Invitation			X		
Manage Polls					X
Manage User Information	X				
Provide Awareness Information				X	
Respond to Invitation			X		

Figure 8-18: Functionality decomposition table

The following components were identified:

- access management, which is responsible for the registration of users and their logging on and off the system;
- session management, which is responsible for the creation and termination of collaboration sessions as well as for the control over the session participants and their navigation into and out of sessions;
- invitation management, which is responsible for the invitation of new users to participate in a session;
- chat management, which is responsible for the control over the exchange of messages within a session, and;
- voting management, which is responsible for the control over the opening and closing of polls, as well as the cast of votes.

Based on these components, we developed a package diagram to capture the static relationship and dependencies between them. For each identified groupware component, a separate package was added to the diagram and some tentative dependencies relationships established. Nevertheless, we could only complement these dependencies later, when the interactional view of the identified components was described.

Figure 8-19 illustrates the package diagram resulting from the decomposition of the EMS into a number of groupware components. Each package represents a functional entity at the component level. The diagram omits from the system level entity part of the components' environment, i.e., the system users.

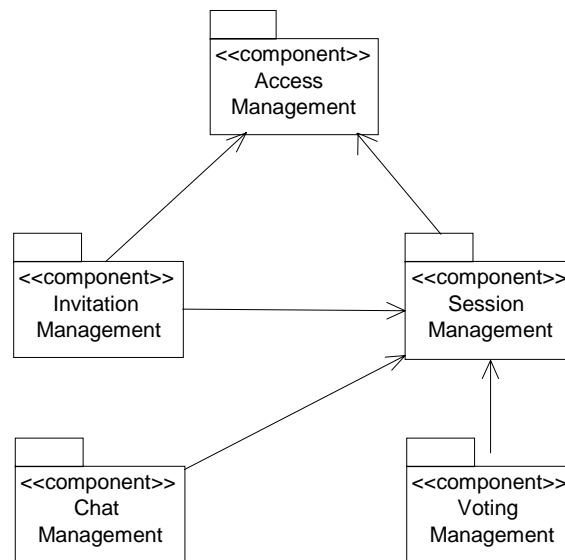


Figure 8-19: Package diagram resulting from the refinement of the EMS

In the sequel we limit our discussion to the session management component. This component was chosen amongst the other groupware components, for its prominent role in the EMS, and because it is slightly more complex than the other components.

#### 8.4.2 Session management component specification

The development of the session management component specification was carried out in two steps. In the first step, this component was specified according to the decomposed required

service perspective. In the second step, the decomposed required service specification was refined into the internal integrated behaviour specification. Although we could also have specified the component according to the required service perspective in the first place, we have chosen not to specify this component according to this perspective because its behaviour is not complex enough to justify the effort in developing this specification.

### Decomposed required service specification

The use case diagram for the component Session Management was developed according to the assignment of use cases to groupware components, and the static relationships between the components defined in the package diagram.

In the development of the use case diagram, there was no need for refining a use case into separate use cases, because the use cases were already simple enough. Further, we have maintained the same associations between actors and use cases, and between use cases themselves, as described at the system level specification. Whenever there was an association between use cases assigned to different components at the system level specification, this association was replaced by another association between a use case in the session management use case diagram, and an actor representing the corresponding component that makes use of the functionality of a use case (or vice versa). Afterwards, the description of each use case was updated to reflect the changes in the diagram.

Figure 8-20 shows the resulting use case diagram for the component Session Management. In this figure, we omit the relationships between actors for the sake of conciseness.

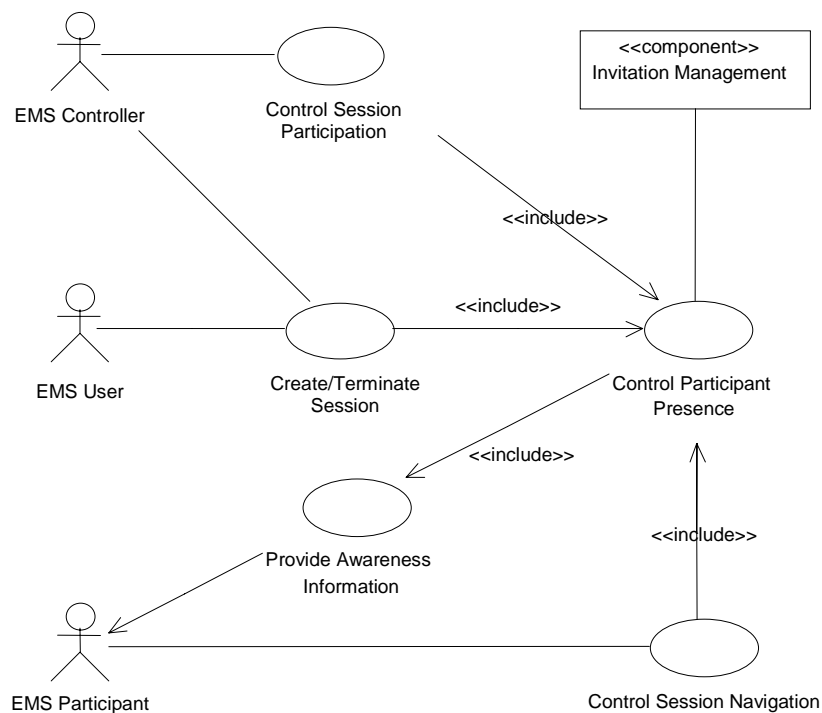


Figure 8-20: Session management component use case diagram

After the use case diagram was obtained, we developed the component interactional view description. We started by defining a suitable informational model based on the informational model development for the EMS as a whole.

Each sequence diagram developed at the system level according to the scenarios defined at that level was refined into a corresponding sequence diagram at the component level. These diagrams contain only the groupware components related to the scenario being modelled.

Figure 8-21 shows the sequence diagram at the component level that corresponds to the diagram depicted in Figure 8-12. In this diagram, we omit the relationship between the information values exchanged during the interactions for the sake of conciseness.

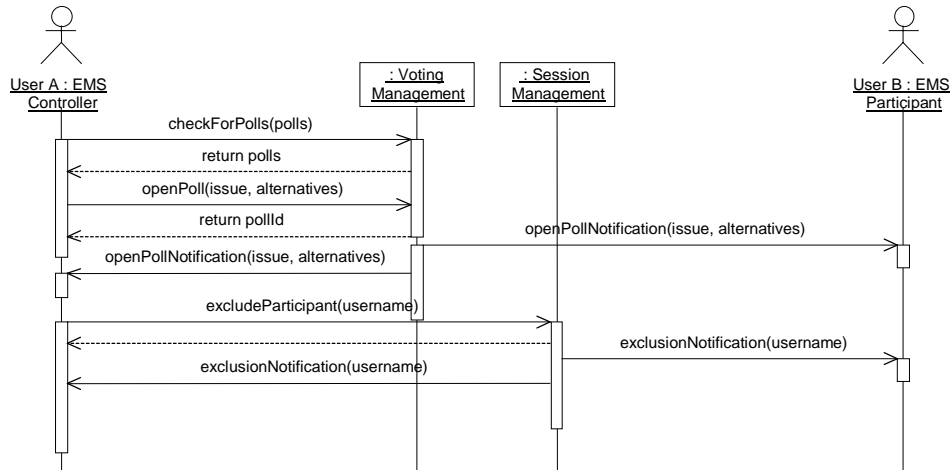


Figure 8-21: Sequence interaction diagram developed for Session Management

The component Session Management interaction diagrams were developed in parallel with the component interface diagram. For each instance of interaction pattern identified, a corresponding operation was specified and added to the diagram.

In contrast with the EMS, for which we defined a single interface, in this case we have grouped the operations related to the component Session Management into separate interfaces according to their main purpose. For example, the operations `createSession` and `terminateSession`, used to respectively create and terminate a cooperative session, were grouped into the interface Session Management Interface, while the operations `joinSession` and `leaveSession`, used to respectively join and leave a cooperative session, were grouped into the interface Participation Management Interface.

Since other components depend on the functionality associated with these four operations in particular, corresponding notification operations have been added to the diagram in two separate notification interfaces, viz., Session Management Notification Interface and Participation Notification Interface.

Figure 8-22 shows the resulting interface diagram developed for the component Session Management. This diagram omits the interface supported by the EMS environment as a whole for simplification purposes. Interfaces supported by the component are connected to the component via a straight line, while the interfaces used to notify the occurrence of the events are connected to the component via an arrow.

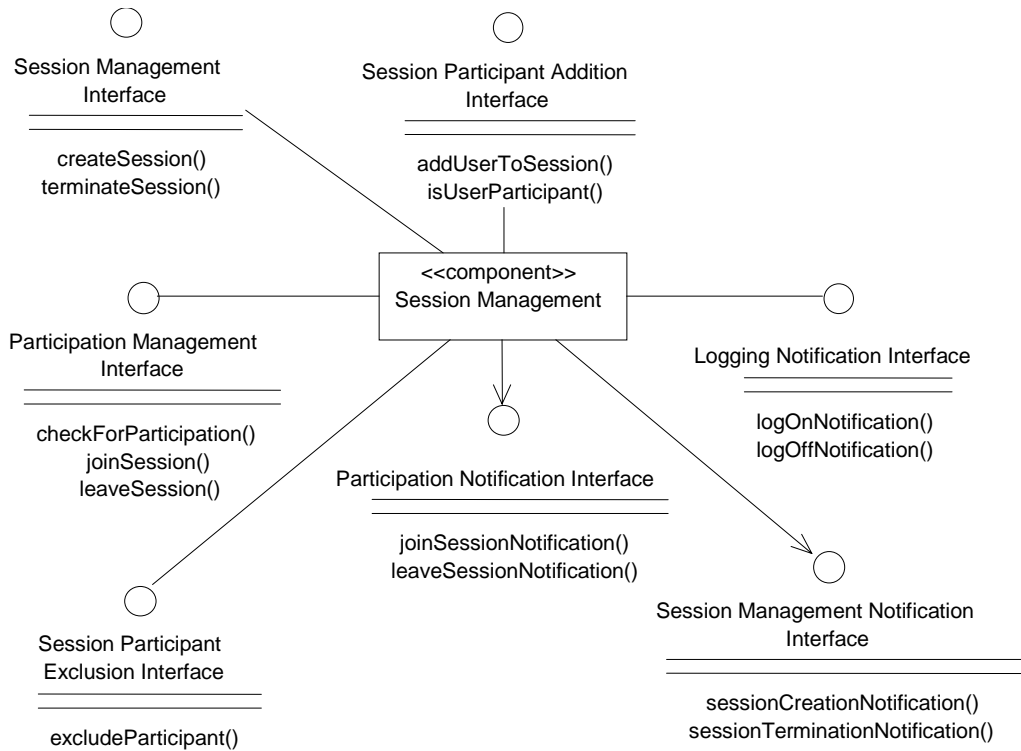


Figure 8-22: Interface diagram for the component Session Management

The behavioural view of the session management component was initially described in a statechart diagram. This statechart diagram corresponds to a decomposition of the statechart diagram of the whole system. It captures the order in which the interactions between the component and a single user should take place. Therefore, the complete state of the component can be obtained superposing the diagrams for all users.

Figure 8-23 depicts the statechart diagram developed for the component Session Management. For each interaction supported by the component (see Figure 8-22), a corresponding event is shown in the diagram. This diagram, however, omits the order in which the interactions `addUserToSession` and `isUserParticipant` could be initiated, since they can be initiated with the component being at any state and their execution result in a transition that do not modify the component state.

