



# RENAISSANCE

*Method and tool support for the  
evolution and re-engineering of legacy systems*

## Modelling for System Evolution

©RENAISSANCE Consortium 1997-1998

**Version 3.0, Published July 1998**

The RENAISSANCE project is partially funded by the European Commission under the Framework Initiative (ESPRIT 22010). The objective of the project is to develop a systematic method to support the re-engineering of legacy systems. Further information about the project is available on the World-Wide-Web at URL:

<http://www.comp.lancs.ac.uk/projects/renaissance/>

The members of the RENAISSANCE Consortium are:

**CAP Gemini Innovation** (Mr Alain Dineur)

Bâtiment Karélian  
7, chemin de la Dhuy  
38340 Meylan, FRANCE  
Tel: +33 476 76 47 47; Fax: +33 476 76 47 48

**CAP Gemini IST** (Mr Alain Paoli)

Tour Anjou  
33 Quai de Dion Bouton  
92814 PUTEAUX Cedex, FRANCE  
Tel: +33 1 41 26 63 36; Fax: +33 1 41 26 52 17

**debis Systemhaus GEI GmbH** (Mr Markus Breuer)

Pascalstraße 14  
D-52076 AACHEN, Germany  
Phone: +49 2408 943 0; Fax: +49 2408 943 119

**INTECS Sistemi S.p. A.** (Mr Giancarlo Savoia)

Via Livia Gereschi, 32  
56127 PISA, Italy  
Tel: +39 50 545 111; Fax: +39 50 545 200

**Telesoft S.p. A.** (Mr Fabio Mungo)

Via Degli Agrostemi, 30  
00040 SANTA PALOMBA (Roma), Italy  
Tel: +39 6 710 551; Fax: +39 6 710 553 50

**Engineering - Ingegneria Informatica S.p. A.** (Mr Dario Avallone)

Via dei Mille, 56  
I-00185 ROMA, Italy  
Tel: +39 6 522 431; Fax: +39 6 522 432 48

**Lancaster University** (Prof. Ian Sommerville)

Computing Dept,  
Bailrigg, LANCASTER LA1 4YR, UK  
Tel: +44 1524 593795; Fax: +44 1524 593608

**SINTEF** (Prof. Reidar Conradi)

O. S. Bragstads plass 2 F  
N-7034 TRONDHEM, Norway  
Tel: +47 73 593 444; Fax: +47 73 594 466

**This document's authors: SINTEF (Eirik Tryggeseth/Andreas Brendstuen), debis  
Systemhaus (Markus Breuer), Gap Gemini Innovation (Claude  
Villerman)**



## Executive Summary

When business needs demand changes to an established core system in the organization, an evolution project must be initiated. All too often, the current system is

- lacking documentation,
- expensive to maintain, and
- uses outdated technology.

These problems are typically exacerbated by the facts that the

- original system developers have left the organization,
- current maintainers do not have sufficient system knowledge, and
- new maintainers may not be skilled in the old technology used.

In order to make decisions about the system evolution, knowledge of the system must be elicited and made explicit.

This document provides the system architect with a bag of techniques for modelling system aspects that must be understood in order to make evolutionary decisions.

The document focuses on two aspects of modelling existing systems for evolution:

- *Context modelling*: This part of the document describes the conceptual aspects that must be understood in order to obtain an overall understanding of the system. Modelling techniques are described together with guidelines for using them, and with annotated examples.
- *Technical modelling*: When decisions have been taken about what strategies to use in the system evolution project, further details about the current system must be known in order to implement these strategies. The technical modelling part of the document identifies properties and relationships that must be identified and modelled in existing systems. The document concerns applications originally implemented using either 3GL or 4GL technology.

We further argue that similar models must be made for the target system, to ensure that the new system does not become tomorrow's legacy system.

The Unified Modelling Language (UML) is chosen as an integrative modelling technique for the technical modelling. We explain how the UML should be used in the modelling, and we give an overview of UML in an appendix.

*This page is intentionally left blank*

# Table of Contents

<b>1</b>	<b>INTRODUCTION .....</b>	<b>1</b>
1.1	Objectives .....	2
1.2	Rationale.....	3
1.3	Modelling activities during system evolution.....	3
1.4	Overview of the document structure .....	6
<b>2</b>	<b>CONTEXT MODELLING.....</b>	<b>7</b>
2.1	Introduction .....	8
2.2	Objective and Content of Context Models .....	11
2.3	Operational Schemas .....	12
2.3.1	Operational Schemas for system evolution .....	12
2.3.2	The elements of operational schemas.....	13
2.3.3	Using operational schemas for modelling the library case.....	15
2.3.4	Tips and tricks .....	16
2.3.5	Rediscovering Operational schemas in the legacy system .....	17
2.4	Use Case Diagrams .....	17
2.4.1	Use Case Diagrams for system evolution.....	17
2.4.2	The elements of use case diagrams .....	18
2.4.3	Tips and tricks .....	20
2.4.4	Using use case diagrams to model the library example .....	20
2.4.5	Realisation of use case diagrams .....	21
2.5	Data Flow Diagrams (DFD).....	23
2.5.1	Data Flow Diagrams for system evolution.....	23
2.5.2	The elements of data flow diagrams.....	24
2.5.3	Tips and Tricks.....	25
2.5.4	Rediscovering data flow diagrams in the legacy system.....	26
2.5.5	Using data flow diagrams to model the library example.....	27
2.6	Extended Entity Relationship Diagrams (EER) .....	29
2.6.1	Entity Relationship Diagrams for system evolution.....	29
2.6.2	The elements of entity relationship diagrams.....	30
2.6.3	Tips and Tricks.....	31
2.6.4	Rediscovering entity relationship diagrams in the legacy system.....	32
2.6.5	Using an entity relationship diagram to model the library example .....	33
2.6.6	Using UML class diagrams for data modelling.....	33
2.7	Block Diagrams.....	34
2.7.1	Block Diagrams for system evolution .....	34
2.7.2	The elements of block diagram .....	35

---

2.7.3	Logical and physical block diagrams .....	37
2.7.4	Tips and Tricks .....	38
2.7.5	Using a block diagram to model the library example.....	40
<b>2.8</b>	<b>Hardware/Network Diagrams.....</b>	<b>41</b>
2.8.1	Hardware/network diagrams for system evolution.....	41
2.8.2	The elements of hardware/network diagrams.....	42
2.8.3	Tips and Tricks .....	43
2.8.4	Using a hardware/network diagram to model the library example .....	44
<b>2.9</b>	<b>Traceability of Context Models.....</b>	<b>45</b>
2.9.1	Tracing context models – Inter-viewpoint traceability .....	45
2.9.2	Going from context models to technical models.....	48
2.9.3	From legacy systems to evolutionary systems .....	49
<b>3</b>	<b>TECHNICAL MODELLING .....</b>	<b>51</b>
<b>3.1</b>	<b>Introduction .....</b>	<b>52</b>
<b>3.2</b>	<b>Technical Modelling of 3GL applications .....</b>	<b>52</b>
3.2.1	The properties of 3GL applications.....	53
3.2.2	Special notations/diagrams to model 3GL applications .....	62
3.2.3	Modelling 3GL applications.....	62
<b>3.3</b>	<b>Technical Modelling of 4GL applications .....</b>	<b>66</b>
3.3.1	The properties of 4GL applications.....	67
3.3.2	Notations to model 4GL applications.....	84
3.3.3	Technical Model of Current System.....	93
3.3.4	Technical Model of Target System .....	94
3.3.5	Reverse-engineering of 4GL Applications.....	95
3.3.6	Guidelines for structuring the Target System.....	97
<b>4</b>	<b>CASE STUDY.....</b>	<b>101</b>
<b>4.1</b>	<b>Description of the example application .....</b>	<b>102</b>
<b>4.2</b>	<b>Context modelling.....</b>	<b>103</b>
<b>4.3</b>	<b>Technical modelling .....</b>	<b>110</b>
<b>5</b>	<b>APPENDIX A: OVERVIEW OF THE UML .....</b>	<b>117</b>
<b>5.1</b>	<b>Introduction .....</b>	<b>118</b>
5.1.1	Modelling with UML Diagrams.....	118
5.1.2	Stereotypes .....	119
<b>5.2</b>	<b>Static structure diagrams.....</b>	<b>119</b>
<b>5.3</b>	<b>Use case diagrams.....</b>	<b>120</b>
<b>5.4</b>	<b>Sequence diagrams .....</b>	<b>121</b>
<b>5.5</b>	<b>Collaboration diagrams .....</b>	<b>122</b>
<b>5.6</b>	<b>State diagrams .....</b>	<b>123</b>

---

---

5.7	Activity diagrams .....	124
5.8	Implementation diagrams.....	125
<b>6</b>	<b>APPENDIX B: DESCRIPTION OF THE LIBRARY EXAMPLE.....</b>	<b>127</b>
6.1	Outline of a Case Study.....	128
6.2	Problem Description.....	128
6.2.1	Assumptions Concerning the Library.....	128
6.2.2	Routine Procedures of the Existing Library .....	129
6.3	Reorganisation of the Library with a View to Computerization .....	130
6.3.1	General Considerations Regarding the Library System .....	130
6.3.2	Book Stock and Accessibility.....	131
6.3.3	Revised Routine Procedures.....	131
<b>7</b>	<b>APPENDIX C: 4GL NOTATIONS.....</b>	<b>133</b>
7.1	Stereotypes used for modelling 4GL applications .....	134
7.2	Alternative notations/diagrams to model 4GL applications.....	137
7.2.1	Modelling 4GL applications with OMT.....	137
7.2.2	Modelling 4GL applications with MD .....	140
7.3	Modelling UNIFACE applications with UML .....	144
7.3.1	UNIFACE properties to be modelled.....	144
7.3.2	Modelling UNIFACE applications with UML.....	150
7.4	How to find UML mappings for not regarded languages.....	154
<b>8</b>	<b>APPENDIX D: DATABASE MODELLING .....</b>	<b>157</b>
8.1	Analysis Model.....	158
8.1.1	Structured analysis model .....	158
8.1.2	Object-oriented analysis model .....	160
8.1.3	Combined analysis model .....	160
8.2	Design Model.....	161
8.2.1	Design activities and database elements.....	161
8.2.2	Notations - Tools .....	164
8.2.3	Comparison and Summary .....	172
<b>9</b>	<b>APPENDIX E: TOOL SUPPORT .....</b>	<b>175</b>
9.1	Tool support.....	176
9.1.1	Operational schemas.....	176
9.1.2	Use case diagrams .....	176
9.1.3	Data flow diagrams.....	177
9.1.4	Entity relationships diagrams .....	177
9.1.5	Block diagrams.....	177
9.1.6	Hardware/network diagrams .....	177

---

**10 APPENDIX E: REFERENCES.....179**

# 1 Introduction

## **Contents**

1.1 Objectives

1.2 Rationale

1.3 Modelling activities during system evolution

1.4 Overview of the document structure

## **Summary**

This chapter introduces the reader to the problems of evolving legacy systems, and gives an introduction to the RENAISSANCE view on modelling architectural aspects of legacy systems in order to evolve them.

The objectives and rationale for the document are presented, as well as a description of how this document concerns from earlier phases of the RENAISSANCE project. Finally, the document structure is explained to the reader.

## 1.1 Objectives

At some point every installed application in an organization's system portfolio must be replaced. The application may not meet current requirements with respect to a number of aspects, e.g. speed, accuracy, usability, portability, integrability, etc.

Some applications in a system portfolio may be isolated and replaced with a COTS<sup>1</sup> application. Some may not be considered useful any more, and their functionality is abandoned. Other applications encapsulate strategic components of the business policy, and are vital to ensure the continued operation of the business. All these systems comprise the organization's heritage of the past; hence they are collectively termed legacy systems.

This report deals with the last of the three categories. Those are the most difficult legacy systems; the organization is severely hampered by their state, but is still strongly dependent upon their functionality. The systems cannot be rapidly replaced, as it is these systems that provide the organization with its business advantage. This functionality may not be available in current replacement applications; on the other hand the functionality provided by the legacy systems is not explicitly captured in a communicative form; source code is typically the only written information available about the system.

The open question for management is then: How can we migrate from an old and expensive to maintain environment, *supported by the legacy system*, to a modern desktop computing environment, *supported by a new system*, while ensuring the business' stability?

RENAISSANCE recognizes this dilemma, and has defined a method to aid the system engineer with identifying an evolution plan to determine how the legacy system can be re-engineered into a modern and efficient system. This report describes a set of techniques for modelling the current *legacy* and the new *target* system. It is only by understanding the existing legacy system, that the system can be efficiently re-engineered into an evolutionary target system.

To make evolution decisions in a re-engineering project, knowledge both about the current and future system must be elicited and explicitly stated. The detailed objectives of this report can therefore be summarized as:

- Provide engineers with guidelines on evolution modelling, both at high and low levels of abstraction.
- Show how different modelling techniques, including the UML, can be used in context modelling (high level) and technical modelling (low level).
- Identify and discuss the properties of third and fourth generation languages (3GLs and 4GLs), and relational database management systems (RDBMSs) and how these influence re-engineering decisions at a detailed level.

---

<sup>1</sup> COTS = Commercial Off The Shelf

---

We will also provide the reader with a list for further reading, and identify tools that ease system modelling for re-engineering.

## 1.2 Rationale

One of the major problems with old systems is that they are difficult to understand since the original system plan has been iterated due to subsequent alterations. The understandability is further exacerbated by the fact that documentation is either lacking or outdated. This has resulted in a system, which is expensive to maintain, hard to adapt, and uses outdated technology.

These problems are typically enforced by the facts that the

- original system developers have left the organization,
- current maintainers do not have sufficient system knowledge, and
- new maintainers may not be skilled in the old technology used.

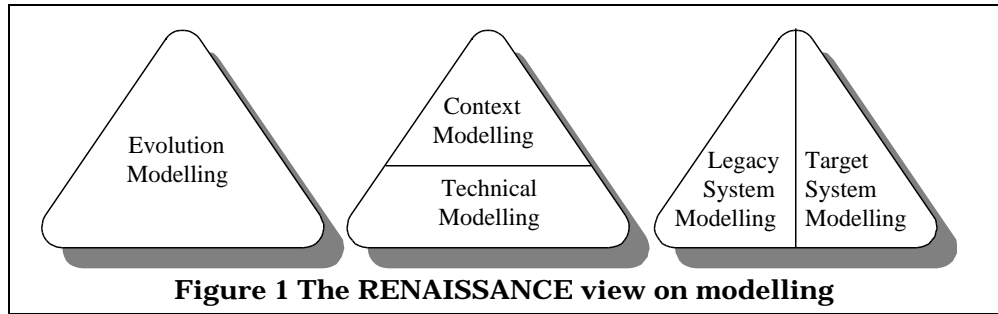
Projects concerned with the evolution of legacy systems have different needs from projects concerned with the development of new systems. In contrast to development projects, whose aims are typically to introduce new computer-supported functionality, re-engineering projects are initiated to improve aspects of existing systems.

Hence requirements for system modelling are different in the context of re-engineering. RENAISSANCE has identified these special needs, and focuses on architectural modelling, rather than functionally oriented modelling which is one of the primary activities in forward engineering. The RENAISSANCE approach to *architectural modelling for evolution* is presented in the subsequent chapters.

## 1.3 Modelling activities during system evolution

Architectural modelling is distinguished by focusing on modelling and documenting existing components and relationships, i.e. the *architecture* or *structure* of the legacy systems. In order to cope with evolution, guidelines must be given for efficient transition from an architectural model of the legacy system to a target system.

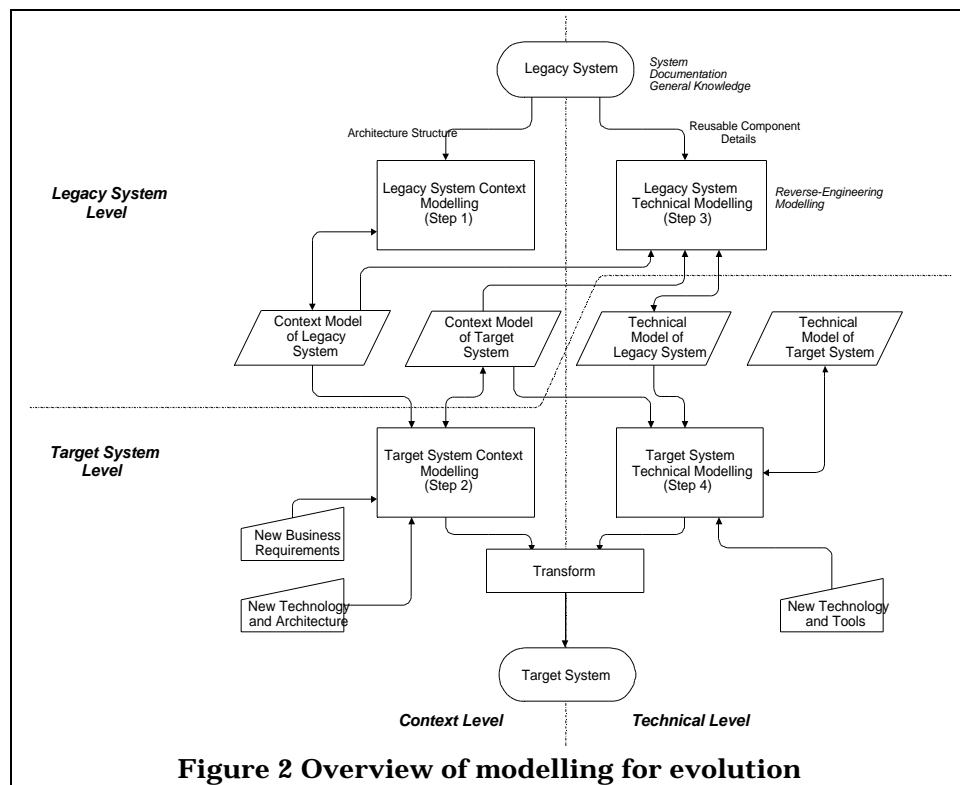
We use the term *context modelling* to refer to modelling of high-level system architecture. Similarly, the term *technical modelling* is used to model the more detailed structure of the system. See Figure 1



To sum up the rationale for this document:

- *Understanding* the architecture of a legacy system is a critical step in deciding which evolution strategies are most appropriate. This report aids the reader in identifying how and what to model in a re-engineering project.
- *Context modelling*. It is often hard for people unfamiliar with the system domain to understand low level (technical) models of a legacy system. Context modelling is used to build a visual framework for discussion of the legacy system.
- *Technical modelling*: Needed to abstract the relations between software components. Hard to read (legacy) code. Technical modelling of the legacy systems complement the context models with detailed information, and is a mandatory step before migrating the system to new technology.

The process of *architectural modelling for evolution* is an incremental and iterative four-stage process, which gradually transform a legacy system into a target system. The process is summarized in Figure 2.



**Figure 2 Overview of modelling for evolution**

The process can be elaborated as follows:

- **Step 1 Legacy System Context Modelling:** The input to this step is the source code for the system to be re-engineered, any system documentation or other documented knowledge about the system. The re-engineer uses a set of design notations to document the context model of the legacy system. The re-engineer iteratively completes the context model, continuously consulting any experts on the legacy system to gain consensus for the emerging model.
- **Step 2 Target System Context Modelling:** When the legacy context model is agreed to cover the interesting aspects of the system to be re-engineered, the context model is augmented with new constraints regarding new business requirements and new technology. This results in a context model of the target system.
- **Step 3 Legacy System Technical Modelling:** While the context model efficiently communicates *what* is being done at a managerial level, the legacy technical model focuses on *how* this functionality is achieved. This activity can be supported by special reverse-engineering tools, which are able to automatically extract information needed from the legacy system code. Examples of new technology and architecture at this level can be window GUI and client/server. The technical modelling uses the context models as a starting point. That is, the context models described in chapter 2 will be the input to the first step in the design.
- **Step 4 Target System Technical Modelling:** A detailed model for the target system is devised, based on the target context model and legacy technical model. Additional constraints are added due to new technological requirements and available tools which are capable to

provide this new technology. Examples of such may be component-based development and COM/DCOM.

The context and technical models of the target system guide the transformation of the legacy system to the target system. The transformation process is not detailed in this report.

The next section gives an overview of the structure of the rest of this report.

## 1.4 Overview of the document structure

- Chapter 2: Context Modelling. The objectives and content of a context model is described, and an assessment method for verifying that a correct level of detail is reached is described. A set of design notations that should be used in the context model are described. For each of the notations, we describe how information can be rediscovered from the legacy system and documented using the notations. Each notation is accompanied with two examples, a general one, and one that describes a particular aspect of an example outlined in Appendix B.
- Chapter 3: Technical Modelling. Aspects that should be considered when documenting the technical models of the legacy and target systems are discussed. Special considerations for how to deal with problems when providing technical models of both 3GL and 4GL applications.
- Chapter 4: 4GL Case Study. This case study shows how context and technical models are constructed for an actual legacy system developed in a 4GL, and how context and technical models are devised for the target application that eventually will replace the legacy application.
- Appendix A: Overview of the UML. For the reader which is unfamiliar with the Unified Modelling Language, a rough description of the different UML diagram types is given.
- Appendix B: Description of the Library Example. This appendix describes a reference example, which is used throughout the report for explanatory and illustrative purposes.
- Appendix C: 4GL Notations. This appendix goes into further detail in how to model 4GL applications for evolution.
- Appendix D: Database Modelling. A pragmatic collection of modelling techniques for database evolution and development is presented.
- Appendix E: Tool Support. Contact info is given for a number of tools supporting the re-engineer in devising diagrams used for architectural modelling.
- Appendix F: References. Contains a list to further related reading.

## 2 Context Modelling

### Contents

- 2.1 Introduction
- 2.2 Objective and Content of Context Models
- 2.3 Operational Schemas
- 2.4 Use Case Diagrams
- 2.5 Data Flow Diagrams (DFD)
- 2.6 Extended Entity Relationship Diagrams (EER)
- 2.7 Block Diagrams
- 2.8 Hardware/Network Diagrams
- 2.9 Traceability of Context Models

### Summary

This chapter describes several modelling techniques to model the context of both the current and the target system.

The RENAISSANCE project has selected six different modelling techniques that describe different and overlapping aspects of a system context. These techniques are operational schemas, use-case diagrams, data flow diagrams, extended entity relationship diagrams, block diagrams and hardware/network diagrams.

## 2.1 Introduction

When designers develop models of an existing or a future application system, they need to take into consideration not only the technical details of the system, but also how the system interacts with the surrounding environment.

*Context modelling*, also known as conceptual modelling, is defined by (Krogstie and Sølvsberg, 1997) as:

*“A model of the phenomena in a domain at some level of approximation, which is expressed in a semi-formal or formal language”*

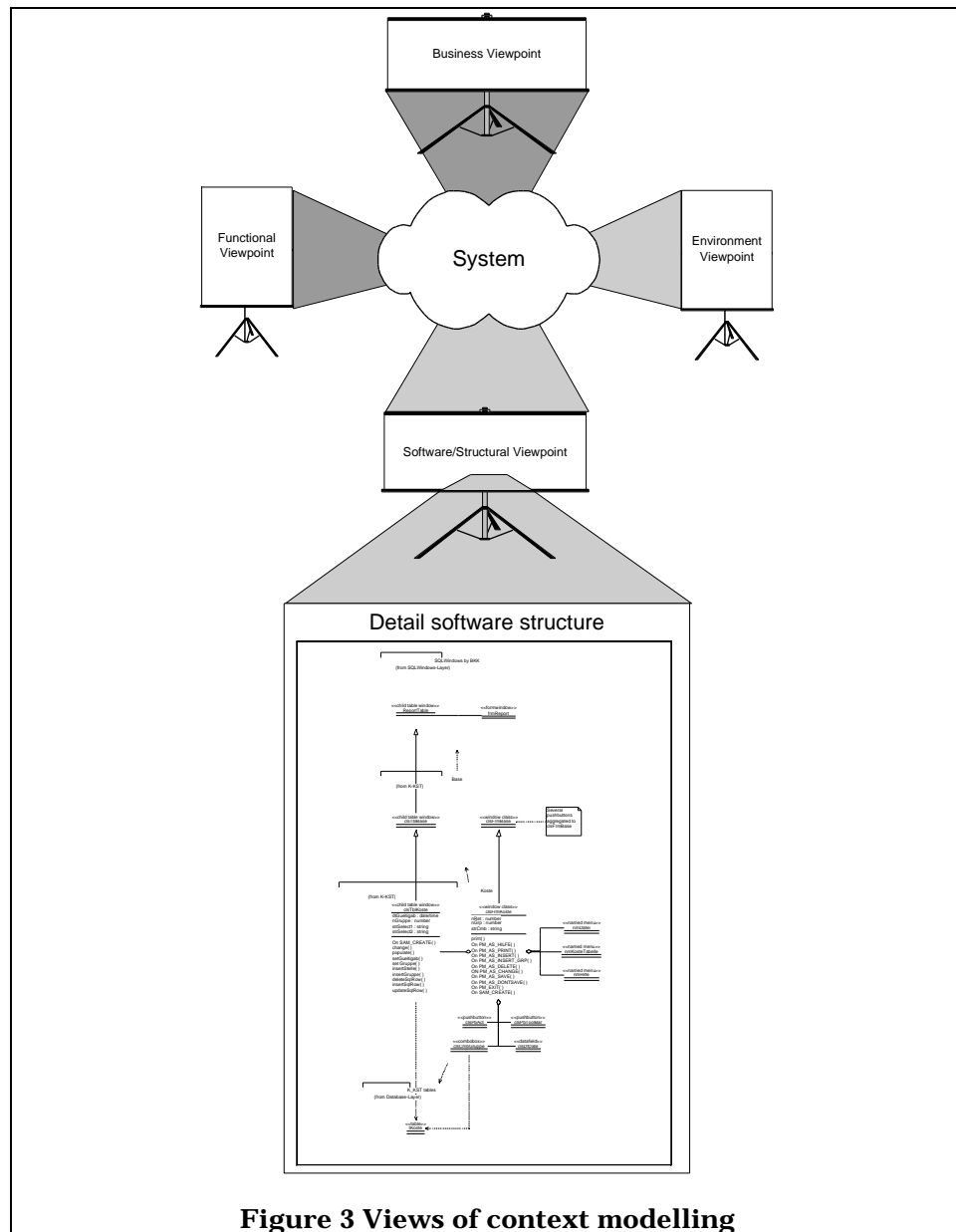
The models produced using the notations provided for context models provide a wealth of information to the users and designers. The following reasons are used for selecting the notation techniques:

- They provide an explicit representation of the system and its requirements, and act as a vehicle for communication about the system among analysts, designers and users.
- They form a basis for the design and implementation of new systems, as well as a basis for detailed modelling of existing *legacy systems* (i.e. current system) for re-engineering.
- The context models document the application system at a high level. Since the context model is less detailed than the design models, the user of the context model can more easily understand the context models before considering the more detailed designs.
- They should be well established in the market, and good tool and training support should be available.
- Together they should cover all modelling dimensions described below. If one dimension may be covered by several notation techniques an individual selection by the user is possible.
- And last, but not at least, context models provide good and consistent documentation of both the current system and the target system to be.

Several modelling dimensions or modelling aspects must be supported when modelling the current system and/or the target system on an abstract level. We have identified the following modelling dimensions or viewpoints as the most important ones for system *evolution* and *maintenance* (Figure 3 illustrates these):

- The **Business Viewpoint** covers the current and target view of the **Business Process**. This viewpoint is used when modelling the current and/or target business processes and focuses on the processes that are already supported or need to be supported. The modelling of the supported (current or to be) business process is a central dimension, which must be supported by context modelling.
- The **Functional Viewpoint** looks at the system from a functional view covering the **functionality** of the system. This viewpoint is used to discover and/or model the system's functionality and behaviour. Understanding the current system's overall functionality and agreeing upon the functionality of the target system, are both very important issues in the context modelling process.

- The **Software/Structural Viewpoint** contains the **software structure** and **data structure** of the system. It is this viewpoint that will lead to the technical model of the system. To understand the overall **software structure** is important to plan any evolution or maintenance activity. Modelling the current and/or the target **data structures** is also very important. Together with the software structure it will describe the overall system structure and complete the architectural software description of the system.



**Figure 3 Views of context modelling**

- The **Environment Viewpoint** sees the system from its environment (current or target). This covers the **communication** and **hardware** dimension of the system under consideration. This viewpoint is used to model and understand the environment the system exists in and/or

will be implemented in. We here refer to **communication** as a dimension for context modelling which is used to model and understand the external communication of a system with its environment is important, as well as the communication and file access of the different processes inside the system. Modelling of **hardware** components and their interaction gives an overview of the physical structure of the system.

Any system model is organized into layers, where models in layer  $n$  is an elaboration of issues described more abstractly in a higher level. This also goes for **Context Models**. However, the fundamental idea behind context models is that they should be at an abstract level and usable for decision-makers and non-technical participants. All other modelling processes, like context modelling involves an **iterative**, step by step process. That is, the models should be achieved through an iterative modelling process where the models are refined by making them more and more detailed. The process should be stopped when more iteration doesn't pay off. It is often very difficult to know when to stop, but have in mind that the last 20% of the refinements to get a "perfect" model will cost you more than the first 80% (Pareto rule).

When modelling context models also have in mind that they will be the starting points for the technical models to be used in the design phase (described in chapter 3). This goes specially for models describing the **software structure** and the **data structure** (see below). Remember that the models describing the software and data structure at context level should be very abstract and easy to understand.

In this chapter, we describe several techniques for modelling the different views and dimensions of an application. In particular, we describe the following techniques:

- Section 2.3 describes a technique for modelling business processes, called **operational schemas**. Business processes are modelled as a sequence of *activities* that must be completed by *roles*.
- Section 2.4 describes how to use **use-case diagrams** to model the communication among *actors* and *functions* in the system.
- Section 2.5 provides a discussion of **data flow diagrams**. Such diagrams are used to model the information flow among processes, data stores and external agents.
- Section 2.6 describes **extended entity relationships diagrams**. (E)ER diagrams are used to document the data structures in the existing application, and form a base for transforming these structures to the re-engineered application.
- Section 2.7 describes **block diagrams** for documenting the relationships among high-level software components in a legacy system (i.e. current system).
- Finally, Section 2.8 describes **hardware/network diagrams** which are used to document the composition of hardware and network used by an application, and the mapping of software components onto this hardware.

Table 1 gives a summary of the modelling dimensions that the different diagramming techniques support.

Diagram Type	Business Viewpoint	Functionality Viewpoint	Structural Viewpoint	Environment Viewpoint
<b>UML diagrams</b>				
Use case	✓	✓		
Sequence	✓	✓	✓	
Collaboration	✓	✓	✓	
Package			✓	
Deployment				✓
<b>Further diagrams</b>				
Operational Schema	✓	✓		
Data Flow		✓		
Entity Relationship			✓	
Block	✓	✓	✓	✓
Hardware/Network				✓

**Table 1 Comparing context views and diagram types.**

We illustrate each technique with two examples: One for explaining the notations used in the diagrams, and one which uses the diagramming technique to model the library example system described in Appendix B. In addition chapter 4 will guide you through a more extensive case-study which describes the process of re-engineering a legacy 4GL system using both *context modelling* and *technical modelling*. The description of all techniques is annotated with tips for how to use it efficiently. Finally, we give the reader pointers to tools that support the modelling techniques described in this chapter.

## 2.2 Objective and Content of Context Models

The overall intention of a context model is to describe the system at an abstract level, with enough information to make informed decisions about an evolution strategy. The content of a context model is dependent on the concrete project situation.

Information should be presented without much detail. It is important to identify the main system concepts, components, and relationships. The context model of target system may be more detailed than that of the current system, since the target system must be realised, while the legacy system only has to be understood. Thus, requirements for modelling are stronger and more specific for the target system.

Table 2 summarises the objectives and content of context models for the current and target system.

	Current System	Target System
Objective	High level documentation	
	Overview of the supported business activities	
	Base for evolution planning and choosing evolution strategy	
	Base for system assessment as a whole and parts of it	Base for selecting the best target system if there are several alternatives
	Identify reusable parts	Base for project planning and technical modelling
	Identify areas for reverse-engineering	
Content	1) Functional overview	
	2) Overview of the supported business processes	
	3) Architectural overview	
	4) Overview of data, and perhaps the structure of the data	
	5) Overview of external interfaces	
	6) Overview of internal processing structure and file access	
	7) Overview of the operation environment, e.g., hardware, network, operating system.	

**Table 2 Context modelling objectives and content**

## 2.3 Operational Schemas

### 2.3.1 Operational Schemas for system evolution

An IT system fulfils business needs by supporting or automating part of them. This IT system is related to human operations and interacts with a technical environment. Operational modelling is related to the modelling of these business needs (described as *business processes*). The scope of these business processes can be very large and cover a whole organisation. As a consequence, and as opposed to other activities of context modelling, the scope of operational schemas extends beyond the scope of the IT system. They cover automated processes as well as manual processes and they make a distinction between internal and external agents.

To avoid a common misunderstanding, business process modelling, addressed by operational schemas, is not directly related to Business Process Re-engineering (BPR), since BPR deals with modifying business processes whereas business process modelling deals with documenting business processes.

In the context of system evolution, operational modelling helps in understanding an existing IT system and the place and role it plays in an overall organisation. This understanding can be the basis for deciding how a system must evolve. Indeed, the value of automating or supporting

business processes or part of them can be analysed and this can ensure that the IT system brings the best value for money.









In the context of legacy systems, implemented as separate programs supporting business functions, operational schemas as well suited to position these existing programs in the context of the overall activity of the organisation.

### 2.3.2 The elements of operational schemas

*Operational schemas* are models of business processes defined as a sequence of *activities*, which must be completed by *actors* playing specific *roles* and interacting with the environment. An operational schema is defined for every major business function by listing the various steps composing it (manual or IS supported) and their interactions with actors or the technical environment.

An operational schema is described by a table where rows correspond to every step of the business process and columns correspond to actors or roles. The process flow between steps is represented as arrows. The description of a step is given in the first column. Elements describing the steps are represented in the columns corresponding to the actor or role performing it. The last column corresponds to the external business process or environment of the business process (See also Figure 4 for an example of operational schema).

Elements composing the description of a step are shown in Table 3.

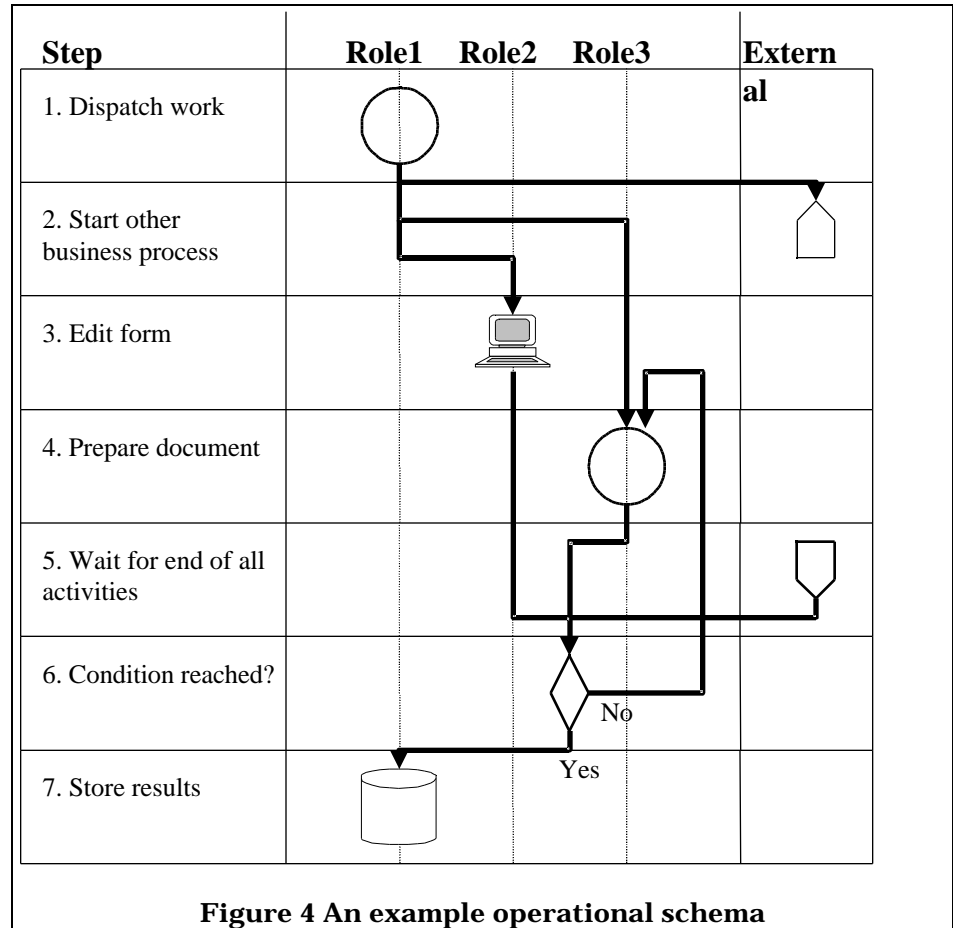
Element	Notation	Usage
Activity	 IS Supported  Manual	Denotes a step in the business process. An activity is performed by an actor of a specified role (the role corresponding to the column where the activity is drawn) and can be either manual or supported by the Information System.
Database		Denotes information stored or retrieved from a database which is needed or generated by an activity.
Activity Flow		Denotes a flow between activities. An activity pointed by the arrow can start when the activity at the beginning of the arrow is completed.
Alternative		Denotes an alternative (yes/no). The flows continues according to the result of this alternative.
Hardcopy		Denotes an interaction (or dependency) with paper based input/output.
Link to external business process		Denotes an interaction with an external business process (input). Input from the external process (modelled using another operational schema) is needed to perform one step of the business process.
Link from external business process		Denotes an interaction with an external business process (output). Output to the external process (modelled using another operational schema) is generated.

**Table 3 Elements of operational schemas**

### 2.3.2.1 Example

Figure 4 represents a typical operational schema. This example business process involves 3 types of actors ("role1", "role2", "role3"), interacts with an external business process ("external") and consists of 7 steps:

1. *Dispatch Work*: from this manual operation, work is dispatched to an external process, an automated activity ("edit form") and a manual activity ("prepare document")
2. *Start Other Business Process*
3. *Edit Form*
4. *Prepare document*
5. *Wait for End of All Activities*: when all activities launched are completed, the business process can continue
6. *Condition Reached*: the business process continues according to the result of a test (proceed if test is positive, redo again the "prepare document" activity if the test is negative).
7. *Store Results*: the combined result is stored in a database



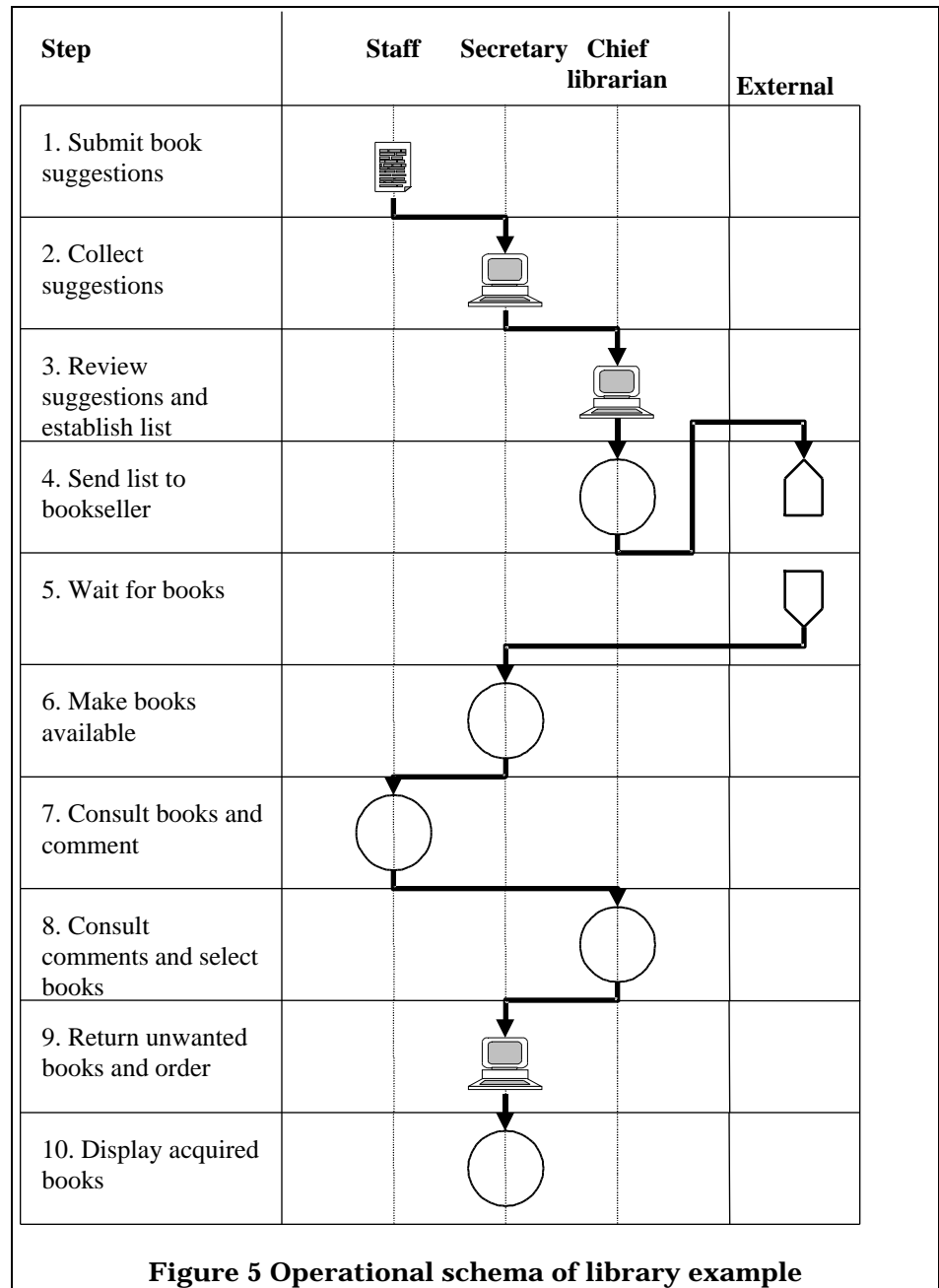
### 2.3.3 Using operational schemas for modelling the library case

Figure 5 presents an operational schema for the “Acquisition” business process of the Library Case.

This business process involves 3 types of actors, “staff”, “secretary” and “chief librarian”. This business process is sequential and involves all actors for manual and automated activities.

Suggestions are managed by an application. Therefore, activities “collect suggestions” and “review suggestions and establish list” are represented as automated activities.

Since the order of books is sent and processed by the (external) bookseller, this business process is considered as external and represented as such on the diagram.



### 2.3.4 Tips and tricks

- Listing all current activities by actor can be a good starting point to find existing business processes. Brown Paper techniques (interactive creation workshop where ideas are posted and organised on a large "brown paper") can be used to find common activity flows and therefore business processes.
- Do not limit the description of business processes to those supported by an IT system. This can be a good opportunity to extend its scope.

- 
- Existing IT systems programs can also be a good starting point to identify parts (automated ones) of existing business processes.
  - To limit the complexity of business processes, the number of roles should be limited (5 to 7 roles at most) as well as the number of steps. Separate business processes interacting between each other can be defined instead.
  - An activity can itself be described as a business process. This allows business processes to be decomposed to make their description simpler.
  - An activity can be described as a use case. This allows the separation of the description of business processes at a high-level and at a more detailed level.

### 2.3.5 Rediscovering Operational schemas in the legacy system

Although operational schemas do not describe the internals of potentially re-engineerable elements composing a legacy system, they help understanding where these elements fit in the legacy system, and especially:

- The roles of these elements in business processes. In particular, operational schemas can support the selection of parts of a legacy system to re-engineer, keep or drop, depending on their importance in the business processes.
- The interactions between these elements
- The interactions between elements and their environment (other systems, actors...)

## 2.4 Use Case Diagrams

Use case diagrams play an important role in system evolution projects. They are an efficient tool to model how different actors communicate with the system that will be evolved.

This section describes three kinds of diagrams:

1. **Use case diagrams.** These are used to model the boundaries (i.e. business process) in a system.
2. **Sequence diagrams** trace the execution of an interaction in time.
3. **Collaboration diagrams**, which are interaction diagrams that show the sequence of messages that implement an operation or a transaction.

Sequence and collaboration diagrams can also be said to describe the realisation of use cases.

### 2.4.1 Use Case Diagrams for system evolution

Use case diagrams were introduced by Jacobson (Jacobson, 1992), and were the core concept in his object-oriented software engineering method. Used in the way Jacobson describes, use case diagrams form the abstract

part of a software project, and provide views to other models which detail the information described by the use cases. In RENAISSANCE, we do not relate the use case diagrams directly to other diagrams, but rather apply the use case diagrams as a particular view of the system context.

A use case diagram is used to show the relationships among actors and use cases within a system. Components in a use case diagram are actors, use cases and relationships between them. The relationships model communication (participation) associations between actors and the use cases. Use case diagrams are an aid to defining what exists outside the system (actors) and what should be performed by the system (use cases).

The actors represent what interacts with the system. They represent everything that needs to exchange information with the system. We differentiate between actors and users. The user is the actual person who uses the system, whereas an actor represents a certain role that a user can play.

An instance of an actor (the user) does a number of different operations to the system. When a user uses the system, she or he will perform a behaviourally related sequence of transactions in a dialogue with the system. This special sequence is what is called a use case. A use case is a complete flow in the system, and hence use case diagrams are different from data flow diagrams as the former focus on modelling the internal flow in the system. Neither do use cases model data flow, but rather interactions among the actors and the use cases.

#### 2.4.2 The elements of use case diagrams

Use case diagrams are simple in nature, and are drawn using a small number of elements. These are listed in Table 4.




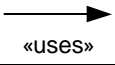
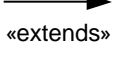
Element	Notation	Usage
Actor		The actor is used to model the roles which communicates with a use case
Use case		A use case describes a particular instance of a system function
Communication link		The communication link indicates that an actor interacts with the system. If the interaction is in a particular direction, an arrowhead may be put on the link
Uses link		Used to indicate that one use case uses the behaviour specified by the other
Extends link		The extension link indicates that one use case may include the behaviour specified by the other use case.

Table 4 Elements of use case diagrams

### 2.4.2.1 An example

The specification of use case external behaviour defines the possible sequences of messages exchanged among the actors and the system.

