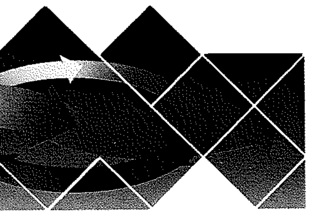
Software Engineering/Object Oriented Methods



Object-Oriented Software Engineering

A Use Case Driven Approach

Ivar Jacobson

Magnus Christerson Patrik Jonsson Gunnar Övergaard

How can software developers, programmers and managers meet the challenges of the 90s and begin to resolve the software crisis?

This book is based on Objectory which is the first commercially available comprehensive object-oriented process for developing large-scale industrial systems. Ivar Jacobson developed Objectory as a result of 20 years of experience building real software-based products. The approach takes a global view of system development and focuses on minimizing the system's life cycle cost. Objectory is an extensible *industrial process* that provides a *method* for building *large industrial systems.*

'In this book Jacobson establishes a new direction for the future of Software engineering practice. It is a thorough presentation of ideas and techniques that are both solidly proven and simultaneously at the leading edge of software engineering methodology.'

Larry L. Constantine
RODP, Organization & System Consultant

'Jacobson is in my opinion one of the foremost methodologists in the field of Software Engineering . . . I strongly recommend . . . this book . . . not only for software managers and designers but for anyone who wishes to understand how the next generation of Software Systems should be built.'

Dave Thomas
Object Technology International

Dr Ivar Jacobson has over 20 years experience in the design of large scale real-time systems in the telecommunications industry. He developed an early object-based design technique, a major portion of which has evolved into the international standard CCITT/SDL. Ivar Jacobson has extensive industrial teaching experience and has served on the OOPSLA, ECOOP and TOOLS program committees. He is founder and Vice President of Technology at Objective Systems in Sweden, which develops and markets the object-oriented method Objectory.

Magnus Christerson, Patrik Jonsson and Gunnar Övergaard are actively involved in the development and use of Objectory at Objective Systems.

ISBN 0-201-54435-0

90000>

9 780201 544350

Addison-Wesley Publishing Company

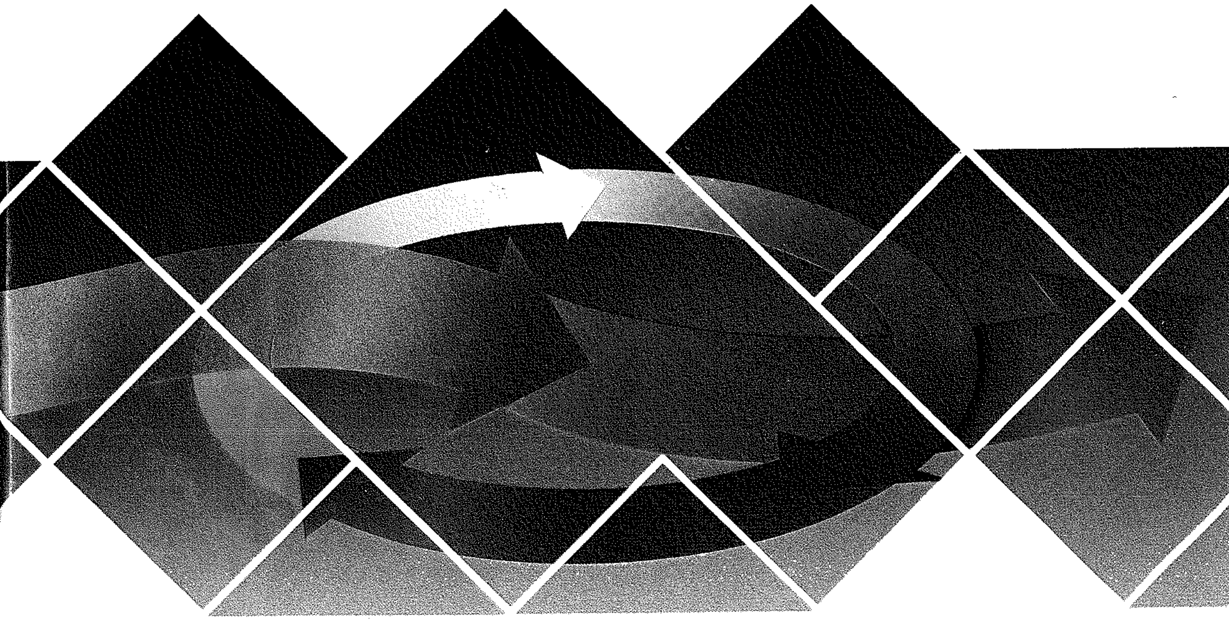
acm PRESS

A Use Case Driven Approach

Magnus Christerson Patrik Jonsson
Gunnar Övergaard

Object-Oriented Software Engineering

A Use Case Driven Approach



acm PRESS

ADDISON-WESLEY

ACM PRESS

Editor-in-Chief	**Peter Wegner**	Brown University
International Editor (Europe)	**Dines Bjørner**	Technical University of Denmark

SELECTED TITLES

Object-Oriented Reuse, Concurrency and Distribution *Colin Atkinson*

Advances in Database Programming Languages *François Bancilhon and Peter Buneman (Eds)*

Software Reusability (Volume 1: Concepts and Models) *Ted Biggerstaff and Alan Perlis (Eds)*

Software Reusability (Volume 2: Applications and Experience) *Ted Biggerstaff and Alan Perlis (Eds)*

Object-Oriented Concepts, Databases and Applications *Won Kim and Frederick H. Lochovsky (Eds)*

Distributed Systems *Sape Mullender (Ed)*

The Oberon System: User Guide and Programmer's Manual *Martin Reiser*

Programming in Oberon: Steps Beyond Pascal and Modula *Martin Reiser and Niklaus Wirth*

The Programmer's Apprentice *Charles Rich and Richard C. Waters*

Instrumentation for Future Parallel Computer Systems *Margaret Simmons, Ingrid Bucher and Rebecca Koskela (Eds)*

User Interface Design *Harold Thimbleby*

Project Oberon: The Design of an Operating System and Compiler *Niklaus Wirth and Jurg Gutknecht*

Object-Oriented Software Engineering

A Use Case Driven Approach

Ivar Jacobson
Magnus Christerson Patrik Jonsson
Gunnar Övergaard

Objective Systems

ADDISON-WESLEY
PUBLISHING
COMPANY

Wokingham, England · Reading, Massachusetts · Menlo Park, California · New York
Don Mills, Ontario · Amsterdam · Bonn · Sydney · Singapore
Tokyo · Madrid · San Juan · Milan · Paris · Mexico City · Seoul · Taipei

Copyright © 1992 by the ACM Press, A Division of the Association for Computing Machinery, Inc (ACM)

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

The programs in this book have been included for their instructional value They have been tested with care but are not guaranteed for any particular purpose The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks The publisher has made every attempt to supply trademark information about manufacturers and their products mentioned in this book. A list of the trademark designations and their owners appears below

Cover designed by Chris Eley
and printed by The Riverside Printing Co (Reading) Ltd
Printed in the United States of America
ISBN 0-201-54435-0

First printed 1992

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library.

Library of Congress Cataloging in Publication Data available

Trademark notice
Ada™ is a trademark of the Ada Joint Program Office, DoD, US Government
CommonView™ is a trademark of Glockenspiel Ltd
Eiffel™ is a trademark of Interactive Software Engineering
HOOD™ is a trademark of HOOD working group
InterView™ is a trademark of Glockenspiel Ltd
KEE™ is a trademark of Intellicorp, Inc
LOOPS™, SMALLTALK™, SMALLTALK-78™ and SMALLTALK-80™ are trademarks of Xerox Corporation
MacApp™, Macintosh™ and Macos™ are trademarks of Apple Computer Inc.
MS-DOS™ and Windows™ are trademarks of Microsoft Corporation
NeWS™ is a trademark of Sun Microsystems Inc.
NewWave™ is trademark of Hewlett-Packard
NeXTStep™is a trademark of NeXT Corp
Objective-C™ is a trademark of Stepstone Corp
Objectory™ is a trademark of Objective Systems SF AB
PROLOG™ is a trademark of Expert Systems International
TMS™ is a trademark of Texas Instruments Inc
UNIX™ is a trademark of AT&T Corporation
X-Windows™ is a trademark of MIT

Foreword

Ivar Jacobson is in my opinion one of the foremost methodologists in the field of software engineering. I take great pleasure in writing this, because he is also a close personal friend. He brings a refreshingly pragmatic point of view to a discipline that often seems be so abstract and arcane as to be hopelessly remote from the real world of 'blue collar' programmers. His methodology is based on some really innovative ideas about modeling the software process, presented within a tried and proven engineering framework. It brings to the task of analyzing, designing and constructing complex software intensive products the same disciplined approach that is to be found in other branches of engineering.

Along with many others I have urged Ivar for some time to publish his methodology in a textbook, so that it would be accessible to a larger audience. I believe that the concepts in Objectory, the first comprehensive object-oriented process for developing large scale industrial systems, are important and should get wider exposure. This book represents over 20 years of experience building real software based products and a great deal of serious thinking about how such systems should be built. If you have any interest at all in software you will enjoy reading it.

Objectory stands out as being a truly object-oriented methodology, in which both the process and the methodology are themselves represented as objects. While some may find this idea of a reflective or 'meta' architecture to be rather exotic, it is in fact intensely practical and absolutely essential. It makes Objectory an extensible methodology which can be specialized to both the organization and the application domains. Simply put, Objectory provides a software process for building not just software, but also other more specialized software processes.

Another key innovation in Objectory is the concept of use cases, which has now been proved effective in a number of real-world projects. Use cases provide the needed linkage between requirements, development, testing and final customer acceptance This idea, which originated in Ivar's work on the AXE switch, has been generalized so that it can be applied in application domains as diverse as command and control and business information systems

Use cases provide a concrete representation of software requirements, which allows them to be both formally expressed and systematically tested. Changes in requirements map directly onto changes in the set of use cases. In this way Objectory provides a solid methodological foundation for rapid prototyping and other forms of incremental software development. Objectory enables managers to move beyond labour intensive hand assembly of software systems, and allows them to transform their organizations into highly automated factories to manufacture software from reusable components

Many feel that we are in the midst of a software crisis, and I agree. High-quality software has become one of the most sought after commodities in the modern world We just can't seem to get enough of it, on time and on budget, to meet the demand. This book will help you overcome the software crisis in your own organization, by showing you how to make software construction into a reliable and predictable engineering activity

One of the more profound insights offered by modern software engineering is that change is inevitable, and that software must be designed to be flexible and adaptable in the face of changing requirements. Objectory, with its reflective architecture, goes one step further, and provides an extensible methodology which can itself adapt to shifts in the business climate or the demands of new technologies No static text can ever capture all the nuances of such a dynamic software entity but this one comes very close I strongly recommend it, not only for software managers and designers, but for anyone who wishes to understand how the next generation of software systems should be built

Dave Thomas

Foreword

Ivar Jacobson has taken the time to create a book that is certain to become essential reading for software developers and their managers In this book, Jacobson establishes a new direction for the future of software engineering practice It is a thoughtful and thorough presentation of ideas and techniques that are both solidly proven and, simultaneously, at the leading edge of software engineering methodology. Jacobson is simply a thinker who has been ahead of his time in creating usable methods for building better, more reliable and more reusable large software systems.

Despite the title, this is not 'another book on object-oriented analysis and design,' nor yet another standard reworking on the word-of-the-week. Once, of course, the word-of-the-week in software engineering was 'modular,' later it was 'structured,' and now, as every programmer or software engineer who reads or attends conferences knows, it is 'object-oriented.'

When the word-of-the-week was still 'structured,' and I wrote the first edition of *Structured Design*, the very idea of systematic methods for software development was radical. Software engineering was in its infancy, and when I introduced data flow diagrams and structure charts, few recognized either the need for notation or the benefits of well-conceived modeling tools for analysis and design.

But things have changed Now, new methodologies are created over cocktails, and books spin out of word-processors so fast that revised or 'corrected' editions appear almost before the original has reached the bookstores. Since nearly everyone now recognizes that a methodology must be supported by a notation, notations proliferate. A new object-oriented design notation can be churned out over a weekend so long as the major objective is simply squiggles and icons with a unique 'look and feel,' and issues of usability and power in modeling are considered unimportant.

And here we have yet another notation supporting one more methodology? Not quite.

It is true that the serious reader will have to surmount both new terminology and new notation to get to the marrow, but this book is different. It was not conceived and written overnight. The methodology it describes has been in use for years to design and build numerous software systems, and its notation has evolved slowly from both manual and CASE-supported application. It is not the work of a writer or consultant with a long booklist, but comes from a practicing software engineer and leader in software engineering who has been doing large-scale object-oriented development for longer than most people even knew that objects existed. Throughout this period, the ideas and methods have been honed by the grindstone of building software and refined by thoughtful reflection and analysis.

What we have here is an approach to object-oriented analysis and design that is fundamentally different from most of the highly touted and more visible methods that clutter the landscape. I believe it is an approach of proven power and even greater promise.

The real power of this approach rests not only in the wealth of experience on which it is based but also in the way in which it starts from a different point of departure and builds an entirely different perspective on how to organize software into objects. Jacobson does not build naive object models derived from simplistic reinterpretations of data modeling and entity object relationship models. He starts from an entirely different premise and set of assumptions uniquely tailored to creating robust, sophisticated object structures that stand the test of time

His approach centers on an analysis of the ways in which a system is actually used, on the sequences of interactions that comprise the operational realities of the software being engineered. Although it fully incorporates the conceptual constructs, the application and enterprise entities that undergird our thinking about software systems, it does not force the entire design into this rigid pattern. The result is a more robust model of an application, leading to software that is fundamentally more pliant, more accommodating to extensions and alterations and to collections of component parts that are, by design, more reusable.

At the heart of this method is a brilliantly simple notion: the use case. A use case, as the reader will learn, is a particular form or pattern or exemplar of usage, a scenario that begins with some user of the system initiating some transaction or sequence of interrelated events. By organizing the analysis and design models around user interaction and actual usage scenarios, the methodology produces systems that are intrinsically more useable and more adaptable to changing usage. Equally important, this approach analyzes each use case into its constituent parts and allocates these systematically to software objects in such a way that external behavior and internal structure and dynamics are kept apart, such that each may be altered or extended independently of the other This approach recognizes not one kind of object, but three, which separate interface behavior from underlying entity objects and keep these independent of the control and coordination of usage scenarios.

Using this approach, it is possible to construct very large and complex designs through a series of small and largely independent analyses of distinct use cases. The overall structure of the problem and its solution emerges, step-by-step and piece-by-piece, from this localized analysis In principle – and in practice – this methodology is one whose power increases rather than diminishes with the size of the system being developed.

Use case driven analysis and design is a genuine breakthrough, but it is also well-grounded in established fundamentals and connected to proven ideas and traditions in software engineering in general and object-oriented development in particular. It echoes and extends the popular model–view–controller paradigm of object-oriented programming. It is clearly kin to the event-driven analysis and design approaches of Page-Jones and Weiss, as well as to the widely practiced event-partitioning methods pioneered by McMenamin and Palmer.

On this ground, Ivar Jacobson has built a work that is nothing short of revolutionary. Rich with specific guidelines and accessible examples, with completely detailed case studies based on real-world projects, this book will give developers of object-oriented software material that they can put into practice immediately. It will also challenge the reader and, I am confident, enrich the practice of our profession for years to come

Larry L Constantine

Preface

This is a book on industrial system development using object-oriented techniques. It is not a book on object-oriented programming. We are convinced that the big benefits of object-orientation can be gained only in the consistent use of object-orientation throughout all steps in the development process. Therefore the emphasis is placed on the other parts of development such as analysis, design and testing.

You will benefit from this book if you are a system developer seeking ways to improve in your profession. If you are a student with no previous experience in development methods, you will learn a robust framework which you can fill with details as you take part in future development projects. Since the focus of the text is on development, the book will be convenient to use in combination with other texts on object-oriented programming. Many examples illustrate the practical application of analysis and design techniques.

From this book you will get a thorough understanding of how to use object-orientation as the basic technique throughout the development process. You will learn the benefits of seamless integration between the different development steps and how the basic object-oriented characteristics class, inheritance and encapsulation are used in analysis, construction and testing. With this knowledge you are in a much better position to evaluate and select the way to develop your next data processing system.

Even though object-orientation is the main theme of this book, it is not a panacea for successful system development. The change from craftsmanship to industrialism does not come with the change to a new technique. The change must come on a more fundamental level which also includes the organization of the complete development process. Objectory is one example of how this can be done.

This book does *not* present Objectory. What we present is the fundamental ideas of Objectory and a simplified version of it. In this book we call this simplified method OOSE to distinguish it from Objectory. To use the process in production you will need the complete and detailed process description which, excluding large examples, amounts to more than 1200 pages. To introduce the process in an organization also needs careful planning and dedication.

It is our hope that we have reached our goal with this book, namely to present a coherent picture of how to use object-orientation in system development so as to make it accessible to both practitioners in the field and students with no previous knowledge of system development. This has been done within a framework where system development is treated as an industrial activity and consequently must obey the same requirements as industry in general The intention is to encourage more widespread use of object-oriented techniques and to inspire more work on improving the ideas expounded here. We are convinced that using these techniques will lead to better systems and a more industrial approach to system development

Part I: Introduction. The book is divided into three parts The first part is a background part and covers the following chapters:

(1) System development as an industrial process,
(2) The system life cycle,
(3) What is object-orientation?,
(4) Object-oriented system development,
(5) Object-oriented programming.

This part gives an introduction to system development and summarizes the requirements on an industrial process. It also discusses the system life cycle. The idea of object-orientation is introduced and how it can be used in system development and during programming is surveyed.

Part II: Concepts. The second part is the core of the book. It covers the following chapters:

(6) Architecture,
(7) Analysis,
(8) Construction,
(9) Real-time specialization,
(10) Database specialization,
(11) Components,
(12) Testing.

The first chapter in this part introduces the fundamental concepts of OOSE and explains the reason why these concepts are chosen. The following chapter discusses the method of analysis and construction. Two chapters discusses how the method may be adapted to real-time systems and database management systems. The components chapter discusses what components are and how they can be used in the development process. Testing activities are discussed in a chapter of their own.

Part III: Applications. The third and last part covers applications of OOSE and how the introduction of a new development process may be organized and managed. This part ends with an overview of other object-oriented methods. This part thus looks as follows:

(13) Case study: warehouse management system,
(14) Case study: Telecom,
(15) Managing object oriented software engineering,
(16) Other object-oriented methods.

Appendices. Finally we have two appendices. Appendix A comments on our development of Objectory and Appendix B summarizes the notation used in the book.

So, how should you read this book? Of course, to get a complete overview, the whole book should be read, including the appendices. But if you want to read only selected chapters the reading cases below could be used.

If you are an experienced object-oriented software engineer, you should be familiar with the basics. You could read the book as suggested in Figure P.1.

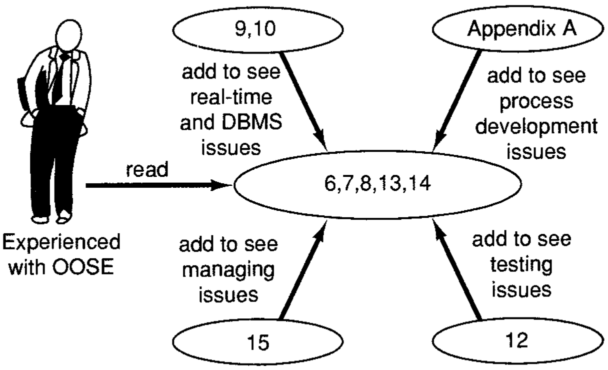

9,10
Appendix A
add to see real-time and DBMS issues
add to see process development issues
read
6,7,8,13,14
Experienced with OOSE
add to see managing issues
add to see testing issues
15
12

Figure P.1

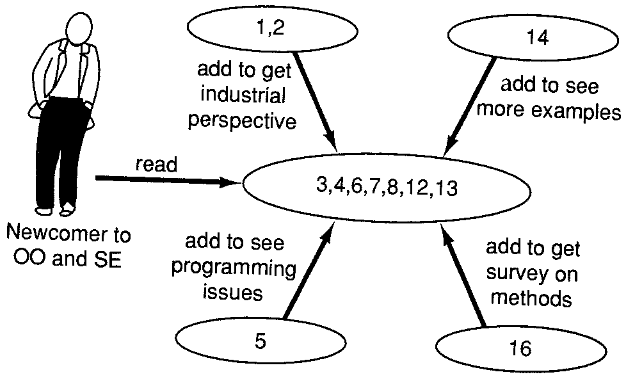

1,2
14
add to get industrial perspective
add to see more examples
read
3,4,6,7,8,12,13
Newcomer to OO and SE
add to see programming issues
add to get survey on methods
5
16

Figure P.2

If you are a newcomer to object-orientation and software engineering you could read the book as in Figure P.2.

If you are an experienced software engineer you could read the book as in Figure P.3.

If you are a manager you could read the book as proposed in Figure P.4.

Hence, although the book is not object-oriented, it is written in a modularized way and can be configured in several different ways. To build systems in this way is the theme of the book, and the technique and notation used above is very similar to the technique used also in this book.

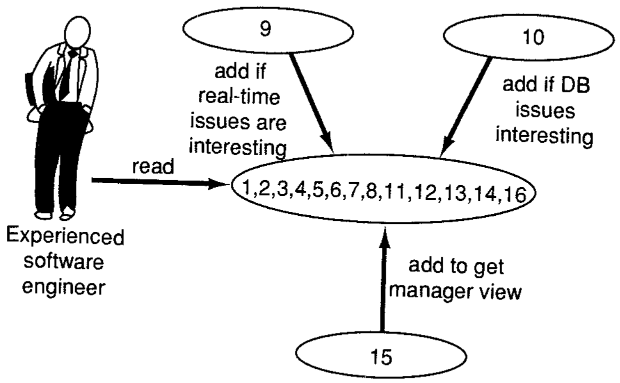

9
10
add if real-time issues are interesting
add if DB issues interesting
read
1,2,3,4,5,6,7,8,11,12,13,14,16
Experienced software engineer
add to get manager view
15

Figure P.3

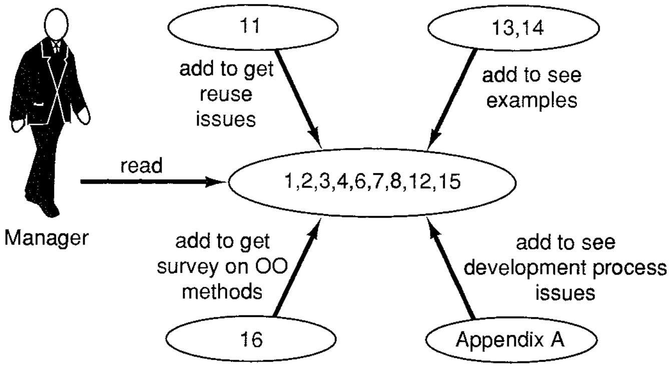


Figure P.4

A short history and acknowledgements

The work presented in this book was initiated in 1967 when I proposed a set of new modeling concepts (notation with associated semantics) for the development of large telecommunication switching systems. The main concepts were signals and blocks. A real-time system is an open system communicating with its environment by signals only. A signal models the physical stimulus/response communication which a concrete system has when interacting with the outside world. Given a signal as input, a system performs internal actions such as executing algorithms, accessing internal information, storing results and sending output signal to the environment. This view presents the system in a very abstract way – as a black box. A less abstract view on a lower level models the system as a set of interconnected blocks. Blocks are modules which can be implemented in hardware or software or any combination of both. A block communicates with its environment only through signals. Signals between two blocks are internal, whereas signals modeling physical communication, that is signals between a block and the environment of the system, are external. Internal signals are messengers conveying data from one block to another within the same system. All entries of a block were labelled and constituted the signal interface of that block, to be specified in a separate interface document. Hence the system can now be viewed as a set of interconnected blocks jointly offering the functions of the system. Each block has a program which it obeys on reception of an input signal, performing internal action, that is executing algorithms, storing and accessing block internal information, and sending internal and external signals to the environment

The proposal can be summarized as an attempt to unify long experience from systems design with the possibilities offered by dramatically new computer technology. Since the two technologies were so different, this was not a self-evident method, neither within Ericsson nor within computer science. There was a rather strong belief that the two represented unrelated technological universes: the new one was so different that it would be meaningless and only a burden to make any attempt to learn from the old one. However, the two techniques were joined and a set of modeling concepts evolved.

The modeling constructs were soon followed by a skeleton of a new design method, the use of which was first demonstrated in the development of the AKE system put in service in Rotterdam in 1971, and more completely demonstrated by the AKE system put in service in Fredhäll, Sweden, in 1974. Naturally this experience has guided subsequent work on the development of the successor to AKE, the AXE system, which is now in use in more than 80 countries worldwide. The modeling constructs were very important and for the AXE system a new programming language and a new computer system were developed in accordance with these early ideas.

Although it is a neighbouring country, the early development of object-oriented programming and Simula in the 1960s in Norway was made independently and in parallel with our work. It was not until 1979 that we 'discovered' object-oriented programming and then it was in terms of Smalltalk. Although object-oriented ideas have influenced our recent work, basically two separate problems are being solved: large-scaleness and small-scaleness.

The modeling constructs introduced during the 1960s were further formalized in research taking place between 1978 and 1985. This research resulted in a formally described language which offered support for object-orientation with two types of objects and two types of communication mechanisms, send/wait and send/no-wait semantics. The language supported concurrency with atomic transactions and with a special semantical construction for the handling of a course of events similar to the use case construct presented later. This work, reported in a PhD thesis in 1985, resulted in a number of new language constructs, initially developed from experience, being refined and formalized. This was a sound basis from which to continue, and take a new approach, the development of the method. The principles of Objectory were developed in 1985–7. Then I further refined and simplified the ideas, generalized the technique from telecom applications, extended it with the inheritance concept and other important constructs like extensions, and joined it with an analysis technique and also to object-oriented programming.

Today these concepts have been further refined. The Objectory process, of which this book describes some fundamental ideas, is the result of work by many individuals, of whom most today work at Objective Systems SF AB, Sweden. Gunnar Övergaard and Patrik Jonsson did much of the writing of the first process description of Objectory analysis and design respectively. Magnus Christerson did much to condense and rewrite the material into the form of this book. They have all contributed to Objectory; especially in the formalization of the Concepts. Magnus has also related the ideas of Objectory to other areas as presented in this book Fredrik Lindström has also been involved in the condensation of the material for this book and in the book project as such. Agneta Jacobson, Bud Lawson, and Lars Wiktorin have prepared material for some of the chapters in this book.

Mårten Gustafsson has substantially contributed to the analysis part of Objectory. Valuable contributions to Objectory have also been made by the following people: Sten-Erik Bergner, Per Björk, Ann Carlbrand, Håkan Dyrhage, Christian Ehrenborg, Agneta Jacobson, Sten Jacobson, Fredrik Lindström, Lars Lindroos, Benny Odenteg, Karin Palmkvist, Janne Pettersson, Birgitta Spiridon, Per Sundquist, Lars Wetterborg and Lars Wiktorin. The following users of Objectory have also contributed by feeding back experiences and ideas to enable improvements: Staffan Ehnebom, Per Hedfors, Jörgen Hellberg, Per Kilgren, Håkan Lidström, Christian Meck, Christer Nilsson, Rune Nilsson, Göran Schefte, Fredrik Strömberg, Karin Villers, Stefan Wallin and Charlotte Wranne. The following persons have done a lot to support the technology described in this book: Kjell S. Andersson, Hans Brandtberg, Ingemar Carlsson, Håkan Dahl, Gunnar M. Eriksson, Björn Gullbrand, Lars Hallmarken, Bo Hedfors, Barbara Hedlund, Håkan Jansson, Christer Johansson, Ingemar Johnsson, Kurt Katzeff, Rolf Leidhammar, Jorma Mobrin, Jan-Erik Nordin, Anders Rockström, Kjell Sörme, Göran Sundelöf, Per-Olof Thysselius, Ctirad Vrana and Erik Örnulf. The following people have given me strong personal inspiration and support: Dines Bjørner, Tore Bingefors, Larry Constantine, Göran Hemdal, Tom Love, Lars-Olof Norén, Dave Thomas and Lars-Erik Thorelli. In Sweden we normally do not thank family and friends in these circumstances, but no one believes that these types of result can be achieved without an exceptional support from them. We are also grateful to the support we have been given from STU (Swedish National Board on Industrial development now reorganized to NUTEK) through the IT-4 program which has been part of the financial support and sponsorship for the writing of this book.

Ivar Jacobson
Nybrogatan 45c, 11439 Stockholm, Sweden

Contents

Part I
Introduction

1

1 System development as an industrial process

1.1 Introduction

The development of software systems, being a relatively young industry, has not reached the level of maturity typically found in more traditional branches of industry. Consequently, products that are developed based on the usage of software technology often suffer from a lack of the established practices that are required for their development and exploitation as commercial products. This lack of experience can be traced to the fact that the emphasis in software development, thus far, has been placed upon the creative processes and methods used in the initial development of computer based systems. This emphasis is found in nearly all of the existing software engineering methods and related tools that have been developed to assist in the realization of software systems. (When we use the word system we not only include software systems, but also systems where hardware and software are integrated parts.)

In this book, we shall consider a method which provides, as do many other methods, support for the creative design of software products, but which also provides an approach for making software development a rational industrial process. Thus the aim is not just to produce well engineered software systems, but to make them viable 'living' products that can be exploited in an industrial environment.

How do we go about providing the software industry with the methods that enable us to deal with the practicalities of the more global view of software products? Largely, we must understand what goes into making 'industrial processes' successful and then apply this knowledge in an appropriate manner to the software industry. In order to gain the necessary insight, we use an analogy to the industrial processes of another well established industry. The analogy is quite useful since it illuminates the scope of the general problem, thus allowing us to understand (draw parallels with) several important

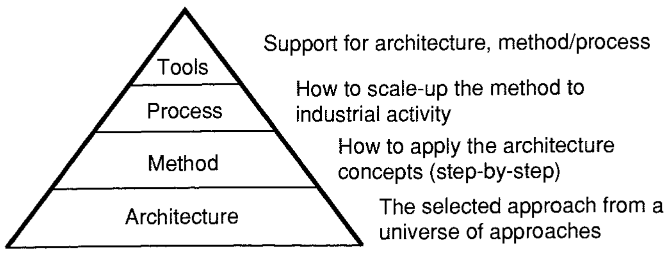

Tools
Process
Method
Architecture
Support for architecture, method/process
How to scale-up the method to industrial activity
How to apply the architecture concepts (step-by-step)
The selected approach from a universe of approaches

Figure 1.1 The constituents of a rational enterprise philosophy.

aspects of object-oriented software engineering (OOSE) that contribute to making software development an industrial process.

1.2 A useful analogy

The building construction industry is one of the most mature branches of industry, having taken advantage of developments since the beginnings of civilized life. Naturally, the advances in this industry have accelerated during the 20th century, which can be regarded as a period in which building construction changed largely from a craft to refined industrial processes. Since we all use buildings and are accustomed to their properties, the selection of building construction as an analogy provides a useful common denominator. By briefly examining some, but certainly not all, of the general properties of this industry, we shall be able to understand the need for, as well as the lack of, related properties in the software industry.

We can note that drawing analogies between the construction industry and computer related processes is not new. Spector and Gifford (1986), via an interview with an experienced bridge designer (Gerald Fox), have documented important analogies between the industrial processes of bridge construction and corresponding computer related processes.

In order to provide rationality in all phases of building construction, it is essential that a well established philosophy (i.e. point of view) guides the work of all parties in the various activities of a building project. The philosophy has its concrete realization in the form of an 'architecture' and related activities of the enterprise which establish the 'way of doing business' as portrayed in Figure 1.1.

By **architecture** of the construction approach, we mean a foundation of concepts and techniques, selected from a universe of potential foundations, that defines the characteristic structure of all buildings designed using the approach. As an example, we could consider the selection of an approach based upon the exploitation of building blocks and components This approach can be compared with other alternatives, for example, an architecture based solely upon customized constructs.

The **method** makes explicit the step-by-step procedures to be followed in applying the architecture to projects.

The **process** provides for the scaling-up of the method, so that it can be applied on a large scale with many interacting activities and parties.

Tools are provided to support all aspects of the enterprise and explicitly the activities of architecture, method and process.

The difference between method and process will be discussed further in the next chapter, but let us just illustrate some properties here. The method is more basic and is described as a project is described with its different activities. A project is ended when the last activity is performed and the building has been taken into operation. Operation and maintenance are often described in a simplified way as a single phase. This is not fully correct since maintenance often involves new projects, including all phases. A process, on the other hand, lasts as long as the product lasts and describes how the different activities interact during the whole life of the product.

Note that it is important not to confuse the term architecture underlying the method with the architecture of a particular product (i.e. 'building') that may be realized by applying the architecture. These building architectures represent 'instances' employing the enterprise philosophy as portrayed in Figure 1.2. Hence one architecture may be used for the building of various houses, and also various architectures may be used for the building of a specific house.

The architecture may be based solely upon utilizing building blocks and components; or solely upon utilizing customized constructs by craftsmen; or any combination thereof Here building blocks are for instance prefabricated sections of a house, and components are off-the-shelf products like windows, doors and bath tubs. Further, the approach may be based upon specific materials, for example, the usage of bricks or reinforced concrete. For each possible approach, a variety of methods could be defined to describe how to work with these constructs. This leads to the definition of a variety of step-by-step procedures, namely methods, where for example appropriate combinations of building blocks and components are utilized Furthermore, the method must be scaled and related also to other activities leading to various processes possible for each method defined. These activities could then be supported by various tools

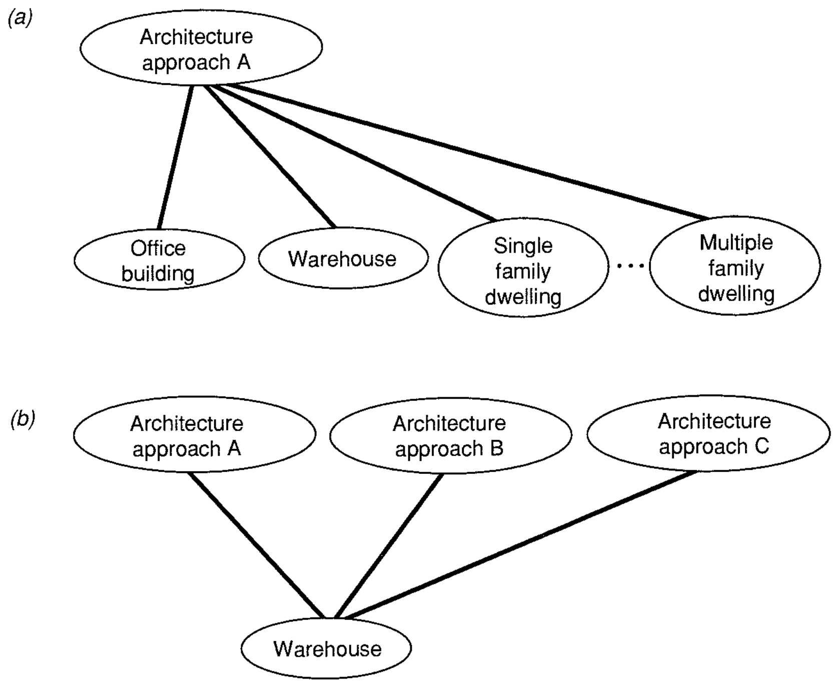


Figure 1.2 In (*a*) one architecture is used for different building products. In (*b*) we can choose from different architectures for a specific product

We now consider how various activities of building construction are supported. In fact, the model we have introduced earlier is applied during each activity of building construction as illustrated in Figure 1.3. The activities are creative design, construction and long term support. Of course, to make the transition smoothly and seamlessly between the phases, well defined interfaces are needed.

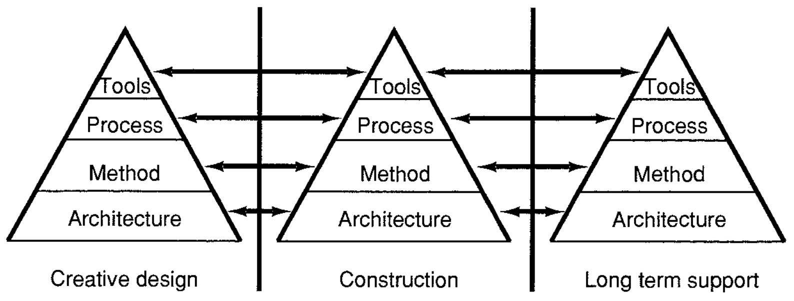


Figure 1.3 The constituents of multiple activities of a rational enterprise.

For each activity, there is a philosophy (point of view and related concepts) from which a particular architecture, method, process and tools of the phase are derived Further, to be successful as a whole, there must exist well defined two-way (seamless) interfaces between these phases. The details of the latter activities are directly related to factors in the preceding activities and wherever possible traceability should be applied, enabling us to return to relevant factors in previous activities when problems arise. We will now look at each activity in more detail.

1.2.1 Creative design

The transformation from a set of requirements and vague notions of what is desired, to a structural plan of the building and a plan of action for its implementation are the creative activities of new development. The requirements for building a house, for example, are expressed in functional terms (an architectural drawing showing the place and function of the rooms) as well as in terms of a construction drawing following specified building standards Building standards have developed based on a long tradition of what constitutes a good house (i.e. a house that can withstand strong winds, moisture, wear and tear, etc.). With respect to determining what constitutes good software, the software industry is maturing, but still has a long way to go.

In planning for the development of a custom designed house, the architectural drawings and construction drawings may be the only basis for examining the building prior to its actual production In some architectural approaches, a minitiarized scale model of the house may be constructed However, when a series of houses is to be developed, where all of the houses have the same basic building architecture, a scale model, and most often one or more sample houses (prototypes), are constructed. The prototypes are useful for potential buyers to evaluate the functionality of the house in terms of their needs as well as serving as a means of 'debugging' and improving the basic building architecture.

In creating a modern building, in addition to building standards and norms, significant attention is given to approaches which exploit large building blocks based upon sub-assemblies of modules and components These practices make large scale building construction economically rational while at the same time insuring quality and safety in the final product.

Creative design thus takes place according to the architectural approach and follows step-by-step method(s) and process(es) with the assistance of tools used in reducing requirements to a viable building project architecture plan, including, when desirable, the creation of prototypes.

1.2.2 Construction

The first activity in construction is to provide implementation details concerning the architectural and constructional plans. That is to go from the more abstract towards a more concrete plan. After a sufficiently concrete plan has evolved, the production (implementation) takes place. Production is thus the last phase of construction. The number of people involved up to the point of production (even for large scale projects) is quite small in comparison with the number of people involved in actual production.

Production is the result of manufacturing the more abstract construction plans as well as the detailed construction plan. Further, the production activity may take advantage of any related models and/or prototypes that may have been developed. Here we can differentiate between custom built houses and houses developed for mass production. In the custom built case, the implementation is typically performed by artisans who are specialists in their particular field (woodworkers, plumbers, perhaps sculptors, etc.) The aesthetic properties of the custom construction are central In the mass production case, we find the need to employ people with less skill, but who can carry out their detailed work in a cost effective manner. Thus a clear unambiguous method and process by which the construction plans are followed (with the assistance of appropriate tools) are essential ingredients of successful mass production.

Responsibility for large scale building projects is most often placed in the hands of an entrepreneur. The entrepreneur takes responsibility for the production according to the building architecture and construction documentation From this point, the entrepreneur develops the detailed construction plan as well as the process in preparation for production. The entrepreneur, in turn, enlists the services of subcontractors who take responsibility for portions of the total project. In order to use subcontractors effectively, standards and norms and the usage of building blocks and components become vital so that the subcontractors can be relied upon to perform their services properly. Further, we find once again the importance of the method, process and tools which explicitly define and document the procedures to be followed by entrepreneur and subcontractors.

1.2.3 Long-term support

Building construction projects, custom designed or mass produced, must take account of the fact that the products should exist for many years Thus the architectural approach of this phase must take account of 'life-cycle' requirements for maintenance, alteration and extension. In the software industry, due to inherent flexibility of alteration, the existence of a philosophy containing an architecture that permits long term support is absolutely essential

1.2.4 Conclusion

During all activities, from the original product requirements, through the creative design activities, construction, production and for long term support; documentation is a vital aspect of rational industrial activity. The documentation must be appropriate ('understandable') for the various parties having a vested interest in the building project Documentation must be kept up-to-date following alterations, variations, experiences and so on, during all phases of the building project. In this area, computer aided tools make a major contribution to their own branch and to all other industrial branches The software industry, however, must learn from the traditions of other branches of industry concerning the information content and management of appropriate documentation.

The ability to reuse technology that has evolved during building projects is an essential part of profitability for those involved in the mass production of housing. Building blocks that have been identified and exploited must be well documented and understood so that they can be applied to new projects. In this regard, we can differentiate between building blocks and components Building blocks are typically larger units which have evolved during specific projects (for example, prefabricated walls); whereas standard components (for example, windows and doors) may have been used as parts of the building blocks. The software industry has, during the 1980s, started to realize the importance of components; however, to a large extent, the maturity associated with identifying and exploiting useful building blocks has not yet evolved.

From our characterization of the building process being based upon an architecture, method, process and tools, we can make the following observations concerning the results of the scaling-up process from which direct analogies can be drawn with the software industry.

- The process must yield a foreseeable result, irrespective of what individual performed the job,
- The volume of output does not affect the process,
- It must be possible to spread out parts of the process to several manufacturers/subcontractors,
- It must be possible to make use of predefined building blocks and components,
- It must be possible to plan and calculate the process with great precision,
- Persons trained for an operation must perform it in a similar manner.

1.3 System development characteristics

Having briefly examined some of the characteristics of a well established industrial branch, we can reason better about software development and identify the problems and areas in which OOSE provides new solutions We have, in fact, already identified several aspects of OOSE relating directly to the building construction analogy. As we move on to consider software development, the reader may think from time to time about applying the description of software system development back to the building construction analogy or by drawing analogies to other industrial and commercial activities. This bidirectional thinking will be quite useful in gaining a deeper appreciation of the industrialization of software development The rational 'architecture' provided in OOSE makes an essential contribution to the long term support, documentation and reuse of both building blocks and components. Further, OOSE places emphasis upon the management of change It is with respect to providing a rational architecture and related method, process and tools for software development that OOSE makes a contribution to the software industry.

1.3.1 Part of a larger activity

System development does not take place in isolation. It is part of a larger activity, often aimed at developing a product in which software is an integrated part. This is the case in large engineering related industries as well as in commercial enterprises such as insurance companies or banks In business data processing the product consists of the administrative services that the data processing department

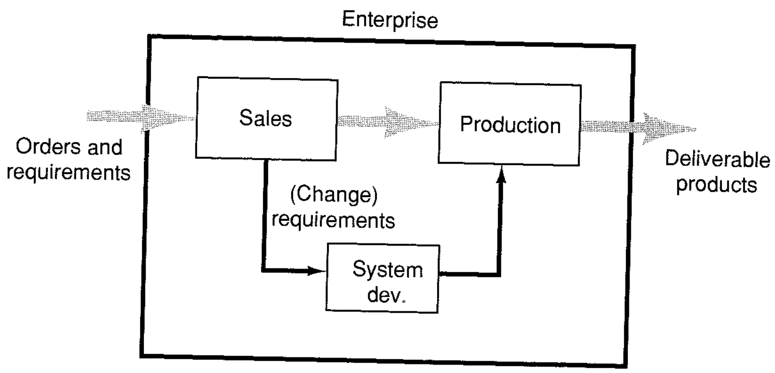


Figure 1.4 System development is normally an integrated part of an enterprise.

offers to the rest of the company Figure 1 4 portrays system development as part of the larger activity of a company or department whose products contain software

The activity viewed as a whole contains at least two other processes besides system development; namely sales and production. The main flow of activity typically passes directly from sales to production. The input consists of new orders from customers and the output consists of systems delivered to customers (We use the term customer broadly – another department within the same company can also be viewed as a customer.) The sales department orders product configurations for delivery to customers and formulates requirements for new products

An order should be formulated in such a way that it is immediately possible to identify the configuration of the final product. The production department delivers a complete system to the customer. Further, it should be possible to formulate an order in terms comprehensible to the customer without the aid of the system development department. Thus no programmers should participate in the production process, only persons skilled in duplicating products, assembling and configuring systems as well as testing them prior to delivery

Development of new services is only initiated as a result of the sales department conveying new customer demands to the development department. Again, terminology comprehensible to customers should be employed so that the need for the participation of the development department in customer contacts is minimized. This is possible if products can be described and ordered as sets of packages of functionality services, or as we will call them in OOSE, **service packages**.

In the development department, new software items (i.e. source code and/or other documents, for the production of systems) are developed, based on the new product requirements. The sales department can be informed about the new service packages later. Thus service packages play a central role in the development phase.

It is in terms of service packages that the staffs of the three subprocesses (sales, production and development) communicate. In order to achieve a rational return on investment, service packages should be designed so that they can be used in a number of different products. It will then be possible to build a large number of applications from a set of standard service packages.

A customer order corresponds to a product order, specified as a combination of service packages. The production department receives the order and assembles the finished product. To do so, they start from the source code for the service packages and transform the programs into the object code of a particular machine configuration. During these processes, it must also be possible to produce all other forms of documentation that are part of the finished system.

Figure 1.5 illustrates how the sales department provides a customer order as a combination of service packages to the production department which assembles the finished product for delivery to a customer. Each service package corresponds to one or more software items which must be configured in the intended combination. Treatment of the list of required service packages represents the initial part of production.

If these processes are to be carried out properly, the service packages must be developed with great care so that they can be configured for a number of product variations. Reuse must be reflected in the way the software is designed. Thus the aim should be to provide software items with substantial reusability both within the system (building blocks) and between different types of systems (components). Remember our analogy of the prefabricated wall (building block) and components such as windows and doors.

The building blocks are highly application related and form the basis for adaptations of the product for different customer categories. A building block corresponds to a service package or part of a service package. Further, the system development department contains a special group of people who are responsible for the development and coordination of general and application related components. The component developers provide essential support to product developers.

Reuse occurs in many product related activities and is not

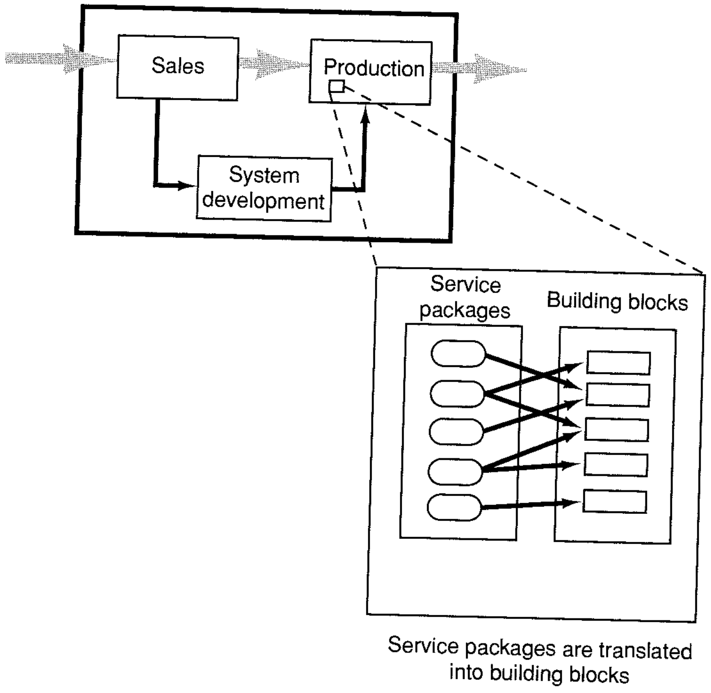


Figure 1.5 Orders from the sales departement form the basis for production. Each delivery to a customer consists of a configuration of a number of service packages which together provide the functionality

limited to programs The output of system development is, in fact, a set of **descriptions**. The descriptions include the source code which can be interpreted by humans and compilers, diagrams, flow charts and so on. All of these descriptions must be framed (self-explained) for reusability. Further, the knowledge of how to organize and manage projects must also be documented in a framed manner and made reusable.

When proper framing is achieved, rationality of software development and product exploitation can be attained. Service packages form the basis for the configuration of a system for a particular customer. Each customer receives their special combination of service packages with relevant documentation which has been assembled from appropriate descriptions. For a new release of a system, it is possible to reuse descriptions from the previous release in a controlled manner so that multiple releases of the same document need not be maintained.

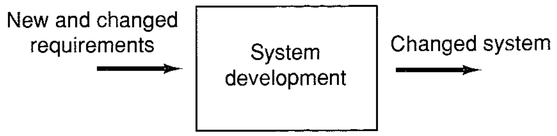


Figure 1.6 System development is a process of successive changes of systems from new and changed requirements

1.3.2 System development

Let us now concentrate on the subprocess of system development. As requirements change, the system changes, see Figure 1.6 The changed system actually consists of altered development information (descriptions) to be used for the production process

System development is carried out in a number of steps, each of which constitutes a more detailed and concrete development of earlier activities Thus it can be observed that system development is a gradual transformation of a sequence of **models** The first model describes the customer's requirements and the last step is the fully tested program. Between these two end points are a number of other models

System development can be viewed as a process of producing model descriptions. This is true of all levels – analysis, design, implementation and testing. In this context, the source code is seen as a description that can be understood by programmers and also by the production process (a compiler and linker). The descriptions present models of different degrees of detail Early models are quite abstract, focusing on external qualities of the system, whereas later models become more detailed and instructional in the sense that they describe how the system is to be built and how it is meant to function.

The aim is to divide the complicated development of a large system into a number of activities and make it possible for several designers to take part at the same time Each partial model is an abstraction of the system which enables the designer to make the necessary decisions at this level in order to move closer to the final model, the tested source code Each modeling step adds more structure to the system. Further, each new model is more formal than the previous one. To make the transitions between the different models as simple and faultless as possible, it must be straightforward to relate the model in one activity to the model in the following activity. We say that two models are **seamlessly** related to each other if notions which were introduced in one model are represented in the other model in a very simple and straightforward manner

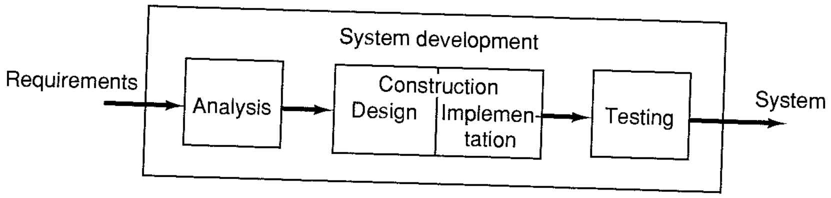


Figure 1.7 System development can be divided into three activities.

In essence, system development consists of three distinct phases that follow each other seamlessly – **Analysis, Construction** and **Testing**, see Figure 1.7. In analysis an application-oriented specification is developed to specify what the system offers its users. At this early stage, when changes are still relatively inexpensive, the aim is also to find a good structure for the system, namely a structure that is robust against change and which is divided into clear, comprehensible and indivisible units that can be ordered (i e service packages). This specification, which we call the analysis model, specifies the functional behavior of the system under practically ideal circumstances and without regard to a particular implementation environment. In other words, initially we disregard any restrictions that might exist in implementation artifacts such as the programming language, the database management system or other surrounding supporting components. It is important, however, to judge whether the analysis model can actually be realized under the given circumstances, for example with regard to performance requirements and/or development costs.

During construction, the idealized conditions of the analysis will gradually be replaced by requirements from the chosen implementation environment. In this phase, how the application-oriented analysis model will be realized with the aid of such components as system software, database management systems and user interfaces is defined. The construction activities constitute design and implementation The design activities formalize the analysis model in terms of the implementation environment and specify the identified building blocks The separate programs (blocks) identified in the design are then coded (i e implemented)

In testing, the system is checked to make sure that all of the service packages in the analysis model have been correctly implemented and that the performance of the system meets requirements Testing takes place at several levels, from specific functions to the system as a whole.

These activities look very similar to a waterfall model. In fact, it is if we are only concerned with a specific project However, our view must raise from the project to the product This change of view is essential and will be further discussed in the next chapter.

1.3.3 The transition from analysis to construction

All development methods divides the development work into different stages which may vary from project to project. In one project, a formal analysis model may be required so that different contractors can be asked for bids; the formal analysis model then guarantees that the final design and implementation correspond to what was ordered. In another project, however, it may not be obvious how the implementation environment affects the system requirements. A less formal analysis model can then be chosen in order to make the analysis less implementation environment dependent.

At present, formalism during the analysis phase should be restricted to the syntax and semantics of the static structure of the system. For this critical phase, no sound, practical, strictly formal technique have come to our knowledge to satisfactoraly specify the system's dynamic behavior. A more practical, descriptive technique is therefore preferable to a mathematical, formal method that is not yet fully mature. A formal technique is better used later on, especially during implementation. Whenever the more formal techniques get mature, these will probably be preferred.

Even though the boundaries between analysis and construction may seem vague, there are certain guidelines for what should be described during analysis and what should be dealt with during construction, see also McMenamin and Palmer (1984).

- The analysis is independent of the implementation environment. Changes in the implementation requirements thus do not affect the analysis result. Even if an important part of the system, such as a database management system, is replaced during implementation, the analysis model is not affected.
- The analysis model is application-oriented. The work is carried out in an ideal world; memory, performance and fault-tolerance requirements are set aside
- The analysis model describes the elements of the application in application-related concepts such as service packages. Given this foundation, the structure of the implementation mirrors the structure of the problem, rather than the other way round.
- The analysis model should not be too elaborate, as some of this work must be adapted to the chosen implementation

environment. Such adaptations may be difficult if the analysis model is too formal.

Given the properties of system development that have been described, we conclude that one approach to rational system development is provided by the conceptual framework of OOSE. This is a basic theme of the book.

1.3.4 Requirements are input to system development

The primary input for the development of a system is a requirements specification. This will have been developed from facts about the environment that the system is to serve. For 'technical' applications, such as tactical command and control systems, process control systems or telecommunication systems, the role of the system in its environment is identified, and the requirements of the system are formulated in terms of the behavior of sensors and actuators. For 'administrative' information systems, such as order-entry systems, personnel administration systems or reservation systems, the work usually begins with an analysis of the needs, problems and development tendencies of the enterprise, see the side box on enterprise development. Based on this analysis, a new enterprise model is built where the computer-based information system forms an important part of the enterprise. In fact, the development of a new and changed enterprise is based on the existing enterprise. This is fully analogous to changing an existing information system. Thus the same observations concerning changes which we made concerning the software system are also valid for the enterprise. Although we here differentiate between technical and administrative systems, we do not provide a precise definition of either In reality, most systems include aspects from both areas.

Enterprise development

Enterprise development can be viewed as a generalization of system development Instead of developing a system, a whole enterprise is developed. In enterprise development, the company is seen from several different perspectives The aim is to identify the problem areas and suggest alternative solutions. One result may be to introduce an information system Enterprise development can be divided into the phases portrayed in Figure 1.8.

The **current state** description is a survey of the framework of the present enterprise including its aims and problems To begin with, the functionally different activities within the enterprise are separated, and an analysis is made to find out how each activity contributes to

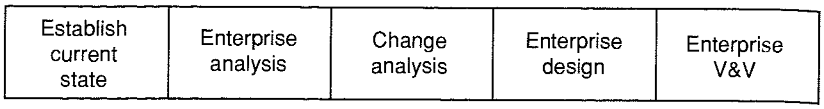


Figure 1.8 The different activities of enterprise development

the whole One activity may comprise several parts of the enterprise that are organized separately

In the **enterprise analysis**, an ideal (analysis) model is made of the present enterprise

In the **change analysis**, a detailed description is made of current problems and needs, and appropriate changes are suggested This will result in a changed analysis model.

In the **enterprise design**, the results from the change analysis are given a form which can be 'physically' realized, given the practical conditions of the organization

During **enterprise verification and validation**, the new enterprise is verified against the initial intentions and also validated in its new context

The enterprise design result serves, among other things, as input data for system development It is then usually given the form of a requirement specification for the information system which is to be developed to support personnel with different roles in the enterprise This requirement specification thus serves as input data for the development of the supporting information system Moreover, several aspects of the intended information system which are directly applicable in the system development have been captured during enterprise development

For administrative systems, the requirement specification is usually developed in a dialogue between customer and producer. It forms the basis for the decision to order the system. When the requirements of the enterprise are less well known, the specification work is preceded by enterprise development (as described above). In some cases, the customers are highly experienced and can provide the specification during the initial contact with the potential producers. In other cases, the customer approaches a producer and solicits assistance in solving his problems. The producer must analyse the customer's situation, try to solve the current problems, and find solutions based upon a new technique. The result is that a requirements specification is developed. In administrative system development, the work to develop a requirements specification constitutes a considerable part of the development activity and is often done in cooperation between the user(s) and the system developer(s).

For technical systems the situation is often somewhat different. A common situation is that the software is to cooperate with other machine or software components in an overall system The requirements of the software system are then given by the interfaces to the environment (sensors and actuators), where the specification can be derived from a knowledge of required structure and behavior.

There are cases in which an operational system exists, developed with another (older) technique which is to be modernized. In these cases, the existing system documentation functions as a basis for the development of a requirement specification. This field is often called re-engineering, see Jacobson and Lindström (1991).

As will be shown, it is practically impossible (for administrative and technical systems) to foresee all the requirements of a system during the introductory specification work. In the next chapter we will consider incremental development to handle the difficulties in developing a good requirement specification.

1.3.5 A system is the output from system development

The output from any system development is a set of descriptions, often of considerable number They function as a basis for production in the production department and as a basis for product description in the sales department.