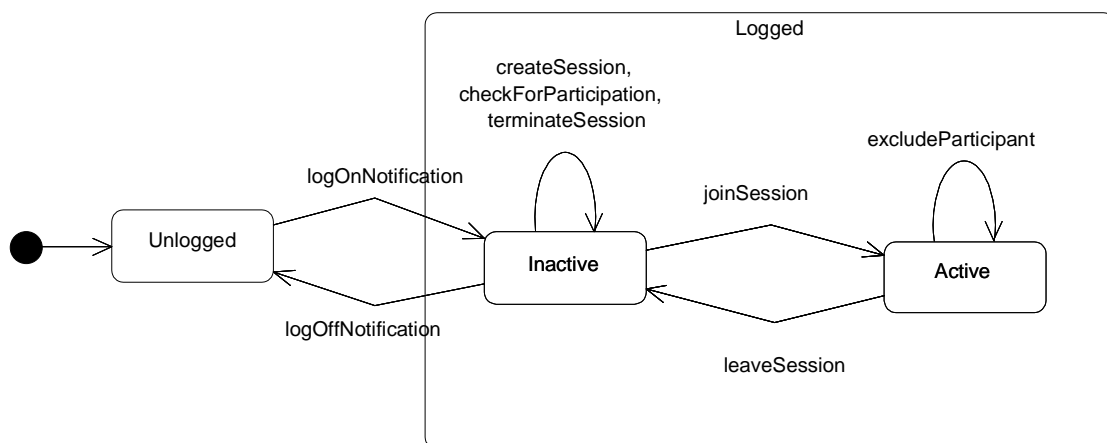


Figure 8-23: Statechart diagram for component Session Management

The behavioural view description of the component Session Management was complemented with an activity diagram for each state transition identified in the component statechart diagram. These activity diagrams capture the order in which other interactions are initiated as a consequence of the execution of the corresponding interaction.

Figure 8-24 shows the activity diagram for the transition triggered by the occurrence of the interaction `joinSession`. According to this diagram, the occurrence of this interaction may be followed by the occurrence of three different response interactions, two of them being exceptions. In case no exception occurs, the normal response interaction should take place concurrently to the occurrence of the interaction `joinNotification`.

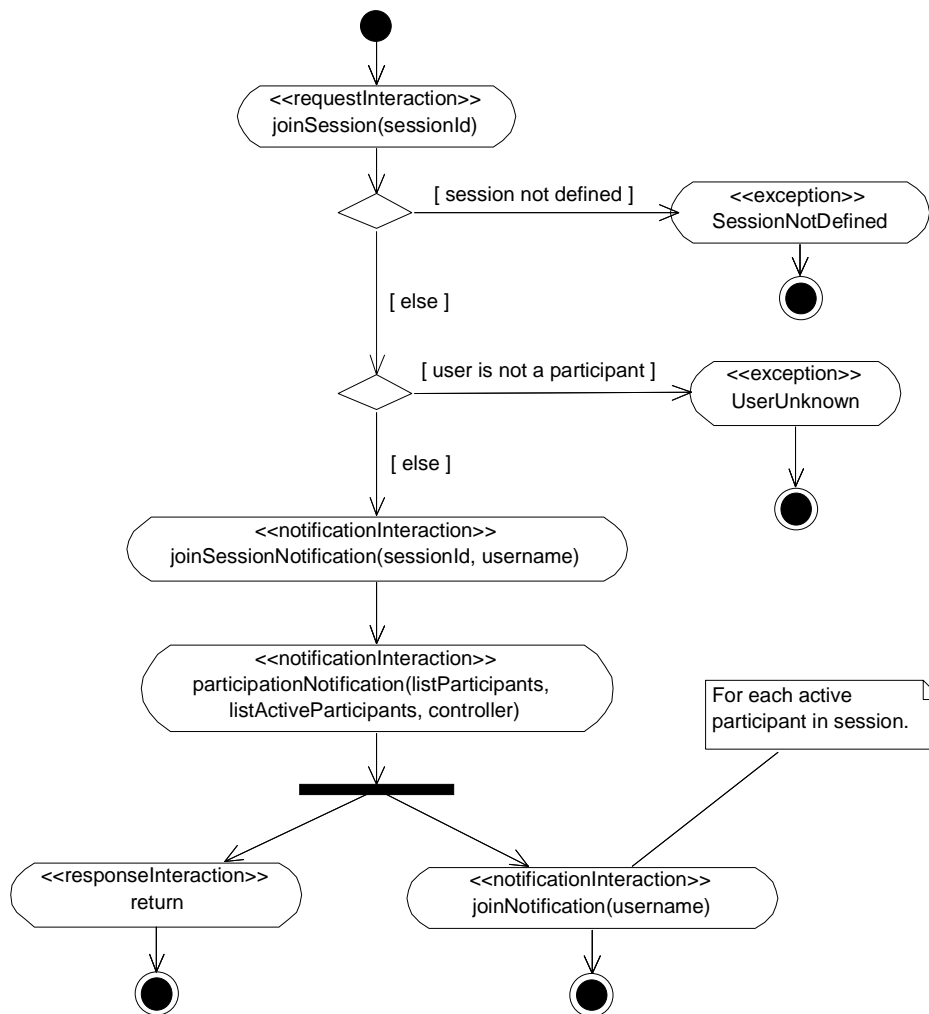


Figure 8-24: Activity diagram for interaction `joinSession`

### Internal behaviour specification

The component Session Management internal behaviour specification was obtained by refining each activity diagram developed at the decomposed required service specification. We refined the relationships between the interaction contributions identified in the diagrams, by inserting a number of actions that model the internal activities of the component.

### 8.4.3 Session management component decomposition

We refined the component Session Management into simple components based on three different collaboration concern layers, viz., user, collaboration and resource. In this case study we did not consider the interface layer since we were mainly interested in the high-level architectural design of the EMS, and not so much in the user interface issues.

In general, the use cases assigned to the component Session Management were simple. The functionality represented by each use case typically contained concerns related to at least two of the concern layers, which would require us to decompose each use case into two or three different use cases in order to properly assign them to the candidate components. Since the requirements associated with each use case were well understood, we have decided not to decompose them further. Consequently, we have not developed a new decomposition table at this stage.

A separate simple component was identified to cope with the concerns and perform the behaviour associated with each one of the collaboration concern layers considered. The following simple components were identified:

- Session Management\_User, which is responsible for implementing the user concern layer of the component Session Management;
- Session Management\_Collaboration, which is responsible for implementing the collaboration concern layer of the component Session Management, and;
- Session Management\_Resource, which is responsible for implementing the resource concern layer of the component Session Management.

Figure 8-25 shows the package diagram resulting from the decomposition of the component Session Management into a number of simple components. This diagram omits other functional entities that are part of the environment of the components.

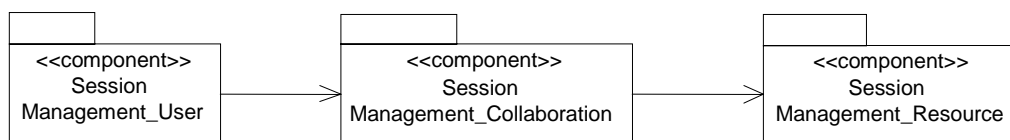


Figure 8-25: Package diagram resulting from the refinement of Session Management

The identified components are coupled at the collaboration coupling level, using a fully centralised architecture for the coupled layers. Since the user layer is uncoupled, a separate instance of the component Session Management\_User should be created to support each separate user of the component Session Management.

Similarly to the component Session Management, the simple components were specified according to exclusively the decomposed required service perspective and internal integrated perspective. In the sequel, we briefly discuss these specifications.

#### Decomposed required service specification

Since we have not assigned use cases to the components Session Management\_User, Session Management\_Collaboration and Session Management\_Resource, we have not developed use case diagrams for these components. In order to complete the structural view description of each component we simply developed a component interface diagram for each one of them. Fur-

ther, we developed another interface diagram containing all components, their supported interfaces and interface dependencies, in parallel with a number of interaction diagrams to capture the interactional view of these components.

Figure 8-26 shows the interface diagram developed to the component Session Management\_Collaboration. The interfaces Session Management\_C Interface and Session Participant Addition Interface are realised by this component, while the interfaces Session Control Interface and User Control Interface are realised by the component Session Management\_Resource (omitted in the diagram). The interface Participant Notification Interface is realised by the component Session Management\_User (not shown in the diagram). Finally, the interfaces Session Management Notification Interface and Participation Notification Interface are realised by other groupware components, also omitted in the diagram.

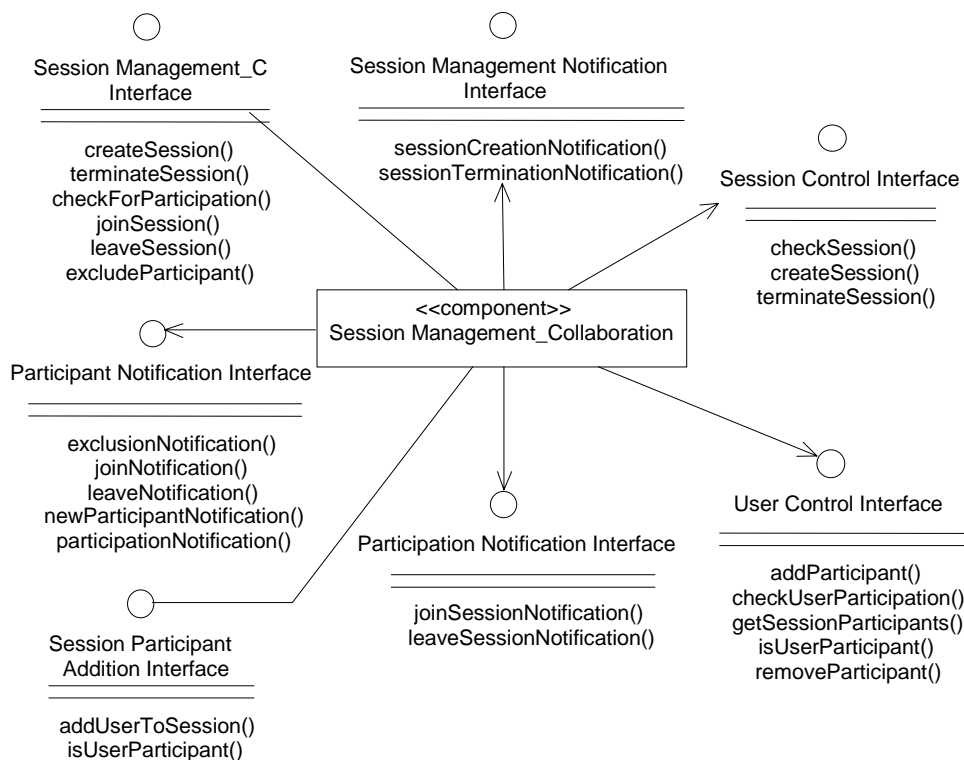


Figure 8-26: Interface diagram for the component Session Management\_Collaboration

The interactional view specifications have been developed based on the scenarios in which the abstract component, i.e., the component Session Management, was being used. For each one of the scenarios previously described, new sequence diagrams were developed, in which the abstract component was replaced by the identified simple components and new interactions between these components were identified and modelled accordingly. These interactions were grouped into operations, which were grouped themselves into interfaces (see Figure 8-26). Each of the identified operations was then properly specified (behavioural view description), including the specification of precondition and postconditions, both informally using text and formally using OCL.

In these diagrams, we used the same informational model developed for the component Session Management. This informational model was suitable because it was already at the required abstraction level.

Figure 8-27 shows a sequence diagram describing a scenario in which a user joins a session. This sequence diagram refines the interactional view description shown in Figure 8-24.