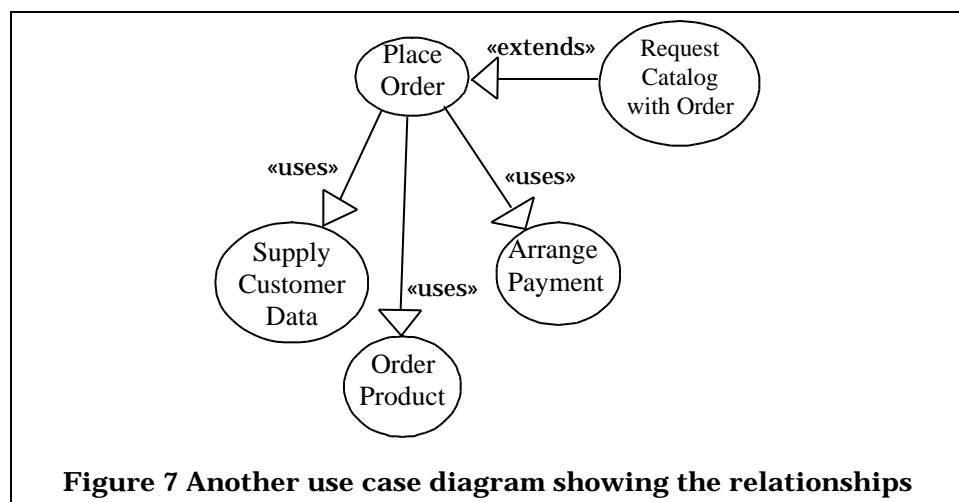
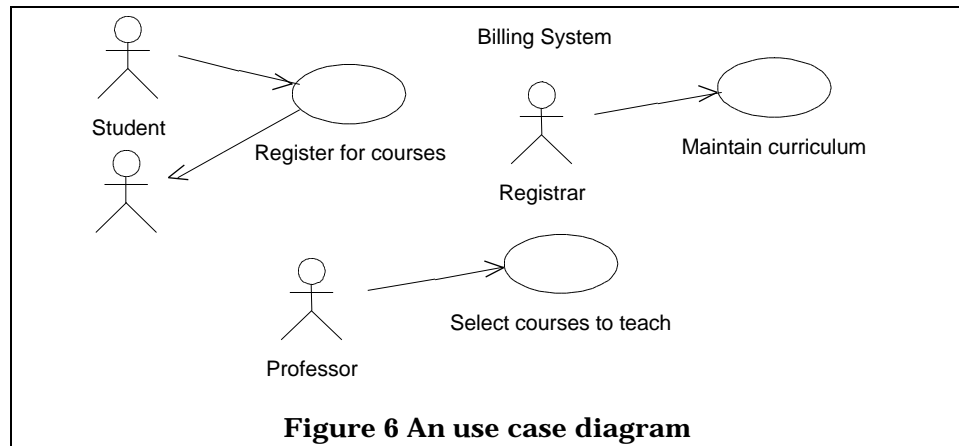


Figure 6 shows different *Actors* drawn as stick men and *Use Cases* drawn as circles and the relationships between them.

There are three meaningful relationships within a use case diagram as shown in Figure 6 and Figure 7:

- **Communicates:** Shown by connecting the actor symbol to the use case symbol by a solid path. The actor to «communicates» with the use case.
- **Extends:** Shown by a generalisation arrow from the use case providing the extension to the base use case. The arrow is labelled with the stereotype «extends». Extends relationships are used to indicate that an instance of use case may include the behaviour specified by another.
- **Uses:** Shown by a generalisation arrow from the use case doing the use to the use case being used. The arrow is labelled with the stereotype «uses». Uses relationships are used indicate than an instance of one use case will also include the behaviour of another.

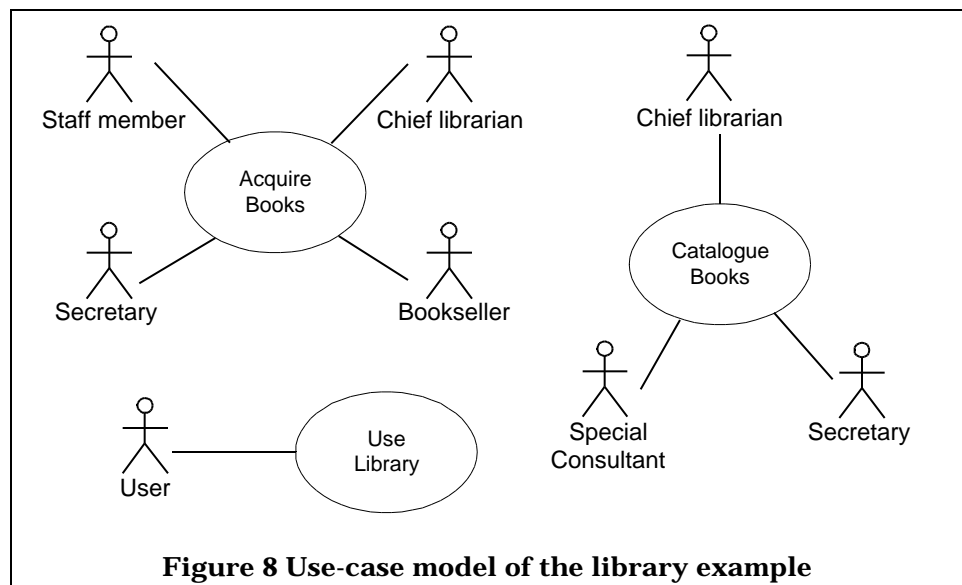
### 2.4.3 Tips and tricks

The following tips are borrowed from Scott Ambler (Ambler, 1995):

- *Create scenarios that the system should and shouldn't be able to handle:* Remember, good user requirements definition describes both what is in and what is out of scope. This means that not only do you want to identify scenarios the system should be able to handle, but also ones that it shouldn't. Identifying what the system won't do helps to prevent "creeping featuritis."
- *Explore business rules:* If your users have told you about a business rule, create a scenario for it. For example, if students can take a maximum of five courses, create a scenario for someone trying to enrol in six. By doing this, you'll bring into question the validity of existing business rules that may or may not make sense anymore. Perhaps the reason why people are only allowed to enrol in five courses per term is because that's how much room was on the paper form when it was originally designed in the sixties. Do we still want our system to conform to this business rule?
- *Transcribe the scenarios immediately:* Once a scenario is identified, record it.

### 2.4.4 Using use case diagrams to model the library example

In Figure 8, we model a high-level view of the library example with a use-case diagram. The example consists of three use cases, "acquire books", "catalogue books", and "use library". We see that different actors play roles in several use cases. The actor is not bound to a particular person, i.e. "staff member" is typically a role which several persons can have, while the chief librarian is one person at a time.



---

## 2.4.5 Realisation of use case diagrams

As stated above, use case diagrams merely covers the boundaries in a system. The use cases are realised by using **sequence diagrams** and/or **collaboration diagrams**. Or, in another way: While the use case diagram identifies the *scenarios* in the system, the sequence diagrams and collaboration diagrams are used to represent them. A *scenario* is merely an instance of a use case.

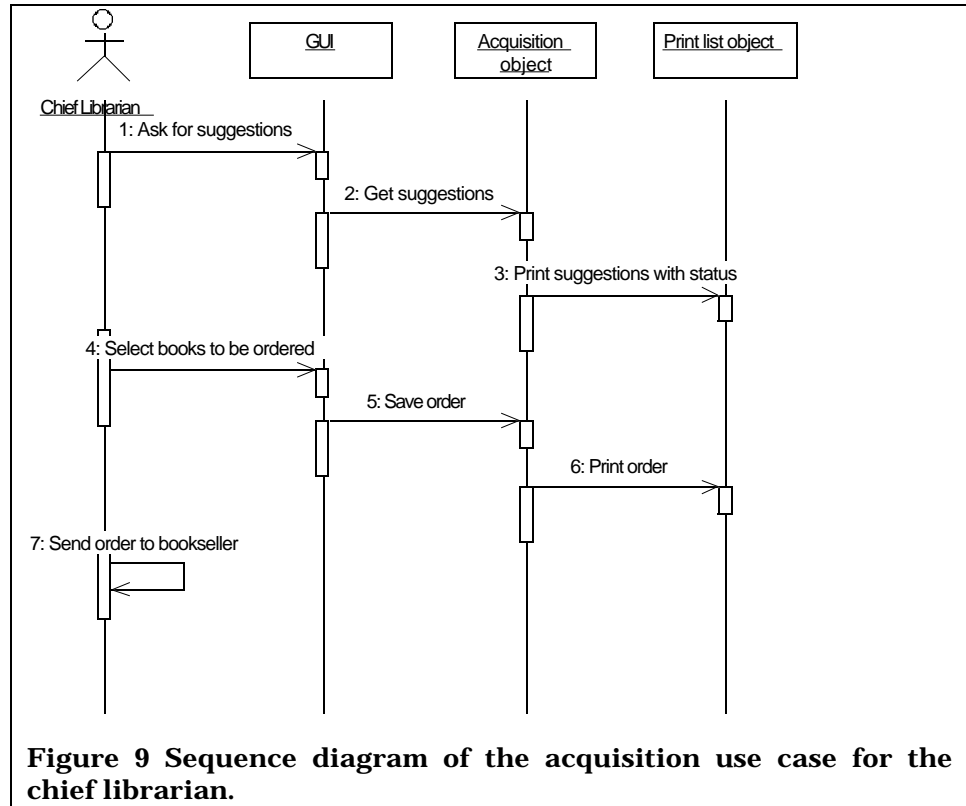
**NOTE:** Sequence diagrams and collaboration diagrams are often **not** needed in the high-level context diagrams. They tend to be detailed and perhaps unsuitable nor needed for assessing the current system. Often one will manage using only the higher-level use case diagrams. However, sequence diagrams and collaboration diagrams are very useful to fill the traditional gap between the context models and the technical models. By applying these techniques and diagrams, one will identify objects (and though classes) which are or, have to be present, in the system.

### 2.4.5.1 Sequence diagrams

This type of diagram and technique uses actors, objects and messages as building blocks. In other literature these diagrams are also referred to as *Event Trace Diagrams*. It is usually used in the traditional analysis phase. Each message, indicated by arrows, is a request for a service from an actor to an object or between objects (or actors). This technique can be used to identify objects in the system, but one should have some basic ideas of candidate object before starting on this technique to avoid too many iterations.

Figure 9 below shows how a sequence diagram may be used to model the *Acquisition use case* seen from the viewpoint of the *Chief Librarian*. The scenario for *selection and purchase of new books* are as follows:

- Suggestions for new acquisitions are now entered directly into the system giving all known details of the book including comments, if desired,
- book selection is based on a suggestion list printed out by the system, displaying all available information and categorizing the suggestions as genuinely new, previously suggested but not purchased, new edition of book already in stock, and book already in stock,
- after completion of the system entries for books to be ordered, a system-generated list of orders on approval is sent by way of an order to the bookseller,
- the purchase of a book is based on a system-generated list of orders for purchase which is sent to the bookseller and the finance department.



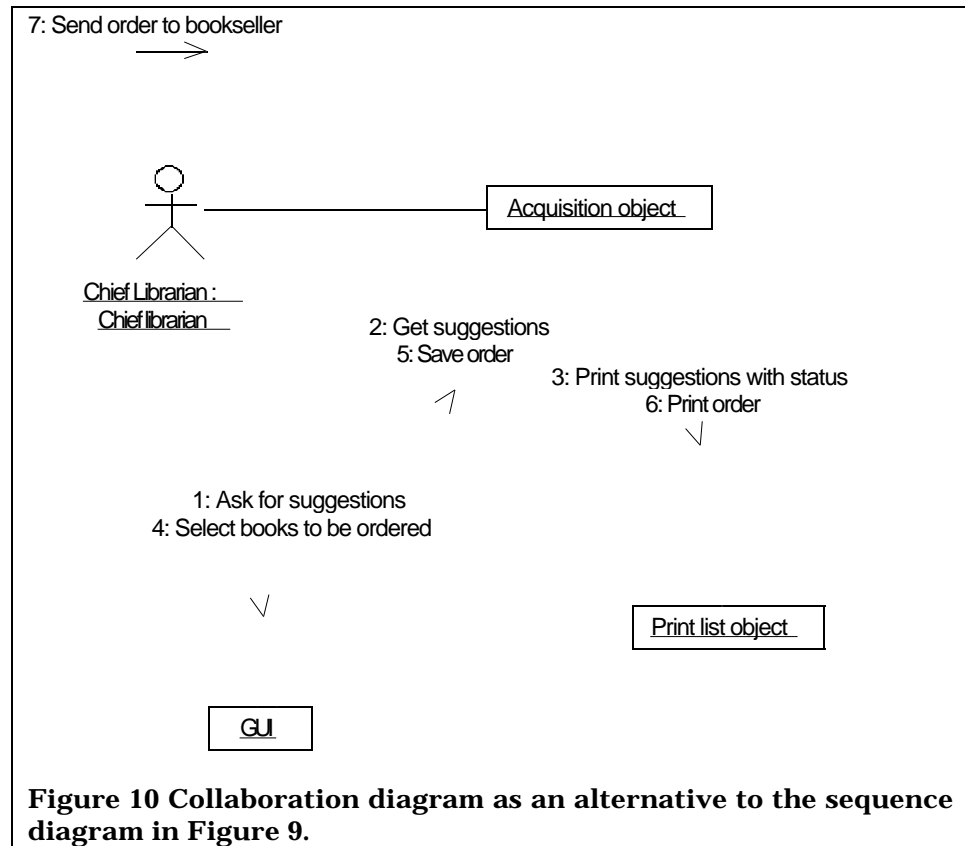
Similar sequence diagrams can be made for all the use cases in the system.

#### 2.4.5.2 Collaboration diagrams

A collaboration diagram can provide another realisation of a use case. A collaboration diagram also called *object interaction diagrams*, is an interaction diagram that shows the sequence of messages that implement an operation or a transaction. A Collaboration Diagram shows objects, their links, and their messages. Collaboration Diagrams can also contain simple class instances. Each Collaboration Diagram provides a view of the interactions or structural relationships that occur between objects and object-like entities in the current model.

Collaboration diagrams illustrate the flow of messages between objects, using message sequence numbers to show explicit ordering. Sequence diagrams also illustrate this flow, but use vertical position to indicate time order. In Rational Rose (a UML toolset) an interaction's collaboration diagram and sequence diagrams are isomorphic: one can be automatically generated from the other, and changes to one are automatically reflected in its counterpart.

Figure 10 below shows the collaboration diagram of the use case described by the sequence diagram in Figure 9 above.



## 2.5 Data Flow Diagrams (DFD)

### 2.5.1 Data Flow Diagrams for system evolution

Data Flow Diagrams (DFD) play an important role in system evolution projects. Most systems provide a number of complex functionality to their users. DFDs are used to model the main functionality of a system and the data flows between them. Typically they are used for:

- ✓ an overview about the functionality of the existing legacy system,
- ✓ an overview about the functionality of the desired target system,
- ✓ to discuss the pros and cons of different functional decompositions,
- ✓ to identify functionality's which can be reused in the new system,
- ✓ to plan the steps of an iterative realisation approach,
- ✓ to define and describe the borders of the existing and the new system,

✓ to describe which process is manipulating which file.





Originally, data flow diagrams were introduced in Structured Analysis (SA). The Data Flow Diagram defines the functionality's and ties them together with the help of the information or material flows. Each functionality can be refined into a low-level diagram that shows that functional at a greater level of detail. In this way, the DFDs build a hierarchical structure. The diagrams are complemented by short textual descriptions (mini specification) and a data dictionary. For context modelling no complex hierarchies will be build. Here they are used to provide overviews with only a few diagrams.

Data Flow Diagrams can be drawn in two ways:

- with the formal notation provided by SA for detailed technical discussions, and
- with free graphics for high level presentations.

## 2.5.2 The elements of data flow diagrams

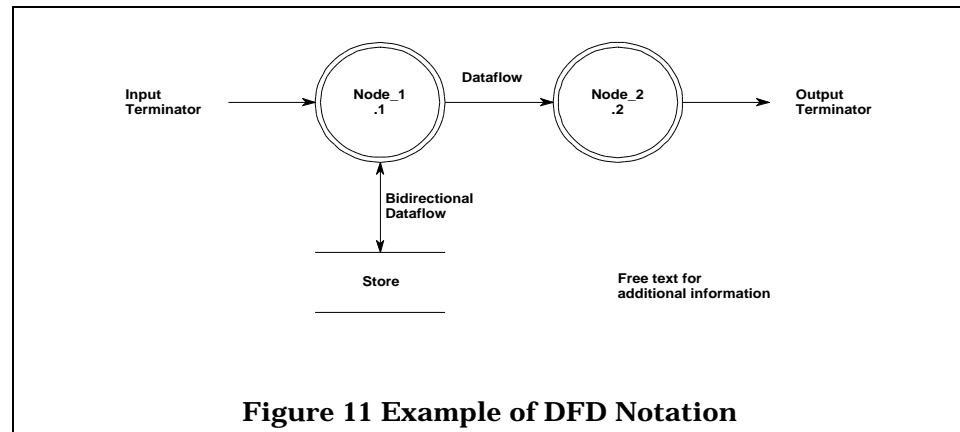
Data flow diagrams consist of the elements described in Table 5. The used notation and terminology varies in detail. We used the notation originally introduced by DeMarco, which is supported by several CASE-tools.

Element	Notation	Usage
Node (Process)		Represents the processes or main functionality of a system.
Data flow		Represents the information flow between these processes. The data flow can be either uni- or bi-directional.
Data store		Is used to store information.
Terminator (External agent)		Is source or receiver of information outside the modelled system.
Free text	ABC	Is used to describe further important aspects.

**Table 5 Graphical elements of Data Flow Diagrams**

### 2.5.2.1 An example

The following example diagram shows how the elements described above can be used to build a data flow diagram.



The data flow diagram in Figure 11 shows two processes. The first process has the process name 'Node\_1' and the unique number '1'. The second process has the process name 'Node\_2' and the unique number '2'. If several diagrams together build a hierarchy, a hierarchical naming schema will be computed and used. The two processes are connected by a unidirectional data flow from 'Node\_1' to 'Node\_2'. The data flow is described by the label 'Data flow'. The example system has one data store named 'Store'. This store is used by the process 'Node\_1' by a bi-directional data flow labelled 'Bi-directional Data flow'. Bi-directional means that 'Node\_1' can store data in the data store and also read data out of the data store. The system is communicating with two terminators (external agents). The first terminator is named 'Input Terminator' and provides data flowing into the system. The second terminator is labelled 'Output Terminator' and receives data from the system.

### 2.5.3 Tips and Tricks

#### General tips

- Use data flow diagrams to visualise the main functionality of the system.
- Use data flow diagrams to define the exact borders of a system and to describe the interfaces.
- Use data flow diagrams in addition to classical textual descriptions.
- Discuss the data flow diagrams with other persons and groups.
- Involve users from all user groups to discover all possible uses of legacy systems.
- The diagrams should not be too complex ( 7+/- 2 nodes)
- The diagrams should be readable.
- If you have a diagram hierarchy it should be balanced.
- Use CASE tools to create and analyse data flow diagrams.

#### Data flows

- Data flows are pipelines used for information packages with known structure.

- For context modelling and analysis it can be information flows, material flows or energy flows. To show the context and interaction is more important than the kind of flow.
- Data flows should have meaningful names. The terminology of the problem space should be used. Beside the data also known properties should be used for naming, i.e. 'checked number' instead of only 'number'.
- Data flows between a process and a store can be nameless.
- Data flows should not be used to model control flows and functional dependencies.
- Data flows between an terminator (external agent) and a process does not mimic sequencing of flows. I.e. the data flow drawn at the top need not to be the first flow.
- Complex data structures should be refined in the data dictionary.
- The overall structure of the underlying data is best described with entity relationship diagrams.

#### **Processes (nodes)**

- Processes are the transformation of input flows into output flows.
- Processes should have a short, characteristic name. Often a verb plus noun is a good solution.
- Processes can be described in related textual mini specifications.
- The number of input and output flows should not be too high.
- A process should not have only input or only output flows.
- Input and output flows should be different.
- Multi-fan output from processes does not control the sequence of the dependent processes.

#### **Data stores**

- Data stores are used to store data.
- Data flows are dynamic, data stores are static or permanent.

#### **Terminators**

- Terminators are entities (persons, organisations, computers, ...) outside the system which interact with the system.
- Terminators are only used in the top-level diagram (context diagram).
- What the terminators do with the data is not part of the system.

### **2.5.4 Rediscovering data flow diagrams in the legacy system**

The main objective of data flow diagrams during context modelling is to provide a functional overview of the current and the target system. These functional overviews are used to identify reusable parts of the current system and to discuss alternatives for the target system.

---

An important question is now, how we can rediscover the functionality in a legacy system? As context modelling is abstract high-level modelling reverse-engineering techniques are not appropriate in most cases. The input comes from (end-) user knowledge and investigating the external behaviour of the current system. Sometimes documentation and especially end-user user documentation and specifications can be a good source to identify the system functionality. Business process models of the current system are another source for getting the required information. Even the business process modelling activity for the target system can be an input for the functionality of the current system. If we model the business process and business activities, which are supported by the target system, we often get information about the current system as 'secondary products'.

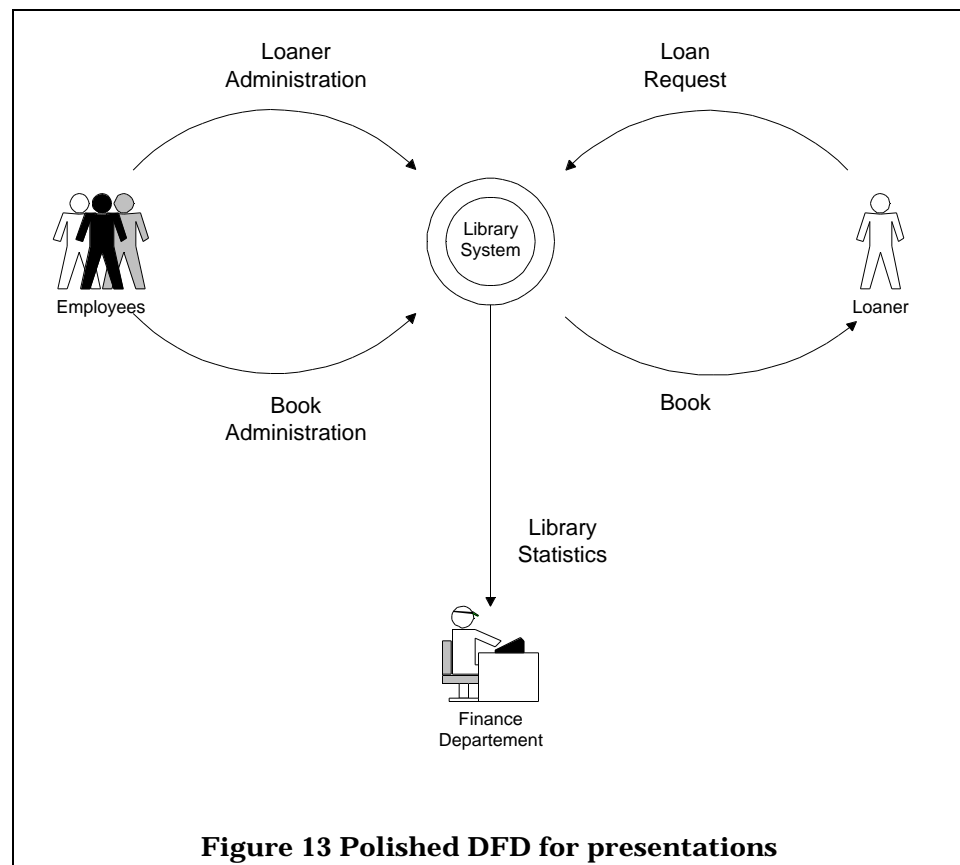
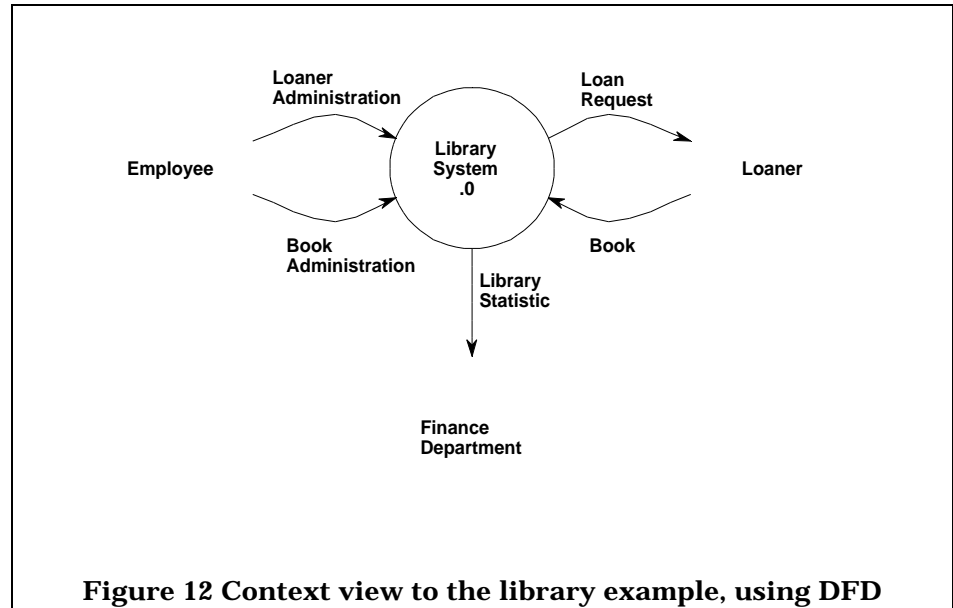
Another objective of data flow diagrams during context modelling is to describe the process structure and file-I/O. Information about this can be derived by monitoring the running current system. In addition the more informal sources like documentation and the knowledge of involved people can be used.

### 2.5.5 Using data flow diagrams to model the library example

As described data flow diagrams can be used in multiple ways during context modelling.

In Figure 12 a data flow diagram is used to model the context of the library system. The diagram contains only one process, the library system. In addition all elements interacting with the library system are drawn as terminators. The data exchanges between the library system and its environment is drawn as data flows between the library system and the terminators. This kind of top-level data flow diagram is called context diagram. It is a very good means to model and document the interface between a system and its environment. It can be used to model the context of an existing system as well as the context of a new system. Changes in the interfaces can be identified and discussed. Constant factors can be described.

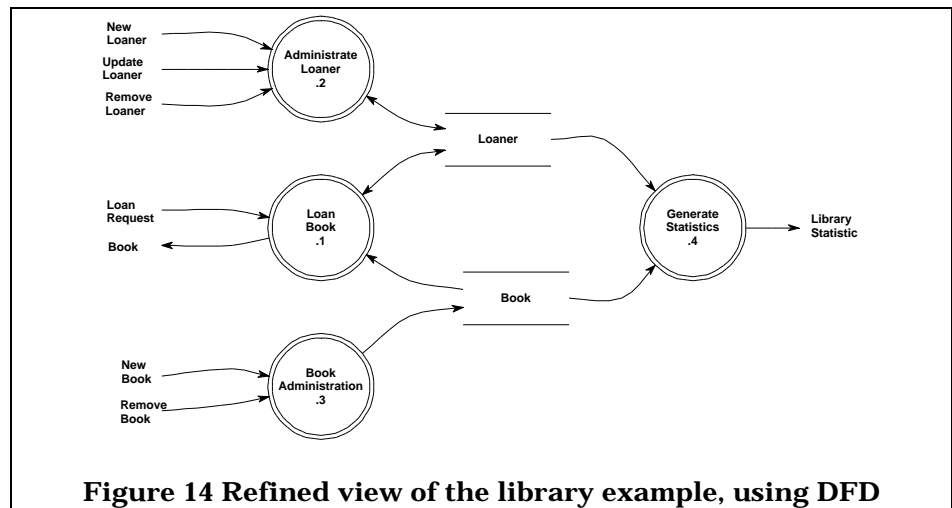
Such top-level diagrams are often made at presentations. Instead of using a formal notation for presenting data flow diagrams at such a high level, a more "polished" version can be made. An example of such a polished version is given in Figure 13.



In Figure 14 the main functionality provided by the library system are modelled as processes (nodes). In addition the major data entities are described as stores and we can see, which functionality is using which

data. The data flow 'Loaner Administration' from the previous context diagram is refined into the three flows 'New Loaner', 'Update Loaner' and 'Remove Loaner'. The data flow 'Book Administration' is refined in the similar way.

Conceptual models of the system functionality can be modelled and described with data flow diagrams in an excellent way. Reusable parts can be identified and extracted. New parts and their integration can be planned.



## 2.6 Extended Entity Relationship Diagrams (EER)

This section describes EER-diagrams as a technique for high-level data modelling.

### 2.6.1 Entity Relationship Diagrams for system evolution

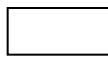
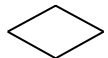

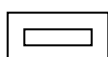
Entity relationship diagrams (ERD) can be used in multiple situations of system evolution projects. They are used to model the main entities and relationships of a system. Typically they are used for:

- ✓ an overview of the data structure of the existing legacy system,
- ✓ an overview of the data structure of the desired target system,
- ✓ to discuss the pros and cons of different data modelling alternatives,
- ✓ to discuss and describe the realisation steps of an incremental approach,
- ✓ as base to derive the data model of the new data base,
- ✓ to identify data structures which can be reused in the new system, and
- ✓ to define the data of the legacy system which must be migrated to the new system.

Entity relationship diagrams have been introduced by (P. Chen 1976). They are technology independent and mainly used for information modelling. Information modelling is a method to model the entities, their attributes and their relationships. The notation of entity relationship diagrams has been refined and extended, i.e. with generalisation (IS-A), weak entities, roles and cardinalities. Today we speak about so-called extended entity relationship diagrams, which include these new elements.

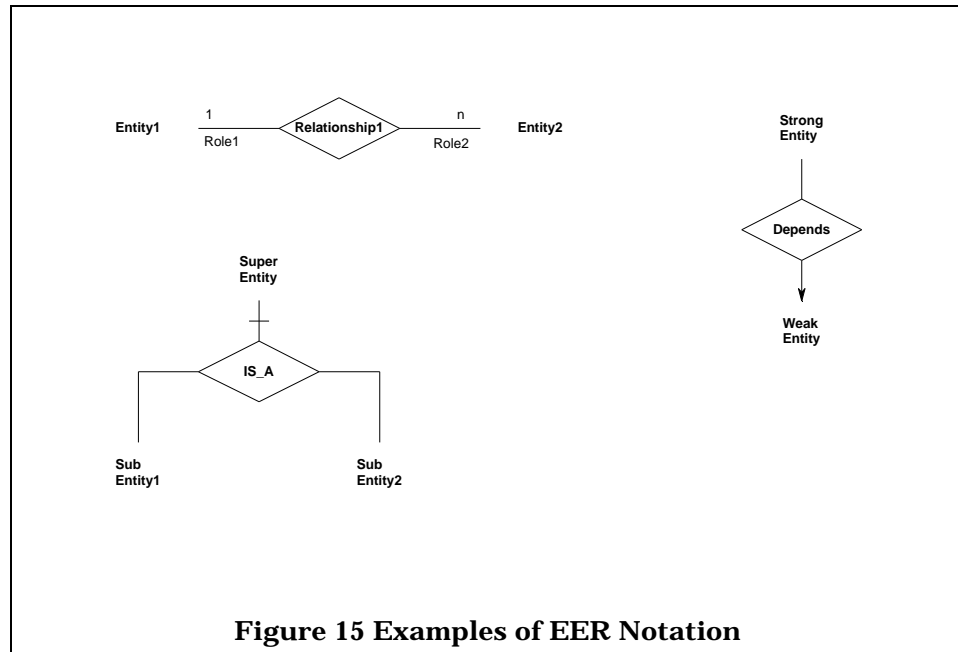
### 2.6.2 The elements of entity relationship diagrams

Entity relationship diagrams consist of the elements described in Table 6. The graphical representation can be refined by further textual details, i.e. attribute descriptions. These descriptions are stored in the data dictionary.

Element	Notation	Usage
Entity		represent a thing which can be identified uniquely
Relationship		represents a relation between entities
IS-A Relationship		a special kind of relationship between supertype and subtype (inheritance, generalisation)
Weak Entity		an entity which cannot exist without a related strong entity
Cardinality	1..n, m	of a relationship
Role	ABC	the role of an entity in a given relationship
Free text	ABC	is used to describe further important aspects

**Table 6 Graphical elements of extended ER diagrams**

### 2.6.2.1 An example



**Figure 15 Examples of EER Notation**

Figure 15 shows how the elements are used to build an entity relationship diagram. We see two Entities 'Entity1' and 'Entity2'. The two entities are connected by a relationship labelled 'Relationship1'. It is a 1:n-relationship. This means one instance of entity 'Entity1' can be connected with n instances of entity 'Entity2'. In the other direction only one instance of entity 'Entity1' can be connected with an instance of entity 'Entity2'.

As well as cardinality, the role that is played by an entity in a relationship can be described. Here we have the role names 'Role1' and 'Role2'. One special kind of relationship is the 'IS-A' relationship. It is used to express a generalisation structure where sub-entities are inheriting from a super-entity. The super-entity is indicated graphically by the cross-line. A weak entity is a dependent entity, which cannot exist without a related strong entity. As a weak-entity can be part of several relationships, the related strong entity and relationship is indicated by an arrow. Note that a weak entity can be weak related to several strong entities. In this case several arrows are pointing to the weak entity.

A cardinality 0..n means a connection with none or several instances. The case 0 can also be expressed by a dashed line indicating a partial relationship.

### 2.6.3 Tips and Tricks

#### General tips

- Information modelling is a method to model the entities, their relationships and attributes (properties) in an accurate way.
- It is sometimes difficult to find the right level of abstraction. When we conduct this work we must be entirely sure of the purpose of the

diagrams. During context modelling we should work on a high abstract level.

- Model only the entities and relationships which are relevant for the application.
- During context modelling attributes should be only modelled if they are important for documentation or assessment.
- Results are achieved only through knowledge of the operation. Without the active involvement of the key people from the user staff, the results will be poor.
- Discuss the entity relationship diagrams with the user staff and refine them.
- Entity relationship diagrams are a business analysis tool, and should be independent of any specific technology.
- Entity relationship diagrams can be used later as a starting point for the concrete database design.

### **Entities**

- Entities are uniquely identifiable things. During context modelling highly integrated entities or entity clusters can be collapsed into one entity to provide better overviews.
- Entities should represent real world things. They should have meaningful names based on the problem space and the objective of the entities should be described.
- To identify entities, textual descriptions, interviews, existing application forms and listings, other models etc. can be used.

### **Relationships**

- Relationships are meaningful interconnections between entities.
- Relationships should represent real world relationships. They should have meaningful names based on the problem space. During context modelling only the key relationships should be described.
- Cardinalities can be used to describe how many instances are related to a relationship. It depends on the concrete project if this detailed level is necessary.

## **2.6.4 Rediscovering entity relationship diagrams in the legacy system**

The main objective of entity relationship diagrams during context modelling is to provide an overview of the underlying data structure. For the current system we can get this overview in two ways:

1. The first way is a more informal one. We study the documentation and collect user know-how. Afterwards we analyse the information and prepare the needed diagrams.

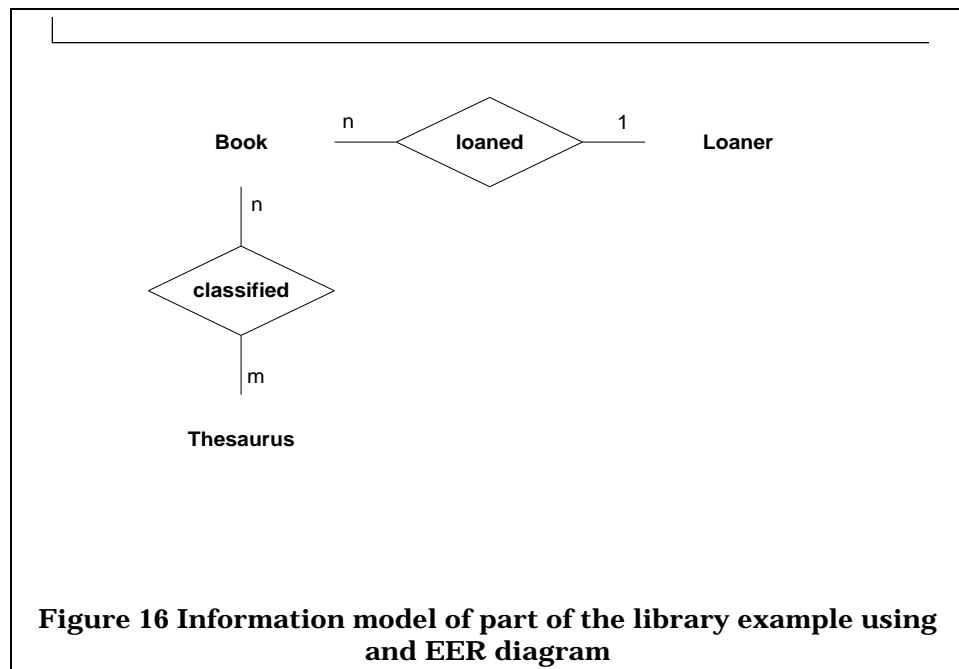
2. The second way starts with a tool supported reverse-engineering step. As the models generated in this way reflect the physical structure and additional abstraction step must follow. In this step the logical structure is rediscovered manually.

In both ways we must avoid to be too detailed. It is not necessary to describe each entity or attribute during context modelling. It is important to identify the key entities, attributes and relationships. The key elements are the elements most important from the application point of view. Entities to support the technology for example can be skipped during context modelling.

### 2.6.5 Using an entity relationship diagram to model the library example

Figure 16 illustrates the main entities and relationships of the library example. The central entity is the 'Book' containing all data related to a book. Exactly one loaner can loan a book. A loaner can loan several books. A thesaurus is used to categories the books. A book can belong to m categories and a category contains more than one book.

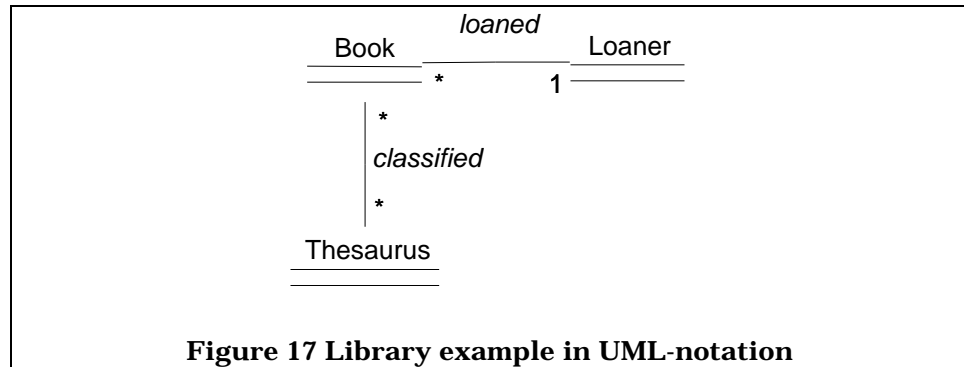
Models of the existing and new system can be described in this way. To plan and control the evolution we can highlight how new data is related to existing.



### 2.6.6 Using UML class diagrams for data modelling

Basically you can also use UML class diagrams for data modelling. The entities can be represented by classes and the relationships by the corresponding UML-relations. The advantage of using UML is that we only need one notation as UML can be used for representing all types of

models. The drawbacks of UML are that it can be too technical and that it is not as well known as ERDs especially in the non-technical environment of context modelling. The following picture shows the library example in UML notation. The asterisk means n-cardinality.



## 2.7 Block Diagrams

The motivation for *block diagrams* is that they make it easier to spot and understand relationships between components in a software system. This is needed in the (re-) engineering of complex software systems to foresee possible problems and conflicts between components and other systems, and to understand and be able to explain the architecture system to other involved actors. Summarised block diagrams can be used to:

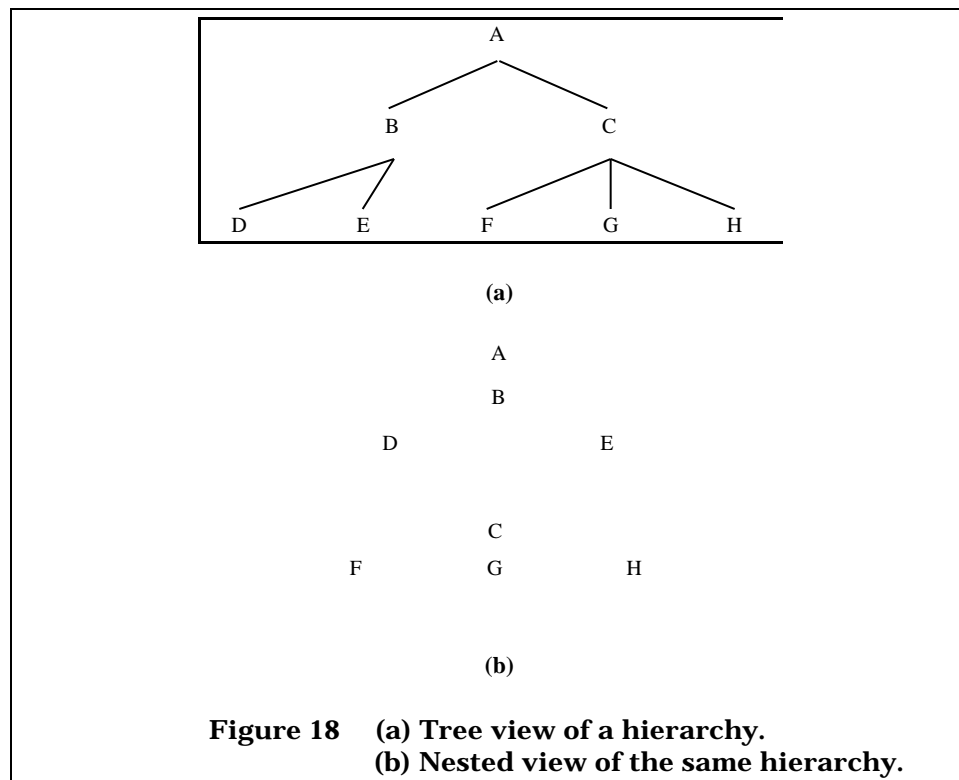
- ✓ Get an overview of the block (logical and physical) structure of the current legacy system,
- ✓ get an overview of the block (logical and physical) structure of the desired target system,
- ✓ discuss the pros and cons of different block structure (software architectures) alternatives,
- ✓ discuss and describe the realisation steps of an incremental approach,
- ✓ derive the software structure of a new system and help defining /understand the hardware structure (architecture),
- ✓ identify blocks or components in the current system which can be reused in the new system, and
- ✓ help identifying and planning the transition from the current system to the new target system.

### 2.7.1 Block Diagrams for system evolution

Block diagrams are in this document treated as diagrams of software components. That is, *blocks* depict software components and a *block diagram* shows the blocks and the dependencies between them.

A set of blocks can be nested within other blocks to allow abstraction to a higher level.

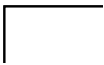
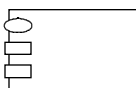
Figure 18 shows two different views of an abstraction hierarchy where *A* is at the highest abstraction level. Part (b) of the figure shows how the hierarchy is nested in the UML. The entire software system under consideration can be depicted as a single high-level block (or *package* in UML terms), *A* in the figure, containing other connected blocks. By allowing this we can have several abstraction levels of block diagrams. Note that the figure does not show the dependencies between the blocks only the hierarchical relations.

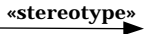


**Figure 18** (a) Tree view of a hierarchy.  
(b) Nested view of the same hierarchy.

## 2.7.2 The elements of block diagram

The notation we have chosen to use for block diagrams is depicted in Table 7 below. However, other notations may be used. The notation we have chosen in this section is supported by the UML and then also by all tools implementing UML.

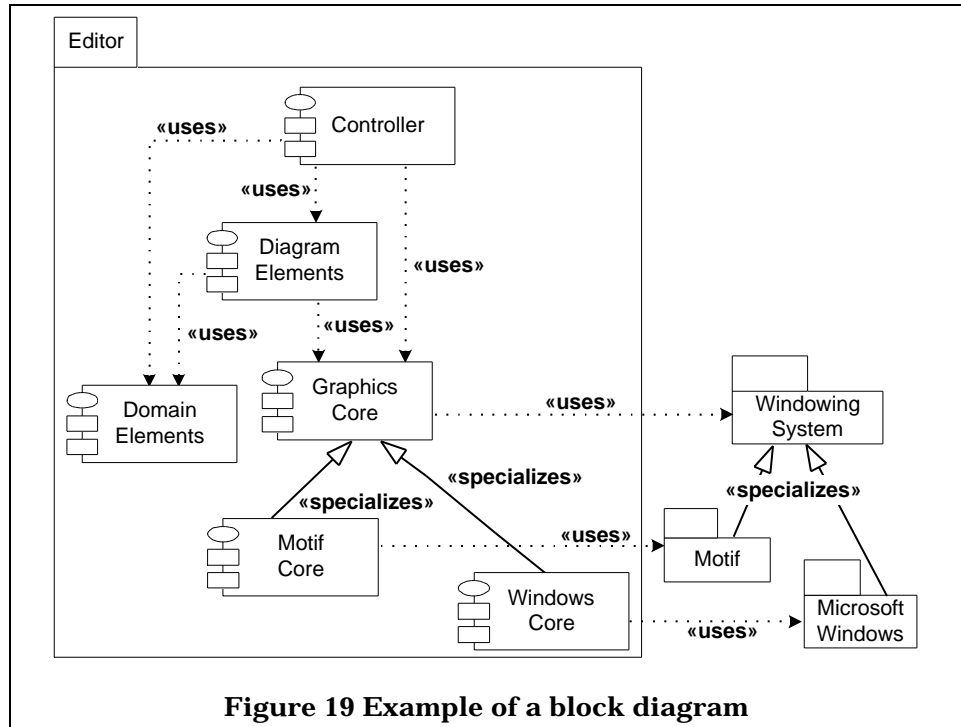
Elements	Notation	Usage
Block		Use the block symbol to model something that you may further decompose. Blocks can be inside blocks to show hierarchy.
Component		Use the component symbol to model something that you choose to view as an atomic unit, which will not be decomposed.

Link		Use a link, annotated with a chosen stereotype, to indicate a relationship and the type of the relationship. Basic “stereotype” indicates the type of the relationship (i.e. “uses”, “contains” or “specialises”).
------	---	--

**Table 7 Graphical elements for block diagrams**

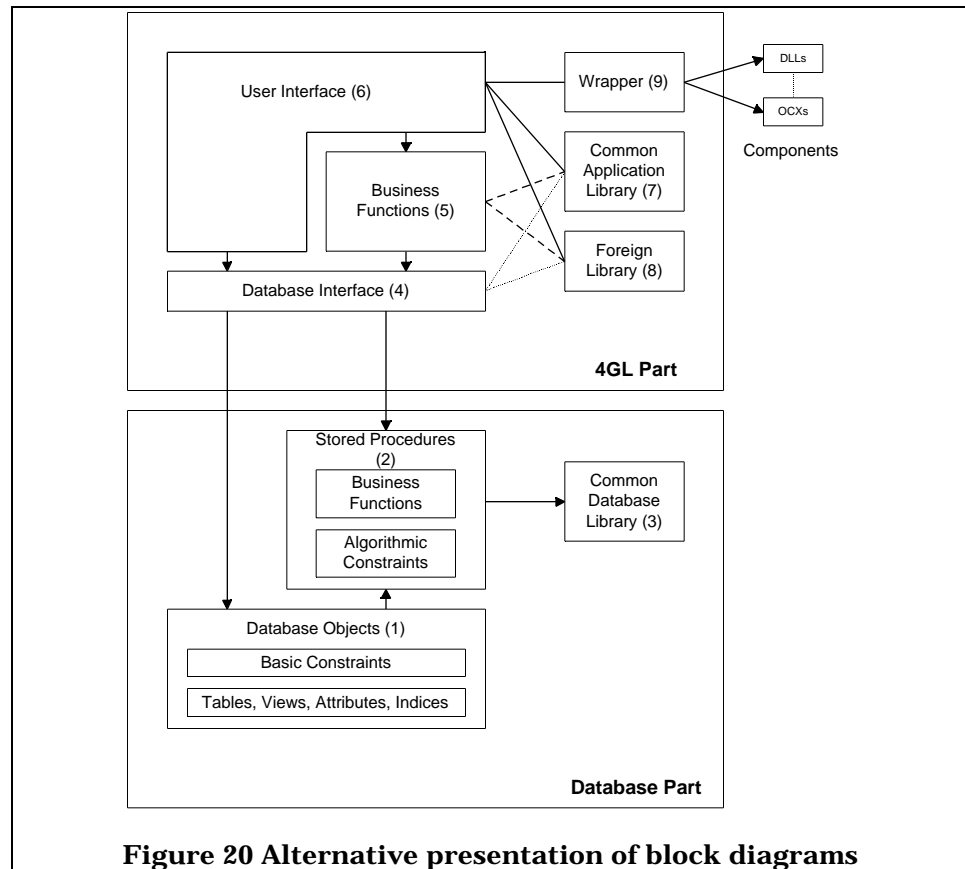
### 2.7.2.1 An example

Block diagrams can be realised by the *component diagrams* and *packaging* features from the UML. A block can be shown by using the notation of *packages* and the dependencies by *arrows* between the blocks. Using the *stereotype* features of the UML, one can name the *dependencies* to depict the type of relation you want between two components. An example is given in Figure 19.