The most obvious result is the program code which constitutes the final, executable model of the system. The result also consists of other related documents which the users need in order to understand and utilize the system in the appropriate manner. In this context, the term users covers not only persons, but also the machines that are in contact with the system in any way. In other words, we include direct users, such as operators in a process industry, as well as personnel involved in error detection and normal maintenance, such as data base administrators In machine cases, the documents provide interface descriptions to other systems In other words, there are a number of different users who must all be satisfied by the set of documents which together constitute the final product.

Document and **product** are the two basic notions which together constitute the description of a system. The superordinate entity is the product (i e the system) consisting of a number of subproducts. All product and subproducts are described in a list of documents referring to the documents that collectively constitute the product. Finally, due to reuse, a product may refer to documents in other products.

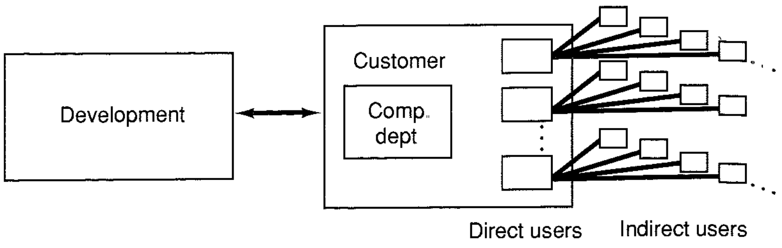


Figure 1.9 The interested parties of system development.

1.3.6 Parties interested in system development

The results of system development are used within other related activities of the enterprise, as well as being the products delivered to the customers of the enterprise. We will discuss here two of the parties interested in these products, namely the customer and the departments of the enterprise.

A system is developed by a producer based upon the order of a customer. A classic problem in all forms of product development is that the orderer and user are different parties. This will lead to conflicts about the aim, as orderers and users seldom come up with the same requirements. The conflict can be diminished, either by making the orderer understand and formulate the users' requirements or by having representatives of the users take part in formulating the requirements. The term user-governed or participative system development is used to classify group work resulting in a shared requirement specification. This work method has been used in the 1980s in the development of administrative systems.

The success of the system is highly dependent upon whether it has been possible to capture and formulate the users' requirements in such a way that the requirements can be formalized and transformed into working programs.

We must not forget that the direct users of the system often work to serve indirect users, see Figure 1.9. For example, policy holders are indirect users of an insurance company's information system, whose direct users are the personnel of the enterprise. If the direct users do not receive the system they need, this will affect the indirect users, namely the policy holders.

In system development, the users obviously play a central role. The system should be specified primarily on the basis of the users and their needs rather than on the buyer's requirements. The system should be validated to determine whether it really functions in accordance with the user requirements and whether it has been documented so that it describes the system from the user's perspective. In other words, we need a development process able to capture and deal with requirements from a user-oriented perspective. Once again, OOSE provides a new solution to this critical aspect.

The department that develops software products is only one of the parties having a vested interest in the development of the system. Other departments within the enterprise, such as sales, production and field service, are parties with an immediate interest in being able to influence the development process.

When the product is an administrative system to be installed in the enterprise to serve its own personnel, the data processing department is typically both the developer and the producer. The users are the personnel of the enterprise

Other groups, organizations and people that are also interested in the development include various managers, accountants, quality assurance groups, configuraton staff, operators, and suchlike.

1.4 Summary

In this chapter, we have introduced an analogy between the industrial processes of the well established building construction industry and the software industry. From this analogy, we have been able to compare and understand related aspects. In particular we have introduced the important terms architecture, method, process and tools and showed how they are related to both branches

Further, we have introduced the major characteristics of software development. We have identified several important areas where OOSE provides new solutions to existing problems in the software industry, in particular, the reuse of all forms of descriptions that are the result of system development. OOSE can be applied to the development of technical and administrative systems. The result of enterprise development provides an important input for the preparation of specification requirements.

The subject of the book has thereby been framed. Finally, it is important to note that the need for an industrial approach to sofware development was recognized many years ago by, amongst others, Doug McIlroy (1976). In a contribution entitled *Mass-Produced Software Components* he writes:

'Software production today appears in the scale of industrialization somewhere below the more backward construction industries. I think its proper place is considerably higher, and would like to investigate the prospects for mass-production techniques in software. [. . .] My thesis is that the software industry is weakly founded, in part because of the absence of a software components subindustry. [. . .] A components industry could be immensely successful.'

2 The system life cycle

2.1 Introduction

In this chapter, we take a closer look at several important aspects of system development which build on the industrial process thinking described in the introductory chapter. Explicitly, we consider the process of change in system development, reuse of program code and documentation and a deeper examination of methodology.

2.2 System development as a process of change

All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version (i.e. practically all systems). Most development methods today focus on new development, see Figure 2.1, treating revision work only briefly, even though it is known that changes constitute the main part of the total life cycle cost of most systems. An industrial process should therefore focus on system changes.

A system normally develops through changes in a number of versions. New development is, from this point of view, only a special case – the first version. That is, new development constitutes a change from nothing into something New development is nonetheless an important activity It establishes an architectural philosophy and constitutes the base of the system which must last throughout all subsequent development. A faulty base will have serious consequences for the life cycle of the system

Let us now relate Figure 2.1 to our earlier presentation of a developing organization being a part of a larger organization. Analysis begins when the sales department sends a requirement specification to the system development department or, more often, when it sends a specification of desired changes to an earlier version. We can call this a delta requirements specification.

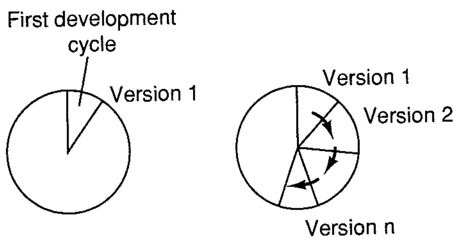


Figure 2.1 The first version of a system represents a minor portion of resource consumption during the life cycle of the system.

In Chapter 1, we showed that development work is characterized by the development of a number of models, each increasingly detailed, see Figure 2 2. All of these models are in reality delta models insofar as each model is a changed version of an earlier model at the same level.

In analysis, the delta requirement specification is the starting point from which to develop a delta analysis model. The delta analysis model is then delivered to those responsible for construction, at which point a delta implementation is produced. Each new version of the system is thus a delta version. System development is thus actually a process of progressive change.

It is often difficult to specify what versions of the system descriptions correspond to a particular release of the system. The number of versions is considerable for a fairly sizeable system. For example, different countries need different adaptations for local standards. In this regard, it is not sufficient to describe the system from the designer's perspective; it must also be described for other parts of the enterprise that will use it (e.g. marketing, production, installation, operation and maintenance). It is therefore vital that the system development enterprise employ an appropriate approach to the handling of successive versions of a system.

To achieve commercial rationality, it is essential to limit the number of versions of a system by using a technique which allows

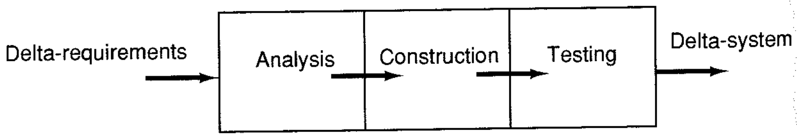


Figure 2.2 For each new version, the same development activities are followed as for the development of a new system. The difference is that the input data are composed of requirements for system changes

different installations of a particular release of a system to be configured in different ways. Each such configuration must be able to exist independently, and it must be possible to change it independently of other configurations of the same system. This implies that the versioning should, at least internal to the organization, not be on a system level only, but also on a subsystem or module level. Furthermore, it must be possible for different releases of the same system to exist side by side, each with its own installation configuration.

In many cases it is desirable to be able to offer the same system with partially different sets of functions for different customers.[1] Consequently, the arguments we have presented thus far are also valid in this situation. The number of change requirements will increase, leading to the need for strict planning of new releases. The system development process must therefore, by necessity, be framed in such a way that it simplifies the development of several parallel projects.

2.2.1 Incremental development

System development is usually regarded as a slow process, which can take several years from beginning to end. Historically, in most computer related projects, the requirements for the system as a whole are specified. Specification is followed by analysis, construction and testing of the complete system. This method can work if *all* requirements of the final system are known from the outset. This is, however, rarely the case.

It is the rule, rather than the exception, that the system requirements of information as well as technical systems are *not* fully known at the outset of the project. Knowledge of the system grows progressively as work progresses. When the first version of the system is in operation, new requirements appear and old ones change. Thus the system as a whole cannot be completely developed in the belief

[1] We use here the word 'functions' to express some functionality of a system. The word 'function' has come to have a bad reputation since it is sometimes used in opposition to object-orientation. This, we think, is unfortunate since it is a good concept to use in object-oriented systems. The same goes for the word 'structured'. In this book we have tried to avoid these words, but we believe that the words will come back into use when a balance between object-orientation and structural or functional techniques has matured.

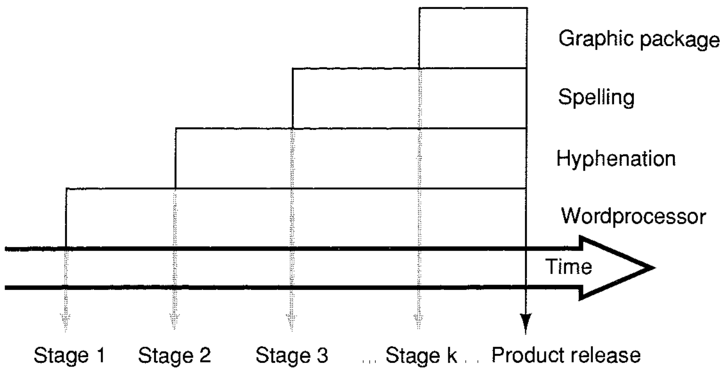


Figure 2.3 Systems are developed incrementally in a series of stages.

that the requirements specification will remain constant during the development period, which can be several years for large systems.

In most cases, it is better to develop the system step by step, beginning with a few of its core functions. As a 'correct' path becomes clear and a better understanding of how the system functionality evolves, new functions can be added. In this way, the system is **incrementally** enlarged until the desired level is reached (i.e the finished product is available, normally the first product release as portrayed in Figure 2.3). Such an incremental strategy also provides faster feedback during the development process. In practice, incremental system development means that we can divide the system into parts, corresponding to customer requested services.

As can be observed, each new stage extends the system with new functionality up to the finished product which comprises the whole of the desired system. Further, all subsequent releases of the system are developed as a series of incremental stages, see Figure 2.4.

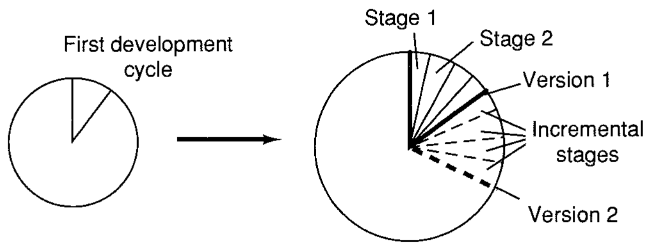


Figure 2.4 Every new version is developed incrementally in a sequence of stages

For the sequencing of development stages to be successful, it is essential to define stages which do not necessitate changing the results of earlier stages as the latter stages are introduced. Thus it is important to capture early those requirements which form the base of the entire system. How this can be done will be discussed later. Each stage is developed as a cycle and is tested before the stage is completed.

2.2.2 Prototyping

It is often difficult to determine how a system is supposed to work. The reasons may be technical or functional; they may have to do with efficiency or the user interface. It is helpful in such cases to develop a **prototype** of the intended system. Owing to the fact that a prototype often highlights certain properties of the intended system, other parts can be disregarded, and need only be given in schematic form. The prototype focuses instead on the properties into which one needs further insight. The prototype allows developers to experiment with a number of different design options. Thus prototyping serves as a complement to incremental system development.

A specific advantage of a prototype is that it can serve as a means of communication between the developer and the customer. It is much easier to express a view about something that can be demonstrated and used, if only partially, than to express an opinion about a specification. A specification cannot capture the dynamics of the system in the same way as a working prototype.

For these purposes, a prototype could be developed in an environment where an acceptable, functioning and easily modified system can be quickly created; this is called **rapid prototyping**. Rapid prototyping is in relatively bad repute today; there are even those who classify it as quick and dirty. This bad reputation has not been earned by the method as such, but rather by the way it has been practiced. In some cases, the prototype has actually been considered so good that it has been kept, and full system development has been deemed unnecessary. This is fine, so long as those involved know what they are doing. Such usage of the prototype may, however, result in malfunctions, or lead to various practical difficulties when modifications are required.

Prototyping is a useful technique for comprehending an application. Furthermore, if the prototype is carefully designed (through usage of the normal development cycle of analysis, construction and testing), it may also prove useful in the final system.

Correctly used, prototyping deserves a far better reputation than it has at the moment.

Finally, we can conclude that prototyping is an excellent working method, keeping in mind that it differs from incremental development in that the aim of prototyping is not to create a product, but to emphasize and demonstrate certain properties of the intended system.

2.3 System development and reuse

A common desire in all development work is the ability to **reuse** the results from earlier development work. Reusability has become a real buzzword in software engineering, often promised to be the solution of the software crisis, see Cox (1990). Of course, reusability is very important, but, relative to other engineering branches, we speak very much about reuse. In other branches this is not a very central area since it is so obvious and is practiced widely. The reason that there is so much discussion about it in software engineering is that we have been so unsuccessful in reusing existing parts. Thus our view of reuse must change into something that is self-evident

The necessity of reusability is of course applicable during coding, since it can influence productivity significantly. This is the usual context when software people talk about reuse. So there is no reason that prevents us having it on the code level, but it is actually not the only interesting type of reuse in software engineering. What can give even higher productivity enhancement is reuse in other development phases. Other parts of the construction phase may benefit when reusing entire designs in several systems. Additionally, reuse should also be viewed as natural during analysis and testing.

When developing an engineered product it is desirable to be able to choose between existing units (application modules and components) and develop the system from a set of reusable parts. In the previous chapter we considered how such reuse is exploited in the building construction industry. We would like to develop software in a similar manner.

Reuse should thus occur on several different levels of granularity. A system is assembled from a set of application modules. A module, in turn, is composed of other modules or of components. The application modules at the lowest level consist of components only. In other words, system developers must have access to a set of components in order to build application modules. The components are thus the finest level of reuse. It is here that reuse must start. However, to increase productivity even more, we also need to reuse larger modules. This can be accomplished by reuse in earlier phases when the system is being structured.

Problems with reuse include the finding, understanding and appropiateness of the thing to be reused. Object-orientation gives a completely new technique that strongly supports these issues. To be able to reuse parts already developed in a product is an important way to decrease significantly the product's life cycle cost. As we will see later, OOSE focuses largely on this when structuring the system.

2.3.1 Components

First of all we need a technique to be able to build really good **components**. By components we mean already implemented units that we use to enhance the programming language constructs. These are used during programming, corresponding to the components in the building industry (i.e windows, doors, etc.). They must be powerful, have simple well-defined interfaces, be simple to find, learn and use, and have a wide area of application. Moreover, it should also be possible to build new components with the aid of existing components.

Traditionally, software components have been available in the form of procedures and functions for numerical and statistical applications. These should be complemented by software components providing, amongst other functions, the buffers, queues, lists and trees that are often needed in the programming of algorithms. They may also provide windows, icons or scroll bars for human user interfaces. These components often form a good base to start with. It is not the number of components, but the usefulness of them that determines the reuse success.

Components are listed in catalogues. For catalogues stored in computer files there are special tools (browsers), by use of which it is possible to consider the components more carefully. The catalogues for electronics components or building construction catalogues provide useful analogies. Let us therefore compare the situation of the programmer and the electronics designer. The electronics designer's shelves contain catalogues of electronic components from different firms, whereas the programmer has computer science and software engineering textbooks on his shelves. On the whole, the programmer must carry his knowledge of standard solutions around in his head while the electronics designer can find standard solutions in handbooks. It may be possible to reuse entire programs, but the opportunities to reuse parts of standard programs are few with today's design techniques. On the other hand, it is often difficult to design entire programs that are reusable Therefore a programmer often finds it easier to write a new program than to find and learn an old one and then change it However, these kinds of handbooks are beginning to appear on the market.

There are, of course, a number of macrolibraries, sub-routine libraries, procedure libraries and so on available for general use. However, with the exception of the libraries for numerical and statistical computations identified above, they have not proved very successful. As a rule, today a programmer does not look for ready made routines that can be referred to via a detailed interface. He/she prefers to build their own routine by using part of an earlier program as a model and then making the necessary additions and changes. In such situations, the well-defined sub-routine library is less suitable. It is only in recent years that component libraries have begun to be found on the market. These are still quite simple, supporting mainly data structures and window management systems, although they are often very good as a starting point

2.3.2 Changeable applications

An application must be designed so that it permits constant alteration. The most rational manner of building an application is to join changeable application modules, each of which is built using components. Consider once again the analogy with prefabricated walls in the building construction industry

The application modules must be framed so that requirements for changes, in all probability, occur only in one module and only rarely in two or more modules. Further, modules should be chosen so that they can be combined in different configurations for different customers or in different installed systems By partitioning the system in this way, modules can be reused for different configurations in one or several systems.

Moreover, the modules should be alterable in a straight forward manner without leading to a greater cost than is motivated by the size of the change. Today, a seemingly small change often results in a disproportionately expensive alteration Thus it must be possible to bring intuitive costs into agreement with real costs.

2.3.3 Reuse of other descriptions

As described in the introductory chapter, a system consists of the program code and a significant number of other related documents forming steps on the way from the requirement specification to the

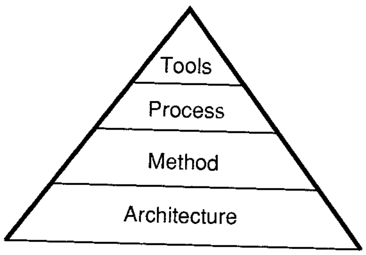


Figure 2.5 The foundation of a method is an architecture. The method may then be scaled up into a process, which, when defined, can be supported by different kinds of tools.

finished system. Apart from these documents, there are various forms of user-related documentation such as handbooks and educational material. It should also be possible to reuse these documents. Document reuse early in the development chain may have considerably greater effects than the reuse of application modules at the source code level.

2.4 System development and methodology

When developing large systems, it is important to know how the different steps of the method cooperate and how they fit into the development **process** as a whole. This section broadens the discussion to encompass both the development process and the basic ideas behind the **method**, as well as what it is that determines the selection of an **architecture** amongst a universe of potential architectures. Finally, we provide a few comments concerning how CASE **tools** should be designed to support the development, starting from the more fundamental properties of the architecture, method and process, see Jacobson (1991).

Consider the pyramid in Figure 2.5, first introduced in Chapter 1, where it is made clear that a method should be developed from an architecture. Further, from the method, a process, and the appropriate tools, can be designed.

2.4.1 Architecture

A key property of a software system is its internal structure. A good structure makes the system easy to understand, change, test and maintain. Thus properties of the system's architecture determine how the system must be treated during its lifetime. Using our previous building construction analogy, we observe that software systems can be compared to houses. A stone house and a log house have different properties; they are built and maintained differently. Unlike a house made of cards, however, they are both sound structures. Similar differences are found in software systems. For example, **function-data** methods that separate data from functions have proved, in the long run, to be a house of cards. Small changes in such a system, for instance a change in the format of a date, can have significant consequences. Object-oriented systems, on the other hand, are composed of a number of communicating and well-delimited objects. Such systems are easier to develop and understand, as well as being simpler to maintain and modify. Thus we can assert that object-oriented systems have important and desirable architectural properties. These architectural properties are extremely important and form the basis of the method. The architecture thus defines the types of models that can be built and what characteristics each model will have. For instance, the following types of models may form an architecture: an analysis model, a design model and an implementation model.

2.4.2 Method

As discussed earlier, a method is a planned procedure by which a specified goal is approached step by step, not to be confused with methodology, which is the science of methods. Most work descriptions for program development are method descriptions. They describe, often in a very abstract manner, how one should think and reason in developing a software system. Most methods also indicate the sequence of steps to be followed. The different steps of a method can be divided into still more detailed elements, all describing how the work is to be carried out assuming a certain underlying basic architecture. A method is based on a preconceived notion of the architecture of the working system. This means that the description of the method is formulated in terms of the concepts of the architecture to be realized.

A basic requirement of a good method is that it simplifies the development of systems of a particular architecture. Thus a good method for object-oriented system development should help us to identify the appropriate objects. This may seem obvious, but many methods for object-oriented development actually treat this requirement quite superficially. They imply that the objects can easily be found directly from the activity being modeled. It is true that many objects of a system should have real-life counterparts, but they must also be motivated from the point of view of the architecture of the system. Which objects are needed, and how detailed they must be, depends to a large extent on how the system is intended to be utilized. It is, for instance, hardly appropriate to include the entire spare parts information system in the system that supports the sales department, whereas it is most important to include it in the support system for the service department.

It is usually not difficult to find suitable information-carrying objects in an enterprise. However, it is the dynamics describing how the system is utilized that are difficult to define correctly. Many people assert that the dynamics need no particular modeling: it simply constitutes operations on information-carrying objects and therefore can be included in them. However, systems often exhibit behaviour that cannot naturally be said to belong to any particular information-carrying object. For this reason it is better to model separate dynamic objects.

2.4.3 Process

A **process** is the natural scaling-up of a method. We described the relationship of method and process for our building construction analogy. To emphasize this essential difference, let us consider one further analogy. Producing a new chemical substance in the laboratory differs greatly from producing the same chemical on an industrial scale in a factory. In the laboratory, the goal is to find a method to produce the chemical. To make this method appropriate for large-scale industrial use, a process must be defined. This usually means changing the working method Nobody would dream of industrializing the laboratory method by simply building a larger laboratory with gigantic test tubes and Bunsen burners. Yet this is often the way system development methods are scaled up for large projects.

The solution lies in changing the working method so that it can be scaled up and carried out with great parallelism – as a process. If the method is developed for use on a large scale from the start, the growing pains will not be so intense when the work has been scaled up. It is therefore advantageous to adopt a development technique for designing large systems from the start.

Let us clarify the difference between method descriptions and process descriptions. Methods are typically described in a waterfall model – a step by step procedure (with various degrees of granularity in the steps). This is similar to saying that a method is described as

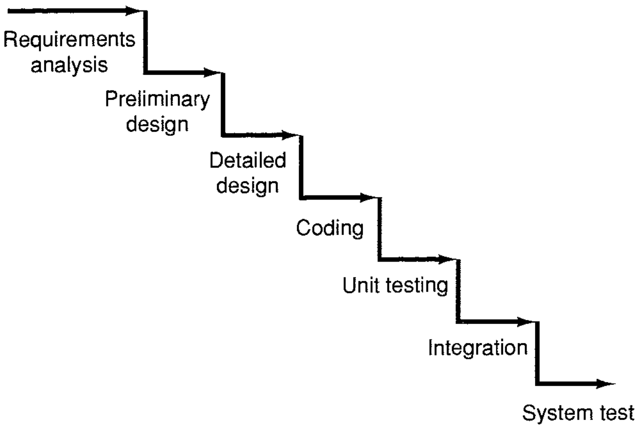


Figure 2.6 A method is described as a project developing the first product version. The figure illustrates a typical waterfall model

a system development project for one product version. It is often described as the first development project because a method is normally presented as starting from nothing and resulting in a first system version, see Figure 2.6. Note that often maintenance is an activity of its own in a waterfall model. However, maintenance also includes analysis of the requirements, design, testing and so on, and thus can be described by these other activities.

A development organization not only has projects for developing the first version, but also several other kinds of projects like maintenance projects. These other projects include projects for changing an existing system, tendering projects, error handling projects and so on. All these different types of projects include activities for analysis, design, testing and so on. Thus the activity descriptions should be reusable in different kinds of projects in an organization ready to develop systems in an industrial manner. A development process defines these activities and can thus offer more than one project type. Different activities are combined for different project types and the process should offer all the project types in which its supported organization participates (i.e. new development, further development, change, error handling, etc.), see Figure 2.7. The process should thus focus on a specific system and describe how a product is handled during its lifetime.

The process will continue to exist so long as the developed system is in operation, whereas a project only exists for a limited period of time. To summarize, a method is described as the (ideal)

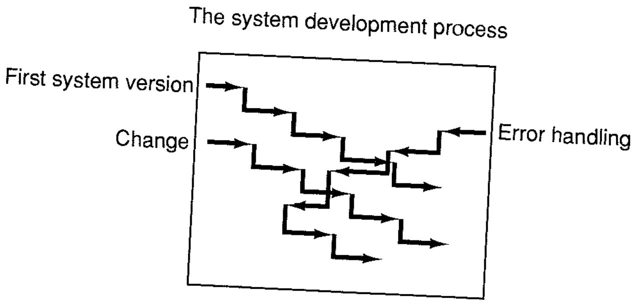


Figure 2.7 A development process should describe the activities required to manage a product over its lifetime. These activities are combined to describe different types of projects

development of a first version of the system, whereas a process is described as the ideal principle of organization managing the system during its entire lifetime.

Just as a method can be split up into a number of phases with underlying steps, a process is composed of a number of interacting subprocesses, see Figure 2.8. These subprocesses define the different activities of the development and must be defined in such a way that they are clearly delimited, thus enabling each activity to be performed as independently of other subprocesses as possible. Each developer is responsible for the work carried out in one or more subprocesses and thus makes use of the information sent to those processes. Moreover, the developer should also design the objects that other processes need as the basis of their work. The term **software factory** has been coined to describe this division into processes and subprocesses. Each of the subprocesses must be described in terms of how it works, the input it requires and the output it produces in the process descriptions.

Taking this point of view, the development process itself can be regarded as a set of communicating objects. Processes are thus a

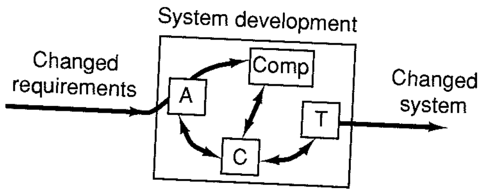


Figure 2.8 System development can be described as a set of interacting subprocesses, such as analysis, construction, testing and components

kind of objects. This will be further examined in Appendix A. A process is thus an application or, if you like, the industrialization, of a method. In summary, a process presupposes a method which, in turn, presupposes a basic architecture

2.4.4 The process expresses more than the method

The transition from principles to practice requires attention to a number of new aspects. A process must be able to express much more than what is covered within the underlying method.

The process description defines how different subprocesses should cooperate and to what extent they should be carried out in parallel, that is, how the people involved in the project should cooperate. Each subprocess is independent of the location of other subprocesses. Development work can therefore be split up and carried out at several locations Another way of expressing this is to say that a process description is an enterprise model for the system development organization, see Appendix A.

When system development is industrialized, the activity becomes less dependent on individuals. A handicraft is extremely dependent on the craftsmanship of an individual artisan. An industrial process has fewer key persons. Most work tasks are well defined and can therefore be moved between different individuals. The whole process is more resistant to disruption due to the advancement of developers or other personnel changes.

If system development is viewed as a process, it becomes natural to see development as a process of change. All work is developed relative to the existing system. This is true both while maintaining one version of a system and while developing a new version. A well-organized process provides version control as well.

It is possible to replace a subprocess with a new subprocess performing an equivalent task. This is one way of adapting the development process to a completely new kind of application, or of making a change in the development environment, such as employing a new programming language. Some subprocesses may be of such a type that they cannot be designed in exactly the same way in all types of projects. They will have to be replaced by a process adapted to each specific project. Yet this special subprocess must retain the same interfaces to other subprocesses. This requirement is especially true for the subprocess responsible for implementing the modeled objects of the design. For example, it will be customized for each specific database management system and programming language.

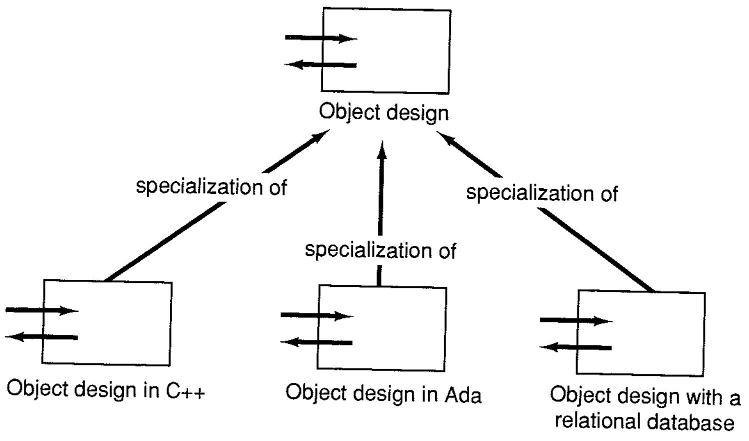


Figure 2.9 Process classes can be specialized.

To be able to describe the **specialization** consistently, we allow process classes to **inherit** other process classes. In an analogous manner to specialization in object-oriented programming languages, we say that a process can be a specialization of another process class, see Figure 2.9. We can start out from a general process containing the principal behavior and make a number of specializations to describe separate properties.

2.4.5 Computer-aided systems engineering

In the sections above, we have argued that the development of large systems requires that the development method be broadened into an industrial process. To be fully efficient, the development work also needs computer-based tools, namely a CASE (computer-aided systems engineering) environment. Introducing tools need not change the actual form of the process, though it often does.

When large systems are developed, all documentation must be consistent. This means, among other consequences, that an object must be given the same name everywhere in the documentation; this is not necessarily a simple task. However, a tool that supports hypertext links, or even better, object-oriented text, see Bergner (1990), simplifies this task. When developers refer to an object in a document, they add a reference to the desired object instead of writing its name. The tool then presents the reference as, for example, the object's name

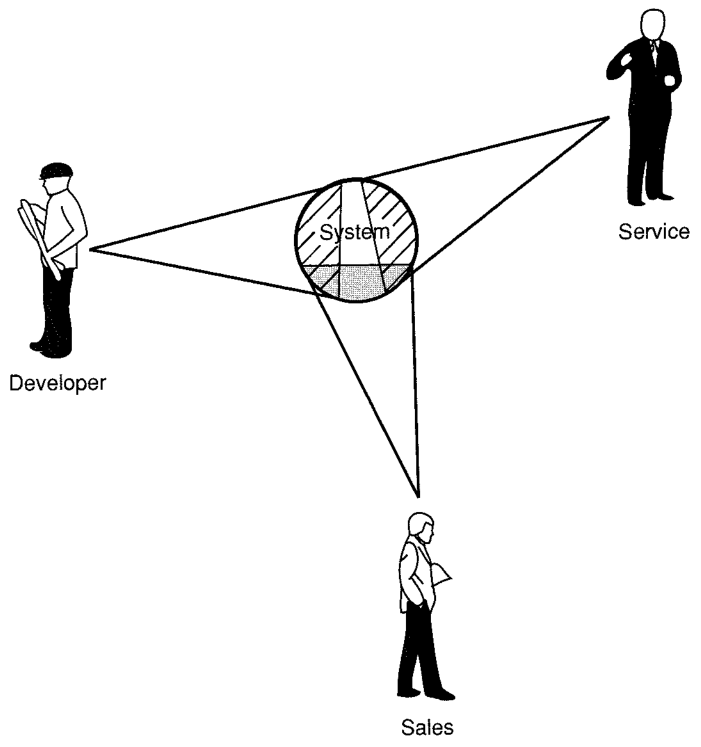


Figure 2.10 Different persons need different views of the system. These views may be composed from the same information

Tools can be used to automate a large number of tasks, especially trivial ones of cumbersome volume. If the tasks in the underlying process are seamlessly connected, a great deal of the work can be automated, as the output of one subprocess can easily become the input of another.

When documenting a complex system, it is desirable to present the same information in several ways so that different people can gain a better understanding of the system that is being modeled, see Figure 2.10 Developers typically want to view associations between objects as purely textual descriptions in a document, arrows in diagrams and compilations in cross-reference editors (browsers). A more sophisticated development environment may adapt this support to different categories of developers Each developer can then receive precisely and only the information that he or she needs. A project leader, for instance, may only be interested in overall object descriptions, whereas a designer needs to see all the details. Working documents as well as finalized documentation are also desirable.

A tool to be used on a large scale must support the process it is based on. The process can be integrated into the tool as an extension, for example, by supporting modeling decisions or handling change proposals.

Computer-aided tools can lead to massive productivity improvements, but they are only part of a greater whole. The choice of a development technique has its roots in the basic philosophy chosen to govern the overall system structure of the designed systems, namely the architecture. To this base is then added a method, a process and, finally, computer-aided tools

2.4.6 Small-scale projects

It seems reasonable that we need a method, a process and tools for very large projects, but can an appropriate method with good computer support be sufficient for small projects? In our opinion it cannot. Coordination around the maintenance activity is necessary even for quite small projects. Yet projects are often mistakenly deemed to be so small that they can be handled without very detailed instructions. In this case, the project members are left to solve problems as they arise. However, *ad hoc* solutions arrived at beside the coffee vending machine lead to bad work routines, inferior systems and project delays. Even so, projects both large and small, are sometimes carried out without a clear idea of how future maintenance work is to be carried out.

The conclusion is evident. What is needed is a system development process that can be packaged and placed into operation in an arbitrary enterprise. The development process naturally precedes the product development. Once it has been decided that program products are to be developed, it becomes natural to choose a reliable development process which suits software products Only then should the details of the product itself be considered. In this book we will primarily describe a method for system development. It is, however, important to realize that, like other methods, this method is not sufficient for successful system development. It needs to be extended into a complete process.

2.5 Objectory

We have now described some important properties that can be expected of a modern development technique:

- It must support the iterative development of a system over the entire life cycle,

- It should view each iteration as a change to an existing system,
- It must support the entire chain from changed requirements to the functioning system.

Objectory (the Object Factory for Software Development) is such a development technique. The framework of Objectory is a design technique, hereafter called **design with building blocks**, derived from the Swedish telephone exchange company LM Ericsson and now spread throughout the whole field of telecommunications. This technique is the essential idea behind the CCITT (Commité Consultatif International Télegraphique et Téléphonique) recommendation of SDL (the Specification and Description Language), see CCITT (1988)

With the building block technique, a system is viewed as a number of connected blocks, each block representing a system service. When all the required system services have been completely specified, building blocks are designed using a top-down approach keeping in mind the criteria of insuring that the system being developed can support changes to its functionality and can be adapted to new technology. This latter property is particularly important if systems are to have a long lifetime.

The building block technique has been employed for about twenty years in the development of large commercial systems. It has been employed in projects involving hundreds of designers, and today more than four thousand designers all over the world are using it. The telecommunications technique has yielded positive results when used in large systems, be they centralized or decentralized.

In combination with two other techniques, **conceptual modeling**, see Bubenko and Lindencrona (1984), and **object-oriented programming** (described in the coming chapters), the building block technique has been significantly improved. This combination has been developed further in Objectory in a number of ways:

- The technique has been generalized from telecommunications; it is now being used in various systems such as information systems, real-time systems, process control systems, CASE tools and graphic presentation systems. Furthermore, the technique has been adapted to different programming languages (for example Ada, C++, Smalltalk or COBOL), database management systems and operating systems, to name just a few. It has, moreover, been extended from being object-based to being object-oriented, see Wegner (1987), as it now supports inheritance between classes.

- The technique has been simplified and scaled down for application to small projects as well as to the large ones for which it was originally intended.
- The three techniques of design with building blocks, conceptual modeling and object-oriented programming have been formalized and tied together, made unambiguous and interrelated.
- Conceptual modeling has been extended with object-oriented concepts and the ability to describe dynamic behavior.

It is worth noting that the Objectory product itself has been viewed and described as a system, which made it possible to develop a new version of Objectory from an earlier version of Objectory. Thus we can conclude, as described in the introductory chapter, that the technique can be employed with equal potential during the enterprise or business modeling phase that often precedes system development. This is further described in Appendix A.

2.6 Summary

All larger systems will submit to changes during their life cycles. This fact must be considered in an industrial process approach to software engineering. System development is actually a process of progressive change as new and changed requirements will continue to be imposed on any product.

Incremental development offers a way to handle such changing requirements during development of a specific version. It also provides a way to speed up the feedback of a development. The complement to incremental development is to do prototyping early. A prototype normally aims at investigating and highlighting uncertain properties at an early stage of the system development. It is essential, though, to decide from the beginning the aim of the prototype; whether it should be incorporated in the product (with maintainability requirements as a consequence) or not

Reusability has failed in software engineering. Awareness of this has been commonplace for quite some time. Still, reuse is kept at a minimum level because of its associated difficulties. To increase significantly the productivity in software development, reuse must be a natural ingredient. Object-orientation gives us new techniques that are far better for supporting reuse than are traditional techniques. Reuse must occur at several different levels. During the coding phase is the most obvious, but it is also important to reuse previous work during other phases.

One of the most, if not *the* most, essential properties of a system is that it must have a stable structure during its lifetime. When defining a system development process, it is therefore important to have concepts and models that strongly support the development of such a structure. We call this foundation an architecture. How to work with these concepts and models in an ideal development is described by a method. The method should also support the goal of getting a stable system structure. However, when scaling up a method for industrial development of a product covering its entire life cycle, often involving several different types of projects, a method is not sufficient. We call the activities to support this a process. The process will express more than the method, since it describes the complete management of a product over its entire life cycle. To manage such an industrial development process, tools must be developed to support the work. The support often consists of different kinds of documentation, although other tools may also be appropriate. In general, as many activities as can be automated should be automated.

We have argued that we must change the way that we currently develop systems. This change has already begun. Important ingredients of this work are object-oriented technology, a process approach to development work and a well-developed tool box. In a few decades, system development will have advanced just as has the building construction industry. A large number of components will be available for reuse at all levels of detail, from small, general components to large, application-oriented modules. To a large extent, the development work will correspond to the process model that we have been describing. The basic architecture and foundations of the method, such as object-oriented technology, will have been developed and enriched.

We have briefly introduced a technique that yields object-oriented systems and that is itself object-oriented. The rest of the book will cover the fundamental ideas of this technique. Object-orientation provides the basis. In the following chapters, the reader will gain a deeper understanding of what object-orientation is and our view of Object-Oriented Software Enginnering, OOSE, and how they can be applied throughout development. The reader will also find case studies describing various application areas.

3 What is object-orientation?

3.1 Introduction

This chapter introduces some of the basic concepts of object-oriented technology. The aim is to introduce the actual idea, not to give strict and precise definitions. An introduction of object-orientation in use both for system development and for programming is found in the two following chapters.

Object-orientation is a technique for system modelling. It offers a number of concepts which are well suited for this purpose. The word 'system' is used here with a wide meaning and can be either a dedicated software system or a system in a wider context, (e.g. an integrated software and hardware system or an organization).

Using object-orientation as a base, we model the system as a number of objects that interact. Hence, irrespective of the type of system being modelled, we regard its contents as a number of objects which in one way or another are related Our surroundings, for instance, consist of objects, such as people, trees, cars, towns and houses which are in some way related to each other. Thus what the objects model depends on what we wish to represent with our object model. Another model of our surroundings would, perhaps, consist of taxation, government and politics as objects. The objects which we select to be included within our model are, therefore, dependent on what the object model is to represent.

People regard their environment in terms of objects. Therefore it is simple to think in the same way when it comes to designing a model. A model which is designed using an object-oriented technology is often easy to understand, as it can be directly related with reality. Thus, with such a design method, only a small **semantic gap** will exist between reality and the model, see Figure 3.1.

Interest in the object-oriented method has grown rapidly over the last few years. This is mainly due to the fact that it has shown

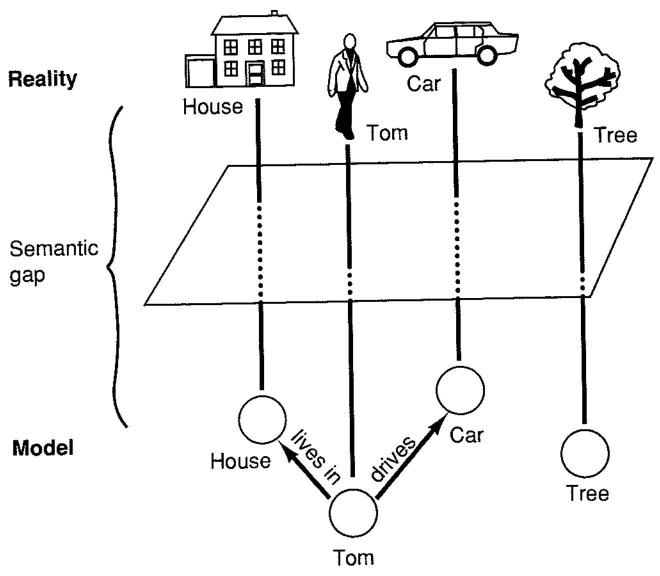


Figure 3.1 Since objects from reality are directly mapped into objects in the model, the semantic gap is minimized

many good qualities. Amongst the most prominent qualities of a system designed with an object-oriented method are:

- **understanding** of the system is easier as the semantic between the system and reality is small,
- **modifications** to the model tend to be local as they often result from an individual item, which is represented by a single object.

In the following sections, we shall introduce the basic concepts within object-orientation. This introduction is only an overview and is independent of both the programming language and the development method used. We shall not give any precise and formal concept definitions, but hope to provide you with a good understanding of these concepts. We shall use the concepts and meanings most commonly used within the object-oriented environment, see Wegner (1987).

We will introduce the concepts by simulating a familiar example. We have chosen an example easy to understand also for

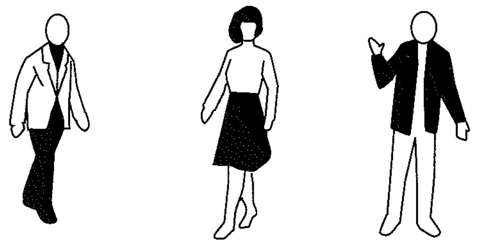


Figure 3.2 Tom, John and Mary are regarded as objects

a person unfamiliar with the fundamental concepts of system development. As mentioned earlier, object-orientation is ideally suited to creating models of real systems, and especially for simulating the system. From the example, we hope that you understand, and also know how to use, the fundamental concepts discussed. The example will be described further in the chapter on object-oriented programming.

3.2 Object

The reality we will describe involves a number of people who perform certain activities Our task is to try to model this system. We shall see that it is very natural to construct a model which simulates this reality.

The reality consists of a number of people, amongst others, John, Mary and Tom. We shall now consider each of these people as an object, see Figure 3.2. (We use 'person' and 'object' in this description to mean the same thing; we actually mean the object that represents the person. As always, one should be careful to separate the reality from the model.)

The first and most important concept that we describe is, of course, the concept of **object**. What is an object? The word object is misused and is used in nearly all contexts. What we mean by an object is an entity able to save a state (information) and which offers a number of operations (behavior) to either examine or affect this state.

An **object** is characterised by a number of operations and a state which remembers the effect of these operations.

An object-oriented model consists of a number of objects; these are clearly delimited parts of the modeled system. Objects

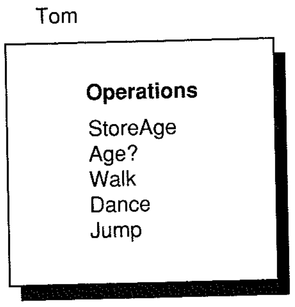


Figure 3.3 The object Tom and its operations

usually correspond to real life entity object, such as an invoice, a car or a mobile telephone. Each object contains individual information (e.g. a car has its registration number).

To each of the objects in the example, we thus attach the behavior and information that we wish to associate with the corresponding person. An example of the information which we, perhaps, wish to save for each object is age, address, male/female and so on. To access or be able to affect this information, we must, for each object, define a set-up of operations which can affect or read the saved information. We can also define operations which perhaps need not affect any internal information, but only perform a behavior (e.g. walk, jump, dance). The only part of the object we can see is its operations, the inside is hidden to us, see Figure 3.3. Note that, from outside, we really see only those operations that exist for behavior and not how they work. We can only see how the different objects perform their behavior if we look inside them.

Within the information any associations to other objects are also specified, for example an invoice knows which customer to invoice. The model's objects therefore have relations with each other. For instance, we wish to model that the people shall recognize each other. These relations can be of two sorts. Firstly, **static** relations, namely relations existing over a longer period, which means that two objects know about each other's existence. Secondly, **dynamic** relations, namely relations by which two objects actually communicate with each other. There is an abundance of different static relations in connection with semantic modelling, see Peckham and Maryanski (1988). We shall not describe these relations in any more detail, but content ourselves with calling them static relations The objects can thus be composed from other objects, for example Tom can be composed of his head, his arms, his legs and his body, see Figure 3.4.

By means of this **composition**, we can structure the object Tom in parts. The reason for structuring may depend on many factors. It

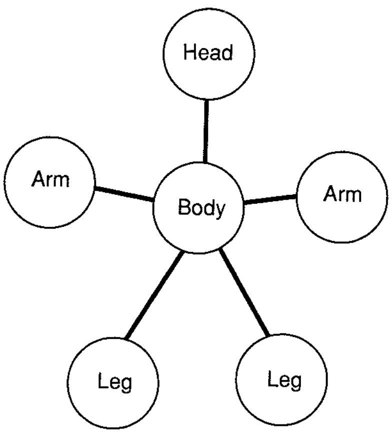


Figure 3.4 A model of a person can be composed with the objects head, legs, arms and body.

often depends on a combination of wishing to detailing the object, increase understanding and wishing to obtain reusable parts.

A similar method of joining different parts together is through use of **partition** hierarchies. This means that an object can be constructed from other objects, and those relations are therefore often called **consist of** relations. The word partition originates from Latin *partitos* and means divide. Figure 3.5 illustrates an example of a partition hierarchy.

A similar possibility for showing how something is interrelated is through the use of **aggregate**. Partition hierarchy and aggregate are used as synonyms, but there is a difference. The word aggregate originates also from Latin (the verb *aggregare*) and means to join together, which is the opposite of to partition. As it is difficult to express this grouping in a single way, a new object and a partition hierarchy can be used to express the aggregate. In a family relationship, the grouping of Man, Woman and Child establishes the aggregate Family. Since we cannot represent this triangular relationship, an object Family is added to express the joining of several objects. This object thus *represents* the aggregate, but it is not the aggregate itself. An aggregate is a union of several objects, and the union as such is often represented by an object of its own.

If we look at the inside of the object, we shall see both its information structure and how its operations work, see Figure 3.6. We can also see the information that the object needs to store, the parts the object consists of and how the behavior for the operations is defined.

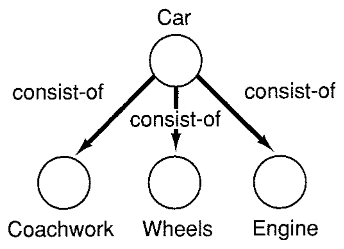


Figure 3.5 Example of a partition hierarchy. The object Family exists only to link together the other objects When one turns to the whole family, one turns, therefore, to the object Family.

The dynamics in an object-oriented model is created through the dynamic relations, by means of objects sending **stimuli** to other objects. We denote by the concept stimulus the event when an object communicates with another object In a programming context, the word 'message' is often used instead, but in order to avoid the message-semantic, we use the stimulus concept. Additionally, the word stimulus indicates that it stimulates some behavior to take action and does not necessarily include any message information. A stimulus, which is received by an object, causes an operation to be performed in the receiving object. This operation can in turn cause new stimuli to be sent Hence, if we wish an object to perform a behavior, we send a stimulus to this object. For example, if we wish Tom to jump, then we send the 'Jump' stimulus to him. When Tom receives the stimulus, he interprets it and performs what has been defined for him to do when this stimulus is received. Tom uses his legs and arms and therefore sends stimuli to his legs and arms, see Figure 3.7

We can also consider more complicated cases. If we want the object Mary to dance, we send a stimulus 'Dance' to the object Mary. When she receives this, she performs the behavior associated with dancing. She, perhaps, will only dance with her friend. She will therefore send a stimulus to this friend to start dancing, see Figure 3.8.

All information in an object-oriented system is stored within its objects and can only be manipulated when the objects are ordered to perform operations. The behavior and information are **encapsulated** in the object. The only way to affect the object is to perform operations on it. Objects thus support the concept of **information hiding,** that is, they hide their internal structure from their surroundings see Parnas (1972). Every one of the object's operations perform part of the object's behavior, and can modify information in the object. In order to use an object, we do not need to know how the object's behavior or information is represented or implemented internally. We only need to know which operations it offers.

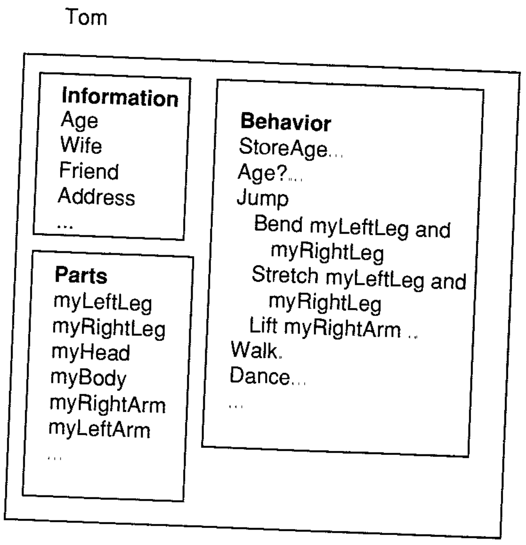


Figure 3.6 The inside of object Tom. Only the behavior for Jump is shown.

Note that we describe the problem in terms of items or objects taken from real life and what we wish these objects to perform. We do not know how John, Tom or Mary perform their tasks or how they look inside. We have used the concepts object, operation and encapsulation to understand the problem. Encapsulation means that all that is seen of an object is its interface, namely what operations we can perform on the object.

These important concepts have their roots in abstract data types, see Aho *et al.* (1983). An **abstract data type** is a model (structure), with a number of operations to affect this model. It is similar to our definition of an object and, in fact, they are closely related. Both are abstractions and are defined in terms of what they perform, not how they perform it. They are both generalizations of something specific, where encapsulation is the central concept. One

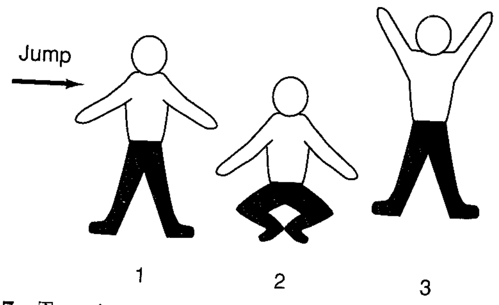


Figure 3.7 Tom jumps as a result of receiving the stimulus Jump

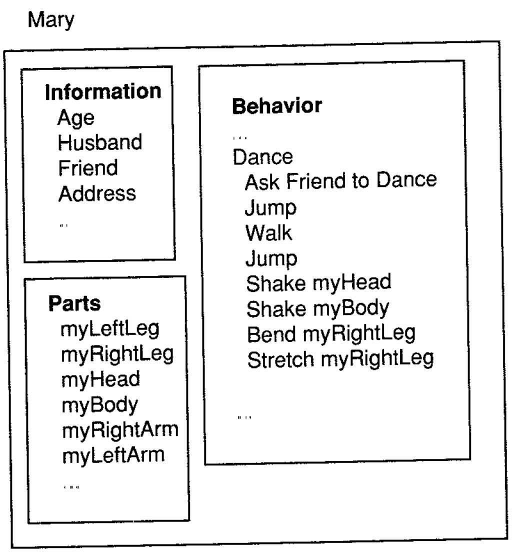


Figure 3.8 When Mary dances, she wants to have someone else to dance with

of the advantages with abstract data types is that they should be able to be used independently of their implementation (information hiding), which means that even if the implementation is modified, we should not need to modify how we use the abstract data types. Another advantage is, quite simply, the reduced complexity due to there being no possibility of becoming involved in their internal structure, but only the ability to use them according to their specifications

So far, we have only introduced the object concept. Many people using the word 'object-orientation' mean only this, but, as we have seen, we have actually not come much further than abstract data types. To be truly object-oriented, we need some further properties and concepts.

3.3 Class and instance

In the system we model, there will be a number of communicating objects. Some of these objects will have common characteristics and we can group the objects according to these characteristics. When we look at the objects in the example, we notice that all three people have similar behavior and information structure. These objects have the same mold or template. Such a group represents a **class**. In order to describe all objects that have similar behavior and information structure, we can therefore identify and describe a class to represent these objects.

A class is a definition, a template or a mold to enable the creation of new objects and is, therefore, a description of the common characteristics of several objects. The objects comprising a certain class have this template in common. As an example we can view this book. The book you are holding in your hand is an instance of the book. The book description at the publisher represent the class from where instances can be created.

> A **class** represents a template for several objects and describes how these objects are structured internally. Objects of the same class have the same definition both for their operations and for their information structure.

A class is sometimes called the object's **type**. However, a type and a class are actually not the same thing. As we mentioned above, an abstract data type is defined by a set of operations. A type is defined by what manipulations you can do with the type. A class is more than that. You can also look inside a class, for example to see its information structure. We would therefore rather view the class as one (of possibly many) specific *implementation* of a type.

By means of the class concept, we can associate certain characteristics with a whole group of objects, see Figure 3.9. We can consider the class as being an abstraction that *describes* all the common characteristics of the objects forming part of the class.

In object-oriented systems, each object belongs to a class. An object that belongs to a certain class is called an **instance** of that class. We therefore often use object and instance as synonyms.

> An **instance** is an object created from a class. The class describes (behavior and information) structure of the instance, while current state of the instance is defined by the operations performed on the instance.

We can thus define a class Person, and each object that represent a person becomes an instance of this class. In our example, we can describe a class Person, where Tom, John and Mary are instances of this class, see Figure 3.10.

The behavior of the instance and its information structure are thus defined by its class. Each instance also has a unique identity. Several different instances can be created from a certain class, where each instance is manipulated by the operations defined by the class. Different instances can be manipulated by different sequences of

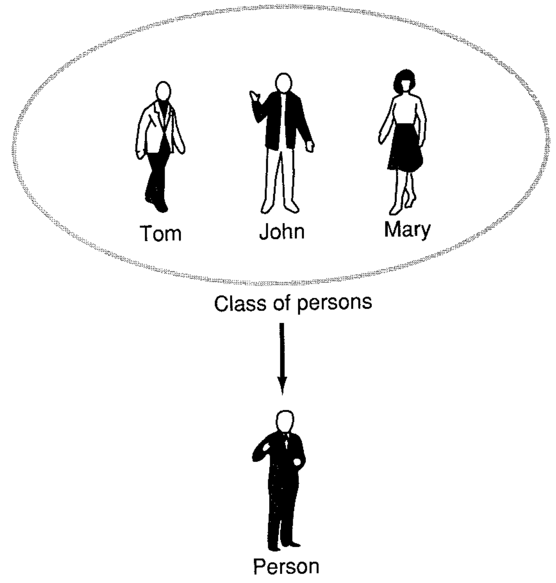


Figure 3.9 The class person describes what is common to all persons.

operations and, as a result, have different internal states. If these instances had been manipulated in exactly the same way, their state would also have been the same.

Now, we do not need to describe, for each person, how it shall appear, as this description is made in one place only, namely in the class. The class will therefore contain the information structure for, for instance age, and operations to examine or modify this age. When we have created an instance (e.g. John), we need therefore to store John's age in this instance. This is achieved by using the operation for storing age.

In the same way, we can make all the peoples' arms, legs and heads into classes. Thus we can define an individual class for head, arm, body and leg. The class for person can now use these classes to construct a person's body, see Figure 3.11.

As all people are instances of the same class, they will have similar behavior. If we wish to describe the fact that men and women have different behavior, for example when they dance, we will need to create two classes, one for Male and one for Female. John and Tom are created as instances of the class Male, and Mary as an instance of the class Female, see Figure 3.12. We must now, for both these classes, describe their behavior and information structure.

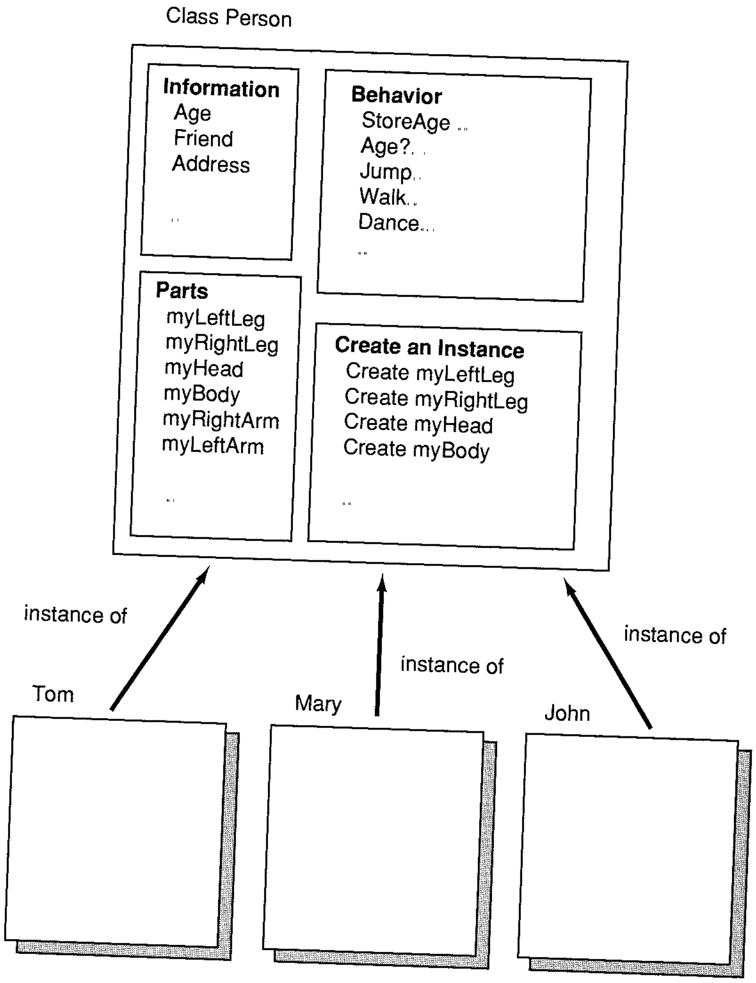


Figure 3.10 Tom, John and Mary are all instances of the class Person.