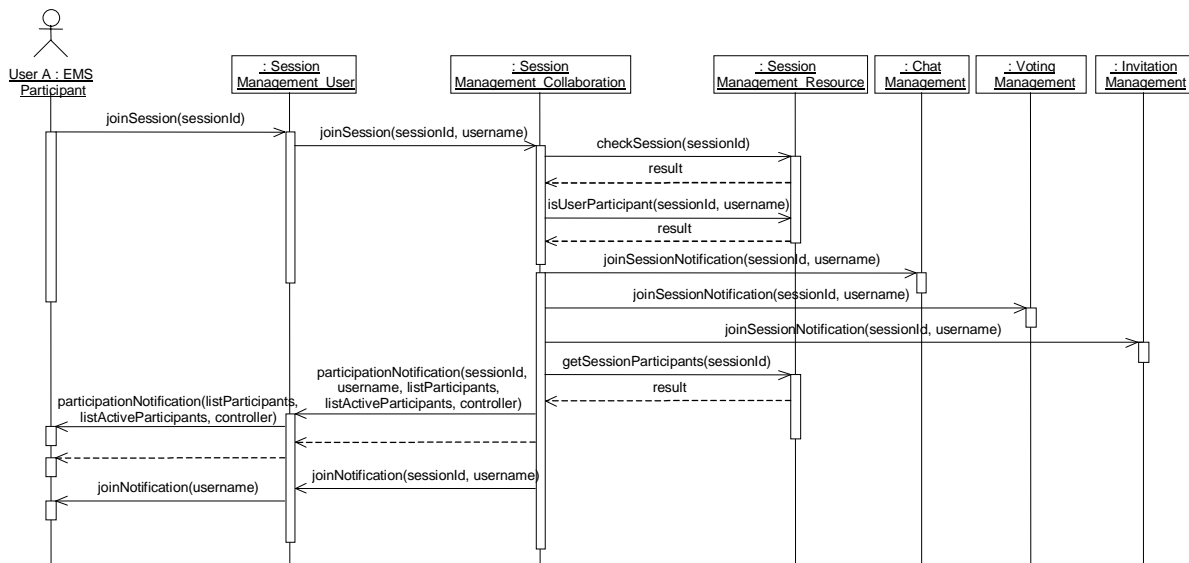


Figure 8-27: Sequence interaction diagram developed for simple components

The behavioural view description of the simple components was complemented with a statechart diagram for each component. The statechart diagram developed for the component Session Management\_User corresponds to the diagram shown in Figure 8-23. For each state transition, a separate activity diagram was developed to capture the relationship between the interaction contribution that triggers the transition and other interaction contributions that are executed during the transition itself.

We developed no statechart diagram for the components Session Management\_Collaboration and Session Management\_Resource because these components were stateless, i.e., the execution of interactions is not dependent of a particular state. Nevertheless, we developed a separate activity diagram for each interaction supported by these components in order to capture the relationship between the corresponding interaction contribution and the interaction contributions of other interactions executed as a result of the occurrence of the supported interaction.

### Internal behaviour specification

Each activity diagram of the decomposed required service specification was refined for the internal behaviour specification, by modelling the actions executed in between the occurrences of interactions.

Figure 8-28 shows two activity diagrams that capture the internal integrated behaviour associated with the occurrence of the interaction leaveSession. Figure 8-28a shows the diagram of the component Session Management\_User, while Figure 8-28b shows the diagram of the component Session Management\_Collaboration. The diagram in Figure 8-28b corresponds to the behaviour that follows the invocation of operation leaveSession(sessionId, username) in the diagram of Figure 8-28a.

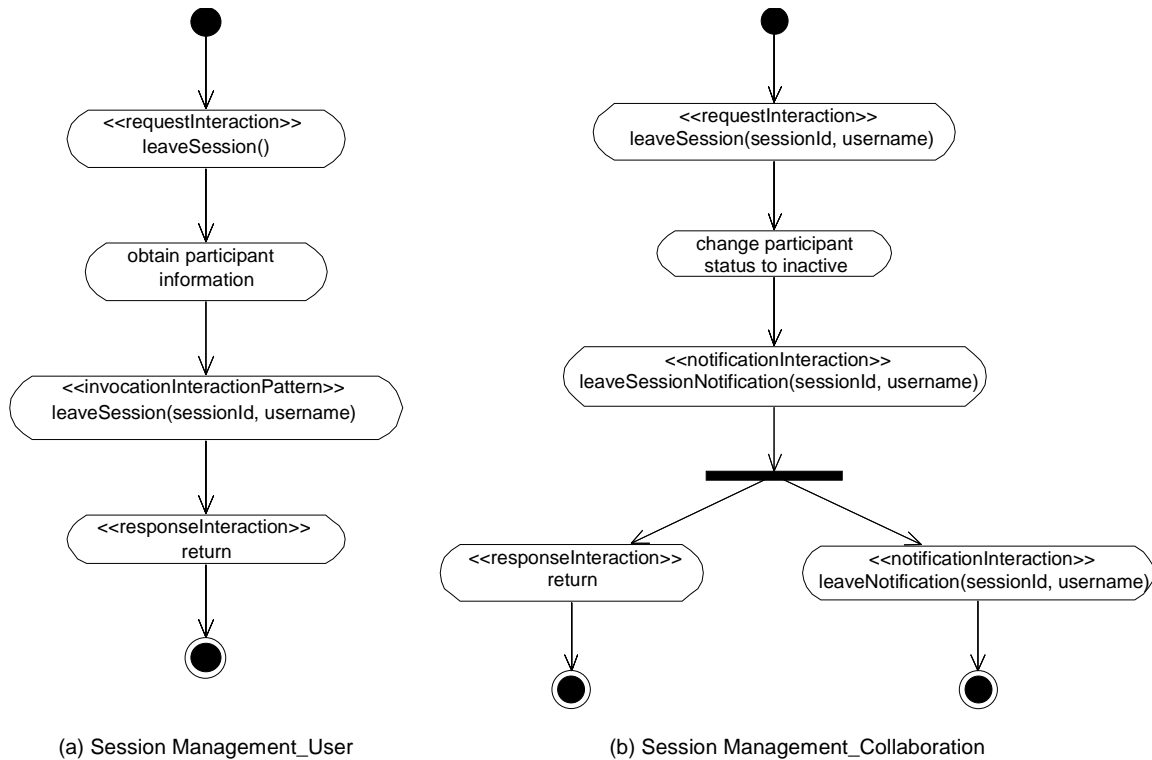


Figure 8-28: Activity diagram developed for components *Session Management\_User* and *Session Management\_Collaboration*

## 8.5 Conclusion

This chapter reports on a case study in which our methodology has been applied in the architectural design of an electronic meeting system. The chapter initially describes the collaboration context used as starting point for the case study. Then, the chapter describes some of the main aspects taken into account and the design decisions at the enterprise, system and component levels. Excerpts of the diagrams developed in the case study are presented and discussed to illustrate some of these aspects and design decisions.

This case study demonstrates the conceptual simplicity and systematicness of our approach for stepwise design. We can also observe throughout the diagrams presented the use of the elements of our profile. However, due to the limitations of the case study itself, as well as the time constraints of this work, some features of our methodology could not be explored in the case study.

We used Rational Rose Enterprise Edition™, from Rational Software Corporation in order to draw the UML diagrams. The complete EMS specification, including UML diagrams and other documents, will be available at the Architecture Group web site (<http://arch.cs.utwente.nl>).

---

# Chapter 9

## Conclusion

This chapter presents a summary of the main conclusions and contributions of this work and outlines a number of issues for further research.

The chapter is structured as follows: section 9.1 presents some general considerations of our work; section 9.2 describes the main contributions of the work; and section 9.3 outlines some directions for further research.

### 9.1 General considerations

The design of component-based systems has lately drawn the attention of the software engineering and distributed systems communities. As system requirements become more complex and distribution aspects become an intrinsic part of these requirements, software engineers and architects have adopted component-based technologies as one of the most adequate means to cope with these issues. Additionally, component-based technologies facilitate the reuse and deployment of these systems.

This work proposes a methodology for the architectural design of component-based groupware systems that tackles the specification of a cooperative work process across multiple abstraction levels. This methodology [FaSF00, FaFS00b] comprises a number of design concepts and design methods associated with a standard and widely accepted design notation, namely UML. As a result, this work also investigates the suitability of UML for the architectural design of component-based systems, identifying shortcomings and proposing improvements as needed.

We have striven for producing a methodology whose main characteristics are conceptual clarity and systematicness. We define a basic set of concepts and specialisations thereof that can be used throughout the design process. We also defined a number of milestones that facilitate the stepwise development of a number of related specifications. We also prescribe which design activities should be carried out within each design step and provide guidelines to develop appropriate models accordingly.

We believe that our methodology is general and flexible enough to be used in the architectural design of not only groupware systems but also other telematics systems in general, although we do not have evidence of that. Similarly, due to the limitations of time and manpower, we have not experimented with the design of large systems, which prevent us from making any considerations regarding the scalability of the methodology.

## 9.2 Main contributions

### 9.2.1 Concern levels, perspectives and views revisited

Our methodology suggests the architectural design of groupware systems according to three different concern levels:

- *enterprise*, which is concerned with a high level representation of a cooperative work process and its relationship with an associated enterprise;
- *system*, which is concerned with the provision of an automated support for (a part of) a cooperative work process;
- *component*, which is concerned with the decomposition of the automated support provided by a groupware system for a cooperative work process into a number of interrelated components.

At each concern level, different specifications of the cooperative work process or groupware system can be developed according to a stepwise approach based on different interrelated perspectives. These perspectives are defined along the design trajectory of each concern level.

At the enterprise level three different perspectives are identified:

- *integrated*, which is concerned with the description of the cooperative work process in itself. According to the integrated perspective there is no identification of individual contributions to the cooperative work process, except for the contribution of the enterprise as a whole;
- *distributed*, which is concerned with the distribution of responsibilities for the execution of the cooperative work process among a set of functional roles and entities defined in the enterprise;
- *distributed with role discrimination*, which is concerned with the description of primary responsibilities for the execution of common units of cooperative behaviour between two functional entities.

At the system and component levels three different perspectives are also identified:

- *required service*, which is concerned with the description of the service provided by an entity to its service users, such that these users can use these services to execute (part of) the cooperative work process;
- *decomposed required service*, which is concerned with the description of the service provided by an entity to its service users and its relationship to the service required by this entity from other entities, called auxiliary entities, to provide its own required service;
- *integrated internal*, which is concerned with the description of the internal behaviour of an entity in terms of the activities executed by this entity in an integrated way and its relationship to the decomposed required service of this entity.

At each concern level and each perspective, three different views can be usually applied to guide the specification of the different sets of concerns:

- *structural*, which is concerned with the static structuring of behaviour;

- *behavioural*, which is concerned with the behaviour of a single entity;
- *interactional*, which is concerned with the cooperative behaviour of two or more entities.

The use of concern levels, perspectives and views allows for a better control over the design process of a groupware system, such that the system designer can better focus on the relevant set of concerns at a time along the design trajectory. Further, concern levels, perspectives and views also help maintain the integrity and conformance of the design itself with respect to the successive transformations that an abstract design must undergo until a design suitable for implementation using current technologies is obtained.

General design concepts are introduced at the enterprise level and further specialised to fit the characteristics of the system and component levels. Based on these concepts, specification techniques and guidelines are presented. These specification techniques are targeted to the characteristics of the set of concerns being modelled. Guidelines to help in the refinement process and refinement assessment afterwards are also presented based on the design concepts and specification techniques.

Figure 9-1 depicts the concern levels, perspectives and views introduced in this work.