**Figure 19 Example of a block diagram**

Block diagrams can also be presented using blocks that better illustrates the architecture of the system, like in Figure 20 below (only for illustration). Presenting blocks in this fashion may result in a better visual presentation in many cases. Such block diagrams can easily be drawn by almost any standard drawing tool.



### 2.7.3 Logical and physical block diagrams

We basically have two types of block diagrams:

1. Diagrams showing dependencies between logically grouped software providing a *logical view* of the system.
2. Diagrams showing dependencies between physically grouped software, providing a *physical view or development view* of the system.

The *logical view* is presented by applying a block diagram where each block represents a collection of logically related software components. E.g. UI related components will be grouped into one block, I/O controls into another one, and the database into a third one, etc.. This is what is usually shown by the high-level UML *component diagrams* (UML v1.0, 1997).

The *physical view* is presented by a block diagram where the blocks reflect physically related software components such as files or subsystems and the dependencies show the relationships between them. These kinds of block diagrams are well suited to map software components onto hardware and network devices. The software components can be mapped onto hardware components and the relationships onto network links making a mapping between block diagrams and *hardware/network diagrams* (hardware/network diagrams are described in Section 2.8).

In most cases we want to have both types of block diagrams to get both a logical view and a physical view of the system. The *logical view* provides

an easy to understand and intuitive model of the functionality of the system. This can be used in presentations to other involved actors of the development process. In addition, a logical view can be used to identify the target architecture and possible reusable components. The *physical view* provides a model that eases the mapping to and from hardware and network devices and will be useful for assessing the hardware and network needs of the system. Sometimes a one-to-one mapping between a logical view and a physical view may be possible. This is, however, usually *not* the case for legacy systems.

## 2.7.4 Tips and Tricks

There are three main approaches to obtain a block diagram:

1. Top-down, starting at the most abstract level.
2. Bottom-up, starting at the lowest abstraction level available, i.e. starting from the output of a reverse-engineering tool.
3. A combination of the top-down and bottom-up approaches.

The suitability of the selected approach depends on what kind of diagram (logical or physical) and whether or not it is a part of a reverse-engineering process. If so, it also depends on what kind of information is available to the reverse-engineering process.

### 2.7.4.1 The top-down approach

#### Logical view

Start with identifying the top-level block. This block depicts the whole system and is what other external systems “see”.

The next step is to identify the main *logical* parts of the system and the dependencies between them. This will be the most abstract view of the system and is either captured from the documentation and/or by a person who knows the system. At this level the block diagram should not be too large and it should be understandable to all actors in the project. How the blocks are identified are mainly up to the designer of the diagram, but he should strive after high cohesion and low coupling between the blocks. That is, the blocks should contain components that are highly related and it should be as few as possible dependencies between the blocks. Examples of blocks at this level are main system, UI, and database blocks.

Then, after the top-level of the system is ready, one may start to refine each of the blocks to achieve a higher level detail. Ideally, this top-down refinement can be carried out until one reaches the implementation level.

#### Physical view

Modelling the *physical* aspects of an *existing (legacy)* system by the top-down approach is not feasible. If you have the code or the structure of the existing system, you will have to use the bottom-up method, or at least the hybrid approach, to obtain a block diagram. However, when developing a *new* system it may be a good idea to use the top-down approach. Using the ideas from above, one will start with identifying highest level components and their dependencies. This can be subsystems and how they depend on

each other. The next level can be the files making up the subsystems and so on.

In general the top-down method can be hard to use with reverse-engineering, especially when the documentation is missing or insufficient and there is no one who know the system well.

#### 2.7.4.2 The bottom-up approach

To make physical block diagrams of a legacy using the bottom-up approach, we need some kind of documentation of the structure or, at least the code, of the system. We must extract the arrangement of software modules from the system's documentation and code to make block diagrams giving a graphical view of the system dependencies.

This kind of reverse-engineering is supported by a number of available tools giving a variety of low-level diagrams such as structure and flow diagrams, e.g. Rigi described in (Tilley et al., 1993). However, such low-level diagrams are often complex and hard to read for non-technical persons. They have to be abstracted to a higher level, making them more readable and easier to work with. This is done by clustering of either logical or physical related components to move the system description to a higher level of abstraction. Repeating this bottom-up approach one will end up with a top-level block diagram, hopefully showing the main structure of the system.

The problems with the bottom-up approach appear when documentation and understanding of the existing system is poor. If also the system is non-decomposable and of high complexity, even good reverse-engineering tools may yield unhelpful outputs. The challenge is to structure and group the components by hand. This can be a very time consuming and expensive process.

The bottom-up approach for making block diagrams is not well suited for development of new systems, particularly large and complex ones. The bottom-up method can in such cases be applied in together with the top-down method, yielding a hybrid approach.

#### 2.7.4.3 The hybrid approach

This approach is a combination of the top-down and bottom-up methods presented above.

#### 2.7.4.4 Summary

Table 8 summaries which approach is best suited for which diagram and process.

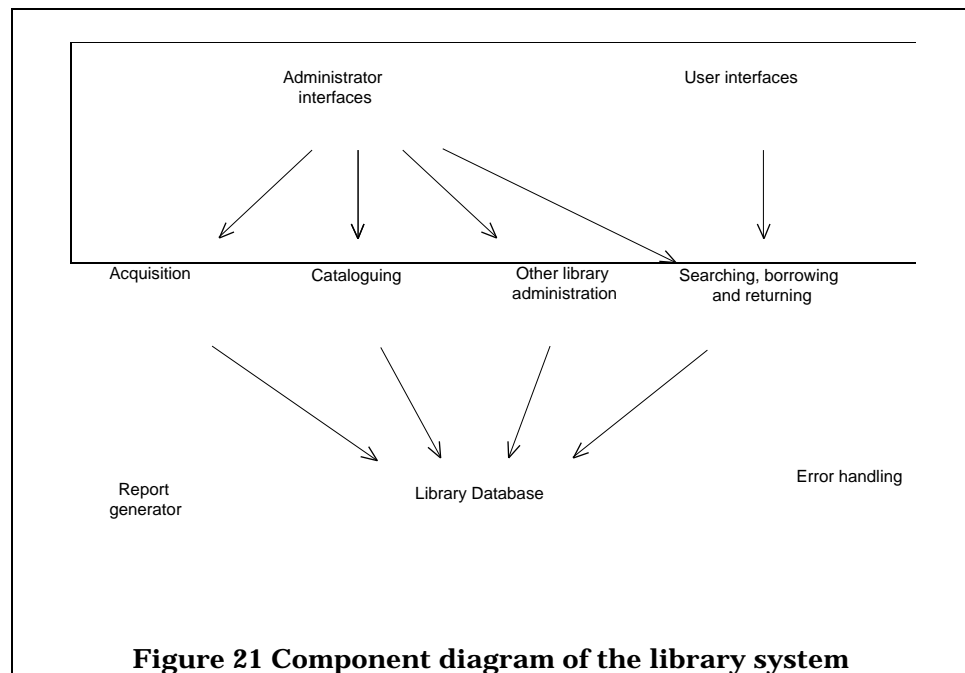
Approach	Logical view	Physical view
Top-down	Good for new development. OK for re-engineering	OK for new development. Bad for re-engineering.
Bottom-up	Bad for new development OK for re-engineering	OK for new development Good for re-engineering
Hybrid	Good	Good

**Table 8 Comparison of diagram type and modelling approach.**

The most used dependency between software components (blocks), are the *uses* or *is used by* dependencies.

### 2.7.5 Using a block diagram to model the library example

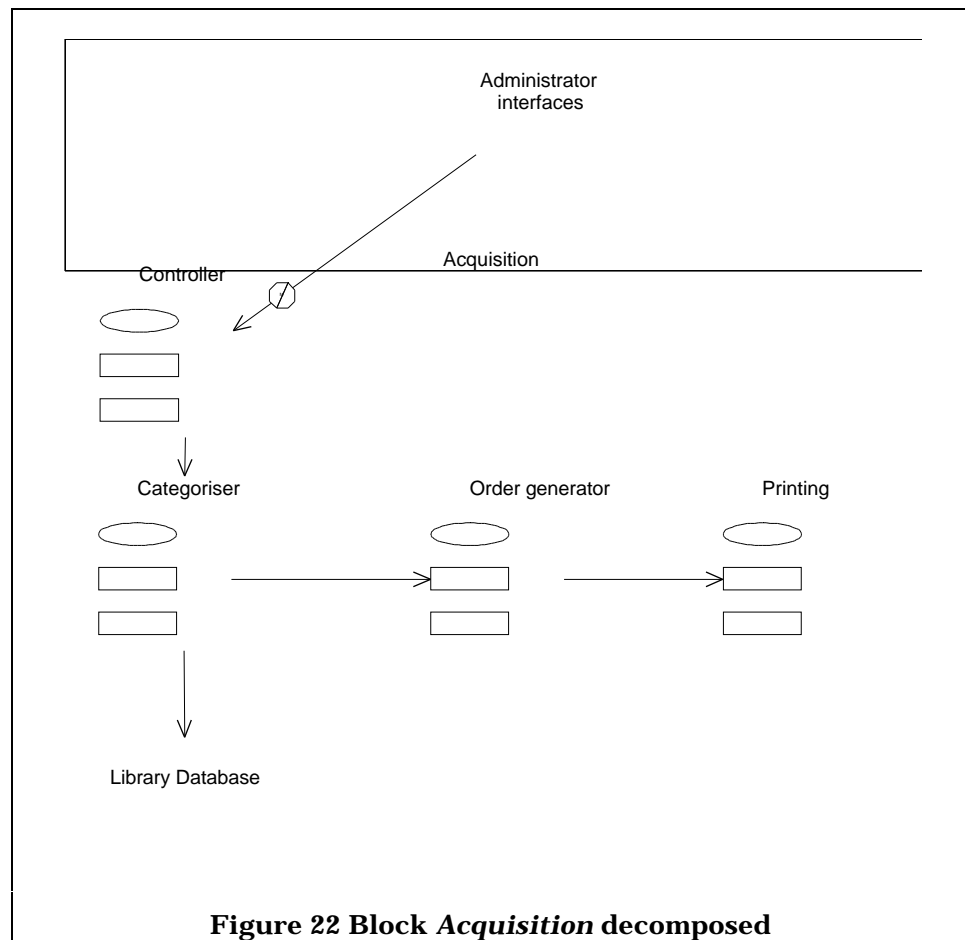
To illustrate the use of block diagrams using the UML component diagram notation a portion of the library system is modelled. Figure 21 shows the top-level block diagram of the system. The block diagram shows the *logical* part of the system, but it may as well also correspond to the final *physical* block or component diagram. The figure has divided the *user interface* into two: One for the *library staff* performing administrative tasks, and one for the *user* of the library. The idea behind this is to have two separate client applications. The user interface blocks then *communicates* (or *uses*) with four other blocks (or components) which encapsulates the application logics of *Acquisition*, *Cataloguing*, *Other library administration* (see Appendix B), and *Searching, borrowing and returning books*. These four components do again communicate (or uses) a database component (*Library Database*) which is the interface to the database. *Report generator* and *Error Handling* are other components that are likely to be found in the system.



To illustrate the hierarchical property of block diagrams, the *Acquisition* component is decomposed to show its internal structure. Figure 22 shows an example of how this can be done. The *Controller* is the component that controls the acquisition process and supports the business process of acquiring books. The *Categoriser* categorises the suggestions from the users of new books to buy and sends it back to the controller (or in other word: the controller *gets* a categorised list from the categoriser). Then,

based on the edited list, an order is generated by the *Order generator*. The order is at last printed using the *Printing* component.

As can be seen from the diagram below, the *Controller* takes care of the requests from the *Administrator interfaces* and the *Categoriser* uses the *Library Database* component (to get a list of suggestions).



## 2.8 Hardware/Network Diagrams

### 2.8.1 Hardware/network diagrams for system evolution

In the framework of system evolution, the hardware and network composition and structure of a system is likely to be replaced or to be modified. The mapping of software entities composing the system on these hardware components needs also to follow this evolution. In addition, the hardware structure of new, distributed, applications is usually more complex than the structure of legacy systems and is an even more necessary part of the documentation of a system.

Therefore, hardware and network diagrams are needed in the framework of system evolution, especially for:

- documenting the hardware (and network) components of a legacy system, including descriptive and sizing attributes,
- designing and documenting the hardware components of the new system, including descriptive and sizing attributes,
- designing and describing the mapping between legacy system hardware components and new system components or for showing the integration of old components in the new system,
- describing the mapping of software (code and data) and run-time components (processes) onto hardware components

Hardware and network diagrams depict the physical components of a system, including all types of computers and other hardware devices involved in the system, and the communication links between them.

The following elements and links need to be captured on hardware and network diagrams:

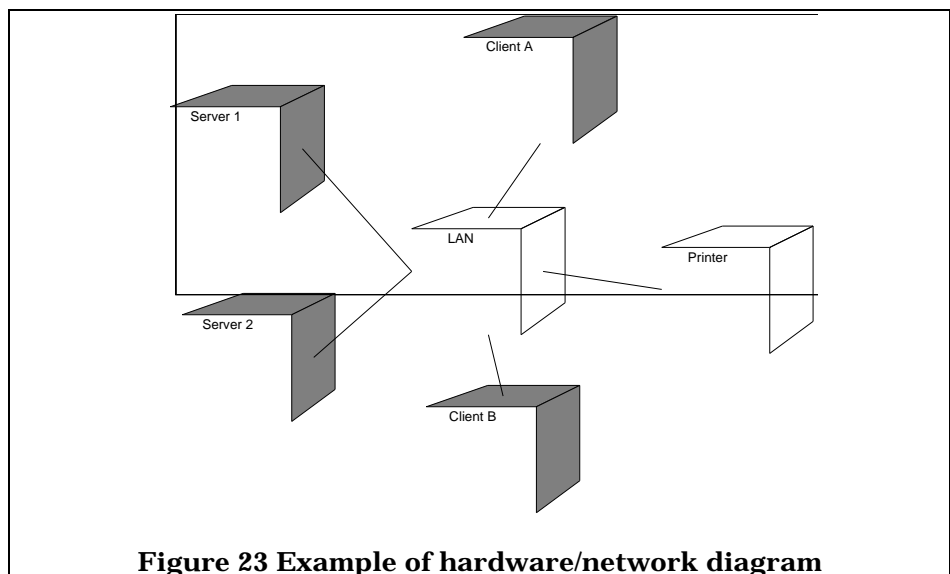
- servers, workstations and other computers (including location and capacity information)
- other devices (printers, scanners...)
- communication links and nodes (routers, firewalls...)

## 2.8.2 The elements of hardware/network diagrams

A *Deployment Diagram* is one of the two *Implementation Diagrams* defined in standard UML. The purpose of a deployment diagram is to show the configuration of software components and their mapping on run-time entities (represented as nodes).

A *Hardware/Network Diagram* is a new type of *Implementation Diagram* that is used only to give a comprehensive view of the hardware architecture of a system. A hardware/network diagram is only composed of nodes as defined in UML.

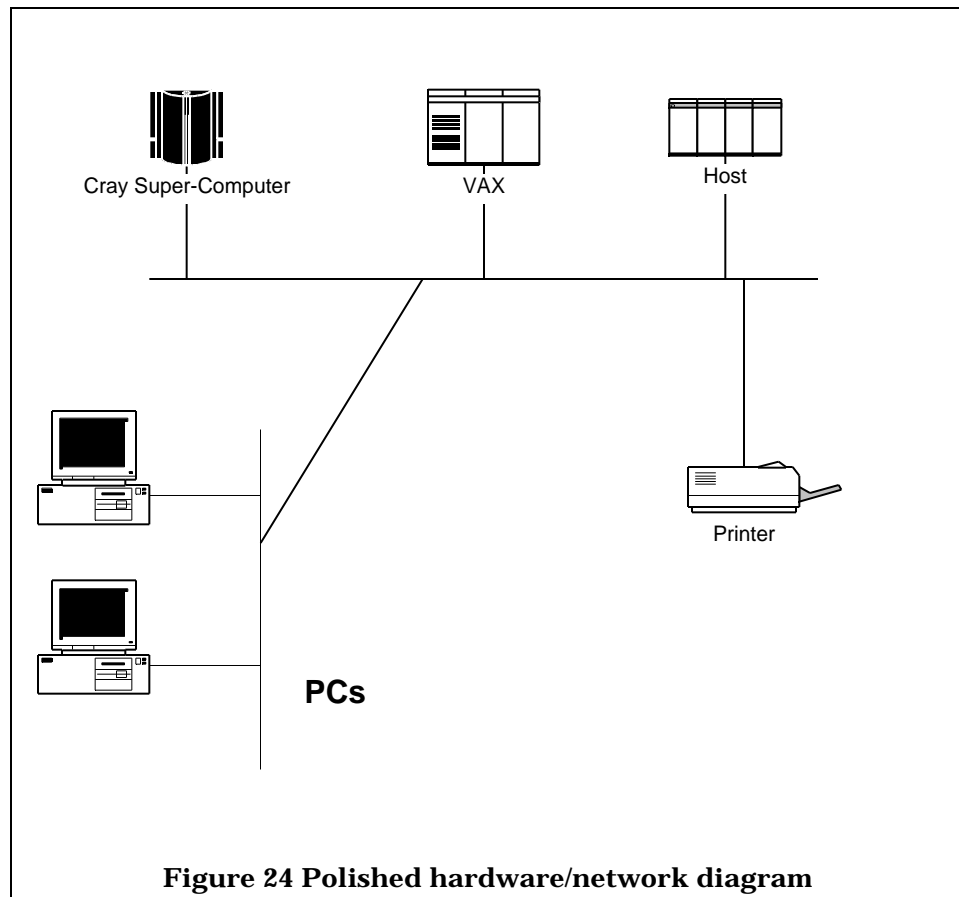
An example of hardware/network diagram is given in Figure 23:



For presentation purposes, it is good to use a more polished diagram, using different symbols to show different type of equipment.

### 2.8.2.1 An example

An example of this is shown in Figure 24.



### 2.8.3 Tips and Tricks

Elements composing hardware/network diagrams are concrete artefacts of a system. When modelling an existing system or a planned system, the decisions to be taken are related to the way elements must be gathered on diagrams, the number and types of such diagrams, and on the level of detail of information represented on these diagrams.

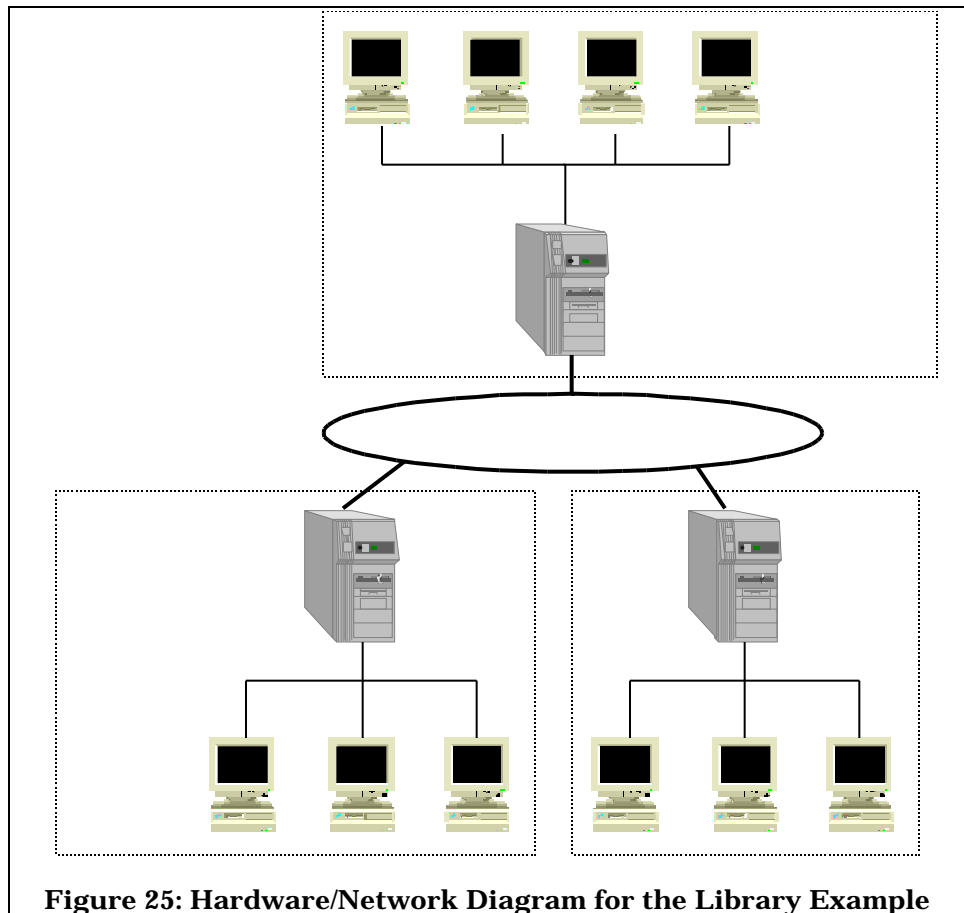
General guidelines for building such diagrams are:

- Make diagrams easy to understand by limiting the number of nodes and links on each diagram.
- Structure the information to be modelled by building hierarchical views with diagrams and sub-diagrams.
- Use several diagrams or views of a same diagram to represent different types of information.

- Create generic diagrams to avoid redundant information, for instance when several geographical locations share a common system architecture.
- Add volume and capacity information when available to help and size the system or to document it. Such information includes size of data storage, network bandwidth, network load, memory size...

#### 2.8.4 Using a hardware/network diagram to model the library example

The following picture depicts a typical hardware/network diagram for the library example. Departments are equipped with a local area network composed of a local department server and department workstations. The library itself also has the same type of topology.



**Figure 25: Hardware/Network Diagram for the Library Example**

---

## 2.9 Traceability of Context Models

Context modelling, as suggested by RENAISSANCE, involves describing a system from different viewpoints. When a software engineer makes a context model, it is important that a system element described in one viewpoint can be related to a different view of the same element described in another viewpoint. Either case tools or process guidance must be given to support this need for traceability.

RENAISSANCE provides the modeller with an arsenal of techniques for context modelling, not all of these are supported by one case tool. In order to alleviate for this drawback, we provide you with some guidance for how to ensure sufficient traceability in your context models.

The purpose of traceability is twofold:

- Describe guidance for how to ensure inter-viewpoint traceability, i.e. among different context modelling diagrams, and
- Describe guidance for how to ensure traceability from the context models to the technical models. More specifically, we describe how to incrementally create a technical model from a context model.

Traceability in software modelling and development is important for two main reasons:

1. It supports the comprehension of the software system.
2. It supports impact analysis of changes made to the model or software.

Traceability involves capturing of relationships between components in the system. That is, keeping track of dependencies within models, between different kind of models, within hierarchies and between models and software.

The importance of traceability can be summarised as follows:

- Traceability makes it easier to answer to any changes to the system, like changes of requirements.
- Traceability makes it easier for different persons to work on different parts of the system by giving the ability to get an overview and insight to related parts or models.
- Traceability makes it easier to reach agreements by presenting the system in an unambiguous way.
- Traceability keeps the costs low for the reasons listed above.

### 2.9.1 Tracing context models – Inter-viewpoint traceability

Traceability between different types of context models is important to ensure that they model the “same” system. This is especially important when modelling is done by several persons with different skills, knowledge and understanding of the system.

Table 1 in the introduction to this chapter presents the overview of the traceability between the different schemas. A more detailed traceability

between the context modelling techniques presented in this chapter is summarised in Table 9 (on the next page).

Diagram type	Element	Maps to
Operational Schemas	Activity	Message in sequence and collaboration diagrams
	Database	Database in (E)ER diagrams Data store in DFD diagrams
	Activity Flow	Communication link in use case diagrams Message in sequence and collaboration diagrams Data flow in DFD diagrams
	Alternative	
	Hardcopy	Data store in DFD diagrams
	Link to external business process	Communication link in use case diagrams
	Link from external business process	Communication link in use case diagrams
Use cases	Actor	Actor in sequence and collaboration diagrams External agent in DFD
	Use case	Operational schema (business process) Process in DFD
	Communication link	Link to/from external business process in operational schemas
<i>Sequence diagrams &amp; Collaboration diagrams</i>	Actors	Actor in use case diagram
	Objects/Classes	Classes in class diagrams (technical)
	Messages/Data flow	Member methods in class diagrams (technical) Flow in DFD
Data flow (DFD)	Node (Process)	Node in deployment diagrams Use case in use case diagrams Actor (system) in use case (sequence and collaboration) diagram Block in block diagram
	Data flow	Message in sequence and collaboration diagrams Communication link in use case diagrams Activity flow in operational schemas
	Data store	Database in (E)ER diagrams Database in operational schemas Hardcopy in operational schemas
	Terminator (External agent)	Actor in use case (sequence and collaboration) diagram
	Free text	
(Extended) Entity Relationship (EER)	Entity	Data store in DFD diagrams

Diagram type	Element	Maps to
	Relationship	Association in class diagrams (technical)
	IS-A relationship	Aggregation in class diagrams (technical)
	Weak Entity	Data store in DFD diagram
	Cardinality	Multiplicity in collaboration diagrams Cardinality in class diagrams (technical)
	Role	
	Free text	
Block	Block	Maps to processor in deployment diagram, or machine in hardware/network diagrams
	Component	
	Link	
Hardware/Network		

**Table 9 Traceability between diagrams**

The traceability matrix in Table 9 can also be used as a template for an instantiated matrix in a re-engineering project. Such a traceability matrix may be used to keep track of the dependencies between models and help project members with the inter-communication. This is not always easy to provide, especially when using different tools and persons in the modelling process.

## 2.9.2 Going from context models to technical models

Another important dimension of traceability, which must exist, shows the dependencies in the hierarchical dimensions. That is, it has to be possible to trace both downwards and upwards between the different levels of detail in the system – from high level context models, to lower level technical models and all the way down to source code and database schemas, and opposite.

In the approach described in this document, context models will gradually be refined to become what we call technical models, which again will be used to generate or manually produce source code and databases.

To obtain this kind of traceability one need to organise the iterative analysis and design phases and techniques to support the linking of models and code in the level of detail dimension. The models used throughout this document can be traced as described below. Of course, if case tool modelling support is used, such traceability is typically built in.

### 2.9.2.1 Operational schemas and use cases

Operational schemas and use cases (elaborated by sequence and/or collaboration diagrams), in addition to help defining (system) roles, also provide the first steps in identifying objects/classes and their data and method members. For non-object oriented systems these techniques help with identifying logical functions and control flows.

### 2.9.2.2 Data flow diagrams

The high-level data flow diagrams in context modelling provide the first pictures of the data flow in the system, including the needed/existing data stores, input required, output provided and direction of data flows. This will be helpful when identifying naturally grouping of subsystems and necessary methods for message passing between modules. Refinement of the high-level data flow diagrams will provide more detailed information on the functionality and data flow of the system. Achieving traceability in data flow diagrams is achieved by, for instance, tag the nodes with unique numbers identifying the node and showing its placing in the hierachy (i.e. node tagged *1.2* indicates node number *2* having the parent tagged *1*).

### 2.9.2.3 (Extended) entity relationship diagrams

Some of the data stores identified in the data flow diagrams are databases. The (E)ER diagrams in the context modelling illustrates the high-level conceptual model of the a database. These can be further refined to more detailed data models, all down to a level which it can be generated database schemas from. The traceability can be kept by tagging the entities and relationships and/or by using a tool (like *ERWin* from Rational Inc.) which supports iterative refinements of entities and relationships. The UML notation may also be used and though also the tool supported traceability possibilities.

### 2.9.2.4 Block diagrams

Traceability of *Block diagrams* is easily obtained by using the same tagging technique as for data flow diagrams. This way one can easily track the “blocks” to its parents and though group the blocks to one parent and vice versa.

## 2.9.3 From legacy systems to evolutionary systems

When a new system is built based on a legacy system or part of it, modelling guidelines can be used to help understand its evolution. This section does not address generic modelling guidelines for building systems. Rather, we address how to derive diagrams for the target system from diagrams of the legacy system.

To understand the evolution of a system, it must be easy to identify what parts of the legacy system are reused and where in the new system. In this respect, only high-level diagrams (context modelling diagrams, and especially block diagrams used to model the main building blocks of the legacy system) are concerned. Modelling high level diagrams for the new system should be based on equivalent diagrams modelling the legacy system, and by highlighting modifications done to integrate or include them.

For instance:

- new operational diagrams should highlight which new tasks are supported by the information system,
- new block diagrams should include both new application parts and legacy application parts,

- hardware/network diagrams should show how old and new computers and devices are connected.

This will not only ensure that diagrams can be reused to help and understand the technical evolution of the system, but it also ensures that modelling allows the identification of reusable parts of the system.

## 3 Technical Modelling

### Contents

- 3.1 Introduction
- 3.2 Technical Modelling of 3GL applications
- 3.3 Technical Modelling of 4GL applications

### Summary

When decisions have been taken about which strategies to use in the system evolution project, further details about the current system must be known in order to implement these strategies. This technical modelling part of the document identifies properties and relationships that must be identified and modelled in existing systems. The chapter concerns applications originally implemented using either 3GL or 4GL technology.

## 3.1 Introduction

The previous chapter described mechanisms for modelling the legacy application at the conceptual or context level.

This chapter focuses on identifying the technical properties common for legacy applications and new systems, and how these properties can be modelled. That is, we look inside the system to model its organisation.

Section 3.2 identifies the technical properties for applications that are developed or will be developed applying one or more 3GLs.

Section 3.3 identifies the technical properties for applications that are developed or will be developed using a 4GL. As a special exercise we describe a Meta model of a SQLWindows. This provides the reader with a detailed example of how the design concepts of a typical 4GL are related. We also describe the important properties of RDBMS, which are usually the common core base of 4GL applications.

In Appendix C: 4GL Notations we move on to describe how these properties should be modelled using the Unified Modelling Language (UML). We describe in detail how the properties identified should be modelled using the diagrams and elements provided in UML. Where appropriate, the mapping from 3/4GL to UML is supported by examples.

If you are not familiar with UML, an introduction to UML concepts and diagrams is provided in Appendix A: Overview of the UML.

## 3.2 Technical Modelling of 3GL applications

In the framework of system evolution, both legacy systems and new systems need to be modelled. Therefore, both old and new 3GL can be found in systems to be modelled.

Generally speaking, Third Generation Languages (3GL) are high-level procedural computer languages (such as COBOL) incorporating advanced semantics and syntax. They may be based on a small or large set or keywords, each usually performing a limited task. Applications developed using 3GL are usually slower and more complex to develop but they offer developers a vast richness of possibilities.

A large number of 3GLs have been used and still exist in current applications. It is hardly possible to draw an exhaustive list of 3GLs and their properties, but for the sake of conciseness, this section focuses on COBOL and FORTRAN as representatives of old 3GLs and on C and C++ as representatives of newer 3GLs. In addition, scripting languages are often used to form the glue of the overall software system and their properties are also described. Databases and data files, which may be used by these programs, are not described in this section.

---

### 3.2.1 The properties of 3GL applications

Only properties that are useful to be modelled in the context of system evolution are reviewed in this section. For instance, properties of 3GL which describe the global data or code structure or properties which links these elements are listed. Other properties, which only express semantics or local definitions, are not listed.

In addition to programming language properties, properties related to the generation of executable programs from source files are listed (e.g. object files), since this information can be very important when reverse-engineering a legacy application.

Therefore, the following elements need to be modelled:

- source files (in programming languages or script languages)
- executable programs
- intermediate files created during the generation process (object files...)
- auxiliary files (libraries, images, icons, messages, screen definitions...)
- configuration files
- database definition files
- data files
- documentation files and
- all other relevant files...

#### 3.2.1.1 COBOL language properties

COBOL (Common Organization Business Oriented Language) was designed in 1957 following a request of the US government to define a common language to better manage the complexity resulting from the use of multiple languages and platforms. COBOL is mainly targeted at business applications.

Several flavours and standards of COBOL exist. This document focuses on COBOL A.N.S. 85 defined in 1985.

COBOL is a programming language based on the use of the English language and structured in DIVISIONS, SECTIONS, paragraphs and sentences composed of verbs, nouns, punctuation and operators.

A COBOL program is usually structured as:

- an IDENTIFICATION DIVISION containing an identification of the program,
- an ENVIRONMENT DIVISION describing communication with external files,
- a PROCEDURE DIVISION containing the processing part of the program,
- a DATA DIVISION structured as
  - a FILE SECTION defining buffers holding data passed between files and main program memory
  - a WORKING-STORAGE SECTION defining internal data used in main program memory

- a LINKAGE SECTION describing data defined in another program and allowing exchange of data between several programs

The main properties of COBOL programs are:

### Source file

A source file contains the textual description, written using the language syntax, of a COBOL program.

### Executable Program File

An executable program file is the result of a generation process, which transforms source files into executable files which can be directly run on a specific target environment.

### Procedure

A procedure is a named sequence of language sentences defined in the procedure division of a COBOL program. A procedure can be explicitly executed using a PERFORM statement.

```
PROCEDURE DIVISION
START.
    OPEN INPUT FDISK
    MOVE INDICAR TO VALCAR
    ...
```

### Logical File

Logical file defined in the file section of a COBOL program.

```
DATA DIVISION
FILE SECTION
FD TESTFILE LABEL RECORD STANDARD
VALUE OF FILE-ID "A:TESTFILE.FIL"
DATA RECORD TESTREC

01 TESTREC
02 TESTCOD PICXXX.
02 TESTNAME PIC X(20).
...
```

### Physical File

Physical file associated to a logical file.

### Data Declaration

Single elements (data), or groups of elements (structure, table)

```

DATA DIVISION
WORKING-STORAGE SECTION
77 I   PIC 9(4).
77 J   PIC 9(6) VALUE ZERO.

01 TABLE.
02 F1  PIC 9(5).
02 F2  PIC XXX.
...

```

Relationships among these elements can be modelled as in Figure 26:

### 3.2.1.2 FORTRAN language properties

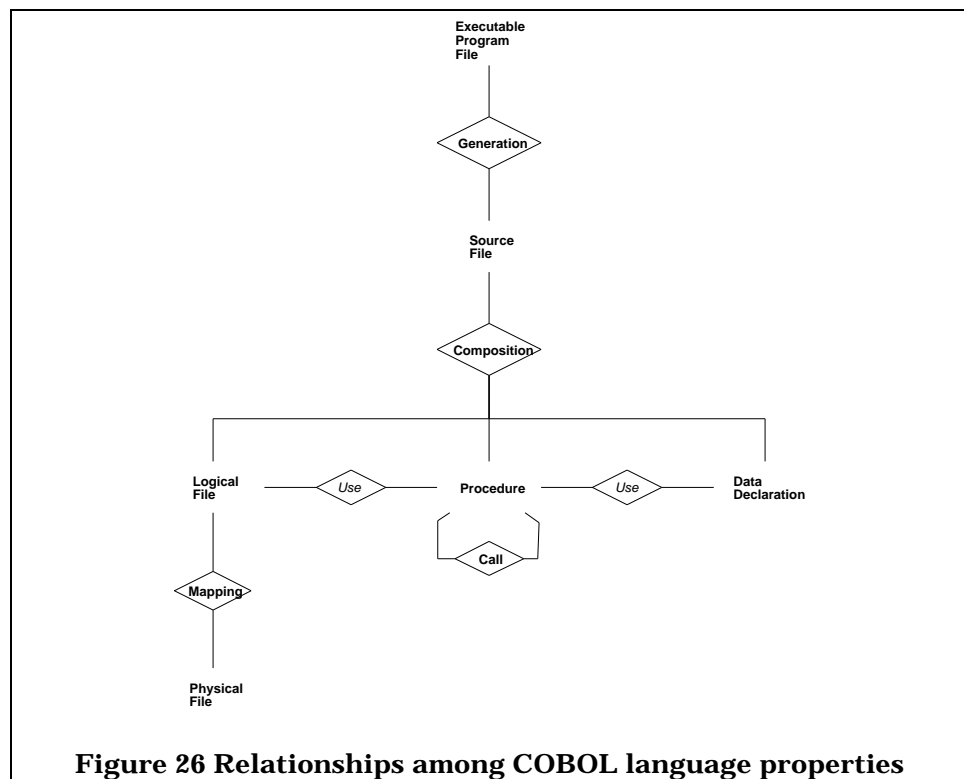
FORTRAN (FORmula TRANslation) is one of the earliest high-level languages and has been designed by a group at IBM in the late 1950s. FORTRAN is targeted at scientific and engineering applications.

Due to the popularity of the language, several flavours and dialects of FORTRAN exist. This document focuses on FORTRAN 77.

The main properties of FORTRAN programs are:

#### Source file

A source file contains the textual description, written using the language syntax, of a FORTRAN program.



#### Executable Program File

An executable program file is the result of a generation process, which transforms source files into executable files, which can be directly run on a specific target environment.

### Subroutine/Function

A subroutine or a function is a named collection of statements, which can be called and executed by a CALL statement or implicitly in the case of a function by using its name:

```
C  EXAMPLE OF SUBROUTINE
  SUBROUTINE TEST(I,J)
    INTEGER I,J
    INTEGER TEMP
    TEMP = I + J
    RETURN
  END

C  CALLING A SUBROUTINE
  CALL TEST(1, 2)

C  EXAMPLE OF FUNCTION
  REAL FUNCTION PERCENT(T)
  REAL T
  PERCENT = T / 100.00
  RETURN

C  CALLING A FUNCTION
  REAL A, B
  A = 76.00
  B = PERCENT(A)
```

### Physical File

A physical file is used in a program by associating it to a unit number and then referring to this unit.

```
OPEN(UNIT = 9, FILE = 'TEST.DAT', ACCESS = 'SEQUENTIAL',
+     STATUS = 'OLD')
```

### Data Declaration

Data used internally can be either simple or structured as an array:

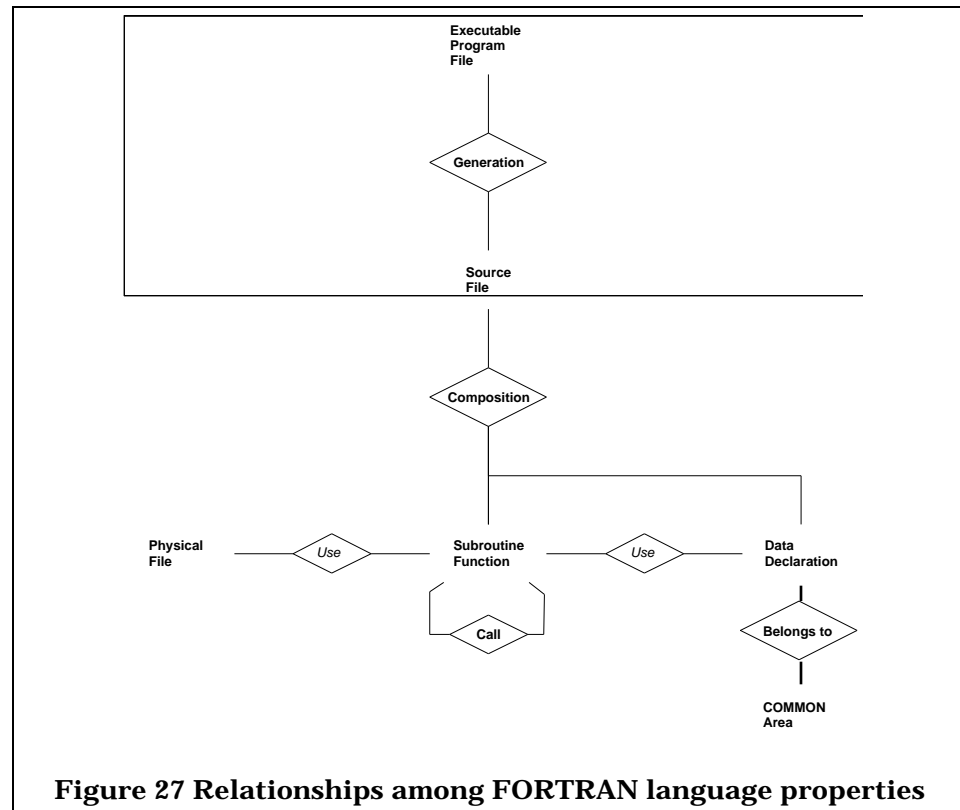
```
INTEGER I
REAL BALANCE, IN, OUT
CHARACTER*20 NAME

REAL TAB(10)
CHARACTER*20 NAMELIST(10)
```

### Common area

Global data declarations shared between programs/subroutines.

Relationships between these elements can be modelled as in Figure 27.



### 3.2.1.3 C and C++ language properties

The C programming language was developed in the early 1970s at Bell Laboratories. C is very popular in the UNIX world and many UNIX operating systems are almost entirely written in C.

From the conceptual viewpoint, C is a procedural based language (functions) whereas C++ is an object based language (classes, methods, instances...). On a technical viewpoint, C++ extends the C programming language, mainly by adding object-oriented concepts and stronger typing. This is the reason why this section presents C and C++ at the same level.

The main properties of C and C++ are:

#### Source File

A source file contains the textual description, written using the language syntax, of C or C++ definitions.

#### Object File

Intermediate file generated in the executable construction process (result of the compilation of a source file).

An object file is usually generated for every source file.

### Executable Program File

An executable program file is the end result of the generation process which transforms source files into executable files which can be directly run on a specific target environment.

Executable programs are usually generated by linking and resolving dependencies between object files.

### Function

A function is a named collection of statements with parameters and possibly a return value.

```
int f(int param1, int param2) {
    int temp;
    temp = param1 + param2;
    return temp;
}
```

### Data Type Definition

Definition of simple or structured data type:

```
typedef int Number;

struct complex {
    double r;
    double i;
};
```

### Class (C++ only)

Definition of a related collection of methods and attributes. Classes acts as user types to create a variable, also called instance of this class, or simply *object*:

```
class Point {
public:
    Point(int xx, int yy);
    void Plot();
private:
    int _x;
    int _y;
};
```

### Variable

Definition of a typed data element. Only the definition and use of global variables has an important impact for the evolution of an application:

```
int i = 2;
```

### Object

Definition of an instance of a class. This is actually a specific case of variable where the type is a class:

```
Point p(1,2);
```

### Method (C++ only)

A method (or member function) is a function defined in the context of a class and which can be used (called) only on an instance of a class. The interface of a method is declared as part of a class definition, but its implementation is usually realised outside the class, in another source file.

```
class Point {
...
    void Plot();
...
};

void Point::Plot() {
    ...
}
```

### Attributes (C++ only)

An attribute is a variable defined in the context of a class. Every instance of a class has its own copy of this attribute. It can be used only with an instance of a class.

```
class Point {
    int _x;
    int _y;
    static int point_count;
};
```

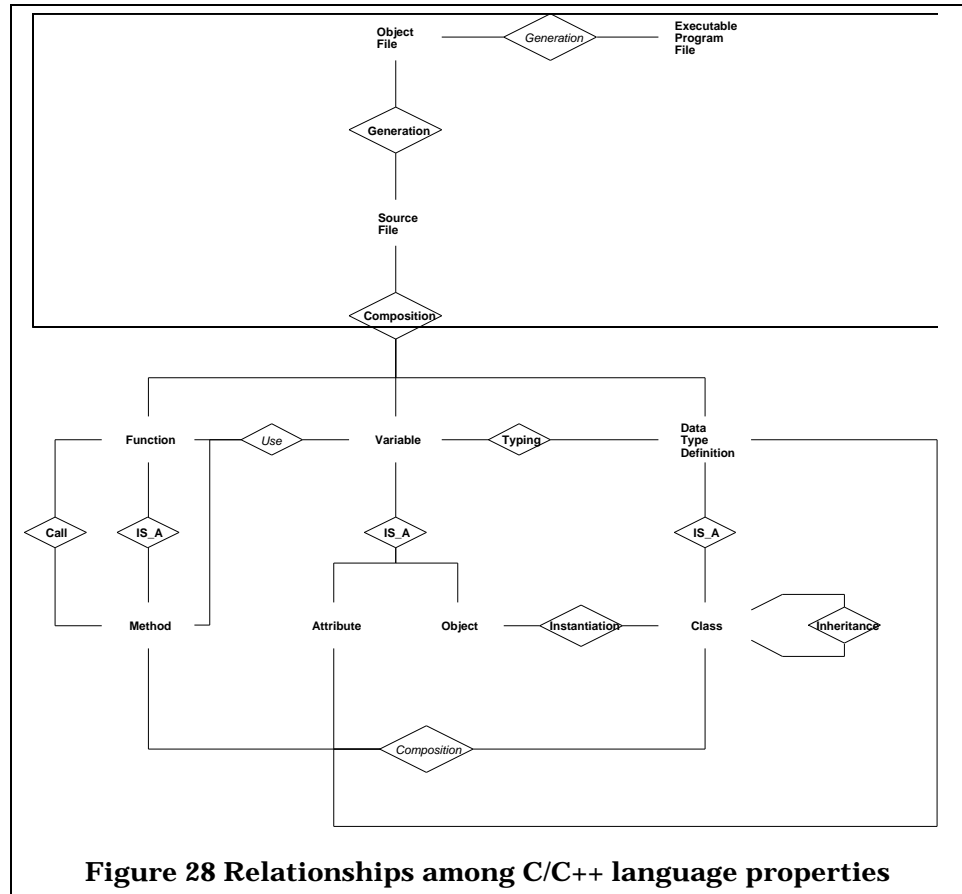
Relationships between these elements can be modelled as shown in Figure 28.

#### 3.2.1.4 Scripts properties

In addition to programs written using high-level programming languages, applications usually also consist of scripts. Scripts are often used as glue to call programs. They are usually based a simple language syntax (sequence of operating system command interface calls) but this syntax can in some cases be almost as structured as high-level programming languages.

Examples of common script languages are REXX, CLIST, JCL and shell scripts.

Their common properties of interest in the context of system evolution can be summarised as:



### Source File

File containing the definition of the script.