Why didn't we make John, Mary and Tom into separate classes instead? This is surely feasible and, if they had different behavior and/or information structure, we would have had no choice but to do so. We have simplified things by saying that all men appear in one way and all women in another. Both these classes have certain parts in common. At the moment, we must repeat these similarities, but later we shall describe how inheritance can be used to avoid this duplication of descriptions.

What is contained inside the classes Male and Female? In Figure 3.13, we can see both the information structure and the operations in each class, and we can see that they have a lot in common. The operation to create a new instance of a class (i.e. create

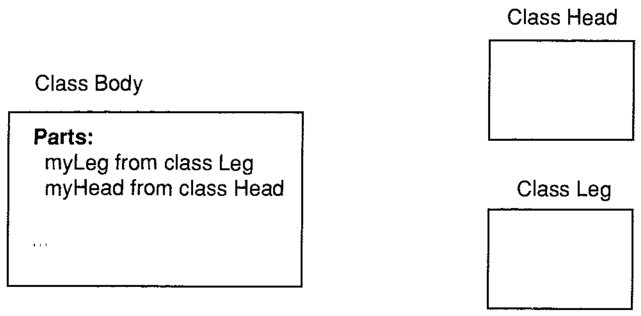


Figure 3.11 Instances of one class can recognize instances of another. The relevant classes are defined in the original class

a new person) creates also legs, arms, head and body for the new person.

We also see how we have defined the operation to dance. It differs only slightly, in this example, between the two sexes. Both men and women move their bodies, but in slightly different ways.

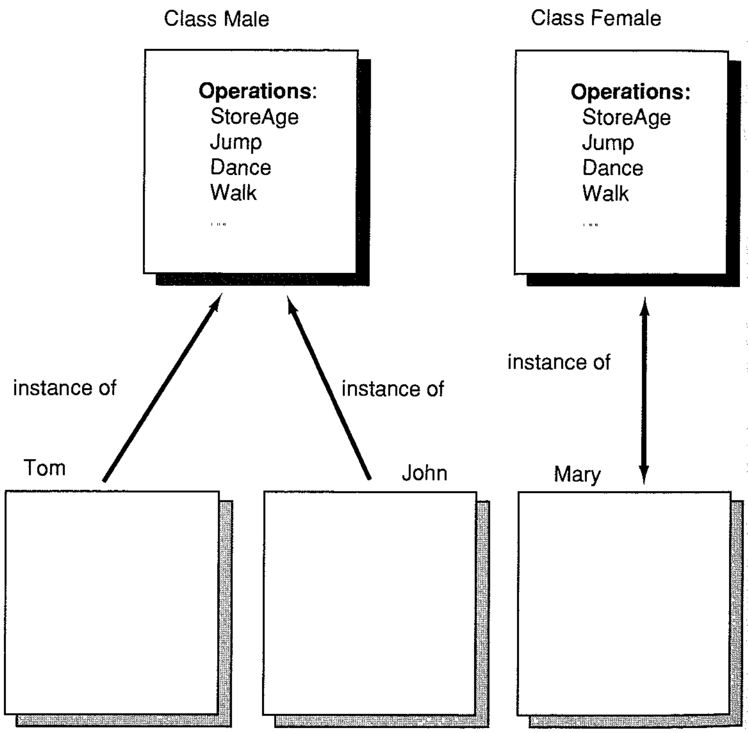
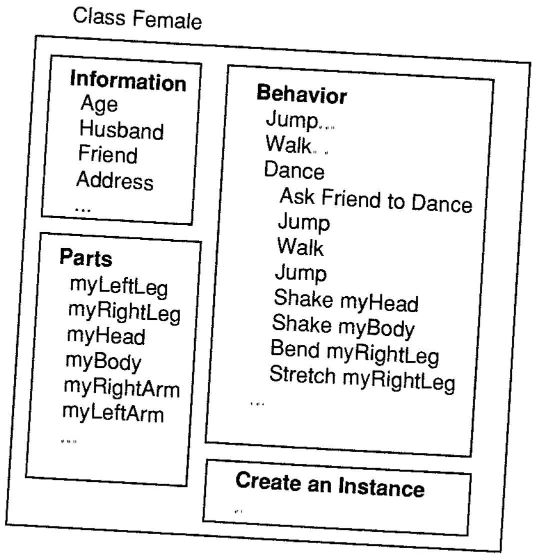


Figure 3.12 Male and Female are separate classes.



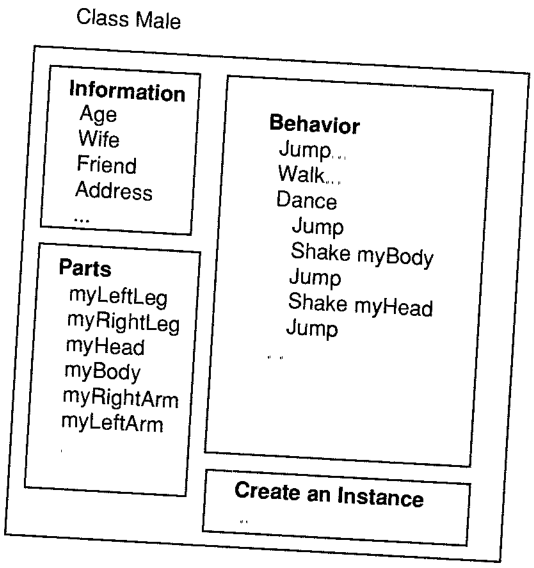


Figure 3.13 The inside of classes Male and Female

Moreover, women will only dance if they have a friend to dance with.

To summarize, we can consider the class as being the definition of operations and information structure for objects, and the instance defining an individual object's state.

3.4 Polymorphism

Instances, created from classes, will together provide us with the dynamic behavior that we wish to model It is when these instances start to communicate with each other that the system's behavior is performed. An instance may know of other instances to which stimuli can be sent. If an instance sends a stimulus to another instance, but does not have to be aware of which class the receiving instance belongs to, we say that we have **polymorphism**. Polymorphism means, at least in object-oriented contexts, that the sending instance does not need to know the receiving instance's class and that this class can be any class

> **Polymorphism** means that the sender of a stimulus does not need to know the receiving instance's class. The receiving instance can belong to an arbitrary class.

All people have a best friend. We can see this in the information structure of the classes Male and Female; both have information on a friend. We have not specified any special conditions for this friend, but we can assume that it is another person. A person's friend must, therefore, be either of the class Male or Female. We therefore have polymorphism in this model, as friend can refer to an instance of several classes. Mary's friend is either Tom or John and as these are both associated with the same class, Male, we have no use for the polymorphism characteristic. However, Tom's friend is either Mary or John, and as they are from different classes, class Male will not know with which class the friend will be associated. Here, the reference friend must be polymorphic. Polymorphism thus means, not only that different *instances* can be associated, but that these instances can belong to different classes. It is in the latter case, as with Tom, that we have use of the polymorphic characteristic.

A stimulus can be interpreted in different ways, dependent on the receiver's class It is, therefore, the instance which receives the stimulus that determines its interpretation, and not the transmitting instance. Often polymorphism is said to mean that one operation can be implemented in different ways in different classes. This is actually only a consequence of what is said above and not polymorphism in itself. (Polymorphism and dynamic binding are also often confused. Dynamic binding means that, not until the stimulus is sent, the stimulus is bound to a certain operation in the receiving instance's class We shall discuss dynamic binding in more detail in the chapter on object-oriented programming.)

As we have not specified any conditions on the class, we should, in theory, be able to associate an instance of any class to friend. It is therefore possible to associate an arm with friend. Therefore it is often necessary to restrict the receivers of a stimulus. If we know, in advance, the receiver's class, then we don't need polymorphism, but if we allow the receiver to be of varying class, within limits, we must specify this restriction in the polymorphism characteristic. Thus, we need to restrict the instances which can be associated. This is often called **limited polymorphism**. We require only instances of the classes Male and Female to be friends. This restriction can be made in different ways. Normally, inheritance hierarchy is used. We shall soon return to this problem.

Polymorphism is a very important characteristic for our models. It is thus the receiver of a stimulus that determines how a stimulus shall be interpreted, not the transmitter. The transmitter need only know that another instance *can* perform a certain behavior, not which class the instance belongs to and thus not which operation that actually performs the behavior. This is an extremely strong tool for allowing us to develop flexible systems. We have, in this way, only specified *what* shall occur and not *how* it shall occur. Through delegating what shall occur in this way, a flexible and modification resistant system is obtained. If we wish to add an object of a new class, this modification will affect only the new object, and not those sending stimuli to it.

3.5 Inheritance

When we describe our classes, it is soon noticed that many classes have common characteristics (behavior and information structure). For instance, when we compare the classes Male and Female, we see that they have much similarity to each other. These similarities can be shared between the classes by extracting them and placing them in a separate class Person. In Person, we describe everything that is common to Male and Female. In this way common characteristics can be shared by several classes. We collect the common characteristics into one specific class and let the original classes **inherit** this class. Male and Female then inherit Person and we now need only describe the characteristics that are new to these two classes. We therefore allow both Male and Female to inherit Person and, in this way, obtain access to all the characteristics defined there, see Figure 3.14. Male and Female now contain the same things as before, but their description has been simplified by means of inheriting Person.

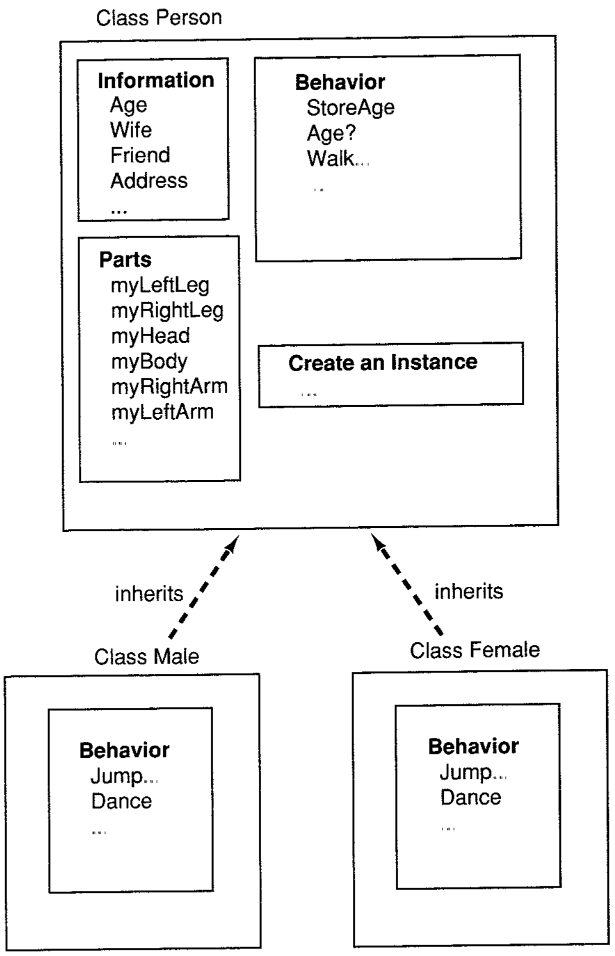


Figure 3.14 Classes with inheritance.

> If class B **inherits** class A, then both the operations and the information structure described in class A will become part of class B.

By means of inheritance, we can therefore show similarities between classes and describe these similarities in a class, which other classes can inherit. Hence we can reuse common descriptions. Inheritance is therefore often promoted as a core idea for reuse in the software industry. However, although inheritance, properly used, is a very useful mechanism in many contexts including reuse, it is

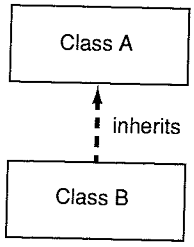


Figure 3.15 Class B is a descendant of class A and class A is an ancestor of class B.

not a pre-requisite for reuse We will come back to this issue in Chapter 11

Through inheritance, we have also obtained another advantage. If we want to modify some characteristic in Person, (e.g. how a person walks), it is sufficient to perform this modification in one place. Thus, if such a modification is made, both Male and Female will inherit this new definition of walk. Hence inheritance is very useful for easy modification of models.

We also see from the examples that the descriptions of Male and Female have been greatly reduced The only information remaining is that which *differs* between them. Inheritance is thus also useful to avoid redundacy, leading to smaller models that are more easy to understand.

The ease of modification does not only occur inside class descriptions. The adding of new classes can be easily done by just describing changes to existing classes. However, the adding of new classes may sometimes involve restructuring the inheritance hierarchy. We shall come back to this issue later.

By means of extracting and sharing common characteristics, we can **generalize** classes and place them higher up in an inheritance hierarchy. In the same way; if we wish to add a new class, we can find a class that already offers some of the operations and information structure required for the new class. We can then let the new class inherit this class and only add anything which is unique for the new class We then **specialize** the class

Classes lying below a class in the inheritance hierarchy are called **descendants** of the class. Classes lying above are called **ancestors**. Sometimes the concepts of sub-classes and super-classes are used instead, but we prefer descendant and ancestor. The reason for this is discussed in the side box on super- and sub-classes. Hence, if class B inherits from class A, then class A becomes class B's ancestor, see Figure 3.15. Class B is then a descendant of class A.

Since inheritance hierarchies may include several classes, we may want to emphasize the relation between two classes. If a class directly inherits from another class, we call it a **direct descendant**. The first class is then the **direct ancestor** of the second class. A direct ancestor is sometimes called the **parent** and a direct descendant is sometimes called the **child**.

In this book, inheritance relations are indicated by a dashed arrow drawn *from* the descendant *towards* the ancestor class We will also write 'inherits' (or shorter, 'ihs') on the arrow to avoid any misunderstandings Note that the direction of the arrow is *from* the descendant towards the ancestor. In some other notations the direction is the opposite. The reason that we use this convention is that the descendant should know about its ancestor, but the ancestor should not know about its descendant.

The inherit association is a **class association**, that is, an association between classes. Class associations are drawn with dashed arrows here. Associations between instances are drawn with a full arrow.

Ancestors developed with the main purpose of being inherited by others are often called **abstract** classes. Most frequently, instances are not created from abstract classes, although it is possible. A class developed with the main purpose of creating instances of it is called a **concrete** class. Also, concrete classes may of course be ancestors of other classes.

The concepts super-class and sub-class

The concepts of super-class and sub-class have varying definitions For instance, super-class is sometimes defined as the class directly above another class and sometimes as all classes above a certain class Sub-class is, in the same way, sometimes defined as the class level existing directly under a given class in an inheritance hierarchy and sometimes as all classes lying under a given class When the concepts are used in the right connection, it is usually understood what is meant, but concept confusion is not uncommon – at least for beginners within object-orientation To realize the confusion, those familiar with Smalltalk can ask the following questions Is the class Object a super-class to all other classes? Which super-class has the class Array? How many sub-classes have the class Object?

Another problem with the concepts is that the words super and sub have a meaning which does not really agree with how they are understood The concept super is often understood as something that is more capable than anything else In this context, it may mean quite the opposite. The inheriting class has often been expanded with some characteristics and is therefore often more capable than its super-class; but a descendant may also involve a restriction or specialization of its ancestor, i.e. a subset It is actually from here that the concept sub originated, the sub-class instances represent a subset of the super-class instances.

Bertrand Meyer (1988) has also observed this problem He proposes that inheritance can be regarded as both an extension and a specialization (restriction) If we consider the class to be a definition of operations with an information structure, then inheritance is an extension If we consider the class to be an implementation of a type, then inheritance can be used to specialize the type, as the instances of descendants represent a sub-collection of all instances of the ancestor.

3.5.1 Using inheritance

An inheritance hierarchy can consist of many levels, that is, classes can be inherited from other classes which, in turn, can be inherited from new classes, and so on How then should you use the inheritance to get a good and robust inheritance hierarchy? We will discuss here some topics from our experience to explain how to get a good inheritance structure.

A class structuring occurs with the help of inheritance hierarchies. This structuring of classes in an inheritance hierarchy enables us to work with classes and to define new classes by specifying the differences between the new ones and the already existing ones If this expansion becomes extensive, then the inheritance hierarchy may become less suitable and a restructuring may be necessary.

For instance, what happens if we add two further individuals to the example, James and Lawrence? They are both men and can therefore be instances of Male. Such an addition is trivial, but what if we want to describe them as old men having a few characteristics differing from those of their younger colleagues? For instance, they cannot dance in the same way as the younger men. Their dance style is a little more rigid To describe the older men, we can add a new class: Old Male. How will this new class be related to the existing ones? It is clear that the class Old Male has all the characteristics of the class Person. It also has a lot in common with Male. Basically we have three relevant possibilities, see Figure 3.16.

(1) Old Male is a descendant of Person and we define the differences between Old Male and Person;

(2) Old Male inherits Male and we define the differences; we make, therefore, a redefinition on dance;

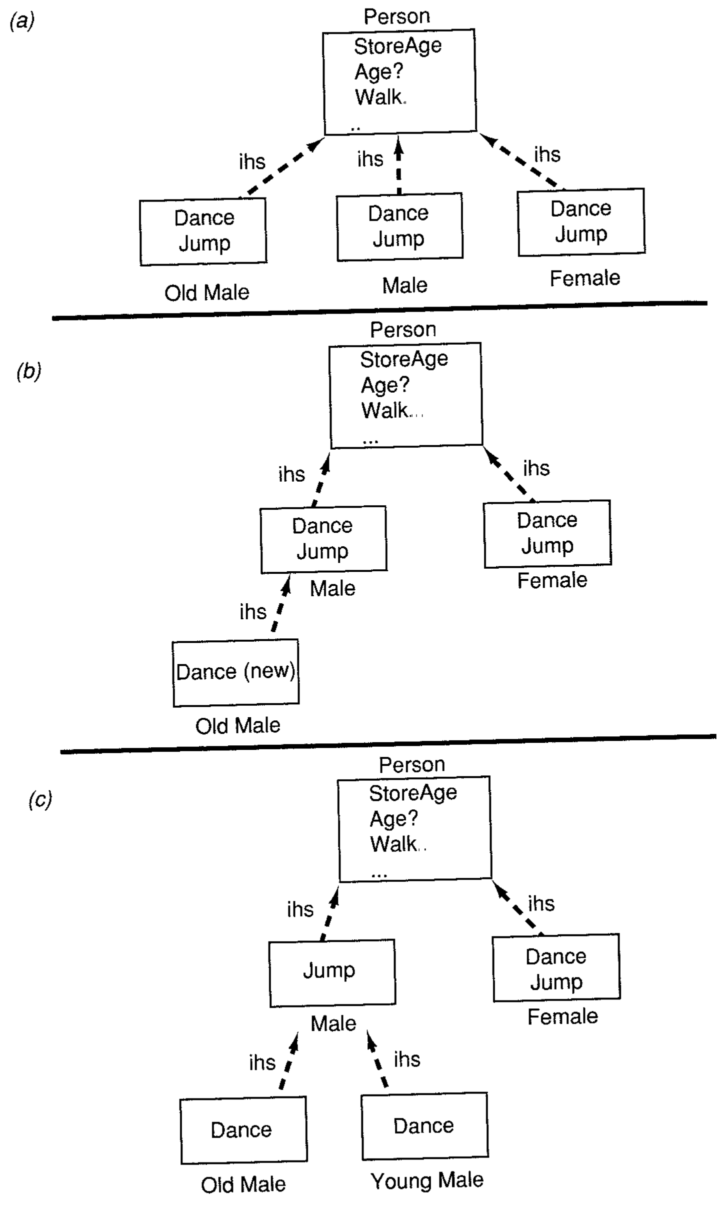


Figure 3.16 Three possibilities to add a new class Old Male in the inheritance hierarchy

(3) We create a further class, Young Male, and define dance even there. In this way, both Old Male and Young Male can inherit Male.

Which of these three alternatives should be selected? This will depend on how the classes are to be used How will further modifications be handled? What do we really want to model? How important is a clear understanding? How difficult is it to restructure the classes? What is the price for restructuring? The best alternative will depend on how you answer these questions.

Ideally, for understanding purposes, we would select the third alternative: to create a further class, Young Male, extract the similarities and place these in class Male With this, we now have no redefinition, and we keep a clear and understandable structure that can be further developed. The cost has been the work involved in restructuring the existing class structure and creating the new class

When a new class is to be added, we try to find a suitable candidate to be the direct ancestor. As we have seen, some operations may need to be modified for the inheritance Normally we have four possibilities to add a new class:

(1) We can go upwards in the inheritance hierarchy to see whether an ancestor is more suited for inheritance (the characteristics that we wish to avoid may not exist in the ancestor),

(2) We can describe the class from basics independent of any other class (i.e. without inheritance),

(3) We can restructure the inheritance hierarchy, so that we obtain a class that suits the inheritance required, see Figure 3.17 for an example. However this is not always possible.

(4) We can redefine the characteristics that we wish to change.

The first two solutions are trivial and we shall therefore not discuss them any further. The third solution is the most acceptable, as it maintains a clean class hierarchy. To restructure in this way, though, often requires a lot of work, partly due to the work involved in finding a better structure and partly due to the consequences that the modification will have on a system designed on the basis of an already existing class hierarchy. This solution is usually not as easy as shown in the figure and thus compromises may need to be made in order to find where the operations will most suitably be defined.

The fourth solution is often called **overriding**. It means that we redefine some behavior and/or some information structure from an ancestor. Whether overriding should be used, or not, is widely discussed in the object-oriented community Overriding is both easy and flexible to use for modifying existing classes, but it can damage understanding of a class hierarchy, as operations with the same name

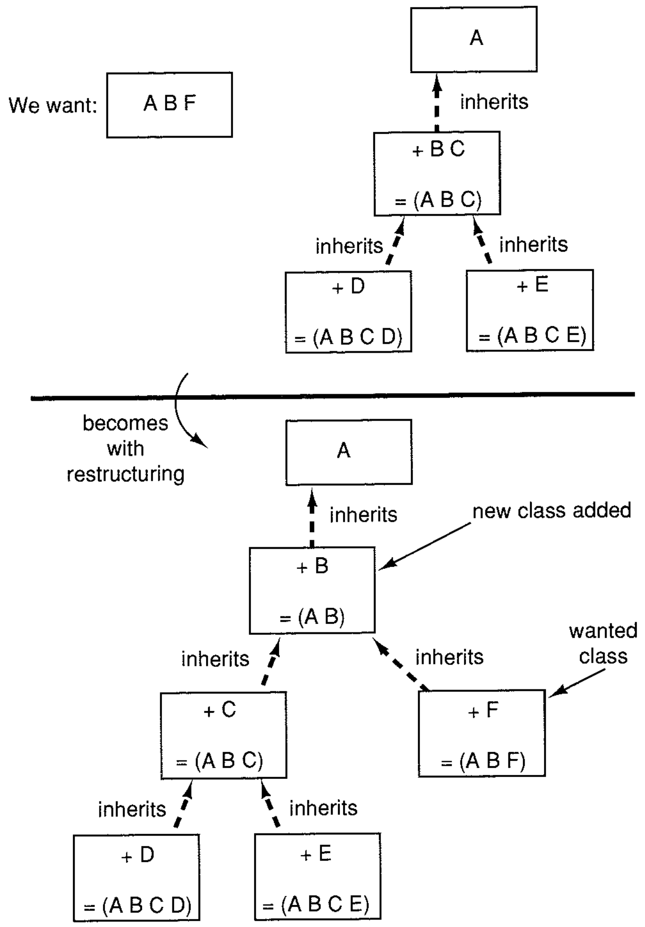


Figure 3.17 By restructuring an inheritance hierarchy, one can, in most cases, obtain the required classes. The letters in the boxes means that the classes has the characteristics represented by these letters which can either relate to behavior or information structure (ABC) means that the class has these properties.

can have different semantical meanings in different classes. An inheritance chain must then be followed upwards to the place where the operation is defined to find its definition. After the inheritance has passed from one class to another, the inherited characteristics can be modified. With override, therefore, inheritance is not transitive. Thus the descendant has not inherited all the ancestor's characteristics; only some of them have been inherited, while others have been redefined.

To understand better how we should use inheritance in a proper way we shall review the main purposes of inheritance.

Reuse. The most common reason for using inheritance is that it simplifies the re-use of code. Reuse can, in principle, occur in two different ways in combination with inheritance. The first is that two classes are found to have similar parts, these parts are extracted and placed in an abstract class, which they both inherit. This abstract class represents the common parts of both classes and need not always be meaningful in itself. The other way of reuse is to commence from a class library. Find a class which has the operations that you need. Inherit this class and make the required modifications.

Sub-typing. A class can be regarded as an implementation of a type. A class A defines a certain behavior. If it is possible to use one of A's descendants in all the places where class A is used, then we say that the classes are **behaviorally compatible**. The descendant represents a sub-type to class A. In practice this often means that the descendant should at least have the same interface as its ancestors. We can say that the ancestor here represents a subset of the behavior common to all the descendants. Sub-typing normally occurs if the inheritance performs only an extension, rather than overriding something already defined in the ancestor. The additions must be disjunct with the existing classes, and therefore must not create any restrictions on them. This use of inheritance is tightly coupled to describing the **role** played by an object; the role is defined by the operations of the object. To think in terms of roles or responsibilities is often appropriate for modeling object-oriented systems and provides a good tool.

Specialization. If the descendant is modified in such a way that it is no longer behaviorally compatible with its parent (i.e. if the parent class can no longer be exchanged with the descendant), the class is said to be specialized. Normally, the operations and/or information structure have been redefined or deleted. A parent class that has been specialised cannot always be replaced by its descendant. An example of specialization is if a class Adult inherits from another class Person, see Figure 3.18. The class Adult has a more restricted age interval than class Person. We cannot, therefore, arbitrarily replace the class Adult with the class Person. Consider the case when one wishes to enter a person's age of 5. This is not possible for instances of the class Adult, but is possible for instances of the class Person.

Conceptual. This use of inheritance corresponds closely to the intuitive semantic, 'a dog is a mammal' (i.e. in all places where we use mammal, we can even use dog, as dog maintains all the characteristics of mammal).

These different ways of using inheritance are not exclusive or disjunct in any way. For example, reuse can be a reason for sub-typing. In order to optimize the use of inheritance and be able to

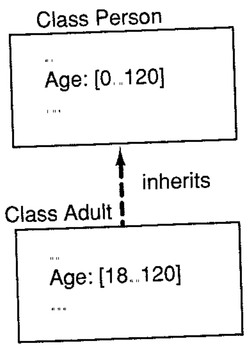


Figure 3.18 Inheritance as specialization. The class Person has defined age to be in the range 0 to 120 years. The class Adult restricts this to the range 18 to 120 years. The class Person is now no longer exchangeable with the class Adult, as one cannot specify an age of, for instance, 5 years for the class Adult.

recognize both good and bad solutions, more experience must be gained and more research done on the use of inheritance. However, from our experience, focusing on an objects protocol normally leads to appropriate inheritance structures. This means that focusing on the sub-typing issue at all times will lead to proper inheritance hierachies that are maintainable and robust. This usage is also natural from a logical perspective. What should be avoided is the use of inheritance for reuse of single operations, that is, even if you can only reuse a single operation, you inherit the class anyway. Although this may be proper and efficient in prototyping purposes, such use of inheritance often leads to code that is not maintainable or robust in the long run. We may talk about this type of use as *spaghetti inheritance*.

The use of inheritance is discussed by, for example, LaLonde and Pugh (1991b), Wegner and Zdonik (1988) and LaLonde, Thomas and Pugh (1986).

3.5.2 Multiple inheritance

When describing a new class, if we wish to use characteristics from two or more existing classes, we can inherit both these classes. We call this **multiple inheritance**. This means that one class can have more than one direct ancestor. The use of multiple inheritance is, as with overriding, controversial in the OO community.

Multiple inheritance can however be justified if we regard inheritance as a way to model roles played by an object. If we regard

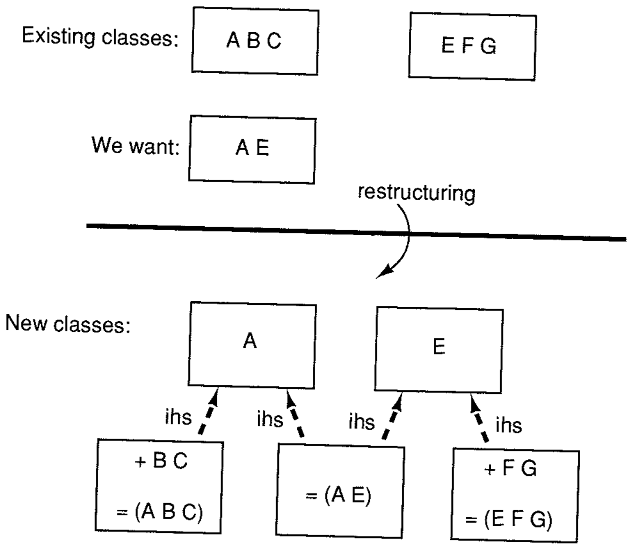


Figure 3.19 Multiple inheritance. The middle class inherits the descriptions from two other classes

inheritance as a way to structure roles, we can, for example, construct a houseboat by inheriting both a house and a boat since a houseboat can play both roles, of being a house and a boat. Critics of multiple inheritance mean that we could just as easily have obtained a boathouse with this use of inheritance, but here we apparently see the use of roles in this context; a boathouse cannot play the role of a boat.

Multiple inheritance is also often justified as a way to provide reusable descriptions. If a description of a class includes behavior which partly exists in two other classes, then this behavior can be extracted so that we can inherit the common behavior from two classes, see Figure 3 19. We wish to create a new class containing the behaviors A and E. The behaviors A and E are integrated with other behaviors in separate classes. By means of extracting A and E and creating individual classes for them, we can inherit these classes and, in this way, share their descriptions.

This use of multiple inheritance is often the main target for the critics of multiple inheritance As was said above, this usage, if not carried out properly, may very well lead to spaghetti inheritance. If it is hard to understand such inheritance in single inheritance contexts, it is far harder in multiple inheritance contexts. The above example, could be proper, but then the roles described by the classes should be viewed instead. We thus see that to use multiple inheritance

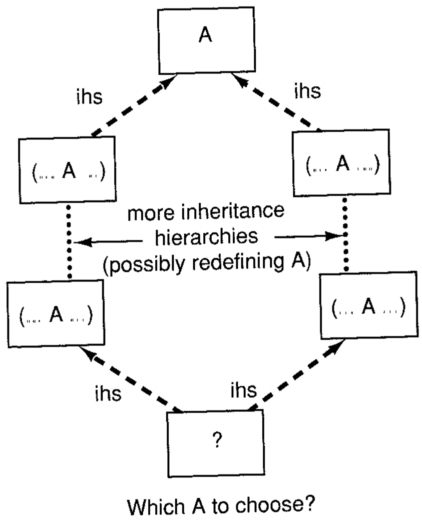


Figure 3.20 Repeated inheritance. The same characteristic exists in two parents How is this detected and which should be chosen?

properly, the sub-typing rule is not in all cases sufficient. In the example the descendants seem to be fine sub-types of the ancestors. It all depends on whether the roles are conflicting Hence we see here that the use of roles provides great support for the development of the inheritance hierarchy.

The main disadvantage with multiple inheritance is that it often reduces understanding of a class hierarchy. The problem arises especially if both ancestors have an operation with the same name, but with different definitions. We must then either define a new operation or explicitly select one of the existing definitions In the majority of programming languages that support multiple inheritance, the user is forced to redefine the name so that it becomes unique. This is the most acceptable solution, as only the user has sufficient knowledge to solve this conflict. A special case is the case in which the same operation exists in both parent classes since these classes may have a common ancestor, see Figure 3 20. This is called **repeated inheritance** and must be solved. In the case of the same operation appearing in two places, the problem is actually solved once this is realized (i e. the selection of which to use doesn't matter). If, though, it was originally the same operation, but has been overridden, then the problem is to know which one to select.

3.6 Summary

Object-orientation is a technique by which to develop models. It maps very well the way people think about reality. Therefore object-oriented models are often easy to understand. Additionally, modifications are often local since changes often evolve from some individual item that is modelled.

An object is a representation of some entity with both behavior and information tied to it. Only the operations that can be performed on the object are seen from the outside of the object. Objects are related and the overall dynamics of the model arises when the objects start to interact by sending stimuli to each other. However, supporting objects is not sufficient to be object-oriented

Many objects may be similar. A class describes these similarities of objects. Every object is then an instance of the class and the behavior and information structure of the instance are defined by its class. To make the description of classes more flexible, polymorphism is used. This means that, when defining relations between objects, one object does not need to know the exact class of the other object. In this way new classes can be introduced without needing to change other classes.

Classes can be described as changes to existing classes with inheritance. This makes the definition of new classes easy as only differences need to be described. However, inheritance is a very strong tool when properly used, but when improperly used it can lead to models that are hard to understand and maintain. As single inheritance is a strong tool, multiple inheriance is a much stronger tool Improperly used, though, multiple inheritance can be very much worse than single inheritance.

4 Object-oriented system development

4.1 Introduction

In order to design large systems, a systematic approach should be adopted. There is an abundance of such approaches and all of them are aimed at producing a good system. But what is a good system? This question can be answered partly from an external viewpoint and partly from an internal viewpoint of the system. The external viewpoint is that of all those who in some way use the system. They want the system to give correct results quickly, to be reliable, easy to learn and use, effective, and so on. The internal viewpoint is that of the system developers and those who have to maintain the system. They want the system to be easy to modify and make additions to, easy to understand, contain reusable parts, be easy to test, compatible with other systems, portable, powerful and easy to manufacture.

The definition of a good system varies though in certain respects between different applications. In some, it is performance that is important and in others perhaps user-friendliness. It can, in fact, depend on the structure of the system, for instance whether it is distributed or centralized. What is common to all (larger) systems is that they will need to be modified.

A system's entropy

The second law of thermodynamics, in principle, states that a closed system's disorder cannot be reduced, it can only increase or possibly remain unchanged. A measure of this disorder is **entropy**. This law also seems plausible for software systems and we can assume that this law is plausible for the systems discussed here; a system's disorder, or entropy, always increases. We can call this **software entropy**.

Within software development, there are similar theories, see Lehman (1985), who suggested a number of laws, where two of them were, basically, as follows:

(1) A program that is used will be modified,

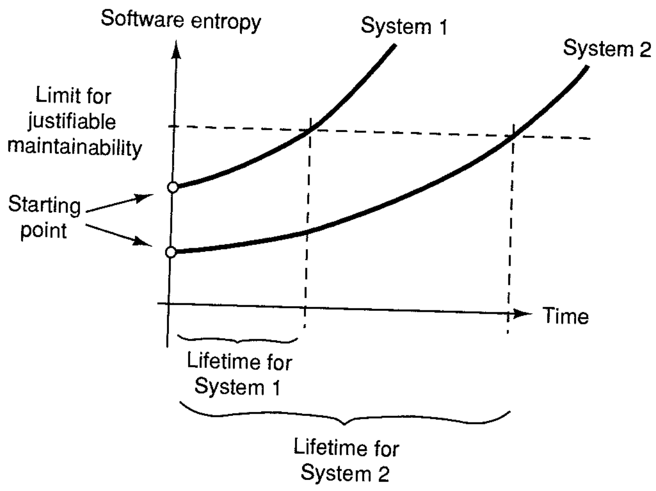


Figure 4.1 A system's entropy and how it increases at different speeds dependent on the starting entropy

(2) When a program is modified, its complexity will increase, provided that one does not actively work against this

Assume that a system initially has a certain software entropy. Experience shows that it is reasonable to assume that the increase in software entropy is proportional to the entropy of the software when the modification started This means that it is easier to change an ordered system than a disordered one, something that all experience shows This would mathematically be expressed as

$$\Delta E \sim E$$

or, with differential calculus

$$\frac{dE}{dt} = kE$$

which is a simple differential equation having solutions as shown in Figure 4 1 We can see from the figure that a system's lifespan is dependent on how well structured the system was initially. When a certain software entropy is reached, it is no longer economically justifiable to continue with this system, as it has become unreasonably expensive to modify. One possibility is that, at such a stage, we apply re-engineering to reduce this software entropy, so that we can continue to maintain the system at a reasonable cost

When we design a system with the intention of it being maintainable, we try to give it the lowest software entropy possible from the beginning. This is one of the aims of a system development method By means of this, we hope to increase the lifespan of the system. We know, though, that sooner or later we shall, nevertheless, reach a limit beyond which it is too expensive to maintain the system.

There is much to learn from these kinds of analogies, but analogies should only be used as idea givers. Even if there are similarities to system development, we are not yet experts in software entropy. The main difficulty with this kind of 'law' is, of course, that people are involved.

In order to design a good system, different system development methods have been proposed to describe either a project for development of a first product verson or a global view of the entire system life cycle Traditionally, the work is structured and described using different types of **waterfall models**, see Figure 4.2. These waterfalls describe the flow of the development process. The work begins by creating a requirement specification for the system. This is normally performed by the person who ordered the system or those developing it in cooperation with the orderers From this requirement specification, an analysis and logical description of the system is made. Alternatively, this can be produced together with the requirement specification. The design of the system is then done and followed by implementation in smaller modules. These modules are first tested individually and then together. When the last integration test has been completed, the entire system can be tested and delivered and the maintenance phase begins. Initially, the idea was that one should complete one phase before the next one is

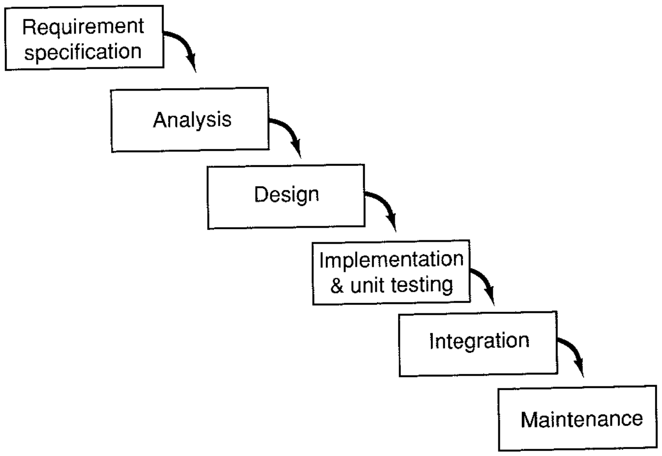


Figure 4.2 A typical waterfall model

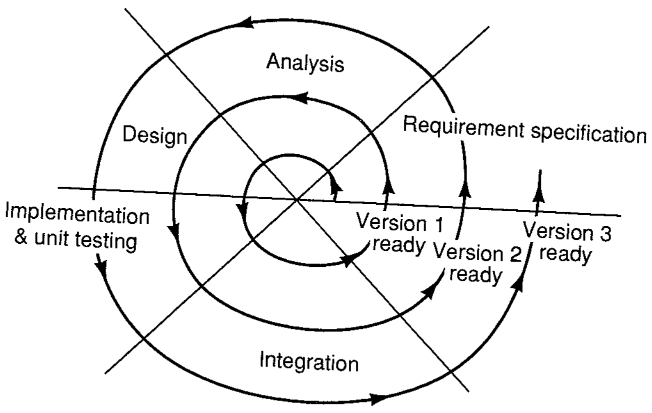


Figure 4.3 A spiral model

started. This principle, though, was abolished relatively quickly and thus a phase was allowed to be started before the previous one was totally complete. The waterfall model has had a tremendous affect on software engineering methods, although it was never intended to be taken so literally when first introduced.

Not long after, it was discovered that water must fall upwards, so to speak, in order to fully describe the development cycle The major problem, though, became, in more and more cases, the maintenance phase, which in reality contained a new requirement specification, analysis, design and so on. Various other models were developed in order to describe these new facts, and one of the more popular is the **spiral model**, see Boehm (1986), shown in Figure 4.3. The spiral model can describe how a product develops to form new versions, and how a version can be incrementally developed from prototype to completed product.

System development usually contains all of these phases (even if they are given different names in different methods), and development always occurs incrementally over them. This development scenario is true irrespective of which method is used. We can characterize system development as being initially rather turbulent, but stabilizing subsequently. The development method used should therefore help to make the development process stable as soon as possible. The idea is that we should work on analysis sufficiently long to understand the system totally, but not so long as to consider details which will be modified during design. This often means that a relatively large part of the work performed is carried out during the analysis phase A typical time division for the projects we have been involved in is shown in Figure 4.4. Initially, a small group of

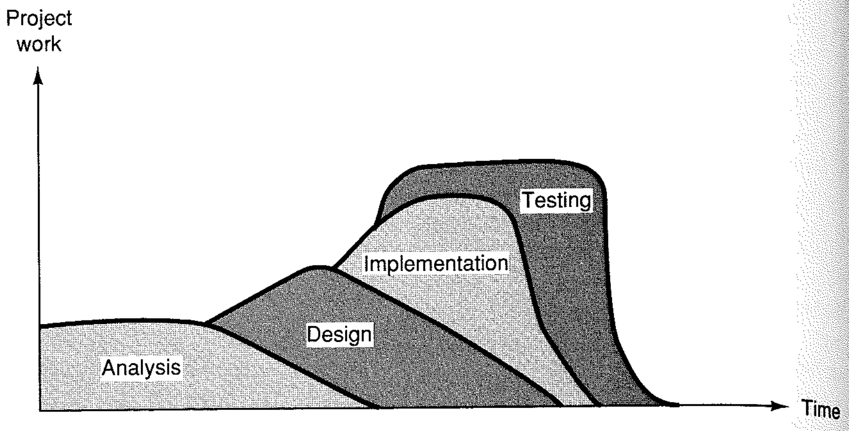


Figure 4.4 Total division of efforts over time between different activities.

people performs analysis and subsequently design. These activities are worked on iteratively. As the system structure stabilizes, more people are involved in implementation and testing. However, analysis and design activities may also be done even when testing is started. At this stage it is mainly changes in the analysis or design models that are incorporated.

4.2 Function/data methods

The existing methods for system development can basically be divided into function/data methods and object-oriented methods. By **function/data** methods we mean those methods that focus on functions and/or data as more or less separated. Object-oriented methods view function and data as highly integrated. These paradigms are shown schematically in Figure 4.5. Most traditional software engineering methods, such as SADT (Structured Analysis and Design Technique), see Ross (1985), RDD (Requirement Driven Design based on SREM), see Alford (1985) and SA/SD (Structured Analysis and Structured Design), see Yourdon and Constantine (1979) and Yourdon (1989) are function/data methods.

Function/Data methods thus distinguish between functions and data, where functions, in principle, are active and have behavior, and data are passive holders of information which become affected by functions. The system is typically broken down into functions, whereas data is sent between those functions. These functions are broken down further and eventually even to source code. This function/data division originates from von Neumann's hardware

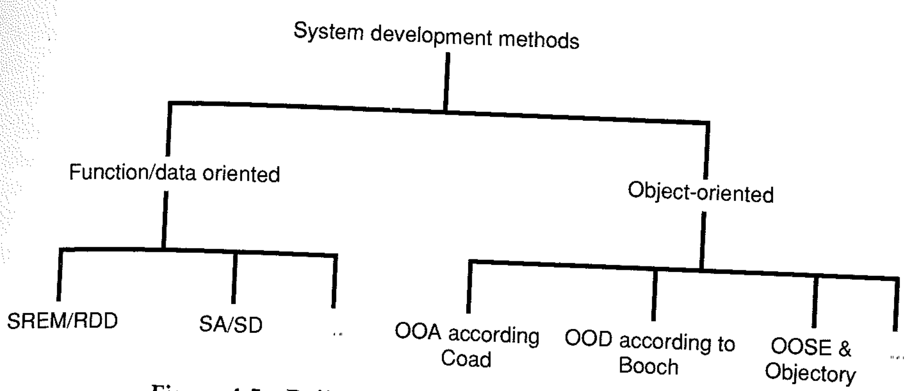


Figure 4.5 Different software development paradigms

architecture, where the separation of program and data is greatly emphasized. The first machine language also had this division in its instruction set-up and it was thereafter logical to give high-level languages a similar structure. This approach has been difficult to escape from, but there have been several different attempts to do so. One such attempt is object-oriented programming, another is functional programming, see Backus (1977) or Bird and Wadler (1988).

A system developed using a function/data method often becomes difficult to maintain. A major problem with function/data methods is that, in principle, all functions must know how the data is stored, that is, its data structure. It is often the case that different types of data have slightly different data formats, which means that we often need condition clauses to verify the data type. The programs, therefore, often need to have either IF-THEN or CASE structures, which really do not have anything to do with the functionality, but relate only to the different data formats. The programs therefore become difficult to read, as one is often not interested in data format, but only in the functionality. Furthermore, if one wishes to change a data structure, then one must modify all the functions relating to the structure. Therefore systems developed using such methods often become quite unstable; a slight modification will generate awful consequences. In the same way, the instability of the development process, which we discussed earlier in this chapter, will become unnecessarily lasting, as any modifications made will create consequences in other areas.

Another problem with function/data methods is that one does not naturally think in the terms one structures in. The requirement specification is normally formulated in normal human language. It often describes in What-terms what the system shall perform, what functionality the system shall support and what items should exist in the system. This is often reformulated into How-terms for the functional break down when the focus changes. In this way we obtain a large semantic gap between the external and internal views of the system Note that nothing in the methods explicitly tells you to change focus, but it is often a consequence when the developers change focus to the implementation. We shall see that, with an object-oriented method, the system will even be internally structured from the model taken from the requirement specification.

This is yet another reason that function/data systems become more difficult to modify. Owing to the fact that they are designed around how a certain behavior shall be carried out (this being a common area of modification), modifications often generate major consequences. Object-oriented methods structure the system from the items which exist in the problem domain This is often a more natural way to describe the system. These items are normally very stable and change very little. The changes which do occur, normally affect only one or a few such items, which means that the changes made are usually local in the system. If we wish to have a stable system, we should consider what has a tendency to change, and then design according to this knowledge. Peter Coad and Ed Yourdon (1991) inspired us to propose the following table:

Table 4.1 Tendency for change in various items

Item	*Probability for change*
Object from application	Low
Long-lived information structures	Low
Passive object's attribute	Medium
Sequences of behavior	Medium
Interface with outside world	High
Functionality	High

However, this topic has not scientifically been investigated. Most object-oriented methods base their structure on the items that have a low probability of change. This can, though, lead to troublesome consequences, as, for example, Constantine (1990) has observed. We have also noted this and the approach we describe later in the book are focused to handle this. The aim of our approach is to manage *all* changes in a controlled and preferably local way. This especially means that the items with a high probability for change must be managed by the method as well.

It has been claimed that it is possible to use a function/data method initially and later, usually in the design or programming phase, change to an object-oriented point of view. However, this becomes unnatural as the division of function and data is a central idea all the way down to the programming level. To start joining program and data, at any stage, to design your objects and classes becomes awkward, as the method is based on separating them. Thus, before programming commences, a paradigmatic shift will occur, making it difficult to use all the object-oriented programming qualities in full. Object-oriented programming requires a different approach than that of the function/data development methods.

The shift in approach means that the decision on which object is fundamental for the system structure is made too late in the development process A central opinion in connection with object-oriented programming is that important items from the application domain will be represented as objects. If one uses a strict functional analysis method in combination with an object-oriented programming language, the object will not be identified until the design stage, by people who have not had contact with either the orderer or the final user. This in turn suggests that the objects will in all probability not represent all the important items from the application domain. Thus one has reformulated items which the system shall work with into How-terms in order to once again reformulate these 'Hows' as objects.

We shall now discuss, from an object-oriented point of view, the different system development phases. We assume that there is an existing requirement specification in some form. This is analyzed with the aim of finding which objects shall exist in the system. The object models obtained are designed when we perform design and implementation. Finally, we will briefly discuss how testing is carried out

4.3 Object-oriented analysis

The purpose of object-oriented analysis, as with all other analysis, is to obtain an understanding of the application; an understanding depending only on the system's functional requirements.

The difference between object-oriented analysis and function/data analysis is, as previously mentioned, the means of expression. While function/data analysis methods commence by considering the system's behavior and/or data regarded separately, object-oriented analysis combines them and regards them as integrated objects. Object-oriented analysis can be characterized as an iteration between analyzing the behavior and information of the system. Moreover, object-oriented analysis uses the object-oriented techniques introduced in the previous chapter.

Object-oriented analysis contains, in some order, the following activity:

- Finding the objects,
- Organizing the objects,
- Describing how the objects interact,
- Defining the operations of the objects,
- Defining the objects internally.

4.3.1 Finding the objects

The object can be found as naturally occurring entities in the application domain. The object becomes typically a noun which exists in the domain and, because of this, it is often a good start to learn the terminology for the problem domain. By means of learning what is relevant in the application domain, the objects will be found. It is often the case that there is no problem in finding objects, the difficulty is usually in selecting those objects relevant to the system. The aim is to find the essential objects, which are to remain essential throughout the system's life cycle. As they are essential, they will probably always exist and, in this way, we hope to obtain a stable system. Stability also depends on the fact that modifications often begin from some of these items and therefore are local. For example, in an application for controlling a water tank, typical objects would be Contained Water, Regulator, Valve, Tank and suchlike and for a banking application typical objects would be Customer, Account, Bank, Clerk and suchlike.

The majority of object-oriented methods today have only one type of object. In this way, one obtains a simple and general model. Yet there are reasons for having several different object types: with only one object type, it can be quite difficult to see the difference between different objects. By means of having different object types, one can more quickly obtain an overview of the system. One can also obtain more support in improving the system's structure by means of having different rules for different objects. An example of this is that a passive object containing persistent information should not be dependent on objects which deal with the interface, as modifications in the interface are very common.

Different object types can be organised from different criteria Some examples are to group them after characteristics, such as active/passive, physical/conceptual, temporary/permanent/persistent, part/whole, generic/specific, private/public, shared/non-shared.

4.3.2 Organizing the objects

There are a number of criteria to use for the classification and organization of objects and classes of objects. One classification originates by considering how similar the classes of objects are to each other This is normally the basis of inheritance hierarchy; a class can inherit another class. Another classification can be made by considering which objects work together with which other objects or how an object is a part of another; for example a house can be built of doors and windows A similar classification is to see which objects are in some way dependent on another and thus having modification as a basis of grouping and perhaps even structure according to whole subsystems.

4.3.3 Object interaction

In order to obtain a picture of how the object fits into the system, we can describe different scenarios or use cases in which the object takes part and communicates with other objects In this way, we can fully describe the object's surroundings and what the other objects expect from our object The object's interface can be decided from these scenarios. We then also consider how certain objects are part of other objects.

4.3.4 Operations on objects

The object's operations come naturally when we consider an object's interface. The operations can also be identified directly from the application, when we consider what can be done with the items we model They can be primitive (e g create, add, delete) or more complex such as putting together some report of information from several objects. If one obtains very complex operations, new objects can be identified from them. Generally, it is better to avoid objects that are too complex.

4.3.5 Object implementation

Finally, the object should be defined internally, which includes defining the information that each object must hold. Even the number of instances that can be created of each object is interesting; one may wish to have alternative ways of storing information. Some of the attributes can be inherited.

To summarize the above points we can note that all of these steps are dependent on other steps, and it is typical in a development that these steps are worked with iteratively. It is important in an analysis phase to concentrate on understanding the problem domain, as the result here will affect the whole of the remaining work. Experience has shown that objects identified from the application domain are very stable and, after some work has been performed, both work and documentation will revolve around these objects.

We mentioned earlier that one of the advantages with object-oriented analysis is that it reduces the semantic distance between the domain and the model. This is due to the fact that object-oriented analysis bases the system structure on the human way of looking at reality, namely the objects, classification and hierarchical understanding used when people understand their surroundings. Thus the result from the object-oriented analysis is easier to understand and thus also easier to maintain than the result from a function/data analysis.

Another advantage of object-oriented analysis is that items with a low modification probability are naturally identified and it is possible to isolate at an early stage items with a high probability of modification.

We shall see later how OOSE contains all of the above mentioned parts, in a slightly different order. Several of the issues discussed here are treated in OOSE in different ways. We shall postpone the discussion of OOSE analysis until a later chapter.

4.4 Object-oriented construction

Object-oriented construction means that the analysis model is designed and implemented in source code. This source code is executed in the target environment, which often means that the ideal model produced by the analysis model must be molded to fit into the implementation environment.

Exactly as with all design activities, it is difficult to achieve a good balance between structure and efficiency. The analysis model has provided us with an ideal structure which we shall try to keep as long as possible. It is, as you know, modification resistant. During the design, though, one must take care to follow all restrictive demands on the system, (e.g. demands of the target environment, maximum memory usage, reliability and response times). All this can affect the structure.

The goal is that the objects identified during the analysis should also be found within the design. We call this **traceability**. We must therefore have straightforward rules for how to transform the analysis model into a design model and the programming language.

The objects may be implemented using previously developed source codes. We call such parts **components**. Such components are often simpler to create in an object-oriented environment, due to the integration of functions and data.

Previously, we mentioned that object-oriented programming demands object-oriented system development as, otherwise, there will occur a difficult paradigm shift. The opposite procedure, to go from object-oriented analysis to traditional programming, also incurs a paradigm shift. However, this causes less of a problem, as even for non-object-oriented languages, we can structure the system to be object-oriented. Additionally, we have the possibility of programming with an object-oriented style, even for other languages, see Jacky and Kalet (1987) or Linowes (1988). Hence object-orientation is also a style of programming and not only the name of a programming language family having language constructs such as inheritance, encapsulation and polymorphism. A paradigm shift will occur, but it can be made relatively smooth. We shall later see that if we have for example a relational data base, which is typically function/data oriented, we can incorporate this inside the objects and perform the paradigm shift locally.

Of course, many good techniques have been developed in function/data methods, which can also be used in object-oriented methods. Just because we shift paradigm, we should not forget all that we have learned. An example of a technique usable within object-orientation is state diagrams. In the chapter on construction, we shall see how these can be used to describe an object.

4.5 Object-oriented testing

The testing of a system which has been developed with an object-oriented method does not differ considerably from the testing of a system been developed by any other method. In both cases, we verify the system, namely check that we have correctly designed the system in accordance with some specifiction. This verification should start as early as possible. The program testing begins at the lowest level with unit testing, and progresses to integration testing, where the units are tested together to see that they interact correctly. Finally, testing of the entire system is done.

Traditionally, integration testing is usually a 'big bang' event and is very critical during system development It is at this stage that the developed parts are put together and we then see whether they actually work together. Such a 'big bang' event is, though, not as dramatic in an object-oriented system.

An object-oriented system consists of a number of objects which communicate with each other. These objects contain both data and behavior, which makes them larger units than one works with (individual routines) in a traditional system development method. An object's operations are developed around specific data and the same designers normally develop all operations for an object (see, nevertheless, the later discussion on inheritance!). This leads to unit testing (i.e. testing on the lowest level) becoming a test of a larger unit than in a traditionally developed system. Integration testing, however, is carried out at an early stage, since communication is essential for the system development. All objects have predefined interfaces which also contribute to a less dramatic integration testing. The integration testing continues on higher levels and more and more objects are put together incrementally.

However, inheritance between classes can create new difficulties with testing. Inheritance means, as you know, that the operations defined in one class are inherited by another class and can also be executed there. There can therefore arise abstract classes of which will never be instances, but they only contain common parts from other classes. Is it, therefore, worth testing these abstract classes? Normally it is worthwhile, since you have then tested an operation once instead of testing it several times in all descendants. However, one must be aware of how the classes are to be used in order to be able to test them properly. Hence an operation in an abstract class may use properties that are changed in the descendants. So when the context of an operation is changed, the operation normally needs to be retested in the new context.

Testing of inheritance hierarchies thus requires a more exhaustive testing method, where one is aware of how the system will appear in operation. For instance, as we have pointed out, it is not always true that if one has tested an operation higher up in an inheritance hierarchy a test is not required lower down. An operation can find itself in another (new) environment and may not have been tested in that environment. This means that if we modify an operation in an ancestor, we may even need to test this operation in the descendants. In the same way, if we add a new descendant, we may

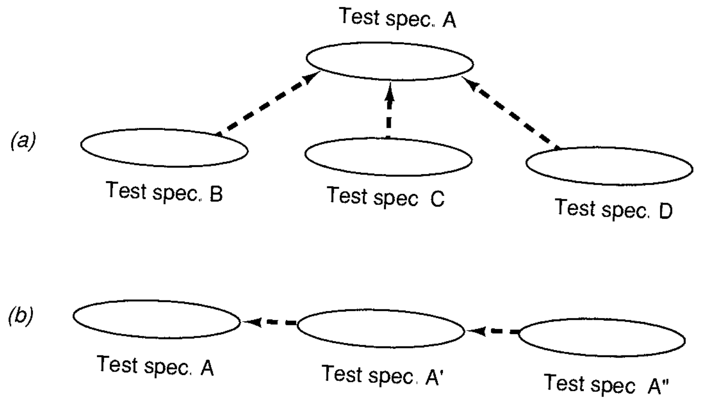


Figure 4.6 Different ways to use inheritance between test specifications. In (*a*), inheritance is used to describe similarities between different test specifications. In (*b*), inheritance is used to describe the incremental development of a test specification

need to test all the operations inherited by this descendant, as they will find themselves in a new environment. Regression and automated testing therefore often play a larger role in the testing of systems developed using an object-oriented technique. Here one must, though, be careful with which test data is used. If one overrides operations in a descendant, the test data developed earlier may not be adequate for the new operation. As test data often starts by testing possible ways to use an operation, it is not certain that the old test data is able to test the new operation. In extreme cases we may therefore need to develop special test data for each level in the inheritance hierarchy. Other problems in testing object-oriented software includes polymorphism. We shall discuss OO-testing in greater detail later.