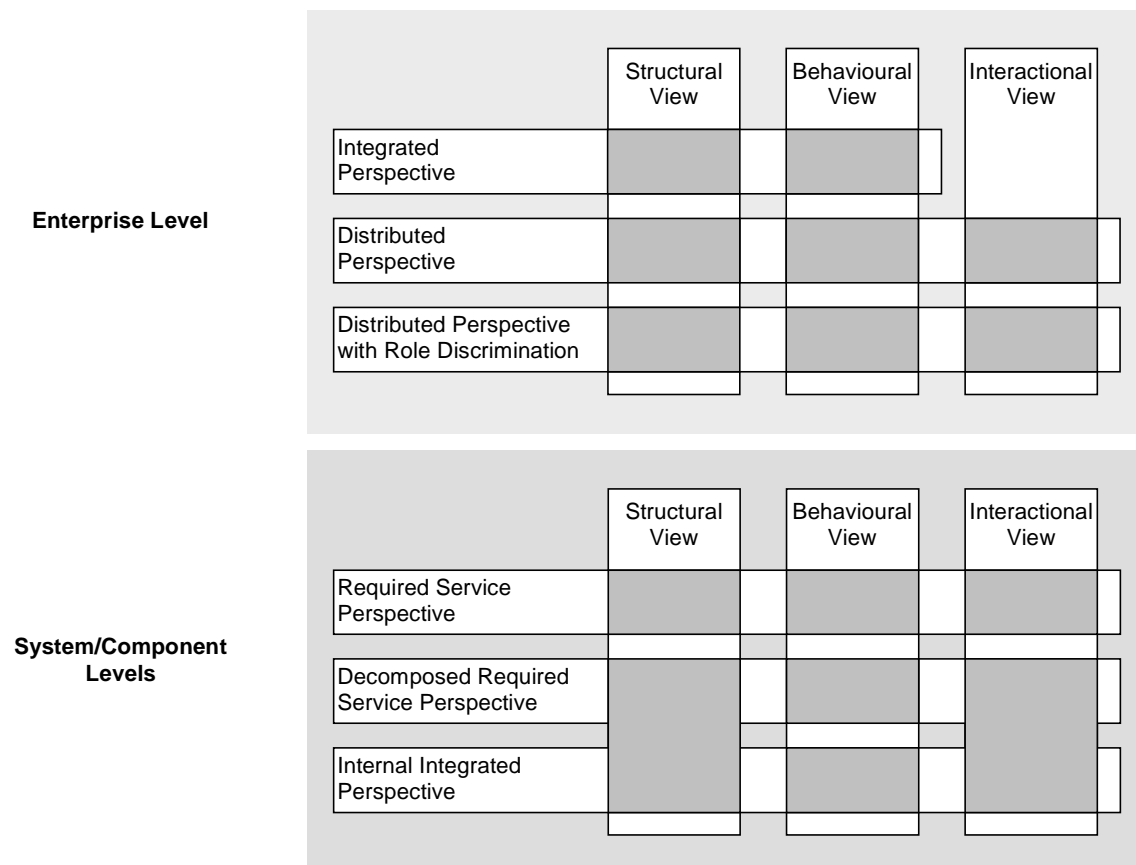


Figure 9-1: Concern levels, perspectives and views

### 9.2.2 Component types and collaboration concern layers revisited

The decomposition of a groupware system into a number of interrelated components is carried out recursively based on different hierarchically related component types. At each decomposition step, a more specific set of components is identified and specified according to the per-

spectives and views defined at the component level. The following component types are identified:

- *application component*, which represents a groupware system on its own, capable of supporting a cooperative work process or context. An application component can be used in isolation or combined with other application components to form, for example, a groupware environment. In this sense, the groupware system under development is considered an application component;
- *groupware component*, which represents a self-contained, mostly independent, piece of groupware functionality. A groupware component is typically not used in isolation but in combination with other groupware components to form an application component;
- *simple component*, which represents a component that implements a specific set of concerns defined within the context of a groupware component. Simple components form the most elementary building blocks at the component level, so they are not further refined at this level.

The use of specific component types facilitates the decomposition process if compared to a decomposition approach that does not make such distinction. Given a certain level during the decomposition process, it is easier to place it along the design trajectory, such that someone, till a certain extent, knows how many decomposition steps can be expected until the architectural design of the groupware system can be completed. In this sense, specific decomposition guidelines are provided to aid in the process based on the different component types.

The use of a component taxonomy scheme also facilitates the discovery, and consequently the reuse, of existing components. Components with characteristics similar to those described here, are usually available in the component-based groupware platforms investigated in this work, especially in the case of groupware components.

The collaboration concerns targeted by a groupware component can be structured according to four consecutive layers:

- *interface*, which is concerned with the interactions between a human user and the groupware component;
- *user*, which is concerned with the collaboration concerns of a single user;
- *collaboration*, which is concerned with the collaboration concerns of multiple users;
- *resource*, which is concerned with the persistent storage of information.

The use of specific sets of concerns help mainly in the identification of simple components and in the assignment of abstract behaviour to these components.

Whenever two or more users of a groupware component collaborate, coupling levels can be defined between the states of the layers defined by this component. In this sense, at most four different coupling levels can be defined, one for each layer. A coupled layer indicates that a single state, shared by all the users of the layer, is maintained for the layer. An uncoupled layer indicates that different states are maintained for the layer. Coupling can be accomplished using centralised, replicated or hybrid architectures.

Collaboration coupling has been investigated prior to this work. However, due to its growing importance in providing flexible collaboration modes and in placing requirements on how behaviour should be structured to achieve such flexibility, this issue is placed within the context

of this work accordingly. Nevertheless, the provision of a detailed account of such issue was outside the scope of this work.

### 9.2.3 UML shortcomings revisited

The architectural design of component-based systems using UML as design notation shows a number of shortcomings associated with this language. The main problems identified are related to:

- *limited structuring mechanisms*, which are most noticeable in the following cases:
  - *structuring of behaviour in activity diagrams*, which is supported only by the use of swimlanes and activity states. The lack of more adequate structuring constructs poses a limitation on the specification of complex behaviours. Structuring in activity diagrams would be better accomplished if a solution similar to behaviour blocks [EJL+99] were available, for example. In principle, the use of behaviour blocks is similar to the use of activity states. However, the former allows for a more abstract specification, without the refinement implications associated with the latter;
  - *structuring of behaviour in interaction diagrams*, which does not support the abstraction and decomposition of complex interaction sets explicitly. For example, it would be desirable to decompose an interaction diagram into multiple explicitly related interaction diagrams, allowing the designer to tackle complexity in a more controlled and systematic way;
- *limited mechanisms for relating specifications across different abstraction levels*, which nowadays can only be accomplished in a rudimentary form;
- *lack of more abstract relationship constructs between activities*, such as the disabling of one activity by another and the synchronised execution of multiple activities. The introduction of such additional types of relationship in activity diagrams, for example, would require a basic change in the semantics of the execution of activities in general and actions in particular: instead of enabling the execution of an action, a transition should enable its completion. A more abstract semantics for actions that could be used to enhance the specification of activities and their relationships is proposed in [Quar98];
- *limitation of behaviour specification in statechart and interaction diagrams*, which is restricted to the specification of (alternative) sequences of activities. This limitation is imposed by the low abstraction level underlying UML, given its target domain (specification of objects). Since an object is typically a fine-grained functional unit, executed within a single processing unit at a time, its externally visible activities, i.e., interactions, are usually sequentially executed. However, this assumption is not valid for the high-level specification of distributed systems in general. We should be able to specify the execution of concurrent interactions, in which no specific ordering is assumed, using, for example, coregions as defined in message sequence charts [RuGG96];
- *limited view of components*, where components are seen as units of deployment instead of units of design. Probably this is the most well known limitation of the current version of UML with respect to component-based design. We should be able to use component instances as we use objects in interaction diagrams. We should also be able to assign statechart and activity diagrams to a component to capture the order in which the interactions associated with this component can take place. Further, we should also be able to relate components that belong to different abstraction or aggregation levels, such that we can on

one hand decompose a coarse-grained component into multiple related finer-grained components and on the other hand compose finer-grained components to form a coarse-grained component.

Currently, a major review on UML is underway with the development of UML 2.0 [Kobr99]. One of main requirements of UML 2.0 is the provision of better support for component-based development.

It is unlikely, though, that UML 2.0 will solve all of the shortcomings described in this work. UML was developed for the specification of object-oriented systems and, as such, the concepts and abstractions used in UML facilitate their realisation in object-oriented implementation technologies. The addition of more abstract concepts, although desirable, may create certain constraints on how these concepts are realised using current technologies, which may not be easily overcome. Therefore a proper balance should be found between abstraction concepts and their realisation capabilities.

Since it was outside the scope of this work, we provide neither specific concepts nor guidelines to specify a groupware system according to the object level. Ironically, UML would provide a much better specification support at this level. Therefore, the biggest benefit of providing UML with more abstract concepts and constructs lies on the proper use of a single notation throughout the whole design process of a software system covering both the architectural design and the detailed design phases.

### **9.3 Directions for further research**

The work presented here can be continued in a number of ways. Particularly we identify four major issues that require further investigation: the expansion of the methodology scope, the use of UML in combination with other (abstract) design notations, the provision of support for the design of tailorable component-based groupware systems and the investigation of the relationship between our methodology and OMG's Model Driven Architecture [OMG01b].

The methodology scope as presented in this work comprised the architectural design of groupware systems, i.e., the enterprise, system and component concern levels. Further, we concentrated our investigation on three main views that can be used to facilitate the design process across all these levels. As a natural follow up on this work, one could investigate the design process at the object level and incorporate the use of other relevant views, such as the informational and deployment views, to the design process.

At the same time, one could improve the methodology by applying it on many other case studies and industrial applications. Besides the case studies reported here, our methodology has also been partially applied in the design of a multimedia conferencing system [SaSF01]. Experiences like this should be stimulated as means of disseminating our work and collecting valuable feedback.

The current UML standard has proven ineffective for the architectural design of component-based groupware systems. New improvements are underway that may solve part of the problems identified in this work. Nevertheless, we should not restrict ourselves to the use of UML, despite the obvious benefits associated with the use of a well-known design notation, especially because we cannot rely on the new UML standard to solve all the shortcomings associ-

ated with the current UML standard. In this sense, one could also investigate the use of other (abstract) design notations in combination with UML itself as proposed in [FFS+01].

We believe that the ultimate goal in component-based design is to provide a methodology to support the development of component-based groupware systems with increased degree of tailorability. However, we can split this goal into two separate goals: (1) to provide a methodology to design component-based groupware, which is covered in this thesis, and (2) to provide guidelines to incorporate tailorability into groupware components. The first goal can be seen as a requirement for the second because we cannot talk about component-based groupware tailorability if we cannot address component-based design in the first place.

As a first step towards the design of tailorable component-based groupware systems, we have already outlined an approach based on monitoring techniques for the provision of tailorability based on legacy components [FaDi00]. However, this approach is limited to one particular type of tailorability: extension of existing functionality. A more comprehensible approach for the provision of tailorable component-based groupware systems could be obtained through, for example, the combination of our suggested approach with the approach for the provision of dynamic reconfiguration of CORBA objects proposed in [Alme01]. However, further investigation is still necessary.

Recently, OMG has started an effort known as Model Driven Architecture (MDA). The main goal of the MDA is to facilitate the integration of distributed systems via the development of general models across the across the different phases of the system lifecycle, including business specification, system design and component construction, assembly, deployment, management and evolution [OMG01b]. These models are computational (business specification) and platform (system and component specifications) independents. In this way, platform independent models can be later specialized into platform specific models, such as CORBA and EJB specific specifications.

To a certain extent, the scope and objectives of our methodology are similar to the MDA. Both attempt at capturing general, system independent information before specifying the system itself, although we constrain ourselves to the cooperative work domain. Further, both aim at producing platform independent specifications that can be later specialised accordingly. Finally, both rely on standard specification notations to accomplish their objectives. Still, additional investigation is needed to assess the additional commonalities between these two developments and how they can influence each other.