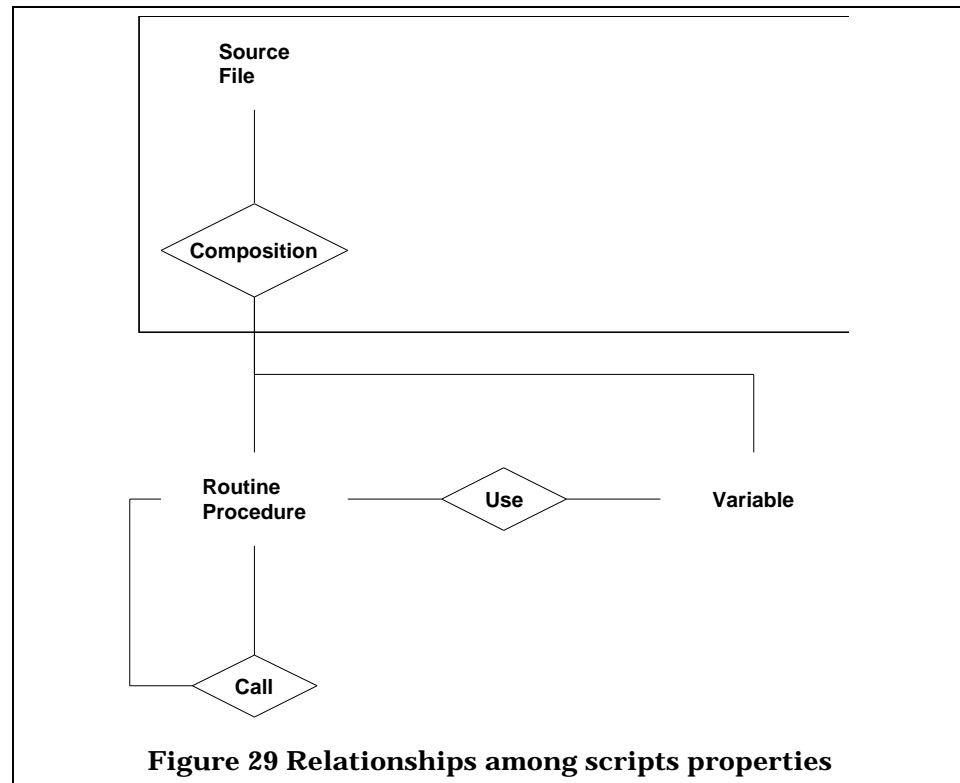
### Routine/Procedure

Routine or procedure defined in the script (if the script language allows it).

### Variable

Script variable.

Relationships between these elements can be modelled as in Figure 29.



### 3.2.1.5 Summary of 3GL properties

Several properties are common to all or several of the 3GL selected. Although no common model and properties can be defined for all 3GL since some 3GLs define specific properties, almost all 3GL can be described using properties listed in the table below:

Property	COBOL	FORTRAN	C/C++	Script
Source File	✓	✓	✓	✓
Executable Program File	✓	✓	✓	
Object File			✓	
Routine/Function	✓	✓	✓	✓
Variable	✓	✓	✓	✓
Data Type Declaration			✓	
Class			✓	
Method			✓	
Attribute			✓	
Object			✓	
Logical File	✓			
Physical File	✓	✓		

**Table 10 Summary of 3GL properties**

### 3.2.2 Special notations/diagrams to model 3GL applications

3GL languages define common properties mostly covered in UML. There is no need for special notations in UML. The following table shows how and where 3GL properties are represented using UML definitions and notations:

Property	UML Diagram	UML Element
Source File	Component	Component
Executable Program File	Component	Component
	Deployment	Component
Object File	Component	Component
Routine/Function	Object	Modelled as a Utility or part of it
	Sequence Collaboration	Implicitly represented by messages (messages can actually be a routine or a function call)
Variable	Object	Modelled as a Utility or part of it
Data Type Declaration	Class Object	Modelled as a Utility or part of it
Class	Class Object	Class
Method	Class	Method of a class
	Sequence Collaboration	Implicitly represented by messages (messages can actually be a method call)
Attribute	Class	Attribute of a class
Object	Object	Instance object
Logical File	Object	Modelled as a Utility or part of it (as an ordinary variable)
Physical File	Deployment	Component

**Table 11 Mapping 3GL property to UML**

### 3.2.3 Modelling 3GL applications

In this section, we focus on the two main modelling activities applied in the context of system evolution:

- Modelling a legacy 3GL system
- Modelling a new system from a legacy system

This section assumes that the reader is already familiar with modelling techniques and it addresses only specific issues relevant to system evolution.

#### 3.2.3.1 Modelling a legacy 3GL system

There are two main facets when modelling an existing system:

- Static modelling
- Functional and dynamic modelling

Functional and dynamic modelling should be done after static modelling since it will rely on information collected during this activity.

### Static modelling

Static modelling consists of modelling the static part of the system, its structure and modelling its environment.

Static modelling is done starting from files defining the configuration of an application if they are available (library definitions, makefiles...). During this first step, the *inventory* of all files composing the system is done:

- source files (in programming languages or script languages)
- executable programs
- intermediate files created during the generation process (object files...)
- auxiliary files (libraries, images, icons, messages, screen definitions...)
- configuration files
- database definition files
- data files
- documentation files
- and all other relevant files...

Even if some information collected above may be redundant, it is recommended that you should identify all information available. Indeed, due to missing documentation or lack of configuration management, generated files (intermediate files or executable files) may not be up to date. In this latter case, obtaining accurate information about the running (or last) version of the system may be difficult or impossible, but this information can often be considered as valid as far as a high level of information is concerned.

In addition to this static information, interviews of operators or developers of the system (when they are still available or even known!) may be a very valuable source of information, even if it is informal.

Once this inventory is done, all relevant files must be collected and stored in a *repository*. For instance, several files that may be stored on a remote, library machine must be uploaded to a working host where investigation and modelling can be performed. The environment or tool used as repository must be selected carefully. Depending on the nature and size of the system to be modelled and the level of information to extract, it may have to handle a high number of entities and relationships:

- > 10,000 files
- > 1,000,000 lines of code
- > 100,000 entities (routines, data declaration...)
- > 100,000 links between files and between entities

In addition, in order to manage links and dependencies between files, it is recommended that the repository supports configuration management.

In addition, if the system or part of it is reused in the evolution plan, the repository must support system version management to follow the modifications from the old legacy system to the new one.

Once files composing the system have been identified and collected, the static structure of the existing system can be obtained in a straight forward way using reverse-engineering tools or manually. Even if powerful reverse engineering and code analyser tools are not available, information can be retrieved in a semi-automated way by creating and applying scripts identifying the main keywords or patterns in files (e.g. identification of all keywords SUBROUTINE and FUNCTION in a FORTRAN source file).

This analysis can remain at a high level in a first step and only context modelling can be performed. During this phase, the following diagrams can be composed:

- Extended Entity Relationships diagrams
- Block Diagrams
- Class and object diagrams
- Hardware and network diagrams
- Implementation diagrams

The identification and analysis of more detailed information can be done depending on the evolution decision:

- Discarded parts of the system do not necessarily need to be further analysed.
- Parts of the system reused as-is in a new system need only to be documented at a high level (interfaces, interactions with the environment) since they will often be reused as black boxes.
- Parts of the system re-engineered or redeveloped need detailed analysis.

Detailed analysis will produce the following diagrams:

- Class and object diagrams

### **Functional and dynamic modelling**

Functional and dynamic modelling deals with the dynamic part of the system, i.e. how the system is running, how the entities composing it interacts and collaborate. This modelling relies on the identification of the static structure of the system.

This analysis can remain at a high level in a first step and only context modelling can be performed. Then, depending on the evolution path selected for the system and its components, this dynamic analysis can be done at a more detailed level. The following diagrams can be composed:

- Operational schemas
- Use cases
- Data flow diagrams
- Sequence diagrams

- Collaboration diagrams
- State diagrams
- Activity diagrams

Recommendations for building such diagrams can be found in the corresponding sections of this document.

Functional and dynamic modelling is much more complex than static modelling since basic information necessary is scattered among many files. Reverse-engineering tools do not automate this modelling. They rather provide some functions to help navigate in the static structure of the system (cross-reference, navigate following call links...). Unfortunately, tracking the execution path of a system is complex and is usually not captured by re-engineering tools.

Even if there is no magic rule for modelling the functional and dynamic information of a system, the following recommendations can be applied:

- Decompose the system into major sub-systems before proceeding with every sub-system.
- Proceed in a top-down way starting from the visible functions provided by the system (user interface, batch programs called...) leading to utility functions.
- Limit the complexity of diagrams by providing guidelines on how to read them (as in sequence or collaboration diagrams) or by limiting the number of entities involved (in use cases or state diagrams).
- Fine grain dynamic analysis must be limited to critical areas of the system. Informal high-level description can be more understandable than a complex diagram.
- Depending on how a part of a system will be (re-)used, modelling should focus on the interactions between entities or on their internals.
- Do not model what is not strictly necessary to be modelled!

### 3.2.3.2 Modelling a new system from a legacy system

When a new system is built based on a legacy system or part of it, modelling guidelines can be used to help understand its evolution. This section does not address generic modelling guidelines for building systems but rather how to derive diagrams for the legacy system into diagrams for the new one.

To understand the evolution of a system, it must be easy to identify what parts of the legacy system are reused and where in the new system. In this respect, only high-level diagrams (context modelling diagrams, and especially block diagrams used to model the main building blocks of the legacy system) are concerned. Modelling high level diagrams for the new system should be based on equivalent diagrams modelling the legacy system, and by highlighting modifications done to integrate or include them.

For instance:

- new operational diagrams should highlight which new tasks are supported by the information system,

- new block diagrams should include both new application parts and legacy application parts,
- hardware/network diagrams should show how old and new computers and devices are connected.

This will not only ensure that diagrams can be reused to help and understand the technical evolution of the system, but it also ensures that modelling allows the identification of reusable parts of the system.

Reusing legacy system components sometimes needs encapsulation or integration techniques. As far as high-level modelling is concerned, this detailed technical information should not be represented. Specific diagrams should rather be built to explain how legacy components are encapsulated or integrated in the new system from a generic angle.

### 3.3 Technical Modelling of 4GL applications

In the literature you do not find much about modelling 4GLs. Often 4GLs have a drag and drop approach where modelling and planning is not necessary. But practise shows that large enterprise-wide 4GL applications need planning and modelling in the same way as 3GL applications. In this section we first try to identify the key concepts and properties of 4GLs which should be modelled. Afterwards we show how we can model them with UML. For system evolution two technical models are interesting, the technical model of the current system and the technical model of the target system. Both models are described in own subsections. Reverse-engineering of 4GLs is an important aspect for extracting information out of the current system. Giving guidelines for structuring the target system completes the section.

4GLs are characterised by a visual development environment and support for database access. In addition they provide high level programming support and nowadays mostly OO and application partitioning features. Following (Ovum Ltd., 1996) 4GLs can be classified into 3 generations.

- The first generation provides means for building GUIs with database access on SQL level. They are typically used to build 2-tier client/server architectures with fat clients.
- The second generation 4GLs provides an own information model decoupling the application logic from the underlying database system.
- The third generation 4GLs provides additional means for a flexible application partitioning. They are well suited for building n-tier client/server architectures.

It must be noted that the borders between these 3 generations are fluid and the whole 4GL market is dynamically changing. Another fact is that some 4GLs support partitioning but no own information modelling. Table 12 is extracted and updated from an OVUM study and shows how important 4GLs can be classified. Main characteristics are drawn bold.

4GL	1 <sup>st</sup> generation GUI building	2 <sup>nd</sup> generation Modelling	3 <sup>rd</sup> generation Partitioning
Centura/SQLWindows	✓		✓
Compuware/unifAce	✓	✓	✓
Sybase/PowerBuilder	✓		✓
Seer	✓		✓
Magic/Magic	✓	✓	✓
Forte/Forte	✓		✓
Oracle/Developer2000	✓	✓	
Informix/NewEra	✓	✓	✓

**Table 12 Classification of 4GLs**

- Beside SQLWindows we have studied PowerBuilder and unifAce applications to identify the main properties of other languages. The results can be find in the summary table at the end of the document.
- Although the used terminology may be different, the supported concepts are fundamentally the same. The added code examples illustrate the concepts and help matching the concrete terminology.

### 3.3.1 The properties of 4GL applications

As the 4GL market is dynamically changing and there is no standard we have worked out the properties of 4GLs in the following way:

- With SQLWindows/Centura we have investigated the properties of one typical 4GL in detail. SQLWindows is a classic 1<sup>st</sup> generation 4GL with very good OO features. The new version of SQLWindows - now called Centura - additionally provides means for application partitioning. So it is on the way providing 3<sup>rd</sup> generation features. The study of the properties is complemented by a meta-model of the language. This model shows the major entities of the language and how they are related.
- To allow a better understanding of the identified SQLWindows properties we have added code examples to the identified properties.
- As 4GL and also 3GL based client/server applications are mainly based on relational database management systems we have described which element of databases must be modelled.
- Although the used terminology may be different, the supported concepts are fundamentally the same. The added code examples illustrate the concepts and help matching the concrete terminology.
- This section is completed by a summary (Table 14) summarising the major properties of 4GLs and database systems.

#### 3.3.1.1 SQLWindows properties

In the following you find a list of properties which have to be modelled, at times with code examples to clarify the implementation of a certain property in SQL/Windows.

### Files and file dependencies

The source code is stored in so-called libraries which can be nested.

Using the include mechanism one library incorporates at design time statically other SQLWindows libraries. There are two methods in including libraries:

- File include, including all components from the library without any selection, and
- Select from, allowing to selectively include individual components from the library

```
Libraries
  File Include: base.apl
  Select From: koste.apl
  Class Definitions
  ...
  Internal Function: fl
  ...
```

The library concept is a necessary feature for any medium to large size development project. It ensures a modular structure and is necessary for work distribution and code sharing.

An overview about the participating files and their dependencies is an important aspect which must be modelled.

### Format descriptions and references

A format can be applied to objects such as data fields and table window columns. The format determines how the data is displayed.

In the FORMATS section of Global Declarations formats can be defined.

```
Formats
  Date/Time: dd-MMM-yyyy
```

Formatting is related to validation, the process of ensuring that the data entered fulfils certain conditions. Validation can be enabled/customised via using „Input masks“. These reference the FORMATS section. SQLWindows performs default validation for number and date/time data types.

The way data is displayed and in which format respectively fulfilling which constraints it is entered provides an essential part of the user/application-interface and therefore must be modelled. Also it should be documented where which format is used.

### External functions and references

By using DLLs and the Windows API one can extend the SQLWindows environment. The EXTERNAL FUNCTIONS section defines the DLLs to be accessed.

```

External Functions
:
Library name: vt50.dll
Function: VisWinSetMeter
Description: nErrCode VisWinSetMeter( hWnd, nPercent )
Export Ordinal: 44
Returns
Number: INT
Parameters
Window Handle: HWND
Number: INT

```

The external function can now be called as any other function defined in the GLOBAL DECLARATIONS section of the application or any command accessible in SQLWindows.

It should be modelled which external functions are used and from which file they are coming. The purpose should be described as well as the interface (parameters, return values). To analyse impacts of changes a cross reference list summarising all usage will be helpful.

### Constants and references

Constants can only be declared in the GLOBAL DECLARATIONS section. They can be referred to wherever a variable can be referred to.

As being of application-wide availability there is need for some thinking about which constants to declare and where to use them. This must be documented.

### Resource definitions and references

Resources let you specify bitmaps, icons or cursors in the GLOBAL DECLARATIONS section which can be referenced application-wide.

```

Global Declarations
:
Resources
Bitmap: resOk
File Name: ok.bmp
Icon: Icon1
File Name: icon1.ico
Cursor: Curl
File Name: curl.cur

Pushbutton: pbOK
:
Picture File Name:
:
Message Actions
:
On SAM_Create
! Set the picture for the push button
! using the resource
Call SalPicSet( hWndItem, resOk, PIC_FormatBitmap )

```

The picture contents of a window object can be specified in the picture file name section. This may be avoided, because by making an „\*.exe“ or „\*.run“ version of the application, SQL/Windows copies the resources from the external files into the application.

It must be modelled which resources are defined and where they are used.

### **Variable definitions and references**

Variables can be declared in these places:

- Global Declarations (Variables section)
- Class, Internal and Window Functions (Parameters, Static Variables and Local Variables section)
- External Functions (Returns and Parameters section)
- Class Definitions (Class Variables and Instance Variables section)
- Form Windows (Window Parameters and Window Variables section)
- Top-level table windows (Window Parameters and Window Variables section)
- Top-level Quest windows (Window Parameters and Window Variables section)
- Child table windows (Window Variables section)
- Child Quest windows (Window Variables section)
- MDI windows (Window Parameters and Window Variables section)
- Dialog Box (Window Parameters and Window Variables section)

User-defined variables can be defined in the CLASSES-section as functional class and can be used like any other variable. Variables can be referred to in qualified and unqualified manner.

There is some need to model and document:

- Initialisation of a variable,
- the objective of a variable,
- its range,
- which functions make use of it and
- in which order this happens.

### **Functions and references**

A function performs a specific task.

It should be modelled:

- which task a certain function performs (its purpose),
- its interface (parameters, return values),
- which global variables, constants etc. it uses,
- which functions and messages it calls and
- where it is called.

Special attention while modelling is paid to parameters and returns, providing a functions interface to the environment, and static variables, surpassing an objects lifetime.

```

Function: MyFunction
  Description: <Purpose of the function>
  Returns:
    Boolean:
  Parameters:
    Date/Time: dtThisDay
    Receive Number: nNumb
  Static Variables:
    Long String: lsStay
  Local Variables:
    Boolean: bOk
    Number: nUseMe
  Actions:
    Call DoSomething()
    :

```

### Messages and references

In 3GLs such as C++, objects can only communicate by explicit function calls. 4GLs like SQLWindows support communication by sending messages from one object to another. The message name is a constant that represents a number. Further details are described in the meta-model section.

Messages are defined in the CONSTANTS section of Global Declarations. A message can be referred to by its name or its number.

```

Data Field: df1
:
  Pushbutton: pb1
  :
  Background Color: Default
  Message Actions
    On SAM_Click
      Set df1= Hello World

```

It should be modelled:

- which messages are handled by an object or class,
- which objects interact by sending and receiving messages,
- the action an object performs on receiving a message,
- which internal action an object causes to send a message.

### Named Menus and references

Named menus are used to create popup menus that windows in an application can share. They must be declared in the NAMED MENU section in Global Declarations.

Example: the from many window applications well known Edit menu: in particular you find the menu items Undo, Cut, Copy, Paste and Clear. The item Cut is expanded to show details.

```

Named Menu
Menu: menuEdit
  Description: Edit menu without OLE support.
  Title: &Edit
  Enabled when:
  Status Text: Undo, Cut, Copy, Paste, Clear
  Menu Item: &Undo
  Menu Separator
  Menu Item: Cu&t
    Resource Id: 52453
    Menu Item Name:
    Status Text: Cuts the selection and puts it on
                  the clipboard
    Keyboard Accelerator: Shift+Del
  Menu Settings
    Enabled when: SalEditCanCut()
    Checked when:
  Menu Actions
    Call SalEditCut()
  Menu Item: &Copy
  Menu Item: &Paste
  Menu Item: &Clear

```

To use a named menu, specify its name in the menu definition of a window:

```

Menu
  Popup Menu: &File
  Named Menu: menuEdit

```

A Windows Menu can refer to any of its own named menus or any global named menus. An MDI child window can also refer to named menus in its MDI window parent.

It should be modelled which named menus are modelled and what are their purposes. In addition it should be described who is using a named menu and what a named menu itself is using.

### Classes

Classes are one important feature of SQLWindows. One class can be derived from one or more other classes. The interface of a class consists of functions and messages. Window classes can contain child windows. Although 3GLs know different mechanisms to encapsulate variables by declaring them as private, protected or public, SQLWindows provides no such possibilities.

It must be modelled:

- which classes exist and which objectives they have,
- the inheritance hierarchy,
- the interface (in detail),
- cross references where a class is used/instantiated and where class functions or messages are used,
- parent-child window relationships,
- variables defined in a class and their usage (description only).

```

Form Window Class: cABCDataForm
Description: Form window class to display a RowData.
Derived From
    Class: cABCServiceForm
Menu
Tool Bar
Contents
    ! buttons that will be visible in 'new'- and 'edit'-mode
    Pushbutton: pbApply
    Pushbutton: pbDiscard
    ! 'exit'-button will only be visible in 'show'-mode
    Pushbutton: pbClose
    Line
Class Variables
Instance Variables
    Window Handle: hWndMyWFParent
    Number: nParentWFType
    Number: nErrInvalidItem
    Window Handle: hWndFocusedItem
    :
Functions
    ! Function that redefine cABCServiceForm base functions:
    Function: IsConnectOk
    Function: Activate
    ! New functionality, never necessary to redefine:
    Function: dlg_IsUserInputEnabled
    :
Message Actions
    On NTFY_WorkflowClose
    On NTFY_Transaction
    On SAM_DIALOG_EndDialog
    On SAM_DIALOG_IsIdle
    :

```

## Objects

In SQLWindows an object can be a class instantiation or an independent object, which means that the object is described directly without „blueprint“. If an object is a class instance only this should be modelled together with possible re-definitions of features. If an object is independent the same features as for classes must be modelled as far, as is relevant for objects.

## User interface

Communication with the user is performed via Graphical User Interfaces (GUI). From an abstract view a GUI is a set of windows or forms. Windows can be opened simultaneously and parent windows can contain child windows. The windows interact with the user by processing mouse actions and keyboard entries. Application functionality is performed via function calls and messaging.

The GUI properties to be modelled and documented are:

- which windows exist,
- their purpose,
- their layout (for ergonomically reasons),
- the window activation sequence,
- the linkage between GUI elements and application functionality.

### **Database interface**

To communicate with the database, SQLWindows provides a set of predefined SQL-functions to execute SQL-strings.

It will be important to model which database elements are manipulated and where this happens. More details are described in the RDBMS properties section.

### **Subsystems**

For large SQLWindows applications it is necessary to have a subsystem structure above the programming level. The subsystems divide the program elements into manageable units. A nesting of subsystems is possible.

It must be described

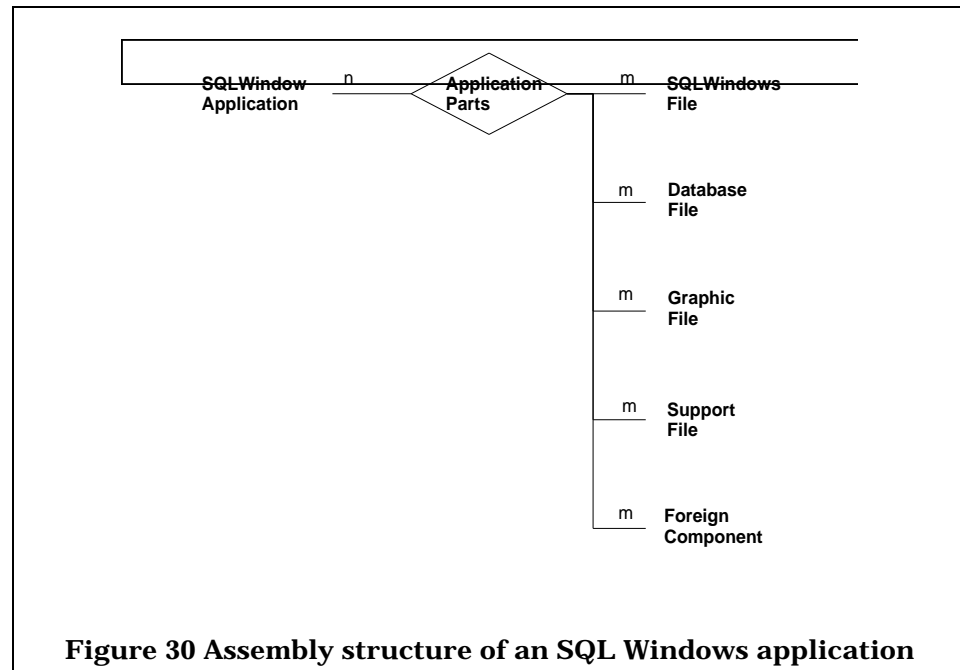
- which subsystems are available,
- their purposes and objectives,
- their interfaces and
- dependencies.

#### **3.3.1.2 A Meta-Model of SQLWindows**

This section contains a meta-model of SQLWindows. The objective of this model is to describe and summarise the main entities and relations which are important to modelling. Entities of local scope are ignored to concentrate on the important parts. As notation we have used the extended entity relationship diagrams introduced in the context modelling chapter. The diagrams are created with the tool ProMod<sup>PLUS</sup>.

### **Files**

An application consists of a number of SQLWindows files or libraries. One library can be used in several applications. Beside the SQLWindows files, the application consists of database files containing database definition information, graphic files for bitmaps and icons, and support files, i.e. help files and configuration files. Further files contains foreign components used to build the application. An example for this are DLL files containing function libraries.



A SQLWindows file is an application file with the suffix '.app' or a library file with the suffix '.apl'. On physical level both types are the same and the extensions are conventions. A SQLWindows file consists of 5 parts:

- a description section for describing the library or application,
- a design-time setting section,
- a library section,
- a global declaration section, and
- several object definitions.

The library section is used to reference other SQLWindows files where the reference can be total or selective. One library can be referenced by many other files. Multiple inclusion levels are possible.

The two biggest parts are the global declaration section and the object declaration section. The global declaration section contains

- default settings,
- format descriptions,
- external function definitions,
- constant definitions,
- resource definitions,
- global variable definitions,
- internal function definitions,
- named menu definitions,
- class definitions and
- application message definitions.

The constants, functions, variables, classes, etc. defined in the global declaration section are visible in every library including the defining library. This means they have a global scope and their scope is not limited to a special file. External functions are functions imported from other DLLs.

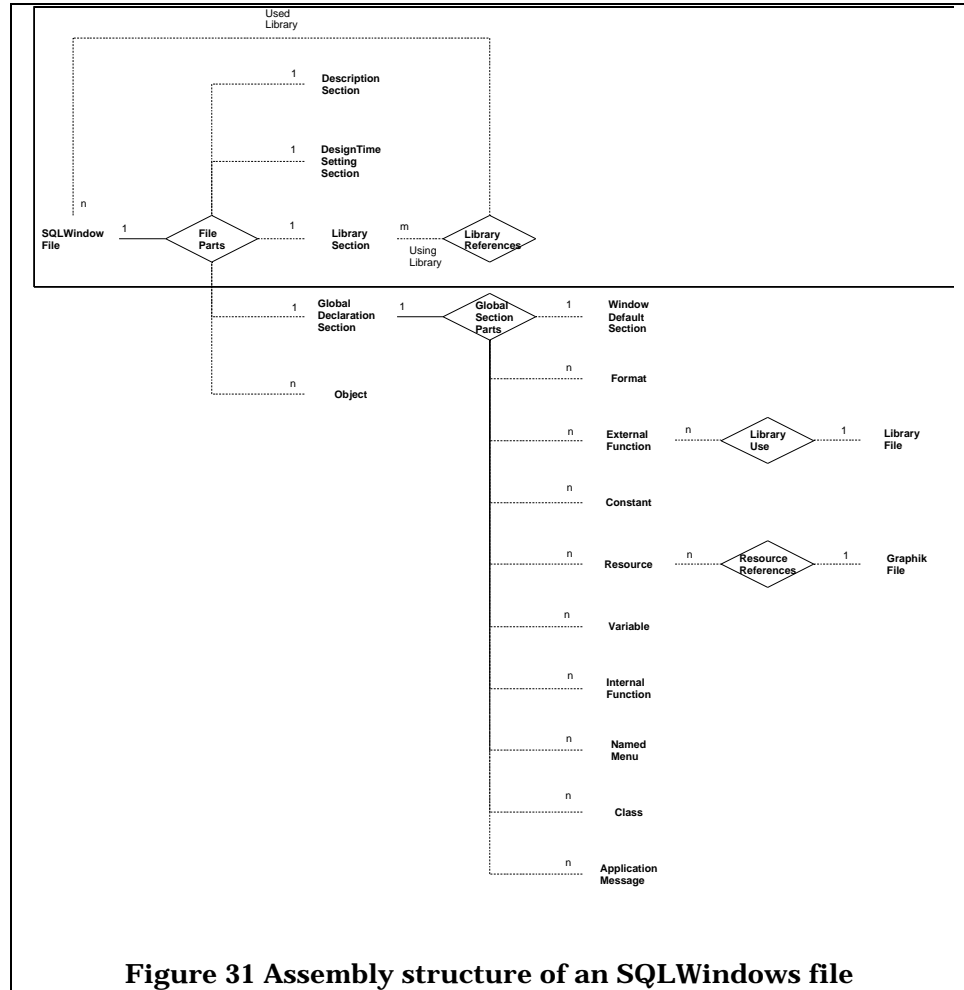


Figure 31 Assembly structure of an SQLWindows file

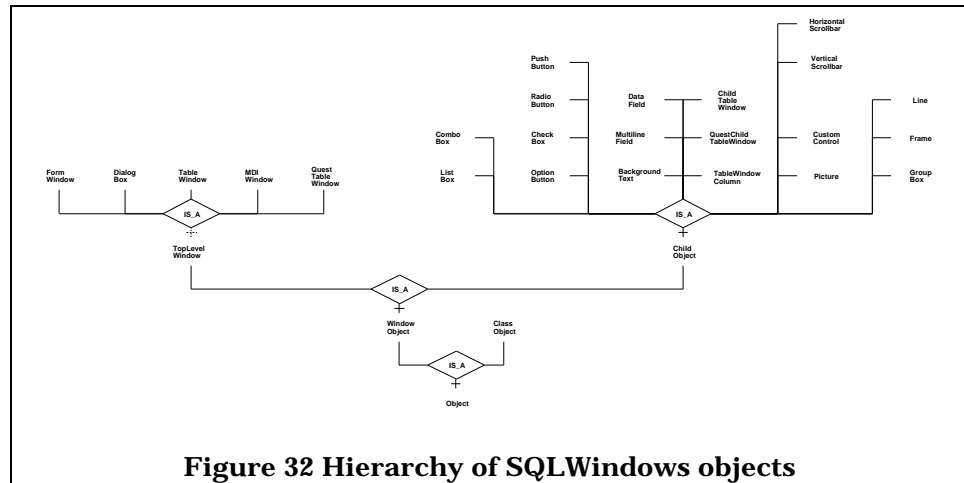
## Objects

Objects can be

- class objects or
- window objects.

Class objects are instantiations of classes and inherit their attributes and behaviour. Object specific overwriting is possible.

Window objects are standard objects provided by SQLWindows. The different types of window objects can be seen in Figure 32.



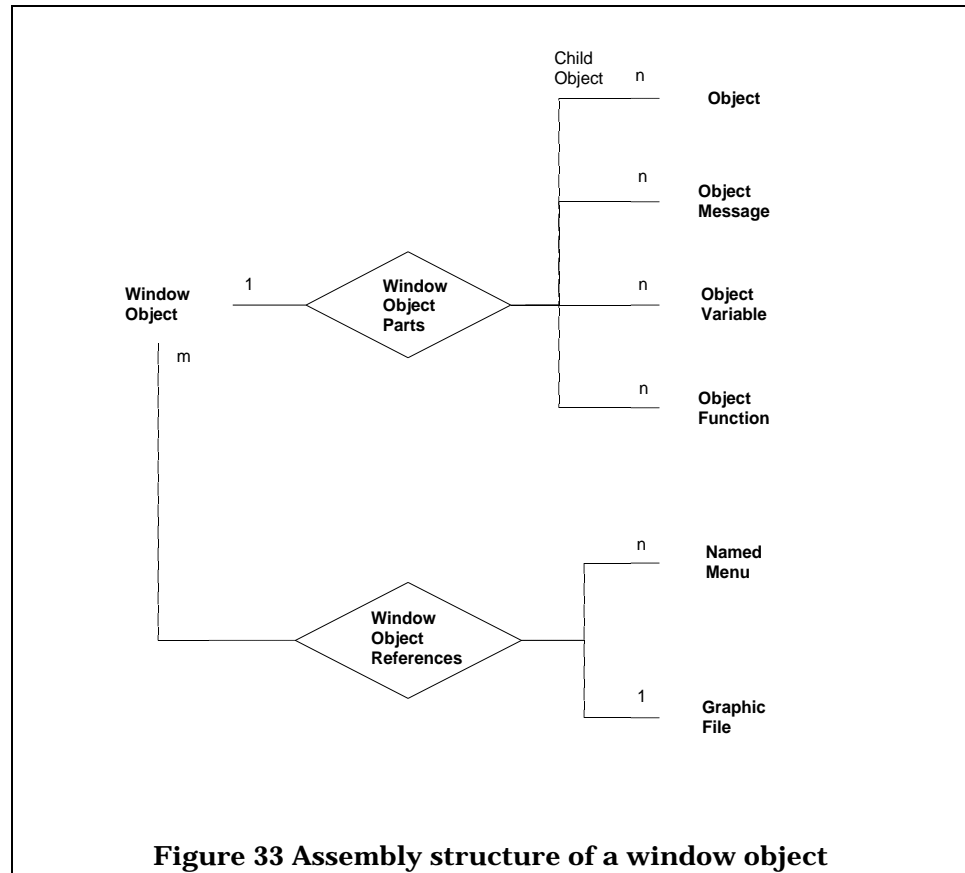
A window object consists of

- child objects,
- object messages,
- object variables and
- object functions.

Window objects can reference

- named menus,
- resource and format definitions and
- graphic files.

The referenced menus, resources, formats and files can be referenced from several objects.



### Classes

A class can be derived from one or more other classes. SQLWindows provides three types of classes:

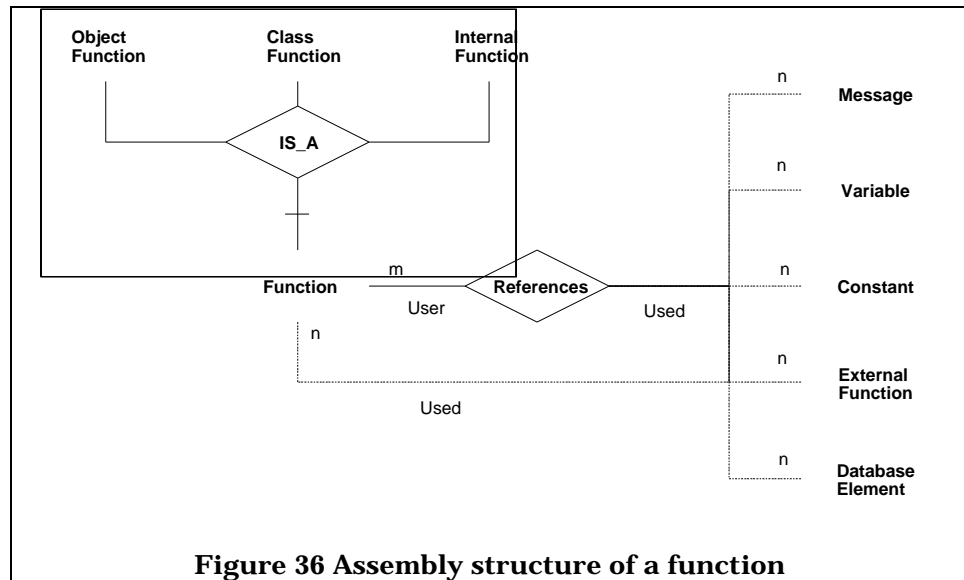
- functional classes, containing class variables, instance variables and class functions,
- general window classes, containing additionally class messages, and
- window classes for GUI building, additionally containing child objects and referencing named menus and graphic files.

Window classes provide the same features for building user interfaces as window objects. As a consequence the same hierarchy of window classes exists.



- an internal function.

The structure of a function is the same. A function can use or reference other functions or messages. A function itself can be used from multiple other functions or messages. In addition a function can use global variables, constants and external functions. Variables can also be defined locally but the local variables are not of big interest for modelling because they have a local scope and their impacts are limited. As the SQLWindows application works on a database, database elements can be used inside a function by special calls.



### Messages

The structure of messages is the same as the structure of functions. Differences are:

- messages do not contain local variables,
- messages carry two predefined parameters whereas the parameter interface of functions is user definable,
- message calls are always late-bound whereas function calls can be explicitly specified as late or early bound (default early bound),
- messages provide no return value,
- messages can be sent simultaneously to several objects.

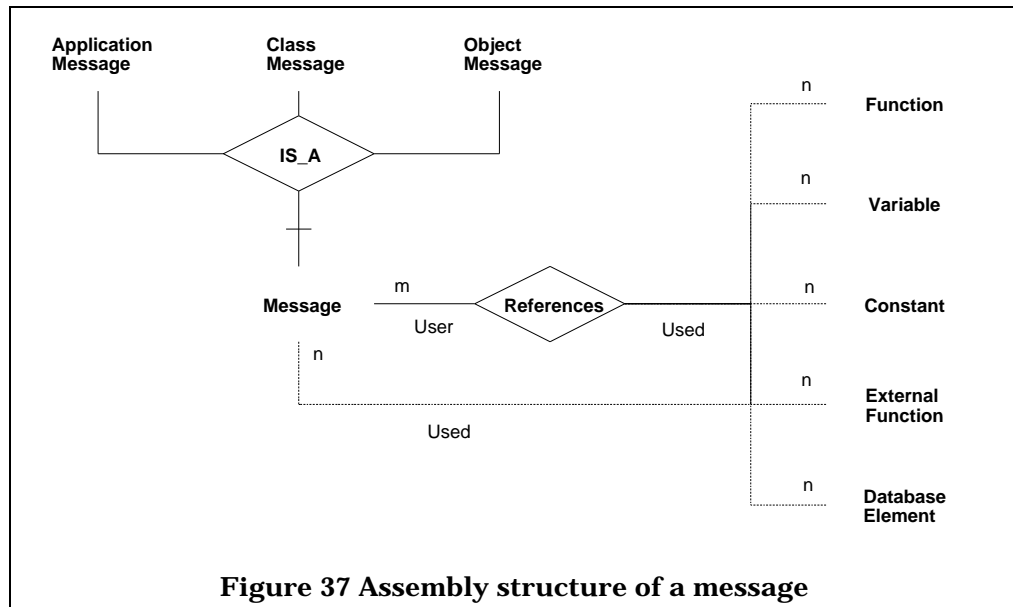


Figure 37 Assembly structure of a message

### 3.3.1.3 The properties of RDBMSs

4GL applications are based mainly on relational database management systems. In this section we describe the properties of RDBMS which are important for the modelling, implementation and evolution of an application. The identified properties are also relevant if the application is written with a 3GL.

When modelling RDBMS two things are important:

- the structure of the database, the elements and their relations and
- the application access to these elements.

To provide an appropriate evolution support the model must help to plan changes and analyse the impacts. If we change the type of an attribute we must know which other database elements (views, stored procedures) are regarded and which parts of the application are accessing this attribute and must therefore be adapted.

The features of current database systems are far more than standard SQL and proprietary elements. The following key elements are common:

#### Tables and references

Tables are the main building block of RDBMSs. A table consists of rows and columns. A row represents a unique set of data belonging together. A column represents an attribute. It will be important to model which tables exist, their purpose and their attributes.

Tables can be used by other database elements, i.e. views, stored procedures, triggers, etc. Information about these usage is important for maintenance. In addition most RDBMS supports table-to-table relationships, i.e. foreign keys, whose modelling is of high importance.

The most difficult but also most important thing to model is the application access to the tables. It is important to know when a table will

be accessed by the application. To be more precise we can also model which attributes are affected and what kind of access happens (read-only; read-write).

```
CREATE TABLE PRESIDENT
(PRES_NAME          VARCHAR(20)          NOT NULL,
 BIRTH_DATE         TIMESTAMP, YRS_SERV  INTEGER,
 DEATH_AGE          INTEGER,
 ELECTION_YEAR      INTEGER,
 PARTY              VARCHAR (20),
 STATE_BORN         VARCHAR(20));
```

### Indexes

An index is an access path to a table. The structure of the index must be modelled for performance reasons and which part of the application uses the index.

```
CREATE INDEX X_ELECTION_YEAR
ON PRESIDENT (ELECTION_YEAR);
```

### Views and references

A view is an alternative way of representing data that exist in one or more tables. From the modelling perspective a view looks like a table and can be used in the same way. So the same properties and references important for tables are also important for modelling views. In addition we must model on which tables (or views) and attributes the view is based.

```
CREATE VIEW WINNER_VIEW (NAME, YEAR_ELECTED, VOTES)
AS SELECT CANDIDATE, ELECTION_YEAR, VOTES
FROM ELECTION
WHERE WINNER_LOSER_INDIC = 'W';
```

### Triggers

A trigger is an event action pair. If a certain events happen a specified action will be performed. An event could be the update of a table, an action could be to delete another entry. Triggers are stored in the database itself and are performed automatically. This means the application does not need to regard them. For triggers it is important to model their purpose, the event starting it and the actions, i.e. manipulated tables, called stored procedures etc.

```
CREATE TRIGGER upd_quan
UPDATE OF quantity ON items
BEFORE (EXECUTE PROCEDURE upd_items)
```

### Stored procedures and references

Stored procedures are functions stored and executed in the database. For stored procedures their purpose and their interfaces (parameter, return values) must be modelled. Stored procedures can call other ones and manipulate tables and views. If global variables are possible they should be documented as well as their manipulation.

Comparable with tables and views the most important thing to model is the application access to the stored procedures. It is important to know who is calling a stored procedure.

```
CREATE PROCEDURE str_compare (str1 CHAR(20), str2 CHAR(20) )
RETURNING INT;
DEFINE result INT;

IF str1>str2 THEN
    result=1;
ELIF str2>str1 THEN
    result=-1;
ELSE
    result=0;
END IF
RETURN result;
END PROCEDURE -- str_compare
```

### Domains and references

Domains are type definitions, which can be used for attribute definitions. It must be modelled which domains are available and where they are used.

#### 3.3.1.4 Summary of 4GL properties

Property	SQLWindows
Subsystems	✓
Files	✓
File dependencies	✓
Format descriptions and references	✓
External functions and references	✓
Constants and references	✓
Resource definitions and references	✓
Variable definitions and references	✓
Functions and references	✓
Messages and references	✓
Named menus and references	✓
Classes	✓
Objects	✓
User Interface	✓
Database interface	✓

**Table 13 Properties of 4GL**

Property	RDBMS
Subsystems	✓
Files	✓

File dependencies	✓
Tables and references	✓
Views and references	✓
Indexes	✓
Triggers	✓
Stored procedures and references	✓
Domains and references	✓

**Table 14 Properties of RDBMSs**

### 3.3.2 Notations to model 4GL applications

After investigating the core concepts of 4GLs we now have a list of 4GL properties which must be modelled. The next question is how can we model these properties in an appropriate way. Here we are not interested in creating a new proprietary notation for 4GL modelling. We have used the Unified Modelling Language (UML) to model the 4GLs. In this section we describe how the 4GL concepts and properties can be mapped to UML.

This section is structured in the following way:

- First we start directly with describing the basic principles of the 4GL - UML mapping and some notes explaining how we use UML.
- Afterwards a table shows the mapping between 4GL properties and UML elements that are used to model them.
- The section is completed by some examples.
- As we use the stereotype concept to tailor UML to the needs of 4GL we summarize the used stereotypes in Appendix C.

<b>Basic principles of the 4GL - UML mapping</b>	
✓	<b>The central notation are the UML class-diagrams. They are used to model the basic elements of our 4GL applications. They provide direct support for modelling the classes and objects of the 4GL applications.</b>
✓	<b>With the help of stereotypes the class-diagrams can be tailored to the special needs, i.e. we can use stereotyped classes to model format descriptions, resources etc. In addition stereotypes are used to indicate the 'type' of classes or objects, i.e. a class can be stereotyped as a 'pushbutton class' or a 'functional class'.</b>
✓	<b>In addition sequence-diagrams or collaboration diagrams are used as snapshots of typical dynamic interactions between objects.</b>
✓	<b>On the physical abstraction level component diagrams are used to model files and file dependencies.</b>
✓	<b>Subsystems are modelled as packages.</b>
✓	<b>The different database properties are modelled as stereotyped</b>

classes. This allows a good mapping between database elements and application elements using them.

- ✓ Complementing textual descriptions are used to specify and describe further details, i.e. the purpose of a class or the parameter interface of a function.

For a good understanding of the mapping and usage of UML the following notes are important:

- We have not used every kind of diagram and notation detail provided by UML. Instead we have started from the properties which have to be modelled. We have identified appropriate UML elements to model them.
- With respect to the first note, use-case diagrams, state diagrams and activity diagrams are currently not used. If future experience shows that there is a need for them, hints for how to use them will be added.
- Stereotypes are a very powerful UML concept. We have used them to tailor UML to a special domain, namely 4GL modelling. The used stereotypes and the implied classification are described later in this section.
- The stereotype concept is not fully implemented in every tool. If stereotypes are not supported, a pragmatic approach to indicate these types (meta-classes) should be used. If the type is clear no further indication is needed. If the type should be highlighted free text is a possible solution.
- Collaboration and sequence diagrams provide the same underlying information. It is a personal and cultural question which kind of diagrams are preferred and we will give no restrictive rule. Often one diagram can be transformed automatically into the other by the modelling tool. We have used sequence diagrams to describe interactions arranged in time sequence. Collaboration diagrams are used if the interaction is arranged around objects and if the time sequence must not be highlighted.
- There should be one package (subsystem) for the database elements. If necessary a decomposition of it is possible.

Table 15 shows the mapping between 4GL properties and UML elements that are used to model them. We also describe how the mapping can be operationalised by tailoring UML with the help of stereotypes. The first column contains the identified 4GL properties which must be modelled. The second column contains the supported diagrams. In the diagram column 'Class' means 'Class diagram' etc.. The third column contains the UML elements used to model the property and some hints to use stereotypes.

4GL Property	UML Diagram	UML Element
Subsystems	Class	Packages are used in class diagrams to model logical subsystems.
	Component	In component diagrams packages are used to model directories or groups of files.

4GL Property	UML Diagram	UML Element
Files	Component	Component items
File dependencies	Component	Dependency relation between component items.
Format descriptions and references	Class	Attributes of a special class collecting the format descriptions. The class itself is stereotyped as «format definitions». The attributes are stereotyped as «format».
External functions and references	Class	Operations of a special class collecting the external function definitions. The class itself is stereotyped as «external function definitions». The operations are stereotyped as «external function».
Global constants and references	Class	Attributes of a special class collecting the constant definitions. The class itself is stereotyped as «constant definitions». The attributes are stereotyped as «constants».
Resource definitions and references	Class	Attributes of a class collecting the resource definitions. The class itself is stereotyped as «resource definitions». The attributes are stereotyped as «resource».
Variable definitions and references	Class	Attributes of a class or object. The attributes are optionally stereotyped as «variable» . Global variables are modelled as attributes of a class collecting the global variable definitions. The class itself is stereotyped as «variable definitions».
Functions and references	Class, Sequence, Collaboration	Operations of a class or object. The operations are stereotyped as «function». Global functions are modelled as operations of a class collecting the global function definitions. The class itself is stereotyped as «internal function definition». Sequence diagrams or collaboration diagrams are used as snapshots of characteristic function interactions.
Messages and references	Class, Sequence, Collaboration	Operations of a class or object. The operations are stereotyped as «message». Application messages are modelled as operations of a class collecting the global message definitions. The class itself is stereotyped as «application message definitions». Sequence diagrams or collaboration diagrams are used as snapshots of characteristic message interactions.
Named menus and references	Class	Operations of a class collecting the resource definitions. The class itself is stereotyped as «named menu definitions». The operations are stereotyped as «named menu».
Classes	Class, Sequence, Collaboration	Class. The class-type (pushbutton-class, frame-window-class, ...) is described by an appropriate stereotype. The relationships between the classes are modelled with the corresponding UML notations, i.e. associations, compositions and generalisations. Sequence diagrams or collaboration diagrams are used as snapshots of characteristic interactions.
Objects	Class, Sequence,	Class. The object-type (pushbutton, frame-window, ...) is described by an appropriate stereotype. As SQLWindows

4GL Property	UML Diagram	UML Element
	Collaboration	objects can have own message and function definitions they are modelled as UML classes with appropriate stereotypes. If the 4GL objects are pure instantiations of classes they can be modelled as UML objects. Sequence diagrams or collaboration diagrams are used as snapshots of characteristic interactions.
User Interface	Collaboration	The user interface is modelled as a collaboration diagram. The windows are represented by the objects. The window activation is modelled by appropriate use-relations.
Database Interface	Class	The relations between application elements and database elements are described by use relations between the application classes and objects and the database classes.
Internal data model	Class	The internal data model is modelled as appropriate stereotyped classes in the same way like the RDBMS elements. The relations between the internal data model and the database is modelled by appropriate use dependencies.
Extended attributes	Class	Properties and behaviour connected to attributes is modelled by complementary textual descriptions.
Tables and references	Class	Class which is stereotyped as «table». The table attributes are represented by the class attributes.
Indexes	Class	Operation of the class representing the corresponding table or view. The operations are stereotyped as «index».
Views and references	Class	Class which is stereotyped as «view». The view attributes are represented by the class attributes.
Triggers	Collaboration	Triggers are modelled in a regarded diagram.
Stored procedures and references	Class, Collaboration, Sequence	Operations of a special class collecting the stored procedures. The class itself is stereotyped as «stored procedure definitions». The operations are stereotyped as «stored procedure». Sequence diagrams or collaboration diagrams are used as snapshots of characteristic stored procedure interactions.
Domains and references	Class	Attributes of a special class collecting the domain definitions. The class itself is stereotyped as «domain definitions». The attributes are stereotyped as domain».