Another problem arises if the object sends stimuli to itself. Such stimuli, perhaps, are only relevant in a descendant and it is therefore not worthwhile to test the abstract class in this situation. It requires a lot of work to go up and down in a class hierarchy in order to understand the consequences of a message which is sent to the object itself. This problem is called the Yoyo problem, see Taenzer *et al* (1989).

When one has an object-oriented approach, one can more or less consider a test as an object. The test thus has a class (the test specification) and one or more instances (when one carries out the test). The test is encapsulated, since we are only concerned with the test result, and not with how it runs internally. We can even inherit common parts of the test specification when we either write the test specification or further develop a test, see Figure 4 6. In the first case, similar parts of test specifications, such as start up sequences, can be reused. In the latter case a test specification can evolve in different versions for different versions of the system.

4.6 Summary

The development of qualitative software systems should be done using some development method. Different methods focus on different quality properties. Additionally, different models have been proposed to describe the development, the most common by far being the waterfall model. Most developments however, have an iterative nature.

Traditional methods focus on function and/or data as separated. Such an approach often leads to problems during maintenance since a function/data structure is quite sensitive to changes. Object-oriented methods do not separate functions and data, but view them as an integrated whole.

Object-oriented analysis aims at understanding the system to be developed and building a logical model of the system. This model is based on natural objects found in the problem domain. The objects hold data and have behavior in terms of which the entire system behavior can be expressed. Since objects in the problem domain will be stable, the overall structure of the system will normally be quite stable. However, changes will of course occur, but since changes often come from the domain, it is hoped that these changes should be local to as few objects as possible

The design and implementation of the analysis model is straightforward. The aim is to keep the logical structure of the analysis model in the final system. Thus an important characteristic of object-oriented development is to have built-in traceability. An object identified during analysis must be found again in the code so that the system is durable to easy modifications.

Testing of object-oriented systems does not substantially differ from testing other kinds of systems. The main differences are that the integration testing does not come as a 'big bang' event, but is done throughout the development. Although inheritance often leads to less code, it must not also lead to less testing, quite the opposite. Inheritance hierarchies may lead to a more exhaustive testing whereby each inherited operation must be tested in each descendant class.

5 Object-oriented programming

5.1 Introduction

Object-oriented programming essentially means programming using objects and the other concepts introduced in Chapter 3. In this chapter, we shall describe how the concepts are used in a programming language and how they can be implemented. This style of programming has been used, until recently, only on a small scale. The major strength of object-oriented programming is that it encourages the reuse of code and that it is usually easier to understand and maintain than other types of programming.

Defining what is and what is not object-oriented programming is a difficult task. Is it a style of programming or is it language dependent (one can write pure C programs in C++!)? We believe it to be, essentially, a style of programming; but to optimize the use of this style of programming, a thorough understanding of the core concepts is required. One also requires a lot of knowledge to be able to use these techniques efficiently. The question of what a good object-oriented programming style is has been discussed We shall not enter into this discussion here, but refer to other sources, such as Rochat (1986), Johnson and Foote (1988) and Lieberherr and Holland (1989).

We have previously introduced the essential concepts in object-orientation. An object-oriented language should, at least, support these concepts, so that the programmer can utilize them to the fullest. Hence an object-oriented language must support the following:

- Encapsulated objects,
- The class and instance concepts,
- Inheritance between classes,
- Polymorphism

We shall describe how an object-oriented language operates by means of lifting its lid and studying its main machinery. The aim

Variable	Value
A	3
B	'atext'
C	nil

Figure 5.1 Example of a Weizenbaum diagram

is to provide an understanding of the different parts of an object-oriented language. We will not present an exact picture, but will content ourselves with providing an outline of the principles. A more precise description of the details can be found in any book that describes a language, for example Goldberg and Robson (1983). We shall not dedicate ourselves to any specific programming language, but will mostly refer to an abstract syntax in our examples and only sometimes to an existing language, usually C++, Eiffel or Smalltalk. Different programming languages have sometimes chosen different solutions; we will discuss some of these when they are of extra interest. Further reading about the languages that we discuss here can be obtained for Smalltalk, Goldberg and Robson (1983) or LaLonde and Pugh (1990, 1991a); Eiffel, Meyer (1988); C++, Lippman (1991) or Ellis and Stroustrup (1990); Ada, Barnes (1982, 1984) or Booch (1987b); Simula, Birtwistle (1979) or Eriksson and Holm (1984); and Objective-C, Cox (1986). Peter Wegner (1987) has made a generally accepted classification of different programming languages in the object-based world. He states in this classification that Ada is object-based because it supports the object concept, but not the class concept. Simula, Eiffel, C++, Smalltalk and Objective-C are examples of object-oriented languages according to his classification since they also support the class and inheritance concepts

To enable us to understand how the object appears, we shall use **environments** and algorithms to understand how these environments should be interpreted. Environments are used a lot within the Lisp world. An environment describes an object's state and behavior. Environments can be illustrated in several ways, such as by a Weizenbaum diagram, see Weizenbaum (1968), or with frames, see Abelson and Sussman (1985). We shall illustrate the environments with diagrams similar to the Weizenbaum diagram. Such diagrams are a description technique in which the majority of programming language computation can be explained. An example is shown in Figure 5.1. By studying these diagrams, we shall examine how the programming language's machinery works.

This chapter's contents follow roughly the contents of the chapter on object-orientation We begin by discussing the object

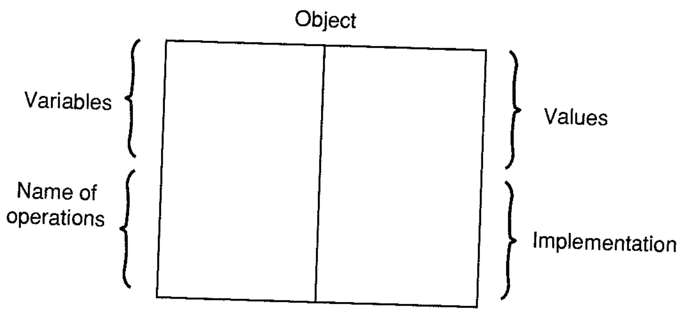


Figure 5.2 A Weizenbaum diagram for an object

concept, and see how this can be solved in a programming language. We then continue to discuss the concepts class and instance. Inheritance is discussed before polymorphism, as inheritance is often used to structure polymorphism in object-oriented languages. Polymorphism and the connection to dynamic binding are then discussed, along with their mutual relationship. We shall use the same example that was used in the chapter on object-orientation to help us in our discussion and, due to this, we shall formalize it.

5.2 Objects

The most important concept that an object-oriented language must support is the object concept. Thus the language must support the definition of a set of operations for the object, namely the object's interface, and an implementation part for the object (which a user of the object should not know about). The object implementation is thus encapsulated and hidden from the user.

Each object has a well-defined **interface** which specifies which stimuli the object can receive, that is, which operations can be performed on the object. Each stimulus received causes an operation to be performed, where the stimulus is interpreted by the receiving object. If one tries to send a stimulus to an object which has no corresponding operation, (i.e. the stimulus is not represented in the object's interface), an error occurs.

An object is implemented internally as a number of variables which store information and a number of operations, or routines, for the object, see Figure 5.2. In Smalltalk, the only way to affect the internal variables is to perform an operation on the object. In, for instance, C++ or Eiffel, it is possible to define whether the user should be able to directly access these variables.

Each object is able to receive a specified number of stimuli. The object interprets this stimulus and performs an operation or, perhaps, directly accesses a variable. In Smalltalk, this stimulus is called a message and the operation executed as a result of receiving a message is called a method. To be able to match stimulus and operation, they must have the same name. In Eiffel, the stimulus is not made explicit, but instead the terminology is that an object performs an operation on another object. If we wanted to send a stimulus (e.g. jump) to an object Tom, it may look as follows:

In Smalltalk:	`tom jump.`
In C++:	`tom->jump();`
In Eiffel:	`tom.jump;`
In Ada:	`jump(tom);`

We can see from the above example that the syntax in the different languages reflects the approach adopted by the program language designer. In Smalltalk, C++ and Eiffel, the object Tom is central and the operation is performed on tom. However, in Ada it is the operation which is central and operates on the object. We shall not describe how the object is internally structured in this section, but shall discuss this when we discuss class and instance, in the next section.

5.3 Classes and instances

In object-oriented languages, each object is described by a **class**. This class is both a module for source code and a type for the class' **instances**. The programming language Ada comes close to this approach with the *package* concept, but the package in Ada is not a type. Inside a package, however, types can be defined with associated operations. These types can in turn be used to create objects, where the package's interface may specify the operations that can be performed on the object.

A class defines the operations that can be performed on an instance. It also defines the variables of the instance. A variable associated with a specific instance is often called an **instance variable**. These instance variables store the instance's state. All object-oriented languages have instance variables, even though they may have a different name. In Eiffel, the variable is called a field and the declaration of it is called an attribute, in accordance with Eiffel's main source of inspiration, Simula. The reason for calling them instance variables, and not just variables, is due to the fact that some languages, such as Smalltalk and, in some senses, C++, have other variables which are only associated with the class. These are called

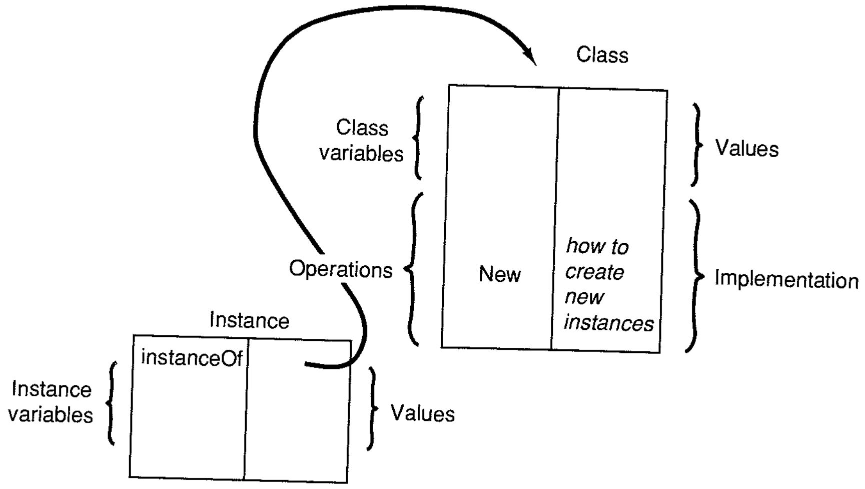


Figure 5.3 Class and instance environment. The class contains the names of operations and their implementation. The instance contains the variables and the values unique to the instance.

class variables. Additionally, variables could also be local to a specific operation. These are called **temporary variables**.

Instances are created from the classes which declare the instance variables. The current values of these variables, though, are stored in the instances. The operations are thus defined in the class. An alternative is that they, as with instance variables, should exist in the instances also. However, this would create much duplication, as the operations are identical for all instances of a specific class. Normally, therefore, each instance has a reference, instanceOf, to the class which contains all the operations, see Figure 5.3. Thus, by means of this reference, each instance is linked to its associated class. In such a way, the instances only store information unique to the individual instance, namely the instance variables, while information common to all instances, namely operations and (possibly) class variables, is stored in the class.

When an instance is to perform an operation its associated class selects the required operation and performs it using the instance's variables. This means that when an operation reads or writes an instance variable, the current instance's variables are used. The operation is selected by means of finding an operation that has the same name as the stimulus (and, possibly, even the correct parameter set-up). The machinery which interprets a stimulus sent to an instance therefore operates according to Algorithm 5.1.

Algorithim 5.1 Operation of the machinery to interpret a stimulus

Given an instance and a stimulus to the instance:
1 In the environment referred to by instanceOf, i e the associated class, the operation corresponding to the stimulus' name is searched for
2 Interpret the selected operation using both the stimulus' parameters and the instance's environment
3 Return any values from the execution of the operation

The algorithm uses the stimulus name to access the required operation from within a table in the class. When a stimulus is sent to an object, an operation is thus performed by applying the algorithm that uses both the stimulus and a specified environment

As the instance's environment only consists of information unique to itself, and all operations are stored in the instance's class, one must be able to perform the same operations on all instances. The difference is in which environments they are executed. Normally, all operations can access all instance variables. The operation environment is thus the current instance's environment, see Figure 5.4.

For an instance to be able to send a stimulus to another instance, variable will have to exist to reference the other instance. Figure 5.5 illustrates an example of how a variable, aPerson, in the instance Mary, refers to another instance, Tom. Thus, in order to send a stimulus Jump to Tom, the instance Mary would need to execute the following:

```
aPerson Jump
```

When the instance referred to by aPerson receives this stimulus, the algorithm outlined above is performed. Thus Tom interprets Jump and performs the behavior associated with this operation.

Encapsulation, and the approach from abstract data types, mean that there is only one way to affect an object's state, and this is through its operations Thus variables can only be directly accessed by defining an appropriate operation. This approach is chosen in Smalltalk, for instance. It is recommended, though, that if an operation is to read a variable, then the operation should have the same name as the variable. In Simula, which is an old language, all instance variables are accessible even from outside the object, unless the variable has been explicitly declared as *protected*. In Eiffel, a compromise between the above two methods has been made; all

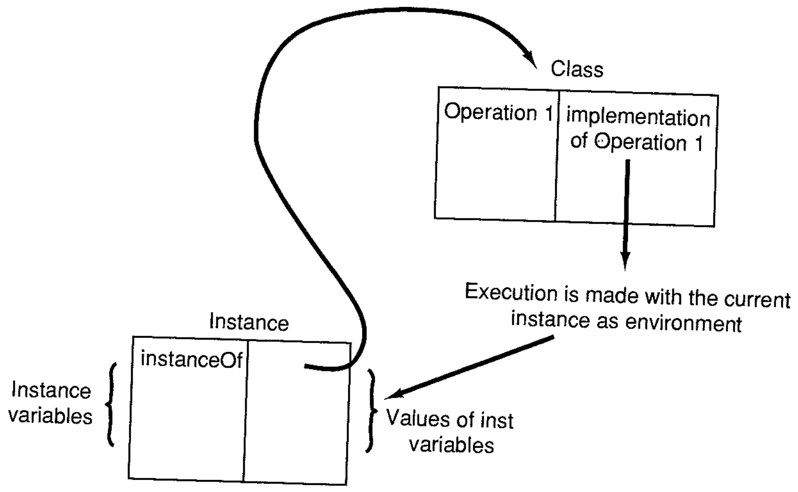

Class
Operation 1
implementation of Operation 1
Execution is made with the current instance as environment
Instance
instanceOf
Instance variables
Values of inst variables

Figure 5.4 Different instances execute the same operation, but in different environments. This means that it is the instance's variables which are used by the operation.

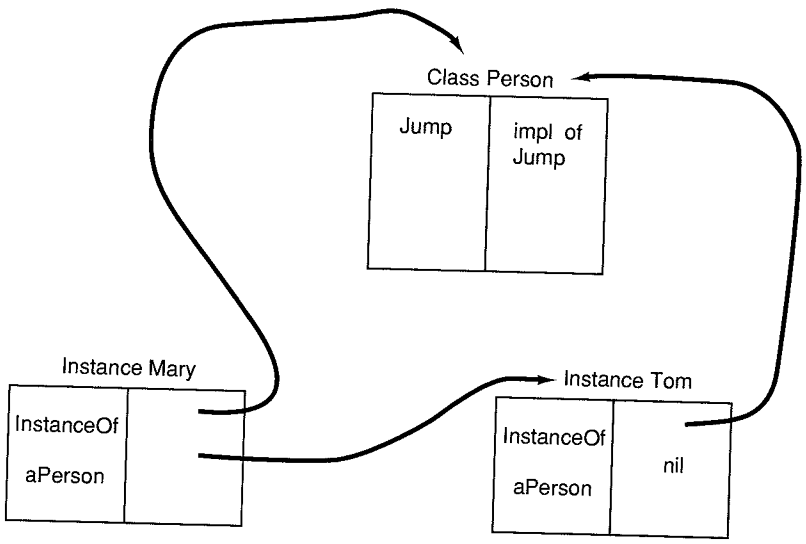


Figure 5.5 The instance variable aPerson refers to the instance Tom (nil means that nothing is referenced)

instance variables are inaccessible, unless the opposite is explicitly stated. This means that if we execute the expression:

```
Age = Aperson myAge,
```

in, for instance, Eiffel or Simula, we do not really know if the operation myAge is performed or if we only read the variable myAge. This is an elegant solution since we could change the implementation without changing an object's clients.

Some languages also support the notion of class variables and class operations. An example of the information class variables may contain is how many instances are created from this class. Several programming languages do not provide the programmer with the possibility to declare and use class variables. This is the case in Eiffel, for example, where the distinction between the class and the instance is carefully made. The class is viewed as a description, or the program text, while instances are viewed as executions of the program text. Thus instances are the only thing that exists during run-time, and not classes In Smalltalk, and to a certain extent in Objective-C, classes are regarded as instances of a meta-class, see the box on classes as objects. In C++, the programmer hasn't really the possibility to declare class variables, but can declare an instance variable as 'static', which gives all instances the same value for this variable. In such a way, it can be used as a class variable.

Classes as objects

Some object-oriented programming languages also view classes as objects, that is as instances of another class. This is the case in Smalltalk and Objective-C An advantage with this is that it is possible to send a stimulus to a class and thus affect all its (existing and not yet created) instances If, for instance, we want all cars manufactured from now to be blue, it is easy to express this using class variables and operations. By means of regarding classes as objects, a more flexible system is obtained, allowing classes to be modified during execution (which makes it more difficult to understand the classes).

As classes are now referred to as instances, they must be instances of some class This class is often called a **meta-class**. Each class has therefore been supplied with a reference, instanceOf, which refers to this meta-class Figure 5.6 illustrates the use of a Meta-class. A meta-class contains operations which all classes can understand. Thus all classes have, at least, the protocol defined by the meta-class. For example, the operation 'new' is often defined in the meta-class. In Smalltalk-78, there is only one meta-class common for all classes, but in Smalltalk-80, each class has an individual meta-class In this way, a whole shadow hierarchy is found behind the 'real' class hierarchy.

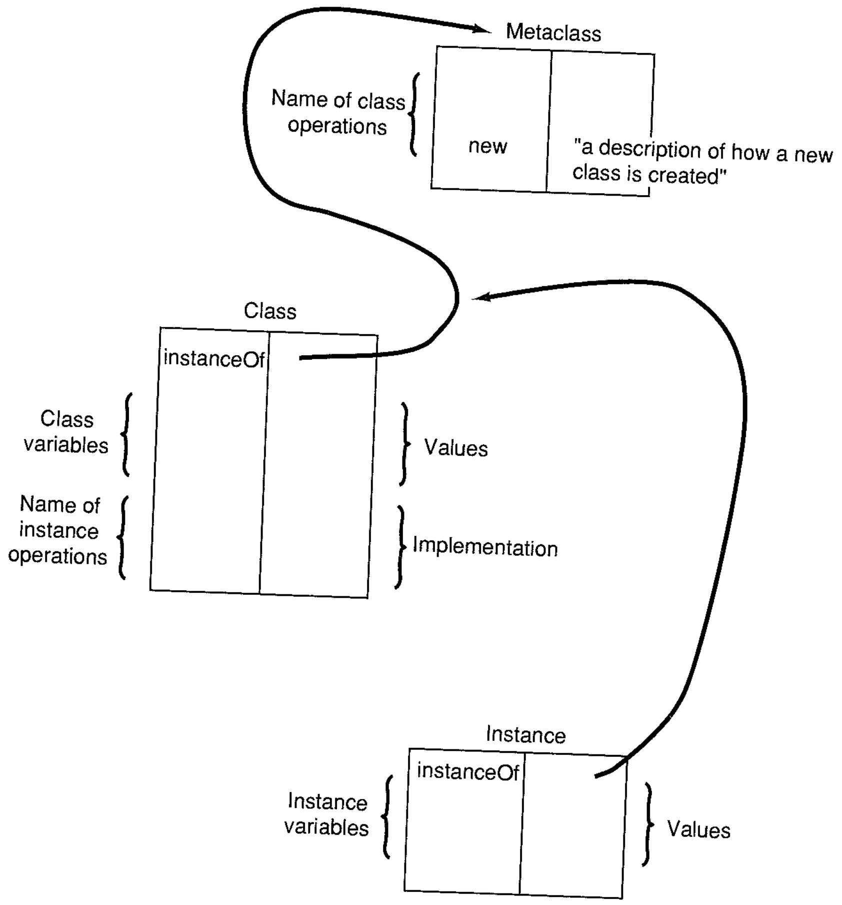


Figure 5.6 Classes and Meta-classes.

Which class is the meta-class an instance of? In Smalltalk-78, the problem of this potential limitless chain has been solved by naming the root class Object as the meta-class' instanceOf In Smalltalk-80, the solution is more complicated

Meta-classes are really only a method to implement the class concept and the underlying system. Even if it is possible, the normal programmer should not use meta-classes The more advanced user can, by means of modifying the meta-classes, modify the language's syntax and semantics; but, as mentioned, this is not recommended for the average programmer

Languages such as Eiffel, C++ and Simula have chosen not to consider classes as objects In Eiffel, this is a very deliberate choice, as the class is regarded as implementing a type, while the object is regarded as an instance of this type The class is static and described in the program text, while the instance is dynamic and exists only during execution Thus, in Eiffel, the distinction between description and corresponding execution is made fundamental. The aim with this approach is to avoid any misunderstanding and obtain a clear distinction between class and instance

Normally, operations are only performed on instances, but some languages allow operations to be performed directly on the class. In the same way as with instance and class variables, we can also have **instance operations** and **class operations**. Class operations are normally used to operate on class variables or to create new instances. In Smalltalk, class operations are common, while in Eiffel, they have been avoided so as to separate clearly the description from the execution.

However, every programming language has operations for creating new instances. This can be seen as a class operation as it operates on the class. In Smalltalk, this is not significant, but in Eiffel, it is considered to be an exception The stimulus is often named new (Smalltalk, C++, Simula and Objective-C) or Create (Eiffel). The stimulus is sent directly to the class, as there will be no instance, as yet, that can receive it. Each class has, therefore, an operation (corresponding to) new or Create, and when the class receives the stimulus, a new instance is created by allocating storage area for the instance and initiating instance variables (set to default values). The operation also makes sure that one of the instance variables, instanceOf, refers to the instance's class Figure 5.7 illustrates in Smalltalk/V, Eiffel and by using a Weizenbaum diagram, how classes are declared with instance variables, how an instance Tom is created, how aPerson is specified to reference Tom and how Tom is made to Jump.

5.4 Inheritance

Inheritance means that we can develop a new class merely by stating how it differs from another, already existing class. The new class then inherits the existing class. The main advantage with this approach is that existing classes can be reused to a great extent. The more general classes are placed higher up in the inheritance hierarchy, whereas the more specialized ones are placed lower down. It is not only the classes which have been designed for the current system that can be reused, but also those designed earlier, maybe in earlier projects. In Smalltalk, this is one of the major ideas. Smalltalk is delivered with an extensive class library and, as such, the program-

Smalltalk/V

```
(1)  Object subclass: #Person

(2)  inst.variable: 'aPerson'

(3)  jump ..

(4)  tom := Person new.

(5)  aPerson := tom.

(6)  aPerson jump
```

Eiffel

```
(1)   class Person

(3b)  export
        Jump, ...

(2)   APerson : Person;

(3a)  Jump is
        ...

(4)   Tom := Person.Create;

(5)   Aperson:=Tom;

(6)   Aperson Jump;
```

Weizenbaum diagram

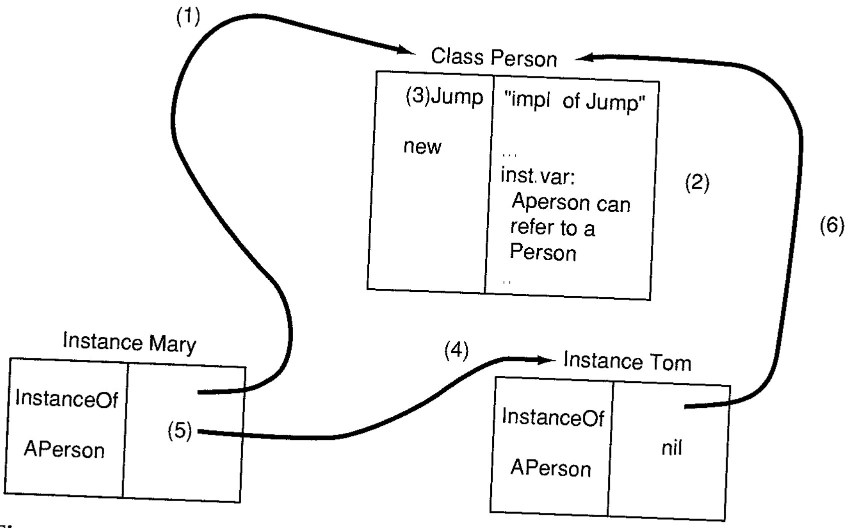


Figure 5.7 (1) declaration of a class, (2) declaration of an instance variable, (3) declaration of an operation, (4) creation of a new instance (Tom), (5) assigning an instance variable, (6) Jump is sent to Tom. The example shows Smalltalk/V, Eiffel and a Weizenbaum diagram The code is not complete

ming is based to a large extent on reusing these classes. Even other object-oriented languages, such as C++, Eiffel and Objective-C, have class libraries.

The inheritance mechanism thus simplifies the process of reuse In traditional programming languages, procedures are the reuse 'level', but in an object-oriented programming language, the reuse levels are at classes. A class contains several operations

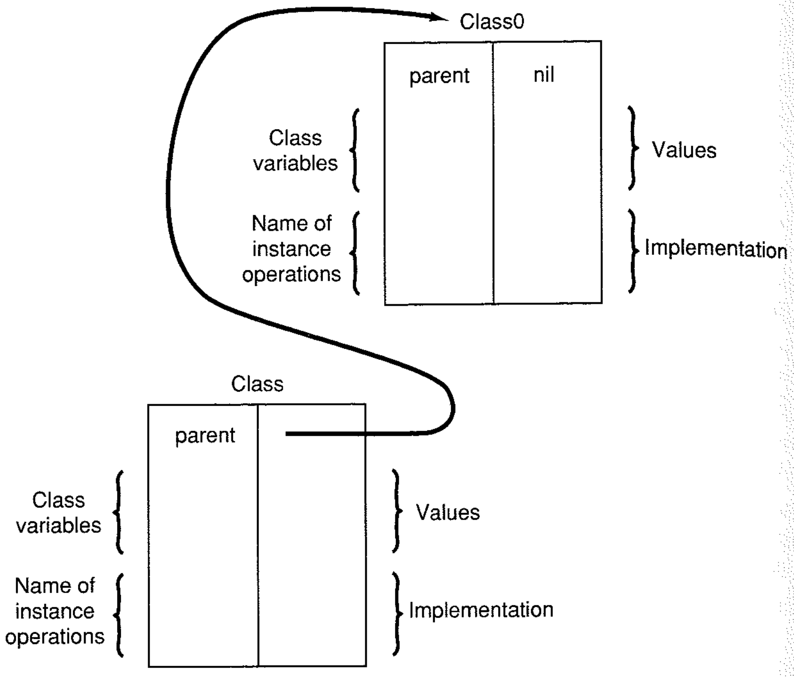


Figure 5.8 Environment for inheriting classes

(procedures) and a data structure. This makes the reuse of classes a much more powerful feature than the reuse of procedures. The inheritance mechanism enables the reuse of whole classes and class hierarchies. However, inheritance is not a prerequisite for reuse as shall be discussed in Chapter 11.

Inheritance enables modifications to be performed in a simple way, as a property common to several classes is implemented in a class inherited by these classes. If we wish to modify this property, we need only modify the corresponding ancestor to update the characteristic in all descendants.

As a result of an operation being possibly defined in a class higher up in the inheritance hierarchy, the previously outlined algorithm for operation searching will need to be slightly modified. If we cannot find the operation in the instance's class, we must proceed to the class' parent, and so on. Thus the class must have a reference, parent, to its parent class. Figure 5.8 illustrates this.

If the operation does not exist in the environment referred to by parent, then the search is continued upwards in the hierarchy until, if necessary, the root class is accessed. (The root class is the class at the top, ie. the root, of the inheritance hierarchy). If the operation is still not found in the root class, then there is no interpretation for the received stimulus and an error occurs. Algorithm 5.2 for binding a stimulus to an operation will, therefore, now look as follows:

Algorithm 5.2 Binding a stimulus to an operation with inheritance

Given an instance environment and a stimulus to the instance
1 In the environment referred to by instanceOf the operation corresponding to the stimulus' name is searched for
2 If the operation is not found, perform Step 1 in the environment referred to by parent Continue until the operation is found If parent does not refer to another class, the stimulus is unknown In this case perform error handling
3 Interpret the corresponding operation using both the stimulus' parameters and the instance's environment
4 Return any values from the execution of the operation

Inheritance can be used in different ways, as discussed in Chapter 3. Abstract classes can be used to define a common protocol for several classes The protocol then only provides the operation's signature, that is which stimuli can be received, while the actual implementation may exist in the descendants. Thus the implementation can be different for different descendants (see also below on polymorphism).

Inheritance can, as previously mentioned, be used to reuse an implementation This type of reuse can sometimes contradict the intuitive picture one has of inheritance. A typical example is how the relation between a stack and a double-ended queue (deque) should be shown. Which class should inherit which? If we regard inheritance as a way to express conceptual relations then neither one of the solutions will be appropriate. A double-ended queue is not a stack, nor is a stack a double-ended queue. The reason for wishing to inherit in this situation is to reuse code. The implementation existing for one can be used for the other As, in this case, it is only the implementation that is required, it is possible to declare, in a class stack, an instance variable to reference the deque. The operations push and pop can then use suitable operations on the local deque for its implementation, see Figure 5.9.

It seems wrong to use inheritance in the above example, as it contradicts what was discussed as proper use of inheritance in Chapter 3. As an alternative to inheritance for reusing code, **composition** can be used, see Taenzer *et al* (1989). This means that we can use classes when we develop new classes, not by inheriting

```
class DEQUE[T] export
  add_at_front, add_at_back,
  get_at_front, get_at_back, ...

feature
  ...
  add_at_front (x: T) is
    -- add x at front of deque
  do
    ...
  end -- add_at_front

  get_at_front : T is
    -- get element at front of deque
  do
    ...
  end -- get_at_front

  ...


class STACK[T ) export
  push, pop, ..

feature
  store: DEQUE[T];
  max_size : INTEGER;

  Create (n : INTEGER) is
          – Create a stack with depth n
  do
     store.Create(l,n);
     max_size:=n;
  end – Create

  push(x: T) is
          – Add x on top of stack
  do
    store.add_at_first(x)
  end – push

  pop: T is
          – get top of stack
  do
    Result:=store.get_at_front
  end – pop

  ...
```

Figure 5.9 How an implementation can be reused without using inheritance. Eiffel-like syntax is used.

them, but by creating instances of them and using these instances, as in the example above. Thus designing a class with the help of instances from other classes is another way to obtain the reuse of code: when we implemented stack, in the above example, we reused the deque by creating instances of it. It is thus possible to reuse code by means of two totally different strategies, with inheritance or through composition, even though they have completely different characteristics. Both these methods are very powerful and should be used in programming Lieberman (1986) discusses delegation of common behavior between objects as a complete alternative to inheritance.

Partition hierarchies, as discussed in Chapter 3, are very useful for showing how the system is constructed, and should be used together with an inheritance hierarchy. These hierarchies enable classes to be structured in a different way to the inheritance hierarchy. Partition hierarchies provide a hierarchy between classes, where class instances can delegate behavior to other instances.

When should we use inheritance hierarchy and when should we use other hierarchies, such as partition hierarchy? The use of inheritance was discussed extensively in Chapter 3. We will here only highlight one simple rule for this decision. We can often view inheritance as 'is-A' and partition as 'has-A'. A dog *is a* mammal and a house *has a* door This simple rule though, is not always sufficient. In languages which allow us to control which operations, including inherited operations, are accessible to individual users, such as C++ and Eiffel, we can specify which operations a user has access to. In C++, we do this by specifying them as public and in Eiffel, by placing them in the export clause. However, in Smalltalk, a user has access to all inherited operations and it is not possible to specify the protocol used by an object, as this is controlled by the inheritance hierarchy. We can see that both strategies have advantages and disadvantages, and so the one selected will depend on a number of factors, some of which were discussed in Chapter 3. Efficiency and how the hierarchy is to be used are two other factors affecting this choice.

The number of inheritance hierarchies in a system varies between different languages. In Smalltalk, there is only one inheritance hierarchy, and thus one root class Behavior, which is common to all system classes, is collected and stored in a root class which in Smalltalk is called Object. All classes inherit Object, either directly or indirectly. Examples of behavior which may be common to instances of all classes is test for determining which class an instance is associated with; comparison between instances; copying of instances etc As Object does not have any parent class, it is also used for error handling in Smalltalk.

In Eiffel, C++ and Simula, there can exist several parallel inheritance hierarchies. One can thus create different hierarchies for different structures in the system It is therefore possible to create new classes without using the existing ones. Both approaches have their advantages and disadvantages. Smalltalk is often used for education in object-oriented programming, as one is forced to work with the inheritance hierarchy. When a new class is to be created, the class *has* to be placed somewhere in the inheritance tree. In this way the programmer is forced to think according to the inheritance hierarchy and also to consider which class is to be used as a basis for the new class to be developed. Programmers who have previously worked in C and begin using C++ may have difficulties both in building inheritance hierarchies and in identifying abstract classes. Therefore C-programmers may initially be educated in Smalltalk and later continue in C++, so that they are forced to learn the use of an inheritance hierarchy and thus learn to use the inheritance mechanism in C++ to a greater extent.

Encapsulation and inheritance are both essential within object-oriented programming Irrespective of this though, they are somewhat incompatible with each other. Encapsulation means that the one using a class should not see its internal representation; but, if we regard a descendant of a class as a user of the class, then the user has complete access to the internal parts of the class This contradiction is due to the fact that we have three types of user: those who use the class via its interface, those who use the class through inheritance, and those who actually implement the class. In C++, therefore, three possibilities for encapsulation of operations and data structures are used: 'public' means that all three user types can access the class operations, 'protected' means that only the class itself or descendants of the class can access the parts of the class, 'private' means that only the class itself can access the parts Of course, these three types are combined when a class is developed. The problem with inheritance and encapsulation has been discussed by, for example, Snyder (1986) and Meyer (1988).

5.5 Polymorphism

The algorithm discussed in the previous section shows clearly that the receiving instance is responsible for searching for and finding the appropriate operation to be executed Polymorphism means that the transmitter of a stimulus does not need to know the class of the receiving instance. The transmitter provides only a request for a specified event, while the receiver knows how to perform this event. In this section, we shall discuss both how polymorphism can be implemented and how it behaves in relation to dynamic binding. The consequences for both typed and non-typed languages are also briefly discussed.

The linking of the received stimulus to the appropriate operation to be executed is performed by **binding** the stimulus to this operation. If this binding occurs during compilation, it is said to be a **static binding**. The polymorphic characteristic sometimes makes it uncertain at compile time, to determine which class an instance belongs to and thus to decide which operation to perform. This uncertainty will not be clear until execution time, when it must of course be known. If this binding occurs when the stimulus is actually sent, that is during run-time, it is said to be **dynamic binding**. Other names for this are late, delayed and virtual binding. Dynamic binding is flexible, but reduces performance; in principle, the operation look-up algorithm is now carried out during execution. This means that each time a stimulus is sent, the algorithm mentioned in the previous section must be executed. However, in most language implementations, sophisticated caching strategies or special function tables are used to minimize this overhead. The advantage with dynamic binding is the flexible system obtained. This flexibility is of most use to systems which are regularly modified. By means of not binding before execution, many of the modifications made will not affect the transmitting object.

Static binding, however, is more secure and efficient. It is more secure, if we have a typed language, due to errors being noticed at the time of compilation and not left to cause failures during run-time. It is more efficient due to the operation look-up algorithm being performed only once during the compilation.

Polymorphism often conveys that the receiver of a stimulus cannot bind the stimulus to an operation before the stimulus is actually sent in run-time as the class is unknown until then. If we do not know beforehand (during compilation) which operation is to be executed on the receipt of a certain stimulus, then we must use some kind of dynamic binding. Dynamic binding is therefore a way of implementing the polymorphism characteristic.

The word polymorphism originates from Greek and means 'many forms' or 'many types'. This means that the referenced item can be of different types. How does this relate to non-typed languages such as Smalltalk? Is polymorphism useful for non-typed languages? Yes, it is. The confusion occurs due to one often referring to Smalltalk as being a non-typed language. In reality, though, it is the *variables* in Smalltalk that lack type, whereas each instance has a very clear type, namely its class It is thus this type that is referred to when discussing polymorphism in Smalltalk In Smalltalk, polymorphism is normally not restricted through the use of an inheritance hierarchy (see below), as is often the case with strongly typed languages. In Smalltalk, the referenced instance can be associated to any class in the system

In strongly typed languages, however, such as Eiffel, Simula and C++, each reference to an instance has a type that specifies the classes to which the reference can refer. When writing the program text, the variables are declared and the stimulus to be sent is specified. Both the instance's class and the stimuli to be sent must then be known. Consequently, it should be known which operations are to be executed at the time of compilation and therefore we should be able to bind the operation and stimulus statically. Is polymorphism then useful for strongly typed languages? Yes, but the reason is that we need not specify *exactly* with which class the receiving instance is associated. Typically we only specified that the instance shall be associated with class A *or* some of class A's descendants. The operation can be defined in a descendant and we thus cannot bind before execution, as it is only then known *exactly* with which class the instance is associated.

Polymorphism can, in fact, be used without having dynamic binding This is the case if we have declared a variable to a parent of the instance's class, and the operation to be performed is declared in this parent class and it is not overridden in any descendant. Irrespective of the descendant class with which the instance is associated, the exact operation will be known during compile time. We can then statically bind the operation to the stimulus. Note that we have still used polymorphism, as we do not know *exactly* to which class the actual instance belonged.

An uncertainty arises though, if the operation, in the above mentioned case, is redefined in a descendant. Can we still statically bind the operation? No, since we do not know the actual class associated with the instance and thus we do not know the correct operation. In Eiffel, dynamic binding will occur in all situations. In C++ and Simula however, the language designer has in this case chosen to statically bind the stimulus to the operation known during compilation, namely that which applies in the parent class. This is then regarded as a deliberate choice by the programmer. If one wishes to delay operation binding until execution, the possibility exists to declare an operation as 'virtual', which means that it can be redeclared in a descendant Then we force the compiler to postpone the binding to dynaminc binding In Eiffel, the possibility exists only

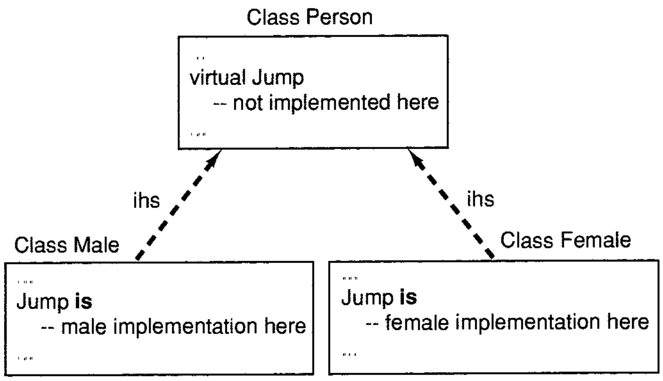


Figure 5.10 Example of use of virtual operations.

of declaring an operation in a parent and forcing descendants to implement it (called deferred) This is used when there is no implementation in the parent class to force the descendants to implement the operation. Figure 5.10 illustrates an example of the use of virtual operations.

Dynamic binding can also be used without the need for polymorphism, as in Prolog for example, where a variable can be bound to an arbitrary term that has no type. Dynamic binding and polymorphism can thus be used independently of each other, and it is unfortunate that they are often mixed up in the object-oriented community.

5.6 An example

We shall here consider how we can use the classes from Chapter 3 to construct a whole game, where people interact with each other. The people involved are a father with his two sons, a woman and an anonymous person. We shall begin by looking at the class Game:

```
Class   Game          |
             parent   | GeneralGame
    instanceVariables | first_son : Male
                      | second_son : Male
                      | woman : Female
                      | father : Male
                      | aPerson : Person

                  new | "how to create a new game instance"
```

```
startGame | first_son := youngMale new("John")
          | second_son := youngMale new("Tom")
          | woman := Female new("Mary")
          | father := oldMale new("Donald")
          |
          | woman storeAge(27)
          | woman friend(father)
          | second_son jump
          | first_son walk
          |
          | woman dance
          | aPerson := first_son
          | aPerson jump
          |
          | aPerson := father
          | aPerson jump
```

The class is created as a direct descendant of the existing class GeneralGame. The instance variables of the game are declared. Game has two operations: firstly, new, which creates and initiates a new game instance, and secondly, startGame, which starts a game. To create an instance of the class Game, the following expression is executed:

```
currentGame := Game new
```

The operation new, in the class Game, is now executed and an instance of Game is created The instance's environment will be as follows:

```
Instance currentGame |
          instanceOf | Class Game

           first_son | nil
          second_son | nil
               woman | nil
              father | nil
             aPerson | nil
```

We can see that the instance contains the instance variables declared in the class. The instance variables have not yet received any values and therefore have a value of nil (no reference). In order to create new people which the instance variables can reference, we require classes for them. We therefore use the class structure developed in Chapter 3, with a class Person, two classes Male and Female which inherit Person, and two classes oldMale and youngMale which inherit

Male. The class Person is created as a descendant to the class Object, and is as follows:

```
Class  Person         |
                parent | Object
     instanceVariables | name : String
                       | age : Integer
                       | myHead : Head
                       | myLegs : Legs
                       | myBody : Body
                   new | myHead := Head new
                       | myLegs := Legs new
                       | age := 0
         StoreAge(Age) | age:=Age
                  Age? | return age
                  walk | myLegs walk
```

Note that the operation new creates new instances of Head and Legs, but not of Body. This is due to males and females having different bodies and this difference is specified in classes Male and Female. These two classes are direct descendants of Person and implement individually their differences to Person. When new is performed on a class, it thus must execute in both the actual class and all its parent-classes.

```
Class  Male           |
                parent | Person
     instanceVariables | --
                   new | myBody:= MaleBody new
                  jump | myLegs bend
                       | myLegs stretch
                       | myBody jump

Class  Female         |
                parent | Person
     instanceVariables | friend : Person
                   new | myBody:= FemaleBody new
          Friend(name) | friend:=name
                  jump | myLegs bend
                       | myLegs stretch
                       | myBody jump
```

```
dance | friend dance
      | myBody shake
      | myLegs bend
      | myHead shake
      | myLegs stretch
```

Note that myBody is a polymorphic reference as it can refer to instances of both classes MaleBody and FemaleBody These classes are not shown, and their creation is an exercise for the reader. The classes oldMale and youngMale contain two variations of the operation dance and their declaration may also be exercises for the reader.

In order to start the game, we perform the operation startGame on the instance currentGame Thus we execute the following statement:

```
currentGame startGame;
```

When this is executed, instances will be created as specified in the operation startGame. We show here only the instance woman after execution of this operation:

```
Instance                |
      woman*instanceOf* | Class Female

                   name | 'Mary'
                    age | 27
                 friend | [reference to "father" instance]
                 myHead | [reference to woman's Head]
                 myLegs | [reference to woman's Legs]
                 myBody | [reference to woman's Body (Female)]
```

5.7 Summary

Object-oriented programming is a style of programming where the core concepts of object-orientations are used. In this chapter we have looked at mechanisms for these concepts.

The thing to program in an object-oriented programming language are the classes. In the class the programmer defines the variables and the operations associated with the class and instances of the class. From these classes, instances are dynamically created during execution of the program.

Inheritance provides a mechanism to create new classes as modifications to existing classes. However, since an operation can be implemented in some ancestor class, an operation look-up alghorithm must be executed to find the correct operation. Inheritance is not a prerequisite for reuse. The construction of a new class can very well benefit from existing classes without inheriting them.

Polymorphism can be implemented using dynamic binding. This means that the actual binding of a stimulus to a specific operation is not done until the stimulus is actually sent during run-time. This often means that the operation look-up must be done dynamically during run-time. However, many language implemetations have solved this efficiently so that the look-up overhead cost is minimized.

Part II
Concepts

6

6.1

6.1.1

6 Architecture

6.1 Introduction

This chapter discusses the architecture of the pyramid, see Figure 6.1. Here, we wish to provide a reason and motivation for the models created and the concepts used when working with Object-Oriented Software Engineering, OOSE. The chapter is relatively abstract, and thus may be skipped on the first reading. It is intended for someone who wishes to obtain a deeper understanding of the architecture layer

6.1.1 System development

In an organization, work is continually modified. One of the ways we use to carry out these modifications is to develop new systems. Here, we use the concept 'system development' to describe the work that occurs when we develop computer support to aid an organization. We wish to emphasize the importance of regarding system development as a means for supporting parts of an organization. Thus the system should be seen from the organization's and user's perspective.

When a requirement for a system is identified, system development begins. The requirement and picture of the system becomes firmly identified. Eventually, one decides to develop the system and write a requirement specification in some form. In this, what one wishes to obtain from the system is specified. If the system is to be developed by someone outside the organization, this specification is used for quotations and ordering of the system; if the system is to be developed internally, the specification is used to plan and control the development process. (Often all of this is actually enterprise modeling which produces a requirement specification (in one form or another), but since we do not discuss it in this book, this naive view of it will be sufficient.)

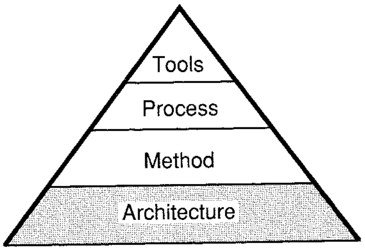


Figure 6.1 This chapter discusses the architecture layer of OOSE.

From this requirement specification, we develop the system. Our wish is to deliver a system with the required quality, and make it as effective as possible When delivery has been made, the most important period for the system begins, namely its operation. This will comprise a large amount of maintenance and further system development, and may span some decades. This part is often the most expensive part in system development, that is, of the system's life cycle cost.

One can, of course, debate whether all these activities belong to the same system development or are rather several different ones, but this is not of interest to our discussion here What is of interest is that this process goes on for a long time and that maintenance and further development forms a major part of this process. This is not unique for development of software systems, but is true for all manufacturing industries. What is unique, though, is that development of software systems is extremely complex, and that we handle this complexity poorly, see Brooks (1987).

OOSE may be used from the time that the requirement specification exists in some form, and all the way until the system has been in service and has been replaced by another system, that is, we include new development, further development and maintenance of the system. We have even gone so far as saying that the main case is further development of a system and that new development is only a special case of this, even though it is a very important case.

6.1.2 Object-orientation, conceptual modeling and block design

As mentioned in Chapter 2, the basis of our approach originates from three totally different techniques which have all been used for a long time These are object-oriented programming, conceptual modeling and block design

Object-oriented programming started with Simula, which was developed at the Norwegan Computing Center during the 1960s. It was developed initially as an extension of ALGOL to handle simulation applicatons, see Dahl and Nygaard (1966), but it was soon discovered by a few groups that it was usable for many other application areas too However, it was not until the 1980s that object-oriented programming established itself widely. The real breakthrough was due to object-oriented programming being very suitable when developing graphical user interfaces, GUIs. Such systems are difficult to describe with traditional programming languages. It is actually only during recent years that it has been widely noticed that object-orientation is usable in most application areas. So object-oriented programming has had a quite difficult childhood, often being rejected. The concepts we have borrowed from object-oriented programming are mainly the idea of encapsulation, inheritance and the relationship between classes and instances.

Conceptual modeling has been used in several different contexts since it came to life in the 1970s; examples are analysis of information management systems and organization theory. The aim is to create models of the system or organization to be analyzed. Depending on which system and which aspects one wishes to model, different conceptual models are created. As one normally models a system where information handling is central, the concept of conceptual modeling is often used as a synonym for data modeling and is often used together with discussion of structuring and use of databases. For more on conceptual modeling, see Hull and King (1987), Tsichritzis and Lochovsky (1982) or Brodie *et al* (1984) In OOSE, we have expanded the technique with object-oriented concepts and the possibility to express dynamic behavior The models we develop are used mainly to understand the system and to obtain a good system architecture These models form the basis for the actual system design.

The method of **block design** originates from LM Ericsson and was developed beginning in the 1960s. It is now in widespread use over the whole telecommunication area, but has also been used in totally different application areas. The ideas were developed from considering how one designed hardware This was constructed using modules with certain functionality being connected together with a well-defined interface. One should be able to do the same thing with software; collect together programs and data into modules (blocks) and describe their mutual communication with signals A new software approach was however required in an attempt to avoid the problems that a program error could create: an error could shut down the whole system By means of encapsulating program and data, these program errors would have less effect. By means of working with blocks, it was also easier to change and introduce new functionality during operation; a new block is loaded and only one pointer is readdressed to point to the new block. Additionally, whether certain parts would be developed in software or hardware were not clear. It was thus desirable to be able to change the implementation easily. Now the technique has been generalized and simplified to be usable in several different application areas. It is mainly used during construction Most of the fundamental concepts were present already in the late 1960s and early 1970s Furthermore, the early modeling concepts have been proposed as contributions to the development of the CCITT standard, Specification and Description Language, SDL. Concepts like blocks with encapsulation of data and behavior, signals between these blocks, and what we now call service packages were present, as well as the interaction diagram technique already in those days.

These three techniques, which have all been used for a long time, have thus been background technologies for OOSE as described in this book. Note that this approach was developed long before many of the widespread methods used today in object-oriented development, like Coad and Yourdon (1991) or Booch (1991) for explanation of why these have not been background technologies. The ideas and working methods have been put together and the concepts made unambiguous and related to each other. The result has shown itself to be a powerful and flexible development method. The technique should not be seen as being fully developed, as further development occurs continuously. We have also started to generalize the method, so that it can be used in even more areas. The latest domain in which we have started to work is enterprise modeling. Just for interest, we can mention here that we use the method, to a large extent, to develop the method itself, see further details in Appendix A. We thus regard Objectory as a system to be developed further in new versions.

We shall discuss first in this chapter what we wish to obtain from an architecture for a system development method, process and tools and what problems we really want to solve. Common characteristics within all the models are then discussed. Each and every model is thereafter covered in an individual section, where the model's object types are briefly presented The discussion in these sections is informal and aims at giving an intuitive understanding of the architecture.

6.2 System development is model building

6.2.1 Models

System development is a complex task Several different aspects must be taken into consideration What we wish to accomplish at the end is a reliable computer program that performs its tasks properly. Typical for system development is that it is so complex and that we can only poorly handle all prerequisities simultaneously For very small programs we can take the requirements and write the program directly, but this is utterly implausible for the systems discussed here What we need to do is to handle the complexity in an organized way. We do this by working with different models, each focusing on a certain aspect of the system. By introducing the complexity gradually in a specific order in subsequent models, we are able to manage the system complexity.

We work with five different models These models are the following

- The **requirements model** aims at capturing the functional requirements,
- The **analysis model** aims at giving the system a robust and changeable object structure,
- The **design model** aims at adopting and refining the object structure to the current implementation environment,
- The **implementation model** aims at implementing the system,
- The **test model** aims at verifying the system.

Each model tries to capture some part or aspect of the system to be built These models are output of the activities shown in Figure 6.2 as discussed in the forthcoming chapters. We will discuss in more detail later how the models relate to the different activities.

Actually, other types of models may be appropriate and thus also be used (e g. specific hardware models) and also some of the

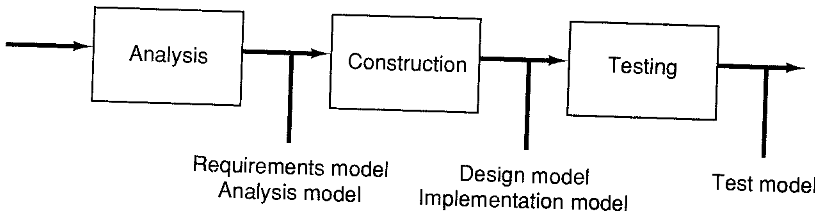


Figure 6.2 The models developed associated with the processes that produce them

models can be merged or only be used as working models not saved. However, we have found these models appropriate to work with in several different kinds of applications of varying size When discussing the different activities in subsequent chapters we will also give some comments on how this model structure can be modified for certain reasons.

The basic idea with these models is to capture right from the start all the functional requirements of the system from a user perspective. This is accomplished in the **requirements model**. Here we describe how a potential user will use the system. This model is often developed in close participation with end users and orderers. When this model becomes stable, the system is structured from a logical perspective into a structure that is robust and, above all, maintainable during the system life cycle. This is done in the **analysis model**. Here, we assume an ideal implementation environment, that is, we do not take into consideration which DBMSs to use, hardware, the current implementation language, whether the system should be distributed or not, real time requirements and so on. We have two main reasons for this The first is that it is much easier to work with ideal circumstances to reduce the complexity and thus focus efforts on giving the application a stable, robust and logical structure. The second reason is that of change Since the implementation environment will change during the life cycle – imagine what has happened with hardware technology in the past ten years – we do not want the current circumstances to affect the system structure. However, the world is not ideal When we have developed an ideal structure, we adopt this maintainable structure with as little disturbance as possible in the **design model**. The reason for this is that we want the design model to be maintainable as well. In this model we decide how to integrate, for instance, a relational DBMS in the application or how to handle a distributed environment. Whereas the analysis model mostly cannot be implemented straightforwardly, the design model should be. When all of these decisions are made and the application further refined and formalized, the **implementation model** is gradually developed. This is the actual code to be composed and/or written. Finally the **test model** is developed to support the verification of the system developed This mainly involves documentation of test specifications and test results.

As we see, the system is gradually refined using these models. These models should not be viewed as sacred or 'the final answer' or anything like that Every organization must decide on which models are appropriate for them. We have found though, that these models form a thorough base to develop many kinds of systems on, and that it is possible to manage the complexity, as it is introduced step by step in the models By focusing on the more important aspects early, a base is laid so that the system structure is modifiable.

The relations between these models are of course important. The transitions between the models are **seamless**. By seamless we mean that we are able to tell, in a foreseeable way, how to get from objects in one model to objects in another model This is absolutely crucial for an industrial development process since the result must be repeatable The method layer will define these transformation rules. To be able to maintain the system it is also necessary to have **traceability** between the models By this we mean that we are able to trace objects in one model to objects in another model. Traceability will in our case actually come as a side effect of the seamless nature of model transformations.

The models are tightly coupled to the architecture, and our aim is to find concepts which

- Are simple to learn and use,
- Simplify our understanding of the system,
- Provide us with a changeable model of the system,
- Are sufficiently powerful to express the information which is required to model the system,
- Are sufficiently defined that different people can communicate the system in these concepts without being misunderstood.

We will discuss these topics in this chapter. In the box on expressable information spaces we give a theoretical perspective to the problems associated with these issues.

Expressible information spaces

This discussion is aimed at giving an intuitive understanding of the problems with system development. We draw an analogy with different areas in mathematics and computer science It should nevertheless only be read as ideas, since the basics are not fully expanded Neither have we handled the terminology very strictly and we make no attempt to be detailed However, we believe that the analogy can help to improve your understanding

An **information space** is a space where information can be expressed It consists of a certain number of dimensions and, in order to create models in this space, it uses concepts that have meaning within this information space Using these concepts, we have the possibility to create models in this space; the more powerful the concept, the larger the information space that we can express The requirement for designing a powerful model is to have concepts extending over (i e expressing) a large information space Our goal, therefore, is to have concepts which span a large and relevant information space.

We can consider the systems we construct as systems to solve problems, for example a telephone exchange shall connect together a number of subscribers, a banking system shall control the accounts of the customers, a process control system shall control a critical process. If a problem can be solved within a finite time, it is called deterministic. The problems we shall solve may therefore be placed in the information space for deterministic problems (otherwise it is not even worth trying!) (This information space is often called the set of NP-complete problems. NP is an abreviation for non-deterministic polynomial and means that one can at least determine whether the problem is solvable in polynomial time.) Amongst these deterministic problems exist the most difficult problems that we can hope to solve within a predetermined time. There are problems that lie outside this space, which are thus not certain of a solution within a predetermined time. An example is 'Guess which number I'm thinking of. You have one try.'

This can feel safe so long as we confine our attention to deterministic problems, and this relates to (nearly) all of the systems we develop. Now, it has been proved that all deterministic problems can be solved with a Turing machine. A computer has the same power of expression as a Turing machine, with the exception of not having infinite memory, but that is seldom the critical problem. Now, the question is: why do we have such difficulty in building a system to solve a problem, which has been proved to be solvable? The answer is that we do not fully master the complexity. Our understanding is too limited. What we want from the concepts is, as you now may realize, to manage the complexity of the information space. We can unquestionably do this with the concepts a Turing machine offers (an infinitely long tape where we can read and write symbols accompanied with a state transition graph). Even if these concepts can express our space, they are so primitive to work with that they can hardly help us. We can compare this with unit vectors in a space or with symbols to count with. If we had only one symbol to count with, we would have to print the symbol 1000 times to express the number 1000; if we had two symbols we can combine these symbols and thus only have a length of 10 symbols ($2^{10} = 1024$). It is therefore not sufficient to have concepts able theoretically to express the information space, they must also be powerful to work with. Another computational tool is lambda calculus. This is just as powerful as a Turing machine, but in order to work out for instance 4×5, a whole page of calculations would be required. Owing to this problem, certain reduction rules have been added to the lambda calculus, but it is still too complex to work with for larger problems.