---

## References

- [ABL+00] Atkinson, C., Bayer, J., Laitenberger, O. and Zettel, J.: Component-Based Software Engineering: The Kobra Approach. In *Proceedings of the 2000 International Workshop on Component-Based Software Engineering*, 2000. Internet: <http://www.sei.cmu.edu/cbs/cbse2000/papers/21/21.html>.
- [AbSa94] Abbott, K.R. and Sarin, S.K.: Experiences with workflow management: issues for the next generation. In *Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work (CSCW'94)*, pp. 113-120, 1994.
- [AcTe98] Action Technologies: *Business Process Integrity in ActionWorks™*. White Paper, 1998. Internet: <http://www.actiontech.com>.
- [AhEL90] Ahuja, S.R., Ensor, J.R. and Lucco, S.E.: A comparison of application sharing mechanisms in real-time desktop conferencing systems. In *Proceedings of the 1990 ACM Conference on Office Information Systems (COIS'90)*, pp. 238-248, 1990.
- [Alme01] Almeida, J.P.A.: *Dynamic Reconfiguration of Object-Middleware-based Distributed Systems*. M.Sc. Thesis, University of Twente, 2001.
- [AtBM00] Atkinson, C., Bayer, J. and Muthig, D.: Component-Based Product Line Development: The Kobra Approach. In *Proceedings of the First Software Product Line Conference*, pp. 289-309, 2000.
- [BaSc89] Bannon, L.J. and Schmidt, K.: CSCW: Four Characters in Search of a Context. *Proceedings of First European Conference on Computer Supported Cooperative Work (ECSCW'89)*, pp. 358-372, 1989.
- [BBR+96] Benford, S., Brown, C., Reynard, G. and Greenhalgh, C.: Shared Spaces: Transportation, Artificiality, and Spatiality. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work (CSCW'96)*, pp. 77-86, 1996.
- [BDL+99] Bergquist, J., Dahlberg, P., Ljungberg, F. and Kristoffersen, S.: Moving out of the meeting room: Exploring support for mobile meetings. In *Proceedings of the Sixth European Conference on Computer Supported Cooperative Work (ECSCW'99)*, pp. 81-98, 1999.
- [BDM+98] Banavar, G., Doddapaneti, S., Miller, K. and Mukherjee, B.: Rapidly Building Synchronous Collaborative Applications by Direct Manipulation. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work (CSCW'98)*, pp. 139-148, 1998.
- [BeBl96] Bellotti, V. and Bly, S.: Walking away from the desktop computer: distributed collaboration and mobility in a product design team. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work (CSCW'96)*, pp. 209-218, 1996.
- [Benn97] Bennett, K.H.: Software Maintenance: a Tutorial. In M. Dorfman and R. H. Thayer (Editors), *Software Engineering*, IEEE Computer Society, pp. 289-303, 1997.

- [Blue99] Bluetooth Special Interest Group: *Specification of the Bluetooth System*. Version 1.0 B, 1999. Available at: <http://www.bluetooth.com/>.
- [BMR+96] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M.: *Pattern Oriented Software Architecture: A System of Patterns*. John Wiley & Son, 1996.
- [Booc94] Booch, G., *Object-oriented analysis and design*. The Benjamin-Cummings Publishing Company, Inc, California, USA, second edition, 1994.
- [BoRJ98] Booch, G., Rumbaugh, J. and Jacobson, I.: *The Unified Modelling Language user guide*. Addison Wesley, USA, 1998.
- [BPH+90] Beard, D., Palaniappan, M., Humm, A., Banks, D., Nair, A. and Shan, Y.-P.: A Visual Calendar for Scheduling Group Meetings. In *Proceedings of the 1990 ACM Conference on Computer Supported Cooperative Work (CSCW'90)*, pp. 279-290, 1990.
- [BrGo92] Brinck, T. and Gomez, L.M.: A Collaborative Medium for the Support of Conversational Props. In *Proceedings of the 1992 ACM Conference on Computer Supported Cooperative Work (CSCW'92)*, pp. 171-178, 1992.
- [BrKE99] Bradner, E., Kellogg, W.A. and Erickson, T.: The adoption and use of 'BABBLE': A Field Study of Chat in the Workplace. In *Proceedings of the Sixth European Conference on Computer Support Cooperative Work (ECSCW'99)*, pp. 139-158, 1999.
- [BrPe84] Bracchi, G. and Percini, B.: The design requirements of office systems. *ACM Transactions on Office Systems*, 2 (2), pp. 151-170, 1984.
- [Bruc98] Bruckman, A.: Community Support for Constructionist Learning. *Computer Supported Cooperative Work: The Journal of Collaborative Computing, Special Issue on Interaction and Collaboration in MUDS*, 7 (1/2), pp. 47-86, 1998.
- [ChBl99a] Churchill, E.F., Bly, S.: Virtual environments at work: ongoing use of MUDs in the workplace. In *Proceedings of the International Joint Conference on Work Activities Coordination and Collaboration (WACC'99)*, pp. 99-108, 1999.
- [ChBl99b] Churchill, E.F. and Bly, S.: It's all in the words: supporting work activities with lightweight tools. In *Proceedings of the 1999 International ACM Conference on Supporting Group Work (GROUP'99)*, pp. 40-49, 1999.
- [ChDa00] Cheesman, J. and Daniels, J.: *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2000.
- [Dour96] Dourish, P.: *Open Implementation and Flexibility in CSCW Toolkits*. PhD Thesis, Department of Computer Science, University College London, London, UK, 1996.
- [Dour98] Dourish, P.: Introduction: The State of Play. *Computer Supported Cooperative Work: The Journal of Collaborative Computing, Special Issue on Interaction and Collaboration in MUDS*, 7(1/2), pp. 1-7, 1998.
- [DSW98] D'Souza, D.F. and Wills, A.C.: *Objects, Components, and Frameworks With Uml: The Catalysis Approach*. Addison-Wesley, 1998.
- [DSWi99] D'Souza, D. F. and Wills, A. C.: *Objects, Components and Frameworks with UML: the Catalysis Approach*. Addison Wesley, USA, 1999.

- 
- [EJL+99] Eertink, H., Janssen, W., Luttighuis, P.O., Teeuw, W. and Vissers, C.A.: A Business Process Design Language. In *1999 World Congress on Formal Methods (FM'99), Vol. I, LNCS 1708*, pp. 76-95, 1999.
- [EIBe82] Ellis, C.A. and Bernal, M.: OFFICETALK-D: An Experimental Office Information System. *Proceedings of the ACM-SIGOA Conference on Office Automation Systems*, pp 131-140, 1982.
- [ElGR91] Ellis, C.A., Gibbs, S.J. and Rein, G.L.: Groupware: Some issues and experiences. *Communications of the ACM*, 34 (1), pp. 38-58, 1991.
- [Elli79] Ellis, C.A.: Information control nets: A mathematical model of office information systems. *Proceedings of the ACM-SIGOA Conference on Simulation, Measurement and Modeling of Computer Systems*, pp. 225-239, 1979.
- [ErPe00] Eriksson, H.-E. and Penker, M.: *Business Modeling With UML: Business Patterns at Work*. John Wiley & Sons (USA), 2000.
- [FaDi00] de Farias, C. R. G., and Diakov, N.: Component-Based Groupware Tailorability using Monitoring Facilities. In *Proceedings of the ACM CSCW 2000 Workshop on Component-Based Groupware (CBG'00)*, Philadelphia (PA - USA), 2000.
- [FaDP99] de Farias, C.R.G., Diakov, N. and Poortinga, R.: Analysis of UML. *Amidst Deliverable: D1.1.2, AMIDST/WP1/N006/V04*, 1999.
- [FaFS00a] de Farias, C. R. G., Ferreira Pires, L., and van Sinderen, M.: A conceptual model for the development of CSCW systems. In *Proceedings of the Fifth International Conference on the Design of Cooperative Systems (COOP 2000)*, Sophia Antipolis, pp. 189-204, 2000.
- [FaFS00b] de Farias, C. R. G., Ferreira Pires, L. and van Sinderen, M.: A component-based groupware development methodology. In *Proceedings of the Fourth International Enterprise Distributed Object Computing Conference (EDOC'00)*, pp. 204-213, 2000.
- [FaFS99] de Farias, C. R. G., Ferreira Pires, L., and van Sinderen, M.: Conceptual frameworks for the development of CSCW systems. *CTIT Technical Report series*, No. 99-16, University of Twente, October/1999.
- [FaSF00] de Farias, C. R.G., van Sinderen, M., and Ferreira Pires, L.: A systematic approach for component-based software development. In *Proceedings of the Seventh European Concurrent Engineering Conference (ECEC'00)*, pp. 127-131, 2000.
- [Faul97] Faulk, S.R.: Software Requirements: a Tutorial. In M. Dorfman and R. H. Thayer (Editors), *Software Engineering*, IEEE Computer Society, pp. 82-103, 1997.
- [FeFa01] Ferreira Pires, L. and de Farias, C.R.G: AMBER: uma linguagem para o desenvolvimento de sistemas distribuídos. In *Electronic Proceedings of the XIX Simpósio Brasileiro de Redes de Computadores (SBRC 2001)*, Florianópolis (Brazil), pp. 82-97, 2001.
- [Ferre94] Ferreira Pires, L.: *Architectural notes: a framework for distributed systems development*. PhD thesis, University of Twente, Enschede, the Netherlands, 1994.
- [FFS+01] de Farias, C. R. G., Ferreira Pires, L., van Sinderen, M., and Quartel, D.: A Combined Component-Based Approach for the Design of Distributed Software
-

- Systems. In *Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'01)*, Bologna (Italy), pp. 2-8, 2001.
- [FGL+92] Fischer, G., Grudin, J., Lemke, A., McCall, R., Ostwald, J., Reeves, B. and Shipman, F.: Supporting Indirect Collaborative Design with Integrated Knowledge-Based Design Environments. *Human-Computer Interaction, Special Issue on Computer-Supported Cooperative Work*, 7(3), pp. 281-314, 1992.
- [FiKC90] Fish, R.S., Kraut, R.E. and Chalfonte, B.L.: The VideoWindow System in Informal Communications. In *Proceedings of the 1990 ACM Conference on Computer Supported Cooperative Work (CSCW'90)*, pp. 1-11, 1990.
- [FiKM96] Fitzpatrick, G., Kaplan, S. and Mansfield, T.: Physical Spaces, Virtual Places and Social Worlds: A Study of Work in the Virtual. In *Proceedings of the 1996 ACM Conference on Computer Supported Work (CSCW'96)*, pp. 334-343, 1996.
- [Fisc00] Fischer, P.: Enabling Component Model Integration. *Application Development Trends*, 7 (11), pp. 35-44, November 2000.
- [FiTK95] Fitzpatrick, G., Tolone, W.J. and Kaplan, S. M.: Work, Locales and Distributed Social Worlds. In *Proceedings of the Fourth European Conference on Computer Supported Cooperative Work (ECSCW '95)*, pp. 1-16, 1995.
- [FKL+88] Fish, R.S., Kraut, R.E., Leland, M.D.P. and Cohen, M.: Quilt: A collaborative tool for cooperative writing. In *Conference on Office Information Systems, SIGOIS bulletin*, 9(2-3), pp. 30-37, 1988.
- [FMK+99] Fitzpatrick, G., Mansfield, T., Kaplan, S., Arnold, D., Phelps, T. and Segall, B.: Augmenting the workaday world with Elvin. In *Proceedings of the Sixth European Conference on Computer Support Cooperative Work (ECSCW'99)*, pp. 431-450, 1999.
- [FoSc97] Fowler, M. and Scott, K.: *UML Distilled: Applying the Standard Object Modeling Language*. MA: Addison-Wesley, 1997.
- [Fuch99] Fuchs, L.: AREA: A cross-application notification service for groupware. In *Proceedings of the Sixth European Conference on Computer Supported Cooperative Work (ECSCW '99)*, pp. 61-80, 1999.
- [GaKr90] Galegher, J. and Kraut, R.E.: Technology for Intellectual Teamwork: Perspectives on Research and Design. In J. Galegher, R.E. Kraut and C. Egidio (eds.), *Intellectual teamwork: social and technological foundations of cooperative work*. Lawrence Erlbaum Associates, Hillsdale, NJ (USA), pp. 1-20, 1990.
- [GHJ+95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, USA, 1995.
- [GiDu97] Ginsburg, M. and Duliba, K.: Enterprise-Level Groupware Choices: Evaluating Lotus Notes and Intranet-Based Solutions. *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, 6 (2/3), pp. 201-225, 1997.
- [GNO+92] Goldberg, D., Nichols, D., Oki, B.M. and Terry, D.: Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35 (12), pp. 61 – 70, 1992.
- [GoSS92] Goldberg, Y., Safran, M. and Shapiro, E.: Active mail—a framework for implementing groupware. In *Proceedings of the 1992 ACM Conference on Computer Supported Cooperative Work (CSCW'92)*, pp. 75 – 83, 1992.