**Table 15 The mapping between 4GL properties and UML elements**

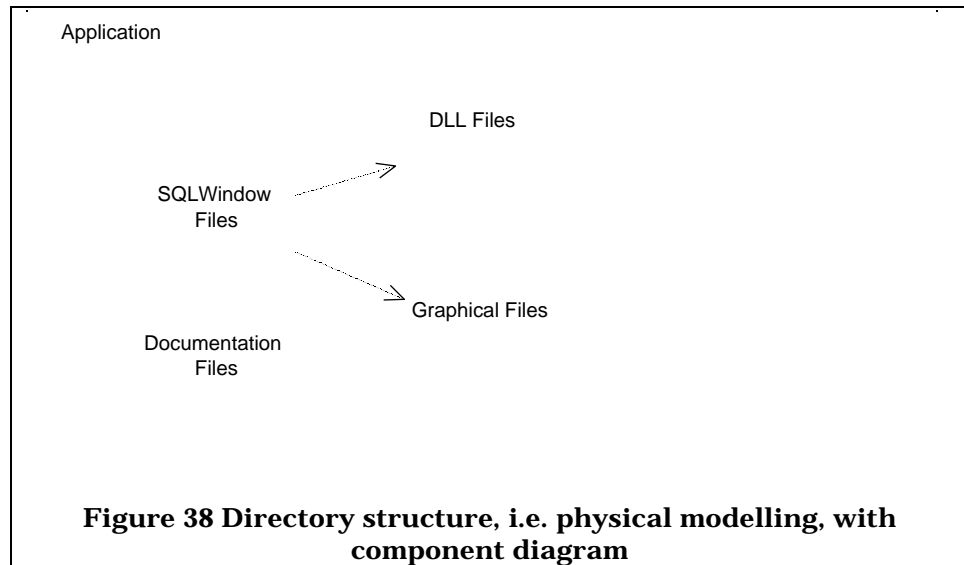
### Example diagrams

The following examples illustrate the described way of modelling. They are created with the demo-version of Rational-Rose. In some cases the tool doesn't support the method fully and we made pragmatic adaptations. The purpose of the examples is to illustrate the use of UML. The application content is neither complete nor representative.

The directory structure of an application is described with component diagrams. Package items represent directories. Dependency relationships (arrows) are used to describe who is using who. If there is an arrow from

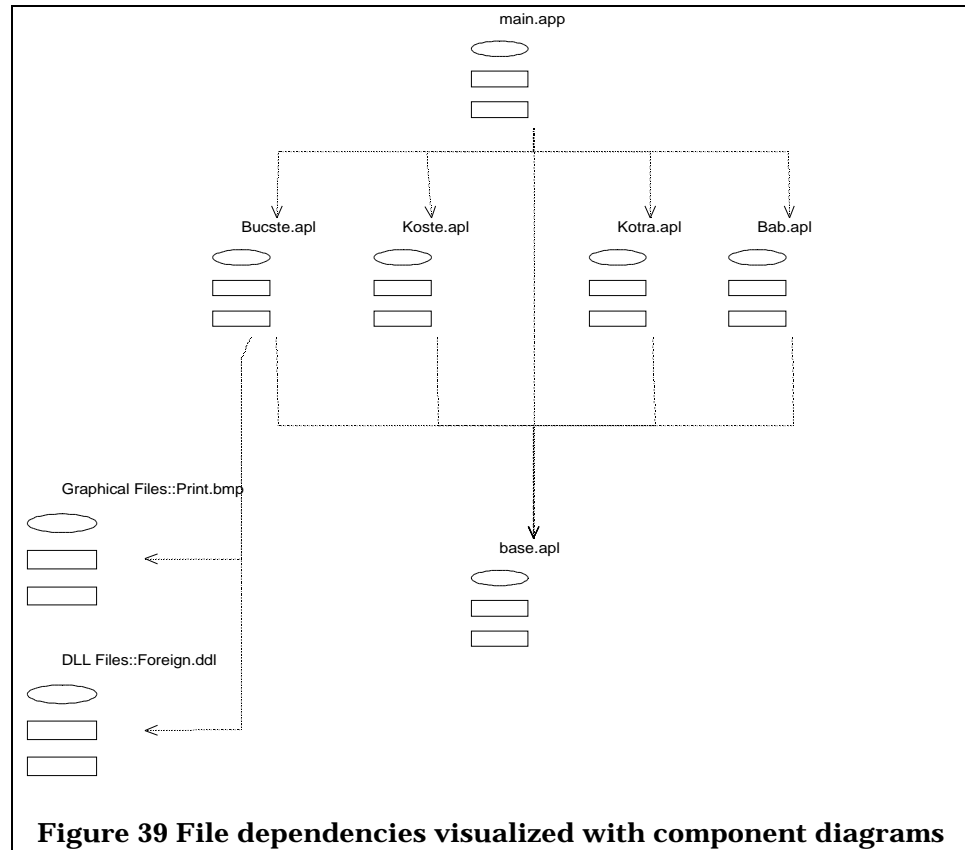
the package SQLWindows Files to DLL files, this means that one or several files from the SQLWindows package use one or several files from The DLL package.

In Figure 38 you find a main folder „application“ incorporating sub directories for SQLWindows files, DLL files, graphical files (bitmaps) and supporting documentation files. The SQLWindows files are using DLL files and Graphical files. The documentation files are independent.



The single files and their dependencies are described with component diagrams. The files are represented by components. The arrows represent the dependency-relationships between the files.

Figure 39 shows a refinement (decomposition) of the SQLWindows file package in the diagram above. Files which are used in this package but defined in another one are prefixed by their package's name, i.e. 'GraphicalFiles::print.bmp'.

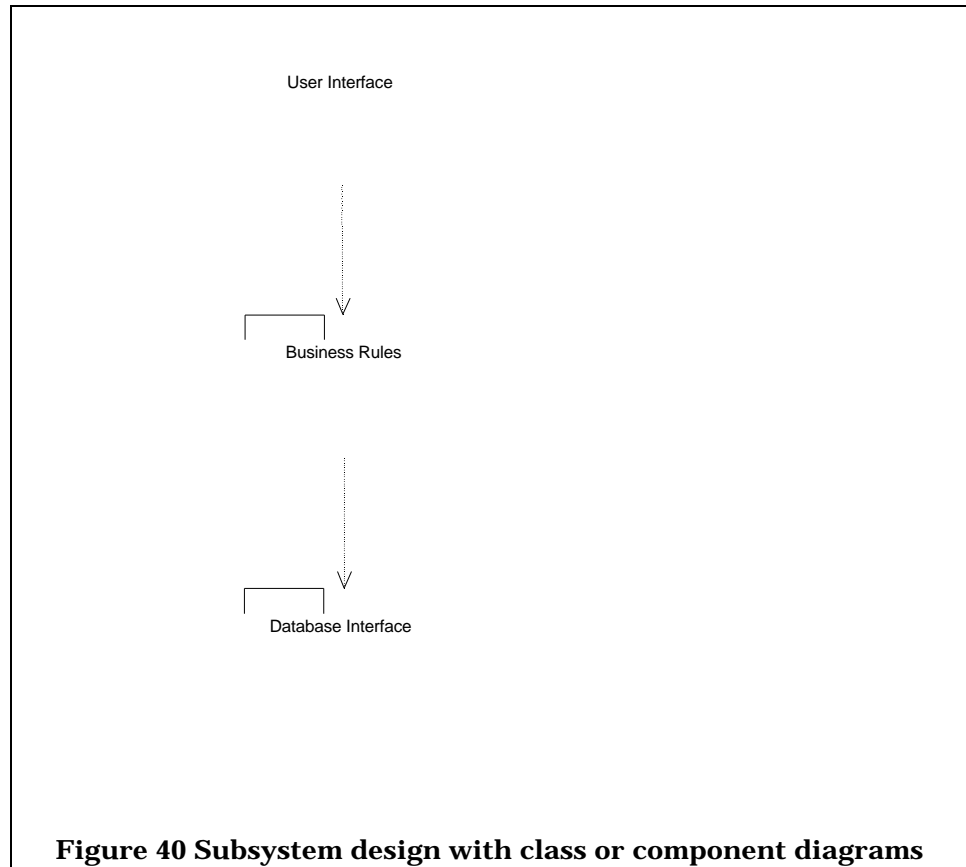


**Figure 39 File dependencies visualized with component diagrams**

The subsystem structure of the application is described in class diagrams. Packages are used for subsystems, use dependencies (arrows) to show which subsystem is using which other.

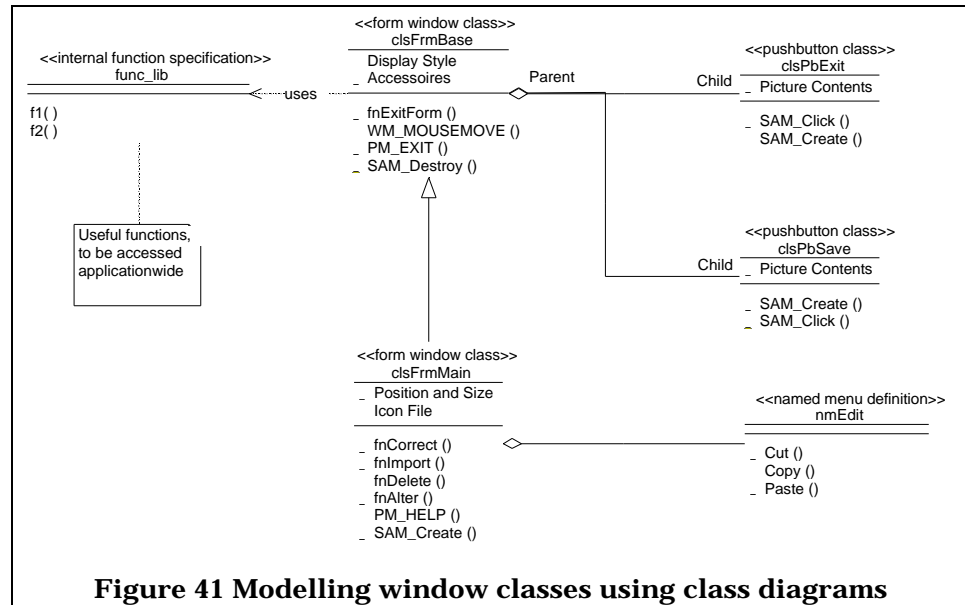
In Figure 40 we have three subsystems represented by packages named 'User Interface', 'Business Rules' and 'Database Interface'. The use dependencies indicate which subsystem is using which other one.

Class diagrams with packages and use dependencies are used to model the overall architecture of the software. They are an alternative to the block diagrams described in the conceptual modelling chapter with a more technical oriented style.

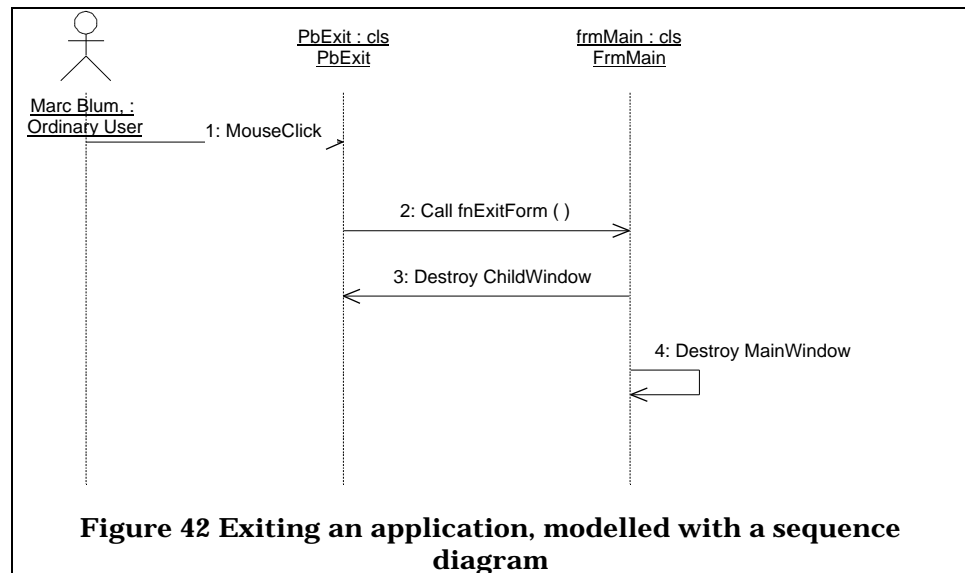


Class diagrams are used to design most of the 4GL properties.

In Figure 41 you find a form window class (clsFrmBase) which contains child window classes (pushbuttons) and uses global functions and constants as defined under global declarations. Another form window class (clsFrmMain) inherits all properties from clsFrmBase and additionally contains a named menu. Stereotypes are used throughout the example to depict window class types. The attribute part of a class contains class default settings. Functions and message actions of a class are listed in the operations part.



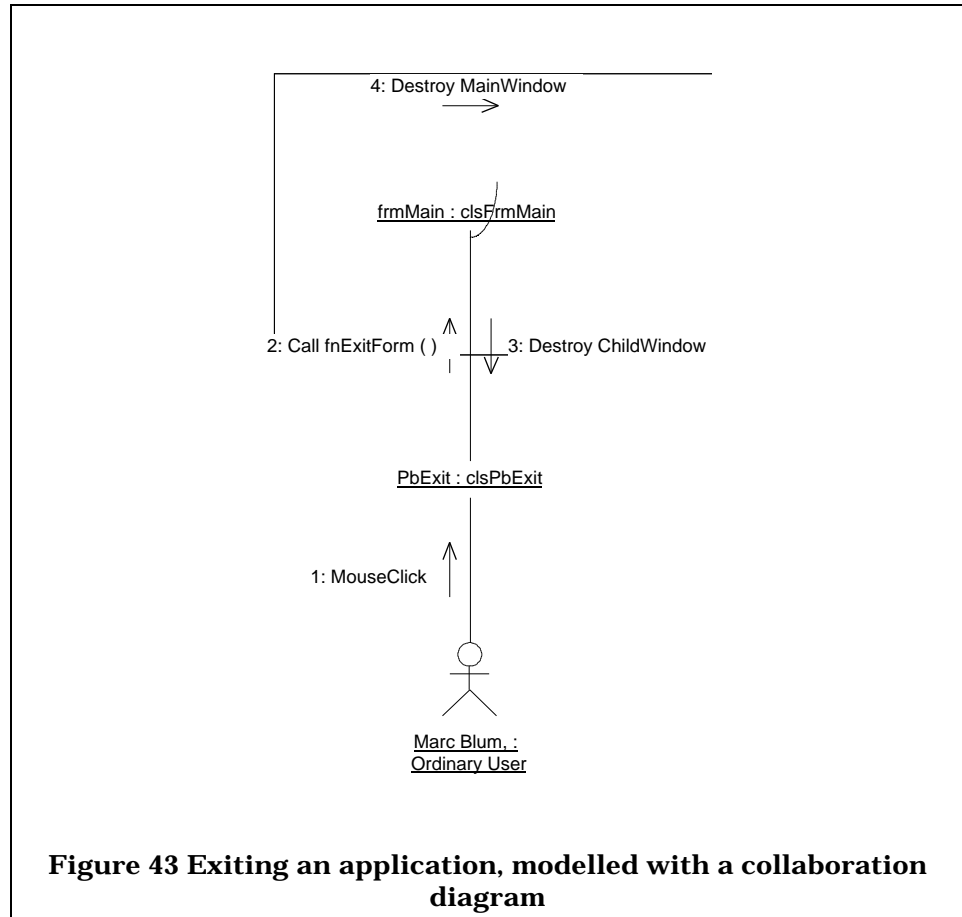
Sequence and collaboration diagrams are used to show the dynamic interaction. Both provide the same underlying information but in different ways. In Figure 42 and Figure 43 you find examples of both diagrams used to model an exit-window functionality.



To model the user interface it is important to describe

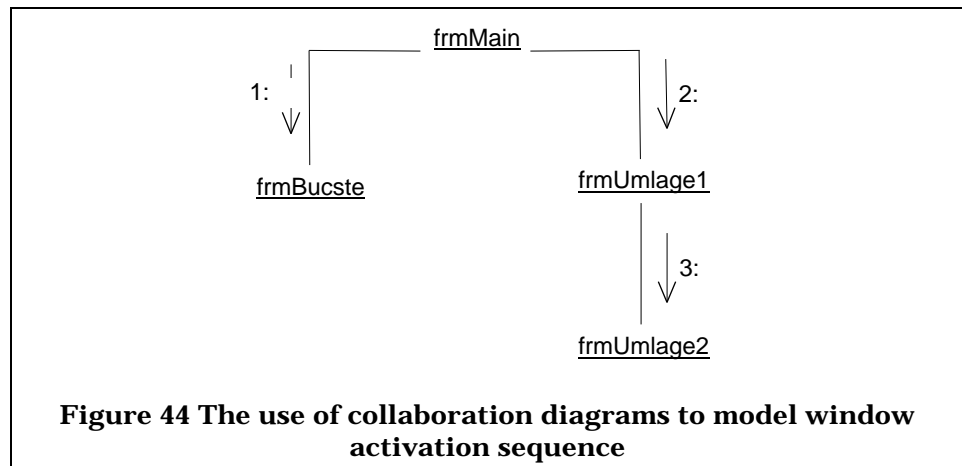
- which windows exist, and
- the window activation sequence.

Both can be done with a collaboration diagram (Figure 44) showing the window objects and their activation as links. The arrows indicate which window is activating which other one.



**Figure 43 Exiting an application, modelled with a collaboration diagram**

The user interface description can be completed by screen-dumps or hand-drawn pictures of the single windows.



**Figure 44 The use of collaboration diagrams to model window activation sequence**

### 3.3.3 Technical Model of Current System

#### Objectives

In the technical model of the current system we partly describe the current system on a detailed technical level. To reduce the needed effort we only model the parts of the current system we want to reuse in the target system. The main objectives of this technical model are:

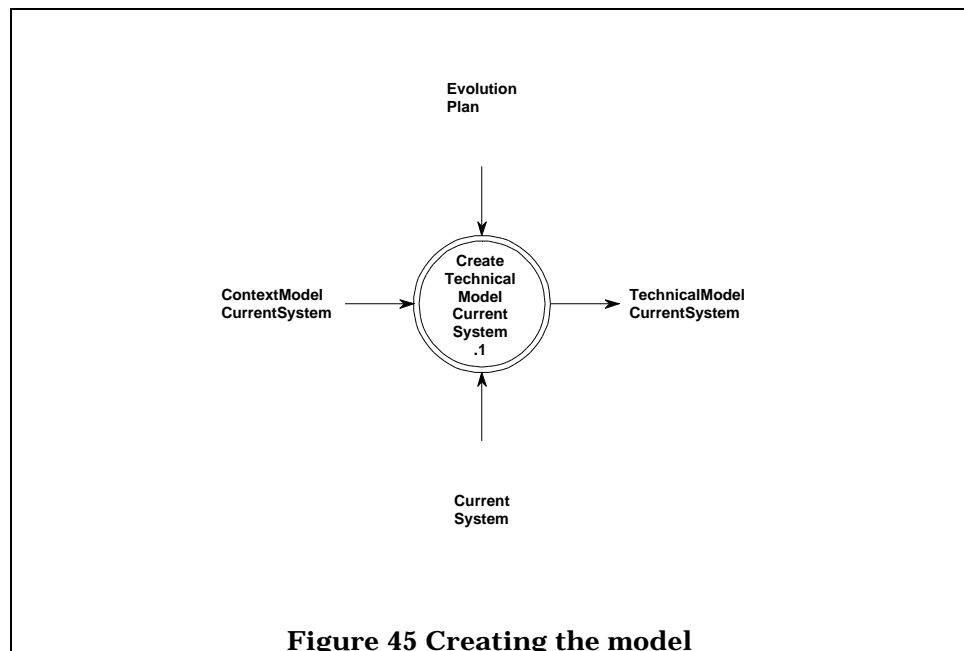
- To document the reusable parts of the current system.
- To have a base for extracting and adapting the reusable parts.
- To have a base for further maintenance in the reusable parts.
- To have a base for defining the interfaces to the new parts.
- To have a base for the technical model of the target system.

The main difference to the technical model of the target system is that we only model the parts of the current system we want to re-use and not the complete system as for the target system.

#### Content

The content of the technical model of the current system depends on the planned objectives and the properties we want to model. As notation we use UML. The most important UML diagrams for technical modelling are the class diagrams. If we can reuse parts without modification, only the interfaces are interesting. If the reusable parts must be adapted, the realisation aspects are also important and must be modelled. In the long term all important properties must be modelled to have good documentation for further evolution and maintenance.

#### Creating the model



The technical model of the current system is a refinement of the context model of the current system. The parts which should be reused are identified in the evolution plan. The required detailed information are get out of the current system. If documentation is available it must be tested if it is up-to-date. Necessary corrections must be done and it must be completed. Perhaps also the notation should be changed. Often reverse-engineering is the only way to get the required information. This can be done manually or tool supported like described in one of the following sections.

### **3.3.4 Technical Model of Target System**

#### **Objectives**

In the technical model of the target system we describe the target system on a detailed technical level. The main objectives of this context model are:

- To provide a detailed documentation of the target system.
- To provide a detailed construction plan for the following transformation and implementation steps.
- To allow the distribution into independent realisation units which can be performed in parallel by different teams or individuals.

#### **Content**

The design model can be described with UML as described in the sections before. The detail level of the technical model must be so, that we can divide the implementation work into realisation units and programmers are able to implement the units without more information.

The technical model of the target system should provide the following information:

- A complete subsystem structure including interfaces and dependencies.
- A description of every major 4GL component including purpose, component interface, other used components, dependencies, realisation descriptions, etc..
- A description of the main interaction sequences and working principles.
- A detailed description of all external interfaces and interactions.
- A complete picture of the technical environment.

## Creating the model

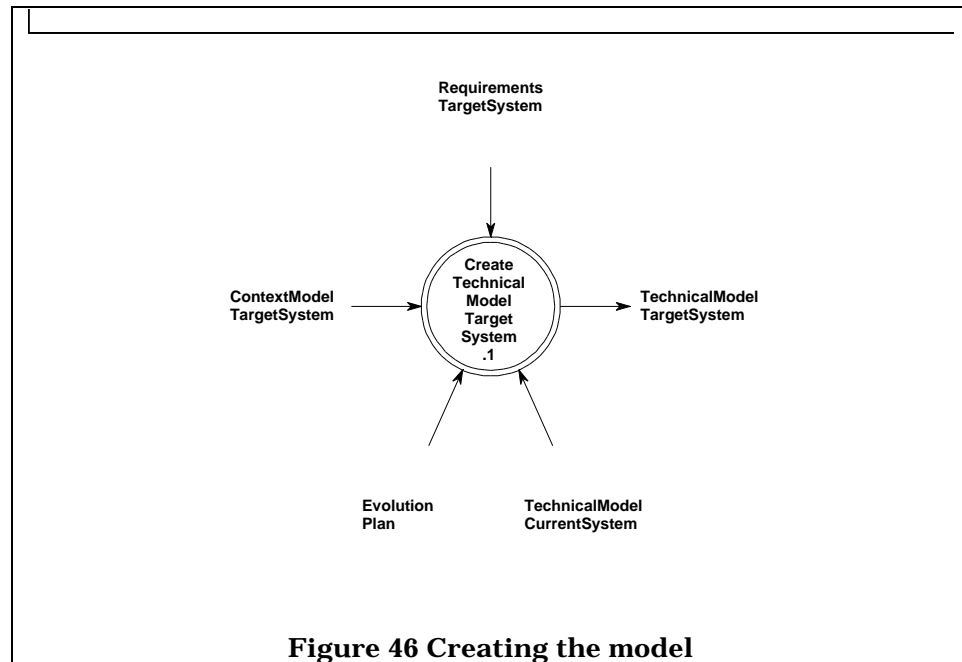


Figure 46 Creating the model

The technical model of the target system is a refinement of the context model of the target system. The parts, which should be reused from the current system, are identified in the evolution plan and the technical model of the current system provides the details. New requirements for the target system are an additional input that must be regarded.

### 3.3.5 Reverse-engineering of 4GL Applications

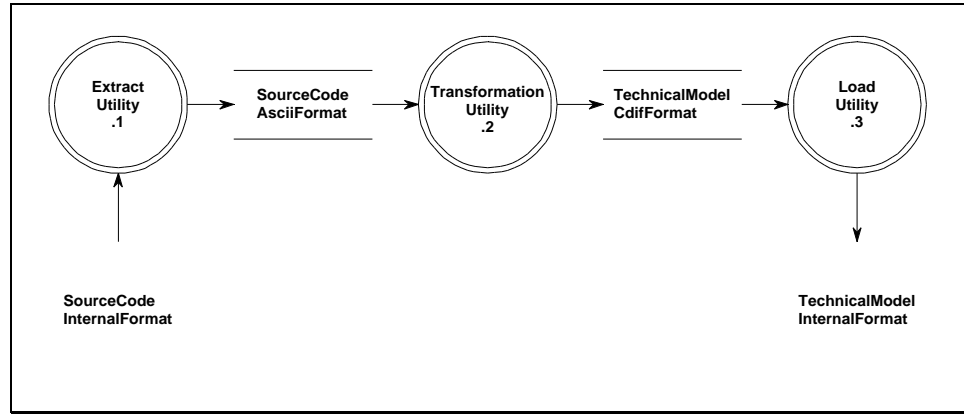
For the transformation and implementation work technical models of the current system are needed. Currently the reverse-engineering tool support for 4GLs is not so good as for 3GLs. Some CASE tools like Rational-Rose, etc. support bridges to 4GLs. Unfortunately CASE tool providers traditionally first concentrate their efforts on the 3GL support (C++, Smalltalk) and the 4GLs are only supported with old versions of the design method and/or the 4GL.

Nevertheless the interaction of 4GL tools and modelling tools provide several possibilities for a utility supported process. Beside their graphical user interfaces most 4GL tools provide two ways to access information:

- The 4GL code can be exported as text files. These files can be evaluated by scripts.
- Most 4GL tools store their information in a repository. If the repository interface is open and documented it can be used to extract the information.

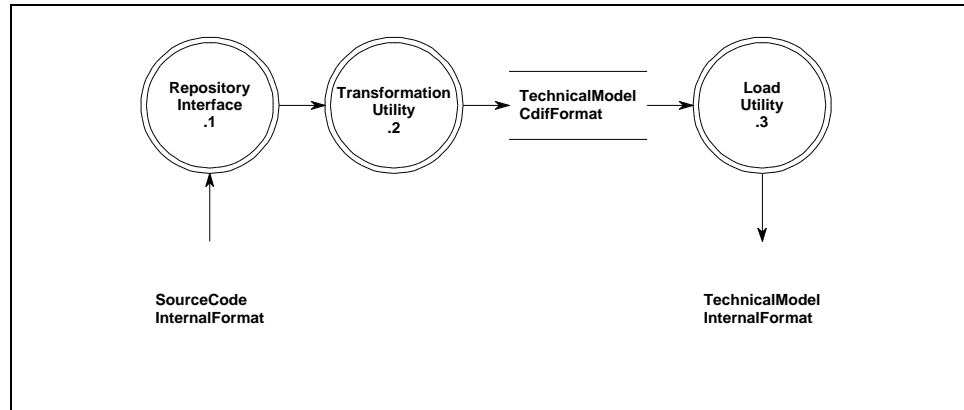
After extracting the information it can be used for manual modelling. An alternative is textual import possibilities provided by the modelling tools. To use them the extracted information must be transformed in the

required format, i.e. CDIF, which can be done automatically by a script. Two approaches of using reverse-engineering utilities are described below:



**Figure 47 Using reverse-engineering utilities**

1. With the help of an **extract utility (1)** the 4GL source code is stored in an ASCII file. The extract utility or functionality is often available as part of the 4GL environment.
2. Then the **transformation utility (2)** scans the source code, creates the technical model and stores it as CDIF file (ASCII representation of a model). This transformation utility must be created which can be a more or less complex job. As a 'utility' is not a 'tool' pragmatic solutions are possible.
3. In the last step an **load utility (3)** reads the CDIF file and stores the technical model in the database of the CASE tool. Load utilities are mostly available as part of the case tool environment.



**Figure 48 Using reverse-engineering utilities**

- With the help of the **repository interface (1)** the **transformation utility (2)** creates the technical model and stores it as CDIF file (ASCII representation of a model). This transformation utility must be created which can be a more or less complex job. As a 'utility' is not a 'tool' pragmatic solutions are possible. As the repository interface provides the information in a structured way this approach is easier

---

to implement as the previous one where we must scan the 4GL code which can be a complex job.

- In the last step a **load utility (3)** reads the CDIF file and stores the technical model in the database of the CASE tool. Load utilities are mostly available as part of the case tool environment.

### 3.3.6 Guidelines for structuring the Target System

Figure 49 shows an evolvable architecture of a 4GL client/server application. Compared to a simple 2-layer approach it contains a more advanced structure allowing the realisation of large enterprise-wide applications with a lot of flexibility. There is a clear structure with higher level subsystems calling lower level subsystems. There is no usage of higher level subsystems by lower level subsystems.

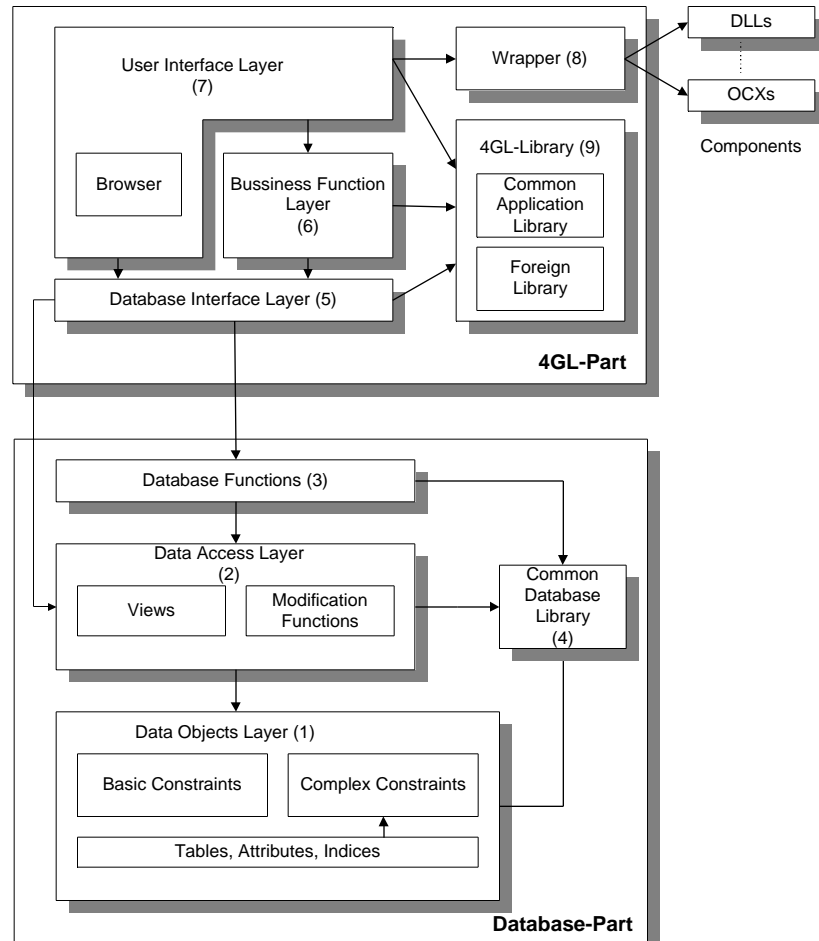
The first layer (1) is built by the data objects. Three sub-layers can be identified:

1. Basic objects of the database system i.e. table views and indices.
2. Basic constraints, i.e. domain constraints, not-null constraints, referential integrity, etc
3. Stored procedures for complex constraint checks. Triggers can activate this.

The second layer (2) is built by the data access layer. Two subsystems can be distinguished:

1. Views for selecting the required data.
2. Modification functions like insert, update and delete which are implemented as stored procedures.

The third layer (3) contains business functions running on the server. They are implemented as stored procedures. Database services (4), which are application independent, are collected into the common database library.



**Figure 49 Evolvable 4GL application architecture**

The database interface subsystem (5) is an abstract layer hiding the database system. It is the only subsystem communication with the database. Business functions (6) realised with the 4GL are encapsulated in a separate subsystem. They are separated from the user interface to have clear responsibilities in the different subsystems and to force reusability. The user interface (7) is the highest level. Inside, a substructure extracting general reusable elements from application specific ones will be realised. If packages or componentware, which are available as DLLs, OCXs, etc., will be integrated, the interface is hidden by wrapper subsystems (8). The wrapper layer abstracts from implementation and integration details. The advantage of this layer is easier exchange and reuse of components.

The last layer is built by 4GL libraries (9): The common application library contains general usable classes and functions, which are developed by the organisation in context. The foreign libraries consists of several subsystem developed by others. One example is the visual tool chest developed and distributed by Centura. Another example is the base classes and objects developed by the customer. An appropriate substructure will be used to order the different packages. A characteristic of this subsystem is that it contains only 4GL packages.





## 4 Case Study

### Contents

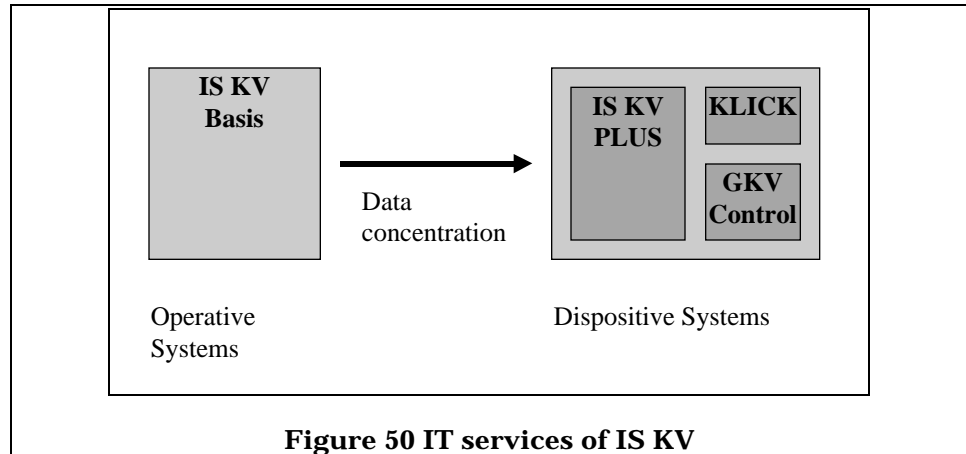
- 4.1 Description of the example application
- 4.2 Context modelling
- 4.3 Technical modelling

### Summary

An example application is used to illustrate the different modelling techniques. The example application KLICK - Kostenstellenrechnung (K-KST) is a cost computing application written in the 4GL SQLWindows. In the first section we shortly describe the K-KST application. The next section describes how the different techniques provided by context modelling can be used. In the last section we describe how UML is used for technical modelling.

## 4.1 Description of the example application

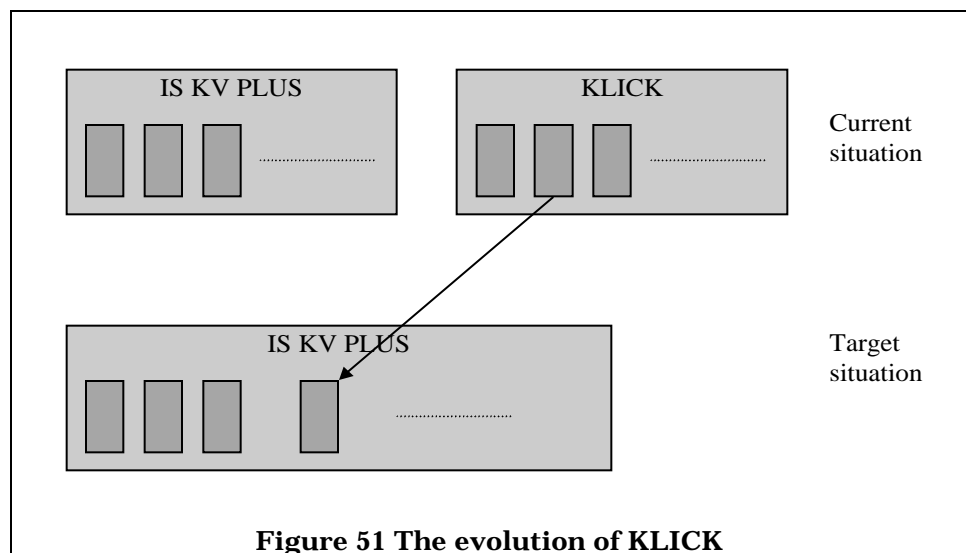
The chosen example application is a cost computing application in the health care domain. The application owner is the ARGE IS KV. The ARGE IS KV is an IT service company of the health insurer. The provided IT services are figured below.



The provided IT services are classified into two groups:

- **IS KV Basis** are a group of operative services
- **IS KV PLUS** is a group of management information services also called dispositive systems.

Currently the dispositive systems consist of three components: IS KV PLUS, KLICK and GKV Control. IS KV PLUS is based on the same technology than IS KV Basis. In contrast to this, KLICK and GKV Control are using other technology. In the long term KLICK and GKV Control should be integrated into IS KV PLUS with a common technology.



KLICK consists of several systems. For each we must check if and how we should transfer it into IS KV PLUS. One of this KLICK systems is the cost computing system KLICK - Kostenstellenrechnung (K-KST). This system is one of the systems with highest priority for the customer.

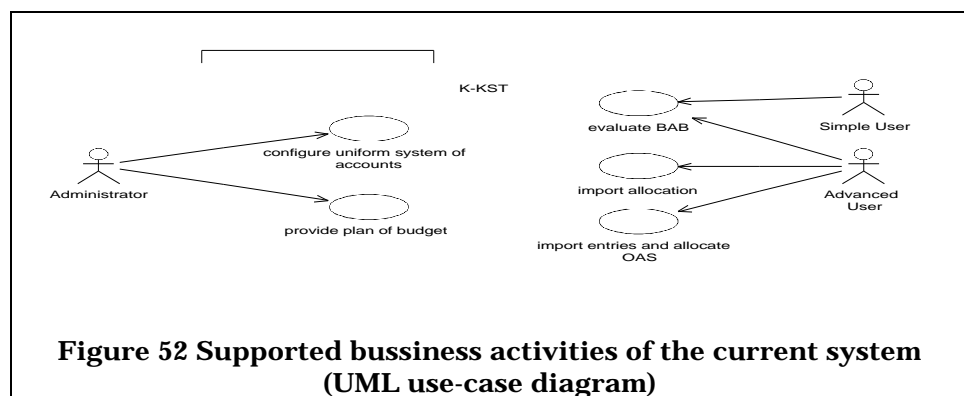
## 4.2 Context modelling

In this section we use the techniques provided by context modelling to give an overview of the K-KST application. In some cases we use two alternative diagrams to illustrate the different modelling diagrams.

- A use-case diagram (UML) is used to describe the supported business activities.
- A context diagram (SA) is used describe, how the K-KST application communicates with its environment.
- A data flow diagram (SA) is used to describe the functionality of the K\_KST application.
- A block diagram and a package diagram (UML) are used to describe the software structure of the application.
- An entity relationship diagram (IM, Chen) and a package diagram (UML) are used to describe the underlying data structures.
- A hardware/netware diagram and a deployment diagram (UML) are used to describe the operating environment.

### Overview of the supported business activities

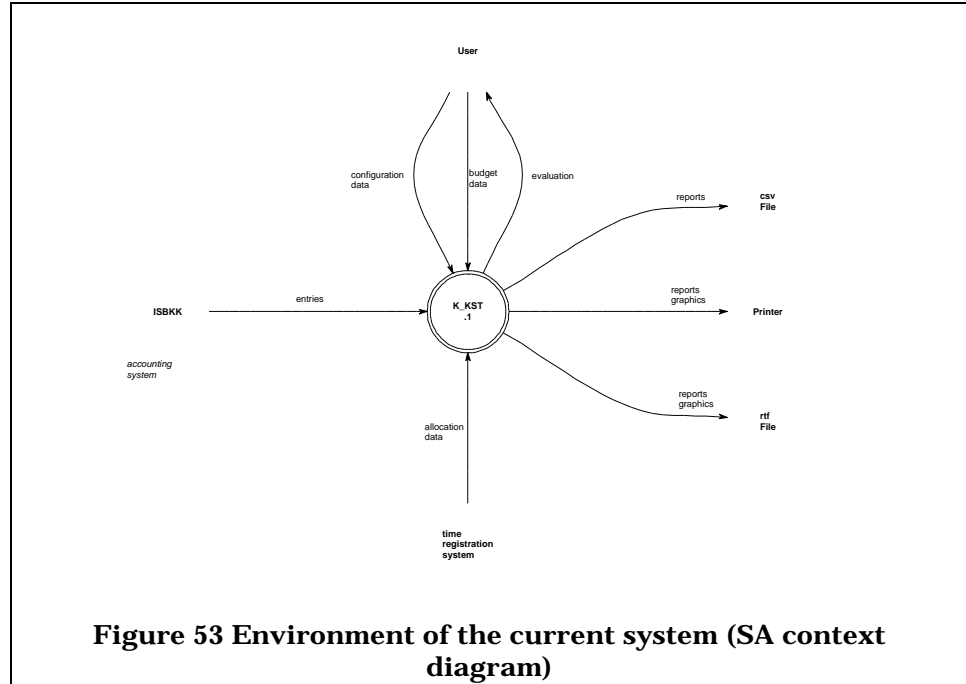
The business activities supported by the current system are modelled using UML use-case diagrams.



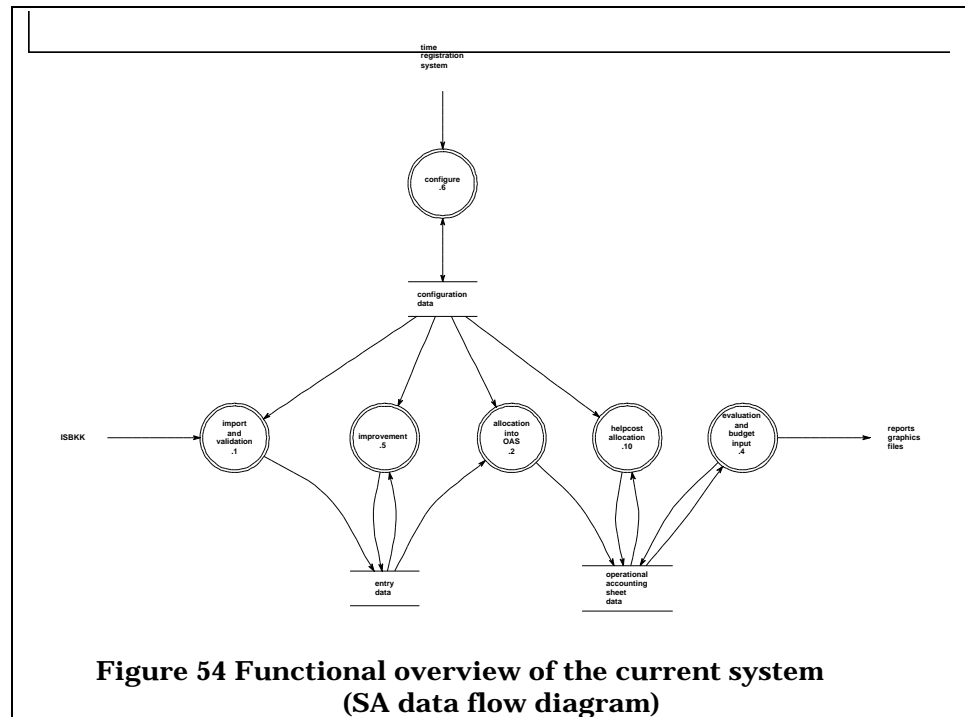
A package named K-KST stands for the system. Five business activities, modelled as use-cases, and three actors can be identified. The Administrator uses K-KST to configure a uniform system of accounts and to provide the plan of budget. These activities are not often performed. The regularly performed activities know an advanced user who imports the entries and integrates them in the operational accounting sheet (OAS), imports allocations and evaluates the OAS. A further simple user only evaluates the OAS.

### Overview of the supported functionality

For modelling the functionality of the current system data flow diagrams (structured analysis) are used. The analysis is performed in two steps: a context diagram shows the interfaces to the system's environment. A first refinement identifies the main functionalities inside the system.



The context is displayed as follows: from the bookkeeping system ISBKK the entries are provided to K-KST in form of a CSV-file. From the time registration system the allocation data are entered. The user (in UML-Use-Cases the actors) interact with the system by entering configuration data and budget data and receiving evaluations. K-KST provides generating reports as CSV-files, printed graphics and saved RTF-files for further text-processing.