The development of programming languages also aims at solving this problems. Initially computers were programmed with very primitive languages using ones and zeros to represent the state of some switches. This was at a very low level and the number of errors made increased exponentially with the size of the programs. Assembly languages were developed to represent these ones and zeros in a more human way. Although this was of great help, it was still too low a level for larger programs Still higher level languages, more readable for humans, were developed to cope with this increase of complexity for the larger programs that were developed Compilers were developed to translate automatically from the high-level language to a lower level machine language. One of the goals for programming language developers is to develop a satisfiable expressable programming language, often for some family of applications, and at the same time keep this language as small and simple as possible. Preferably the language concepts should be close to the way people think of a problem

The family of object-oriented programming languages is no exception to this; quite the opposite, they try to follow the way people think and directly map this onto an executable program. Although this is a tremendous improvement for software developers, it is still too low-level to manage the complexity and understanding of the large systems, the development of which we are discussing in this book.

Actually, much of the development of programming languages has been to increase linearly the level of abstraction of the computers. Starting off with the von Neumann architecture for the first computers, see von Neumann (1945), where the basic principle was to build the computer out of five parts; *memory, central control, arithmetical unit, input* and *output*. Computers have since then in most cases been built in basically this way (often called the von Neumann architecture). The first programming languages are naturally designed to cope with this architecture and this forced programmers to think the way computers do It had control statements that manipulated data in the memory In the beginning it was necessary, for reasons of efficiency, to think as computers, but also higher level languages have kept this division of data and programs However, even worse to understand is that we have also kept this division at higher level analysis and design, as is done in function/data methods of software development In object-oriented languages and methods this tradition is finally broken to incorporate data and programs, and to encapsulate the data into the programs We should not blame John von Neumann for us being on the wrong track for so many years, quite the opposite; he is one of the greatest mathematicians of our century and has contributed a lot to the evolution of our field It is the rest of us that lacked the creativity to realize that programming paradigms should not be developed in the same way as computer hardware paradigms, but rather as people think

6.2.2 Architecture

System development thus includes the development of different models of a software system. Our goal is to find a powerful modeling language, notation, or as we will say, a modeling technique, for each of these models. Such a set of modeling techniques (one for each model) defines the **architecture** upon which the system development method is based. Another, more formal way to express this is: *the architecture of a method is the denotation of its set of modeling techniques.* Here we use the term denotation in the way it is traditionally used in the area of formal semantics for programming languages; the denotation of a language construct is what the construct stands for, or the semantics of the construct. Intuitively, you may also think of architecture as the set of all (good) models you can build using the method-defined modeling technique. We may thus view the architecture as the class of models that can be built with a certain modeling notation.

A modeling technique is normally described by means of **syntax**, **semantics** and **pragmatics**. By syntax we mean how it looks, semantics is what it means, and by pragmatics we mean heuristics and other rules of thumb to use the modeling technique.

The modeling techniques are used to develop models. These models should be powerful enough to build the systems we are intersted in developing. The techniques should be easy to use and contain a few, but powerful modeling objects to enable easy learning. They should, most of all, help us to handle the complexity that characterizes the systems we build.

To build these models, we require a **method** to show us how to work with the modeling techniques in order to develop systems. The method describes how we, with the aid of the modeling techniques, can create models of different systems. The specific system architecture we then obtain is formulated in terms of the modeling objects used. A specific system's architecture is therefore the result obtained after applying a method on a system.

An object-oriented view of architecture

In order to make a comparison with object-orientation, we can regard the architecture as a class. For each system we design, we create an instance of this class. The specific system architecture is thus an instance of the architecture that the method is based on. All system architectures have the same characteristics (modifiable, understandable, etc.), but they can all look different. One specific new system development can be seen as sending a Create stimulus to the architecture class which then applies the development method on the architecture for a given problem, see Figure 6.3, and thus creates a new instance (system architecture). The development method can thus be seen as an operation (in Smalltalk: method!) on this architecture, where parameters can be, for instance requirement specification, implementation environment and so on.

To make this development method usable in larger contexts, we

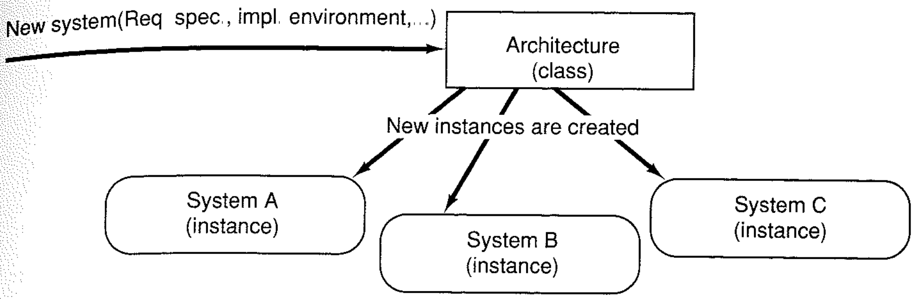


Figure 6.3 The architecture forms the basis (the class) on which new specific system architectures (instances) are created The actual instantiation is described by the method

need to define a **process** and also **tools** To continue with the previously mentioned analogy with object-orientation, we can regard this process as being the union of all operations one can perform on the instances.

6.2.3 Development processes

We mentioned previously that we assume that the requirement specification exists in some form From this specification the system is developed in a first version. Almost all systems will then be further developed continuously, including maintenance of the system. Maintenance will of course also include analysis of new requirements. In Chapter 2 we discussed the problems of describing a development process using a waterfall model A waterfall model only describes an ideal new development. In reality, a system development process is a number of different waterfalls, as discussed in Chapter 2.

Instead of focusing on how a specific project should be driven, the focus of the process is how a certain **product** (deliverable application) should be developed and maintained during its entire life cycle. This means that instead of describing a **project** in a waterfall description, we have to divide the development work for a specific product into processes, where each of these processes describe one activity of the management of a product. The processes work highly interactively. Each process handles a specific activity of the system development, see Figure 6.4 For instance, most types of projects will involve some construction activities. This is described in the construction process. In this manner, the product will be managed by a number of processes. The development work thus extends over all these processes and the processes exist during the whole system

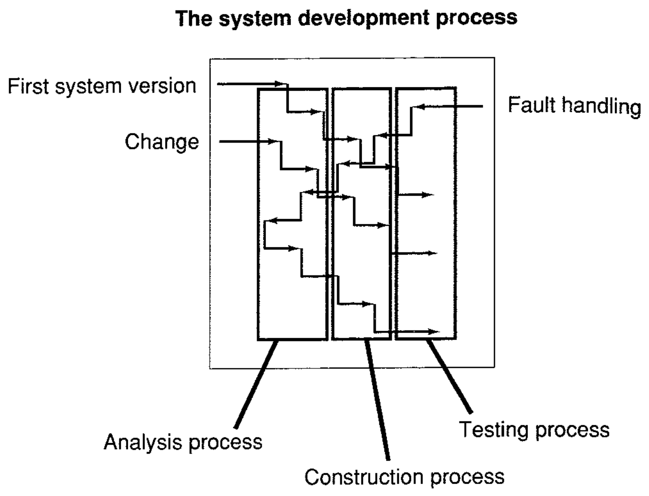


Figure 6.4 Processes intersect several waterfalls.

development, namely during the whole system life cycle. All development work is thus managed by these processes Each process can in its turn consist of a number of communicating subprocesses.

The main processes are the analysis, construction and testing processes, see Figure 6.5. Linked mainly to the construction process, there is also a component process.

In the **analysis process**, we create a conceptual picture of the system we want to build. Here different models are developed in order to understand the system and to communicate it to its orderer and to the construction process. Here the requirements model and the analysis model are developed. In the **construction process**, we develop the system from the models created within the analysis process. This process develops two models; the design model and the implementation model. This process thus includes the implementation and results in a complete system. The **testing process**

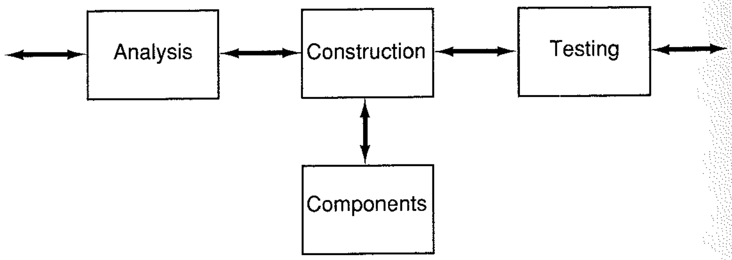


Figure 6.5 Main processes and their relations

integrates the system, verifies it and decides whether it should be passed on for delivery.

Apart from these main processes is the **component development process**, which mainly communicates with the construction process. This process develops and maintains components to be used mainly during construction. The components are implemented construction parts, which can be used in several different applications. The component process is thus not tied to a specific product, but is a multiproduct process.

In Chapters 1 and 2, we discussed the concepts of architecture, method, process and tools. In this chapter, we shall discuss the architecture forming the basis of the method and process, that is, the concepts and semantics of each model. In the following chapters, we shall discuss the method layer of each main process in more detail. Each process has its own chapter. The discussion within these chapters will be confined to the method base from which each process is built. We will also discuss some parts of the processes that are of special interest although most aspects of the process and tool layer are omitted in this book.

Each process can be supported by tools. These tools are essential, especially when the process is used on a large scale. With the help of these tools, we can automate many work steps and also obtain an invaluable aid in keeping documentation consistent.

We thus regard a development as communicating processes. This suggests that they are not some set of fixed and complete procedures able to replace each other mechanically, but quite to the contrary, they have an intensive communication between each other and each of them depends highly on the work done in the other processes. The system development thus iterates over these processes. This is one of several essential differences between method and process.

6.2.4 Processes and models

During a development, we create models of the system we are to design. To design these models, we work from a process description that describes the processes with which we develop the system. Each such process works with models of the system. These models are expressed, or placed, in a certain information space. Each process takes one or several models and transforms it into other models, see Figure 6.6. The final model should be a complete and tested description of the system. This description normally consists of source code and documentation.

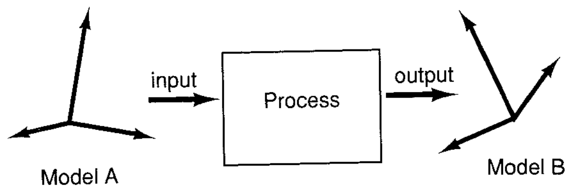


Figure 6.6 Processes transform one model to another model.

To develop software by transforming models from a requirement specification to source code has founded its own paradigms within software engineering, see Agresti (1986) Depending on at which level the transformations are made, they can be divided into either operational or transformational paradigms. The **operational paradigm** first converts the requirement specification into a totally problem oriented operational specification which the users can test out, see Zave (1984) Once this is complete and passed, one can proceed to the implementation factors affecting development. Now, the design and implementation of the system can take place. In the **transformational paradigm**, the transformations are performed on a much lower level, see Partsch and Steinbrüggen (1983). Each transformation should be proved correct, to guarantee that the final result is also correct. All transformations are saved for administration of the system. Today, one can only develop very small systems (programs) using the basic ideas of this paradigm since it is extremely hard to define correct transformations.

The transformations in OOSE cannot entirely be associated with either one of these schools We are, in fact, closely related to the operational approach In comparison with the transformational approach, we are not as formal, but the design and implementation require a lot of intellectual and creative work. Certain parts can be performed mechanically and thus can be supported by CASE tools.

We can regard the system's requirement specification (and what it really means) as a model, placed in an information space. This information space normally has quite unspecific concepts, that is, they are often not very precise, resulting in the need to clarify what is really meant by the requirement specification The requirement specification is one-dimensional in the sense that it is often only a textual description, where the references are forwards or backwards in the text. It is also quite usual that one 'forgets' requirements in the requirement specification. Irrespective of all this though, it represents the initial model within our chain of transformations.

The analysis process produces two models, see Figure 6.7. From the requirement specification, a model is created where one

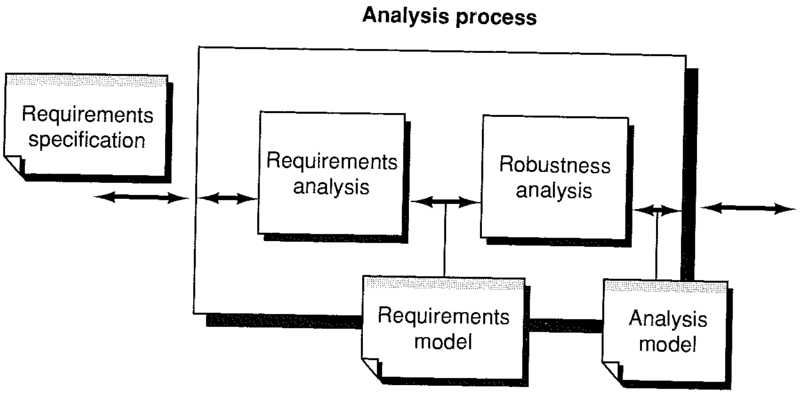


Figure 6.7 The analysis process and subprocesses and the models they produce Note that the arrows between processes are bidirectional to illustrate that information may flow in both directions

specifies what functionality is required from the system We call this model the **requirements model**. In this, we specify all functionality that the system should be able to perform This is mainly done by **use cases** in the **use case model** which forms a part of the requirement model. The use case model will also form the basis for both the construction and testing processes, and it controls a large part of the system development We see here a typical example of reuse on a higher level, on a model level, where one model will be used as input to several processes. The requirements model also forms the basis for another model created by the analysis process, namely the **analysis model** The analysis model forms the basis for the system's structure. In this, we specify all the logical objects to be included in the system and how these are related and grouped. These two models provide the result of the analysis process. They will provide input data for the construction process

In the construction process, we design and implement the system, see Figure 6 8. We shall see that the requirements model and the analysis model provides much support for this process. First a design is made that results in a **design model** where each object will be fully specified The implementation subprocess will then implement these objects and thus result in the **implementation model** which consists of the source code.

This implementation model provides, along with the design and requirements models, input data for the testing process, see Figure 6.9. The testing process tests the implementation model, partly from the requirements model and the design model, and produces a **test model** This test model is really the result from the testing of the implementation model

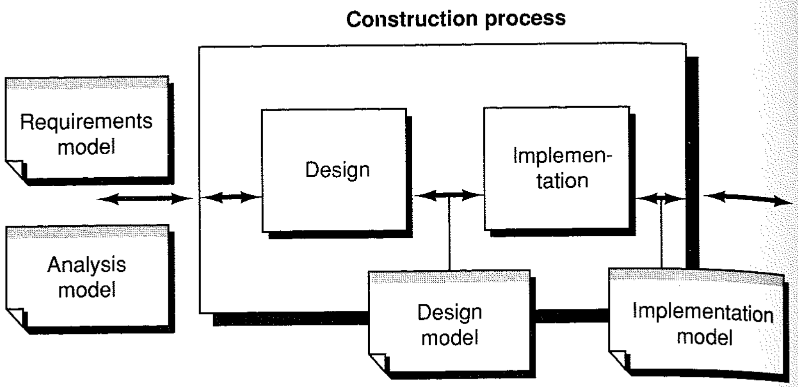


Figure 6.8 The construction process with its major subprocesses and the models produced.

These transformations of models are not as mechanical as has been indicated, quite the opposite, the development of these models is an incremental and creative activity requiring much effort from talented developers. The development work thus flows over these processes which interact with each other The processes thus follow a product and exist as long as the product exists. For a specific project, the issue is to man all or part of these processes We will come back to this issue in Chapter 15

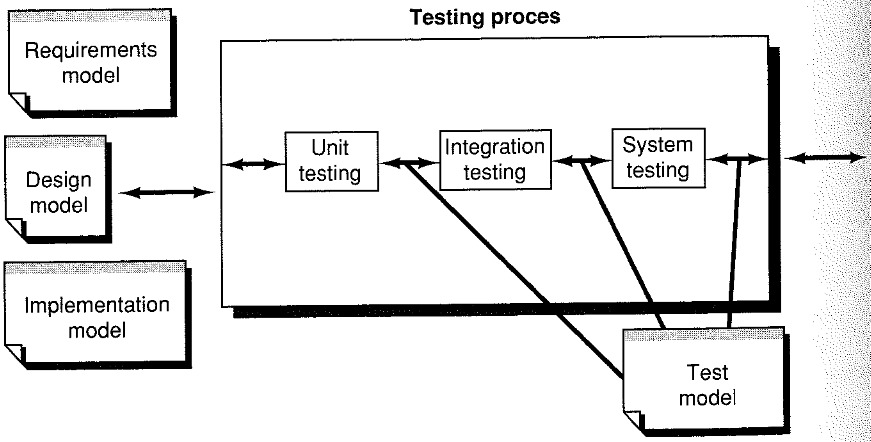


Figure 6.9 The testing process with its major subprocess

As has been discussed, an extremely important characteristic that all our models must support, is traceability We must, from one model, be able to trace an element to an element in another model, and from this new element, be able to return to the same element in the first model For example, we wish to be able to trace something from the test model when a failure is detected, and find the reason for this failure within the implementation model. We may further wish to trace the fault back to the design and analysis models and also to the requirements model and, perhaps, even to the requirements specification. When we modify a model, we wish to see directly its effect on the other models This traceability between models can be difficult to maintain manually during the iterative development work. A CASE tool to maintain this traceability is therefore essential for a large development project.

Before we review each model in depth, we shall first discuss several common characteristics for the objects which we use in the models. In the following chapters we will look at how we work with these models and how they are transformed to new models.

6.3 Model architecture

We have seen that system development is basically concerned with developing models of the system. The work is concerned with both identifying and describing **objects** in a certain information space, and with building models using these objects. Before we look closer at these models, we discuss the object concept common to all models

The object concept was introduced in Chapter 3. All the models built during development in OOSE are built using objects. With the help of objects, we will build **object models**. The objects that we work with in the models all have the properties discussed in Chapter 3

For each model we develop, there exist different types of objects. By modeling with objects in all models, we gain all the benefits of object-orientation, namely locality of changes, encapsualtion, reuse and so on

What, then, is a good object? Much of the work within object-oriented analysis and design consists of trying to find a good object. There are several such criteria in various object-oriented methods. Usually, one claims that the object should have an interpretation in reality One also says that the objects should be obvious, tangible things, things that one can focus on. Others say that the objects are just there for the picking.

The methods available today to find objects are based closely on learning and analyzing the terminology in the problem area, see Shlaer and Mellor (1988), Wirfs-Brock *et al.* (1990), Coad and Yourdon (1991a,b), Booch (1991) or Rumbaugh *et al* (1991). (Object-oriented methods are discussed in Chapter 16.) By analyzing the terminology, one can extract what seem to be candidates for objects in the system. When one evaluates how the system needs to handle these objects, one can decide whether the object should be included or left out of the model being created

However, a good object does not exist on its own. We believe that the criterias should reflect good object models instead. An object can be perfectly right for one model, but totally wrong in another. This means that an object must be placed in a context to see whether it is an appropriate object Therefore, what is really of interest is how an object works with other objects and under what conditions.

What, then, is a good object model? The most important criterion is that it should be robust for modifications and help the understanding of the system As we know for certain that all systems we build will be modified, we must create a robust model structure. Therefore we must analyze how modifications will affect the system. Our structure should be affected by this analysis After we have worked with a model for a while, a stable structure will evolve for the system

By working a long time with the early models, we will obtain a good understanding of the system The development process must therefore be designed so that it results in a sound and robust structure as fast as possible This reduces the risk of having to change the system structure at a later stage and it should also force us to a sound and understandable structure. We will see that the first model designed in OOSE is determined totally by the user's functional requirements, often resulting in that modifications will be local (due to simple traceability), as they often depend on how a user's perspective gets modified We obtain, in such a way, a user governed structure

6.4 Requirements model

6.4.1 Actors and use cases

The first transformation made is from the requirement specification to the requirements model. The requirements model consists of:

- A use case model,

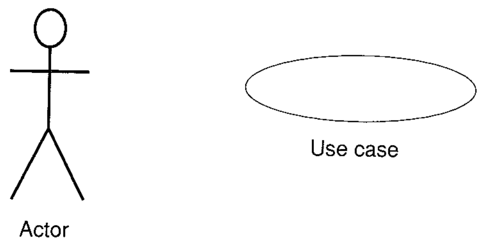


Figure 6.10 The use case model consists of actors and use cases

- Interface descriptions,
- A problem domain model

The **use case model** uses **actors** and **use cases**, see Figure 6 10. These concepts are simply 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. Since the actors represent what is outside the system, we do not describe them in detail. Actors are not like other objects in the respect that their actions are non-deterministic. 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. These instances exist only when the user does something to the system. The same person can thus appear as instances of several different actors. For instance, for a system we may have the actors pilot and clerk. Jim Smith is a user who sometimes act as a pilot and sometimes acts as a clerk; but he performs different use cases dependent on the roles he plays.

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 behaviorally related sequence of transactions in a dialogue with the system. We call such a special sequence a **use case**. An example of a use case could be to Acknowledge a flight to be performed by a Pilot. Each use case is a specific way of using the system and every execution of the use case may be viewed as an instance of the use case. When a user inputs a stimulus, the use case instance executes and starts a transaction belonging to the use case. This transaction consists of different actions to be performed. A transaction is finished when the use case instance again awaits an input stimulus from an

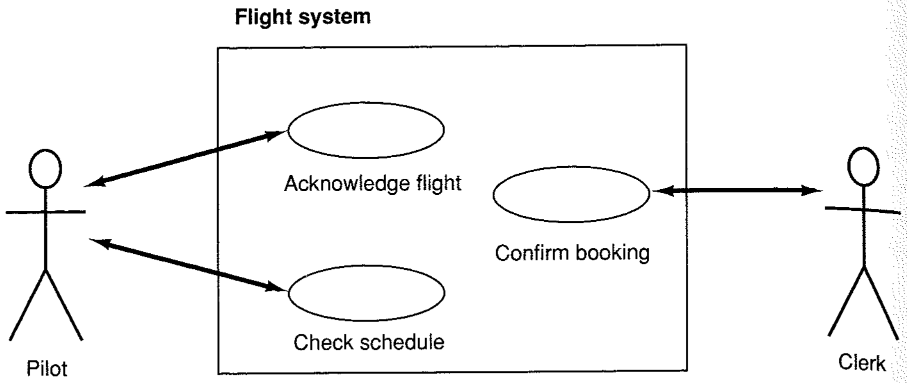


Figure 6.11 Example of a use case model The system is bounded by a box. Each actor is represented by a person outside the box, while the use cases are represented as ellipses inside the box.

actor instance. The use case instance exists as long as the use case is operating.

These use case instances follow, as do all instances in an object-oriented system, a specific class. When a use case is performed, we therefore view this as we instantiate the use case's class. A use case class is a description. This description specifies the transactions of the use case. The set of all use case descriptions specifies the complete functionality of the system. See Figure 6.11 for an illustration of a use case model.

To view use cases as objects, classes and instances is often unnatural to people used to OOP; but let us refer to the definitions in Chapter 3. There we stated that an object should have behavior and a state. A use case is a complete flow in the system. Apparently such a flow does have a state (how far it has reached, what the state of the system is). It also has behavior. We may view every interaction between an actor and the system as the actor invoking new operations on a specific use case (e g. 'start use case'). Thus a use case may be viewed as an object Let us investigate the class and instance concepts. We see that many flows invoked by an actor have similar behavior (the same use case). In this way we can describe this use case and view this description as the use case's class Likewise, when we start a use case we may view this as we create instances of this class. We see that use cases fit into all these definitions. The purpose of this is that we can use all the benefits of object-orientation when working with these concepts. Another possible way of looking at the use case is to view the system as an object and the use cases as operations that get invoked on the system, but this is not the way we have chosen. If the reader has a hard time viewing use cases as objects, it is actually not very crucial when reading this book However, the reason we do it is to make it easier to be able to work with architecture matters. We may thus view use cases as transactions with internal states, having something that represents the course as such which we can manipulate.

Hence the use case model is described by a number of actors and use cases For the use cases, we make detailed **descriptions**. When the system is in operation, instances are created from the descriptions in this model. We will later see that these descriptions are crucial for the identification of the actual objects in the system

6.4.2 Use case driven design

As we design in this way, the system model will be **use case driven**. When we wish to change the system behavior, we remodel the appropriate actor and use case. The whole system architecture will be controlled from what the users wish to do with the system As we have traceability through all models, we will be able to modify the system from new requirements. We ask the users what they want to change (which use case) and see directly where these changes should be made in the other models

Another important characteristic of the requirements model is that we can discuss with the users and find out their requirements and preferences. This model is easy to understand and formulate from the user perspective so that we can easily talk to the users and see if we are building the correct system according to their requirements. Since this is the first model to be developed, we can thus evaluate whether the users are pleased with what we are about to design, before we start to build the actual system, see the operational paradigm of Agresti (1986).

To support the use case model it is often appropriate to develop also **interfaces** of the use cases Here a prototype of the user interface is a perfect tool. In this way we can simulate the use cases for the users by showing the user the views that she or he will see when executing the use case in the system to be built. Additionally, to communicate with the potential users, and also to get a stable basis for the descriptions of the use cases, it is often appropriate to sketch a logical and surveyable **domain object model** of the system. Such an object model should consist of problem domain objects and serve as a support for the development of the requirements model. This approach will be further discussed in Chapter 7.

The requirements model can thus be regarded as formulating the functional requirement specification based on the needs from the

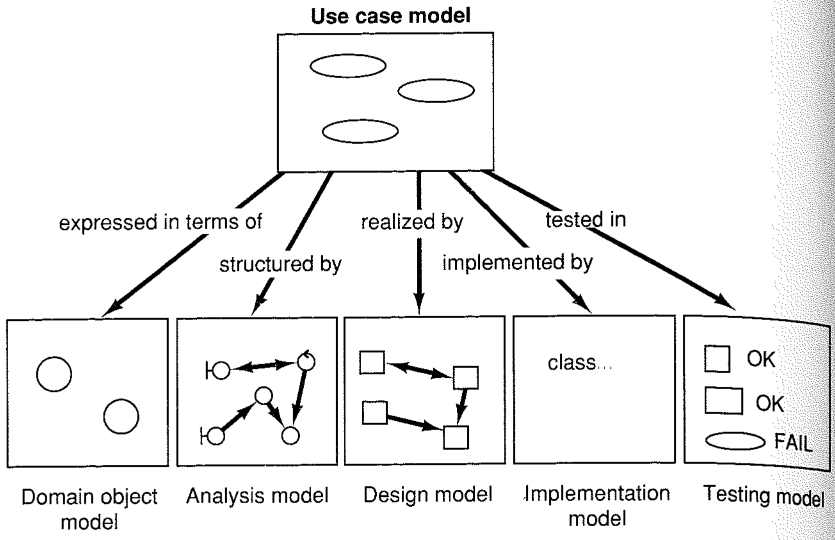


Figure 6.12 The use case model is used when developing all other models.

system users. In reality, a requirements model would do fine as a part of the requirement specification, and we may perhaps use this for system tendering

The use case model will control the formation of all other models, see Figure 6 12. It is developed in cooperation with the domain object model and it may be expressed in terms of domain objects. The functionality specified by the use cases is then structured into a logical, robust but implementation environment-independent model, the analysis model, that is stable to changes. This model is adapted to the actual implementation environment and further refined in the design model using the use cases to describe how the use cases flow over the design objects The use cases will then be implemented by the source code in the implementation model. Finally the use cases will give us a tool when testing the system, and then mainly during integration testing. The use case model will also give us support when writing manuals and other operation instructions.

6.5 Analysis model

6.5.1 The objects of the analysis model

We have seen that the requirements model aims to define the limitations of the system and to specify its behavior. When the requirements model has been developed and approved by the system

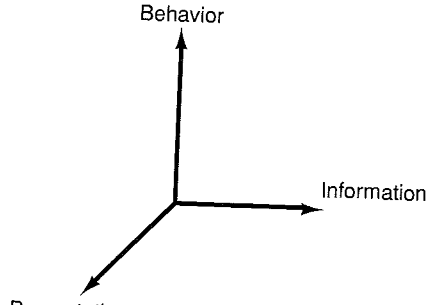


Figure 6.13 The dimensions of the analysis model

users or the orderers, we can start to develop the actual system. This starts by developing the **analysis model**. This model aims at structuring the system independently of the actual implementation environment. This means that we focus on the logical structure of the system. It is here that we define the stable, robust and maintainable structure that is extensible.

It would have been possible to use also the object model developed in the requirements model as a base for the actual construction of the system. In fact, many other object-oriented methods use such a model as the only input for construction. However, this does not result in the structure most robust to future changes. We will come back to the reasons for this. The analysis model represents, from our experience, a more stable and maintainable system structure that will be robust for the entire system life cycle. We will come back to this issue also in more detail later.

Since many future changes will come from changes in the implementation environment, these changes will not affect this logical structuring. We shall see when discussing how to adapt to the actual implementation environment that we want as few changes as possible to this ideal, logical and stable structure.

In the information space for this model, our aim is to capture **information**, **behavior** and **presentation**, see Figure 6.13. The information dimension specifies the information held in the system, both short term and long term. Along this dimension, we describe the system's internal state. In the behavior dimension, we specify the behavior which the system shall adopt. Here, we specify when and how the system shall change state. The presentation dimension provides the details for presenting the system to the outside world.

The analysis model is built by specifying objects in this information space. One possibility is that of having objects which only express one dimension. This is the case with function/data methods, where functions are placed along the behavioral axis and

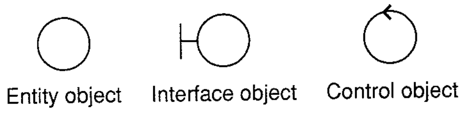


Figure 6.14 The object types used to structure the system in the analysis model

data are placed along the information axis. If we design in this way, we will obtain a system which is sensitive to modification, as we must often modify some behavior when we modify the information structure. We do not want this situation to arise, so our objects should be able to contain both information and behavior, and even the presentation of this if required.

Many object-oriented analysis methods choose to have only one object type, which can be placed anywhere within this space. We have chosen to use three object types. The simple reason for this is to help to get a structure that will be more adaptable to changes. The object types used in the analysis model are **entity objects**, **interface objects** and **control objects**, see Figure 6 14. We will soon come back to the purposes of these object types. Each of these object types capture at least two of the three dimensions discussed above. However, each of them have a certain inclination towards one of the dimensions, see Figure 6 15

Hence we have different purposes with the object types. The entity object models information in the system that should be held for a longer time, and should typically survive a use case. All behavior naturally coupled to this information should also be placed in the entity object An example of an entity object is a person with his or her associated data and behavior. The interface object models behavior and information that is dependent on the interface to the system. Thus everything concerning any interface of the system is placed in an interface object. An example of an interface object is the user

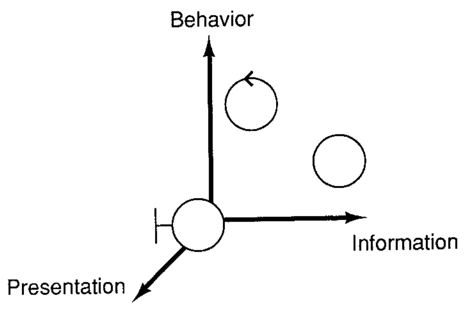


Figure 6.15 The dimensions and object types of the analysis model.

interface functionality for requesting information about a person. The control objects model functionality that is not naturally tied to any other object. Typically such behavior that consists of operating on several different entity objects, doing some computations and then returning the result to an interface object. A typical example of a control object is calculating taxes using several different factors. This behavior could be placed on any of the other types of objects (since they are also able to model behavior), but the actual behavior does not in fact belong to any specific entity object or interface object.

Then why do these three object types give us a stable system? The basic assumption is that all systems will change. Therefore stability will occur in the sense that all changes will be local, that is, affect (preferably) only one object in the system. Let us first consider what kinds of changes are common to a system. As we mentioned in Chapter 4, one of the most common changes to a system is changes of its functionality and its interface. Changes of the interface should typically affect only interface objects. Changes of functionality are somewhat trickier. Functionality may be placed over all object types: so how do we achieve locality in these changes? Is it functionality that is coupled to information held by the system, for instance how to calculate a person's age, such changes affect the entity object representing that information. Changes of functionality of an interface, for instance how to receive and collect information of a person, should affect the corresponding interface object. Changes of interobject functionality, for instance how to calculate taxes from several different criteria, should be local to a control object. Typically such functionality unique to one or a few use cases is placed in a control object.

We thus do not believe that the best (most stable) systems are built by *only* using objects that correspond to real-life entities; something that many other object-oriented analysis and design techniques claim. When comparing a model developed with one of these techniques and an analysis model using OOSE, great similarities will be found with the entity objects in the analysis model and the objects yielded in other methods. Thus entity objects often represent a problem domain object, although this is not always necessarily so, as we will see further on. However, behavior that we place in control objects will in other methods, be distributed over several other objects, making it harder to change this behavior.

The reason that we model with these three object types is thus to have locality in changes of the system. Then how do we know that these are the 'right' object types? This is something you cannot draw any conclusions on until a system has been changed a number of times. Our experience comes from the block design technique metioned earlier. That was an object-based technique where objects (there called blocks) typically had counterparts in the problem domain. The techniques were used in developing telephone exchanges and the experience was quite good; it was very modular and quite easy to introduce most kinds of modifications However, certain modifications are unreasonably expensive to introduce For example, in the system there was a block called Trunk representing data and behavior that should be tied to a trunk (a telephone line between exchanges) When the system had been in operation for more than ten years, and thus had undergone several changes, we investigated what had happened to this block. Of course, many changes had been made to it to adapt to the new requirements and these changes were local to the block This included also many new operations that had been introduced to the block. What we found however, was that 23 (!) operations had been added that were unique to only one specific use case The reason for this was that when a new use case was introduced, it was hard to reuse operations since they were not general enough, so it was easier to add a new operation to handle this specific use case However, often operations were reused and it was quite easy (cheap) to introduce new use cases (functionality), but this was thus not always the case

Thus the actual problem here was that too much specific functionality had been incorporated in the blocks (entity object in OOSE) Instead we should model this specific functionality in the control object so that, firstly the (operations of the) entity object will be more reusable in several different use cases, and secondly the specific functionality should be local and not distributed so that it is easier to introduce changes in this functionality

Similar observations of object-oriented systems are not very common since very few object-oriented structured systems have undergone changes over several versions. Even so, the experience existing with an object-oriented structure is that in general it is more maintainable than a function/data structure However, there exist similar observations on system changes One example is by Scharenberg and Dunsmore (1991), where a system for inventory control was developed with a 'problem domain object' analysis method. The observations were initially that these initial classes were only used as more complex data types and with a not-so-stable structure. The system was refined and further developed, leading to another class structure that was more stable An 'artificial object domain' was reached which did not directly reflect the problem domain, but instead was more stable to changes and more robust. The 'artificial object domain' consisted of three object types very similar to the three object types we use

6.5.2 The requirements model is structured in the analysis model

The analysis model is formed from the use case model Each use case will be entirely divided into objects of these three types In the requirements model we specify the entire functionality of the system. This functionality should now be structured to get a robust and extensible structure. In this way the use cases will be partitioned into analysis objects, see Figure 6.16. In practice this means that the functionality specified in the use cases should be allocated to different objects In this way, each use case will be offered by objects in the analysis model. An object can of course be common to several use cases. This is actually desirable since it gives objects that are reusable in several different use cases, which suggests that they will also be reusable in future changes of the system. This transformation from use cases into objects forms one of the most important parts of OOSE. It is now that we form the basis of the system architecture. We shall here only touch upon the principles for this transformation This transformation will be discussed in more detail in Chapter 7

Basically, we partition the use case according to the following principles:

- Those use case functionalities which are directly dependent on the system's environment are placed in interface objects,

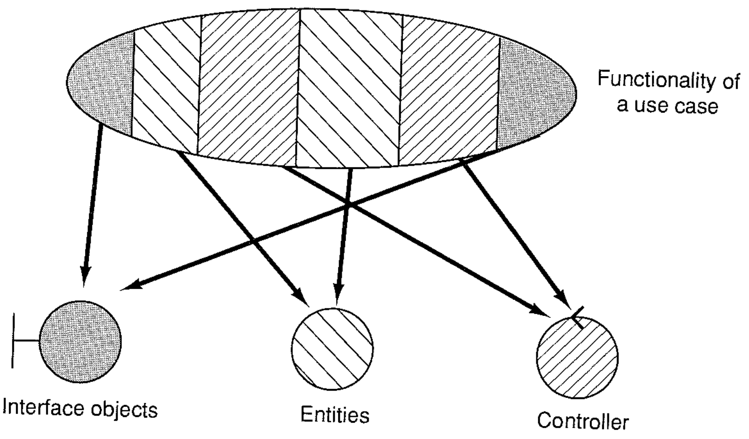


Figure 6.16 The functionality of the use case should be partitioned and allocated to the objects, thus yielding that the analysis model will offer the use cases of the system

- Those functionalities dealing with the storage and handling of information which are not naturally placed in any interface object are placed in the entity object,
- Functionalities specific to one or a few use cases and not naturally placed in any of the other objects are placed in control objects

By means of performing this division, we obtain a structure which helps us to understand the system from a logical view and also one which gives high localization of changes and thus less sensitivity to modifications For instance, changes of an interface, a quite common modification as we have seen, affect only one object, the interface object that handles this interface. Changes of data format would affect only the entity object (structure and behavior) where the change should be introduced.

The allocation is quite difficult in practice, and we shall discuss it much more in the analysis and construction chapters. It is when doing this partition of functionality that we decide upon the robustness and maintainability of the system The basic underlying principle is to get locality in changes

Since it is often hard to draw sharp borders between functionalities in a system, in practice the developers are forced to make many judgements of where to split functionality between objects. We have seen from real projects that the best developers often reason in terms of the potential changes to this system. How will these potential changes affect this structure? And from this they develop a structure that is stable for the most likely changes. We have also seen that this 'speculation of potential changes' rapidly becomes a skill for an experienced developer. By giving the developer the architecture of OOSE, she or he can use his creativity to develop stable and robust systems.

As the objects are identified and specified, we also describe how these objects are related through different types of associations. We shall not, though, discuss these here, but wait until the analysis chapter since it more naturally comes in when working with the objects.

The analysis model is now designed directly from the user's requirements, that is, the requirements model This analysis model will now form the basis for the specific system's architecture, and we see that no consideration has been made of the actual implementation factors. This provides an ideal and sound system architecture, as it is the problem which controls the architecture and not the factors of the actual implementation environment. During the design work, we shall take into consideration these points, but it will then only be a matter of adapting to these circumstances; the basic system structure, as far as possible, remains untouched.

6.6 The design model

6.6.1 The design model's object

In the construction process, we construct the system using both the analysis model and the requirements model. Initially, we create a design model that is a refinement and formalization of the analysis model. The initial work when developing the design model is to adapt to the actual implementation environment. The analysis model was developed assuming ideal conditions. We must now adapt it to reality. As mentioned previously, we have two main reasons for not introducing the implementation environment earlier. The first was that we did not want it to affect the basic structuring of the system, since the current circumstances most probably will be changed in one way or another during the system life cycle. The second reason was that we do not want the problem to be blurred by the complexity typically introduced when looking at the implementation environment. In this way we can focus on the essentials when developing the most important aspect of the system, namely, the basic structure of it.

We can thus regard this design model as a formalization of the analysis space, where we adapt the analysis model so that it fits into our implementation environment. The design space has, relative to the analysis space, been expanded to include also a dimension for the implementation environment, see Figure 6.17. This dimension means that further concepts are introduced, which the analysis model must be adapted to suit.

This means that we want to keep and massage the analysis model to fit the actual implementation environment at the same time as we refine it. Our goal is to refine it so far that it is straightforward to write the source code from it. Since the analysis model has all the properties we want for the system, we want this structure to form the base for the design model. However, there will be changes to this model when introducing, for instance, a relational DBMS, a distributed environment, performance requirements, a specific programming language, concurrent processes and so on. This is the reason that we develop a new model.

How much should we work with the analysis model and when should we start with the design model? This is the classical question

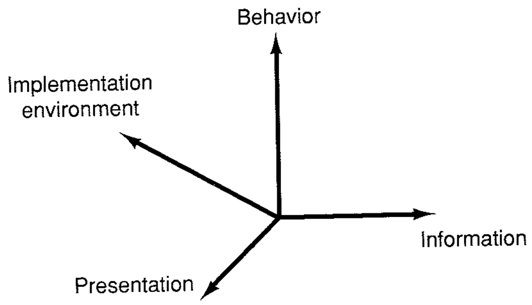


Figure 6.17 In the design space, yet another dimension has been added to the analysis space to include also the implementation environment.

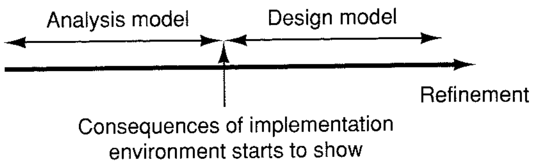


Figure 6.18 The transition from the analysis model to the design model should be made when the consequences of the implementation environment start to show.

'When is analysis ready?' There is no uniformly applicable answer to this question On one hand we want to do as much work in the analysis model as possible, where we can focus on the essentials, but on the other hand we do not want to do so much that we need to change it when adapting to the implementation environment. What we really want is a continuum of refinement in the models where the switch of models will occur where we start to see consequences of the implementation environment, see Figure 6.18.

Hence, when the transition from analysis to design should be made must be decided for each specific application. If there will be no adaptation problems with the DBMS, distributed environment, real-time requirements, concurrent processes, hardware adaptions, and so on, it is fine to be quite formal in the analysis model However, if these circumstances will highly affect the system structure, the transition should be made quite early. The goal is not to redo any work at a later phase that has been done in an earlier phase. In one project using objectory, the development team saw that in one part of the application the implementation environment would have very little consequence since it would be entirely within one process at one site and with few consequences from the database technology used. Here the analysis model was refined quite far, including operations on the objects and also the parameters of these operations.

In another part of the application the consequences were much greater Here they foresaw consequences from distributed hardware involving different operating systems, a changeable protocol between the different nodes (not yet standardized), hard real-time requirements and so on. This part of the analysis model was developed in a much more shallow way, postponing important decisions to the design model

One possibility is to continue working on the analysis model, and continuing on that model also when incorporating the implementation environment into it. However, this is not recommended from a product oriented view. When doing further development on the product, the analysis model is needed to reason about when to incorporate the changes, since it has far less complexity than the design model. In such a way, it is desirable to keep the ideal analysis model of the system during the entire system life cycle Likewise, many changes of a system come from changes in the implementation environment Such changes are then easily incorporated, since it is the same analysis model that will form the basis of the new and modified design model. In this way we may thus view the design model as a specialization of the analysis model for a specific implementation environment. However, the cost of keeping the analysis model must be judged for each product.

This also yields the question of when changes in the analysis model should be made when working with the design model. If a change of the design model comes from a logical change in the system, such as that two object should be logically related, then such changes should also be updated in the analysis model However, if the change is a consequence of the implementation environment, for instance that two objects should not communicate directly due to the process structure chosen for instance, then such changes should not be incorporated in the analysis model.

The structure which we mainly work with are thus basically the same as for the analysis model. However, now the view has changed since this is a step towards implementation. We therefore use the concept of a **block** to describe the intention of how the code should be produced. The blocks are the design objects and they are drawn as rectangles, as shown in Figure 6 19. One block normally aims at implementing one analysis object. Here it could be possible to use different types of blocks as well, if preferable, for instance interface blocks, entity blocks and control blocks to highlight the traceability.

Important to understand though, is that the blocks are *not* the same objects as the analysis objects. We have briefly touched on this before as we mentioned that it is desirable to keep the ideal analysis model. Changes will be introduced in the design model, for example

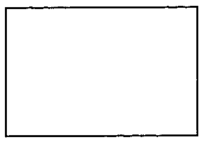


Figure 6.19 The object type used in design is the block, and it is the design object of the functionality in the analysis model

to split one block into two due to the need to handle, for instance, loosely coupled process communication. Such a split should not affect the analysis model and we thus do not want them to illustrate the same model. To highlight the difference we use the term block instead of object

Another difference between the analysis and design model is that the analysis model should be viewed as a conceptual and logical model of the system, whereas the design model should take us closer to the actual source code We can view it as a drawing or map of the code to be developed This means that we change the view of the design model into an abstraction of the source code to be written later on Hence the design model should be a drawing of how the source code should be structured, managed and written. Since we want a strong and easy-to-maintain traceability from the analysis model over the design model to the implementation model (source code), we will try to map the design objects (blocks) to the module concept used in the programming language we are implementing in. We will discuss this topic in greater depth later.

We thus view the blocks as an abstraction of the actual implementation of the system The blocks will thus group the source code. To know how to implement the blocks, we need to refine the design model further. We do this by describing how the block will communicate during execution. To describe the communication between the blocks we use **stimuli**. The concept of stimuli was introduced in Chapter 3 A stimulus is sent from one block to another to trigger an execution in that block. This execution may send new stimuli to other blocks.

To describe a sequence of stimuli, we use **interaction diagrams.** In these, we can describe how several blocks communicate by sending stimuli to each other. As a base for these interaction diagrams we use the use case model again. Thus we describe in detail, for each use case, how and which stimuli will be sent and in what order. We thus describe the use case as a sequence of stimuli sent between the blocks. An interaction diagram is shown schematically in Figure 6.20.

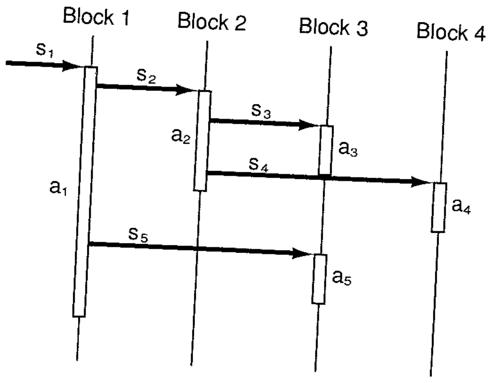


Figure 6.20 An example of an interaction diagram The diagram shows how stimuli, s_n, are sent between the blocks and invoking activities, a_n, in these blocks

When we have described all sequences, that is, all use cases, including any alternative flows and error flows, we have described all external communications between the blocks. From this we have also received a complete interface description of each block The notation to describe stimuli should therefore be the one used in the chosen programming language. The design model thus consists of a complete description of the blocks with their interfaces.

We mentioned that the blocks should be viewed as abstractions of the code to be written and that it is desirable that traceability between blocks and the code is easy. In, for instance, C++, a typical block will be mapped to one file (actually two, h and c-files) including one or several classes. In Ada it is natural to map a block to a package. We denote this module concept by the generic term **object module**, that is, in C++ an object module is the class, in Ada it is a package. The interfaces of the blocks will thus be mapped on the object module of the particular language, so serving as the interfaces of these modules (the h-file in C++ and the specification-part of the Ada package).

One problem in the construction process is that complexity increases enormously. To master the system development, it is essential to be able to manage this complexity. Therefore it is important to have concepts in the information space which allow us to manage the complex systems that we build

To be able to manage the system more abstractly, we use the concept of **subsystem**. A subsystem groups several objects. Subsystems may be used in both the analysis model and the design model. Subsystems may also include other subsystems; the concept is recursive. In this way we may have a hierarchical structuring of the system to manage its complexity. The highest level we use is the **system** level. The system is the actual application we are working with. The system also defines the borders of the application. We will discuss the subsystem concept in more depth later and we shall also see that subsystems may be used as a help to manage product development.

6.6.2 Working with the design model

During the construction work, we proceed from the analysis model. For each object in the analysis model, we assign a block in the design model. This transformation occurs totally mechanically and can be performed by a tool. The transformation is seamless. Depending on, most importantly, the implementation environment, we may need to make a deviation so that this one-to-one relationship may be modified. In the design model we thus formalize the analysis model and adapt it so that it fits into the required implementation environment.

When we have created the block structure, we draw interaction diagrams to show how these blocks communicate. Normally, we draw an interaction diagram for every use case. In reality, the use case model will henceforth form the basis even for the construction process, and we thereby guarantee that the system we construct is exactly what the users want.

In the first part of the construction process we work mainly with the blocks. This is often an appropriately detailed level to work with. However, when maintaining and developing the product further into new versions, it is often appropriate to extract the interfaces to a subsystem level, and thus only specify the interfaces between subsystems early on, whereas inside each subsystem, the development team can work with the blocks. In this way all the teams do not need to know the internal structure of each subsystem. In Chapter 8 we discuss different techniques to manage this.

6.7 The implementation model

The implementation model consists of the annotated source code. The information space of this model is the one that the programming language uses. Note that we do not require an object-oriented programming language; the technique may be used for any programming language to obtain an object-oriented structure of the system. However, an object-oriented programming language is desirable since all fundamental concepts can easily be mapped on language constructs.

The base for the implementation is the design model. Here we have specified the interface of each block and also have described the behavior of what is expected behind this interface. How this description may look will be discussed in the construction chapter.

We stated earlier that it is strongly desirable to have an easy match between a block and the actual object module (i.e. class, package, module or whatever the concept would be in the programming language). In many cases we can do a very easy match from one block to one class in the implementation language. When we have a very smooth implementation envirmonment this is typically the case. However, in more complex environments this is not always the case. In one project where Objectory was used, an entity block ended up using 17 (!) classes in C++ for the implementation. However, it was a very complex implementation because a relational DBMS was used The implementation included things like type conversion, keying, error handling, versioning and suchlike. This is not a normal case. Typically, one block will map on about 1–5 classes. In the example chapters later on, we will illustrate some more complex blocks.

A very powerful implementation tool is the ability to use **components**. These are fully implemented parts which enable us to build a system with more powerful concepts (higher abstraction) than the programming language can offer us. Components can be regarded as completed building elements which are already placed in the implementation space and that can be used directly in our system. Components and how to use them will be discussed in more detail in Chapter 11.

6.8 Test model

The test model is the last model developed in the system development. It describes, simply stated, the result of the testing. The fundamental concepts in testing are mainly the **test specification** and the **test result**.

What is tested is initially the lower levels, such as object modules and blocks. These are tested by the designers Lower subsystem levels are then tested. The integration test does not come as a 'big bang', but rather these tests are introduced on varying levels when integrating larger and larger parts. One tool for integration testing is to use the use case model, to integrate one use case at a time. This is normally performed by an independent testing group. The test is thus performed by starting to test the lower levels, in order later to test the use cases and finally the whole system. The requirements model forms again a powerful tool here; as we test each use case, we check that the objects communicate correctly in that particular use case. Similarly, we check the user interfaces described in the requirements model. We thus see that the requirements model is **verified** by the testing process.

We may also view a test as an object just as we view use cases as objects By doing this we can view the test specification as the test's class, and thus we can also inherit common parts or compose them from several test specifications, which is valuable when writing and thus reusing the specifications. In this way we view a test execution as an instance of this class. The instance thus, quite clearly, has behavior and also a state. The outcome of such an execution is a test result We thus have test results for all different types of tests. The testing model will be discussed in more deatil in Chapter 12.

6.9 Summary

This chapter discussed the underlying architecture of the OOSE method. Here we have defined the concepts and the models that are developed when using OOSE

System development includes many activities during the system life cycle. When developing a product, it is essential to focus on all these activities. When having a life cycle view, one of the most important properties of a system is that the system is maintainable, that is, changes can be made to the system at a reasonable cost.

System development is a complex task and by building models at different levels of granularity this complexity can be managed. Five different models are developed for a product in OOSE. The transitions between these models are seamless, which means that the model transformation is repeatable between different developers. Another important property is that we should have traceability between the models, that is, we should be able to trace changes from one object in one model to objects in another model. The transition between these models are handled by processes These processes manage a product over its life cycle. Four processes are used; analysis, construction, testing and components. In all models, objects are handled.

The first model, the requirements model consists of actors and use cases supported by an intuitive domain object model and interface descriptions. The actors model something that will interact with the system and a use case specifies a flow that a specific actor invokes in the system. The object model gives a conceptual, easy-to-understand picture of the system and the interface descriptions describe the system interfaces in detail. The requirements model will thus completely define the functional requirements of the system from a user's perspective. Since the model has a user perspective, it is easy to communicate with potential users in terms of this model.

The analysis model is developed from the requirements model. The aim is to get a logical and robust structure that will be maintainable during the system life cycle. Three object types are used. Interface objects are used to model functionality that is directly dependent on the system interfaces. The entity objects model information that the system should manage for a longer time and behavior tied to this information. Entity objects typically survive a specific use control case. The objects should model functionality that is transaction-oriented for a specific use case and that should be kept together for maintainability purposes. Typically, they do not survive a use case. The reason for having these three object types is that a change should preferably be local to one object only.

The design model refines the analysis models further and takes into consideration also the current implementation environment. The blocks describe the intention of how to implement the system. The ideal analysis model must often be changed due to a complex implementation environment, even if this is not desirable. However, the basic system structuring should be kept as far as possible as defined in the analysis model. In the design model, the blocks are further specified using the use case model to explicitly specify the interfaces and the communication between the blocks.

The implementation model consists mainly of the source code written to implement the blocks. Here common, and sound implementation techniques should be used. OOSE does not require any object-oriented language to be implemented in, although this is preferable since all the essential concepts exists in these languages.

The testing model is developed when testing the system. Testing is done on several different levels of granularity. The lower level objects are tested initially when performing unit testing. The use case model is used as the primary tool when doing integration testing.

7 Analysis

7.1 Introduction

7.1.1 Why an analysis process?

The aim of the analysis phase is to analyze, specify and define the system which is to be built. The models developed will describe *what* the system is to do. The basis of this modeling is basically the requirements (expressed in some form) on the system. It is then important to carry on a dialogue with the prospective orderers and users, so that the system that is built is really what is wanted. In the analysis phase, it will then be possible to build models that will make it easier for us to understand the system.

The models that are developed during analysis are fully application oriented, and no consideration is taken of the real implementation environment where the system is to be realized, for instance the programming language, DBMSs, distribution or hardware configuration that should be used. This will be our definition of analysis; modeling the system with no regard to the actual implementation environment, see McMenamin and Palmer (1984). The purpose is to formulate the problem and to build models able to solve the problem under ideal conditions. As the models are entirely problem oriented, and no attention is paid to the real implementation environment, they are fairly straightforward to develop from a functionality viewpoint. The models can be discussed from the application's perspective, with application oriented concepts. Thus it will be possible to discuss the models with the users of the system without using implementation terms.

Sooner or later the system will have to be adapted to the prevailing conditions, of course. This is done in construction, when all those considerations are taken into account that have been neglected in analysis. The fact that little heed is taken to the implementation environment will guarantee that the ensuing system architecture will be based on the problem and not on the conditions

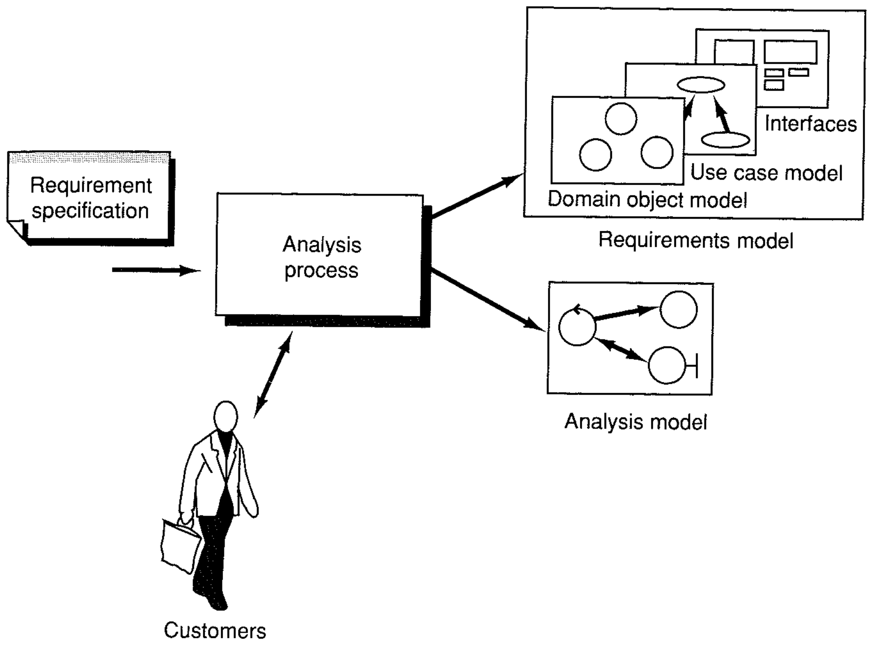


Figure 7.1 On the basis of the requirement specification, both the requirements model and the analysis model are developed in the analysis process

prevailing during the implementation. It will, of course, be impossible to develop the models entirely without consideration of their realization; the models and their architecture must be built so that everything in them can be realized. Another great advantage with this procedure is that our analysis models will remain intact, if and when the implementation conditions change. Hence we can use the same models without changing them even if the implementation environment changes.

7.1.2 What is done in analysis?

Two different models are developed in analysis; the requirements model and the analysis model. The real base is the requirement specification and also discussions with the prospective users, see Figure 7.1. The first model, the requirements model, should above all make it possible to delimit the system and to define what functionality should take place within it. For this purpose we develop a conceptual picture of the system using problem domain objects and also specific interface descriptions of the system if it is meaningful for this system. We also describe the system as a number of use cases that are performed by a number of actors. The actors constitute the environment of the system, and the use cases are what takes place within the system. The use case concept is one of the unique concepts of OOSE.

The analysis model gives a conceptual configuration of the system, consisting of control objects, entity object and interface objects. The purpose of this model is to develop a robust and extensible structure of the system as a base for construction. Each of the object types has its own special purpose for this robustness, and together they will offer the total functionality that was specified in the requirements model. The analysis model does not have any counterpart in other methods, but is unique to OOSE. Although, since the techniques that we use are orthogonal to the rest of the development, it is possible to add this model to other approaches as well. To manage the development, the analysis model may group objects in subsystems.