- 
- [Gree91] Greenberg, S.: Personalizable groupware: Accomodating individual roles and group differences. In *Proceedings of the Second European Conference of Computer Supported Cooperative Work (ECSCW '91)*, pp. 17-32, 1991.
- [Grei84] Greif, I.: The user interface of a personal calendar program. In *Human Factors and Interactive Computer Systems—Proceedings of the 1982 NYU Symposium on User Interfaces*, Y. Vassiliou (Editor), Ablex Publishing Corp. (USA), pp. 207-222, 1984.
- [GrSa87] Greif, I. and Sarin, S.: Data sharing in group work. *ACM Transactions on Information Systems*, 5(2), pp. 187-211, 1987.
- [GRW+92] Greenberg, S., Roseman, M., Webster, D. and Bohnet, R.: Human and Technical Factors of Distributed Group Drawing Tools. *Interacting with Computers, Special Issue on CSCW: Part 1*, 4(3), pp. 364-392, 1992.
- [HaFa99] Haberman, S. and Falciani, A.: *Mastering Lotus Notes R5*. Sybex (USA), 1999.
- [HaWi92] Haake, J.M. and Wilson, B.: Supporting collaborative writing of hyperdocuments in SEPIA. In *Proceedings of the 1992 ACM Conference on Computer Supported Cooperative Work (CSCW'92)*, pp. 138-146, 1992.
- [HeSi00] Herzum, P. and Sims, O.: *Business component factory: a comprehensive overview of component-based development for the enterprise*. John Wiley & Sons, USA, 2000.
- [Hoft98] ter Hofte, G. H.: *Working Apart Together: Foundations for component groupware*. PhD Thesis, Telematics Institute, Enschede, the Netherlands, 1998.
- [Holl95] Hollingsworth, D.: *The Workflow Reference Model*, WFMC-TC-1003, Issue 1.1, January 1995.
- [Hrub98] Hruby, P.: Structuring Design Deliverables with UML. In *Proceedings of UML'98 International Workshop*, pp. 251-260, 1998.
- [HsKI96] Hsu, M. and Kleissner, C.: ObjectFlow: Towards a Process Management Infrastructure. *Distributed and Parallel Databases*, 4(2), pp. 169-194, 1996.
- [HuMe00] Hummes, J. and Merialdo, B.: Design of Extensible Component-Based Groupware. In *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, 9 (1), pp. 53-74, 2000.
- [IBCo99] IBM Corporation: *IBM MQSeries Workflow: Concepts and Architecture, version 3.2.1*. Document Number GH12-6285-02, 1999. Internet: <http://www-4.ibm.com/software/ts/mqseries/workflow/>.
- [IsMi91] Ishii, H. and Miyake, N.: Toward an open shared workspace: computer and video fusion approach of TeamWorkStation. *Communications of the ACM*, 34(12), pp. 37-50, 1991.
- [ISO95] ISO/IEC: Reference Model of Open Distributed Processing: Part 1, *International Standard ISO/IEC JTC1/SC21/WG7*, 1995.
- [JaBR99] Jacobson, I., Booch, G. and Rumbaugh, J.: *The unified software development process*. Addison Wesley (USA), 1999.
- [Jaco95] Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1995.

- [JaTW99] Jackson, R.L., Taylor, W. and Winn, W.: Peer collaboration and virtual environments: a preliminary investigation of multi-participant virtual reality applied in science education. In *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC'99)*, pp. 121-125, 1999.
- [Joha91] Johansen, R. (Editor): *Leading Business Teams: How Teams Can Use Technology and Group Process Tools to Enhance Performance*. Addison-Wesley (USA), 1991.
- [KaSt00] Kandé, M.M. and Strohmeier, A.: Towards a UML Profile for Software Architecture Descriptions. In *Proceedings of the 3<sup>rd</sup> International Conference on the Unified Modeling Language (UML 2000): Advancing the Standard, LNCS, Vol. 1939*, pp. 513-527, 2000.
- [KMS+00] Kahler, H., Mørch, A., Stiemerling O. and Wulf, V.: Introduction of the Special Issue on Tailorable Systems and Cooperative Work. In *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, 9 (1), pp. 1-4, 2000.
- [KnPr90] Knister, M.J. and Prakash, A.: DistEdit: A distributed toolkit for supporting multiple group editors. In *Proceedings of the 1990 Conference on Computer Supported Cooperative Work (CSCW'90)*, pp. 343-354, 1990.
- [KnWe97] Knudsen, C. and Wellington, D.: Calendaring and Scheduling: Managing the Enterprise's Most Valuable, Non-Renewable Resource—Time. In D. Coleman (Editor), *Groupware: Collaborative Strategies for Corporate LANS and Intranets*. Prentice Hall (USA), pp. 115-141, 1997.
- [Kobr99] Kobryn, C.: UML 2001: a standardization odyssey. *Communications of the ACM*, 42 (10), 29-37, 1999.
- [KrHW93] Kreifelts, T., Hinrichs, E. and Woetzel, G.: Sharing To-Do Lists with a Distributed Task Manager. In *Proceedings of the Third European Conference on Computer Supported Cooperative Work (ECSCW'93)*, pp. 31-46, 1993.
- [KrLj98] Kristoffersen, S. and Ljungberg, F.: MobiCom: Networking Dispersed Groups. *Interacting with Computers*, 10 (1), pp. 45-65, 1998.
- [Kuut91] Kuutti, K.: The concept of activity as a basic unit of analysis for CSCW research. In *Proceedings of the Second European Conference on Computer Supported Cooperative Work (ECSCW'91)*, pp. 249-264, 1991.
- [Larm97] Larman, C.: *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall, USA, 1997.
- [Lew98] Lewandowski, S. M.: Frameworks for component-based client/server computing. *ACM Computing Surveys*, 30 (1), pp. 3-27, 1998.
- [LiPr00] Litiu, R. and Prakash, A.: Developing adaptive groupware systems using a mobile component framework. In *Proceedings of the ACM 2000 Conference on Computer Supported Cooperative Work (CSCW'00)*, pp. 107-116, 2000.
- [LiPr99] Litiu, R. and Prakash, A.: DACIA: A Mobile Component Framework for Building Adaptive Distributed Applications. *Technical Report CSE-TR-416-99*, University of Michigan, EECS, 1999.
- [LiWM99] Li, W., Wang, W. and Marsic, I.: Collaboration transparency in the DISCIPLE framework. In *Proceedings of the 1999 International ACM Conference on Supporting Group Work (GROUP'99)*, pp. 326-335, 1999.

- 
- [LJL+90] Lauwers, J.C., Joseph, T.A., Lantz, K.A. and Romanow, A.L.: Replicated architectures for shared window systems: a critique. In *Proceedings of the 1990 ACM Conference on Office Information Systems (COIS'90)*, pp. 249-260, 1990.
- [LuHe98] Luff, P. and Heath, C.: Mobility in collaboration. In *Proceedings of the 1998 ACM Conference on Computer-Supported Cooperative Work (CSCW'98)*, pp. 305-314, 1998.
- [MaCr90] Malone, T. W. and Crowston, K.: What is Coordination Theory and how can it help design cooperative work systems? In *Proceedings of the of the 1990 ACM Conference on Computer Supported Cooperative Work (CSCW'90)*, pp. 357-370, 1990.
- [MaGP99] Mark, G., Grudin, J. and Poltrock, S.E.: Meeting at the desktop: An empirical study of virtually collocated teams. In *Proceedings of the Sixth European Conference on Computer Support Cooperative Work (ECSCW'99)*, pp. 159-178, 1999.
- [MaLF95] Malone, T.W., Lai, K.-Y. and Fry, C.: Experiments with Oval: a radically tailorable tool for cooperative work. *ACM Transactions on Information Systems*, 13 (2), pp. 177-205, 1995.
- [Mars99] Marsic, I.: DISCIPLER: a framework for multimodal collaboration in heterogeneous environments. *ACM Computing Surveys*, 31 (2es), Article No. 4, 1999.
- [MaSc96] Marx, M. and Schmandt, C.: CLUES: Dynamic Personalized Message Filtering. In *Proceedings of ACM 1996 Conference on Computer Supported Cooperative Work (CSCW'96)*, pp. 113-121, 1996.
- [McSa93] McCarthy, D.R. and Sarin, S.K.: Workflow and Transactions in InConcert. *IEEE Data Engineering Bulletin*. 16 (2), pp. 53-56, 1993.
- [MGL+87] Malone, T.W., Grant, K.R., Lai, K.-Y., Rao, R. and Rosenblitt, D.: Semistructured Messages are Surprisingly Useful for Computer-Supported Coordination. *ACM Transactions on Office Information Systems*, 5(2), pp. 115-131, 1987.
- [MiCo96] Microsoft Corporation: *DCOM Technical Overview*. Microsoft Corporation White Paper, November 1996.
- [MiCo98] Microsoft Corporation: *An Introduction to Microsoft Transaction Server*. Microsoft Corporation White Paper, January 1998.
- [MiRa96] Michailidis, A. and Rada, R.: A review of collaborative authoring tools. In R. Rada (ed.), *Groupware and Authoring*. Academic Press, London, 1996, pp. 9-43.
- [MoMC98] Moran, T.P., van Melle, W. and Chiu, P.: Spatial Interpretation of Domain Objects Integrated into a Freeform Electronic Whiteboard. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'98)*, pp. 175-184, 1998.
- [Morc97] Mørch, A.: Three Levels of End-User Tailoring: Customization, Integration, and Extension. In M. Kyng and L. Mathiassen (eds.): *Computers and Design in Context*, Cambridge, MA: The MIT Press, pp. 51-76, 1997.
- [MWF+92] Medina-Mora, R., Winograd, T., Flores, R. and Flores, F.: The action workflow approach to workflow management technology. *Proceedings of the ACM 1992 Conference on Computer Supported Cooperative Work (CSCW'92)*, pp. 281-288, 1992.
-

- [Nard96] Nardi, B.A. (editor): *Context and Consciousness: Activity Theory and Human-Computer Interaction*. Cambridge, MA: MIT Press, 1996.
- [NaWB00] Nardi, B.A., Whittaker, S. and Bradner, E.: Interaction and outeraction: instant messaging in action. In *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work (CSCW'00)*, pp. 79-88, 2000.
- [Neib99] Neibauer, A.: *Running Microsoft Outlook 2000*. Microsoft Press (USA), 1999.
- [NKC+90] Neuwirth, C.M., Kaufer, D.S., Chandhok, R. and Morris, J.H.: Issues in the Design of Computer Support for Co-authoring and Commenting. In *Proceedings of the 1990 ACM Conference on Computer Supported Cooperative Work (CSCW'90)*, pp. 183-195, 1990.
- [OCL+93] Olson, J.S., Card, S.K., Landauer, T.K., Olson, G.M., Malone, T. and Leggett, J.: Computer-supported co-operative work: Research issues for the 90s. *Behaviour & Information Technology*, 12 (2), pp. 115-129, 1993.
- [OHE96] Orfali, R., Harkey, D. and Edwards, J.: *The Essential Distributed Objects Survival Guide*, John Wiley & Sons (USA), 1996.
- [OMG00a] Object Management Group: *The Common Object Request Broker: Architecture and Specification, version 2.4*. October 2000.
- [OMG00b] Object Management Group: *Meta Object Facility (MOF) Specification, version 1.3*, 2000. Available at <http://www.omg.org>.
- [OMG00b] Object Management Group: *The Common Object Request Broker: Architecture and Specification, version 2.4*. October 2000.
- [OMG01] Object Management Group: *Unified Modeling Language 1.4 specification*, 2001. Available at <http://www.omg.org>.
- [OMG01b] OMG Architecture Board: *Model Driven Architecture (MDA)*, 2001. Available at <http://www.omg.org>.
- [OMG97] Object Management Group: *A Discussion of the Object Management Architecture*. January 1997.
- [OMG99b] Object Management Group: *Analysis and Design Platform Task Force: White Paper on the Profile mechanism, Version 1.0*. OMG Document ad/99-04-07, 1999.
- [OMG99b] Object Management Group: *CORBA Components - Volume I*. OMG TC Document 99-07-01, 1999. Available at <http://www.omg.org>.
- [OMI+94] Okada, K.-I., Maeda, F., Ichikawaa, Y. and Matsushita, Y.: Multiparty Videoconferencing at Virtual Social Distance: MAJIC Design. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work (CSCW'94)*, pp. 385-393, 1994.
- [Ould95] Ould, A.A.: *Business Processes: Modelling and Analysis for Re-Engineering and Improvement*. John Wiley & Son, England, 1995.
- [PaDK96] Patterson, J.F., Day, M. and Kucan, J.: Notification servers for synchronous groupware. In *Proceedings of ACM 1996 Conference on Computer Supported Cooperative Work (CSCW'96)*, pp. 122-129, 1996.