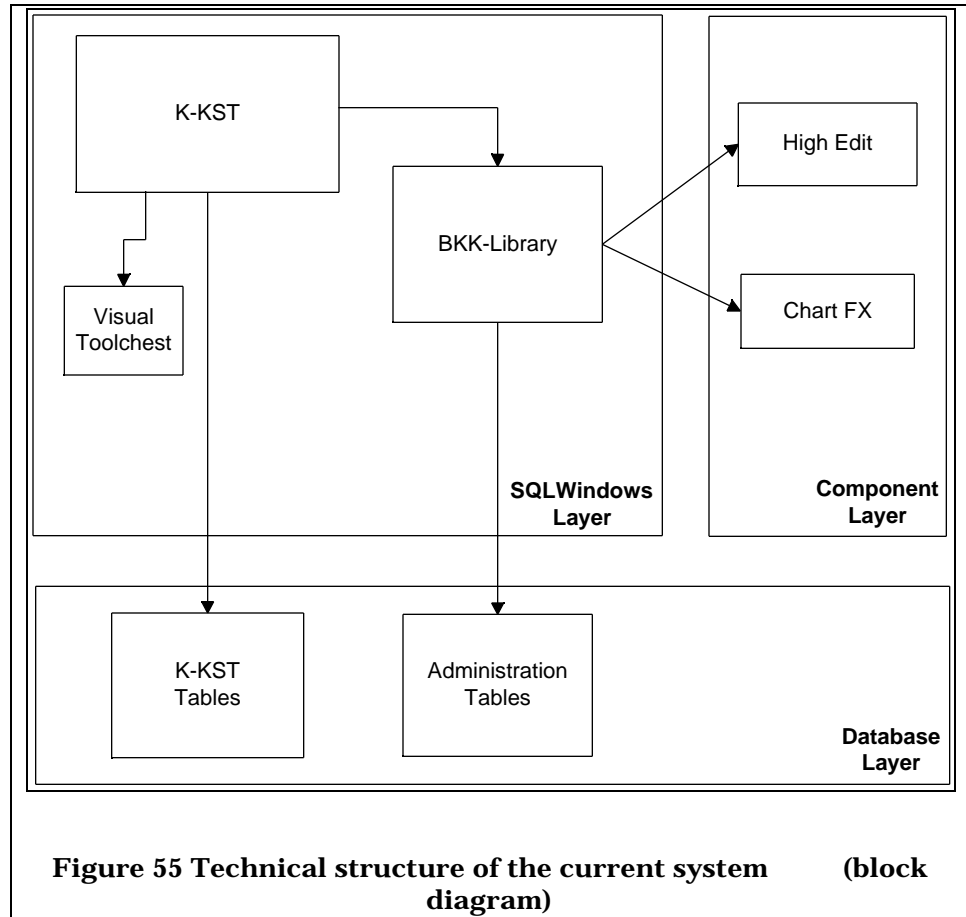


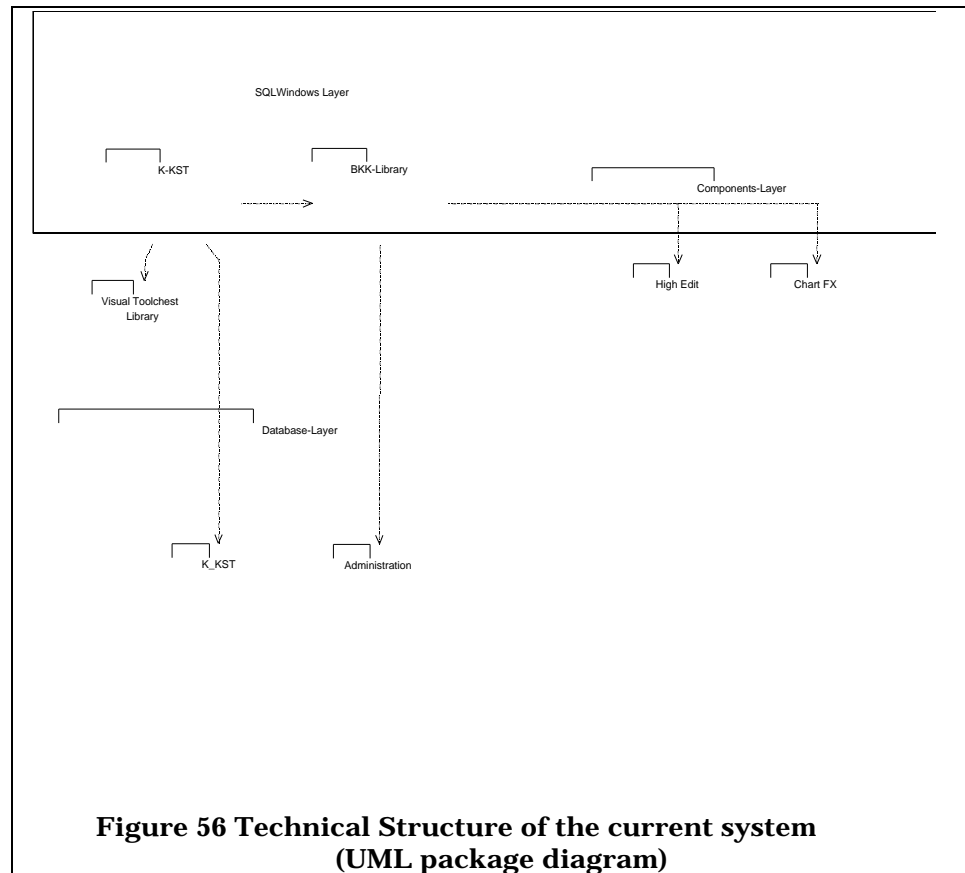
In a first refinement six main processes and three data stores can be identified. The incoming CSV-file is imported into the entry data and validated by using configuration data, which is imported from the time registration system. An improvement of the entries may be executed. Allocation into OAS (operational accounting sheet) transfers data into OAS data. A help cost allocation may take place on these data. Finally the OAS data will be evaluated and the budget data inserted into OAS data. From the OAS data different evaluations are computed.

### Overview of the technical structure

To understand the way the system is technically build, block diagrams are used to visualise the system architecture. For evaluation purposes the same picture is designed by using UML-elements. Each logical block is mapped to a UML-package. The following description is identical to both diagrams.

The whole application can be seen as build of three layers incorporating seven logical blocks. The SQLWindows layer represents the 4GL part which consists of the main block K-KST which is build by debis, the BKK-library containing base classes and the Visual Toolchest which is a Third-Party software product for SQLWindows applications. The component layer incorporates a word processor (High Edit) and a diagram generator (Chart FX). In the database layer administration tables, which are only of importance to the system and not to the business processes, and the K-KST tables which contain the operational data can be identified.

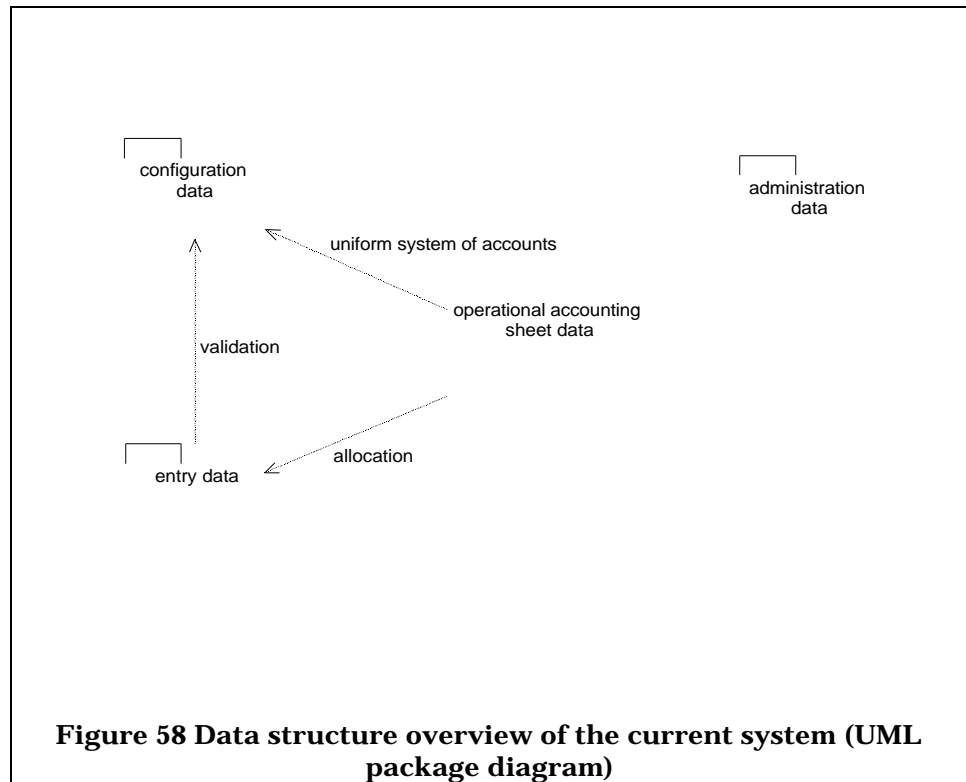
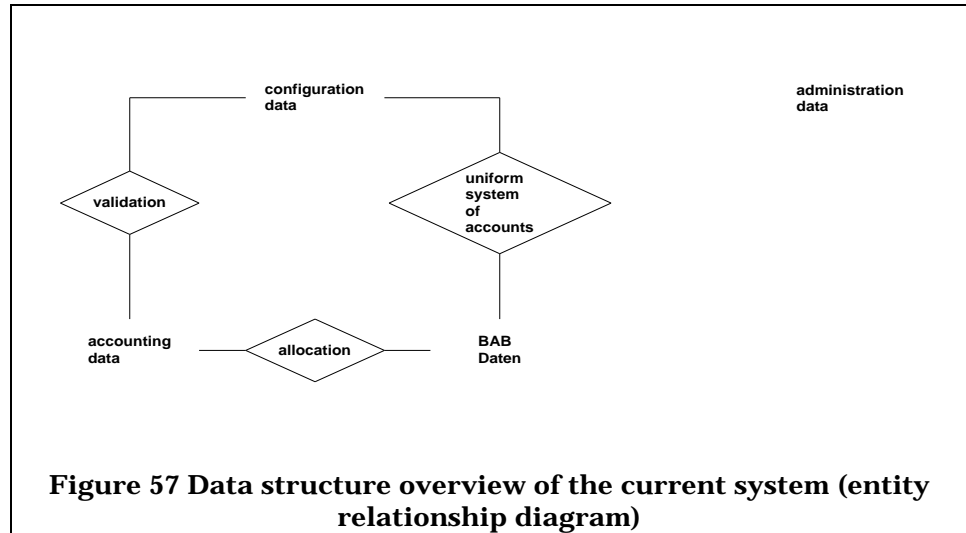




### Overview of the data structure

The global structure of the data stored in the database is modelled by using ERDs or alternatively UML packages.

Although the administration data shows no further substructure, the K-KST data is subdivided into configuration data, OAS data and accounting data with the relations validation, uniform system of accounts and allocation. This structure matches exactly the stores identified in the first refinement DFD. In the UML package diagram dependencies between data are direction-sensitive. One sees that the contents of OAS data depends on the contents of accounting data and configuration data. The contents of accounting data depends on the contents of configuration data.

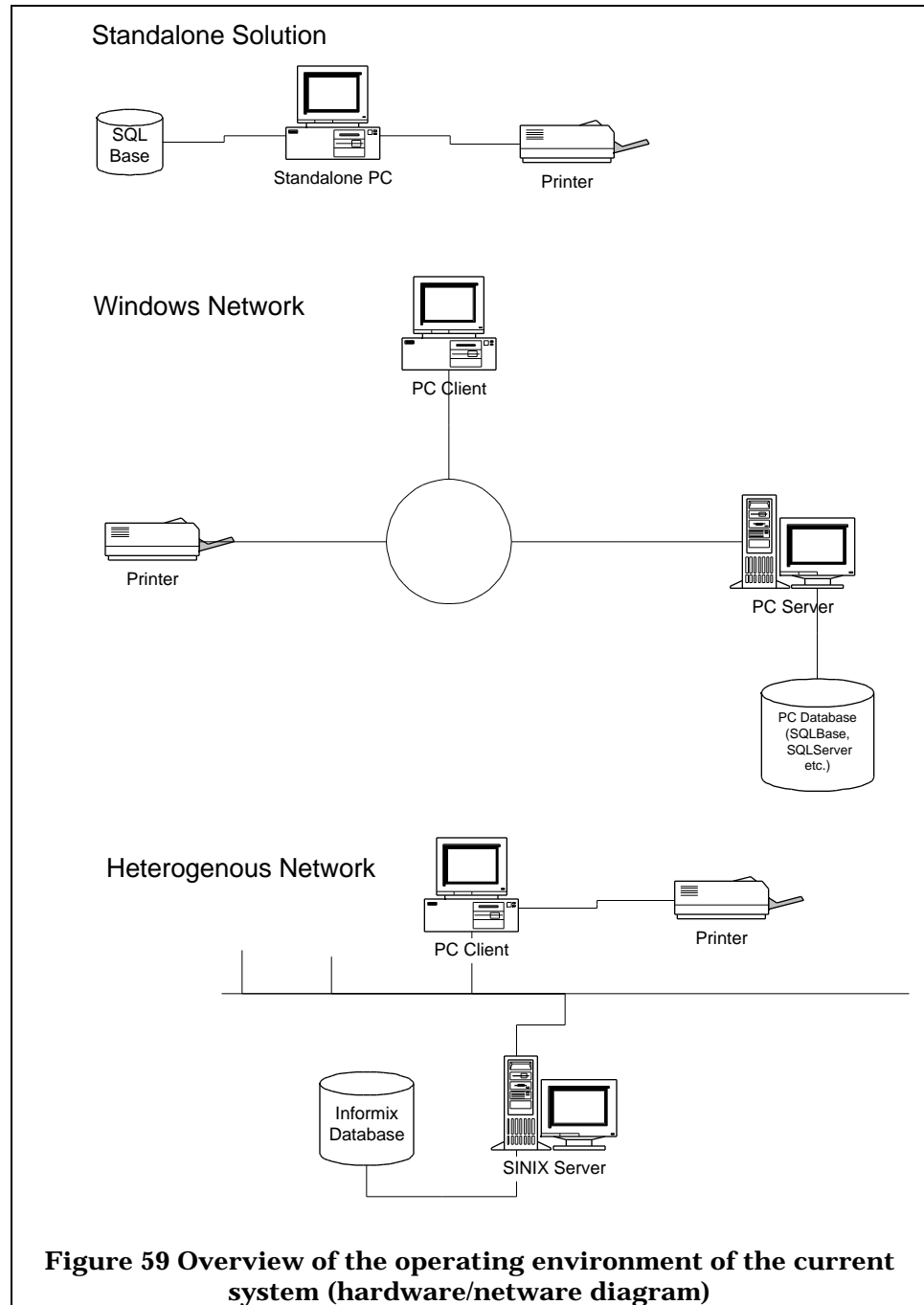


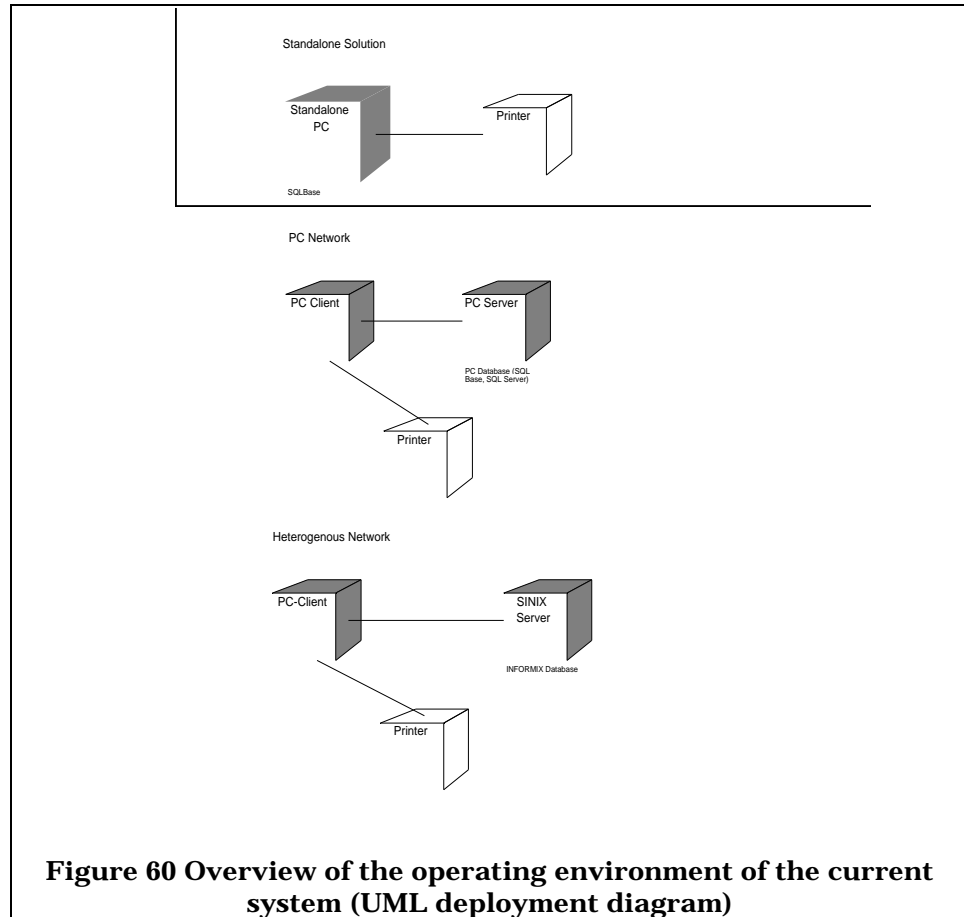
### Overview of the operating environment

The operating environment is modelled as hardware/netware diagram. For evaluation purposes the same contents is provided as UML deployment diagram.

The current system is likely to be configured in several ways. A standalone solution is possible, based on a single PC running a single-user PC-based

database like SQLBase. Even more flexibility offers a network solution where a PC server is possible or even a UNIX-based system with a Informix database running on it.





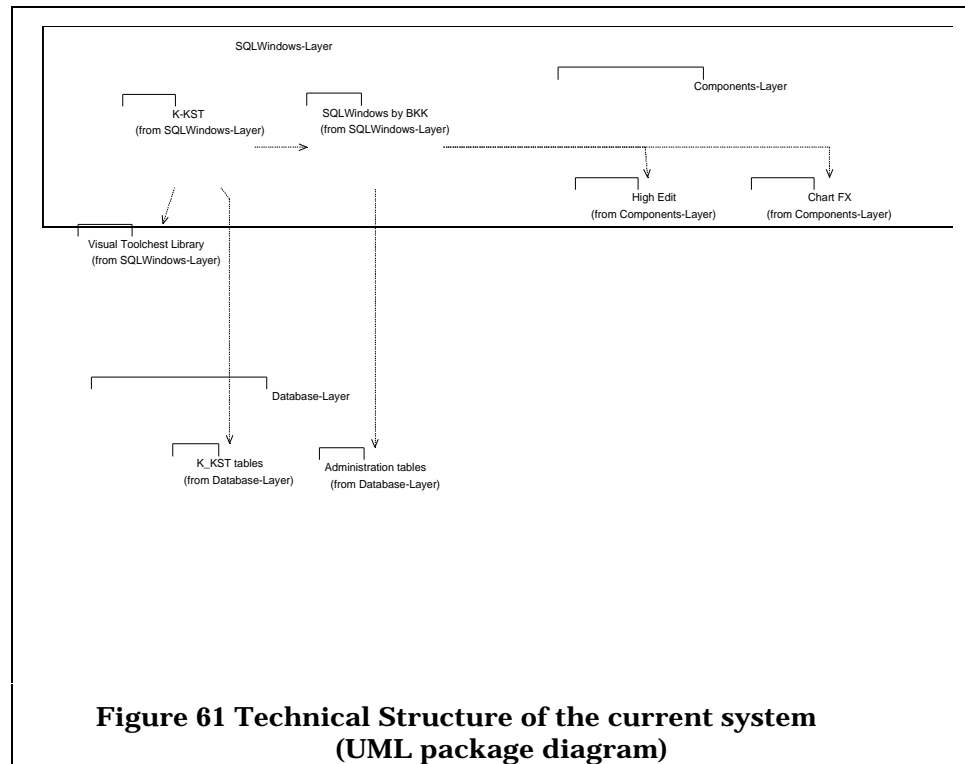
### 4.3 Technical modelling

In this section selected parts of the current system are described in detail. The objective of this section is to describe, how the proposed 4GL modelling techniques can be used for SQLWindows/Centura applications. We have not the objective to provide a complete technical model of the K-KST application. All used diagrams are UML diagrams.

- Package diagrams are used to describe the subsystem structure of the K-KST application on multiple layers.
- A class diagram is used to describe the realisation of forms.
- A class diagram is used to describe the underlying database.
- A collaboration diagram is used to describe the user interface dynamics.
- A sequence diagram is used to describe the message and function sequences.

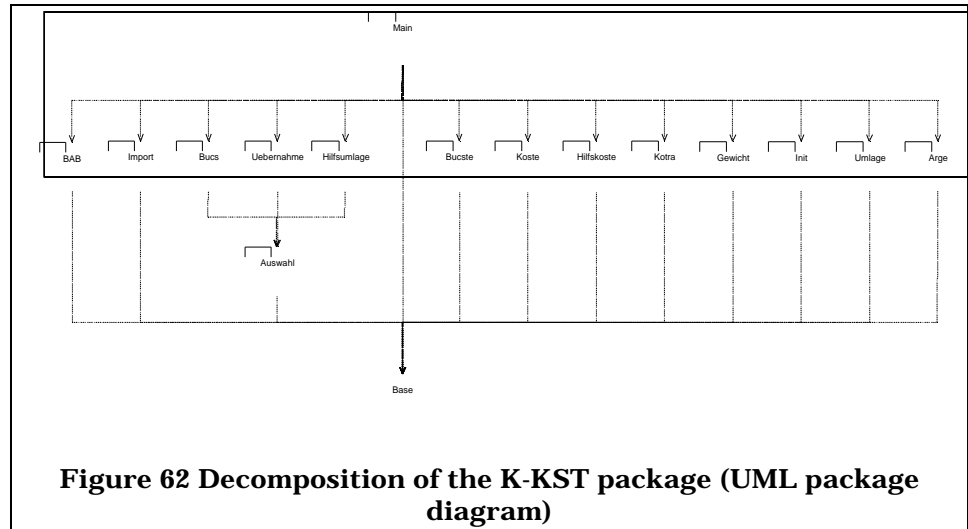
#### Subsystem structure of the current system

We start with the overview of the technical structure of the current system. This overview is the interface between context modelling and technical modelling and the start point for the detailed description.



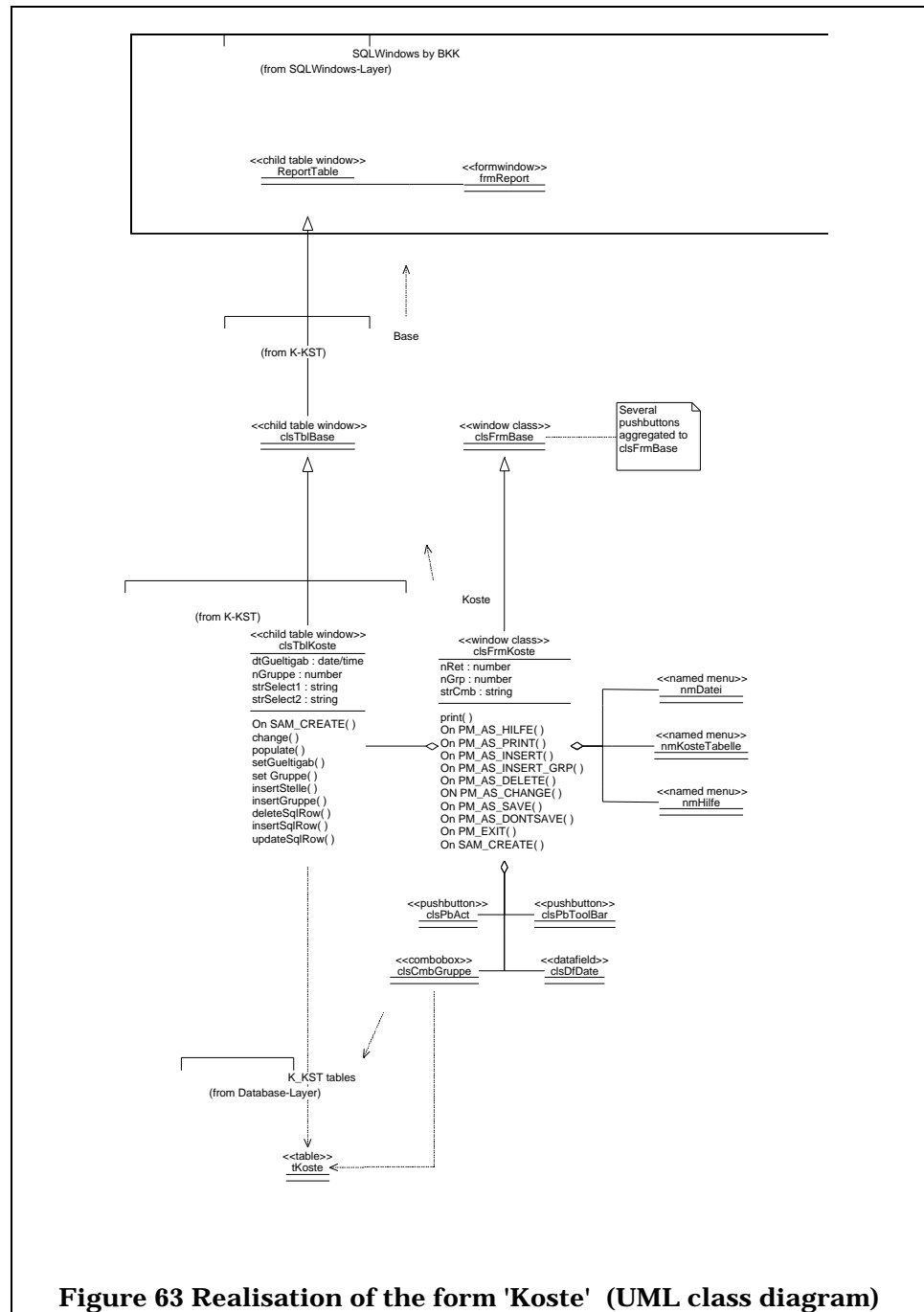
On the first refinement level the K-KST application consists of three layers represented as packages. The first package contains the SQLWindows parts of the system, the second package the database parts and the third one integrated component-ware.

On the second refinement level we see that the SQLWindows layer consists of the K-KST package, a BKK library package and the visual toolchest package. The K-KST package contains the main part of the application, the BKK library contains SQLWindow components provided by the client and which are integrated into the K-KST application. The component-ware layer consists of a text processing and a graphical component.



**Figure 62 Decomposition of the K-KST package (UML package diagram)**

The composition of the K\_KST package is shown in the diagram above. For each form we have one package representing it. The details of the forms are modelled inside the packages. We have one additional package named 'Base' containing SQLWindows components used by the other forms. The dependencies between the packages are modelled with call-relationships.

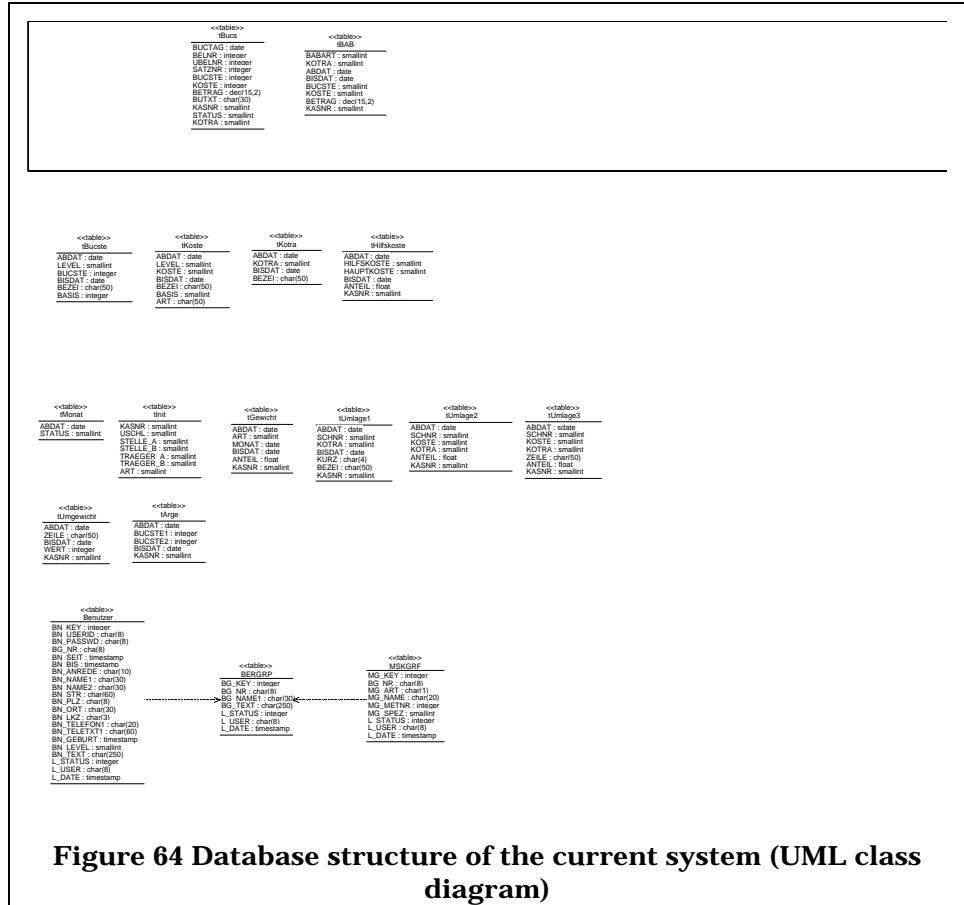


**Figure 63 Realisation of the form 'Koste' (UML class diagram)**

The diagram above shows the realisation of the form 'Koste'. We have one package, named 'Koste', for this form. The form itself is realised as class, named 'clsFrmKoste'. The messages which can be handled by this form and the defined functions are documented as operations of this class. The internal variables are modelled as attributes. The form consists of several components which are represented by own classes and connected by aggregation relationships. The form class itself is derived from a base form class named 'clsFrmBase'. This is indicated by an inheritance-relationship. The related classes from other packages are also shown in

the diagram. The form `ëKosteí` is manipulating the database table `ëtKosteí`, which is modelled as a class of the package `K-KST` tables. The use-relationship between the `SQLWindows` class `ëclsTblKosteí` and the database table class `ëtKosteí` is documenting the interaction between 4GL and database part.

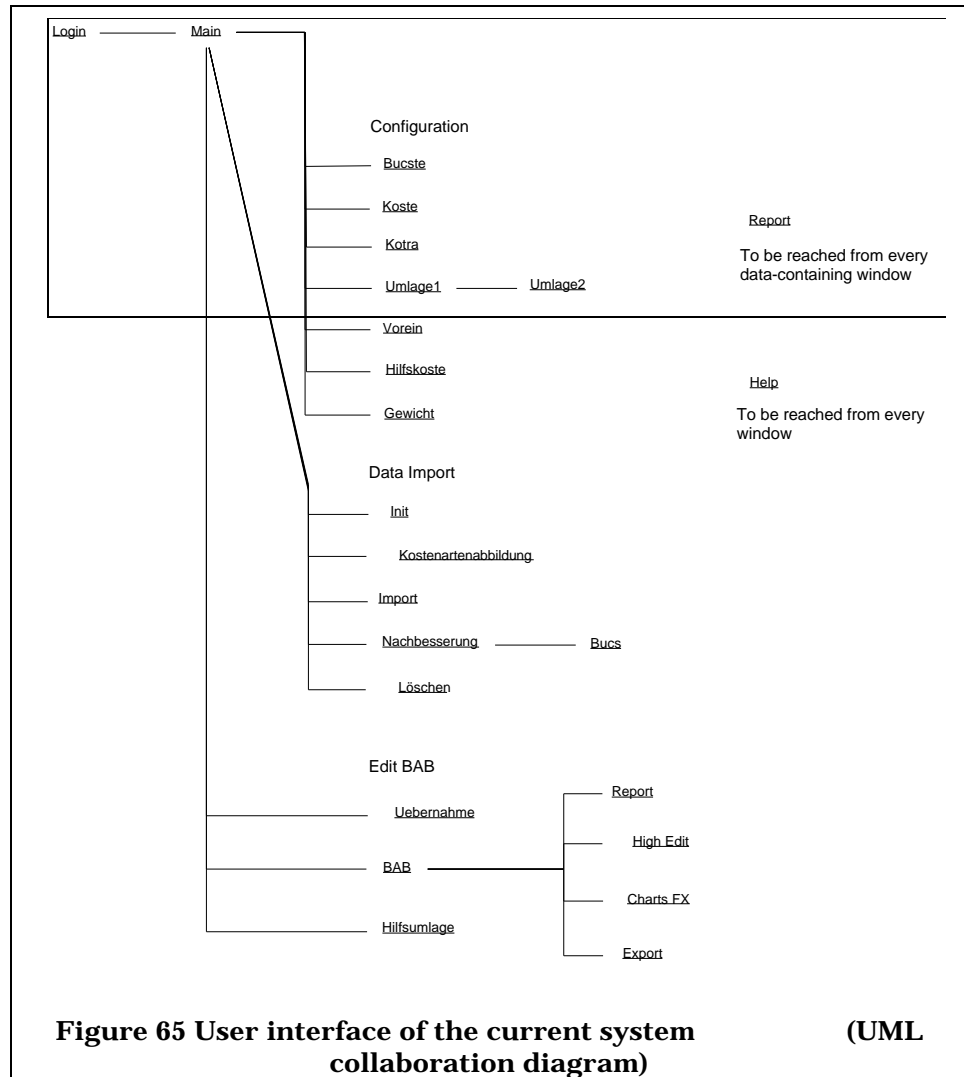
### Database structure of the current application



**Figure 64 Database structure of the current system (UML class diagram)**

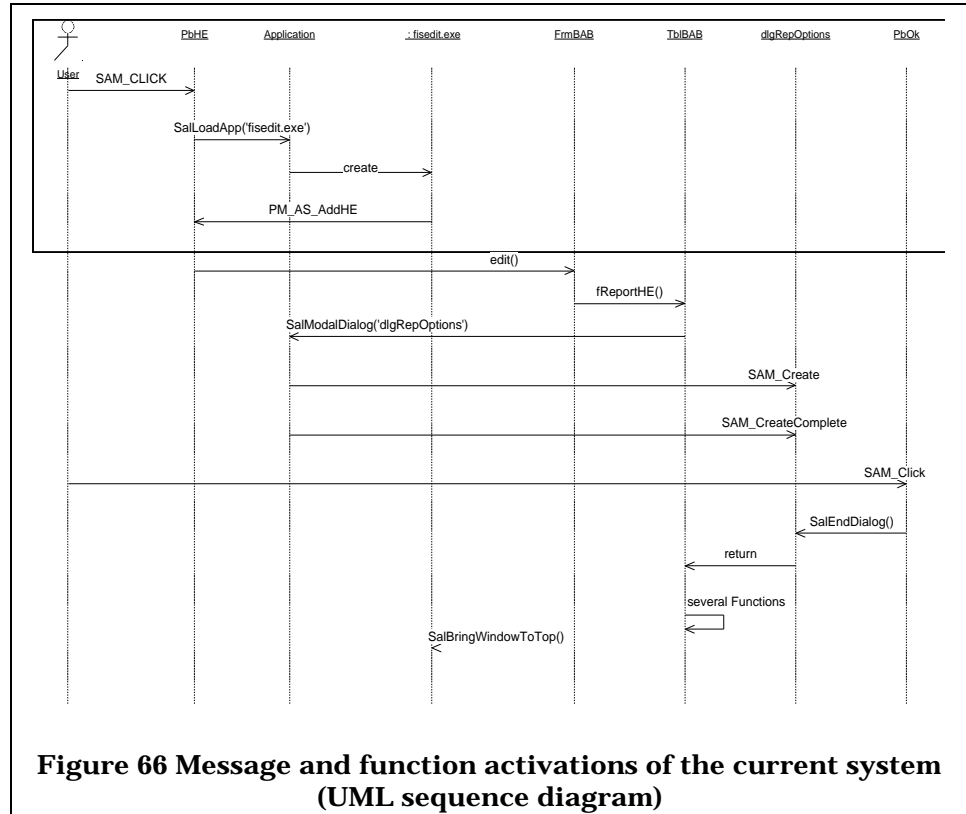
The structure of the `K-KST` database is shown in the figure above. The database tables are represented as classes, the table attributes (fields) as class attributes. As all relationships between the different tables are realised in the application we see no dependencies on database level. The reasons for this was to be database independent (`SQLBase`, `Informix`).

## User interface of the current system



In the diagram above we see the form activation sequences as collaboration diagram. Each form is represented by one object. The relations indicate which form is indicated by which one.

### Message and function activations of the current system



**Figure 66 Message and function activations of the current system (UML sequence diagram)**

The diagram above shows a typical message and function activation sequence of an SQLWindows application. The lines represent objects of the application, the arrows activations of messages and functions. The name of them is indicated above the arrow.

## 5 Appendix A: Overview of the UML

### Contents

- 5.1 Introduction
- 5.2 Static structure diagrams
- 5.3 Use case diagrams
- 5.4 Sequence diagrams
- 5.5 Collaboration diagrams
- 5.6 State diagrams
- 5.7 Activity diagrams
- 5.8 Implementation diagrams

### Summary

This chapter provides an overview of the UML and can be skipped by readers which are familiar with this language.

## 5.1 Introduction

The Unified Modelling Language (UML) is an adaptation and an extension from works of Grady Booch, Jim Rumbaugh, Ivar Jacobsen and others. The main focus of UML is to provide an object-oriented modelling language that can be used to model most types of systems. To do this, UML provides a rich set of notations and promises to be supported by all major CASE tool vendors. The UML does not enforce any particular process. It is however possible to utilise methods with well-defined processes such as Booch, OMT and OOSE in UML, because UML supports most methods. Even if the UML does not mandate a process, its developers has recognised the value of using processes with UML. The processes enabled with the UML are *use-case driven, architecture-centric, iterative, and incremental* processes.

To summarise what UML is all about; it is a language for specifying, constructing, visualising and documenting the artefacts of a software system. Why has the RENAISSANCE project chosen to use UML for evolution modelling?

1. First , the UML supports modelling of most types of systems and does not enforce a specific process. Since the RENAISSANCE project consists of partners from different countries and companies that support various kinds of software systems, it is necessary to have an flexible modelling approach.
2. Second, the UML is being presented to the Object Management Group as a proposal for a standard modelling language for object-oriented development. This means that the UML might be widely accepted by software developers.
3. Third, the UML is supposed to be able to model both object oriented systems and non-object oriented systems.

The rest of this chapter presents a condensed introduction to the diagrams used for modelling a system using the UML.

### 5.1.1 Modelling with UML Diagrams

The UML model formalizes the problem domain and software system. The model contains classes, logical packages, objects, operations, component packages, modules, processors, devices, and the relationships among them. A model is organized using different kinds of diagrams. These offer support for visualising and manipulating the model's components and their properties. Icons are used to represent a model's components and can appear in none, one, or several of a model's diagrams to illustrate multiple views of the model.

UML provides several diagram types for representing different perspectives of the system model. The model may contain several diagrams of a particular type. The rest of this chapter presents the different types of diagrams provided by the UML.

### 5.1.2 Stereotypes

A *stereotype* is a metaclassification of an element in the UML, and identifies the *type* of the element in the UML. For example, predefined *class stereotypes* include Event, Exception, Metaclass and Utility. The primary advantages of stereotypes are first that it is possible to refer to the type of the element, as in “That class is an Exception class;” and second that the UML is extensible by the user of the method through the definition of additional stereotypes. We have used this UML feature to extend UML for modelling 3GL and 4GL applications as discussed in Chapter 3.

Stereotypes are indicated by enclosing the name of the stereotype inside guillemots (« »), as for example «domain».

## 5.2 Static structure diagrams

Static structures are modelled using either *Class diagrams* or *Object diagrams*.

A Class diagram is used to describe classes and the relationships among. Class diagrams make it possible to use process-specific extensions defined for various diagrams (stereotypes).

A Class diagram can be specified by the following:

- Graph of modelling elements that may contain types, packages, relationships and instances (class diagram containing only instances is an «object diagram»)
- Can be interpreted as a static structural diagram.
- Is a collection of static declarative model elements with relationships.
- Can be organised into packages.

Figure 67 depict different classes and the relationships between them. Each class consists of three compartments. The top *name compartment* holds the class name and other general properties of the class. The middle compartment holds a *list of attributes*. The bottom compartment holds a *list of operations*.

An object diagram describes a snapshot of the detailed state of a system in a point in time. A static object-diagram is an instance of a class diagram. Dynamic object structures may be described using sequence or collaboration diagrams, described later in this chapter.

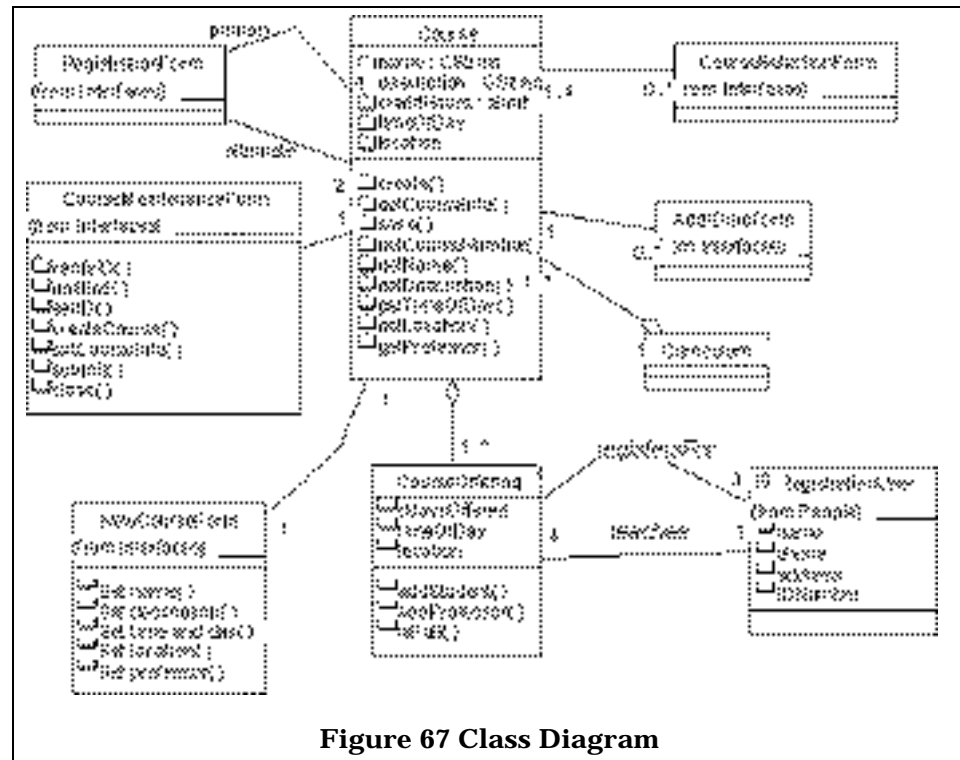


Figure 67 Class Diagram

### 5.3 Use case diagrams

A use case diagram shows the *relationships* among *actors* and *use cases* within a system. The relationships model communication (participation) associations between actors and the use cases. Use case diagrams are an aid to defining what exists outside the system (actors) and what should be performed by the system (use cases).

The actors represent what interacts with the system. They represent everything that needs to exchange information with the system. We differentiate between actors and users. The user is the actual person who uses the system, whereas an actor represents a certain role that a user can play. We regard an actor as a class and users as instances of this class. The instances exist only when the user does something to the system. The same person can thus appear as instances of several different actors.

An instance of an actor does a number of different operations to the system. When a user uses the system, she or he will perform a behaviourally related sequence of transactions in a dialogue with the system. This special sequence is what is called a use case. A use case is a complete flow in the system, and hence use case diagrams are different from data flow diagrams as the former focus on modelling the internal flow in the system. Neither does use cases model data flow, but rather interactions among the actors and the use cases.

The specification of use case external behaviour defines the possible sequences of messages exchanged among the actors and the system.

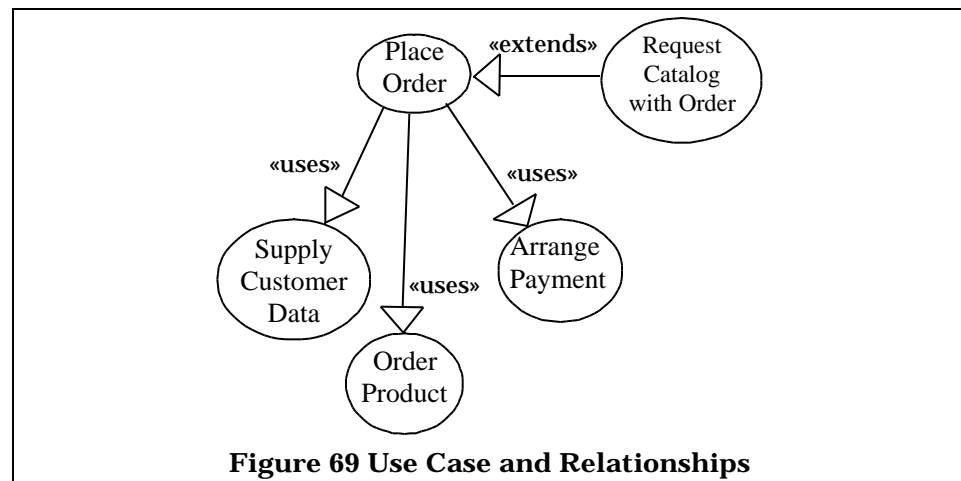
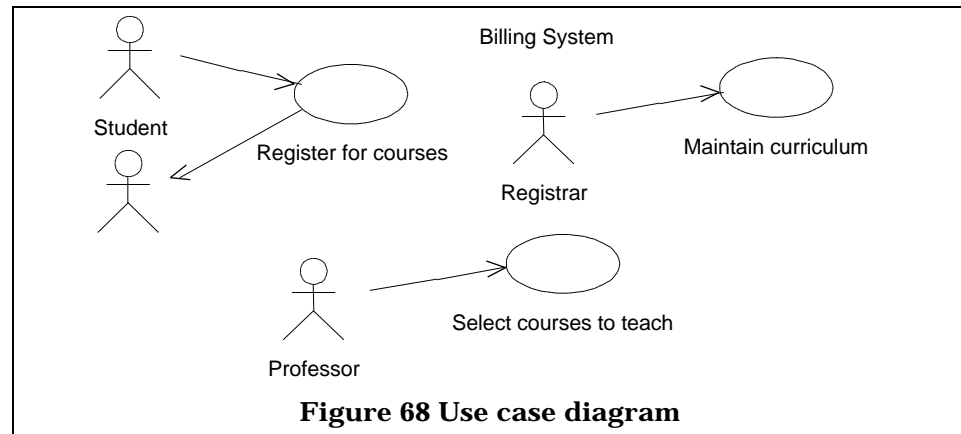


Figure 68 shows different *Actors* to the left and *Use Cases* to the right and the relationships between them.

There are three meaningful relationships within a use case diagram as shown in Figure 68 and Figure 69:

- *Communicates*: Shown by connecting the actor symbol to the use case symbol by a solid path. The actor is said to «communicate» with the use case.
- *Extends*: Shown by a generalisation arrow from the use case providing the extension to the base use case. The arrow is labelled with the stereotype «extends». Extends relationships are used to indicate that an instance of use case may include the behaviour specified by another.
- *Uses*: Shown by a generalisation arrow from the use case doing the use to the use case being used. The arrow is labelled with the stereotype «uses». Uses relationships are used indicate than an instance of one use case will also include the behaviour of another.

## 5.4 Sequence diagrams



- A description of the affected objects (including their relevant relationships, attributes and operations)
- A description of the sequences of messages exchanged among the objects to perform work.

Two aspects needed for a full specification of behaviour:

- *Context* supplied by the collaboration.
- *Interactions* supported by the collaboration.

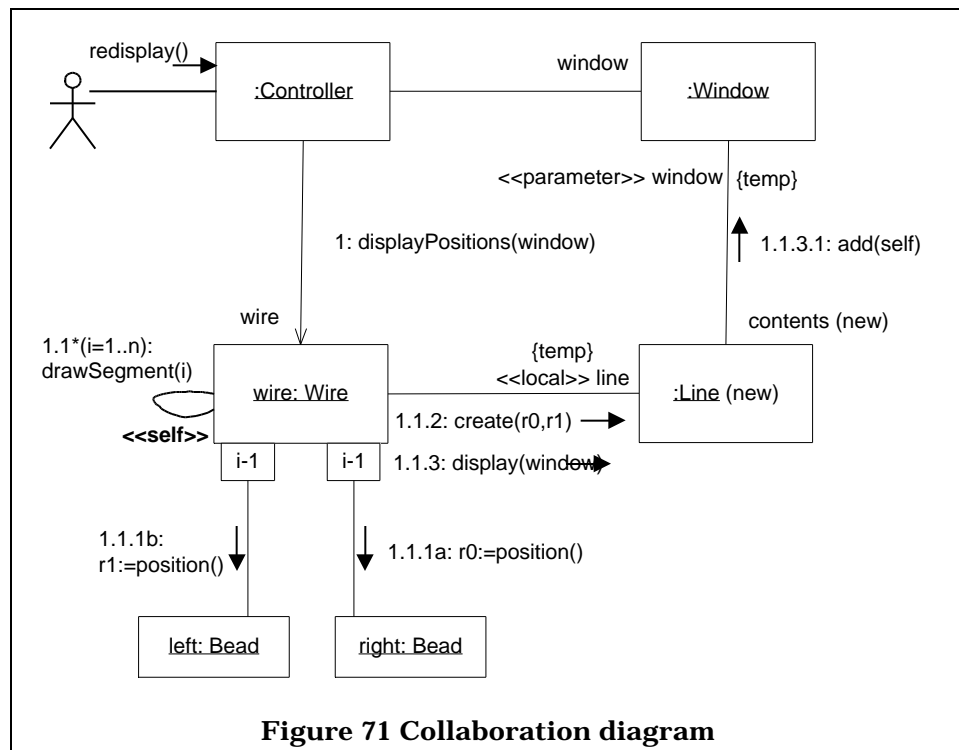


Figure 71 shows different objects connected by links. Links are used to carry messages between objects. The *invoker* of an interaction is represented in the figure above as an actor symbol. Messages are used to implement an operation and are numbered starting with number 1. Nested numbers are used for a procedural flow to represent call nesting.