The analysis model comprises a total functional specification of the system we wish to develop, without any attention to the implementation environment. In the construction process we will construct the system from the analysis model. This is when adaptions are made to the implementation language, the database management system, the architecture of the computer system and so on. Thus the design model and the analysis model are both models of the system we wish to build, but with various purposes. From an object in one model we can directly trace to an object in the other model, and *vice versa*. We call this **traceability**.

7.1.3 An example system

Throughout the discussion of the analysis and the construction activities, we will show how the different concepts are used in practice, by developing a system. The system controls a recycling machine for returnable bottles, cans and crates (used in Europe to hold several bottles). The machine can be used by several customers at the same time, and each customer can return all three types of items on the same occasion, see Figure 7.2.

Since there may be different types and sizes of bottles and cans, the system has to check, for each item, what type was turned in. The system will register how many items each customer returns, and when the customer asks for a receipt, the system will print out what he turned in, the value of the returned items and the total return sum that will be paid to the customer.

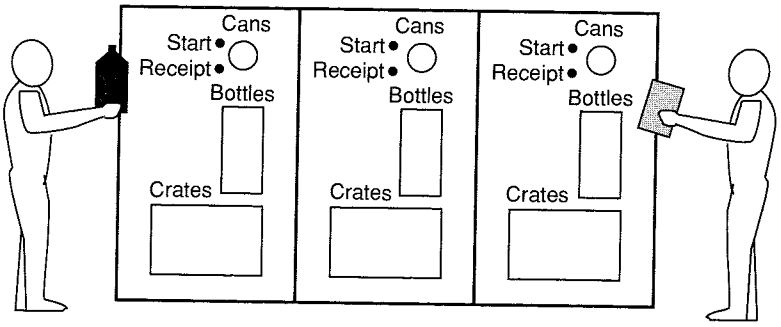


Figure 7.2 The recycling machine can receive several different types of returnable items from several customers at the same time.

The system is also used by an operator. The operator wants to know how many items of each type have been turned in during the day. At the end of the day, the operator asks for a printout of the total number of items that have been turned in to the system on that particular day. The operator should also be able to change information in the system, such as the deposit values of the items. If there is something amiss, for instance if a can gets stuck, or if the receipt roll is finished, the operator will be called by a special alarm signal.

This example system is chosen as a case study that is easy to describe and to understand; it is not intended as an example of a good recycling machine. The system should be viewed as a toy example and should not be used as a basis for real system modeling. The descriptions that we give are too simple to be used in system design; nor should they be used as patterns. The system is small, so the models will be small; consequently, all the properties of OOSE will not be fully obvious in this example. Later chapters of the book contain examples of OOSE used in larger systems, and they will therefore give a clearer picture of how OOSE is used in larger developments.

7.2 The requirements model

7.2.1 System development based on user requirements

The requirements model aims at delimiting the system and defining what functionality the system should offer. This model could function as a contract between the developer and the orderer of the system and thus forms the developers view of what the customer wants.

Therefore it is essential that this model should be readable also for non-OOSE practitioners.

This model will govern the development of all the other models, so this model is the central one throughout the whole system development. The requirements model will be structured by the analysis model, realized by the design model, implemented by the implementation model and tested in the testing model. Not only will the other models be verified againts the requirement model, but the other models will actually be developed directly from this model. The requirements model will also function as a basis for the development of operational instructions and manuals, since anything that the system should do is described here from a user's perspective. The requirements model, and especially the use cases, will be the unifying thread running through the whole of OOSE.

Since the whole system development starts from what the users wish to be able to do with the system, we build the system from the users' point of view. In this way, it will be easy to discuss the requirements model with the users, and changes of the model will be simple to make.

The requirements model consists of three parts; the use case model, the problem domain object model and user interface descriptions. The use case model specifies the functionality the system has to offer from a user's perspective and we define what should take place inside the system. This model uses **actors** to represent roles the users can play, and **use cases** to represent what the users should be able to do with the system. Each use case is a complete course of events in the system from a user's perspective. If appropriate, interface descriptions may also be developed. These will specify in detail what the user interface will look like when the use cases are performed. To give a conceptual picture and a better understanding of the system, we use objects that represent occurrences in the problem domain. This model will serve as a common foundation for all the people involved in the requirements analysis, developers as well as orderers.

7.2.2 Actors

To be able to identify what use cases are to be performed in the system, we will identify the users of the system. This is done by means of **actors**. Actors model the prospective users; the actor is a user type or category, and when a user does something he acts as an occurrence of this type. One person can instantiate (play the roles of) several different actors. Actors thus define roles that users can play.

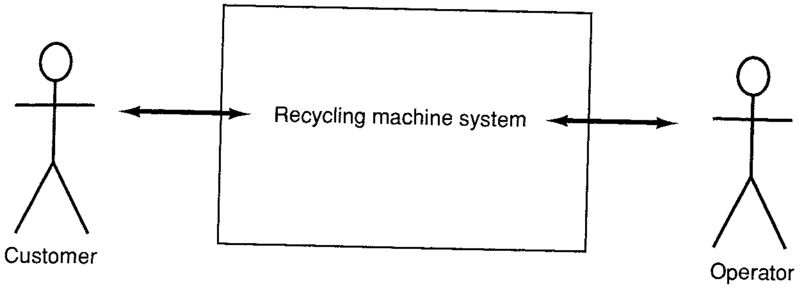


Figure 7.3 The recycling machine has two actors: the *Customer* and the *Operator*

The actors model anything that needs information exchange with the system. Actors can model human users, but they can also model other systems that are going to communicate with the system. The essential thing is that actors constitute anything that is external to the system we are to develop.

Thus we will have to make a **system delimitation** to define where the limit is between actors and use cases. We will identify an actor as everything external that is to communicate with the system.

In our example with the recycling machine we can see from the problem description that we have two actors, namely *Customer* and *Operator*. These were the only ones who should interact with the system. They interact with the system in different ways: the *Customer* turns in bottles and suchlike and receives a receipt from the machine, while the *Operator* maintains the machine and takes out daily reports, see Figure 7.3.

To stress the difference between actor and user, we think of an actor as a class, that is, a description of a behavior. A user, on the other hand, can play several roles, that is, serve as many actors. For instance, think of Brian. He is the operator of the machine, he normally acts as an instance of the *Operator* actor. However, sometimes he deposits his own bottles and cans, and then he acts as an instance of *Customer*.

To find actors can entail some work, and all the actors are seldom found at once. A good starting point is often to check why the system is to be designed. Who are the actors that the system is supposed to help? The actors who are going to use the system directly (maybe in their daily work) we call **primary** actors. Each one of these actors will perform one or some of the main tasks of the system. In our example, *Customer* is a primary actor, for it is for the customers that the system is built.

As all the important functions of the system are investigated, more and more primary actors are identified. Beside these primary actors there are actors supervising and maintaining the system. We call them **secondary** actors. The secondary actors exist only so that the primary actors can use the system. In our example, *Operator* is a secondary actor, for we would hardly need an operator if there were no customers

We make the division into primary and secondary actors because we want the system structure to be decided in terms of the main functionality. The primary actors will govern the system structure. Thus, when identifying use cases, we will also start with the primary actors. In this way, we can guarantee that the architecture of the system will be adapted to the most important users. Changes of the system will mainly come from these actors.

It is often simple to find actors that model people. It is, however, more difficult to identify those which are machines, but as the use cases are specified, it will be clearer that they are necessary. These actors can of course be both primary and secondary actors.

Normally, we do not view functionality from the underlying system as actors. For instance, a system clock is normally not modeled as an actor. However, nothing stops us from modeling this as an actor. In some cases, for instance where the underlying system has an active role against the application, it is very meaningful to model it as an actor.

7.2.3 Use cases

After we have defined what is outside our system we can define the functionality inside it. We do this by specifying **use cases**. A use case is a specific way of using the system by using some part of the functionality. Each use case constitutes a complete course of events initiated by an actor and it specifies the interaction that takes place between an actor and the system. A use case is thus a special sequence of related transactions performed by an actor and the system in a dialogue The collected use cases specify all the existing ways of using the system.

To understand use cases we can view their descriptions as state transition graphs. Each stimulus sent between an actor and the system performs a state change in this graph. We can thus view a use case as existing in different states. The related transactions mentioned above are thus the transitions in this graph.

The reason that we have identified actors initially is that they will be the major tool for finding the use cases. Each actor will perform a number of use cases to the system. By going through all actors and defining everything they will be able to do to the system, we will define the complete functionality of the system.

Like actors, use cases can be instantiated and this is done every time a user performs a use case in the system. Thus there is also here a relationship between class and instance. This topic was discussed in greater depth in Chapter 6

As several use cases can begin in a similar way, it is not always possible to decide what use case has been instantiated until it is completed Consider for instance a telephone exchange system. One actor here is a subscriber, and a typical use case is to make a local telephone call This use case starts when the subscriber lifts his telephone. Another use case is to order a wake-up call. Both use cases starts when the subscriber lifts his telephone. But when a subscriber lifts his telephone we do not know which use case he wants to perform. Thus use cases may start in a similar manner and we may not know which use case was done until it is over. In other words, the actor does not demand that a use case should be performed; he only initiates a course of events that will finally result in a complete use case.

The use cases are identified through the actors. For every complete course of events initiated by an actor we identify one use case. By viewing each actor, we decide which use cases this actor is supposed to perform To identify the use cases, we can read the requirement specification from an actor's perspective and carry on discussions with those who will act as actors. It will help to ask a number of questions, such as:

- What are the main tasks of each actor?
- Will the actor have to read/write/change any of the system information?
- Will the actor have to inform the system about outside changes?
- Does the actor wish to be informed about unexpected changes?

In the recycling system we can identify a number of use cases. Let us use the actors as a starting point The *Customer* is the primary actor so let us start with her. She should of course be able to return deposit items. This forms one use case, *Returning Item*. This use case should include everything until a receipt is received to be complete. Are there any more use cases for Customer? When an item is stuck is an alternative for this use case and we will discuss later how to handle alternatives. Otherwise, any more use cases are at the moment not obvious, so let us continue to the *Operator* The *Operator* is a secondary actor He should be able to get a daily report of what items have been deposited today. This will be one use case, call it *Generate Daily Report*. He should also be able to modify information

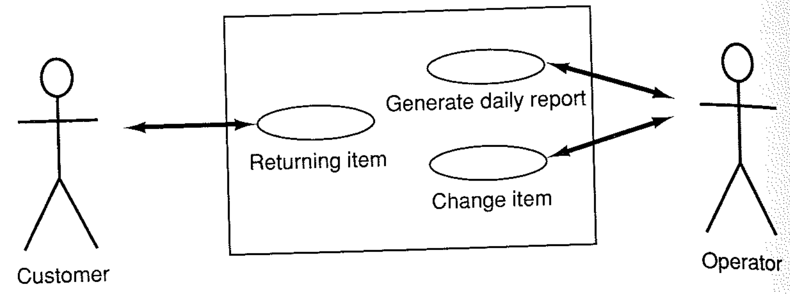


Figure 7.4 The first attempt of the use case model for the recycling machine.

in the system, such as each item's deposit value. Let us call this use case *Change Item*. A use case is drawn as an ellipse with its name beneath the ellipse, see also the side box on notation. We thus now have the use case model as illustrated in Figure 7.4.

The use cases can be summarily described as follows:

> *Returning Item* is started by *Customer* when he wants to return cans, bottles or crates. With each item that *Customer* places in the recycling machine, the system will increase the received number of items from *Customer* as well as the daily total of this particular type. When *Customer* has turned in all his items, he will press the receipt button to get a receipt where the returned items have been printed as well as a total return sum.
>
> *Generate Daily Report* is started by *Operator* when he wants to print out information about the returned deposit items of the day. The system will print out how many of each deposit item type have been received this day, as well as the overall total for the day. The total number will be reset to zero to start a new daily report.
>
> *Change Item* is used by *Operator* to change information in the system. The return value as well as the size of each returnable item can be changed, and new types of items can be added.

On the notation convention

The notations used in figures has the convention of one standard size for each object type and the name of the object beneath the symbol. Additionally, relations are expressed in the figure. However, no other information is normally introduced in the figure. Some other methods introduce much more information in a figure, such as having information inside the symbols.

We have chosen this notation so as to have a changeable and surveyable picture. If we had information inside the symbols, these would need to be extended in size when we add more information This extension affects the picture and may force us to move the other objects as well. Hence the picture then depends on how far the details of the model have evolved Additionally, as we add more and more information it is very easy to lose the surveyability of the model.

Instead we think of the figure as a graphic view of information concerning the model in some information bank (database). We therefore assume a CASE tool to handle this information bank. To add information to the model, we would just like to point at a specific object and retrieve some information aspect on this object, such as a use case's detailed description or operations on an object In this way there is no need to have different sizes of objects, but rather we can preserve a view that does not change when we change information from some other view.

It is not always obvious what functionality should be placed in a separate use case, and what is only a variant of one and the same use case The complexity of the use cases is important in this context. We have a number of different ways of expressing variants. If the differences are small, they can be described as variants of the same use case, either as just a selection in a use case or as separate variants of a use case. If the differences are large they should be described as separate use cases For instance, remember the use case telephone call. If the person being called does not answer, is that another use case than if he answers? It will surely be different behavior In this case we would regard them as variations of the same use case since they are logically highly correlated.

We have as yet only discussed the identification of use cases. This is often a very iterative process where several attempts are often made. When the picture stabilizes, each use case should be described in more detail. Then we describe the **basic course** which is the most important course giving the best understanding of the use case. Variants of the basic course and errors that can occur are described in **alternative courses**. Normally a use case has only one basic course, but several alternative courses. The descriptions could be made earlier, but since the requirements model will undergo several changes initially, too much work should not be done early on, that will be thrown away.

Thus we will describe each use case in more detail Below is the flow description of the use case *Returning Item*:

> The course of events starts when the customer presses the 'start-button' on the customer panel The panel's built-in sensors are thereby activated

The customer can now return deposit items via the customer panel The sensors inform the system that an object has been inserted, they also measure the deposit item and return the result to the system.

The system uses the measurement result to determine the type of deposit item: can, bottle or crate

The day total for the received deposit item type is incremented, as is the number of returned deposit items of the current type that this customer has returned

When the customer has returned all his deposit items he asks for a receipt by pressing the 'receipt-button' on the customer panel.

The system compiles the information that is to be printed on the receipt For each type of deposit item, its return value and the number of returned items by the current customer is extracted.

The information is printed out, with a new line for each deposit item, by the receipt printer

Finally, the grand total for all returned deposit items is extracted by the system and printed out by the receipt printer

When analyzing and describing the use cases in detail, the system is studied very closely. It is not unusual then that unclear points in the requirement specification are revealed in this process. Vagueness in the requirement specification thus becomes obvious at a very early stage.

Since the use cases often focus on a certain functionality of the system, it is possible to analyze the functionalities in an incremental way. In this manner we can develop use cases for different functionality areas initially and later join these use cases together to form the complete requirements model. We can thus focus on one problem at a time and analyze it, and later bring the results together. Hence the entire problem must not be taken into consideration at the same time, but it is possible to take one part at a time and thus possibly also do parallel development.

Extends

A powerful concept that is used to structure and relate use case descriptions is the **extend** association. Extend relates how one use case description may be inserted into, and thus extend, another use case description. In this way extensions of use cases can be described in a very simple way and, especially, changes and additions of functionality are more easily made.

The use case where the new functionality should be inserted, must be a complete course in itself. Hence this description is entirely independent of the inserted course. In this way, the description pays no attention to any courses that can be inserted, and this complexity is thus avoided. We thus describe the first basic use cases totally

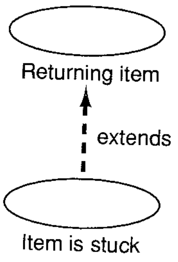


Figure 7.5 *Item is Stuck* is inserted into *Returning Item* when a deposit item gets stuck.

independently of any extended functionality. Similarly, we can add new extensions without changing the original descriptions.

In our example we can see one use of the extend concept. When an item is stuck the system should issue an alarm. Hence we can describe this alarm as a use case description that extends the *Returning Item* use case description. This new use case we call *Item is Stuck*, see Figure 7.5. The extend association is drawn with a dashed arrow since it is a class association (concerns the descriptions). Instance associations, as mentioned, are drawn with full lines. We have actually already seen such associations in the use case model, namely in the figure where actors communicated with use cases. This is an instance association since it is always an instance of an actor that will communicate with an instance of a use case.

Thus *Returning Item* is described entirely independently of this new flow, which makes the description simple. The use case *Item is Stuck* can be described as follows

> *Item is Stuck* is inserted into *Returning Item* when *Customer* deposits an item that gets stuck in the recycling machine. *Operator* is called and *Customer* cannot turn in more items until *Operator* informs him that the machine can be used again.

What we see here is another use case property: they can communicate with several different actors. *Customer* will start the use case, but *Operator* will also communicate with it.

By means of the extend association the system will be given a good and modifiable structure, as we will see later. As it is possible to describe several use cases independently of each other, their descriptions will be simple. To understand extend better we can view a typical situation where we have a simple use case Login/Logout, and describe all use cases that may be done, when a user is logged

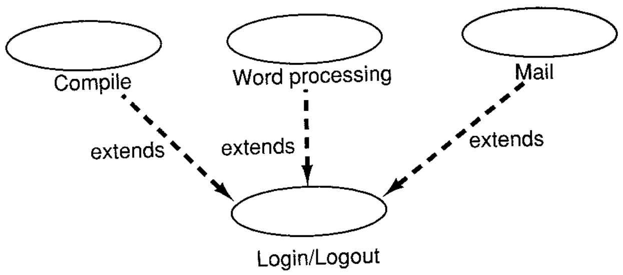


Figure 7.6 A common situation is having a use case Login/Logout into which several other use cases can be inserted.

in with extend to the use case Login/Logout. It will then be simple to add new options without having to make changes in the old use cases; new use cases are simply defined which can be inserted into Login/Logout, see Figure 7.6. (This situation is so common that models often assume this functionality without specifying it.)

Another example of use of the extend association is when you want to collect statistics in a specific use case. Say, for instance, that you have a use case that controls an industrial process. Everytime you execute this use case, you also want to measure and collect some statistics in this process. This you can describe by having a use case *Collect Statistics* as an extension-use case to the control use case.

Extend is thus used to model extensions of other, complete use cases. Here are some examples of when to use extend:

- To model optional parts of use cases,
- To model complex and alternative courses which seldom occur, for instance *Item is Stuck* in our example,
- To model separate subcourses which are executed only in certain cases,
- To model that several different use cases can be inserted into a special use case, such as Login/Logout mentioned above or a menu system with options

We might view the extend association as an interrupt in the original use case when the new use case is to be inserted. The original use case does not know whether an interrupt will happen or not.

For each use case that is to be inserted into another use case, we state the position in the use case where it is to be inserted. This position is stated as exactly as possible, and is described in the inserting use case (not in the original use case). The position is expressed as a reference to a place in the description.

What happens when a course is inserted in this way is as follows. The original use case runs as usual up to the point where the new use case is to be inserted. At this point, the new course is inserted. After the extension is finished, the original course continues as if nothing happened. When we described *Item is Stuck* we were somewhat inaccurate: the use case is not inserted only when an item gets stuck, but the insertion always takes place. Actually it is always checked whether an item has got stuck. If so, the whole course is initiated; otherwise the original course *Returning Item* continues directly.

7.2.4 Interface descriptions

When describing the use cases and communicating them to potential users, it is often appropriate to describe the interfaces in more detail. If it is an MMI (man–machine interface) we can use sketches of what the user will see on the screen when performing the use case or more sophisticated simulations using a UIMS (user interface management system). In this way we can simulate the use cases as they will appear to the users before even thinking of how to realize them. We can thus liven up the use case descriptions with real computer interaction by the potential users. Such a technique will eliminate several possibilities of misunderstandings. If it is hardware protocols, we can refer to different standards. These interface descriptions are thus an essential part of the use case descriptions and should accompany them.

When designing user interfaces, it is essential to have the users involved in this development. By doing a user interface design at this early stage, this can be guaranteed. Additionally, when designing user interfaces, it is essential that the interface as such reflects the user's logical view of the system. The problem domain model (discussed next) is exactly such a perspective. By using this model when developing the user interface as a conceptual base to define concepts and semantics of the system, we can guarantee that the user interface will be consistent with the user's logical system perspective. This is actually one of the most fundamental principles of human–interface design; the consistency between the user's conceptual picture of the system and the system's actual behavior.

It is not only user interfaces that are interesting to specify in detail at this early stage Since the requirements model may work as a functional requirement specification, it may at this stage also be interesting to define other system interfaces such as communication protocols that should be standardized, for instance Hence these interface descriptions may also take the form of protocols to other systems.

In the recycling machine the interfaces are quite trivial (being mainly a push-button machine), so we do not go into any detail for the interface description in this example. However, in the case study chapters later we will discuss this issue more and give examples of the development of system interface descriptions, and especially user interfaces, accompaning the requirements model. We will also illustrate how the problem domain model will be used when developing the user interfaces.

7.2.5 Problem domain objects

When working with the requirements model it can sometimes be elaborate to define the task of the system and especially the system delimitation. This is typically the case when the requirements specification exists only in a very vague form. Then, a very good tool is to start to develop a logical view of the system using problem domain objects, that is, objects that have a direct counterpart in the application environment and that the system should handle information about

Such a problem domain model will be a strong support also when specifying the use cases. Then, this model will define the concepts that the system should be working with. In this way we will have a glossary that can be used to formulate the functionality of the use cases. In the recycling system we see that we have used the words 'returnable item', 'can', 'bottle' and 'crate' quite extensively without really defining them. When several people are involved in the specifications of the use cases, such a problem domain model can thus be of great value

The major benefit of such a model, though, is that it is a very good tool with which to communicate about the system. Since the users and orderers will recognize all the concepts, the model can be used when defining what the system will do A technique we have used when working with such a model is to give the customer a pen and paper and ask him to draw a picture of his view of the system. By reasoning with him, a quite extensive problem domain model will evolve In this way we will also have a common terminology when reasoning about the use cases, and so lessen the probability of misunderstandings between the developer and the potential user.

Many other object-oriented methods, such as those of Coad and Yourdon (1991) and Booch (1991), focus entirely on such models,

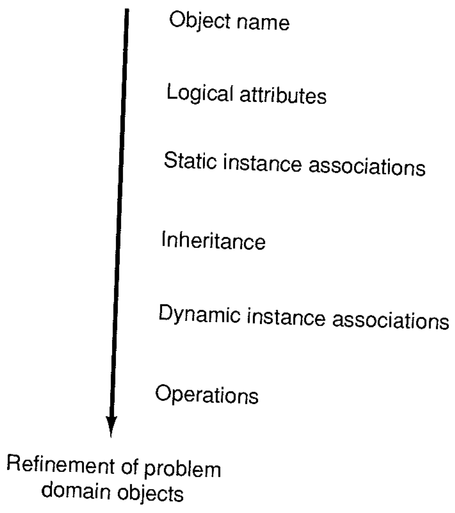

Object name
Logical attributes
Static instance associations
Inheritance
Dynamic instance associations
Operations
Refinement of problem domain objects

Figure 7.7 Suggested possibilities to refine the problem domain objects By static instance associations we mean associations that are used when an object should statically know of another object, namely a static reference to an object. Dynamic instance association means such associations where one object can send stimuli to another object, which implies a dependence on the other object's protocol

and the heuristics in these methods can very well be used for the identification of these objects In these methods, this model will also form a base for the actual implementation, that is, these objects are directly mapped on classes during implementation. However, this is not the case in OOSE, as discussed in the previous chapter. Our experience with such an approach tells us differently. Instead we develop the analysis model that is more robust and maintainable for future changes, rather than using a problem domain model to serve as the base for design and implementation.

Then how extensive should a problem domain model actually be? In Figure 7.7 we have illustrated different possible degrees of refinement

Since the main purpose is to form a common base of understanding to develop the requirements model, and not to entirely define the system, we believe that the object name and possibly also the logical attributes and the static instance associations (i.e the static references between these objects) are an appropriate level to stay at. However, it is of course possible to do even further refinements of these objects if it helps understanding and thus specifying the functionality of the system completely using these objects. It is thus fully possible to express the entire functionality as behavior associated to these objects, but this will not yield the most robust and extensible structure for the system. Keep in mind that too much work here may

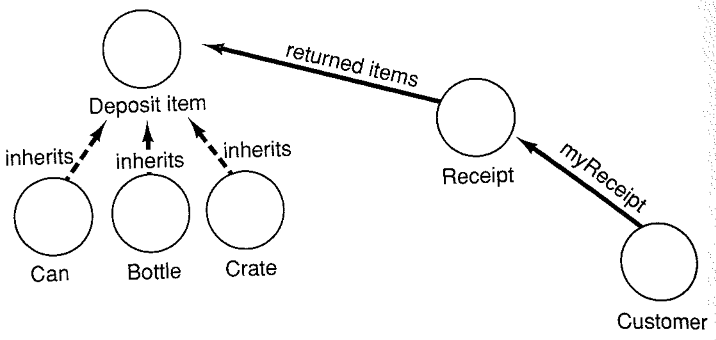


Figure 7.8 A problem domain model of the recycling machine.

result in it being hard to free yourself from this structure when developing the more stable and maintainable analysis model. Experience shows that many (if not all) of these domain objects will show up as entity object in the analysis model. However, it is dangerous to do this mapping mechanically, since there may very well be changes (e.g. if it is obvious that information of an object is actually not needed in the system) Additionally, often there will be even more entity object in the analysis model than objects in the problem domain model.

In the recycling system, a problem domain model would be quite small since it is such a trivial system. The main concepts used are *Can, Bottle* and *Crate*. These will all be handled in the same manner and we can thus identify an abstract class *Deposit Item*. Since the customer will have a receipt printed when all items have been returned, we may also need to be able to manage the object *Receipt* and *Customer*. We will now have the domain model showed in Figure 7.8.

However, in larger systems its use becomes more obvious. In the example chapters later, we illustrate the use of a problem domain model as a support for understanding the system in more detail.

This problem domain model can be used for several different purposes. We have discussed above its role as a support for the formulation of the use case descriptions and also for the MMI design. We can also elaborate this model in more detail to gain a better understanding of the system. In this way we can focus more on the problem domain objects. Hence we can then fully specify the functionality of the system using such a model including formal operations on objects. In that case we can also use it as a requirements model expressed without use cases However, when continuing the development with the method presented here, the use cases will be of much more help in the forthcoming work, and therefore it is recommended to elaborate these in detail instead.

Another use of a problem domain model is when doing enterprise modeling. Then it is essential to capture all the important and fundamental concepts in one model. Actually, the extension of OOSE to include enterprise modeling will deliver such a problem domain model as output, i.e. input to the system development process. Accompanying this will be a first version of a requirements model of the systems required in the enterprise. In such a way the transition between enterprise modeling and system development can be done in a seamless way.

This means that a problem domain model, developed with some technique, will serve as a very solid input to system development. In some projects we have been involved in, there already existed such a problem domain model. Then the idea of the system to be developed was quite mature, and the development of the requirements model and the analysis model could be done in a quite straight-forward manner without too many iterations

7.2.6 Further refinement of the requirements model

The requirements model as described thus far will be sufficient to specify the functionality of the system. However, we can elaborate this model further not only to enhance reuse, but also to prepare for the transition to the analysis model. This work is thus not really very interesting to the orderer of the system.

This refinement is mainly done by identifying similar parts of the use cases and extracting these similar parts. In this way we only have to describe the similar part once instead of in all use cases showing this behavior. Any changes to this part will thus automatically affect all use cases that share this part. Such use cases that we extract, we call **abstract** use cases since they will not be instantiated on their own, but are only meaningful to describe common parts between other use cases. The use cases that really will be instantiated we call **concrete** use cases.

The descriptions of the abstract use cases are thus used in the descriptions of the concrete use cases. This means that when an instance of a use case follows the description of a concrete use case, at a certain point it continues to follow the description of the abstract use case instead. This relationship is thus some kind of inheritance. However it does not have exactly the same semantics as inheritance has in an object-oriented programming language. Therefore we give it a different name. We call this relation a **uses**-relation. Intuitively,

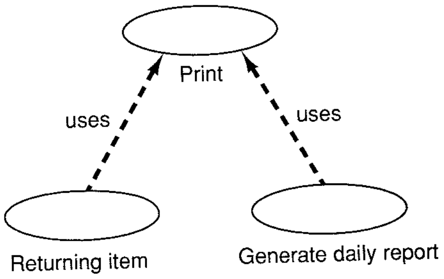


Figure 7.9 An abstract use case *Print* has been identified to describe common parts between two other use cases

this is also easier to understand than inheritance as used in an OO language, since it is not discrete operations that are used, but rather sequences. Therefore these sequences may have to explicitly be interleaved in the concrete use cases. Since it is a class-association it is drawn as a dashed arrow drawn from the concrete towards the abstract use case that is to be used.

Normally, similar behavior between use cases are identified *after* the use cases have been described. However, in some cases it is possible to identify them earlier. In the recycling example we see that the two use cases *Returning Item* and *Generate Daily Report* will both print out a receipt. We can thus identify an abstract use case *Print* that performs this printing, see Figure 7.9.

The reuse between use cases can thus also be multiple uses. Several parts may have been extracted from a use case that are common with other use cases. One specific use case can then use all these abstract use cases.

In use case decomposition, the entire course is always used. The course need not be an atomic sequence, although this is often the case. However, we may have a situation where the use cases can be used through interleaving into the concrete use case, see Figure 7.10. In the left-hand side abstract use case we have a sequence that consists of the subsequences A and B These sequence should be integrated with subsequences C and D as illustrated in Figure 7.10.

Abstract use cases can be used by other abstract use cases. It is difficult to state exactly when there is no point in extracting more abstract use cases. A good rule of thumb is that when the level of separate operations on objects has been reached, you ought to stop. The effort to find common sequences has then been carried too far. With some experience, it will soon become easy to discern where the limit should be drawn.

A technique to extract abstract use cases is to identify abstract actors. Such an abstract actor typically describes a **role** that should

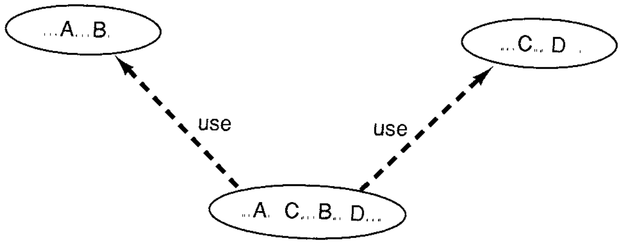


Figure 7.10 A concrete use cases uses the two abstract use cases and decides explicitly how the interleaving is to take place

be played against the system. When different actors thus should be able to play similar roles they may inherit a common abstract actor. The advantage of abstract actor modeling is that it expresses similarities in use cases. If the same (part of) use case should be performed by several different actors, the use cases then need to be specified only with respect to one actor instead of to several.

In the recycling system the actors actually have one common behavior: they can receive a receipt It is therefore possible to identify one abstract actor, *Receipt Receiver*, which both *Customer* and *Operator* inherit, see Figure 7.11. Now both our concrete actors can receive a

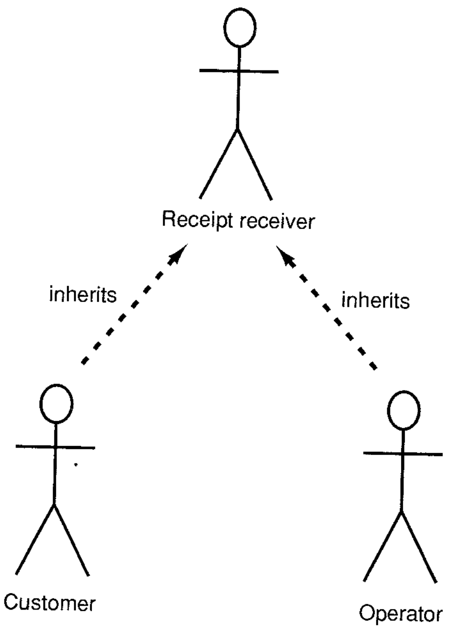


Figure 7.11 *Receipt Receiver* is an abstract actor which is inherited by both *Customer* and *Operator*.

receipt and this needs to be specified only once

Abstract actors can also be used to specify different privileges in a system. An example of this will be shown in Chapter 13.

The use-association is thus used when two or more use cases have common behavior. Normally, there is no reason to create abstract use cases that are used only by one use case. However, in fact we had this situation with the extend association. An extend use case may be viewed as an abstract use case since they are seldom meaningful on their own.

Since the extend and uses between use cases are both class associations, you may ask when should uses be chosen rather than extends and vice versa? In fact, in most cases the choice is quite obvious, and causes no problems. An important criterion is how strongly functionally coupled the use cases are. If the course to be extended is an independent course in itself, and the course has very little to do with what it is inserted into, extend should be chosen. If, on the contrary, the courses are strongly functionally coupled, and the insertion must take place every time to obtain a complete course, use should be chosen There is also a difference in how they are found; uses is found through extraction of common sequences from several different use cases, whereas extends is found when new courses are introduced, or when there are extensions to an existing use case that the user wishes to perform in some specific cases. You can be deceived by this rule, since extend use cases can also be common to several use cases.

7.2.7 Discussion

As has been pointed out several times, the use case is the core running through all OOSE activities when developing all models. Our experience with use cases is that they help focus on the problem, as they constitute a strong tool to define the system functionality. Additionally, since they are logical and straightforward to find, they also force the development forward as the use cases are identified and specified They will also be a strong support when developing the subsequent models since these models are based upon the use cases. In this way, a disciplined way of working is natural as you control the work by the use cases. It is also possible to estimate the work in subsequent models since you know how many use cases you have and you can predict the time to handle one use case. This topic is further discussed in Chapter 15

When working with use cases, a common question is how complete the use cases should be. For instance, in the recycling

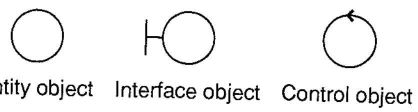


Figure 7.12 The object types used to structure the system in the analysis model

example, we could have viewed the sequence when the customer pushed the start button and inserted the items as one use case and the sequence when he pushed the receipt button as another use case. But we chosed to have them as one complete use case instead. Generally, it is better to have longer and more extensive use cases than several smaller ones. We thus want complete and real courses and not several subcourses. Otherwise there will be very many use cases that are hard to overview and also it will be harder to see how they are related, in time, for instance This is discussed in more detail in Chapter 13.

7.3 The analysis model

When the requirements model has been developed, and often also signed by the orderers, we can focus on the structuring of the system. This is initially done by developing the analysis model. In the analysis model we describe the system using three different types of objects: **interface objects, entity object** and **control objects**, see Figure 7.12. Each of these objects has its own purpose and will model one specific aspect of the system. We also use subsystems to group these objects into manageable units.

In the requirements model we have specified what is to take place within the system. The analysis model aims at creating a good platform for the system design and will also form the basis of the design The requirements model is thus structured by the analysis model.

The work to develop the analysis model really entails distributing the behavior specified in the use case descriptions among the objects in the analysis model, see Figure 7.13 An object can be common for several different use cases. Thus we should state explicitly which object is responsible for which behavior in the use case This does not mean that the behavior must be broken down into operations at this stage, although this is possible. Instead, a more natural procedure is to write a verbal description of the responsibilities of or roles played by each object.

We will now study the different object types more closely and discuss how it is possible to find them from the use cases

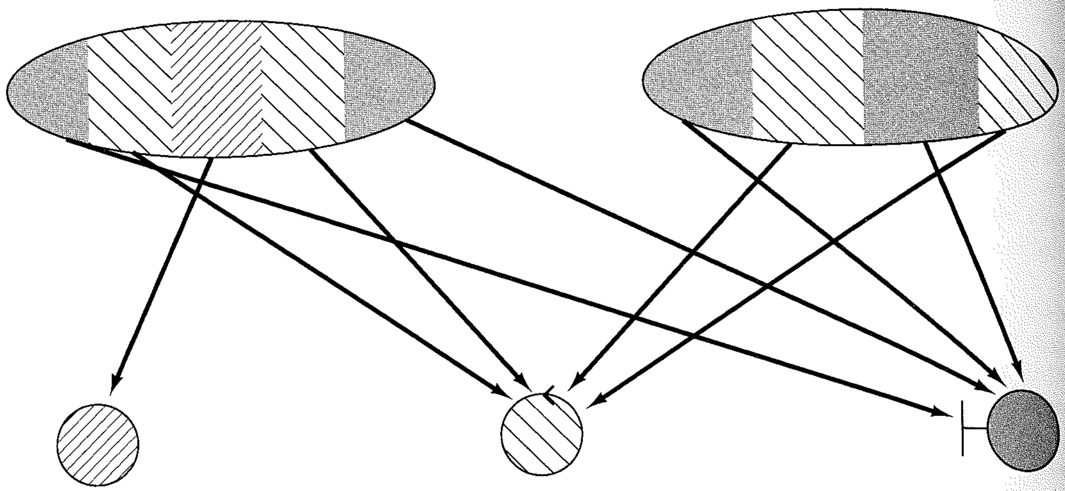


Figure 7.13 Each use case is distributed among analysis objects Several use cases can have objects in common.

7.3.1 Interface objects

All functionality specified in the use case descriptions that is directly dependent on the system environment is placed in **interface objects**. It is through these objects that the actors communicate with the system The task of an interface object is to translate the actor's actions to the system into events in the system, and to translate those events in the system that the actor is interested in into something which is presented to the actor. Interface objects can, in other words, describe bidirectional communication between the system and its users.

Interface objects are quite simple to identify. We have at least three strategies Either they are clearly identified from the system interface descriptions accompanying the requirements model, or we can start from the actors, or we can read the use case descriptions and extract the functionality that is interface specific. Let us initially use the second alternative, namely to start from the actors.

Each concrete actor needs its own interface for its actions on the system. In many cases an actor may need several interface objects. In the example with the recycling machine, each of the concrete actors, *Customer* and *Operator*, needs its own interface object to the system. The *Customer* needs the panel with the push-buttons and the slots in which to insert the items and the *Operator* needs his interface to be able to change information in the system and to generate daily reports Furthermore, we need an interface object to call the *Operator*

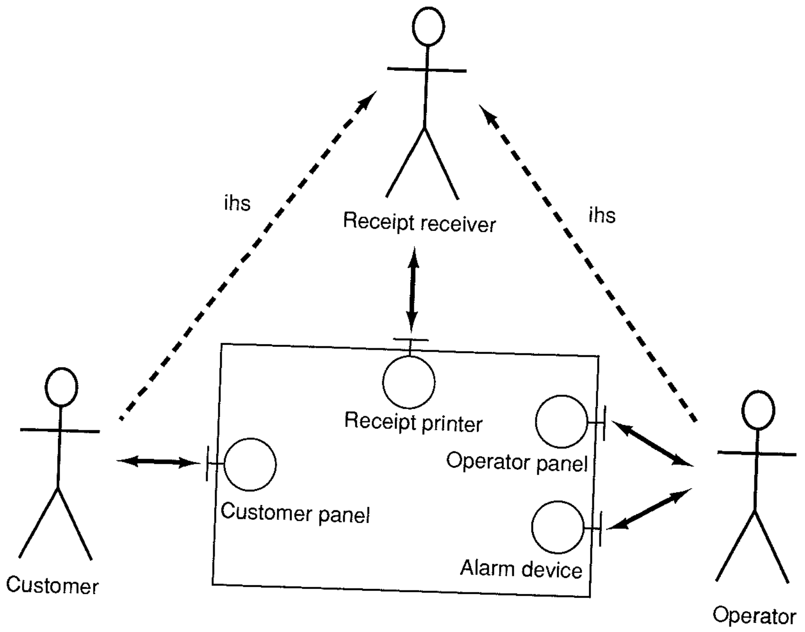


Figure 7.14 We have four interface objects in the recycling machine: *Customer Panel, Operator Panel, Receipt Printer* and *Alarm Device.*

when an alarm is issued, as well as one to print out a receipt. We thus have the interface objects as shown in Figure 7.14. Finding an interface object for an abstract actor, as in this case, does not always occur

Let us look at the third strategy, namely to identify interface objects from the use case description. We have marked the places in the use case description below where interface functionality are involved. Note that the use case description has been taken directly from the requirements model.

The course of events starts when the customer presses the 'Start-Button' on the customer panel The panel's built-in sensors are thereby activated

The customer can now return deposit items via the customer panel The sensors inform the system that an object has been inserted, they also measure the deposit item and return the result to the system.

The system uses the measurement result to determine the type of deposit item: can, bottle or crate.

The day total for the received deposit item type is incremented, as is the number of returned deposit items of the current type that this customer has returned

When the customer has returned all his deposit items *he asks for a receipt by pressing the 'Receipt-Button' on the customer panel.*

The system compiles the information to be printed on the receipt For each type of deposit item, its return value and the number of returned items by the current customer is extracted.

The information *is printed out, with a new line for each deposit item, by the receipt printer*

Finally, the grand total for all returned deposit items is extracted by the system and *printed out by the receipt printer*

We see that this technique yields the same interface objects, namely *Customer Panel* and *Receipt Printer* for this use case.

The following is a short description of the interface objects identified

The *Customer Panel* is the interface that *Customer* uses for his or her actions to the recycling machine The panel has a start button and a button for receipt request The panel also has a slot for each type of returnable item where there are sensors noting what kind of item the *Customer* turns in The sensors also measure the size of each returned item

The *Operator Panel* is the panel that *Operator* uses to change information in the system Via the panel, *Operator* can also order the daily total receipt

The *Alarm Device* controls a loudspeaker which emits a signal It can also receive simuli from a button which states that this signal can be turned off

The *Receipt Printer* controls a small printer which writes text on a paper roll After a printout, the paper is cut off When the paper roll is almost finished, the operator should be called.

On naming associations

It is evident that the interface objects are not entirely independent of each other, but that they must know of each other to be able to solve certain tasks For instance, the *Receipt Printer* must know which *Alarm Device* to sound when the paper roll is finished. This is solved by the introduction of **acquaintance associations** between the objects. An acquaintance association is a static association between instances and means that an instance knows of the existence of another instance. It does not give the object the right to exchange information with the other object; then a dynamic association is needed as will be discussed below. Instance associations are, as mentioned, drawn with solid directed lines.

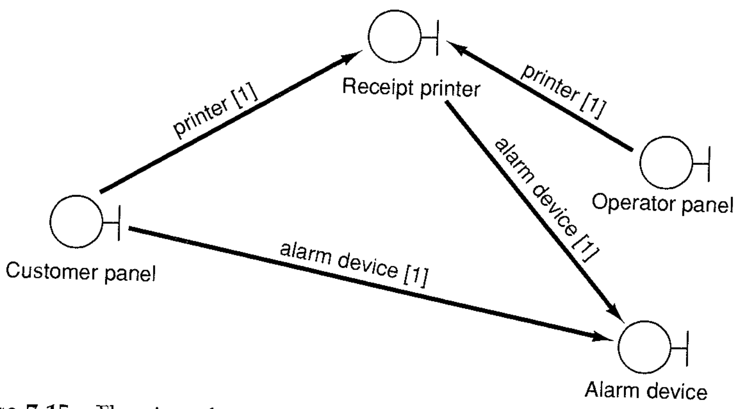


Figure 7.15 The interface objects and aquaintance associations of the recycling machine.

An object may associate several other instances of the same class. Therefore we should also describe how many instances can be associated with the acquaintance association. This is done by assigning a **cardinality** to each association. This cardinality says how many instances can be associated. We also give the acquaintance association a name clarifying what the relationship entails. The properties of these associations and naming conventions are discussed in the box on naming associations. A complete view of the interface objects in the example is shown in Figure 7.15. The cardinalities are in this example all [1] since the different interface objects can only know of one object each. Other examples of cardinalities are [0..N] which means that we might associate any number between 0 and *N*, and so on. An acquaintance association is thus a static instance association that is drawn with a solid, directed line having a name and a cardinality. Note that the associations are only uni-directional.

On naming associations

Models including relations often represent a model of the real world. It is common to name associations so that the sequence object–relation–object naturally forms a sentence. Consider for instance the relation shown in Figure 7.16. Here a verb phrase is used and we

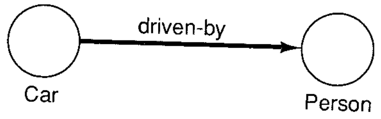


Figure 7.16 An example of how an association could be named using a verb phrase.

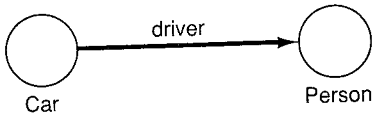


Figure 7.17 An example of how an association could be named using a noun phrase

can read 'A *Car* is *driven by* a *Person*' This is very convenient

However, another way of expressing the same relation is to express what **role** another object plays in relation to the first object. Then instead a noun phrase should be used as shown in Figure 7.17. Here it is harder to read directly a complete sentence, but instead we can use the role played to express the relation 'The object *person* plays the role of a *driver* to *Car*.'

We see that both of these strategies will be appropriate in this example In the literature, often the first verb phrase strategy is chosen, and it is also often used as an argument of how useful object orientation is; you can directly read sentences from the model Actually, this is not unique for object orientation The same technique is very often used in data modeling, see Barker (1989)

However, we would like to promote the second solution instead; the noun phrase The reasons for this are as follows.

(1) There is a large and fundamental difference between data modeling and object orientation. In data modeling you think of the model as a flat structure viewed from above In this way it is natural to see the relation as a binding between two objects. The relationship here is often bidirectional. In object orientation however, the view is instead made from an object When you regard this model, you place yourself at an object and then see what references you have to other objects. In this way you want to see what roles exists around you Here the relationship is unidirectional This fundamental difference is often hard to get used to for people used to data modeling. But object orientation often models the reality, so think about how the world around you looks As one Objectory user, experienced in data modeling, stated: 'If I'm married to my wife, I want my wife be married to me. So why aren't the relations bidirectional?' True, but what if you know about the king of Sweden? Does that mean that he knows about you? Normally not. Relations in the real world may be both bidirectional and unidirectional However, the relations always start from the objects In the married example, the wife knows about his husband and the husband knows about his wife. To treat the associations as unidirectional is thus the more general case

(2) Data modeling is often used in combination with relational databases. Here the relationship between objects will not be

explicit, but rather it will be implicitly expressed as foreign keys and JOINs between tables In an object-oriented implementation however, the relationship will be made much more explicit Here the relationship will exist in some way, normally as an instance variable implemented in some class

If the name of such a variable would be driven-by, or even worse, has-a, it would be almost impossible to understand the implementation It is better to have names like driver or engine instead In this way we increase the traceability between the models and the actual code (This argument is strongly related to the first, in being actually a special case of it)

(3) This naming principle will normally give better names, avoiding names like has-a, consists-of or operated-by Instead, the roles an object plays are expressed explicitly. These role definitions may sometimes give rise to subtypes of refered objects Refer to the example above; the driver may need some additional information in addition to being a person The driver will then naturally form a subtype of person This technique will thus help us to formulate proper inheritance relations as discussed in Chapter 3

As a consequence of the noun-phrase strategy we will sometimes end up having the same name for the association as for the referred object This should not be viewed as something wrong, but quite the opposite; it is good This is because you have made a proper use of the object, using it as a role for what it was intended for in the beginning We thus name the association as a **role** or as a **subtype** of the object associated

To avoid confusion between objects and relationships, we normally write object names with capital first letter and relationships with a lower case first letter.

A special type of acquantaince association is the **consists-of** association, which is used to express that an object is composed of other objects. Such a structure where a uniting object has associations with participating parts is sometimes called an **aggregate**, as discussed in Chapter 3 This is common with interface objects. In a window system, for instance, we want to express that a window can consist of buttons, menus, and scroll bars Each such unit can then be modeled by an interface object of its own. The result will be an interface structure forming a tree. In *Customer Panel* the buttons and the sensors can become interface objects of their own, depending on how we wish to model the customer panel. This is illustrated in Figure 7.18. The interface object *Customer Panel* we call the **central** interface object. This is also often called a **containment hierarchy** or a **partition hierarchy** as discussed in Chapter 3.

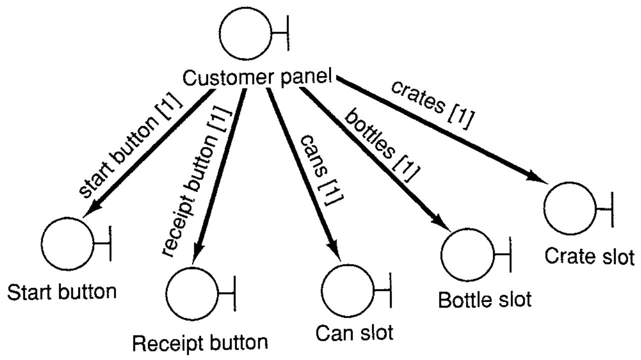


Figure 7.18 An example of an interface structure using a containment hierarchy.

Having identified the interface objects, it will be easy to modify an interface in the system. If, for instance, we wish to exchange the receipt printer in the recycling machine system for another one, the changes will concern only this interface object By having everything that concerns a specific interface in one object, every change of that interface will be local to this object. Since changes of interfaces are one of the most common changes, it is vital that these be manageable.

It soon becomes obvious that there are two different types of interfaces to model There are those that are interfaces to other systems and there are those that are interfaces to a human user.

For interface objects which communicate with other systems, the communication is usually described in terms of communication protocols. These interface objects can be of a type that translates to a standardized protocol, or may be of a type that just sends out stimuli that are produced internally without any complex conversions. This constitutes another advantage: if the protocol is exchanged, these changes will be local to this interface object. Actually, in one project where we knew that the protocol was to be changed, we modularized the interface object so that changes (in this case climbing the OSI layers) would be very easy to incorporate. An important problem arises when there are continuous signals from the outside world, such as in measurement and/or control systems. Then our interface object must either sample the input signal or trigger when certain values are passed, since internally in the system, there is only discrete communication via stimuli. The interface objects must then translate from continous information to discrete information. Quantification problems may then arise and should be taken care of.

The other kind of interface objects are human users. Today, we often request graphical user interfaces (GUIs) between the system and the user. Such interface objects can be complex to model; but they can be modeled as whole structures of the interface objects as previously showed. Several different techniques exist for good interface design. It is fundamental that the user experiences a logical and coherent picture of the system. This can be accomplished by using the problem domain objects as discussed earlier. By doing interface design at that early stage and having the prospective users take an active part, we also guarantee that the interfaces will satisfy the user needs. In interface intensive applications it is not abnormal that the user interface will be the major part (up to 80%) of the entire application. When developing interface intensive application, supporting tools should be used. Such tools include different kinds of user interface management systems (UIMs), see Hix (1990), but also different kinds of windowing systems, frameworks and development environments with predefined parts such as X-Windows, NeWS, MacApp, NewWave, Smalltalk, NeXTStep's Interface Builder, Windows and Presentation Manager. Additionally, several different kinds of class libraries, for example, C++, exist that also support the development of GUIs like InterView or CommonView. Note, however, that the use of class libraries is often redundant when UIMS systems are used.

The purpose of each separate object was discussed in Chapter 6. It became evident that interface objects are suited for presentation, but that they can also handle information and have behavior. How much information and behavior should be tied to an interface object must be decided from case to case. In one extreme, the interface object only sends on the stimuli it receives from the actor to other objects in the system, without actively participating itself in the course of events. In the other extreme case the behavior of the interface object is very complex, and complex information is tied to the interface object, and it can function almost independently of other objects.

How is it possible to decide what behavior in the use case should be tied to a particular interface object? Generally, it is the potential changes that should decide this. Any change of functionality directly coupled to the interface should be local in the interface object. Other changes should not affect the interface objects. Experienced users are very ingenious when inventing potential changes. This is a skill that should be learned and applied in all modeling activities. Every approach of a model should be viewed in the light of possible changes of the system.

Thus, to identify which part of the flow in a use case should be allocated to interface objects, we focus on the interactions between the actors and the use cases. This means that we should look for units:

- Which present information to the actor or request information from him,
- The functionality of which is changed if the actor's behavior is changed,
- Where a course is dependent on a particular interface type.

We can differentiate between several different strategies of how to allocate the functionality, see Hartson and Hix (1989).

(1) **Computation dominant** control or **embedded** control is the case where we place the controling functionality internal to the system, that is, in the control objects and the entity objects. Here the interface objects do not have very much functionality. This structuring can be efficient in execution but hard to prototype from, since not much functionality is introduced in the interface object. As we will see later, we can have the overall system sequencing locally anyway, in the control objects, for ease of modifications.

(2) **Dialogue dominant** control is the case where we place much control functionality in the interface objects and these objects model much of the functionality of the system. In this case we do not have many control objects in the model. This strategy is easy to do prototypes from, but increases the complexity of the interfaces since different abstraction levels are mixed, for example mixing of event detection and global control.

(3) **Mixed** control places the control on both sides allowing invocation of dialogue from the computational side and vice versa This offers more flexibility, but requires more discipline by the programmers to maintain dialogue independence.

(4) **Balanced** control is the case where we place the control separate from both the dialogue and the computation. The global control component, which typically is a control object, governs sequencing among invocations of dialogue and computational functions.

Which type of control to choose must be decided from application to application. OOSE allows modeling of all four types, and actually highlights which strategy was chosen since the analysis models will look different for each strategy. In most cases, however, OOSE promotes alternative (4); the separation of control from other types of functionality The reason for this is to have a high locality in future changes of the functionality.

7.3.2 Entity objects

To model the information that the system will handle over a longer time we use **entity objects**. Typically such information survives use cases, so the information should be kept even if the use case has been completed. Besides the information to be handled, we also allocate the behavior that naturally belongs to this information to the entity object

The entity object are identified from the use cases, just the same as the interface objects. Most entity object are found early and are obvious These 'obvious' entity object are often identified in the problem domain object model. Still others can be harder to find. Entities usually correspond to some concept in real life, outside the system, although this is not always the case. It is very easy to model too many entity object, in the belief that more information is necessary than is really called for The hard thing is to model only the entity object actually needed It is therefore essential to work in a structured way when modeling the entity object. The needs of the use cases should be the guidelines and only such entity object as can be motivated from the use case descriptions should be included.

In the example with the recycling machine, we look at what information must be kept for a longer time. Below is the use case description, with italics to show where we mention some information functionality:

> The course of events starts when the customer presses the 'Start-Button' on the customer panel The panel's built-in sensors are thereby activated
>
> The customer can now return deposit items via the customer panel The sensors inform the system that an object has been inserted, they also measure the deposit item and return the result to the system
>
> The system *uses the measurement result to determine the type of deposit item can, bottle or crate*
>
> The *day total for the received deposit item type is incremented*, as is the *number of returned deposit items of the current type that this customer has returned*
>
> When the customer has returned all his deposit items he asks for a receipt by pressing the 'Receipt-Button' on the customer panel
>
> The system compiles the information that is to be printed on the receipt For each type of deposit item *its return value and the number of returned items by the current customer* are extracted
>
> The information is printed out, with a new line for each deposit item, by the receipt printer
>
> Finally, the grand total for all returned deposit items is extracted by the system and printed out by the receipt printer

From this text we can reason as follows. Since we must remember how many of each type of cans, bottles and crates have

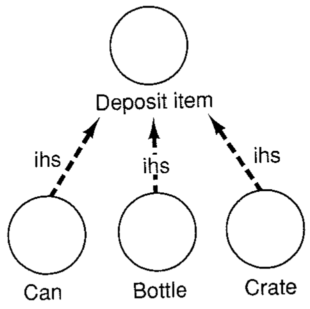


Figure 7.19 *Can, Bottle* and *Crate* have common properties inherited from *Deposit Item*

been deposited this day, we need something that handles this information We thus identify the entity objects *Can, Bottle* and *Crate.* These entity object can also handle their size, deposit values and other information that can be tied to these objects These enitites have common properties and thus could actually be modeled as instances of the same class. However, we want to handle the sizes of the objects a little differently. Cans have a width and a height, bottles have a neck width and a bottom width and crates have height, length and width. However, some properties can be extracted (e.g. deposit value, day total) and placed in an abstract entity object. These common properties which the other entity object will inherit we place in *Deposit Item*, see Figure 7.19.

To store information, objects use **attributes**. To each entity object we can thus tie several attributes Each attribute has a **type,** which can be of a primitive data type, such as integer or string, but it can also be of a composite data type which is more complex and that is especially defined An attribute is described as an association with a name and cardinality indicating the attribute's type, see Figure 7.20 Note the similarities between this and the acquaintance association. Actually, attributes and entity object have many properties in common and it can sometimes be hard to know when to use an entity object and when to use an attribute. This will be discussed further soon. Attributes can be used in all object types to describe the information to be stored

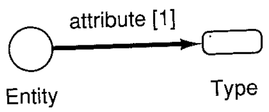


Figure 7.20 An attribute of an object.