- 
- [Pale99] Palen, L.: Social, individual and technological issues for groupware calendar systems. In *Proceedings of the 1999 Conference on Human Factors in Computing Systems (CHI'99)*, pp. 17-24, 1999.
- [Papo97] Papows, J.: Deploying Second-Generation Intranets with Lotus Notes. In D. Coleman (Editor), *Groupware: Collaborative Strategies for Corporate Lans and Intranets*. Prentice Hall (USA), pp. 347-369, 1997.
- [PaSS94] Pacull, F., Sandoz, A. and Schiper, A.: Duplex: a distributed collaborative editing environment in large scale. *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work (CSCW'94)*, pp. 165-173, 1994.
- [PaSY00] Palen, L., Salzman, M. and Youngs, E.: Going wireless: behavior & practice of new mobile phone users. In *Proceedings of the ACM 2000 Conference on Computer Supported Cooperative Work (CSCW'00)*, pp. 201-210, 2000.
- [Patt95] Patterson, J.F.: A taxonomy of architectures for synchronous groupware applications. *SIGOIS Bulletin*, 15 (3), pp. 27-29, 1995.
- [Peng93] Peng, C.: Survey of collaborative drawing support tools: Design perspectives and prototypes. *Computer Supported Cooperative Work (CSCW): The Journal of Collaborative Computing*, 1, pp. 197-228, 1993.
- [Pere97] Perey, C.: Desktop videoconferencing. In D. Coleman (Editor), *Groupware: Collaborative Strategies for Corporate Lans and Intranets*. Prentice Hall (USA), pp. 321-343, 1997.
- [Poll88] Pollock, S.: A Rule-Based Message Filtering System. *ACM Transactions on Office Information Systems*, 6(3), pp. 232-254, 1988.
- [QFS+97] Quartel, D., Ferreira Pires, L., van Sinderen, M., Franken, H. and Vissers, C.A.: On the role of basic design concepts in behaviour structuring. *Computer Networks and ISDN Systems*, 29 (4), pp. 413-436, 1997.
- [Quar98] Quartel, D.: *Action relations: basic design concepts for behaviour modelling and refinement*. PhD thesis, University of Twente, Enschede, the Netherlands, 1998.
- [Raym94] Raymond, K.A.: Reference Model of Open Distributed Processing: a Tutorial. In *Proceedings of the II IFIP International Conference on Open Distributed Processing*, pp.3-14, 1994.
- [RBP+91] Rumbaugh, J., Blaham M., Premerlani, W., Eddy, F. and Lorenzen, W. *Object-oriented modelling and design*. Prentice-Hall, Englewood Cliffs, 1991.
- [RIS+94] Resnick, P., Iacovou, N., Suchak, M., Bergstrom, P. and Riedl, J.: GroupLens: An Open Architecture for Collaborative Filtering of Netnews. In *Proceedings of ACM 1994 Conference on Computer Supported Cooperative Work (CSCW'94)*, pp. 175-186, 1994.
- [RoGr92] Roseman, M. and Greenberg, S.: GroupKit: a Groupware Toolkit for Building Real-Time Conferencing Applications. In *Proceedings of the ACM 1992 Conference on Computer Supported Cooperative Work (CSCW'92)*, pp. 43-50, 1992.
- [RoGr96] Roseman, M. and Greenberg, S.: Building Real Time Groupware with GroupKit, A Groupware Toolkit. *ACM Transactions on Computer Human Interaction*, 3(1), pp. 66-106, 1996.

- [RoUn00] Roth, J. and Unger, C.: Developing synchronous collaborative applications with TeamComponents. In *Designing Cooperative Systems: the Use of Theories and Models, Proceedings of the 5th International Conference on the Design of Cooperative Systems (COOP'00)*, pp. 353-368, 2000.
- [Royc70] Royce, W.W.: Managing the development of large software systems: concepts and techniques. *Proceedings of the Western Electronic Show and Convention (WESCON)*, pp. 1-9, 1970. Reprinted in: *Proceedings of the 9<sup>th</sup> International Conference on Software Engineering (ICSE'87)*, pp. 328-338, 1987.
- [RuGG96] Rudolph, E., Graubmann, P. and Grabowski, J.: Tutorial on Message Sequence Charts. In *Computer Networks and ISDN Systems*, 28 (12), pp. 1629-1641, 1996.
- [SaHC93] Sasse, M.A., Handley, M.J. and Chuang, S.C.: Support for collaborative authoring via e-mail: the MESSIE environment. In *Proceedings of the Third European Conference on Computer Supported Cooperative Work (ECSCW'93)*, pp.249-264, 1993.
- [Same97] Sametinger, J.: *Software engineering with reusable components*. Springer-Verlag, 1997.
- [Sand92] Sanderson, D.: The CSCW implementation process: an interpretative model and case study of the implementation of a videoconference system. In *Proceedings of the 1992 ACM Conference on Computer Supported Cooperative Work (CSCW'92)*, pp. 370-377, 1992.
- [SaSF01] Santos, H.F., de Souza, W.L., de Farias, C.R.G.: Conferência Multimídia - Metodologia e Componentes para Aplicações Cooperativas. In *Proceedings of the 7th Brazilian Symposium on Multimedia and Hypermedia Systems (SBMidia'01)*, pp. 187-202, 2001.
- [SaTr94] Santos, A. and Tritsch, B.: Cooperative multimedia editing tool for enhanced group communication. *Computer communications*, 17 (4), pp. 277-287, 1994.
- [SBF+87] Stefik, M., Bobrow, D.G., Foster, G., Lanning, S. and Tatar, D.: WYSIWIS revised: early experiences with multiuser interfaces. *ACM Transactions on Office Information Systems*, 5(2), pp. 147-167, 1987.
- [Scha93] Schach, S.R.: *Software Engineering*. Aksen Associates (USA), 1993.
- [Ses00] Sessions, R.: *COM+ and the Battle for the Middle Tier*. John Wiley & Sons (USA), 2000.
- [SFB+87] Stefik, M., Foster, G., Bobrow, D.G., Kahn, K., Lanning, S. and Suchman, L.: Beyond the chalkboard: computer support for collaboration and problem solving in meetings. *Communications of the ACM*, 30 (1), pp. 32-47, 1987.
- [SGH+94] Streitz, N.A., Geißler, J., Haake, J.M. and Hol, J.: DOLPHIN: integrated meeting support across local and remote desktop environments and LiveBoards. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work (CSCW'94)*, pp. 345-358, 1994.
- [SHH+92] Streitz, N., Haake, J., Hannemann, J., Lemke, A., Schuler, W., Schütt, H. and Thüring, M.: SEPIA: a cooperative hypermedia authoring environment. In *Proceedings of the ACM Conference on Hypertext*, pp. 11-22, 1992.
- [SiGr99] Simmons, A.J.H. and Graham, I.: 30 Things That Go Wrong In Object Modelling With UML 1.3. In H. Kilov, B. Rumpe and I. Simmonds (Editors), *Behav-*

- 
- ioral Specifications of Businesses and Systems*. Kluwer Academic Publishers (Netherlands), pp. 237-257, 1999.
- [SiJF99] van Sinderen, M.J., Josten, S.M.M. and de Farias, C.R.G.: Workflow Automation Based on OSI Job Transfer and Manipulation. *Computer Standards & Interfaces*, 21(5), pp. 403-415, 1999.
- [Sind95] van Sinderen, M.: *On the design of application protocols*. PhD thesis, University of Twente, Enschede, the Netherlands, 1995.
- [SKB+98] Sarwar, B.M., Konstan, J.A., Borchers, A., Herlocker, J., Miller, B. and Riedl, J.: Using Filtering Agents to Improve Prediction Quality in the GroupLens Research Collaborative Filtering System. In *Proceedings of 1998 ACM Conference on Computer Supported Cooperative Work (CSCW'98)*, pp. 345-354, 1998.
- [SKS+96] Schuckmann, C., Kirchner, L., Schümmer, J. and Haake, J. M.: Designing object-oriented synchronous groupware with COAST. *Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work (CSCW'96)*, pp. 30-38, 1996.
- [SiBi00] Slagter, R.J. and Biemans, M.C.M.: Component groupware: a basis for tailorable solutions that can evolve with the supported task. In *Proceedings of the International ICSC Conference on Intelligent Systems and Applications (ISA 2000)*, 2000.
- [SiDo01] Slagter, R. and ter Doest, H.: *The CoCoWare Component Architecture*. GigaCSCW deliverable (D2.1.4), 2001.
- [SiHo99] Slagter, R. and ter Hofte, H.: *Component models and component groupware architectures*. TI/RS/99023, GIGACSCW/D2.1.1, Telematica Instituut, the Netherlands, 1999.
- [SMD+97] Spellman, P.J., Mosier, J.N., Deus, L.M. and Carlson, J.A.: Collaborative virtual workspace. In *Proceedings of the 1997 International ACM Conference on Supporting Group Work (GROUP'97)*, pp. 197-203, 1997.
- [SMM+94] Swenson, K.D., Maxwell, R.J., Matsumoto, T., Saghari, B. and Irwin, K.: A Business Process Environment Supporting Collaborative Planning, *Collaborative Computing*, 1(1), pp. 15-34, 1994.
- [SoCh94] Sohlenkamp, M. and Chwelos, G.: Integrating communication, cooperation, and awareness: the DIVA virtual office environment. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work (CSCW'94)*, pp. 331-343, 1994.
- [Staf99] Staffware: *Staffware 2000*. White Paper, 1999. Internet: <http://www.staffware.com>.
- [StCr98] Stiemerling, O. and Cremers, A. B.: Tailorable Component Architectures for CSCW-Systems. *Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Programming*, Madrid (Spain), pp. 302-308, 1998.
- [Steg97] Stegman, C.: Electronic Mail and Messaging. In D. Coleman (Editor), *Groupware: Collaborative Strategies for Corporate LANS and Intranets*. Prentice Hall (USA), pp. 85-111, 1997.
- [StHC99] Stiemerling, O., Hinken, R. and Cremers, A. B.: Distributed Component-Based Tailorability for CSCW applications. *Proceedings of the Fourth International*
-

- Symposium on Autonomous Decentralized Systems (ISADS '99)*, IEEE Press, Tokyo (Japan), pp. 345-352, 1999.
- [StHC99a] Stiemerling, O., Hinken, R. and Cremers, A.B.: Distributed Component-Based Tailorability for CSCW Applications. In *Proceedings of the 4th International Symposium on Autonomous Decentralized Systems (ISADS '99)*, pp. 345-352, 1999.
- [StHC99b] Stiemerling, O., Hinken, R. and Cremers, A.B.: The EVOLVE Tailoring Platform: Supporting the Evolution of Component-Based Groupware. In *Proceedings of the 3rd International Enterprise Distributed Object Computing Conference (EDOC'99)*, pp. 106-115, 1999.
- [Stie00] Stiemerling, O.: *Component-Based Tailorability*. PhD thesis, University of Bonn, Bonn, Germany, 2000.
- [SuMi00] Sun Microsystems: *Enterprise JavaBeans™ Specification, Version 2.0*. Proposed Final Draft, October 2000.
- [Syri97] Syri, A.: Tailoring Cooperation Support through Mediators. In *Proceedings of the Fifth European Conference on Computer Supported Cooperative Work (ECSCW '97)*, pp. 157-172, 1997.
- [Szyp98] Szyperski, C.: *Component software: beyond object-oriented programming*, Addison-Wesley, USA, 1998.
- [TaIR94] Tang, J.C., Isaacs, E.A. and Rua, M.: Supporting Distributed Groups with a Montage of Lightweight Interactions. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work (CSCW'94)*, pp. 23-34, 1994.
- [TaMi91] Tang, J.C. and Minneman, S.L.: Videodraw: a video interface for collaborative drawing. *ACM Transactions on Information Systems*, 9 (2), pp. 170 – 184, 1991.
- [TaRu94] Tang, J.C. and Rua, M.: Montage: providing teleproximity for distributed groups. In *Proceedings on 1994 ACM Conference on Human Factors in Computing Systems: “celebrating interdependence” (CHI'94)*, pp. 37-43, 1994.
- [Teeg00] Teege, G.: Users as Composers: Parts and Features as a Basis for Tailorability in CSCW Systems. In *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, 9 (1), pp. 101-122, 2000.
- [Teeg96] Teege, G.: Object-Oriented Activity Support: A Model for Integrated CSCW Systems. *Computer Supported Cooperative Work (CSCW): The Journal of Collaborative Computing*, 5(1), pp. 93-124, 1996.
- [Tho98] Thomas, A.: *Enterprise JavaBeans™ Technology: Server Component Model for the Java™ Platform*. Patricia Seybold Group, December 1998.
- [TrRB93] Trevor, J., Rodden, T. and Blair, G.: COLA: A lightweight platform for CSCW. In *Proceedings of the Third European Conference on Computer Supported Cooperative Work (ECSCW '93)*, pp. 15-30, 1993.
- [TrRM94] Trevor, J., Rodden, T. and Mariani, J.: The use of adapters to support cooperative sharing. In *Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work (CSCW'94)*, pp. 219-230, 1994.
- [Vaug01] Vaughan, J.: UML hits the street. *Application Development Trends*, 8 (09), 2001.