## 5.6 State diagrams

The state diagram is used to show the sequences of states that an object or an interaction goes through during its life in response to received stimuli, together with its responses and actions.

The notations and semantics used for state diagrams are statecharts with some minor modifications.

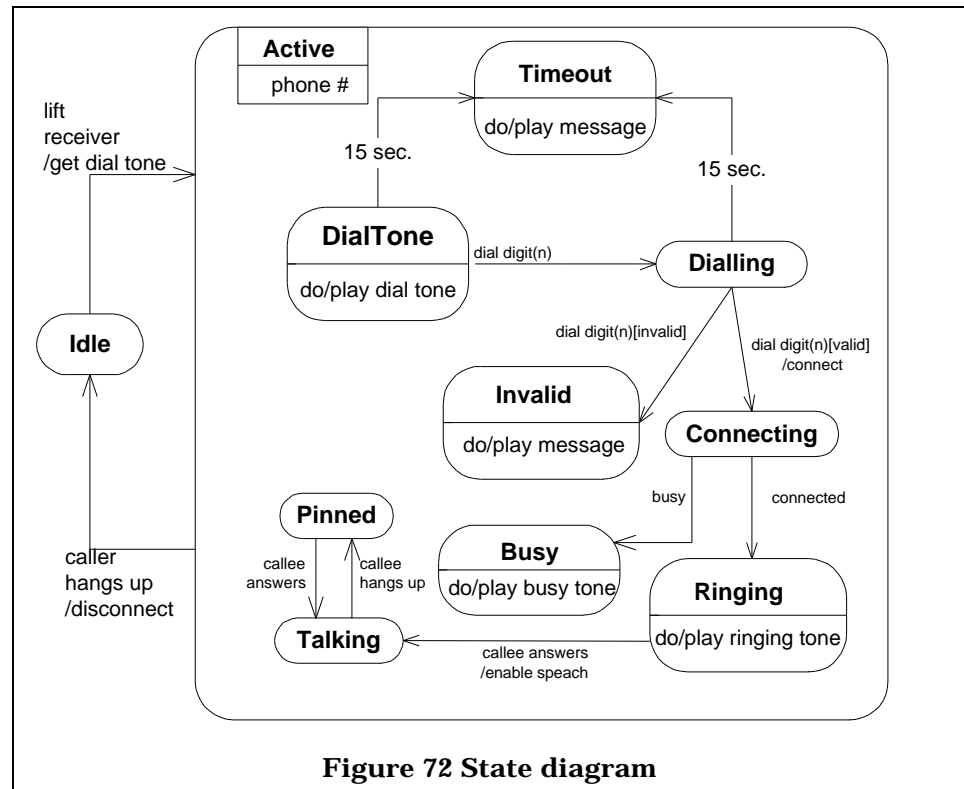


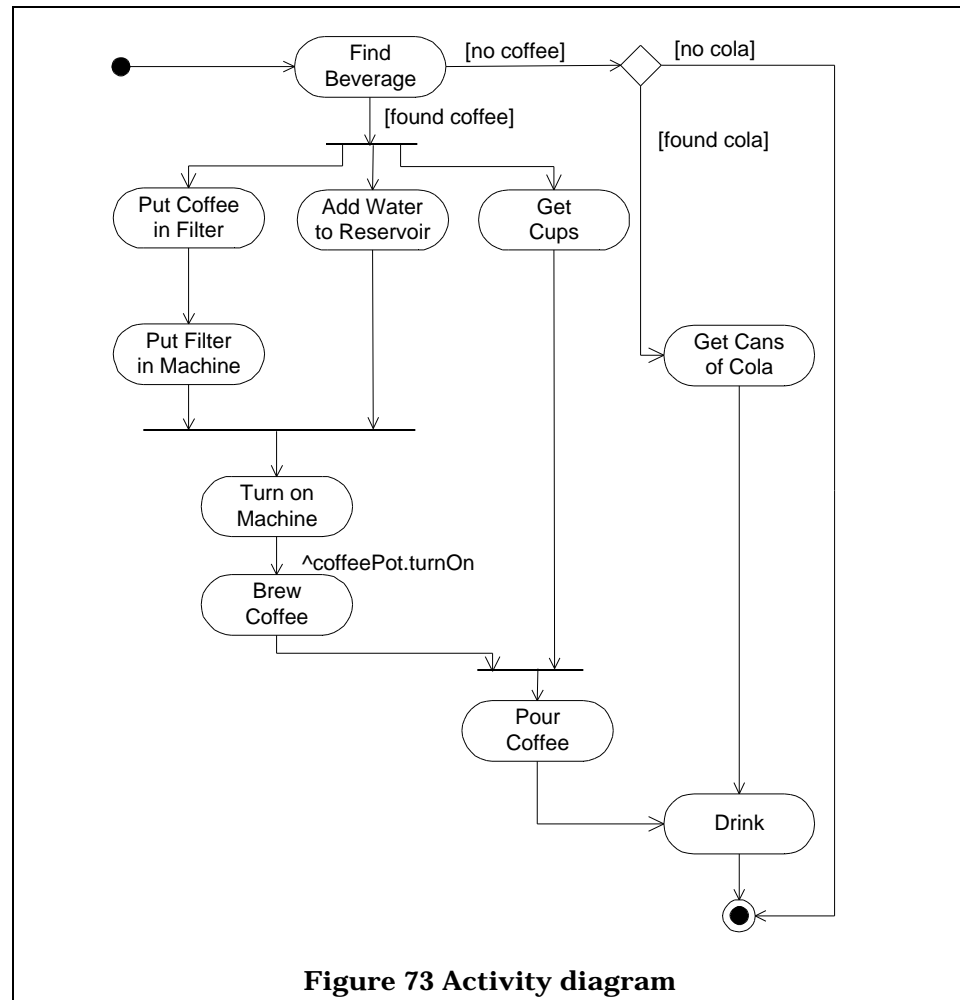
Figure 72 State diagram

Figure 71 shows an example of a state diagram. A state is represented by a rectangle with rounded corners. A state can also contain a specification of what will be done in this state (the bottom compartment). The biggest rounded rectangle in Figure 72 (Active) represents a superstate containing several states. An arrow represents a transition. The format of a transition is: *event-signature* [*guard-condition*] / *action-expression*. A circle and an arrow is used to represent default transition.

## 5.7 Activity diagrams

The activity diagrams purpose is to focus on flows driven by internal processing. The diagram is a special case of a state diagram in which all the states are action states and in which all of the transitions are triggered by completion of the actions in the source states. Use activity diagrams when all or most of the events represent the completion of internally-generated actions. Use ordinary state diagrams in situations where asynchronous events occur.

Figure 73 shows an example of an activity diagram. *Activity state* are shown as a shape with straight top and bottom and with convex arcs on the two sides. The *action-expression* is placed in the symbol. An activity state must have at least one outgoing transition. A diamond shape is used to represent a decision.

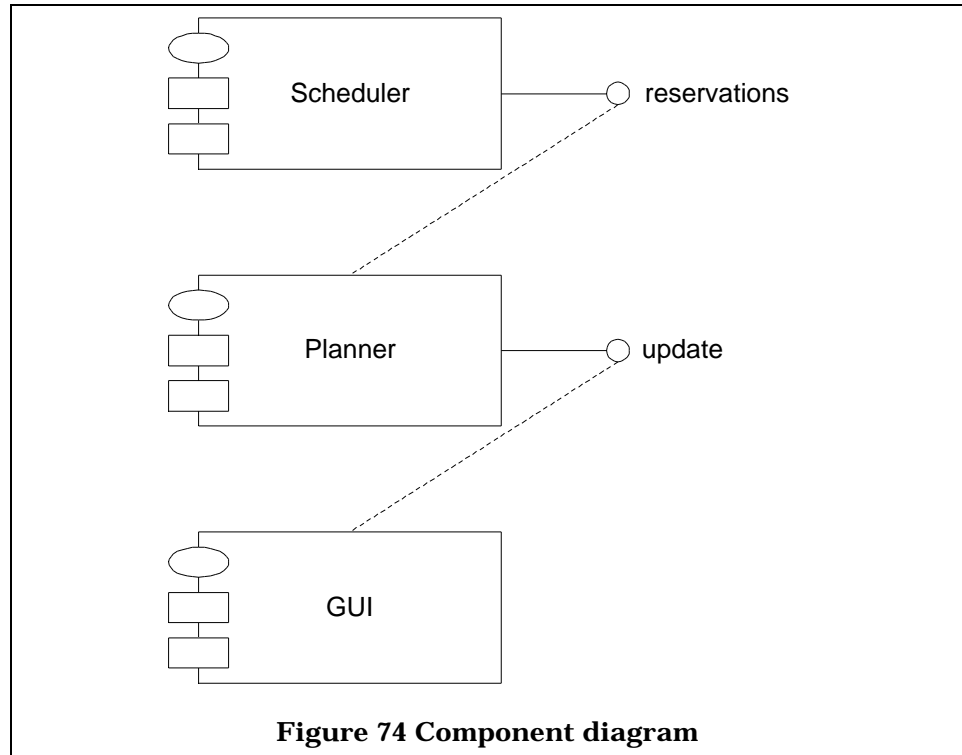


## 5.8 Implementation diagrams

Used to show aspects of implementation, including source code structure and run-time implementation structure. Implementation diagrams are divided into two subtypes of diagrams:

**Component diagram:** Shows the structure of the code itself. This diagram type shows the dependencies among software components, including source code components, binary code components, and executable components. A component diagram has only a type form, not an instance form.

Figure 74 shows an example of a component diagram. The dashed arrows are used to show dependencies. A component is represented as a rectangle with an ellipse and two rectangles connected to one of its sides. A line between the rectangle and a small circle is used to represent a component's interface.



Deployment diagram: Shows the structure of the runtime-system. This diagram shows the configuration on run-time processing elements and the software components, processes, and object that live on them.

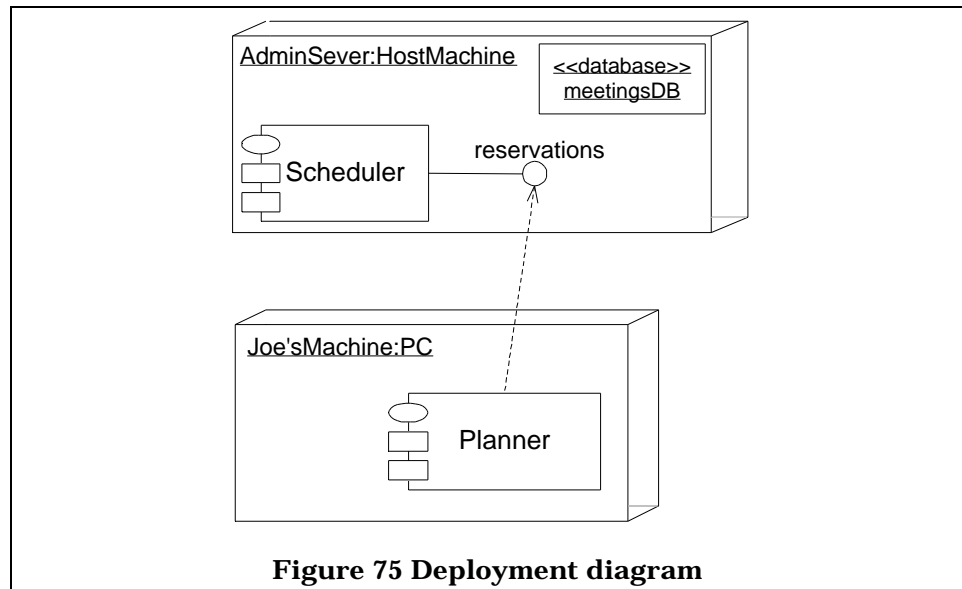


Figure 75 shows an example of a deployment diagram. The 3-dimensional view of a cube as shown in Figure 75 is used to represent a node which is a run-time physical object that represents a computational resource. A node can also contain components and objects. Objects are represented by a rectangle.

## **6 Appendix B: Description of the Library Example**

### **Contents**

- 6.1 Outline of a Case Study
- 6.2 Problem Description
- 6.3 Reorganisation of the Library with a View to Computerisation

### **Summary**

This chapter describes the case used as an example during this report. The case is a description of an old-fashioned manual library system which will be automated.

## 6.1 Outline of a Case Study

The case presented here is written by Benkt Wangler and is similar to the library case presented in appendix II of the article "A Comparative Evaluation of System Development Methods" by C. Floyd in the book "Information System Design Methodologies: Improving the Practice", North- Holland, 1986 (T.W. Olle et al., editors). Part of it is taken from a similar case developed by Janis Bubenko.

## 6.2 Problem Description

This section describes the problem to be solved.

### 6.2.1 Assumptions Concerning the Library

#### 6.2.1.1 Purpose and Organisational Environment

The function of a technical library is to collect books, register them and make them accessible to the staff of a firm. The specific orientation of the library depends on the particular fields dealt with by the various departments of the firm.

The firm consists of a number of self-contained departments. A department is a grouping-together of staff members all of whom are engaged in work in a specific field. The entire firm's staff uses the library. From the point of view of the organizational hierarchy the library is on the same level as the departments. There exists a skeleton plan for the expenditure of financial resources for the initial acquisition and enlargement of stock.

#### 6.2.1.2 Library Personnel

A chief librarian is in charge of the library. He is responsible to the management for resource expenditure in compliance with the skeleton plan. He has the responsibility for new acquisitions and for cataloguing.

A secretary performs the administrative duties (cataloguing, ordering books).

All departmental heads act as special consultants to the library in connection with the acquisition of new books and their systematic classification.

#### 6.2.1.3 Book Stock and Accessibility

The library has a stock of several thousand books (including works of which several copies are available). In addition, it keeps periodicals and project reports.

The book stock is located in a room accessible to all firm staff (open-access arrangement). The library personnel share a common working area, which is used exclusively for activities, connected with the library. Books may be

taken out of the library room. Periodicals are not registered and therefore may not be taken out.

## **6.2.2 Routine Procedures of the Existing Library**

### **6.2.2.1 Acquisition**

Acquisition consists of the selection and purchase of new books. All staff members are entitled to submit notes with suggestions for new acquisitions on paper cards. These notes are collected by the secretary who checks them periodically for relevance and, if necessary completes the details given.

The chief librarian then goes through the processed cards and decides which books are to be ordered. All books are, initially, ordered on approval. The secretary types a list of orders and sends this to the bookseller. Once received, the books are available for inspection in the library for approx. one month.

With due regard to the opinions of interested members of staff and/or the special consultants, the chief librarian selects the books to be purchased.

The secretary returns the unwanted books to the bookseller together with a list of the books selected for purchase. The relevant orders for payment are sent to the finance department, the invoice from the bookseller being filed away. The newly acquired books are, to begin with, displayed separately. Each year approx. 100-150 books are purchased.

### **6.2.2.2 Cataloguing**

The shelf catalogue lists books in the order of their arrangement on the shelves. Each book is assigned a classification number, which determines the physical location of the book within the library.

The shelf catalogue also serves as an inventory of book stock and therefore contains the price and date of purchase of each book. In addition, the catalogue serves as a record of the books borrowed.

In the alphabetical catalogue the books are listed according to formal criteria (author/editor and title) in alphabetical order.

The systematic catalogue comprises literature of similar subject matter, which is put into the context of its broader subject-area. Thus, it lists the books according to their content on a systematic basis, denoted formally by means of decimal classification.

To classify a new acquisition, the secretary prepares an index-card. Both book and index-card are given a classification number. With the help of the special consultants, the chief librarian classifies the book in accordance with the system used in the library.

All the catalogues take the form a card catalogues; the index-cards look like this:

Author's name, first name	Classification number
Subject title	
Subtitle	
Edition notation	
Supplement notation (illustrations, maps)	
Publication notation (place, publisher, year)	
Number of pages	
ISBN	Decimal classifiers

ISBN consists of four parts: geographical area no, publishing house no, title no and check digit. The total ISBN identifies uniquely a certain edition of a title. If a work (for instance an encyclopaedia) consists of several volumes, each volume will have a separate ISBN if they are sold separately, otherwise they will have the same ISBN.

### 6.2.2.3 Using the Library

By consulting the catalogues, the library user is able to obtain information about the book stock. Next to the catalogues is a key explaining the terms used in the systematic classification.

The procedure for borrowing books is carried out by the user himself. This is done by making the corresponding entry on the reverse of the relevant index-card in the shelf catalogue, giving the current date and name and telephone number of the user. There is also system of fines for failure to return books.

To return books, the user puts the borrowed book back in its place on the shelf and then cancels the corresponding entry in the shelf catalogue.

## 6.3 Reorganisation of the Library with a View to Computerization

### 6.3.1 General Considerations Regarding the Library System

The respective duties of the library personnel should remain essentially unaltered. The aim of the data processing system is to reduce the amount of administrative work as much as possible by minimising routine duties, in particular paper work. The system should serve to build up computerised records of book information and help the library personnel to cope with a continuously expanding stock.

Procedures enabling the user to gain information should be improved and accelerated. The procedure for borrowing books should be quick and simple. There should be facilities for keeping a statistical record of books taken out.

The system may be operated in single-user mode and it should be designed for occasional use by non-DP-specialists. The chief librarian is expected to be familiar with DP systems.

---

## 6.3.2 Book Stock and Accessibility

The book stock is expanded at the rate of approximately 10 new books per month. The arrangement of the books and the classification system remain unaltered. Bound catalogues taking the form of large volumes in which all the available titles are listed replace the card catalogues. As was previously the case, periodicals are not registered.

## 6.3.3 Revised Routine Procedures

### 6.3.3.1 Acquisition

Selection and purchase of new books are modified with respect to the manual procedure described in Section 6.2.1 in the following points:

- suggestions for new acquisitions are now entered directly into the system giving all known details of the book including comments, if desired;
- book selection is based on a suggestion list printed out by the system, displaying all available information and categorizing the suggestions as genuinely new, previously suggested but not purchased, new edition of book already in stock, and book already in stock;
- after completion of the system entries for books to be ordered, a system-generated list of orders on approval is sent by way of an order to the bookseller;
- The purchase of a book is based on a system-generated list of orders for purchase, which is sent to the bookseller and the finance department.

### 6.3.3.2 Cataloguing

All catalogues mentioned in Section 6.2.2 are now based on system entries for the books, the classification numbers and decimal classifiers being added by the secretary. Updated catalogues can be printed out whenever needed; they are available in the form of bound volumes for use by the library personnel and the library users; they secure access to the book stock in case of system failure.

In addition to the catalogues already available, a title catalogue listing book titles in alphabetical order is set up. Also, the system permits listing out partial catalogues, for example, a list of recent entries according to classification numbers as an aid to preparing classification number labels for new books.

### 6.3.3.3 Using the Library

Locating a book can now be carried out as a computer-aided procedure. The user can retrieve the classification number for a particular book from the system by title and/or author. If necessary, he is shown all the titles by a particular author to enable him to find the desired book. If the book has been taken out, he is shown by whom and from which date.

The system also supports the user in searching for literature on a specific subject by providing him with all catalogue entries belonging to a particular subject-area according to the decimal classification.

To borrow and return books, the user is now required to make an entry into the system. The system keeps a list of currently borrowed books for each staff member.

In case of system failure, borrowing and returning of books can be recorded informally on cards and transferred later into the system by the secretary.

#### **6.3.3.4 Additional Administrative Tasks**

In connection with the running of the library, a number of administrative tasks must be carried out periodically. They can now be performed more easily with the help of the DP system:

- inventory of the book stock to detect missing books is supported by the system-generated shelf catalogue;
- detection of dormant stock is supported by system-generated statistical records listing all books that have been taken out less than x times in a certain period;
- withdrawal of books which are found to be missing, dormant or no longer wanted is facilitated by a deletion function permitting consistent updating of all catalogues;
- extending the systematic classification is supported by a system function.

## 7 Appendix C: 4GL Notations

### Contents

- 8.1 Stereotypes used for modelling 4GL applications
- 8.2 Alternative notations/diagrams to model 4GL applications
- 8.3 Modelling UNIFACE applications with UML
- 8.4 How to find UML mappings for not regarded languages

### Summary

This chapter describes further details to model 4GL applications. First we summarize the stereotypes used to tailor UML. Then alternative notations are used, namely the OMT and MD notations. As the 4GL chapter is primarily guided by the 4GL SQLWindows we have added an additional section specialised to the 4GL UNIFACE. The last section describes how we can develop good modelling possibilities for other 4GLs not directly regarded in this report.

## 7.1 Stereotypes used for modelling 4GL applications

Table 16 summarises the stereotypes defined for tailoring UML to the needs of 4GL modelling especially SQLWindows modelling. The first column indicates the diagrams in which the stereotype can be. In the diagram column 'Class' means 'Class diagram' etc.. The second column contains the UML element which is classified with the stereotype. The third column contains the stereotype. There are stereotypes for all object types provided by SQLWindows. This is indicated by dots. The same is done for stereotyping classes. The last column indicates when to use this stereotype.

Diagram	UML Element	Stereotype	Usage
Component	Component Item	«app», «apl», «dll», «bmp»	to indicate the file type
Class	Class used to model 4GL classes	«form window class» «MDI window class» «list box class» «pushbutton class»	to indicate the 4GL class type
Class	Class used to model 4GL objects	«form window» «MDI window» «list box» «pushbutton»	to indicate the 4GL object type
Class	Class used to model format descriptions	«format definitions»	for a class collecting format definitions
Class	Attributes representing format descriptions	«format»	to indicate that the attribute is a format description
Class	Class used to model global constants	«constant definitions»	for a class collecting constant definitions
Class	Attributes representing constant definitions	«constant»	to indicate that the attribute is a constant description
Class	Class used to model resource descriptions	«resource definitions»	for a class collecting resource definitions
Class	Attributes representing resource descriptions	«resource»	to indicate that the attribute is a resource description
Class	Class used to model global	«variable	for a class collecting global

Diagram	UML Element	Stereotype	Usage
	variables	definitions»	variable definitions
Class	Attributes representing variables	«variable»	to indicate that the attribute is a variable definition; this stereotype is optional
Class	Class used to model global functions	«internal function definitions»	for a class collecting global functions
Class	Operations representing functions	«function»	to indicate that the operation is a function definition
Class	Class used to model application messages	«application message definitions»	for a class collecting application messages
Class	Operations representing messages	«message»	to indicate that the operation is a message definition
Class	Class used to model named menus	«named menu definitions»	for a class collecting named menus
Class	Operations representing named menu entries	«named menu»	to indicate that the operation is a named menu
Class	Class representing a database table	«table»	for a class describing a database table
Class	Operation representing a database index	«index»	to indicate that the operation is describing a database index
Class	Class representing a database view	«view»	for a class describing a database table
Class	Event Item		triggers
Class	Class used to model stored procedures	«stored procedure definitions»	for a class collecting stored procedures
Class	Operations representing stored procedures	«stored procedure »	to indicate that the operation is a stored procedure
Class	Class used to model domain definitions	«domain definitions»	for a class collecting domain definitions
Class	Attributes representing domains	«domain »	to indicate that the attribute is a domain definition

**Table 16 Stereotypes defined for tailoring UML to the needs of  
4GL modelling, especially SQLWindows modelling**

## 7.2 Alternative notations/diagrams to model 4GL applications

In this consultancy report we have used the Unified Modelling Language (UML) as basic notation for detailed and technical modelling. We are convinced by the technical features of UML to master the challenge of modelling evolving systems. The power of commitment for UML, shown by a lot of companies and organisations, can make UML to the coming object-oriented standard notation. Nevertheless we are currently living in a heterogeneous method and notation world. As the essential ideas of this report can also be applied with other notations we provide some alternative mappings in this section. If you are using one of the described notations you can use the results directly. If you are using a notation not mentioned directly it will help you to map the ideas to your favourite notation.

### 7.2.1 Modelling 4GL applications with OMT

As class diagrams are the main diagrams used for modelling 4GL applications and as UML raised from OMT, the mapping to OMT is quite straight-forward. Instead of using stereotypes to indicate meta-classes etc. we use textual annotations provided by the tools.

As nearly all object-oriented methods and notations provide class diagrams in some form the mapping can be used in a similar way.

<b>Basic principles of the 4GL - OMT mapping</b>
<ul style="list-style-type: none"> <li>✓ The central notation are the OMT class-diagrams. They are used to model the basic elements of our 4GL applications. They provide a direct support for modelling the classes and objects of the 4GL applications.</li> <li>✓ With the help of annotated text the class-diagrams can be tailored to the special needs, i.e. we can use annotations to indicate format descriptions, resources etc.. In addition annotations are used to indicate the 'type' of classes or objects, i.e. a class can be indicated as a 'pushbutton class' or a 'functional class'. If we are using straight naming conventions such annotations are not necessary.</li> <li>✓ In addition event-trace-diagrams are used as snapshots of typical dynamic interactions between objects.</li> <li>✓ Which logical elements (class, objects, etc.) to be implemented in which file can be described by textual annotations connected to the logical elements.</li> <li>✓ The window interaction is modelled by a state-transition diagram.</li> <li>✓ Subsystems are modelled directly as OMT subsystems.</li> <li>✓ The different database properties are modelled as annotated classes. This allows a good mapping between database elements and application elements using them.</li> </ul>

- ✓ Complementing textual descriptions are used to specify and describe further details, i.e. the purpose of a class or the parameter interface of a function.

Table 17 shows the mapping between 4GL properties and OMT elements that are used to model them. We also describe how the mapping can be operationalised by tailoring UML with the help of textual annotations. The first column contains the identified 4GL properties which must be modelled. The second column contains the used diagrams. In the diagram column 'Class' means 'Class diagram' etc.. The third column contains the OMT elements used to model the property and some hints to use textual annotations.

Some remarks concerning the OMT usage:

- We have not used every kind of diagram and notation detail provided by OMT. Instead we started from the properties which must be modelled and have identified appropriate OMT elements to model them.
- Annotations provided by the tools are used instead of stereotypes. The stereotype concept is much clearer from a conceptual view but the annotated texts provide a pragmatic alternative.
- The main software architecture should be modelled by an appropriate subsystem structure.
- There exists no equivalent to the component diagrams of UML. Here further method independent notations must be used if required.

4GL Property	OMT Diagram	OMT Element
Subsystems	Subsystem	Subsystems are used to model logical subsystems. A hierarchical structure with scope control is possible.
Files	Class	Textual annotations indicating the file.
File dependencies		
Format descriptions and references	Class	Attributes of a special class collecting the format descriptions. The class itself is annotated as «format definitions». The attributes are annotated as «format».
External functions and references	Class	Operations of a special class collecting the external function definitions. The class itself is annotated as «external function definitions». The operations are annotated as «external function».
Global constants and references	Class	Attributes of a special class collecting the constant definitions. The class itself is annotated as «constant definitions». The

4GL Property	OMT Diagram	OMT Element
		attributes are annotated as «constants».
Resource definitions and references	Class	Attributes of a class collecting the resource definitions. The class itself is annotated as «resource definitions». The attributes are annotated as «resource».
Variable definitions and references	Class	Attributes of a class or object. The attributes are optionally annotated as «variable» . Global variables are modelled as attributes of a class collecting the global variable definitions. The class itself is annotated as «variable definitions».
Functions and references	Class, Event-trace	Operations of a class or object. The operations are annotated as «function». Global functions are modelled as operations of a class collecting the global function definitions. The class itself is annotated as «internal function definition». Event-trace diagrams are used as snapshots of characteristic function interactions.
Messages and references	Class, Event-trace	Operations of a class or object. The operations are annotated as «message». Application messages are modelled as operations of a class collecting the global message definitions. The class itself is annotated as «application message definitions». Event-trace diagrams or collaboration diagrams are used as snapshots of characteristic message interactions.
Named menus and references	Class	Operations of a class collecting the resource definitions. The class itself is annotated as «named menu definitions». The operations are annotated as «named menu».
Classes	Class, Event-trace	Class. The class-type (pushbutton-class, frame-window-class, ...) is described by an appropriate annotation. The relationships between the classes are modelled with the corresponding OMT notations, i.e. associations, compositions and generalisations. Event-trace diagrams are used as snapshots of characteristic interactions.
Objects	Class, Event-trace	Class. The object-type (pushbutton, frame-window, ...) is described by an appropriate annotation. As SQLWindows objects can have own message and function definitions they are modelled as OMT classes with appropriate annotations. If the 4GL objects are pure instantiations of classes they can be modelled as OMT objects. Event-trace diagrams are used as snapshots of characteristic interactions.
User Interface	State-	The user interface is modelled as a state

4GL Property	OMT Diagram	OMT Element
	transition	transition diagram. The windows are represented by the states. The window activation by appropriate transitions.
Database Interface	Class	The relations between application elements and database elements are described by use relations between the application classes and objects and the database classes.
Internal data model		The internal data model is modelled as appropriate annotated classes in the same way like the RDBMS elements. The relations between the internal data model and the database is modelled by appropriate use dependencies.
Extended attributes		Properties and behaviour connected to attributes is modelled by complementary textual descriptions.
Tables and references	Class	Class which is annotated as «table». The table attributes are represented by the class attributes.
Indexes	Class	Operation of the class representing the corresponding table or view. The operations are annotated as «index».
Views and references	Class	Class which is annotated as «view». The view attributes are represented by the class attributes.
Triggers	Collaboration	Triggers are modelled in a regarded collaboration diagram.
Stored procedures and references	Class, Event-trace	Operations of a special class collecting the stored procedures. The class itself is annotated as «stored procedure definitions». The operations are annotated as «stored procedure». Event-trace diagrams are used as snapshots of characteristic stored procedure interactions.
Domains and references	Class	Attributes of a special class collecting the domain definitions. The class itself is annotated as «domain definitions». The attributes are annotated as domain».

**Table 17 Mapping between 4GL properties and OMT**

### 7.2.2 Modelling 4GL applications with MD

Modular Design (MD) is an advanced structured design method. It enhances the design method Structured Design (SD) by modularisation, information hiding and a subsystem concept. Today we can classify Modular Design as an object-based method. The results of this section can be transferred easily to other object-based, i.e. HOOD.

<b>Basic principles of the 4GL - MD mapping</b>
<ul style="list-style-type: none"> <li>✓ The central notations are the MD subsystem diagrams. They are used to model the basic elements of our 4GL applications. Most 4GL elements are modelled as MD modules.</li> <li>✓ With the help of annotated text the subsystem-diagrams can be tailored to the special needs, i.e. we can use annotations to indicate format descriptions, resources etc.. In addition annotations are used to indicate the 'type' of classes or objects, i.e. a class can be indicated as a 'pushbutton class' or a 'functional class'. If we are using straight naming conventions such annotations are not necessary.</li> <li>✓ Message and function interaction can be modelled with module diagrams.</li> <li>✓ Which logical elements (class, objects, etc.) to be implemented in which file can be described by textual annotations connected to the logical elements.</li> <li>✓ The window interaction is modelled by a state-transition diagram.</li> <li>✓ Subsystems are modelled as directly as MD subsystems.</li> <li>✓ The different database properties are modelled with complementing entity-relationship diagrams.</li> <li>✓ Complementing textual descriptions are used to specify and describe further details, i.e. the purpose of a class or the parameter interface of a function.</li> </ul>



Some remarks concerning the MD usage:

- Annotations provided by the tools are used instead of stereotypes. The stereotype concept is much clearer from a conceptual view but the annotated texts provide a pragmatic alternative.
- The main software architecture should be modelled by an appropriate subsystem structure.
- There exists no equivalent to component diagrams of UML. Here further method independent notations must be used if required.
- As most 4GL are going towards OO, the usage of a structured method is not so satisfying. If you have no choice this not-optimal mapping and modelling is better than no modelling. If you have a choice the OO choice is most times the better one.
- Entity relationship diagrams are a good notation for modelling database properties. They can be an alternative to model them. In this case the traceability problem between 4GL properties modelled with an OO notation and database properties modelled with the entity relationship notation must be solved.

Table 18 shows the mapping between 4GL properties and MD elements that are used to model them. We also describe how the mapping can be operationalised by tailoring MD with the help of textual annotations. The

first column contains the identified 4GL properties which must be modelled. The second column contains the used diagrams. In the diagram column 'Subsystem' means 'Subsystem diagram' etc.. The third column contains the MD elements used to model the property and some hints to use textual annotations.

4GL Property	MD Diagram	MD Element
Subsystems	Subsystem	Subsystems are used to model logical subsystems. A hierarchical structure with scope control is possible.
Files	Subsystem	Textual annotations indicating the file.
File dependencies		
Format descriptions and references	Subsystem	Constants of a special module collecting the format descriptions. The module itself is annotated as «format definitions». The constants are annotated as «format».
External functions and references	Subsystem	Functions of a special module collecting the external function definitions. The module itself is annotated as «external function definitions». The functions are annotated as «external function».
Global constants and references	Subsystem	Constants of a special module collecting the constant definitions. The module itself is annotated as «constant definitions».
Resource definitions and references	Subsystem	Constants of a module collecting the resource definitions. The module itself is annotated as «resource definitions». The constants are annotated as «resource».
Variable definitions and references	Subsystem	Variables of a module. Global variables are modelled as variables of a module collecting the global variable definitions. The module itself is annotated as «variable definitions».
Functions and references	Subsystem, Module	Functions of a module. Global functions are modelled as functions of a module collecting the global function definitions. The module itself is annotated as «internal function definition». Module diagrams are used for the function interactions.
Messages and references	Subsystem, Module	Functions of a module or object. The functions are annotated as «message». Application messages are modelled as functions of a module collecting the global message definitions. The module itself is annotated as «application message definitions». Module diagrams are used for the message interactions.

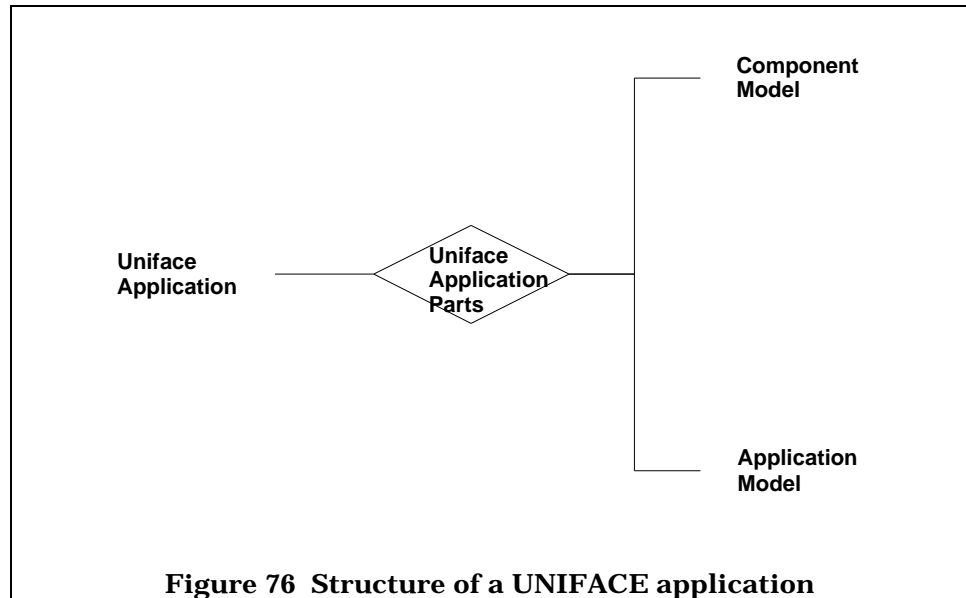
4GL Property	MD Diagram	MD Element
Named menus and references	Subsystem	Functions of a module collecting the resource definitions. The module itself is annotated as «named menu definitions». The functions are annotated as «named menu».
Classes	Subsystem, Module	Module. The class-type (pushbutton-class, frame-window-class, ...) is described by an appropriate annotation. The relationships between the classes are modelled as use-relations. Module diagrams are used for the dynamic interactions.
Objects	Subsystem, Module	Module. The object-type (pushbutton, frame-window, ...) is described by an appropriate annotation. Module diagrams are used for the dynamic interactions.
User Interface	State-transition	The user interface is modelled as a state transition diagram. The windows are represented by the states. The window activation by appropriate transitions.
Database Interface		
Internal data model	Entity relationship	As appropriate entity-relationship diagram.
Extended attributes	Entity relationship	Properties and behaviour connected to attributes is modelled by complementary textual descriptions.
Tables and references	Entity relationship	Entity which is annotated as «table». The table attributes are represented by the entity attributes. Relationships are used to model the relationships between the entities.
Indexes	Entity relationship	Textual annotations connected to the corresponding entities.
Views and references	Entity relationship	Entity which is annotated as «view». The view attributes are represented by the entity attributes.
Triggers	Entity relationship	Textual annotations connected to the corresponding entities.
Stored procedures and references	Subsystem, Module	Functions of a special module collecting the stored procedures. The module itself is annotated as «stored procedure definitions». The functions are annotated as «stored procedure». Module diagrams are used for the dynamic interactions.
Domains and references	Entity relationship	Textual annotations connected to the corresponding entity attributes.

**Table 18 Mapping between 4GL properties and OMT**

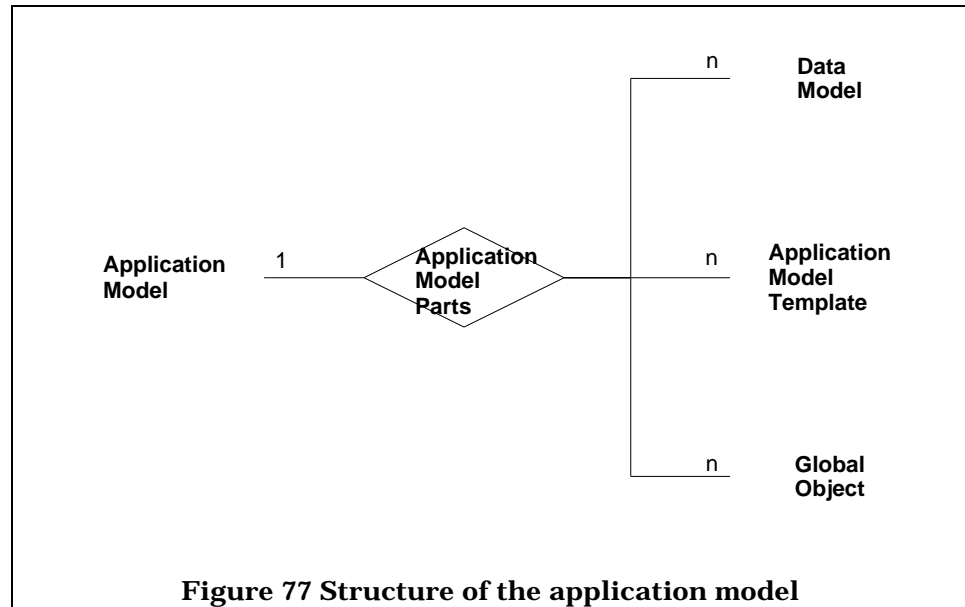
## 7.3 Modelling UNIFACE applications with UML

### 7.3.1 UNIFACE properties to be modelled

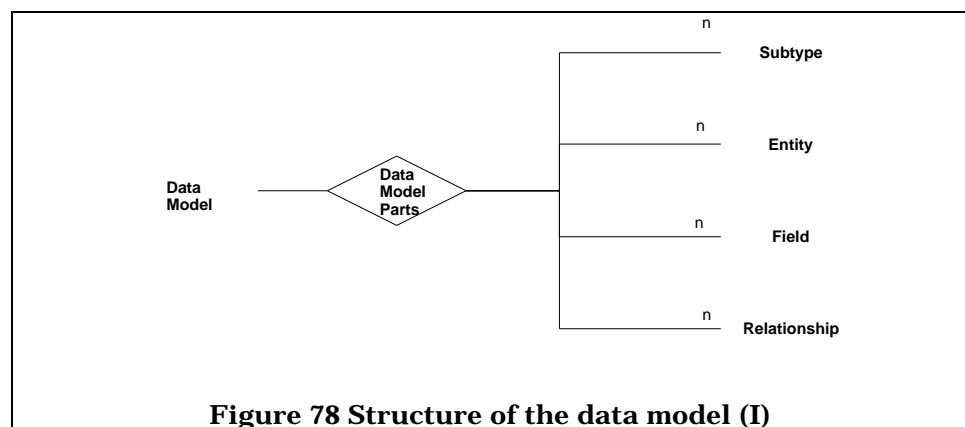
An UNIFACE **application** consists of two parts, the application model and the component model. The parts plus a domain specific refinement should be documented. It should be described which parts (subsystems, packages) exist, their interfaces, dependencies and purposes.

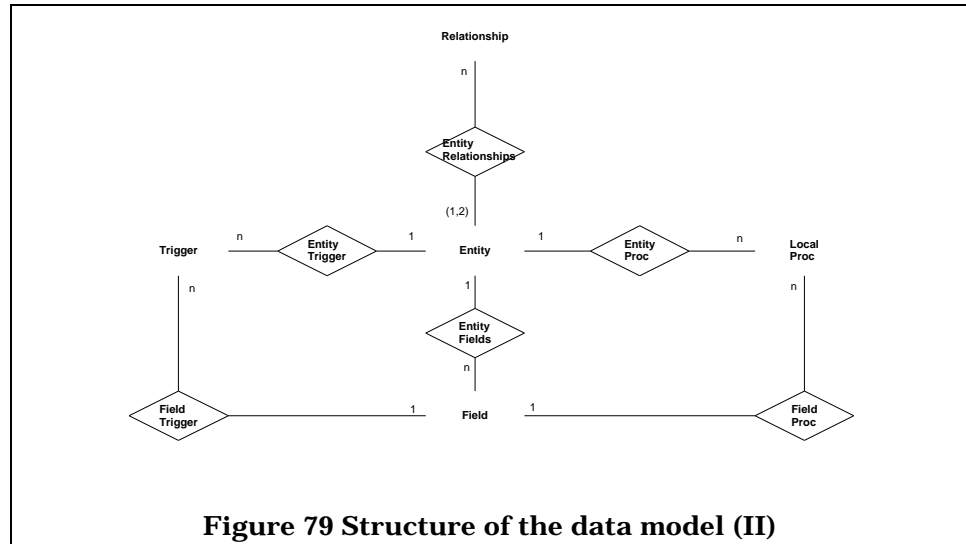


The heart of any UNIFACE application is the **application model**. The application model consists of the data model (entities, relationships, ...), application model templates and global objects. UNIFACE allows to have one or multiple underlying application models. Multiple are a good structuring possibility to force modularity and incremental implementation.

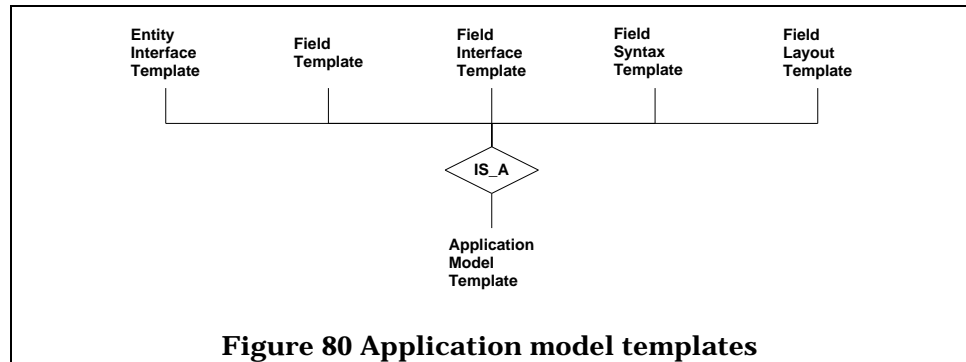


The **data model** contains the underlying entities, fields, relationships and subtypes. For an entity we can describe its purpose and the fields of the entity. For the single fields we can describe their purpose and their type. In addition the relationship between the entities should be described in the design model. If we use subtypes their purpose should also be documented. If we define trigger and local procs they should be documented. Beside the description of the elements themselves we can also the other UNIFACE elements which are using them. These cross-reference list will be very helpful in future maintenance and evolution phases.

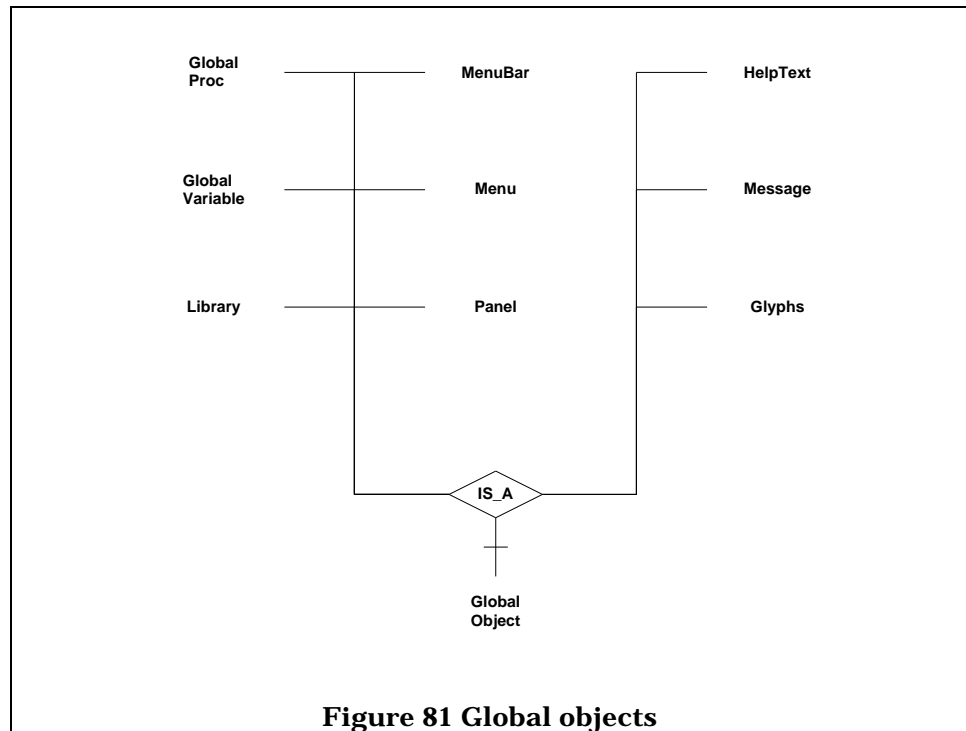




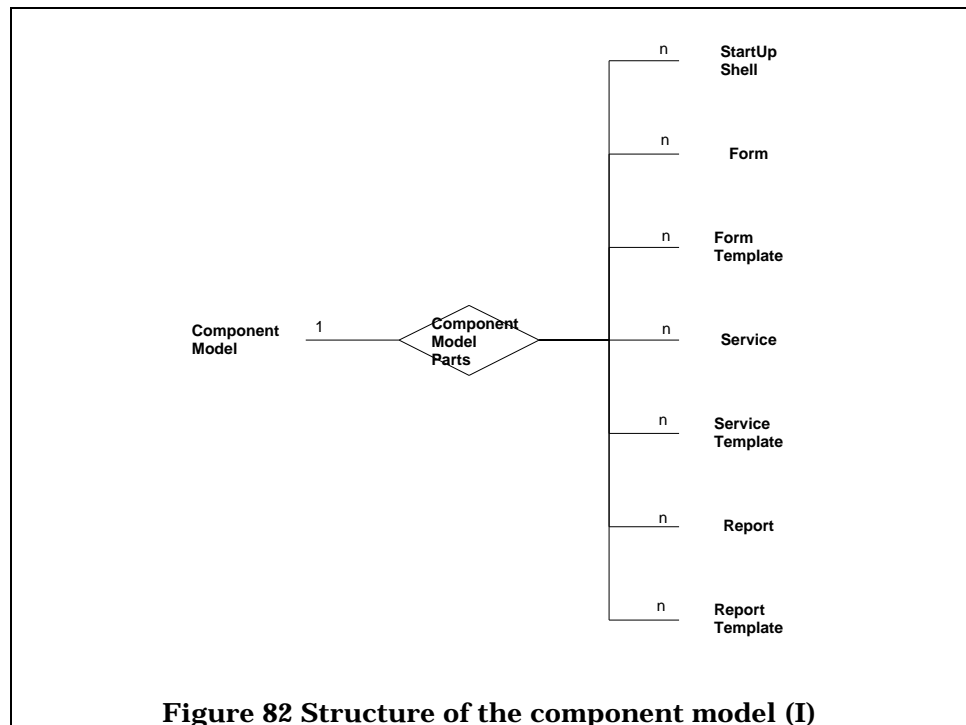
In the UNIFACE application model we can often choose between a direct specification and an indirect specification with templates. **Application model templates** for entity interface specifications, field specifications, field interface specifications, field syntax specifications and field layout specifications are available. If we use templates we should describe which templates exist, their objectives and purposes and where they are used.