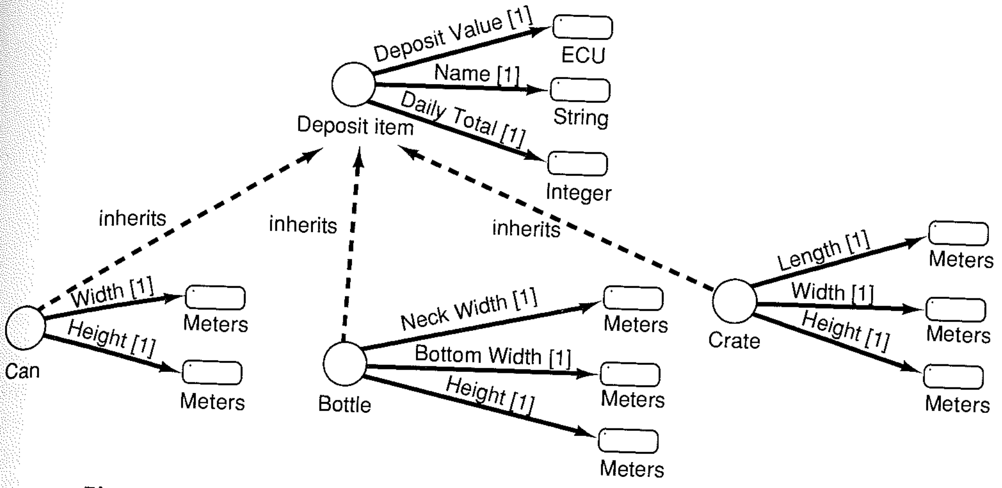


Figure 7.21 Attributes of some of the recycling machine entity objects

The attributes of an entity object develop as the use cases are analyzed. We have already mentioned some of the attributes of the entity object in the recycling example These are shown in Figure 7.21

Let us continue analyzing the recycling system We now have the means to handle the information about the items and also their daily amount To be able to print out the receipt to a specific customer when he is through with his deposition, we will need something to keep track of all the items which he has turned in and that thus should be printed on the receipt. This is actually not information that will survive the use case, but since it is very essential information for the system, we model it with entity object anyway. We thus identify an entity object *Receipt Basis* Since a specific customer may insert several different kinds of items, we must keep the information on how many of each type he has turned in. We have here two alternatives. The first is that the *Receipt Basis* handles only one specific type of item and, when several different types are deposited, we create new instances of *Receipt Basis*. The second alternative is to have only one instance of the *Receipt Basis* handling all items, and if several different types of items are inserted this will be handled internally to *Receipt Basis* Since we want to encapsulate as much as possible, to decrease dependency between the objects, we choose the second alternative, namely to have only one *Receipt Basis* for each customer *Receipt Basis* thus keeps track of all items the customer turns in. Since we need to know which type of items we are counting we will need an aquaintance association between *Receipt Basis* and *Deposit Item* Note that we will actually never count instances of

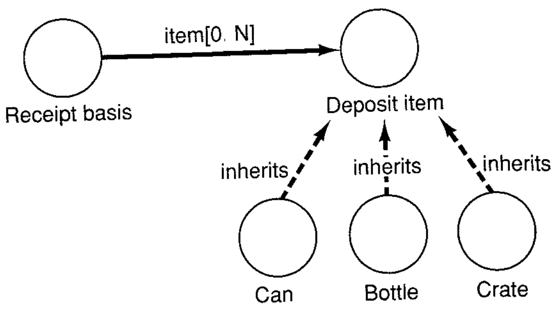


Figure 7.22 The entity objects that are necessary to deposit items in the recycling machine

Deposit Item, but here we use the polymorphic property that is fundamental to all object-oriented systems. We thus do not need to know exactly which class the instances we are counting belong to. By the association we have limited the polymorphism to being a descendant of *Deposit Item*. We will set the cardinality to [0..N] since we may count several different instances with *Receipt Basis*. We thus have the picture shown in Figure 7.22.

Receipt Basis keeps track of how much a specific customer returns and of which type. We thus will have an attribute storing the number received for each type in the entity object. The number information that the *Operator* wants at the end of the day will be placed in the entity object for the items. We must thus create an instance of each type of item that the system is able to receive at system start-up. In them we will store information about the deposit value, the size of the items and the number of items that have been returned that day. The *Receipt Basis* will have acquaintance associations with these instances, and in an instance of *Receipt Basis* we will store information about the number of each particular item type that the customer has returned. Hence, for each specific association, we assign a number. This can be done by just noting this attribute of the relation or adding a new entity object to hold this information.

It is not always easy to decide whether a certain piece of information should be modeled as an entity object or as an attribute. To be able to decide, we must see how the information will be used. Information that is handled separately should be modeled as entity object, whereas information that is strongly coupled to some other information and never used by itself should be made into an attribute of an entity object. In other words, what is decisive is how the use cases handle the information. Certain information can become an entity object in one system, while it may very well become an attribute in another system. If it is used from two different directions and for different reasons, it should form a separate entity object.

It is usually easy to find the necessary entity object, but it is much more difficult to identify what operations and which attributes these entity object are to offer. The only way to manipulate an entity object is via the operations. Therefore the identified operations must be sufficient for all those who wish to use the entity object The detailed description of the use cases is an extremely valuable means to find the desired operations By following them, we will very naturally come up with the necessary operations The whole course of events is described in the use cases, and by extracting those parts that concern our entity object, the operations will appear naturally In Chapter 8 we will describe the technique used in OOSE to do this in a structured way. The technique uses mainly interaction diagrams. Although these are described in the construction chapter, they may very well be used here to identify operations. We will later discuss the pitfalls and benefits of identifying operations in the analysis model relative to the design model. The normal case is *not* to identify operations in the analysis model, since these often change in the design The normal case is therefore to postpone the identification until construction, but we will discuss some aspects of operations here anyway

Operations can be more or less complex. One extreme case is that an entity object comprises only reading and writing operations, and the other extreme case is having whole courses of events included in the operations. As always, the right thing is a middle course between these extremes The same basic rules apply to entity object as to other objects, that is, everything (behavior and information) that is naturally connected with the entity object should be placed in it In the same way it is important to see what consequences any changes will have. The aim should be that any type of change should be as local as possible. Beginners may sometime only use entity object as data carriers and place all dynamic behavior in control objects (as will be discussed later). This should, however be avoided. Using this extreme, we will end up with a function/data structure; and we have earlier discussed the problems with such a structure Instead, quite a lot of behavior should be placed in the entity objects An often appropriate way to realize how behavior should be placed is to model initially not using control objects at all, that is, just using interface objects and entity objects. In fact, we can build all systems with just these two object types. However, when such a model has been developed, you will notice that there are certain behaviors that are not naturally placed, from a maintainability view, in entity objects or interface objects; or even worse, they are spread over several objects This behavior should be placed in the control objects When continuing with the recycling example and allocating behavior in more detail, as we will do in the construction chapter, please feel free to eliminate the control objects from the model and think of how

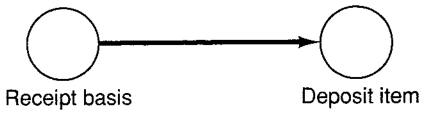


Figure 7.23 *Receipt Basis* increments *Deposit Item* number when a new item has been received.

you would place behavior without them.

The following is a list of typical operations that must be offered by an entity object:

- Storing and fetching information,
- Courses that must be changed if the entity object is changed,
- Creating and removing the entity object.

An operation on an entity object may mean that the entity object proceeds to another entity object and asks for information about something, see Figure 7.23. This communication takes place through **communication associations** A communication association models communication between two objects. Through these associations, an object sends and receives stimuli. The association starts from the object that is to perform the manipulation (i.e. send the initiating stimuli), and is directed to the object where the manipulation is to take place. Since it is an instance association, it is solid; and since it is a dynamic, we will not name the association.

As the entity objects are modeled, it will be found that similar entity objects occur in several use cases. It must then be decided whether there should be more than one entity object. Even if the use cases do not make exactly the same demands on them, the entity object may offer operations so that the use cases may use them in the way that they wish. It is important not to overuse the chance to create operations on entity object. The operations are, after all, going to be implemented later. The basic rule to follow when deciding if two entity object actually are one and the same is to see what occurrences they represent. If it is the same occurrence, there should be only one entity object, otherwise not. When it is decided that entity object should be merged, operations, associations and attributes should also be integrated. The same functionality that existed in the integrated entity objects should be found in the new entity object.

7.3.3 Control objects

We have now partitioned the flow of the use case into interface objects and entity objects. In some cases all flow has been placed on objects of these two types In that case no control objects is needed for that use case. However, in more complex use cases, there often remains behavior that is not naturally placed in any of these two object types. Such behavior is placed in **control objects**. The reason that such behavior is hard to place in any of the other object types is that it is behavior that really does not belong to the interface of the system, neither does it belong to how the information is handled. One possibility is to spread the behavior, anyway, over these object types, as suggested by some methods, but that solution is not ideal from a changeability perspective. A change in such behavior (often functionality) could then affect several objects and thus be hard (expensive) to incorporate; see the discussion on control objects in Chapter 6

The control objects typically work as glue or cushions to unite the remaining objects so that they form one use case. They are typically the most ephemeral of all the object types and usually last only during the performance of one use case. It is, however, difficult to strike a balance between what is placed in entity objects, control objects and interface objects. We will here give some heuristics as to how to find and specify them

The control objects are normally found directly from the use cases. In a preliminary draft we will assign one control object for each concrete and abstract use case Each use case normally involves interface objects and entity objects. Thus behavior that remains after the interface objects and entity object have obtained their parts will be placed in the control objects.

Deviations from this first approach can be made for several reasons. The first is the extreme case when no behavior is left to model in the use case. A control object need, of course, not be modeled then. If, on the contrary, there remains behavior of a very complicated type after the distribution among interfaces and entity object, the functionality may be divided into several control objects. They should then have limited tasks, and will thus be simpler to understand and describe If a control object is somehow coupled to several different actors, this might indicate that the behavior is different for the actors, and that it should therefore be split up among several control objects. The aim should be to tie only one actor to each control object, as changes often are caused by behavior of the actors. The reason for this is that system changes often start from actors, and, if each control object is dependent on only one actor, then changes in the system can be isolated.

Typical types of functionality placed in the control objects are thus transaction-related behavior, or control sequences specific to one or a few use cases, or functionality to isolate the entity object from the interface objects. The control objects are thus the ones that unite courses of events and thus will carry on communication with other objects.

Let us again look at the use case description to see if we have any functionality typical for control objects We have marked in italic these stages

> The course of events starts when the customer presses the 'Start-Button' on the customer panel The panel's built-in sensors are thereby activated
>
> The customer can now return deposit items via the customer panel The sensors inform the system that an object has been inserted, they also measure the deposit item *and return the result to the system.*
>
> *The system uses the measurement result to determine* the type of deposit item: can, bottle or crate
>
> *The day total for the received deposit item type* is incremented, as is the number of returned deposit items of the current type that this customer has returned
>
> When the customer has returned all his deposit items he asks for a receipt by pressing the 'Receipt-Button' on the customer panel.
>
> *The system compiles the information that is to be printed on the receipt For each type of deposit item* its return value and the number of returned items by the current customer *are extracted.*
>
> *The information is printed out*, with a new line for each deposit item, by the receipt printer
>
> *Finally, the grand total for all returned deposit items* is *extracted by the system and printed out by* the receipt printer

This is one concrete use case that inherits an abstract use case *Print* The first option, namely one control object for each (abstract and concrete) use case would have given us two control objects here. The *Returning Item* is a use case that involves the behavior of coupling the interface objects and the entity objects together This behavior is a good candidate for a control object. The *Print* use case, however, involves the behavior of printing on the line printer: but this behavior should be naturally tied to the interface object *Receipt Printer*. We thus do not assign a control object for this abstract use case. The other use cases we have not described in detail here yet, so it is harder to tell whether or not a control object is needed for them. However, as a first approach, we could assign control objects even for these use cases. Whether this was correct or not will be seen at the latest during construction, when explicitly defining the courses and the stimuli sent between the objects. The control objects will thus be as in Figure 7 24

The description of a control object is found indirectly from the use cases by checking how the use case runs over the other object

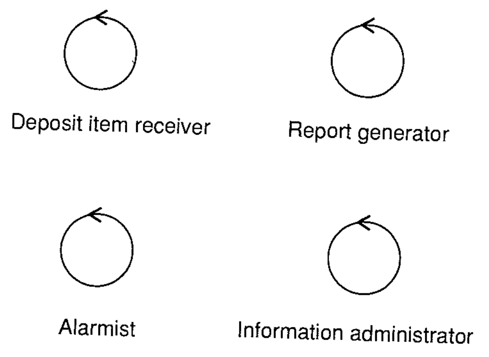


Figure 7.24 Controllers in the recycling machine

types In those places where the control object needs to be involved, it is stated what role the control object will take and what that role involves.

Uses between use cases often are mapped in communication associations between the corresponding objects. If use cases have an extends association, this association can normally be transferred directly into an extends association between objects.

To model the extends in the recycling example we choose to use extends between the control objects. The reason for this is that we do not want to blur the specification of the *Deposit Item Receiver* object. Instead we describe the *Alarmist* as something that will be injected when an alarm occurs. We thus have the picture as shown in Figure 7 25. Note that in this example there are no communication associations between the control objects.

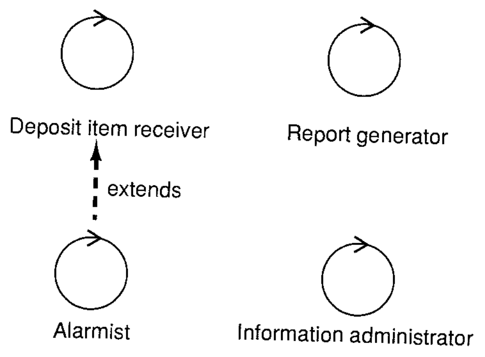


Figure 7.25 The complete view of the control objects in the recycling example

7.3.4 Working with the analysis objects

When working with the development of the analysis model, normally one is working with one use case at a time. Thus, for one specific use case, one identifies interface objects, entity objects and control objects before continuing with the next use case. However, since the analysis objects are orthogonal to the use case in the sense that one object may participate in several use cases, this process is iterative. This means that when a set of objects already exist, these may be modified to fit also the new use case. The goal is to form as stable a structure as possible, reusing as many objects as possible.

The specification of the object should be made in a text form. The simplest way of doing this is just to copy the text from the use cases and paste it into the object descriptions. However, this is not recommended since the text will then be out of context and thus can be hard to comprehend. It is better to describe each object's **role** and **responsibilities** bearing in mind those of the other objects.

We have mentioned the use of operations to specify the objects. This is possible, but it introduces a formalism that the model may not be sufficiently stable to handle yet. Changes would then have serious consequences. Even if operations are used, a formal syntax should definitely be avoided. When specifying an operation, it is tempting to almost use pseudo-code to do the specification, but this should be avoided. Any changes (and there will be changes!) will be very hard to introduce since the specifications will be rewritten. When describing behavior, concentrate on what will happen, not how it will happen. Generally, in system development the formalism should come creeping in when the structures grow stable and should be introduced incrementally

A technique to find operations on the objects from the use cases will be described in the next chapter This technique could very well be used, as mentioned, in the analysis model. It is, however, from our experience too much work to use it in both analysis and design. If it is used in analysis, then these operations should be kept as far as possible in the design model. Thus, if you realize that the implementation environment will have very little effect on your analysis model, you can define operations during analysis; otherwise we recommend doing it during design.

When the objects have been identified and described, though, we may express how the objects offer the use case; not as formally as specifying what stimuli will be sent between the objects, but in a more prose-like form. Initially, then, we often create a **use case view** of the objects showing how they will be used in the use case.

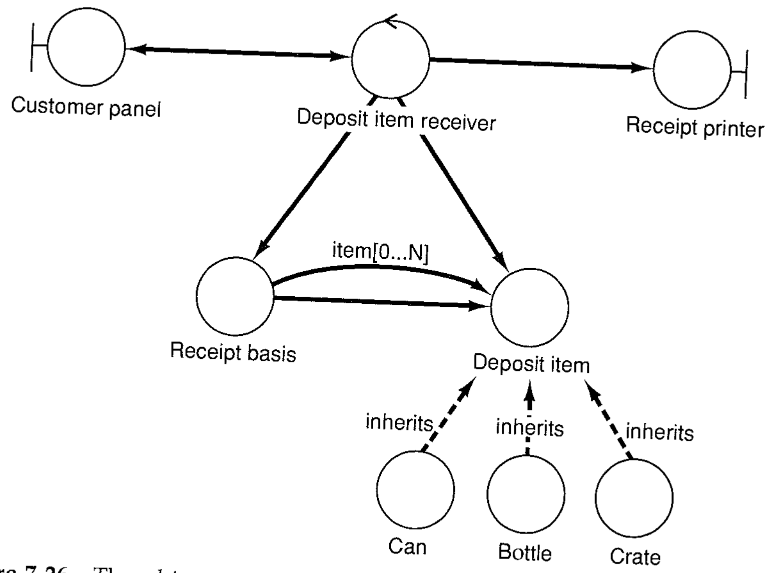


Figure 7.26 The objects supporting the use case *Returning Item*

Let us look at the use case *Returning Item*. Its view is shown in Figure 7.26.

We may also describe the use case in terms of these objects to show how they will provide the use case. This description will help you to understand the use case and how it has been distributed over the objects Here is such a description for the use case *Returning Item*.

> The use case starts in *Customer Panel* when *Customer* presses the start button Sensors are then activated
>
> *Customer* can now turn in his returnable items via the *Customer Panel* *Customer Panel* will tell the active object *Deposit Item Receiver* that an item has been handed in, as well as the measures of the item
>
> *Deposit Item Receiver* will use the entity objects *Can*, *Bottle* and *Crate*, together with the measured values from *Customer Panel*, to find out what type of item has been received The daily total of the received object of this type will be enumerated in the object
>
> By means of the entity object *Receipt Basis*, *Deposit Item Receiver* will keep track of how many items of the current type the customer has turned in so far
>
> After turning in his returnable items, *Customer* will ask for his receipt This is done via *Customer Panel*
>
> *Deposit Item Receiver* finds the information to be printed on the receipt A line is printed for each item that has been turned in, stating

the number of each type as well as the return value for this number. Finally, the total return value sum is printed on the receipt. The information on the receipt is taken from the *Receipt Basis*

The information is printed on the interface object *Receipt Printer*, which is also told when the printout is completed

We have seen here that the analysis model will not be a reflection of what the problem domain looks like This is important to understand. The reason is simply to get a more maintainable structure where changes will be local and thus manageable. We thus do *not* model reality as it is, as object-orientation often is said to do, but we model the reality as we want to see it and highlight what is important in our application.

7.3.5 Subsystems

When the analysis objects have been identified, the system will contain a large number of objects. For a medium sized project, typically between 30 and 100 objects have been specified. It is seldom possible to get a clear survey of this number of objects, so the objects need to be placed in groups. This can be done at one or several levels depending on the size of the system. Such groups of objects are called **subsystems**. The system thus consists of a number of subsystems which can contain subsystems of themselves. At the bottom of such a hierarchy are the analysis objects. Subsystems are thus a way of structuring the system for further development and maintenance.

The task of the subsystems is to package the objects so that the complexity is reduced. The subsystems also work as handling units in the organization, for example at development, marketing, sales and delivery. A subsystem can be a compulsory system unit, but it can also be an optional unit.

The lowest level of subsystem is to be viewed as change units. We call these **service packages**. These should be viewed as atomic; if the customer wants it, he will have the whole of it, otherwise he will get nothing When the system is to undergo a minor change, this change should concern no more than one such subsystem, or rather the objects contained in this subsystem. This means that the most important criterion for this subsystem division is predicting what the system changes will look like, and then making the division on the basis of this assumption. One and the same subsystem should therefore preferably be coupled to only one actor, since changes are usually caused by an actor.

The division into subsystems should also be based on the functionality of the system. All objects which have a strong mutual functional coupling will be placed in the same subsystem, see Embley and Woodfield (1988) or Yourdon and Constantine (1979). To identify objects with a strong mutual coupling, we can start from one object and study its environment. Another criterion for the division is that there should be as little communication between different subsystems as possible.

What is most convenient is to start looking for optional subsystems. Optional subsystems are not only those that are optional at a specific delivery; anything that *could be* optional should be taken into consideration.

In the recycling example we may view all objects handling the alarm as optional (the machine would work without an alarm) and also functionally related. We thus may have one subsystem *Alarm* that includes the control object *Alarmist* and the interface object *Alarm Device*.

After the optional subsystems have been identified, what remains is often those objects that are central to the system and therefore always have to be delivered. To distribute these among different subsystems, we will need to look at the functionality of the system. All the objects having to do with a particular part of the functionality will be placed in the same subsystem. To identify functionality parts, we can look at an object. What will happen if we remove this object? Objects that then become superfluous in one way or another, and are connected to the removed object, should therefore be part of the same subsystem. There are several similar criteria to be used to decide whether two objects are strongly functionally related. There follows a list of a few of them.

- Will changes of one object lead to changes in the other object?
- Do they communicate with the same actor?
- Are both of them dependent on a third object, such as an interface object or an entity object?
- Does one object perform several operations on the other?

The aim is to have a strong functional coupling within a subsystem and a weak coupling between subsystems. A good start is to begin by placing the control object in a subsystem, and then place strongly coupled entity object and interface objects in the same subsystem.

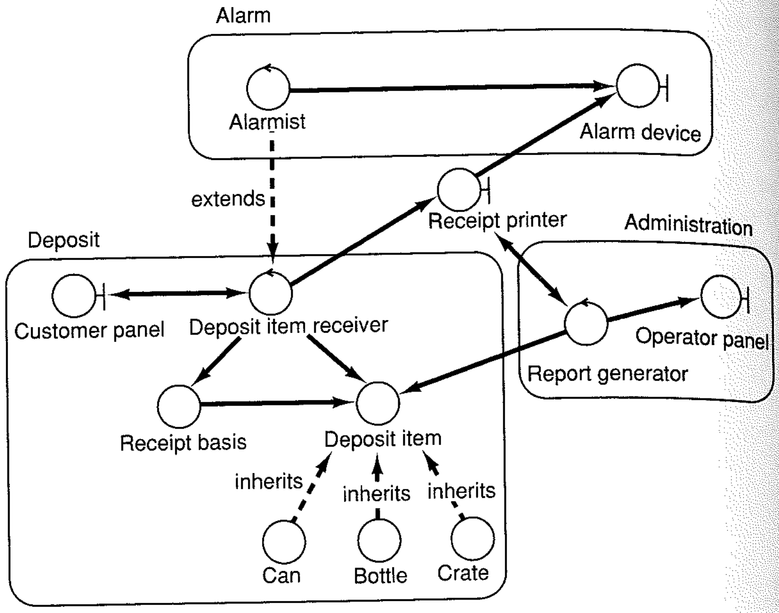


Figure 7.27 Subsystems in the recycling machine.

If it seems very difficult to place a particular object in a subsystem, it can be placed outside all subsystems. This may be due to the fact that it is completely separate from other subsystems, or that it has the same functional coupling to two or several subsystems so that it is difficult to choose which one to place it in.

Let us apply these criteria to the recycling example. We have already identified the *Alarm* subsystem. The functionality to handle all depositing is functionally strongly coupled and thus could be placed in one subsystem, *Deposit*. All objects involved in the management of the system could be placed in one subsystem, *Administration*. The interface object *Receipt Printer* could be placed in any of these two subsystems since it is used in both places, so we choose to place it outside the subsystems. We thus have the picture in Figure 7.27. Any further levels of subsystem are hardly needed here; the next level would be the system itself.

We can test some criterion on this division, for instance how many actors are dependent on a certain subsystem. *Deposit* is only dependent on *Customer*, and *Administration* is only dependent on *Operator*. *Alarm* is not related directly to one actor, but is used indirectly by both actors. The reason why it was placed as a separate subsystem was that it constituted a certain functionality part of the system

When the division into subsystems is made, in some cases it may also be desirable to modify the analysis objects also. This may be the case, for instance, when an entity object has separate behavior that is functionally related to more than one subsystem. If this behavior is then extracted, it may be easier to place the entity object in a subsystem.

To express how subsystems are related we can assign a **dependsOn** relation between subsystems This relation then means that objects in one subsystem will use, in some way, objects in another subsystem. In the example above we see that the *Administration* subsystem depends-on the *Deposit* subsystem since *Report Generator* uses *Deposit Item*. Hence, if we deliver one subsystem, we must also deliver any subsystem that this subsystem depends on.

The subsystem division in small projects is normally made at the end of the analysis, when the architecture is clear In larger projects, however, it often must be done much earlier, in many cases even before the analysis model has been developed For large projects there may thus be other criteria for subsystem division, for example:

- Different development groups have different competence or resources, and it may be desirable to distribute the development work accordingly (the groups may also be geographically separated),
- In a distributed environment, a subsystem may be wanted at each logical node,
- If an existing product can be used in this system, this may be regarded as a subsystem.

In large systems it is often essential to develop the system in **layers**. Then a subsystem for base functionality is developed on which applications are built. The applications uses this base functionality. Figure 7 28 illustrates an example of such an architecture.

7.4 Summary

The analysis process aims at defining and specifying the system to be built Two models are developed, the requirements model and the analysis model. Both of these models are logical in the sense that they do not incorporate any requirements from the actual implementation environment

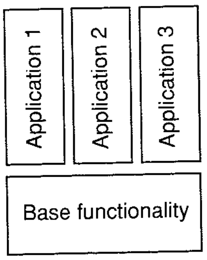


Figure 7.28 An example of structuring the system into one subsystem for base functionality used by all application programmers and separate subsystems for each application

The requirements model uses actors and use cases to describe in detail each and every way to use the system from a user's perspective. Actors then model external factors like people or machines that interact with the system Use cases are the flows that these actors will perform on the system Use cases can be understood intuitively by non-technical personnel and thus can form a basis to communicate and define the functional requirements of the system in collaboration with the potential users. As a support to define the requirements model, a problem domain object model can be a strong tool, as it will form a common platform determining what the system will handle. To attain a high degree of maintainability, these problem domain objects will not form a base for the realization of the system. As a complement to the use cases and the problem domain objects, it is also appropriate that the requirements model be accompanied by descriptions of the system interfaces.

The use case model will form a thread running through all development work in OOSE. Both when structuring the system, to define operations on the objects, and, not least, as a tool to do integration testing. We thus have a high degree of reusability for this model.

The analysis model is the second model developed in the Analysis process This model aims at forming a logical and maintainable structure in the system. It is logical in the sense that the actual implementation environment is not taken into account The reason for this is mainly to focus on the essential system functionality. Three object types are used to structure the system: interface objects, entity objects and control objects The interface objects will model all functionality that concerns the system interfaces, the entity object model all functionality that handles the actual information kept in the system for longer periods and the control objects model such functionality that is not naturally tied to any of the other objects (often mainly behavior) These object types are identified when the use cases are analyzed and broken down. The objects should completely offer the functionality of the use cases. To structure the system in larger units subsystems are used. Subsystems group the objects in the analysis model.

The development of these models forms an iterative process, as they will undergo many changes before they become stable. Therefore not too much effort should be put into these models before they have reached a mature state. However, even if it is often wise to sketch subsequent models when developing models, the early models must be stable before work starts on the subsequent models.

8 Construction

8.1 Introduction

8.1.1 Why do we have a construction process?

We build our system in construction, based on the analysis model and the requirements model created during analysis. The construction process lasts until the coding is completed and the included units have been tested. Construction consists of **design** and **implementation**.

Following the construction process we have the testing process in which the use cases and the entire system are tested and certified. This does not mean that you must wait until all parts have been constructed before starting the verification of the system, instead we try to do as much as possible in parallel. Additionally, verification involves a lot more than only testing code. If possible we also try to start construction before analysis has been completed.

What, then, is the purpose of the construction phase? Could we not write the source code directly from the analysis model? There we described the 'objects' in the system and how they are related.

There are three main reasons for having a construction phase:

(1) The analysis model is not sufficiently formal. In order to seamlessly change to the source code we must refine the objects; which operations should be offered, exactly what should the communication between the different objects look like, which stimuli are sent, and so on?

(2) The actual system must be adopted to the implementation environment. In analysis we assumed an ideal world for our system. Luckily enough there is no ideal world, so in reality we must make adaptations to the environment in which the system is to be implemented. This means that we must initially transform the analysis model from the analysis space to a space with still more dimensions, see Figure 8.1. We must for example

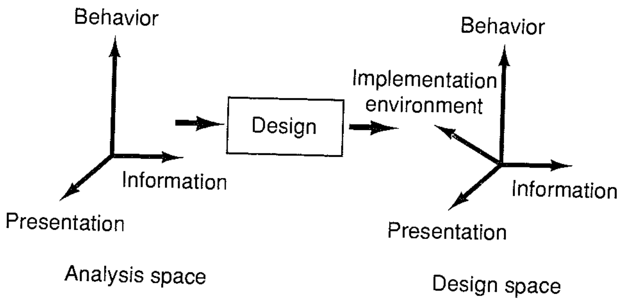


Figure 8.1 Construction initially transforms a model in the analysis space into a model in the design space This is mainly done in the design part of construction

consider performance requirements, real time requirements and concurrency, system software, the properties of the program language, the data base management system to be used, and so on

(3) We want to validate the analysis results. As our system is growing and formalized, we will see how well the analysis model and the requirements model describe the system. During construction, we can see at an early stage whether the result of the analysis will be appropriate for construction If we discover unclear points in the analysis model or the requirements model, we must clarify them, perhaps by returning to the analysis process.

These three reasons may look as though there are deficiencies in the result of the analysis phase that we must clarify here. This is an incorrect view since the purpose of analysis is to understand the system and to give it a good structure. It is consequently important to understand that the considerations given in construction should influence our system structure as little as possible; we want to keep the good properties in the system that the analysis model has focused on. It is the application itself that mainly controls the structure, not the circumstances when implementing it. We therefore made an informal and comprehension-oriented analysis where these considerations did not disturb the work.

Changes in the system architecture to improve **performance** should as a rule be postponed until the system is being (partly) built. Experience shows that one frequently makes the wrong guesses, at least in large and complex systems, when it comes to the location of the bottle necks critical to performance To make correct assessments regarding necessary performance optimization, in most cases, we need something to measure Otherwise we will only make more or

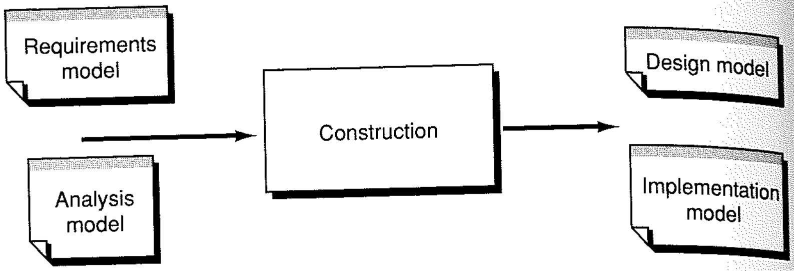


Figure 8.2 The input and output models of construction.

less appropriate guesses. And since we have nothing to measure until the system has been built, we cannot make these optimizations at an early stage. One way of avoiding this dilemma is to make simulations of the system in which you simulate the critical parts. Another way is to perform prototyping to get an early opinion of what the system will look like, but this is a risky method since you can easily make (too) rough simplifications. A prototype always aims at highlighting some issue. General conclusions on other issues can therefore not be drawn from a prototype that does not highlight the particular issue. However, if you know for sure where the performance critical parts are, for instance because you have a very good application knowledge, optimizations could be done at an early stage. For a similar discussion, see Barry (1989).

8.1.2 What is done in the construction phase?

Construction activity produces two models, the **design model** and the **implementation model**. Construction is thus divided into two phases; **design** and **implementation** which each develop a model, see Figure 8.2. The design model is a further refinement and formalization of the analysis model where consequences of the implementation environemnt have been taken into account. The implementation model is the actual implementation (code) of the system.

To develop the design model we perform three main steps.

(1) *Identify the implementation environment.* This step includes identifying and investigating what consequences the implementation environment will have on the design. Here all strategic implementation decisions should be made. How will the DBMS be incorporated into the system? What component libraries will be used and how? How should processes and process communication be handled? Error handling and garbage collection? And so on. This work aims at drawing

conclusions on how these circumstances should be handled in the system. This step can (and should) be done in parallel with the analysis work so that it is ready when the actual design starts.

(2) Incorporate these conclusions and develop a *first approach to a design model*. Here we use the analysis model as a base and translate the analysis objects into design objects in the design model that fits the current implementation environment. From a project perspective this could be directly incorporated in the analysis model, but for maintenance purposes and understandability this is not recommendable. When doing further development, the analysis model forms a logical basis for understanding the system and thus is an essential model to keep during the entire system life cycle

(3) Describe how the *objects interact* in each specific use case. Here the design model is formalized to describe all stimuli sent between the objects and also to define what each operation will do to each object The use case model will be of great help during this work as it will help us to specify each specific flow in the system in detail. This step gives us the object interfaces.

The implementation activity then implements each specific object From the design model we get very detailed specifications of all objects, including their operations and attributes. Various techniques may be used here and we will discuss some of them later. The programming language and the component library used will of course be fundamental tools of the implementation.

We will now discuss these steps in more detail and make some comments on the various techniques used. The recycling example analyzed in the previous chapter will be used and a design and implementation will be made for it.

8.2 The design model

8.2.1 Traceability

The design model will further refine the analysis model in light of the actual implementation environment. Here we will explicitly define the interfaces of the objects and also the semantics of the operations. Additionally we will decide how different issues such as DBMSs, programming language features and distribution will be handled.

The design model will be composed of **blocks** which are the design objects. These will make up the actual structure of the design model and show how the system is designed. These blocks will later be implemented in the source code.

The blocks will abstract the actual implementation. The implementation of the blocks may be one specific class in the source code, that is, one block is implemented by one class. However, often a block is implemented by several different classes. The blocks are therefore a way of abstracting the source code.

The module level of the programming language we denote by the generic term **object module**, see Constantine (1990). In an object-oriented language these modules, that the programmer actually writes, will be the actual classes. When we use a specific language we have a direct correspondence to a language concept and then this concept should of course be used, for example *class* in an OO language or *package* in Ada. Here we will either use the term object module or class.

The first attempt at a design model can be made mechanically based on the analysis model. The transformation is made so that initially each analysis object becomes a block This transformation rule means that we obtain a clear **traceability** in the models. We started modeling the system in the analysis model in a manner providing a robust structure for the system As each analysis object is traceable to a block, changes introduced in the analysis model will be local in the design model and thus also in the source code. Note that the traceability is bidirectional, that is, it also goes the other way – we can trace a class in the source code back to the analysis and see what gave rise to it.

Traceability is a tremendously important property in system development. Each major system will be altered during its lifetime. Whether the changes emanate from changed requirements or responses to trouble shooting, we will always need to know where the changes need be made in the source code.

Here we have the great advantage of traceability; you can easily find your way in the system even if it has been subjected to major changes. It is also important to have a high functional locality (i.e. high cohesion) see Yourdon and Constantine (1979) in order to know that changes will not influence large parts of the system.

In this chapter we will construct the example described in the previous chapter. This description will be in survey form and simplified, but the mode of work should appear clearly enough to permit a complete construction to be done fairly easily. As we go along we will introduce new terms and concepts that are used in the example.

Our analysis model is as shown in Figure 8 3. We will from now on concentrate on the use cases *Returning Item* and *Item Stuck*,

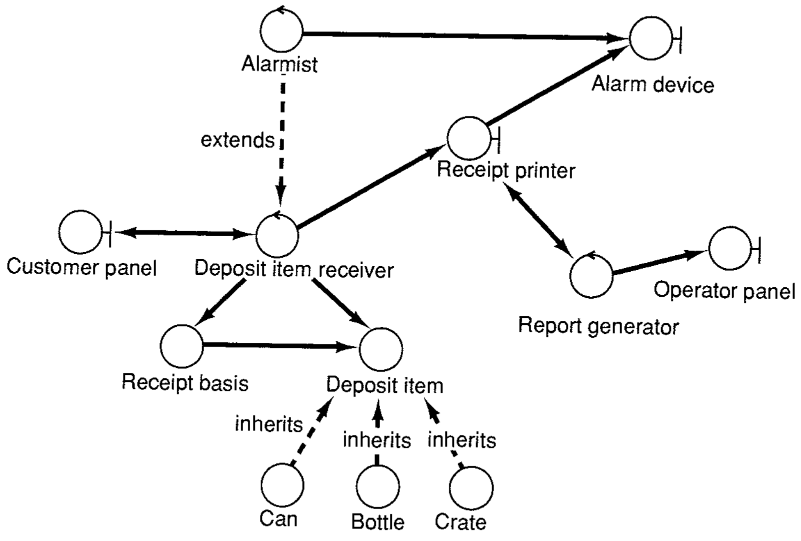


Figure 8.3 Analysis model of the recycling machine.

and consequently we will focus on the objects that participate in these use cases.

Based on this analysis model we can now mechanically, and seamlessly, find the first attempt at a design model. The first attempt is simply to assign the same model as the analysis model. To differentiate between the two models, we use another notation and draw these blocks as rectangles instead of circles, see Figure 8.4.

The semantics of the design model is somewhat different from that of the analysis model. The analysis model is developed in logical terms and is only a conceptual picture of the system to be built. Therefore it is essential to keep and freeze the analysis model for future maintenance even after the design is finished. The design model, however, is an abstraction of how the actual system really is built. The first attempt at this model is a direct mapping from the analysis model Its final structure, however, will reflect how the implementation environment has affected construction. The goal is to keep the structure found in the analysis model and not violate it unnecessary in the design We also want the design to have a logical and robust structure.

The semantics of the blocks should thus reflect the semantics of the objects existing in the actual system, and likewise, the associations between the objects should also reflect how the objects in the system are really related. For example, most programming languages do not have any way to implement the extends association.

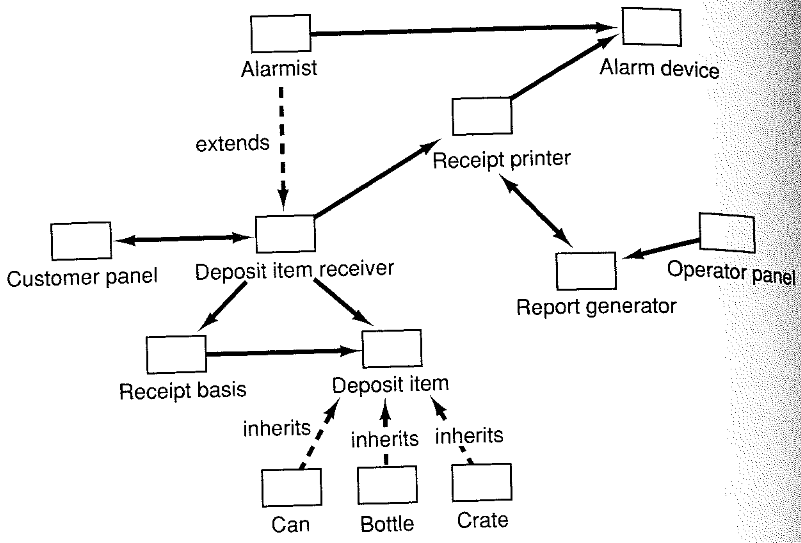


Figure 8.4 First attempt to design the structure of the recycling machine.

During design we therefore must decide how this association should be implemented and change the design model to reflect this. Similarly, if the programming language does not offer any technique to support inheritance, we must reflect on how the inheritance in the analysis model is really implemented. We will discuss these topics in more detail later

This is consequently the first idea for a system architecture. We emphasize that this is the first idea because it may be changed, since we must consider the implementation environment. We may have to break up or divide blocks because we want to distribute them on different nodes in our computer system, we may even need completely new blocks to encapsulate an existing DBMS. How to consider the implementation environment will be discussed shortly.

The design model enables us to reduce the complexity of the system. The blocks are actually an abstraction mechanism for the source code. By speaking in terms of blocks we can discuss the system on a survey level and understand the architecture of the system. Through this abstraction mechanism we reduce the complexity radically, which means that it is easier to build a correct system that will avoid errors due to complexity. It has been noted, see Levendel (1990), that it is qualitatively better to avoid construction errors by reducing the complexity at an early stage than to search the system for faults when it is completed and to correct each error The latter variant is (unfortunately) often typical of today's traditional system development.

8.2.2 The implementation environment

To adapt the design model to the actual implementation environment, we must first identify the actual technical constraints that the system should be built under. This identification may (indeed should) be done early, ideally before the analysis model is developed since it affects how far the analysis model should be refined.

What do we include in the implementation environment? The most apparent parts are perhaps the target environment (where the system should execute during operation), programming language and existing products that should be used (e g. DBMSs), but in fact everything that affects the realization of the system must be included in this concept We will here discuss some of these aspects in more detail, but let us first discuss the overall strategy to handle the implementation environment.

Since one of the more common changes of a system is a change in the implementation environment, it is preferable to handle this in the same 'changeable' way as the rest of the system. This implies that as few objects as possible should be aware of the constraints of the actual implementation environment Therefore the overall strategy to handle it should be to have such *commitments locally* and to *encapsulate* them as much as possible. In this way any change of environment should be local and not have an effect on several objects. (This is not unique to object-oriented software engineering, but rather a traditional way of good programming style. However, in the light of the design model this will be apparent here, and thus we will have a controlled way of handling it)

So, what is included in the implementation environment? We will here discuss some topics.

To be able to change parts in the **target environment**, these parts should be encapsulated in a new block. You will thus create new blocks that represent occurences in the target environment. For example, if your application will handle files in the operating systems, you should create a block that interfaces the application's file handling and the operating systems's file handling as shown in Figure 8.5.

In Figure 8 5, the *File Manager* block should offer operations such as *Create file* to the application objects. This block may then have different implementations for different configurations of the system. There exist different realizations for this. Either an abstract class that specifies the interface can be defined, which is implemented

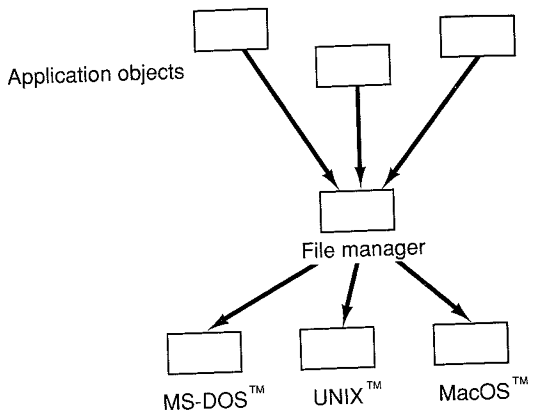


Figure 8.5 A special block should be included to encapsulate the special properties of the target environment

in descendants for different environments that are instantiated dependent on the platform Or the object can check which platform it is running on during run-time and execute the correct statements for this environment A third way is to decide this when the system is delivered for different configuations In C++ (or C) you would then have several c-files (one for each platform) implementing the same h-file which specifies the interface to the application. When producing the software it would then be decided which module should be linked into the final product. What way is chosen is dependent above all on the programming environment. The first is natural for object-oriented programmers, while the second would typically yield CASE-statements in languages where polymorphism is not present. The third, which may very well be combined with the first, is more a choice of how the production and delivery of the software should take place

Likewise it is important to investigate whether the target environment will execute in a distributed way on different processors or in different processes We will discuss this in more detail in Chapter 9 where we have chosen to discuss operating systems processes

Which **programming language** is used will affect the design. We will later discuss how the concepts we use will be translated to the programming language, but also the more basic properties of the language and its environment are fundamental for the design. The existence of inheritance, multiple inheritance, typing, standards and portability are examples of such properties. Furthermore, strategies for handling of errors during run-time must be decided early. Ada offers the possibility of exceptions and Eiffel assertions C++ does not have any such mechanisms (in version 2 of the language) It is essential to decide early on how this strategy should be incorporated in the design Likewise the memory management strategy must be decided early Some object-oriented languages have automatic garbage collection, but in other languages you must as a programmer clean away instances that are not used anymore yourself. Then you must keep track of all references to these instances

Closely related to the programming language is the use of **components** for programming. The use of such component libraries may also affect the design. One example is the design of the interface objects Components (or tools) exist to build interfaces, and this will affect how the design is made. The use of components implies several new questions, which we will postpone discussion of to Chapter 11.

Often **existing products** are used when developing new systems Examples of such products are database management systems (DBMS), user interface management systems (UIMS), network facilities and other internally or externally developed applications that should be incorporated in this new system. The normal strategy to handle these products is to introduce new blocks to encapsulate them in the same manner as was described for the target environment. In Chapter 10 we will discuss in more detail blocks that encapsulate a relational database.

Products that are used only during developments (mostly development environments) may also affect the design. Such products may include compilers, debuggers, preprocessors and other tools. These may affect how the code is written and configured Other such implications may come from standards and coding rules.

During construction we must also take into consideration any **requirements for performance or limitations of memory**. Such requirements may also affect the design. One common example is when databases are used and frequent transactions are too slow. Then a redesign (or modification) of the database design may be necessary. Access paths to the database may then be changed or we may have to introduce new index tables in the database, or change existing ones, to speed up operations on the database. However, the principal strategy on these topics should be, as previously mentioned, to postpone such optimizations until they are needed or you are absolutely sure that they will be needed. It is far too common that a good design is ruined for performance optimizations that are only sub-optimizations Not seldomly the real bottlenecks are missed and then new optimizations are necessary.

To investigate potential optimization problems early, simulation or prototyping may very well be used before the actual design is done Then the designers have much a better basis for determining where the real optimizations should be made. Of course, extensive experience in the application area may also serve to help judge where optimizations should be made at an early stage The message here is that if you are not sure of the correctness of a performance optimization, then you should not make it until you are sure of how it should be done

The **people** and **organization** involved in the development could also affect the design. For instance, if the design has to be done at different sites, a division of the design work is necessary. Different competence areas of the development staff may imply that the design should be done in a specific way. We will not discuss these issues in this book, but the principal strategy should be that such factors should not affect the system structure. The reason for this is that the circumstances (organizations, staffing, competence areas) that are in effect today will probably change during the system's life cycle. To build these circumstances into the system is not a good idea for the maintenance of the system.

All of these issues should affect the design as little as possible for maximum robustness. The overall principle is that changes can and should occur, but all changes should be justified and documented, also for robustness reasons. Principally, we may have to change the ideal design model in various ways:

- To introduce new blocks in the design model which does not have any representation in the analysis model,
- To delete blocks from the design model,
- To change blocks in the design model (splitting and joining existing blocks),
- To change the associations between the blocks in the design model

Normally, adding blocks to handle the environment is a good change Adding blocks for other functionality should normally not be done, since they should be introduced through the analysis model.

Deleting blocks is more suspicious. Why are you deleting the block? If you have good reasons for it (often implementation reasons), it is fine, but if you're actually changing the logical structure of the system, such a change should be made in the analysis model first. If it is not, you should view such a change with great reservation.

Splitting and joining blocks are also suspicious changes. Any such change will often decrease the robustness of the system and should be done with great care For implementation or performance

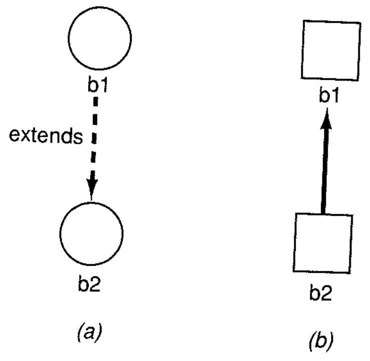


Figure 8.6 Behavior b1 extends behavior b2. This will normally be implemented with a stimulus being sent from b2 to b1.

reasons it may be justified, but the designer should devote a great deal of judgement to such changes.

Changes of associations are perhaps the most common change in the design model. These changes often come from the implementation environment. Synchronization and communication between processes is one example of where the communication between objects may be changed. Other examples are the actual implementation of associations. Additionally, we must decide how to implement the associations. For instance, the extends association has no direct implementation technique in the common programming languages. We must therefore decide how this association should be implemented.

What extends actually means is that a behavior, b1, shall be inserted into another behavior, b2, see Figure 8.6. We want the (description of) behavior b2 to be completely independent and unknowing of behavior bl. Ideally we should want bl itself to take the initiative and insert itself into b2. However, there is no mechanism in some ordinary programming languages by which this can be expressed. One solution to this is to have b2 take the initiative by sending a stimulus to b1. How we do this in the source code will be described when we implement the blocks. We will therefore have a communication association from b2 to b1 where this stimulus could be sent.

In the example of the recycling machine we have one extends association. We choose to implement this in the manner described above. The other associations in the model, however, will not be changed since they are straightforward to implement.

In the analysis model we have a conceptual picture of the functionality. The extends association has come from the *Alarm* use

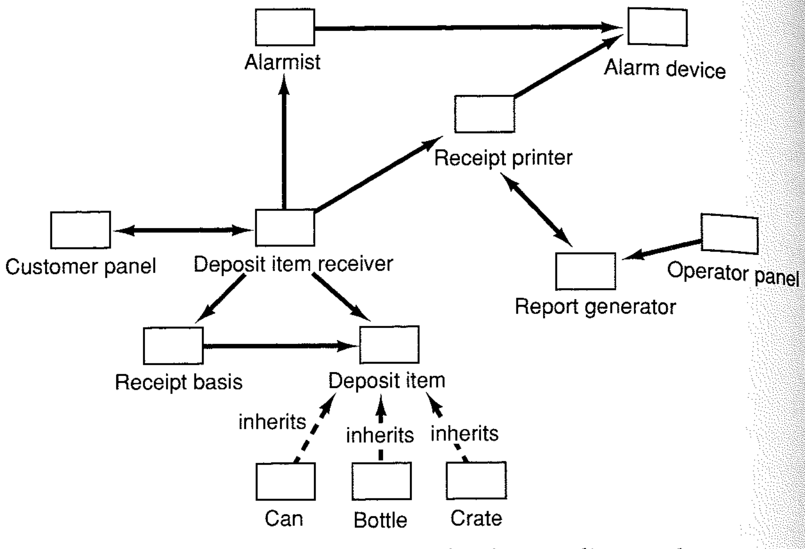


Figure 8.7 The refined designed model for the recycling machine. Note the communication association between *Deposit Item Receiver* and *Alarmist*.

case. When an alarm is detected during insertion of items, the Alarm use case should be inserted The detection is done by *Customer Panel* which sends the error stimuli to the system. The receiver is now *Deposit Item Receiver* extended with *Alarmist* It is thus the *Alarmist* that will take part in this use case The communication association should thus be between *Deposit Item Receiver* and *Alarmist* since this is the way the stimuli will be sent in the design. We thus now have the design model shown in Figure 8.7

Any inheritance associations must also be reviewed if we cannot implement it in the programming language. Such associations can often be designed with a communication association between the blocks. This should then reflect that a stimulus is sent upwards through the 'inheritance hierarchy', so called '**call-through**'. Another possibility is to delete the abstract block and implement this functionality in the concrete block instead. In the example we will use C++, which supports inheritance We will come back to implementation of inheritance in languages that do not support it directly.

The implementation environment for the recycling machine is made very simple. We will run the application in one process only. (In Chapter 9 we will discuss an implementation using several processes.) Additionally, here we will not have a database for persistency. (In Chapter 10 we will discuss an implementation using a relational DBMS.) The programming language will be C++, and this will normally not introduce any seams from this refined design model.

What should be reviewed, however, is how the hardware interface of the recycling machine should be incorporated in the application system. This interface can be viewed as an existing product introduced in the system. Normally these products would yield new blocks to encapsulate the product as discussed previously, but we could also use the block implementing interface objects as this encapsulating object. We will assume that a primitive software interface does exist to the hardware panel, and we will develop C++ classes that encapsulate this interface.

We should also mention something about the usage of components. At an early stage it is essential to decide upon which component libraries to use. This is to encourage the use of components before the actual implementation work is done. When a component library exists, investigations of this should start to reveal what functionality it will offer. To use a component library effectively, it is essential to be familiar with the library; this will take some time studying the classes and their operations. The use of, and especially finding where to use, components will be discussed more in Chapter 11. We will here assume that we have a class library in C++ for commmon data structures that can be used, as for instance the NIH library, see Gorlen *et al.* (1990). In Chapter 11 we will give some rules of thumb to find places where components could be used. One of these rules is to look for acquaintance associations with cardinality [0..N]. This will typically yield a list or array to hold several references. We have this situation in our example, namely between *Receipt Basis* and *Deposit Item*. *Deposit Item Receiver* creates one instance of *Receipt Basis* for every new customer. The *Receipt Basis* then holds references to the different *Deposit Items* that the customer has turned in and also to how many of each type. The *Receipt Basis* will thus be implemented using such a component. We now have the first sketch of the implementation of the block *Receipt Basis* shown in Figure 8.8. We will come back to the details of this issue when we consider the implementation of this block.

The blocks that we have now identified will correspond to the modules in which the source code is included. We will later show the files for the block *Receipt Basis*. In this way the blocks become a manner of grouping the source code. These blocks will contain our source code and our object modules (classes). Since different customers may want different configurations, different blocks may be delivered to different customers.

Before we can implement the blocks we must describe in detail how they are to communicate with one another. This is done by

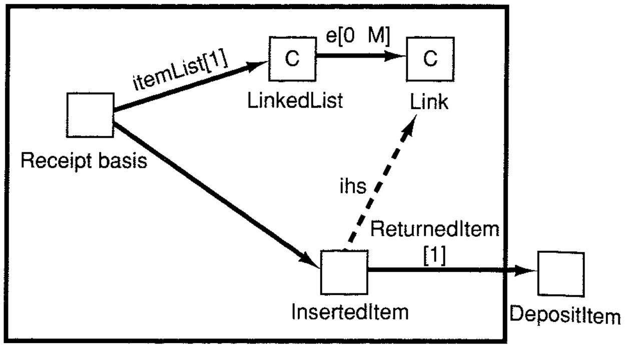


Figure 8.8 The first proposal of the classes that implements the *Receipt Basis* block Classes with a C are ready made components (from the NIH library).

defining the stimuli to be sent between them When we have described the block interfaces completely, we can implement them seamlessly using the rules that describes how the different terms are implemented in the relevant programming language as described later.

8.2.3 Interaction diagram

When we have identified the system architecture we describe how the blocks are to communicate. This is done by **designing the use cases** which were described in the analysis phase. As mentioned before, the use cases constitute the key to OOSE. They control the analysis phase and generate the analysis model; they are designed in construction and define the external requirements on the blocks; they will be tested in the testing phase to make sure the system is correctly built; and finally, they will be the basis for preparation of the system manual(s).

For each concrete use case we will draw an **interaction diagram**. The interaction diagram describes how each use case is offered by communicating objects. The diagram shows how the participating objects realize the use case through their interaction. The interaction takes place as the blocks send stimuli between one another. As we draw the interaction diagrams, we will also define all the stimuli including their parameters sent. The main purpose of the use case design is thus to define the protocols of the blocks

With the use case model we have described what takes place in each use case The analysis model describes which objects offer this behavior, and now in the design model we refine the description of the use cases by showing in the interaction diagram how the objects behave in every specific use case.

The work on identifying the blocks in the design model is performed rather quickly; at best it is done automatically by means of a tool. On the other hand the design of use cases implies a great amount of work. Here we must define exactly how the participating objects shall communicate. The descriptions made in analysis, which are the basis of this work, may now have to be changed which may yield a change in the system structure. This may then give rise to proposals for changes in the analysis model or the requirements model.

Since the work on designing the use cases is done early in construction, we will discover whether the system architecture is good or whether it should be modified at an early stage when the inherent resistance against changes is low. This means that the architecture has a chance to stabilize to a good architecture. Note that this is actually a consequence of the work done in the analysis phase.

When we design a use case we start by identifying the blocks participating in the use case. This is easily done by looking at which blocks offer the use case in the analysis model; the corresponding blocks will be included in the design. This identification is consequently also a completely mechanical process. Furthermore, there may be new blocks introduced during construction (e.g. for the implementation environment) that should also participate in the use case.

The interaction diagram is a type of diagram used for a long time in the world of telecommunications. There it has the same function as we are striving for, namely to describe the communication between different blocks.

Each participating block is represented by a column. These columns are drawn as vertical lines in the diagram. The order between the columns is insignificant and should be selected in the best manner for clarity. The aim is to obtain a good overview. The skeleton of an interaction diagram is shown in Figure 8.9.

If there will be several instances of a block's classes, we can draw the instances either as different columns or as one and the same column depending on which is most legible. All behavior of an object shall be attached to the column representing the actual block.

In the interaction diagram we have drawn a column for the *Deposit Item*. When the use case is carried out, this object will actually be *Can*, *Bottle* or *Crate*. We have consequently also used polymorphism in the interaction diagram; this is a very useful technique.

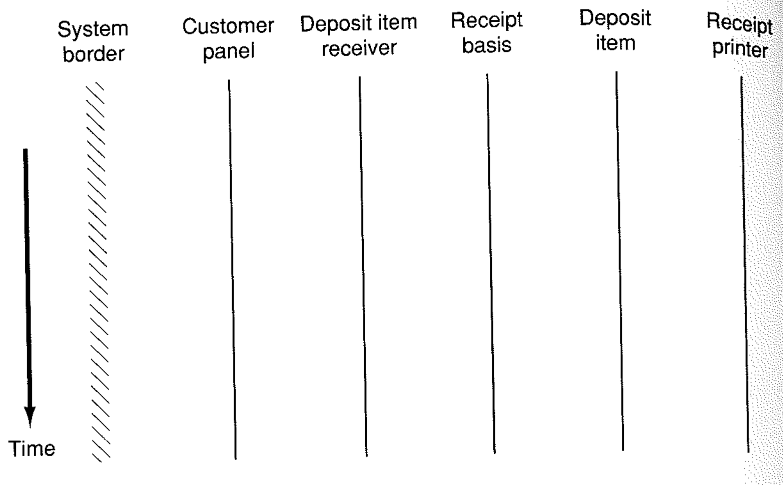


Figure 8.9 The skeleton for the interaction diagram for the use case *Returning Item* consists of columns for each block that participates, and a column for the system border.

In almost all the interaction diagrams we also have a column for the surrounding world, namely for what is outside that we want to describe. We usually call this column the **system border**. This column represents the interface with everything outside the blocks in the diagram, such as external actors, and consequently it can correspond to different interfaces outside the system.

The time axis in the interaction diagram is viewed as going downwards. The use case thus starts with the behavior described at the beginning of the interaction diagram. The time axis of the interaction diagram is not linear, but should be regarded as event controlled The distance between two events in the diagram has no relation to the real time between these events.