- 
- [Vayd99] Vayda, T.: Organizing for components: managing risk and maximizing reuse. *Components Strategies*, 1999.
- [VFQ+00] Vissers, C.A., Ferreira Pires, L., Quartel, D.A.C. and Sinderen, M.J.: *The architectural design of distributed systems*. Reader for the Design of Telematics Systems, University of Twente, the Netherlands, 2000.
- [Vlie00] van Vliet, H.: *Software Engineering: Principles and Practice*. John Wiley & Sons (USA), 2000.
- [WaK199] Warmer, J. and Kleppe, A.: *The object constraint language: precise modeling with UML*. Addison-Wesley, USA, 1999.
- [WAP99] WAP Forum: *WAP 1.2 Specification Suite*, 1999. Available at: <http://www.wapforum.org/>.
- [Wege98] de Weger, M.: *Structuring of Business Processes: An architectural approach to distributed systems development and its application to business processes*. PhD thesis, University of Twente, Enschede, the Netherlands, 1998.
- [Weis95] Weiser, M.: The computer for the 21st century. *Scientific American, Special Issue: The Computer in the 21st Century*, pp. 78-80, 85-89, 1995.
- [WfMC99] Workflow Management Coalition: *Terminology & Glossary*. WfMC-TC-1011, Issue 3.0, February 1999.
- [Wier98] Wieringa, R.: A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys*, 30 (4), 1998.
- [Wier99] Wieringa, R.J.: Embedding Object-Oriented Design in System Engineering. In H. Kilov, B. Rumpe and I. Simmonds (Editors), *Behavioral Specifications of Businesses and Systems*. Kluwer Academic Publishers (Netherlands), pp. 287-310, 1999.
- [WiKi94] Willians, S., Kindel, C.: The Component Object Model: A Technical Overview. *Microsoft Corporation White Paper*, October 1994.
- [Will00a] Williams, J.D.: Raising components. *Application Development Trends*, 7 (9), pp. 27-32, 2000.
- [Will00b] Williams, J.D.: Consuming Components. *Application Development Trends*, 7 (10), 2000.
- [Wils99] Wilson, C.: Application Architectures with Enterprise JavaBeans. *Component Strategies*, 2(2), pp. 25-34, 1999.
- [WSM+90] Watabe, K., Sakata, S., Maeno, K., Fukuoka, H. and Ohmori, T.: Distributed Multiparty Desktop Conferencing Systems: MERMAID. In *Proceedings of the 1990 ACM Conference on Computer Supported Cooperative Work (CSCW'90)*, pp. 27-38, 1990.
- [ZWS+99] M., Wu, L., Sun, L., Li, Y. and Bing, Y.: VCS: a virtual environment support for awareness and collaboration. In *Proceedings of the 7<sup>th</sup> ACM International Conference (part 2) on Multimedia*, pp. 163-165, 1999.



---

# Index

Entries in *italics* refer mostly to existing groupware systems and platforms. Other entries refer predominantly to definitions.

## A

abstraction, 45  
  levels, 47  
  versus refinement, 47  
*Action Workflow*, 26  
*Action-Interaction Theory*, 70  
*ActionWorks Metro*, 26  
*Active Mail*, 30  
*Activity Theory*, 67–68  
activity unit, 76  
  action, 80  
  interaction, 81  
    actor, 82  
    initiator, 113  
    reactor, 83, 113  
ad hoc development, 2  
auxiliary entity, 111  
awareness, 2

## B

*BABBLE*, 28

## C

*Calendar Manager*, 30  
Catalysis, 58–59  
*CES*, 29  
chat applications, 27–28  
*CLUES*, 27  
*COAST*, 4  
co-authoring systems, 29  
*CoCoWare*, 40–41  
*COLA*, 4–5  
collaboration concern layers, 153  
  coupling, 156  
collaborative virtual environment, 31–32  
*Collaborative Virtual Workspace*, 31  
*CoMedia*, 29  
component. *See* software component  
component model, 13  
  COM, 15–17  
  CORBA, 20–21

  EJB, 17–18  
  component versus object, 13–14  
concern level, 49  
  component, 52  
  enterprise, 52  
  object, 53  
  system, 52  
conferencing systems, 32–33  
consistency management, 2  
constraint, 186  
container, 21  
*Conversation Board*, 28  
cooperative process, 72  
cooperative work metamodel, 64  
*Coordination Theory*, 66–67

## D

*DACIA*, 41–42  
design, 51  
design methodology, 7  
  quality properties, 7  
development for reuse, 179  
development with reuse, 179  
*DISCIPLE*, 39–40  
*DistEdit*, 29  
*DIVA*, 31, 33  
*DOLPHIN*, 29  
*DreamTeam*, 42–43  
*Duplex*, 29

## E

e-mail systems, 26–27  
enterprise, 61  
  entity, 61  
  metamodel, 63  
  model, 62  
*EVOLVE*, 38–39

## F

functional behaviour, 76  
functional entity, 76

functional role, 76

## G

*GCW*, 32

group scheduling and calendaring systems, 29–31

*GroupDraw*, 28

*Groupkit*, 3–4

*GroupLens*, 27, 32

*GroupSketch*, 28

groupware, 1

application, 24

environment, 25

platform, 2, 3, 25

system, 24

toolkit. *See* groupware platform

*GROVE*, 29

## I

*ICQ*, 28

*InConcert*, 26

*Information Lens*, 27

interaction point, 113

invocation pattern, 114

ISCREEN, 27

## L

latecoming, 2

*Live*, 37–38

*Lotus Notes*, 32

*Lotus Organizer*, 30

## M

*MAJIC*, 33

*MERMAID*, 33

*MESSIE*, 29

*Microsoft Outlook Express*, 27

model, 7

*Montage*, 33

*MOOSE Crossing*, 31

*MPCAL*, 30

*MQSeries WorkFlow*, 26

MUD environment. *See* collaborative virtual environment

## N

*NetMeeting*, 33

*Netscape Messenger*, 27

*Network*, 28

notification pattern, 114

## O

observable equivalence, 170

*Officetalk-D*, 26

*OOActSM*, 71–72

*Outlook Calendar*, 30

## P

*Pattern of Four Deliverables*, 59

*PCAL*, 30

perspective. *See* view

decomposed required service, 116

distributed, 81

distributed with role discrimination, 82

integrated, 80

integrated internal, 116

required service, 115

*PREP*, 29

process model, 51

## Q

*Quilt*, 29

## R

reference specification, 79

*Regatta*, 26

relationship, 76

resource, 76

*RTCAL*, 33

## S

*SEPIA*, 29

shared whiteboard systems, 28–29

software component, 6, 11

application, 151

composite, 13

compound. *See* composite

groupware, 151

platforms, 22–23

repository, 180

simple, 12, 151

software development methodology. *See* process model

specification, 45

*Staffware*, 26

step-wise design, 47

stereotype, 186

subject entity, 111

system, 24

---

## T

tag definition, 186  
tagged value, 186  
tailorability, 5  
    customisation, 6  
    extension, 6  
    integration, 6  
*Tapestry*, 27  
*Task Manager*, 70–71  
*TBI/InConcert*, 26  
*TeamRooms*, 31  
*TeamWorkStation*, 28  
telematics system, 24  
*Tickerchat*, 28  
time-place matrix, 34  
*Tivoli*, 28

## U

Unified Modeling Language, 8  
    four-layer architecture, 185  
    profile, 185

    heavyweight extension, 186  
    lightweight extension, 186

Unified Process, 56–58

*Usenet*, 32

user entity, 111

## V

VCS, 32

*Videodraw*, 28

*VideoWindow*, 33

view, 48

    behavioural, 54

    interactional, 54

    structural, 54

*Visual Scheduler*, 30

## W

*Waterfall Glen*, 31

*waterfall model*, 51

workflow management systems, 26

*Worlds*, 32



---

# Summary

Component-based groupware development has gained increasing support in the past few years. This research area aims at constructing groupware systems by assembling prefabricated, configurable and independently evolving building blocks, the so-called software components.

A lot of attention has been given by the CSCW community to the development of component-based groupware toolkits, partially neglecting the provision of guidelines to cope with reuse of existing components and the design of new components. Thus, this work focuses on how to systematically design component-based groupware systems. We provide a methodology based on the standard modelling notation Unified Modeling Language (UML).

Our design methodology is mainly based on the architectural concepts of concern levels and views/perspectives. Concern levels are used to structure the design trajectory of a groupware system as a whole. Views and perspectives are used to focus the design activities on specific sets of concerns within a concern level.

Three concern levels form the basis of our architectural design trajectory, viz., enterprise, system and component. The enterprise level is concerned with the abstract representation of a cooperative work process. The system level is concerned with the provision of an automated support for (a part of) the cooperative work process specified at the enterprise level. The component level is concerned with the provision of the automated support specified at the system level in terms of a number of interrelated components.

Within the enterprise level three different perspectives are identified, viz., integrated, distributed and distributed with role discrimination. The integrated perspective is concerned with the integrated description of a cooperative work process, i.e., without specifying individual contributions of the enterprise entities. The distributed perspective is concerned with the specification of the individual contributions of the enterprise entities to the cooperative work process. The distributed perspective with role discrimination is concerned with the specification of the primary responsibility for the execution of units of behaviour common to two enterprise entities.

Within the system and component level three different perspectives are also identified, viz., required service, decomposed required service and integrated internal perspective. The required service perspective is concerned with the description of the service provided by an entity to its service users. The decomposed required service perspective is concerned with the description of the service provided by an entity to its service users and its relationship to the service required by this entity from other entities, called auxiliary entities, to provide its own required service. The integrated internal perspective is concerned with the description of the internal behaviour of an entity in terms of the activities executed by this entity in an integrated way.

Three separate views are used across all concern levels and perspectives defined within these levels, viz., structural, behavioural and interactional. The structural view is concerned with static aspects of behaviour. The behavioural view is concerned with dynamic aspects of the

behaviour of a single entity, while the interactional view is concerned with dynamic aspects of the behaviour of multiple entities.

General design concepts are introduced along the different concern levels. Based on these concepts, specification techniques and guidelines are presented. These specification techniques are targeted to the characteristics of the set of concerns being modelled. Guidelines to help in the refinement process and refinement assessment afterwards are also provided.

The architectural design of component-based groupware systems using UML as design notation shows a number of shortcomings associated with this language in its current version (UML 1.4). To cope with some of these limitations a number of extensions to UML are provided and documented in a profile.

In order to illustrate the main aspects of our methodology a comprehensive case study involving the architectural design of an electronic meeting system (EMS) is described. The EMS designed contains both chat and voting functionality.