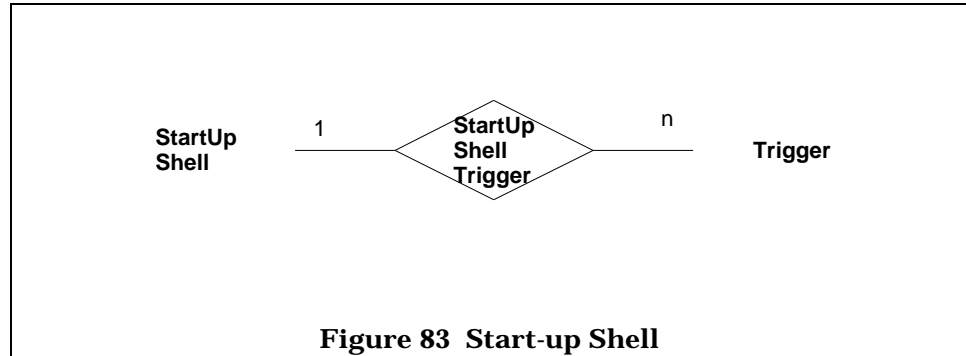
The third part of the application model is the global objects. **Global objects** allow the central definition of menus, panels, etc. In addition we can define global procs and variables. If we use global objects we must document which global objects exist, their objectives and purposes and where they are used.



The component model consists of forms, services and reports. For each we can also define templates which are then instantiated. In addition we have the start-up shells which start the applications. A good documentation of the existing components and their purposes is important. In addition the interfaces and dependencies between the components should be documented.

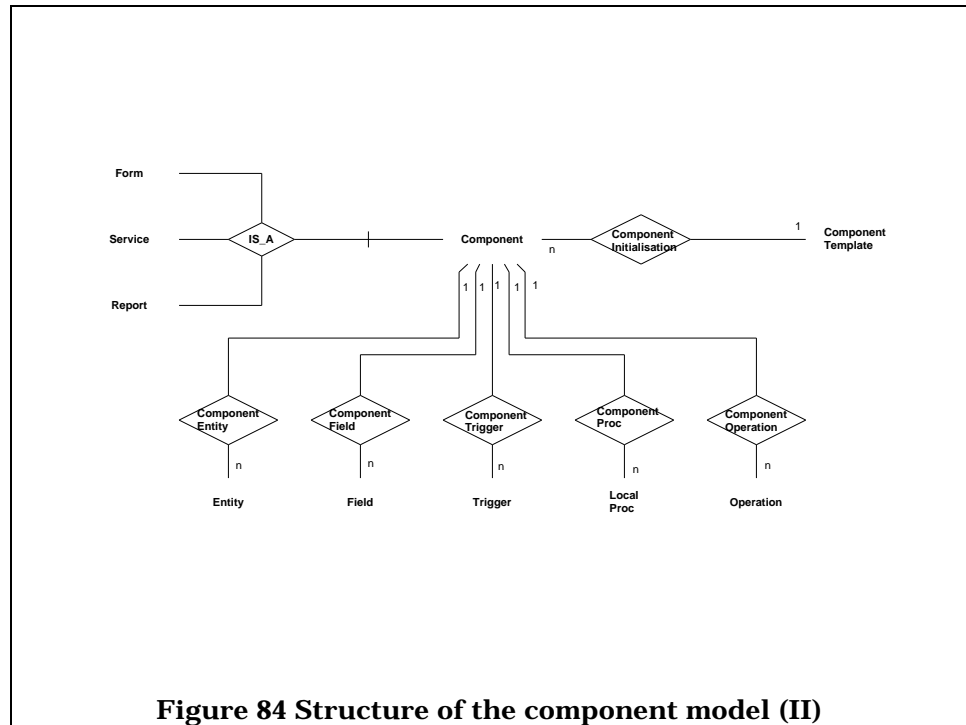


The start-up shell contains triggers. It should be documented which triggers are defined and which other forms, global procs, etc. are activated by the start-up shell and its triggers.



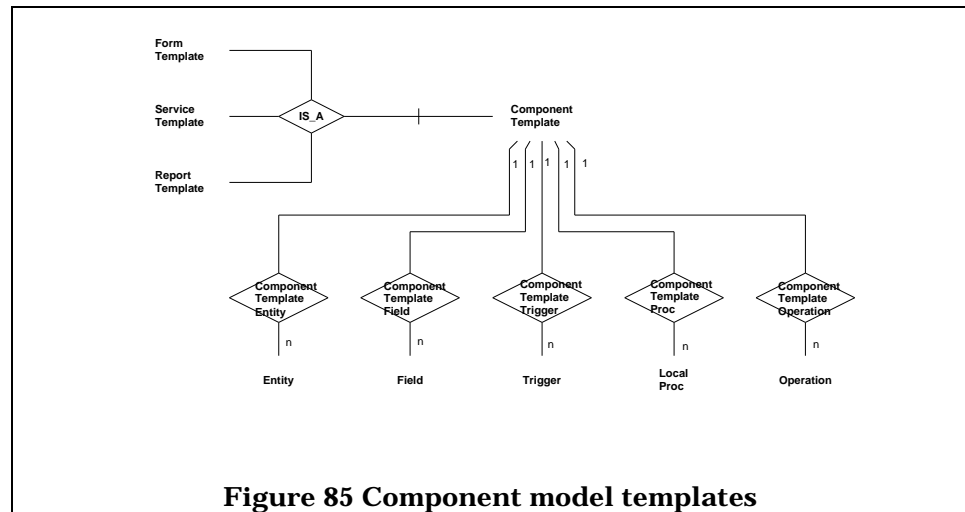
**Figure 83 Start-up Shell**

The following figure illustrates the main elements of the component model and their relations. For each component we should describe its purpose and the entities and fields it is using. If it activates other components we should also document this. The internal structure of a component (trigger, local operations, local variables, etc.) should be documented as needed.



**Figure 84 Structure of the component model (II)**

The component model templates have the same structure as the components. So we should also document the properties described above. In addition we should describe which component is instantiated from which template.



The following table summaries the main properties of an UNIFACE application. These properties are important for designing an UNIFACE application and a good documentation will be very helpful for the future maintenance and evolution of the application.

Property
Entity
Field
Subtype
Relationship
Entity Interface Template
Field Template
Field Interface Template
Field Syntax Template
Field Layout Template
Global Proc
General Purpose Variable
Global Variable
Library
Menu Bar
Menu
Panel
HelpText
Message
Glyphs
Trigger
Operation
Local Proc
Local Variable

Property
Start-up Shell
Form
Form Template
Service
Service Template
Report
Report Template

**Table 19 UNIFACE Properties**

### 7.3.2 Modelling UNIFACE applications with UML

This subsection is structured in the following way:

- First we describe the basic principles of the UNIFACE - UML mapping and explain how we use UML.
- Afterwards a table shows the mapping between UNIFACE properties and UML elements that are used to model them.

Basic principles of the UNIFACE - UML mapping
<ul style="list-style-type: none"> <li>• The central notation are the UML class-diagrams. They are used to model the basic elements of our UNIFACE applications.</li> <li>• With the help of stereotypes the class-diagrams can be tailored to the special needs, i.e. we can use stereotyped classes to model entities, forms, services and reports. Additional stereotyped classes are introduced to model global procs, global variables, menu definitions, etc.</li> <li>• In addition sequence-diagrams or collaboration diagrams are used as snapshots of typical dynamic interactions.</li> <li>• Subsystems are modelled as packages.</li> <li>• Complementing textual descriptions are used to specify and describe further details, i.e. the purpose of a class or the parameter interface of a operation.</li> </ul>

For a good understanding of the mapping and usage of UML the following notes are important:

- We have not used every kind of diagram and notation detail provided by UML. Instead we have started from the properties which have to be modelled. We have identified appropriate UML elements to model them.
- Stereotypes are a very powerful UML concept. We have used them to tailor UML to a special domain, namely UNIFACE modelling. The used stereotypes and the implied classification are described later in this section.
- The stereotype concept is not fully implemented in every tool. If stereotypes are not supported, a pragmatic approach to indicate these

types (meta-classes) should be used. If the type is clear no further indication is needed. If the type should be highlighted free text is a possible solution.

- Collaboration and sequence diagrams provide the same underlying information. It is a personal and cultural question which kind of diagrams are preferred and we will give no restrictive rule. Often one diagram can be transformed automatically into the other by the modelling tool. We have used sequence diagrams to describe interactions arranged in time sequence. Collaboration diagrams are used if the interaction is arranged around objects and if the time sequence must not be highlighted.

The following table shows the mapping between UNIFACE properties and UML elements that are used to model them. We also describe how the mapping can be operationalised by tailoring UML with the help of stereotypes. The first column contains the identified UNIFACE properties which must be modelled. The second column contains the used diagrams. In the diagram column 'Class' means 'Class diagram' etc.. The third column contains the UML elements used to model the property and some hints to use stereotypes.

UNIFACE Property	UML Diagram	UML Element
Entity	Class	Class. The class is stereotyped as «entity».
Subtype	Class	Class. The class is stereotyped as «subtype».
Field	Class	Attributes of a class. The attributes are stereotyped as «field».
Relationship	Class	Association between the regarded classes.
Entity Interface Template	Class	Attributes of a special class collecting entity and field templates. The class itself is stereotyped as «template definitions». The attributes are stereotyped as «entity interface template».
Field Template	Class	Attributes of a special class collecting entity and field templates. The class itself is stereotyped as «template definitions». The attributes are stereotyped as «field template».
Field Interface Template	Class	Attributes of a special class collecting entity and field templates. The class itself is stereotyped as «template definitions». The attributes are stereotyped as «field interface

UNIFACE Property	UML Diagram	UML Element
		template».
Field Syntax Template	Class	Attributes of a special class collecting entity and field templates. The class itself is stereotyped as «template definitions». The attributes are stereotyped as «field syntax template».
Filed Layout Template	Class	Attributes of a special class collecting entity and field templates. The class itself is stereotyped as «template definitions». The attributes are stereotyped as «field layout template».
Global Proc	Class Sequence Collaboration	Operations of a special class collecting the global proc definitions. The class itself is stereotyped as «global proc definitions». The operations are stereotyped as «global proc». Sequence diagrams or collaboration diagrams are used to as snapshots of characteristic interactions.
General Purpose Variable	Class	Attributes of a special class collecting the general purpose variables. The class itself is stereotyped as «global purpose variables». The attributes are stereotyped as «general purpose variable».
Global Variable	Class	Attributes of a special class collecting the global variable definitions. The class itself is stereotyped as «global variable definitions». The attributes are stereotyped as «global variable».
Library	Class	Package
Menu Bar	Class	Operations of a special class collecting the menu and menu bar definitions. The class itself is stereotyped as «menu definitions». The operations are stereotyped as «menu bar».
Menu	Class	Operations of a special class collecting the menu and menu bar definitions. The class itself is stereotyped as «menu definitions». The operations are

UNIFACE Property	UML Diagram	UML Element
		stereotyped as «menu».
Panel	Class	Operations of a special class collecting the panel definitions. The class itself is stereotyped as «panel definitions». The operations are stereotyped as «panel».
HelpText	Class	Attributes of a special class collecting the help text definitions. The class itself is stereotyped as «help text definitions». The attributes are stereotyped as «help text».
Message	Class	Attributes of a special class collecting the message definitions. The class itself is stereotyped as «message definitions». The attributes are stereotyped as «message».
Glyphs	Class	Attributes of a special class collecting the glyph definitions. The class itself is stereotyped as «glyph definitions». The attributes are stereotyped as «glyph».
Trigger	Class Sequence Collaboration	Operations of a class. The operations are stereotyped with as «trigger».  Sequence diagrams or collaboration diagrams are used to as snapshots of characteristic interactions.
Operation	Class Sequence Collaboration	Operations of a class. The operations are stereotyped with as «operation».  Sequence diagrams or collaboration diagrams are used to as snapshots of characteristic interactions.
Local Proc	Class Sequence Collaboration	Operations of a class. The operations are stereotyped with as «local proc».  Sequence diagrams or collaboration diagrams are used to as snapshots of characteristic interactions.
Local Variable	Class	Attributes of a class. The attributes are stereotyped with as «local variable».
Start-up Shell	Class	Class. The class is stereotyped

UNIFACE Property	UML Diagram	UML Element
		as «startup shell».
Form	Class	Class. The class is stereotyped as «form».
Form Template	Class	Class. The class is stereotyped as «form template».
Service	Class	Class. The class is stereotyped as «service».
Service Template	Class	Class. The class is stereotyped as «service template».
Report	Class	Class. The class is stereotyped as «report».
Report Template	Class	Class. The class is stereotyped as «report template».
Subsystems	Class Component	Packages are used in class diagrams to model logical subsystems.  In component diagrams packages are used to model directories or groups of files.
User Interface	Collaboration	The user interface is modelled as a collaboration diagram. The windows (forms) are represented by the objects. The window activation by appropriate use-relations.

**Table 20 UNIFACE - UML Mapping**

## 7.4 How to find UML mappings for not regarded languages

On a first look there are a lot of differences between the different 4GLs but if we take a closer look at their structure they are mainly based on the same concepts. Therefore the described modelling ideas can be transferred and tailored to an arbitrary 4GL in an easy way. The following process was successful proved:

1. Identify the properties which must be modelled
  - Identify the main concepts, structures and elements of the 4GL. These can be documented well with UML class-diagrams itself or entity relationship diagrams like in this consultancy report.
  - Define which of these must be modelled and what should be documented. Here you should also decide the detail level of your modelling.
2. Define an UML mapping for these properties. Often the whole mapping is based on only a few basic mapping ideas. If you know how to

represent the central properties in UML the rest will follow automatically. The mapping examples of this report can be helpful.

3. Test the modelling on an example. Test on a limited example if you can model and document everything you want to document. In addition test if it is practicable, i.e. needed effort, possible tool support etc.
4. If necessary improve the modelling by repeating the previous steps. To find a good 4GL modelling notation some discussions and iterations are necessary.

Notes:

- The properties to be modelled and the UML mapping should be documented.
- When you identify the properties, define the mapping, etc. you learn a lot of new aspects of the used 4GL which often go down in the daily work. After this work you will have a lot of new ideas for using the 4GL.

Instead of UML you can also use your favourite method/notation if it supports the necessary concepts. The key question is not which notation to use. The key question is what do we want to model.



## 8 Appendix D: Database Modelling

### Contents

- 8.1 Analysis Model
- 8.2 Design Model

### Summary

The objective of this appendix is to describe modelling techniques for database evolution and development. This includes beside the statically database parts (tables, attributes, etc.) also the dynamic parts (packages, stored procedures, etc.). We have focused our work to the design phase. In addition, we have also described possible ways to manage the analysis phase. When starting this work we have a unique approach for all projects in mind. This seems to be unrealistic. Instead we describe different approaches and give hints for selecting the optimal approach in a concrete project. We have not developed a new way of database modelling. Instead, we have used a pragmatic approach by collecting and using current project practice.

A structured and an object-oriented way to create an analysis model are first described in section 8.1. Then we describe a combined approach integrating the best of both.

Section 8.2 starts with identifying design activities and database elements that must be specified in the design step. Afterwards different

available notations and tools are described. A final section of this chapter compares the different notations and gives some recommendations for using them.

## 8.1 Analysis Model

The analysis model must describe what should be done by the application. First, we describe a structured approach for creating an analysis model, then an object-oriented approach, and finally a combined approach.

### 8.1.1 Structured analysis model

Analysis property	Description
Entities	Name, Purpose, Primary keys, important attributes
Relationships	Name, Purpose, Cardinality
Functionalities	Classical SA

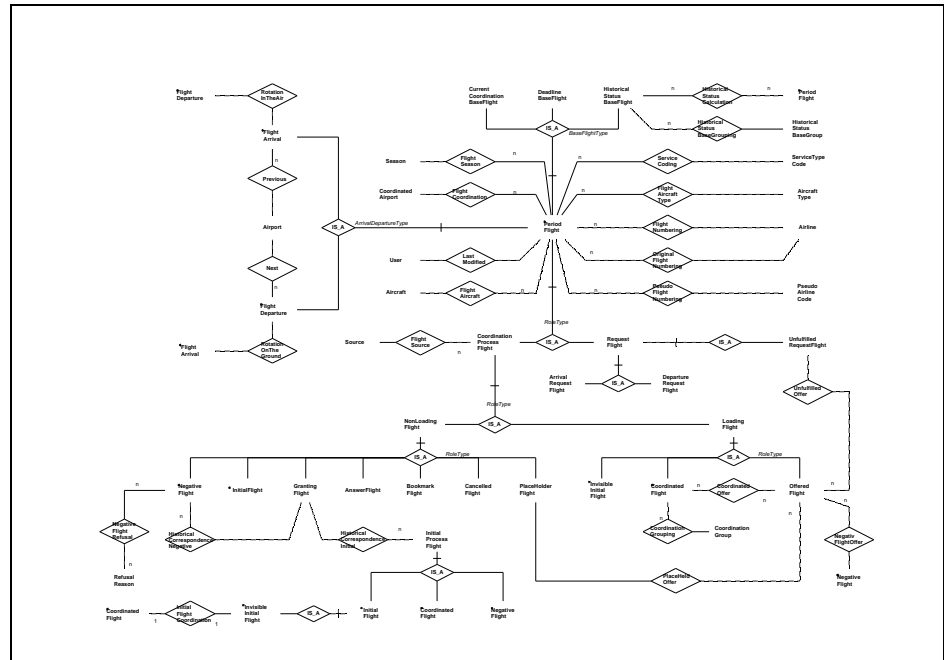
If the analysis model is produced with structured methods, we can use the following tool support:

- The data view is modelled with ProMod<sup>PLUS</sup> IM.
- The functional view is modelled with ProMod<sup>PLUS</sup> SA.

#### Data view:

- Focus on single use-cases (scenarios).
- One ERD for each scenario.
- The ERD should describe the entities and relationships.
- For each Entity we must describe its name, purpose (what, responsibility), the primary keys and business rules.
- For each relationship, its name, the participating entities and the cardinalities must be described.

Figure 86 shows a typical ERD of the analysis model.



**Figure 86 Typical ERD of the analysis model.**

A typical entity description can be found in the following box.

<p>*</p> <p>This entity represents an aircraft type. This is either a general aircraft type or one customised for a specific airline. A general aircraft type may have only customisation per airline.</p> <p>Note that the following are referred to via relationships</p> <ol style="list-style-type: none"> <li>1. Annex 16 classification</li> <li>2. drive type</li> <li>3. Weight classification</li> <li>4. SizeType</li> </ol> <p>*</p> <p>= # "AircraftTypeId"</p> <p>+ @ "AircraftTypeCode_3L"</p> <p style="padding-left: 20px;">*3 Letter IATA Code *</p> <p>+ @ "AircraftTypeCode_4L"</p> <p style="padding-left: 20px;">*4 Letter IATA Code *</p> <p>+ @ "AircraftTypeDescription"</p> <p style="padding-left: 20px;">* Long description of the aircraft type *</p> <p>+ @ "MTOW"</p> <p>+ @ "NumberOfSeats"</p> <p>+ @ "NoisePoints"</p>
---

**Functional view:**

The functionality is described with SA. There is no difference in the analysis model if the functionality will be implemented with database features, i.e. stored procedures, or with 4GL features.

A problem was the missing mapping between the functional and the data view.

**8.1.2 Object-oriented analysis model**

Analysis property	Description
Objects, classes	Name, Purpose, Important attributes
Relationships	Name, Purpose, Cardinality
Functionalities	Informal text
Events	Informal text

If the analysis model is produced with object-oriented methods, we can use the following tool support:

- ProMod<sup>PLUS</sup> (OMT)
- Rational-Rose (UML)

The following aspects are important for creating the analysis model:

- Focus on single use-cases (scenarios).
- Use-Case diagram can be used to show which use-cases are available. The relations between the different use-cases can also be described.
- For each use-case, a sequence diagram can be used to describe the events. Participating objects and their functionalities.
- The relationships between the objects can be described with class diagrams.

**8.1.3 Combined analysis model**

In this section a combined approach is described. This approach collects the best ideas from the structured and object-oriented world.

1. Identify the use-cases and draw one or several use-case diagrams.
2. Use unique textual templates for describing the use-cases.
3. For each use-case you can model a Sequence diagram and/or an activity diagram describing the functionalities and possible events of this use-case.
4. For each use-case you can model an ERD describing the underlying data structure.
5. Prepare a context diagram identifying and specifying as detailed as necessary the system interfaces.

6. If necessary use high-level data flow diagrams to show the data flow between the different use-cases.

## 8.2 Design Model

The design model must describe how the system is realised and must be a construction plan for the (complete) database. Therefore it should not only describe the static database part (tables, attributes, etc.) but also the dynamic parts (packages, stored procedures, etc.). In Section 8.2.1 we describe what must be done in the design activity, and identify the database elements that should be modelled. In Section 8.2.2 we describe notations to model them.

### 8.2.1 Design activities and database elements

#### 8.2.1.1 What must be done in the design activity?

In this section we describe what must be done during database design.

##### Architecture (high level)

- The overall architecture of the database must be described.
- The strategy for error handling must be described.
- The locking and multi-user handling must be described.
- The strategy for transaction management must be described.
- The structure of physical processes at run-time must be described. The communication mechanisms must be described.
- Naming conventions must be specified.
- Documentation guidelines must be specified.
- Programming guidelines must be specified.
- Interfaces to other applications using the database must be specified
- The implementation tools must be specified.

##### Design (detailed level)

- The important database elements (see below) must be described. If necessary, the level of detail must be so fine that another person can implement it.

#### 8.2.1.2 Database elements

The following database elements should be described by the design model.

Database Property	Description	Optimising
<b>Static aspects</b>		
Tables	Name Purpose Number of entries	
Attributes	Name	

<b>Database Property</b>	<b>Description</b>	<b>Optimising</b>
	Type Purpose	
Views	Name Purpose Used tables	
Constraints	Value ranges Not-null constraints	
Referential integrity	Foreign table and key(s)	
Cardinalities		
Alternate keys	Participating attributes	
Indices	Name Participating table and attributes Number of entries	X
Physical distribution Table spaces (Oracle) Segments (Sybase)	Name Purpose Contained tables	X
Synonyms		
<b>Dynamic aspects</b>		
Triggers	Purpose Activation (Table, Attribute, Operation) Event( Activated stored procedure, manipulated tables, ...)	
Stored procedures	Name Purpose Provided interface Used interfaces Scope	
Packages (Oracle)	Name Purpose Provided interface Used interfaces	
Global Variables	Name Type Purpose Scope	

<b>Database Property</b>	<b>Description</b>	<b>Optimising</b>
Constants	Name Type Value Purpose Scope	
Key Generators (Oracle)		
Types	Name Purpose Structure	
<b>Further aspects</b>		
Domains	Name Type Purpose Who is using it	
Subsystems	Name Purpose Use-relations Contained elements	
Privileges and rights		
Distributed databases		
Active servers		
Processes and tasks		

## 8.2.2 Notations - Tools

### 8.2.2.1 Bachmann - ERWin

Figure 87 shows an ERWin diagram of the design model.

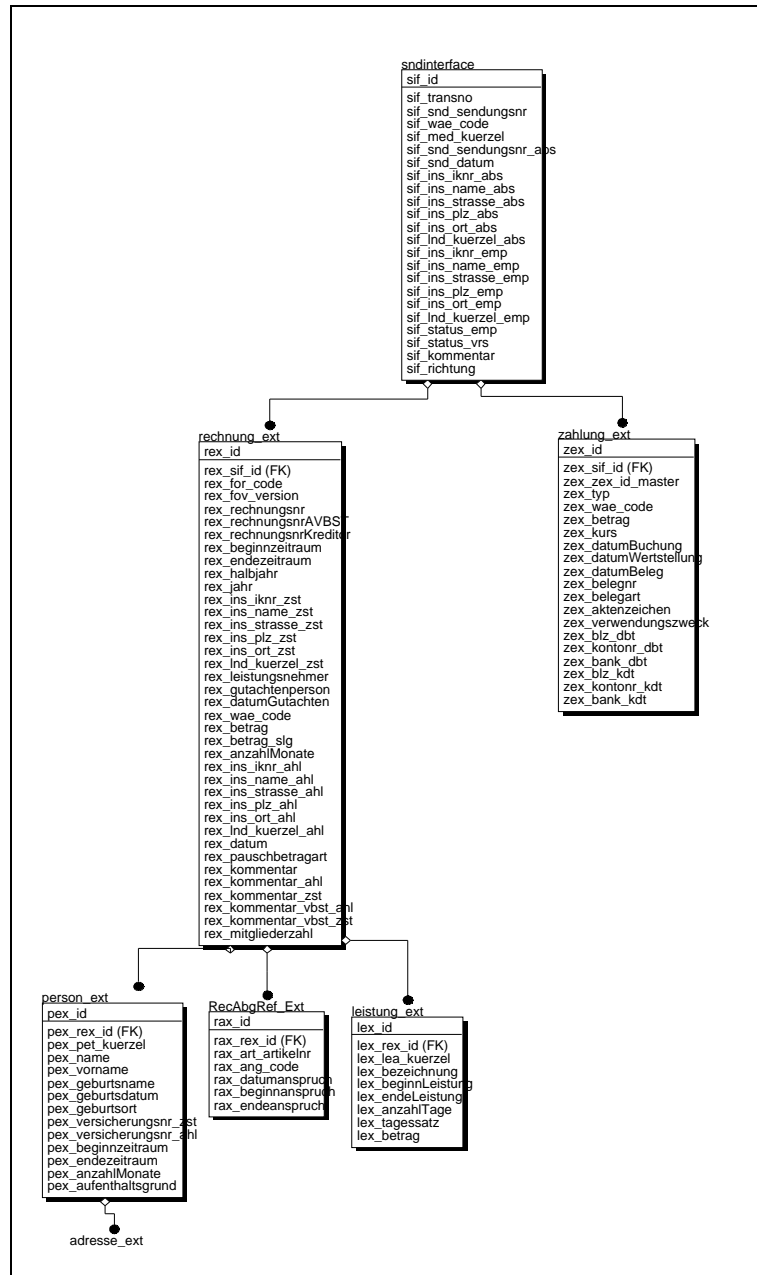


Figure 87 ERWin diagram of the design model



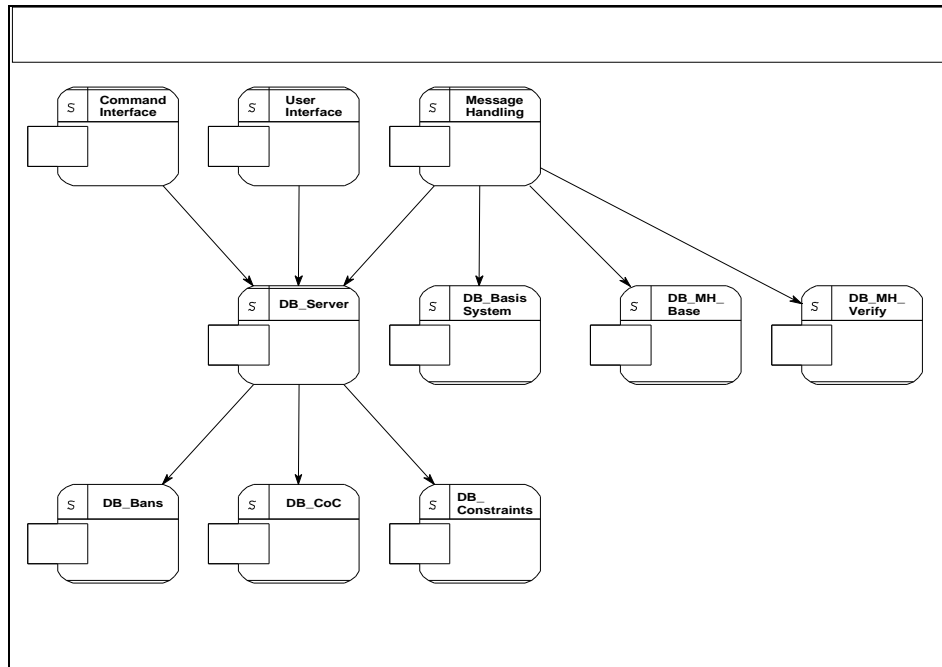
```

+ @NumberOfSeats
+ @NoisePoints
+ @SizeClass
+ @WeightClass
+ @NoiseCategory
+ @CustomInd
+ @DriveType

```

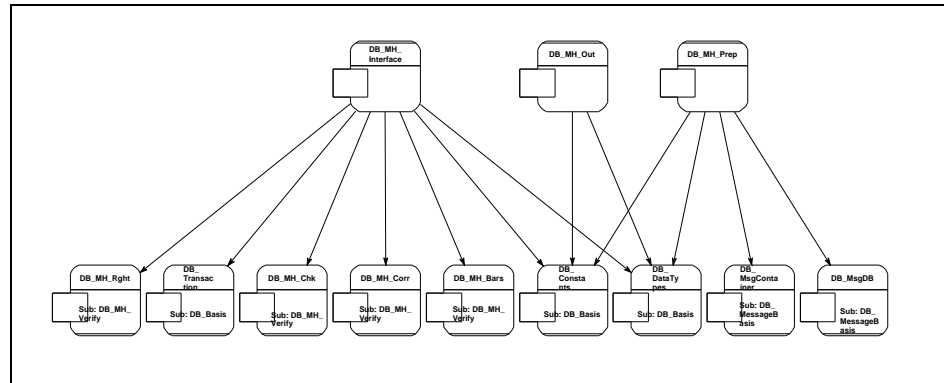
### 8.2.2.3 MD - ProMod<sup>PLUS</sup>

Figure 89 shows how subsystems in MD are used to order and arrange large applications in a hierarchical way.



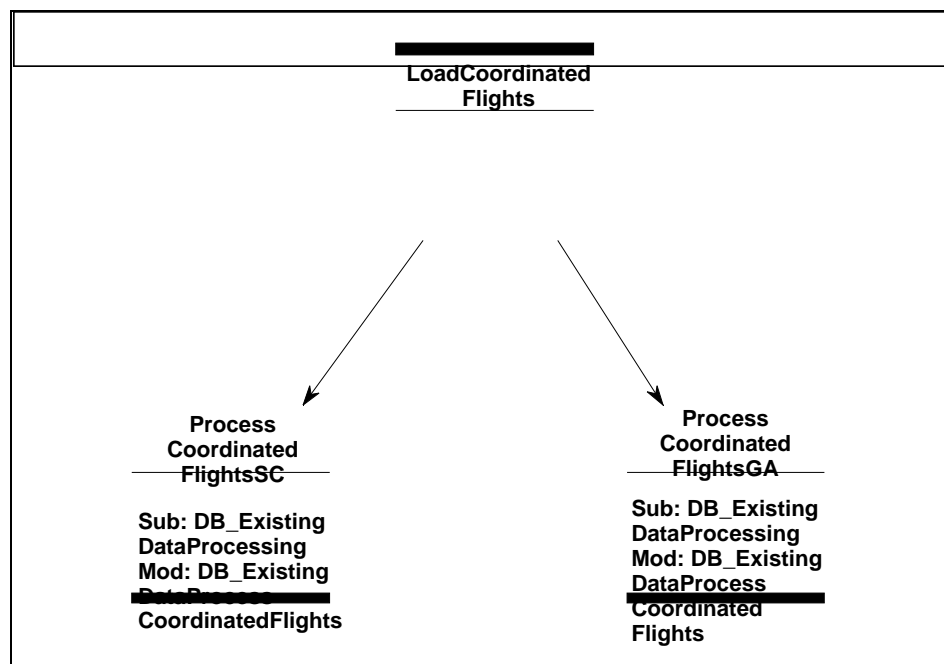
**Figure 89 Subsystem hierarchy in MD**

Figure 90 shows use relationships among Oracle packages.



**Figure 90 Use relationship among Oracle packages**

Figure 91 shows the stored procedures of one package and the call-relation between them.



**Figure 91 Call relation among stored procedures**

The textual presentation of the package is shown in the following box.

**PURPOSE**

This module provides the functions for loading the coordinated flights from the existing system into the database.

**ENDPURPOSE**

**EXPORT**

**FUNCTION**

```
LoadCoordinatedFlights;  
  
IMPORT  
FROM  
DB_ExistingDataProcessing::DB_ExistingDataProcessCoordinatedFlights:  
FUNCTION  
    ProcessCoordinatedFlightsGA,  
    ProcessCoordinatedFlightsSC;
```

The following box contains a description of the stored procedure:

**PURPOSE**

Load the existing coordinated flights data into a dummy table and process them. The procedure takes no arguments but prompts the user for file names.

Note: the import for the coordinated flights is separated for general aviation and s/c flights and for the different seasons.

Thus the procedure is called several times.

**ENDPURPOSE**

**DATA**

dataFile : String;

**BEGIN\_PC**

CREATE TABLE kfutdummy (

    saison    char(3),

    airport  char(3),

    a\_d      char(1),

    fltno    char(7),

    date1    char(5),

    date2    char(5),

    opsdays char(7),

    prevnext char(3),

    route    char(2),

    origdest char(3),

    time     char(4),

    stc      char(1),

    suffix   char(1),

    suffix\_r char(1),

    actype   char(3),

    fltno\_r  char(7),

    time\_r   char(4),

    route\_r  char(2),

    altairport char(3),

    insforce char(1),

    dev      char(2)

)

host sqlload userid=fluko/fluko control=LoadCoordinatedFlights.ctf  
data=&dataFile log=LoadCoordinatedFlights.log

```
* load the coordinated flights data into the dummy table, the values are
asked for upon execution *
* depending on the data which was loaded either SC or GA is called *

$DB_ExistingDataProcessing::DB_ExistingDataProcessCoordinatedFlight
s:ProcessCoordinatedFlightsSC
$DB_ExistingDataProcessing::DB_ExistingDataProcessCoordinatedFlight
s:ProcessCoordinatedFlightsGA

* the table kfutdummy must be dropped manually *

END_PC
```

#### 8.2.2.4 UML - Rational-Rose

Figure 92 shows an integrated modelling approach with UML. Database tables and views are modelled as classes. Stereotypes indicate that a class is representing a table or a view. If a view is using a table, this is indicated by a dependency relationship. In the same way database packages are modelled as classes with the stereotype <<package>>. The stored procedures are modelled as functions of the package. The dependencies between the packages are modelled by corresponding relationships. If a package is manipulating a table this is also indicated by a corresponding relationship.

In addition, the figure contains 4GL elements and their relations to the database elements.

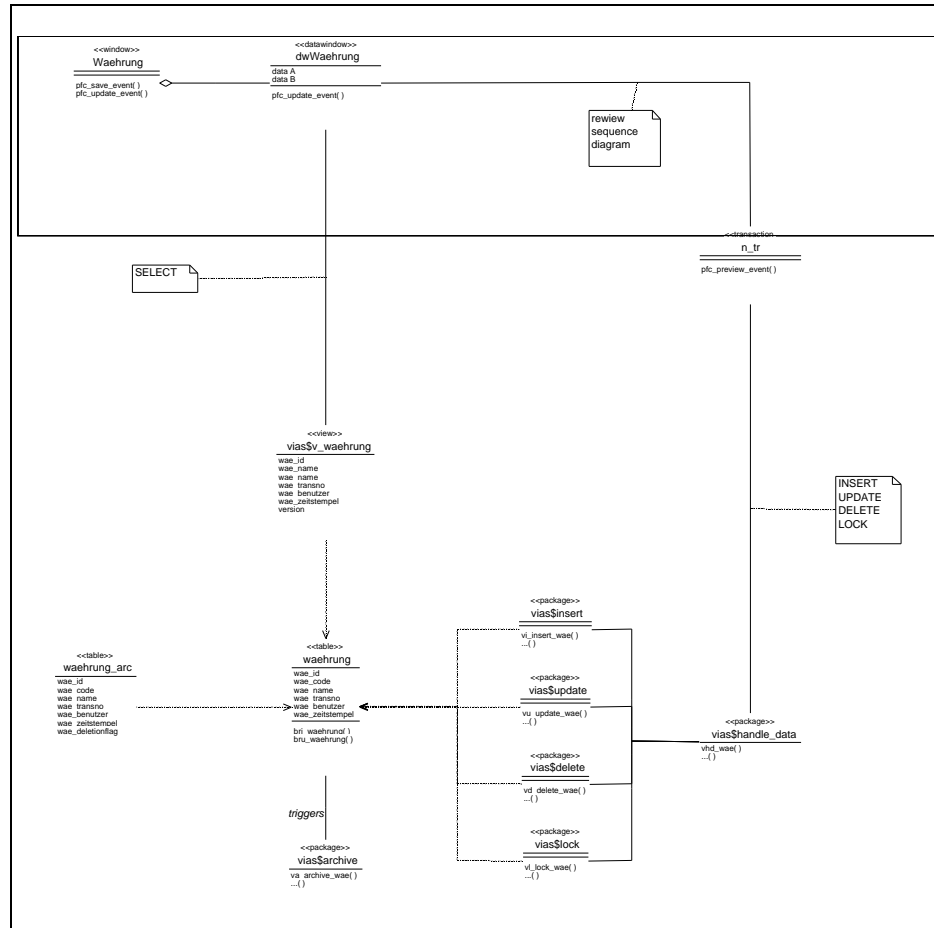


Figure 92 Integrated modelling approach with UML

### 8.2.3 Comparison and Summary

The following table summarises which database property can be modelled with which method or tool. The following shortcuts are used:

- M - Modelling (basic feature)
- I - Modelling (individual extension)
- C - Code-generating
- R - Reverse-engineering

Database Property	ERWin	IM	MD	UML
<b>Static aspects</b>				
Tables	M C R	M C	I	M (C)
Attributes	M C R	M C	I	M (C)
Views			I	M (C)
Constraints	M C R	M C		
Referential integrity	M C R	M C		
Cardinalities	M C R	M C		
Alternate keys	M C R			
Indices	M C R	M C		
Physical distribution Table spaces (Oracle) Segments (Sybase)	M C R			
Synonyms				
<b>Dynamic aspects</b>				
Trigger	M C R		M	M
Stored procedures			M	M
Packages (Oracle)			M	M
Global Variables			M	M
Constants			M	M
Key Generators (Oracle)				
Types			M	I
<b>Further aspects</b>				
Domains	M C R		M	M
Subsystems			M	M
Privileges and rights	M C R			
Distributed databases				

Database Property	ERWin	IM	MD	UML
Active servers				
Processes and tasks				

### Conclusions:

- The static aspects can be modelled well with ERWin (Bachmann) and ProMod-PLUS (IM). Both provide appropriate modelling and code generating features. In the case of ProMod-PLUS the Perl scripts corrects the current generating problems in a sufficient manner.
- If you need reverse-engineering features, this will be supported by ERWin but not by ProMod-PLUS.
- The dynamic aspects can be modelled with UML and MD. The basic mapping is that packages are described as classes (UML) or modules (MD). Stored procedures are described as operations (UML) or functions (MD).
- Both approaches for modelling the dynamic aspects provide modelling support but no code generating or reverse-engineering support.
- The function charts of MD can be used to describe the activations of stored procedures inside one package. There is nothing equivalent in UML. On the other hand, the sequence diagrams of UML provides good snapshots and overviews.
- One problem to be solved is the integration of the static and dynamic aspects. Several approaches to solve this problem are possible:
  - One approach can be to model parts of the static aspects as classes (UML) or modules (MD). Then you can describe the relations in your model. The problem is the consistency if you model the static parts two times. Here a good tool support will be helpful.
  - There is a bridge between ERWin and Rational-Rose.
  - If you use ProMod-PLUS IM for the static part and ProMod-PLUS IM for the dynamic part, you can use the traceability feature for mapping them. An entity can be linked to subsystems, modules and functions. A relationship can also be linked to subsystems, modules and functions.
- The overall architecture can be modelled well with packages (UML) or subsystems (MD).

### Further aspects:

- We have more experience with ProMod-PLUS but the future of the tool is quite open.
- UML has the potential for a future standard but the tool (Rational-Rose) has currently no good maturity level.



## 9 Appendix E: Tool Support

### Contents

#### 9.1 Tool support

### Summary

This chapter describes several tools that may be used to support the modelling techniques presented in the document.

## 9.1 Tool support

This section summarises the information about tools for context modelling which was given in Chapter 2. The tool information is summarised in the following table:

Tool		Diagramm					
Name	Vendor	UML	Operational Schemas	Data Flow Diagram	Entity Relationship Diagram	ERK Diagram	Hardware Architecture Diagram
Visio	Visio		X	X	X	X	X
Rational-Rose	Rational	X					
Select	Select Software Tools	X					
ProMod-PLUS	debis Systemhaus			X	X		
System Architect	System Architect			X			
Teamwork	Cheyenne Software			X			
ERWin	Logic Works				X		
S-Designer	Sybase				X		
netViz	Quyen Software						X
ClickNet	PinPoint						X
SysDraw	Microsystems Engineering						X

### 9.1.1 Operational schemas

Operational schemas can be drawn using standard presentation tools. If the tool supports making templates, e.g. like Visio from Visio Corp., this is the best.

There is no direct support in UML to model operational schemas. Sequence diagrams represent a sequence of steps by they only include IT system objects and are purely sequential. Activity diagrams can represent the non-sequential nature of flow in business processes, but they do not represent it according to actors. In addition, the nature of steps involved (e.g. manual or automated) cannot be represented in both diagrams. However, for UML supporting tools allowing it, the concepts of both UML diagrams can be combined to represent operational schemas.

### 9.1.2 Use case diagrams

There are few tools to support the creation and evolution of use-case diagrams. Visio may help us to understand the relationships. Templates

---

can also be created (using Visio) for supporting the use case modelling notations.

Nevertheless, use-cases and their relationships are best understood by using a graphical tool. The benefits of a CASE tool that incorporates use cases can be easily visualised because of its traceable functionality. Rational Rose from Rational Software Corporation (<http://www.rational.com>) includes capabilities for use case modelling. Another CASE tool supporting use cases is the SELECT Enterprise toolset from Select Software Tools (<http://www.microway.com.au/select/>).

### **9.1.3 Data flow diagrams**

Several good tools support the creation of data flow diagrams. An excellent tool for creating data flow diagrams following SA is ProMod<sup>PLUS</sup>. Others are System Architect or Teamwork. Data flow diagrams with free graphics can be drawn with PowerPoint or Visio.

### **9.1.4 Entity relationships diagrams**

Several good tools support the creation of extended entity relationship diagrams. An excellent tool is ProMod<sup>PLUS</sup> that provides a good abstraction level for context modelling. Tools like ERWin and S-Designor do not use such a high abstraction level and operates nearer to the physical database design.

### **9.1.5 Block diagrams**

Standard tools such as Visio from Visio Corp is perfectly suitable for making block diagrams. The notation is simple, so any drawing tool may be used.

Rational Rose supports the notion of “blocks” or grouping of components into packages. One may package identified components or classes into more abstract packages to illustrate and model the architecture of the system.

### **9.1.6 Hardware/network diagrams**

Usually, hardware and network information is represented using standard presentation tools like Microsoft PowerPoint with libraries of icons representing hardware elements. Other tools used are network diagramming tools, specialised for this task, like netViz from Quyen Software, ClickNet from PinPoint Software, Visio tools from Visio Corp. And SysDraw from Microsystems Engineering. This latter category of tools usually allows easy, hypertext-like, navigation within the diagrams and elements composing the whole system.

Network discovery tools can also provide a good input for documenting hardware and network architecture. These tools can discover devices connected to a network and build automatically diagrams which can be imported in other tools to be manually extended.



---

## 10 Appendix E: References

- Ambler, S.: "The Object Primer - The Application Developer's Guide to Object-Orientation", 1995, SIGS Books, ISBN 1-884842-17-8
- Batini et. al: "Conceptual Database Design: An Entity-Relationship Approach", Benjamin/Cummings, 1991, ISBN 0-805302-44-1
- Chen, P.: "The Entity-Relationship Model - Toward a Unified View of Data", ACM Transactions on Database Systems, Vol.1, No.1, March 1976
- DeMarco, T.: "Structured Analysis and System Specification", Yourdan Press 1978, ISBN 0-917072-07-3
- Flavin, M.: "Fundamental Concepts of Information Modelling", Yourdon Press, 1981, ISBN 0-917072-22-7
- Fleming F., Halle, B.: "Handbook of Relational Database Design", Addison-Wesley, 1989, ISBN 0-201-11434-8
- Hawryszkiewicz, I.: "Introduction to System Analysis and Design", Prentice Hall, 1988, 0-13-484585-4
- Jacobson, I.: "Object-Oriented Software Engineering - A Use Case Driven Approach", Addison-Wesley, 1992, ISBN 0-201-54435-0
- Krogstie, J. and Sølvsberg A.: "Information Systems Engineering: Conceptual Modelling in a Quality Perspective", compendium in course 45161 at NTNU, Trondheim, February 1997.
- McMenamin, S./Palmer, J.: "Essential System Analysis", Yourdan Press, 1984, ISBN 0-13-287913-1
- Navathe and Elmasri: "Fundamentals of Database Systems", Addison-Wesley, 1994, ISBN 0-805417-48-1
- Ovum Ltd.: "4GLs and Client-Server Tools", London 1994

Robertson, J./Robertson, S.: "Complete System Analysis", Dorset House, 1994, ISBN 0-932633-25-0

Tilley, S., Muller, H., Whitney, M., Wong, K.: "Domain-Retargetable Reverse Engineering", in Proc. Int'l Conference on Software Maintenance, Montreal, Quebec, Canada, September 1993

UML v1.0: "Unified Modelling Language", Version 1.0, Set of documents submitted for standardization, Rational Software Corp. Available on URL: "<http://www.rational.com>", 13 January, 1997.

UML v1.1: "Object Management Group: UML V1.1", September 1997.