At the left edge of the interaction diagram, to the left of the system border, we describe the sequences. This description is textual and consists of structured text or pseudo-code. If we use pseudo-code, constructs that exists in the current programming language could be used This is to ease our later migration to the actual implementation. However, it makes us dependent of the specific language The text describes what is happening in precisely this part of the use case. Such a part we call an **operation**. We also mark the column to which the operation belongs with a rectangle representing the operation. The textual description consequently belongs to the block where the operation takes place and where it shall later be implemented, and whose column therefore is marked at the place of

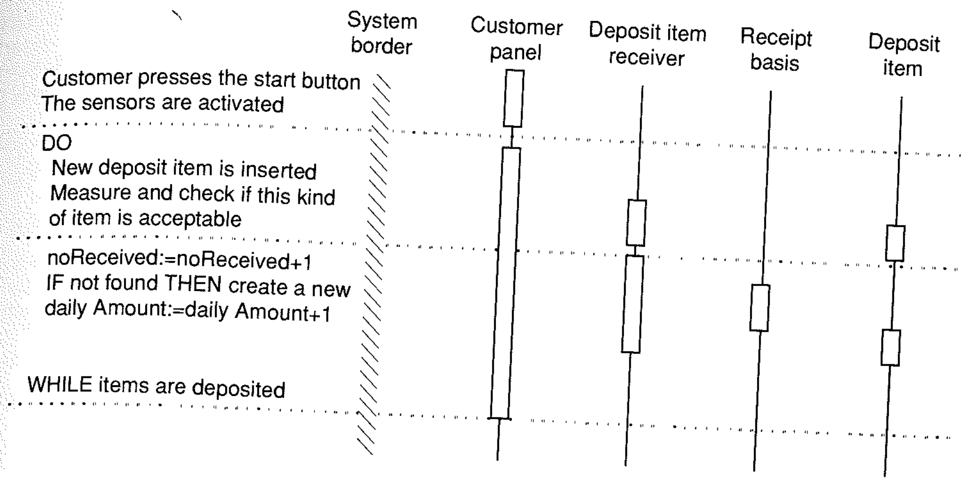

System border
Customer panel
Deposit item receiver
Receipt basis
Deposit item
Customer presses the start button
The sensors are activated
DO
New deposit item is inserted
Measure and check if this kind of item is acceptable
noReceived:=noReceived+1
IF not found THEN create a new
daily Amount:=daily Amount+1
WHILE items are deposited

Figure 8.10 Interaction diagram of the recycling machine with operations in the participating blocks. The time axis points straight down Only part of the use case has been included

the operation. Between operations we may draw horizontal grid lines for clarification purposes This technique could also be used for any other partitioning of the use case. This is shown in Figure 8.10.

Parallel sequences, such as when several processes participate in a use case, can also be described in the interaction diagram. This will be further discussed in Chapter 9

8.2.4 Definition of stimuli

The interaction diagrams are controlled by events. A new event gives rise to a new operation. These events are **stimuli** that are sent from one object to another and initiate an operation.

A stimulus is drawn in the interaction diagram as a horizontal arrow that starts in the column corresponding to the sending block and ends in the column corresponding to the receiving block. Most interaction diagrams (and thus use cases) start with a stimulus from the outside, that is, it is drawn from the system border From the use case description we can do a use case design. Here is the decription of the use case *Returning Item*.

The course of events starts when the customer presses the 'Start-Button' on the customer panel The panel's built-in sensors are thereby activated

The customer can now return deposit items via the customer panel The sensors inform the system that an object has been inserted, they also measure the deposit item and return the result to the system.

The system uses the measurement result to determine the type of deposit item: can, bottle or crate

The day total for the received deposit item type is incremented, as is the number of returned deposit items of the current type that this customer has returned

When the customer has returned all his deposit items he asks for a receipt by pressing the 'Receipt-Button' on the customer panel.

The system compiles the information that is to be printed on the receipt For each type of deposit item its return value and the number of returned items by the current customer is extracted.

The information is printed out, with a new line for each deposit item, by the receipt printer

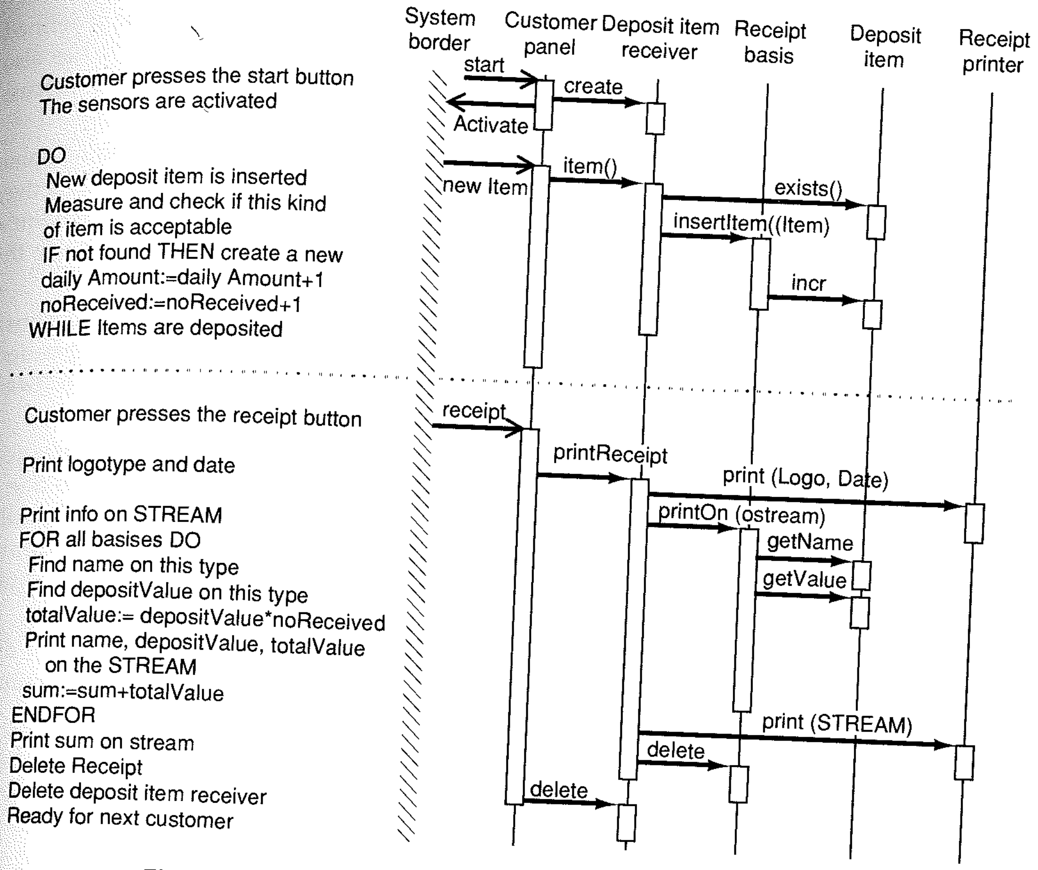
Finally, the grand total for all returned deposit items is extracted by the system and printed out by the receipt printer

From this description we can develop an interaction diagram. In this example the interaction diagram will look as shown in Figure 8.11.

The use case starts as the customer presses the start button. The block *Customer Panel* then activates the sensors that are external to our system Now the customer can start feeding in the return bottles, empty cans and crates. This is solved by a DO...WHILE statement that is ended when the customer requests a receipt. In this loop we measure the item and check if it is acceptable. If so, we increment the attribute that knows how many objects the customer has fed in of this type, and how many have been fed in in total today. Note that the incrementation of the daily number is delegated to the block *Receipt Basis*.

When the customer is ready he or she presses the button for receipt. The *Deposit Item Receiver* receives a stimulus for printing the receipt and starts by printing the logo and today's date. Then it handles over a stream to the *Receipt Basis* that prints the receipt information on the stream. (A stream is a buffer to store characters to be printed.) The *Receipt Basis* puts together the information to be printed and prints it on the stream. Finally, the total sum in favour of the customer is printed. The stream is sent to *Receipt Printer* for printing. The *Receipt Basis* is then deleted and the *Customer Panel* is ready for use again.

The interaction diagram for *Returning Items* is only drawn for the basic course. Naturally, you must consider faults that may occur, for example you may press the start button without submitting any returnable items, but this is not considered here in order to get a survey presentation.



```
Customer presses the start button
The sensors are activated

DO
  New deposit item is inserted
  Measure and check if this kind
  of item is acceptable
  IF not found THEN create a new
  daily Amount:=daily Amount+1
  noReceived:=noReceived+1
WHILE Items are deposited

Customer presses the receipt button

Print logotype and date

Print info on STREAM
FOR all basises DO
  Find name on this type
  Find depositValue on this type
  totalValue:= depositValue*noReceived
  Print name, depositValue, totalValue
    on the STREAM
  sum:=sum+totalValue
ENDFOR
Print sum on stream
Delete Receipt
Delete deposit item receiver
Ready for next customer
```

Figure 8.11 Interaction diagram for the use case *Returning Item*

When working on the design of the use cases you will define the stimuli that each object should be able to receive. The definition comprises the name and parameters of each stimulus. To define stimuli is one of the more critical parts of a development. Since several persons are normally involved in the design of the use cases, the stimuli to one and the same block will be designed by different people. What should be striven for is to reuse existing stimuli as much as possible. Then it becomes much easier to design the objects since the stimuli define the external requirements on each object. When you define the stimuli you should keep the following in mind

- The reusability increases if we have few parameters. Having few parameters will make it easier to understand and also increase the probability that the stimuli are similar to one another. A stimulus with many parameters should be redefined and possibly divided into several stimuli. This point has also been noticed by Johnson and Foote (1988) who also discuss other rules for obtaining a reusable design

- Stimuli that invoke similar behavior shall have the same name regardless of the parameters they contain or which objects they are sent to. Hence there is no conflict between duplicate names. If stimuli have the same names it is easier to see clearly the similarity between the two stimuli.
- The naming of stimuli shall reflect the distribution of responsibilities between blocks. To make each object as independent as possible is one of the basic principles in all software engineering contexts.
- Each stimulus should clearly show the exchange of information, that is, the name must be selected carefully. Naming is one of the most difficult skills in programming and therefore rules for this purpose often exist. The names indicate an intuitive semantics and consequently the name is extremely important to help understanding and finding for reuse purposes.
- Handling and creation of new instances or initiation of new processes are performed in the same manner as ordinary stimulus handling

The description of a use cases is normally divided into **basic courses** and **alternative courses**. The basic course is the most common (important) sequence in the use case. It is always designed first. The alternative courses constitute all other sequences that the use case can take, that is, typically fault handling sequences. When the basic course has been designed, you continue with the alternative courses. What is striven for is to describe as many alternative sequences as possible. The more alternative sequences we describe, the more sequences have been anticipated and hence the more we have increased the robustness of the designed system.

A stimulus can have different semantics, e.g. be either interprocess or intraprocess communication. An intraprocess stimulus is a normal call inside one process. This kind of stimulus we normally call a **message**. A message corresponds to a message in Smalltalk or invoking an operation in Eiffel or C++. That is, it is the normal mode of communication in object-oriented languages. An interprocess stimulus is a stimulus sent between two processes. Such a stimulus we usually call a **signal**. Signals may be either synchronous or asynchronous (the execution of the sender continues directly after the signal is sent). In Ada, the *rendezvous* mechanism for communication

Message
Signal

Figure 8.12 Different kinds of stimuli may be used in OOSE. Messages are intra-process and signals are interprocess communication.

between tasks is synchronous, one task must wait for the other task to finish. In, for instance, CHILL, see CCITT (1984), a programming language used in the worlds of telecommunications, signals are asynchronous; when a signal is sent the sender continues its execution immediately and does not wait for the receiver The receiver may be active when the signal arrives and therefore the signal must be queued at the receiver until processed

Generally, the concepts used in OOSE are generic terms. When we specialize OOSE to a specific implementation environment we should use the concepts of the implementation environment. For instance, when working with Ada, we should talk about *rendezvous* mechanisms instead of signals.

Messages and signals will be used in this book and will be drawn differently in the interaction diagram, as shown in Figure 8.12. Messages are drawn with a closed arrow and signals with an open arrow.

If necessary, the exact semantics of the stimuli should be specified from project to project. For instance, different kinds of synchronizations or time-outs should be specified when necessary. Then again, the chosen operating system or the programming language's properties should normally be used. We will discuss the use of signals further in Chapter 9 In this chapter we will assume that all stimuli are messages

8.2.5 Structure of interaction diagram

Interaction diagrams give the designer a unique possibility of seeing the entire sequence in a use case at a survey level. The designer can therefore quickly get a picture of how the sequence progresses over the objects participating in the use case You can also quickly see what the structure of the use case looks like. This structure often tells us a lot about the realization characteristics of the use case Here we can, in principle, validate the structure of the use case that was defined in the analysis model. We will now look at a use case that is designed in two fundamentally different ways

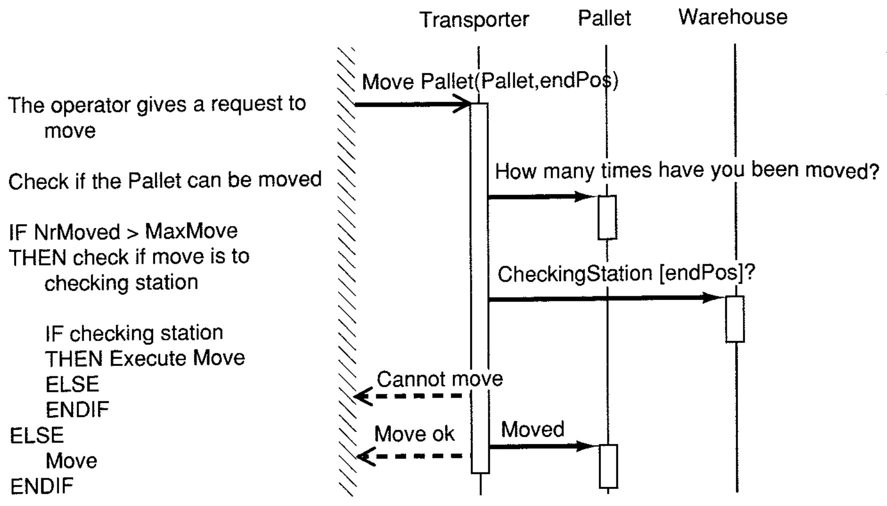

Transporter
Pallet
Warehouse
Move Pallet(Pallet,endPos)
The operator gives a request to move
Check if the Pallet can be moved
How many times have you been moved?
IF NrMoved > MaxMove
THEN check if move is to checking station
CheckingStation [endPos]?
IF checking station
THEN Execute Move
ELSE
Cannot move
ENDIF
ELSE
Move ok
Moved
Move
ENDIF

Figure 8.13 Transport with the *transporter* controlling the detailed flow.

The use case is based on a real application and is built to handle a warehouse with automatic trucks that move pallets. A technician only gives an order specifying where the pallet is located and where it should be moved to. All loads are placed on pallets that can be moved around automatically in the warehouse. Since the pallets are moved automatically, the loads must be checked at intervals, say every tenth move, to make sure that the loads are still stable on the pallet. We must therefore keep in mind how many times a specific pallet has been moved. We will now design the use case to move a pallet. This is done by drawing an interaction diagram for it. We will look at two different variants for implementing this.

Figure 8.13 shows the first case. The use case starts as the technician sends a stimulus to *Transporter*. *Transporter* then asks *Pallet* how many times it has been moved. If it has been moved the maximum number of permitted times we must check whether the move shall be made in the warehouse or if the move is to be made to a checking station. If it is to a checking station we can make the move, otherwise we must not move the pallet

In this interaction diagram we can see that everything is handled and controlled by *Transporter*. This object controls the flow and operates on the other objects, and then it decides what to do.

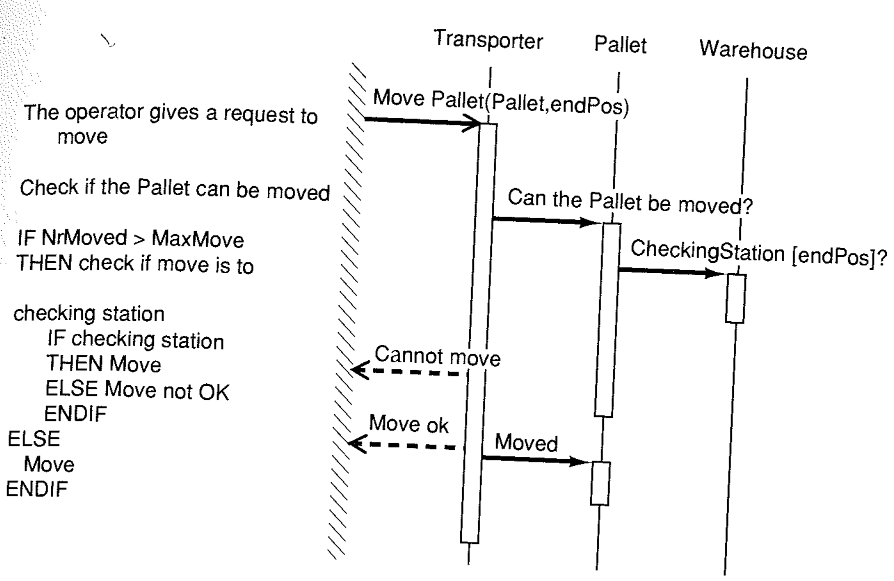


Figure 8.14 A use case design where the load itself knows if it can be moved

This object must therefore contain rules for moving a load. We have a **centralized** structure.

In the second design, see Figure 8.14, we have delegated the decision as to whether the *Pallet* can be moved or not to *Pallet* itself. *Transporter* now only asks whether the load can be moved or not. *Pallet* now checks whether it is permitted to move the load and whether it shall be moved to a checking station or not It only replies yes or no to the *Transporter*, and then *Transporter* moves the load if this is possible

Here we can see that the decisions are decentralized and are made by the unit knowing the conditions, in our case *Pallet* *Transporter* is only interested in whether the pallet can be moved or not, and has now been freed of knowing on which conditions the load can be moved We have a **decentralized** structure

The difference between these use cases is actually fundamental and it has come about already in the analysis phase where we have allocated the use cases to the analysis objects. Which of the use case designs is actually most sensitive to changes? Let us assume that we want to move painted bicycles in the warehouse The bikes must not be moved until they have dried, perhaps for six hours. Then, we must add a check of attributes that keeps track of whether there is a painted bike on the pallet and how long ago it was painted. These

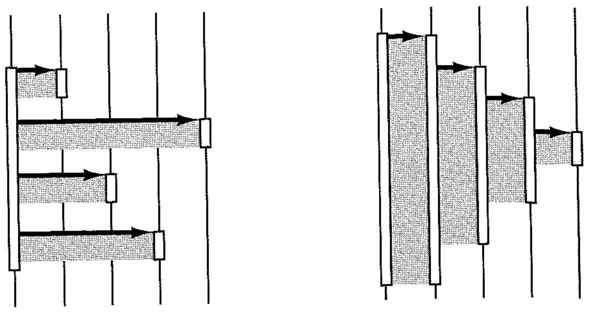


Figure 8.15 The interaction diagrams show clearly what the structure of the use case looks like. The extreme points are the fork structure and the stair structure

attributes belong to the bike, but must be checked by *Pallet*. In the first use case design we must make changes in both *Transporter* (which must ask *Pallet* what it contains) and *Pallet* (where we must add new attributes). In the second design we only have to make changes in *Pallet* (where we add the attributes and the checking required).

In the interaction diagram, we can see the structure of the use cases very clear. The interaction diagram helps us to assess how decentralized the system is We can distinguish two extreme types of structures for interaction diagrams, see Figure 8 15

The first extreme we call a **fork diagram**. This type is characterized by the fact that it is an object acting as a spider in a web and so controls the other objects. Much of the behavior is placed in the controlling object that knows all other objects and often uses them for direct questions or commands

A fork diagram indicates a centralized structure. Typically the control sequence is placed in one object, often a control object. The other objects are typically used as information carriers or as an interface to a user A fork structure often sets great requirements on the designer of the controlling object, since this often becomes much more complex than other objects.

The other extreme we call a **stair diagram**. This type is characterized by delegated responsibility. The included objects only know a few of the other objects and know which objects can help with a specific behavior. Here we have no 'central' object.

Stair diagrams often indicate a decentralized structure. Here each object has a separate task and it only knows the surrounding objects it needs in order to help carrying out this task.

Which of these two types should be chosen? Which is the best? Often it is claimed that the stair structure is more object-oriented and thus better. The more the responsibilities have been spread out, the better. However, this is not always true. What we want is to be able to introduce changes easily and also to design reusable objects and stimuli. Different kinds of changes can of course occur.

To handle changes of how a sequence is actually performed, you should encapsulate the actual sequence and thus obtain a decentralized structure. If you want to change the actual order of the operations, it is better to have a centralized stucture, since then such changes will be local to one object. We thus see that both structures have their benefits. What is crucial is whether the operations have a *strong connection* to each other. A strong connection exists if the objects:

- Form a 'consists-of' hierarchy, such as country–state–city,
- form an information hierarchy, such as document–chapter–section–paragraph–character,
- represent a fixed temporal relationship, such as advertisement-order–invoice–delivery–payment,
- form a (conceptual) inheritance hierarchy, such as animal–mammal–cat

We have found the following principal rules:

- A decentralized (stair) control structure is appropriate when:
  - The operations have a strong connection (see above),
  - The operations will always be performed in the same order,
  - You want to abstract or encapsulate behavior.
- A centralized (fork) control structure is appropriate when:
  - The operations can change order,
  - New operations could be inserted.

This also leads to the conclusion that these two structures very well may be, and indeed should be, combined to yield a stable and robust structure.

We see here that the *ordering* of operations is also a potential source of changes. We thus cannot only regard behavior or data from a robustness view but also need to consider ordering. The natural solution for robustness is normally to encapsulate things that can be changed. Hence we should also be able to encapsulate ordering of operations. That is exactly what is done in the centralized approach, where we have the ordering of operations defined in only one object.

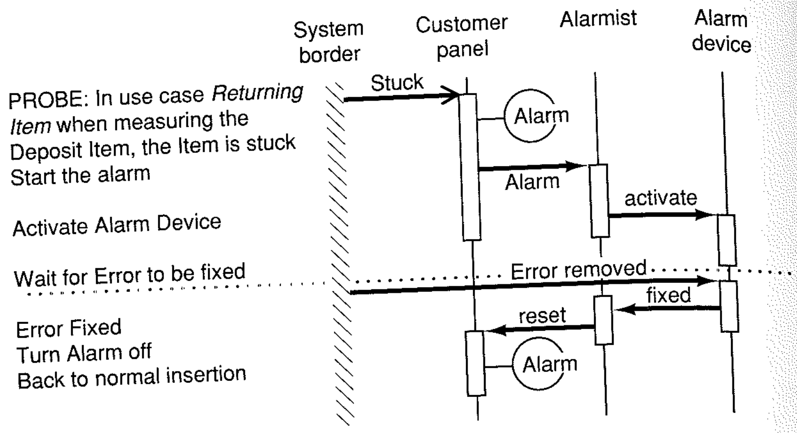


Figure 8.16 Probe position *Alarm* for use case *Item Stuck* in the recycling machine The new behavior is inserted in the probe position

8.2.6 Use cases with extends

One of the most important properties in the use case model is the possibility of extending a use case by means of extends. The association will be the extends association also in the analysis model. We must also be able to describe this behavior in the interaction diagrams. This is done by means of **probes**. A probe is a position in a description where an extended behavior can be inserted. The description in which there is a possibility that an extended new behavior shall be added should not show that a behavior can be inserted. The reason for this is that the original description shall be unchanged due to any inserted behavior and therefore easy to understand You should then not drown in all extensions possible. We have already discussed the fact that extends must normally be implemented by a stimulus sent *from* the object where the insertion should take place.

An extend between two use cases means that when one use case is executing, obeying its description, this use case is extended to follow also the behavior of the description of the second use case. The probe then gives the position in the first use case where the extension will take place In the interaction diagram drawn for the inserted use case, we must specify the probe as exactly as possible. Often an extends association is connected with a condition. This condition is also indicated together with the probe description. Thus we normally give a probe position and the condition. The position is a place in the description of an operation in a block, see Figure 8.16.

The description of the *Item Stuck* use case thus extends the description of *Returning Item*. The probe position therefore refers to a place in the description for *Returning Item*. During execution, the probe gives the place where the *Item Stuck* behavior is inserted.

The use case *Item Stuck* shall extend *Returning Item* when an item is stuck. From the description in the interaction diagram we shall be able to find exactly where the probe is to be inserted. We describe *Item Stuck* in just the same way as other use cases. The description of *Returning Item* has not been affected and hence can be described independently of *Item Stuck*.

8.2.7 Homogenization

The use cases are normally designed in parallel and by several designers who can work more or less independently of one another. This means that stimuli are defined by several different designers. Normally you should agree on the definition rules for stimuli, at least in the project or for the product, but preferably in the entire organization. We have discussed earlier how stimuli can be handled uniformly, but despite this there will be stimuli with the same meaning or purpose but with different definitions. We therefore have to **homogenize** all stimuli when the use cases have been designed. Homogenization can of course also be done as the work goes on with the design of use cases

Homogenization means that we try to get as few stimuli as possible, which should be maximally reusable and easy to work with. What we should strive for is a set of powerful stimuli that can be reused. Sometimes it may be sufficient to change the name of a stimulus or to change the parameters, but we may also have to divide some stimuli or create a completely new set of stimuli. This new set of stimuli must (of course) have the same power of expression as the old set, but should consist of as few stimuli as possible

For instance, imagine you have the following set of stimuli defined on an object Person by different developers:

```
What_is_your_phone_number
Where_do_you_live
Get_address
Get_address_and_phone_number
```

These could be homogenized to

```
Get_address
Get_phone_number
```

We have reduced the number of stimuli, but we can get everything that we got with the first set-up of stimuli. It must be decided from time to time on which level the stimuli should be defined. It is very hard to give any strict rules, but reusability and robustness should be the main criteria

You can compare homogenization with orthogonalization in mathematics, but you should normally not be as strict as to require all stimuli to be orthogonal in some information space. The reason is that we also set the requirement that our stimuli must be efficient. To send a stimulus is perhaps very expensive (in some sense). So it may be preferable to keep a more complex stimulus than to divide it into more homogenized stimuli which will always be used together. An important criterion in homogenization is consequently how the stimuli are to be used The purpose of homogenization is to prepare for a not-too-complex block design – all operations that the stimuli should trigger must be implemented later

In the same manner as we homogenize stimuli, we should review all the use cases to see whether we can find common behaviors that we missed earlier Sub-sequences can perhaps be taken out, and maybe we will find common parts of use cases not discovered during the analysis phase (typical examples are fault handling in different situations)

8.3 Block design

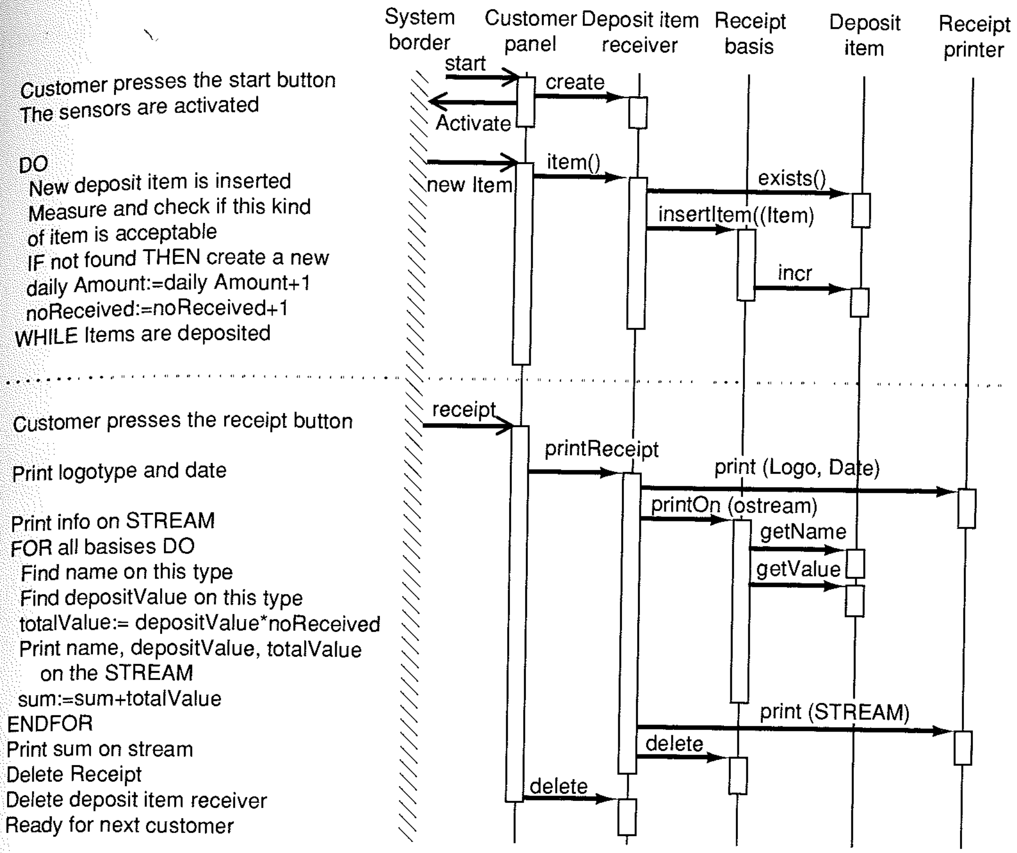
8.3.1 The block interface

When we have designed all the use cases or at least all the use cases for a specific block, we can design the block(s) Through the design of the use cases we now have a complete description of all external requirements of the block. Of course we also have requirements from our analysis model that specifies what each block shall implement. From these requirements we can now design the block.

The actual implementation of the blocks in the source code can start when the block interfaces start to stabilize and are frozen. Often we refer to the source code as the implementation model since it too is a model with a finer granularity than the design model.

When the implementation of blocks starts, normally ancestor blocks should be implemented prior to descendant blocks. Hence, in the recycling machine example, the block *Deposit Item* should be designed prior to *Can, Crate* and *Bottle*

Through the use case design, we have implicitly specified the protocol of each block. By taking all interaction diagrams where a



```
Customer presses the start button
The sensors are activated

DO
  New deposit item is inserted
  Measure and check if this kind
  of item is acceptable
  IF not found THEN create a new
  daily Amount:=daily Amount+1
  noReceived:=noReceived+1
WHILE Items are deposited

Customer presses the receipt button

Print logotype and date

Print info on STREAM
FOR all basises DO
  Find name on this type
  Find depositValue on this type
  totalValue:= depositValue*noReceived
  Print name, depositValue, totalValue
     on the STREAM
  sum:=sum+totalValue
ENDFOR
Print sum on stream
Delete Receipt
Delete deposit item receiver
Ready for next customer
```

Figure 8.17 The interaction diagram for the use case *Returning Items*

block participated and extracting all the operations defined for that block, we will have a complete picture of the requirements of a block. These requirements have appeared during the design of use cases where normally several people were invoved and the requirements are usually not collected in one document The requirements exist explicitly though, and they are connected to the appropriate block. It is therefore simple to extract these requirements mechanically, and this can consequently be done by means of a CASE tool.

Let us see how this looks for a block in the example. We thus go through all interaction diagrams and extract all the operations defined on a block. Figure 8 17 repeats the interaction diagram for the use case *Returning Items*.

From this interaction diagram we can extract the following interface for *Deposit Item*.

```
exists()
incr
getName
getValue
```

and the following for *Receipt Basis*:

```
insertItem(item)
printOn(ostream)
delete
```

Hence, from the interaction diagram, we can develop the first interface to the blocks. This can be expressd in the programming language. Below are the header files of the C++ classes implementing the *Receipt Basis* block We have used in-line expansion where appropriate. Note that we use the design previously discussed for the components, see Figure 8.18 The components used are LinkedList and Link from file linkedlist h Note that class *InvertedItem* is only supporting the actual implementation

```
// File receiptbasis h

#include ⟨linkedlist h⟩
#include ⟨stream h⟩

class InsertedItem : public Link   {
  private:
    DepositItem *returnedItem;
    int          totalNumber;
  public:
    InsertedItem(DepositItem *di)   {returnedItem = di;
                                      totalNumber = 1};

    DepositItem *getItem()   {return returnedItem};
    void          incr() {totalNumber++;}
    int            getTotal()   {return totalNumber;}
};

class ReceiptBasis   {

  private:
    LinkedList *itemList;
    float      sum;
  public:
    ReceiptBasis()   {itemList = new LinkedList; sum = 0;}

    void      insertItem(DepositItem*);
```

```
        void      printOn(ostream&);
        ~ReceiptBasis()   {itemList->removeAll();
                                  delete itemList;}
    };
```

By going through all interaction diagrams, we will get the complete interface to a block. We will also have the first description of the operations from the text in the left margin.

Hence, at this stage it is possible to *freeze* the interface of the blocks. By freezing the interface of the blocks we can start block design activities in parallel.

In addition to the requirements set by the use case design, there are also other requirements on the block. These requirements can also be identified from the analysis model where we find for example which attributes and attribute types the block should have. Other requirements may come from the requirements specification, for example real time requirements or memory space requirements.

In many cases a block will correspond to exactly one class and its instances. Then it is easy to map the interface of an object in the design model to a specific class interface in the source code. Other times, as we have seen, many classes must be used to implement one block. In most cases, the reason is to handle the implementation environment, but other common cases include the implementation of attributes, use of components and the separation of common functionality to abstract classes.

To handle the interface of a block we can use concepts from the programming language as shown above. However, the blocks are also an abstraction and an encapsulation mechanism. Thus we want to express the interface of a block more explicitly and also be able to encapsulate it's actual implementation. We do this by defining

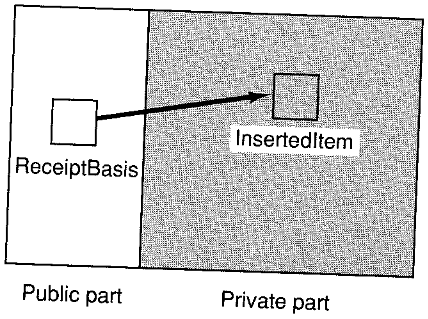


Figure 8.18 The block *ReceiptBasis* with its public and private classes.

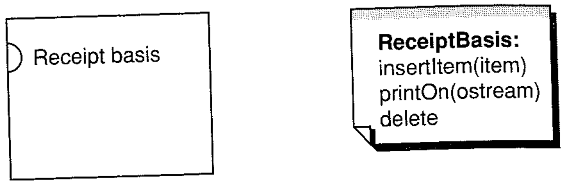


Figure 8.19 The block *ReceiptBasis* and its exported type. The type's specification is documented in some way clarifying its role to be played.

public and **private** object modules. The public object modules can be accessed from the outside of a block and are thus exported from the block, but the private object modules cannot be accessed from the outside. In the example above we will only make the ReceiptBasis class public, and encapsulate the details of the blocks's internal structure. This is illustrated in Figure 8.18.

This is thus a way of handling the complexity of the source code, to hide internal implementation details. However, in this technique we must determine early which classes should be public, and thus take early decisions concerning the internals of the block. What is actually needed is just to be able to specify an interface to a role that the block should export. We need then only define **types** on a block that can be implemented in various ways. Using this technique we could define these types as **responsibilities** and **contracts** as disccused by Wirfs-Brock *et al.* (1990). This is illustrated in Figure 8.19.

8.3.2 Object behavior

As an intermediate level for the internals of the object, before looking at the actual implementation, we may use **state transition graphs**. Their purpose is to provide a simplified description that increases understanding of the block without having to go down to source code level, and to provide a description that is less dependent on the selected programming language. In these graphs we describe which stimuli can be received and what shall happen when a stimulus is received. For a discussion on state transition graphs, refer to the box.

On states and state machines

States are actually dependent on what you want to describe and are in that sense relative. For our recycling machine we can have states indicating that it is full of bottles, that it is busy or idle, that the standard height of cans is set to 20 cm, and so on We see here that there is not only one state in the machine but many To be absolutely correct and complete, a state is the union of all values describing the present situation With a state we must be able to fully recreate the present situation We can quickly see that it is impossible to describe each state since this would mean that we would have to define an almost infinite number of values If we want to describe a state in a computer, we must define what each memory cell contains, and this is impossible in practice in a development

When we describe how our blocks shall work internally, it is suitable in our case to use state machines, especially to implement objects that emanate from control objects

In order to continue and to discuss states we differentiate between internal states and computational states The **internal state** is what characterizes all the values of our variables that are important for our description, as well as variables relating to the application and variables that are included due to the implementation environment The **computational state** describes how far we have come in the execution, as well as the potential future execution In principle we only want to describe an object behavior in terms of computational states It is essential to understand the fact that underlying the computational states we have internal states that consequently contain the information we use to move between the computational states The internal states are also used to describe the state of the object.

A computational state transition graph thus describes the class When a class is instantiated, an instance is created that follows a path in this graph throughout its lifetime. For a class that can be traced to a control object, we normally have several states in this graph, whereas for a class that emanates from an entity object, we usually have only one computational state The state indicates to some extent the potential that this object has at that moment We can clearly see which stimuli the object can receive, and we can also see what will happen in these cases When the object receives a stimulus, it will follow a path in the graph and enter another state (that may be the same from which it started) The path selected is consequently dependent on the received stimulus, but also on the internal, underlying state.

States and state transitions can be described in many ways. We will here show some different notations that can be used For the example we will use a stack. A stack is a linear structure to store elements All insertions and deletions take place at one end An insertion is performed by a push operation and a deletion is done by a pop operation. Another name for a stack is a LIFO (last-in–first-out) list, since the last inserted element is the first to be deleted The most obvious way is to use ordinary state transition diagrams, see Sudkamp (1988), often called Mealy machines. In a Mealy machine

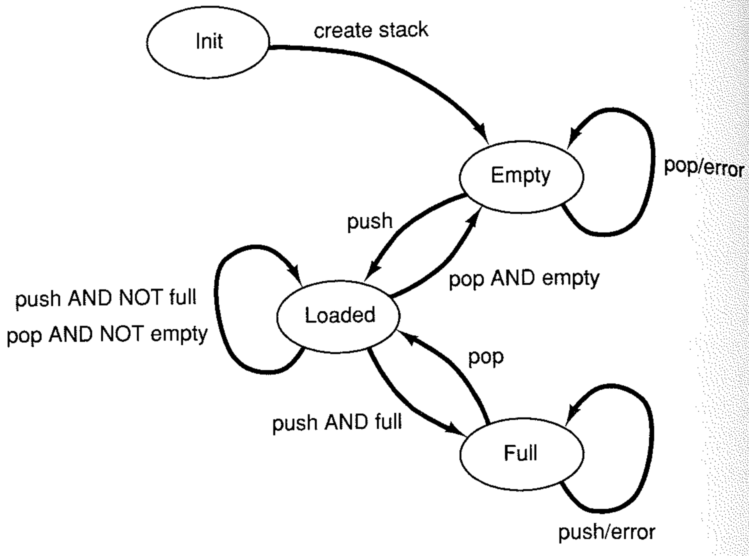


Figure 8.20 A state transition graph for describing a stack.

each transition must be well defined by the stimulus received. However, this is not plausible in our graphs since it also depends on the interval state. To make it practical we need to express conditions at the transition edges. This technique is illustrated in Figure 8.20.

Hence a transition is dependent on the stimuli received and on underlying internal state expressed with some condition. This can be described so as to place the conditions on the edges between the nodes, as in Figure 8.21 Not until a certain computation has been made can we see which path to take. Mealy machines do not have this facility to describe a path selection during a state transition, and we have therefore modified the semantics.

This description could also be made in textual form as shown in Algorithm 8.1 Here each state is represented by a textual construct as all computation. A similar notation has been described by Berzins and Luqi (1990).

Other techniques can be used to describe the computation of an object, for instance state tranistion graphs as used in structured methods, see Ward and Mellor (1985), the notation used in JSD, see Jackson (1983). SDL, the Specification and Description Language, a CCITT (1988) standard used for a long time, is still another technique for this notation. To describe a specific operation, further techniques such as data flow diagrams can be used. Which technique is chosen for this description is actually not very important; the important

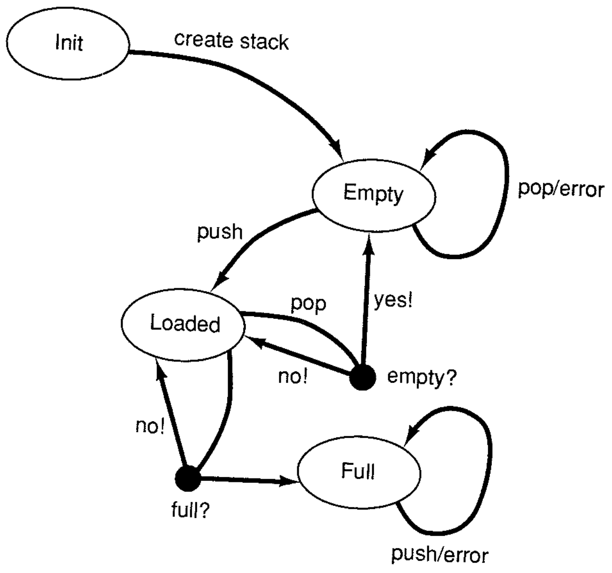


Figure 8.21 State transition graph with conditions on the edges.

thing is that it helps you to abstract the actual coding. For our examples here we will use a notation inspired by SDL, but in principle any notation could be used. Our notation is described in Figure 8.22.

These symbols are connected to describe the computation of an object. The **start** symbol indicates where the creation of an object starts. A **state** symbol shows a stable state of the object between different operations. **Send message** performs an operation on another object and corresponds to a **receive message** in the object to be operated on. To describe the return from such an operation, a **return** symbol is used. Sending and Receiving signals are denoted by the corresponding symbols. The actual computation, not including explicit communication, is denoted by a **Perform task** box. The **branch** symbol is used to describe alternatives in the execution and the **Destroy** object describes where an instance is deleted. The **Label** symbol is used to make the diagrams easier to read and is often used to describe loops and the like

Every operation must start beneath a state, or the start symbol, and end in a (possibly the same) state. This means that after each state we must have a receive symbol, either a receive message or a receive signal. The return symbol is only used if we have a return value. If no return value is explicit the default condition is that a receive message is ended just before the next state. Let us look at an example. In Figure 8.23 we have used this notation to describe the

Algorithm 8.1 The description of stack using a textual notation

```
machine Stack
  state init
    input createInstance
      nextState empty
    otherwise error;

  state empty
    input push(e)
      do store[e] on top
        nextState loaded
    input (e)pop
      print "Stack is empty."
    otherwise error;

  state loaded
    input push(e)
      do store[e] on top
      if isFull then
        nextState full
    input (e)pop
      do return top,
        delete top
        if isEmpty then
          nextState empty
    otherwise error;

  state full
    input push(e)
      print "stack is full"
    input (e)pop
      do return top,
        delete top
      nextState loaded
    otherwise error;

endmachine;
```

same computation as we have previously done to describe the stack. The '--' in the state symbol refers to the immediately previous state.

This notation maps directly onto the interaction diagrams presented earlier, see Figure 8 24. In (*a*) we have extracted an operation from the interaction diagram. This operation will be described by the state transition in (*b*) This sequence could be generated

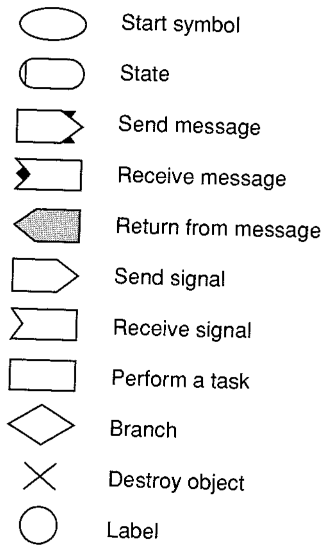


Figure 8.22 The notation used in the state transition graphs in this book

automatically from the interaction diagram. However, in the general case the exact semantics could not be generated. Additionally, the interaction diagram describes one specific path through the graph, but the graph must describe all paths. Therefore only path skeletons can be generated and the developer must relate the paths and states to each other

In this manner we can describe the classes of all objects in the system The complete state transition graph for *Deposit Item* extracted from this interaction diagram is shown in Figure 8.25

An object that will perform the same operation independent of state when a certain stimulus is received we call a **stimulus-controlled** object. The order in which the stimuli are received in the object is of no significance; it performs the same task every time it receives the same stimulus. This is the way objects normally act in object-oriented contexts; the operations can (in principle) be executed in an optional order. This is sometimes called 'the shopping list approach' (Meyer 1988) since it reminds us of a shopping list; the order in which you pick out the items is uninteresting, what is important is that all the items can be picked out. Objects that implement entity objects are usually stimulus-controlled.

Objects that select operations not only from the stimulus received, but also from the current state, we call **state-controlled** objects. These show a strong temporal relation between the received stimuli and the stimuli they are prepared to receive. Objects that

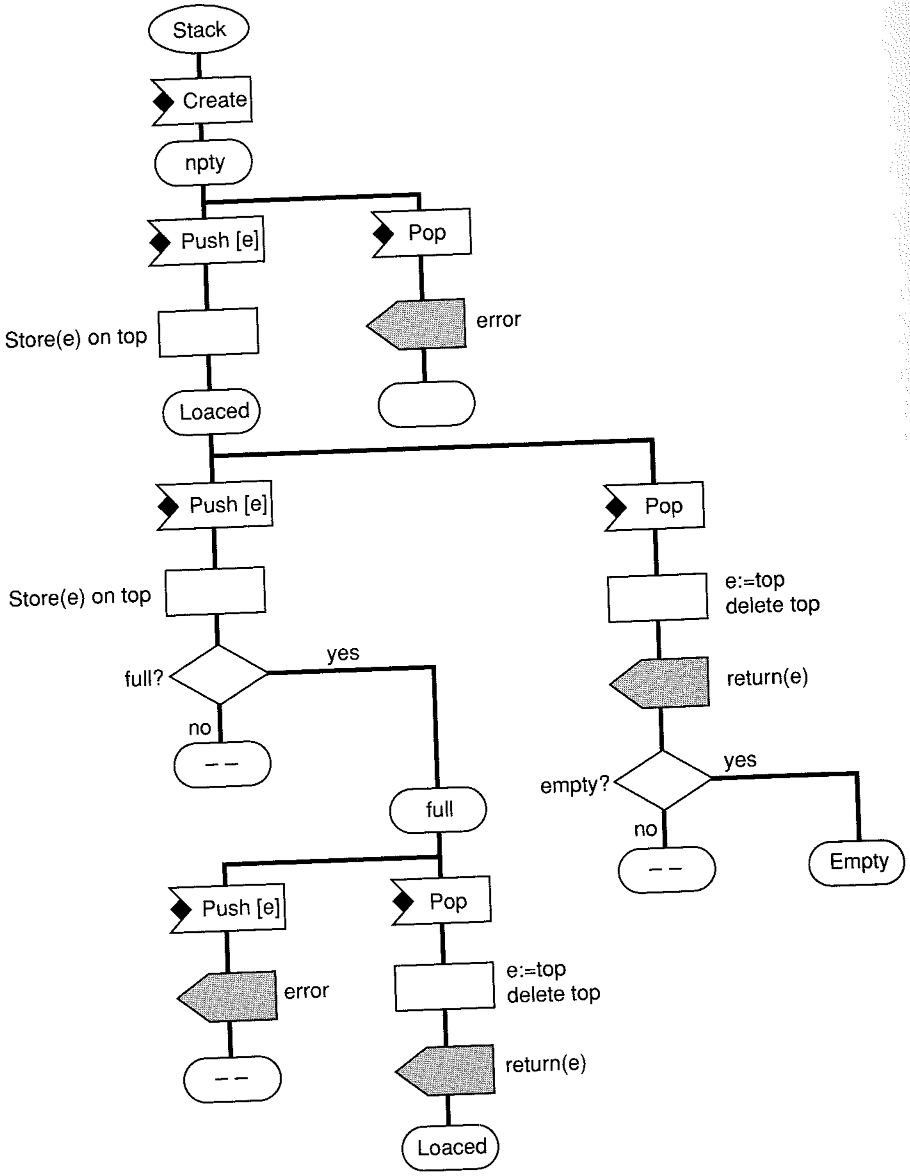


Figure 8.23 The stack described with the notation for a state transition graph used in this book.

implement control objects often have a tendency to be state-controlled. This, of course, does not mean that they are independent of the stimulus received, but they should be regarded as having a strong connection between the stimuli they can receive and, above all, in

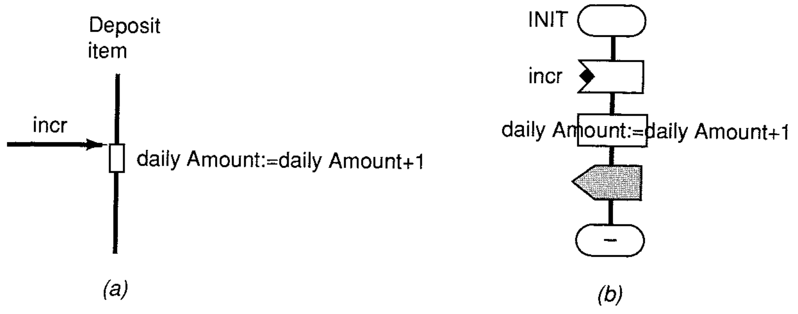


Figure 8.24 The operation in (*a*) will yield the state transition described in (*b*)

which order they are received. In the same way stimuli-controlled objects also have an internal state (how otherwise would the information be represented internally?), but their behavior is controlled more by the stimuli they receive than by the information they contain. In the recycling example, the *Deposit Item Receiver* block is a state-controlled object. Its state transition graph is shown in Figure 8.26. When a stimulus is sent, we have used the common object-oriented notation for the receiver, for instance RB create means that create is sent to the block RB (*Receipt Basis*).

We see that this graph is more complicated. For stimulus-controlled objects and for simple state-controlled objects it is not necessary to draw a state transition graph since this would be of little value owing to their simplicity. The benefits of these graphs increase the more complicated the behavior is. Especially in behavior where the states are important, these graphs will be of great help. In Chapter 14 we will see the benefits of these graphs for complicated sequences

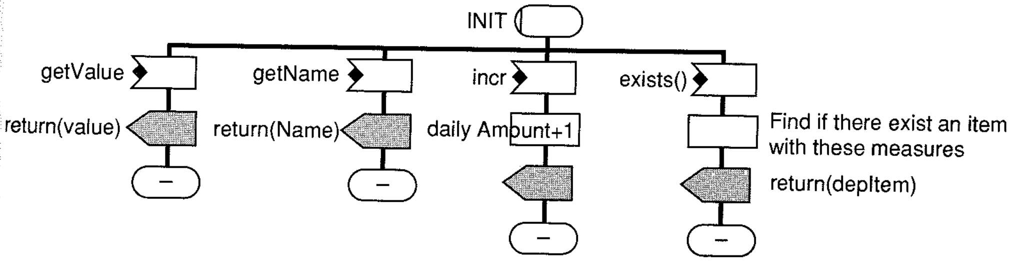


Figure 8.25 State transition graph for *Deposit Item*

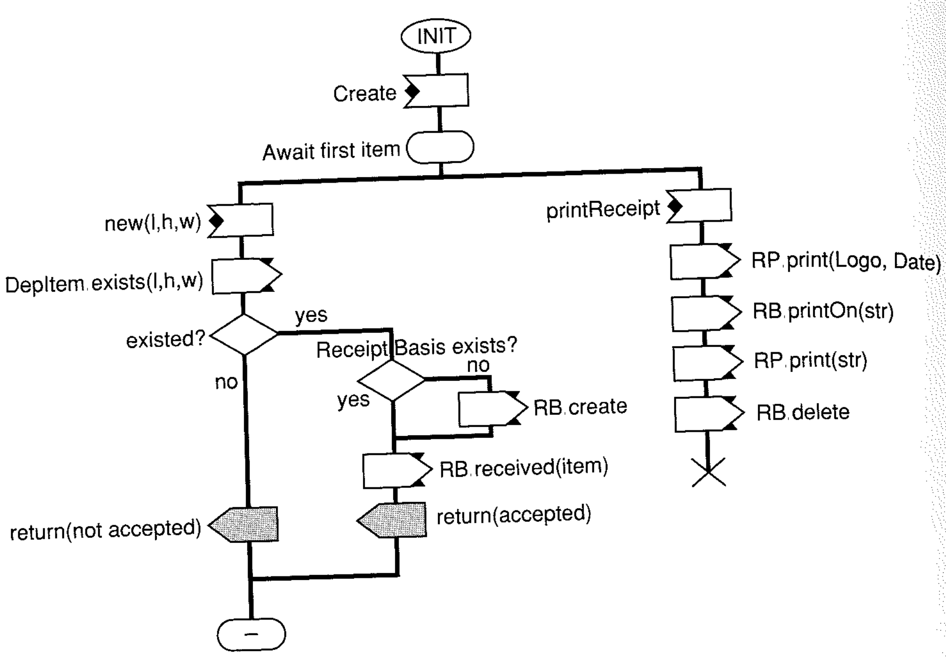


Figure 8.26 The state transition graph of *Deposit Item Receiver*

We should also describe the state transition graph of the *Receipt Basis*. See Figure 8.27. Here we have several task boxes. These boxes represent the computation that is encapsulated in the object.

8.3.3 Internal block structure

When we have identified these properties of the block, we can outline the internal block structure The internal structure of the block will consist of object modules as previously discussed. If our implementation language is object-oriented, these object modules will in reality become classes in the language, but if we do not have classes naturally in the language, then they will be module units in the source code. We have already discussed the internal structure of a block.

The blocks used in the design model are actually an abstraction mechanism for these object modules. Often the block can be implemented as only one class, but otherwise we may need a handful of classes for each object. Generally, there will be more classes than blocks. However, these will often be components, but, for instance, attribute types that you have defined may also be new classes. Other

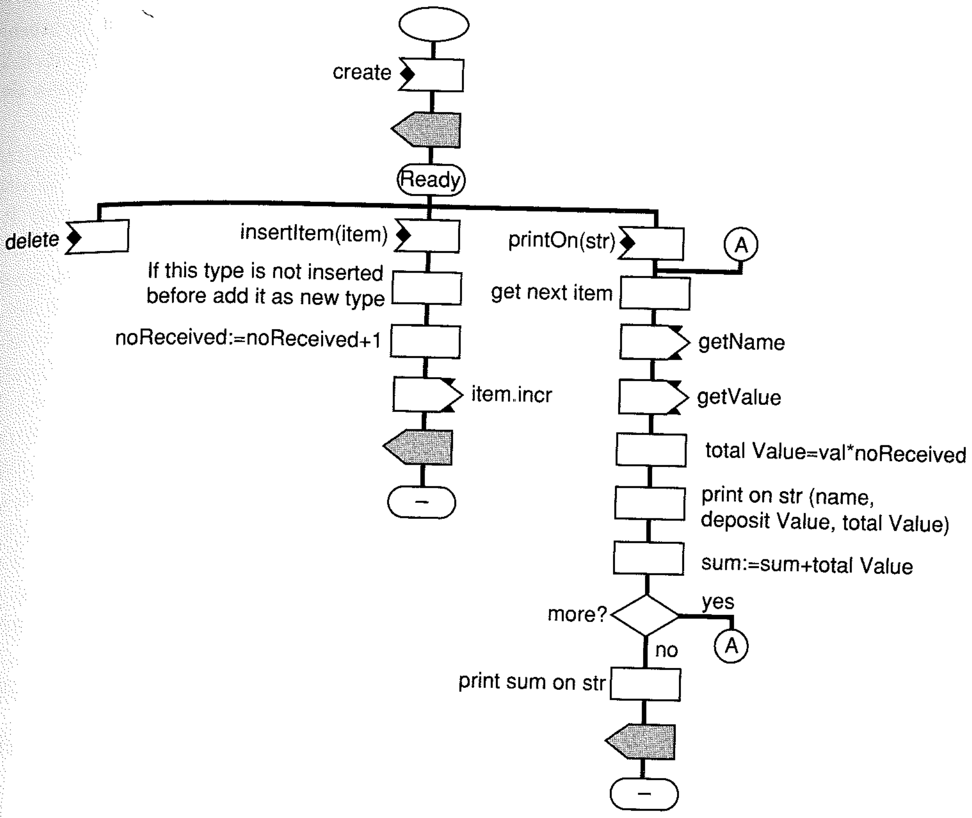


Figure 8.27 The state transition graph of *Receipt Basis*

reasons for introducing new classes can be seen in the internal structure for *Receipt Basis*. There we used components, but we also introduced a new class to handle the reference to the inserted item.

When the structuring of a block is done, we define the classes in the system. Here it is of course of interest to design reusable and high-quality classes. In the same manner as we talked about homogenization between stimuli, we also want the classes to be homogenized. This means that we do not want two classes offering similar functionality, unless they are related through inheritance. Additionally we want classes to offer functionality that is internally strongly coupled. For instance, if half of a class' operation accesses half of the instance variables and the other operations operate on the other variables, we should consider splitting it in two classes. Other heuristics of good class design include judgement of the potential reusability value of the class. Will a change make the class more reusable to others? However, it takes time to design a good class, and it is not always efficient to develop every class in the system to be as general as possible. A common estimation is that it takes something like 5–10 times longer to design a component class than an ordinary class.

The most straightforward way to implement a block's interface is to have one class implementing the entire interface. This is often an appropriate solution. However, there may be times when it is better to split the block interface to be implemented by two or more classes. An example of this is when the block apparently plays several roles. Then there may be reason to separate these roles to be implemented by different classes, although this should not be taken as a rule. Another typical example is when you encounter a situation in which the actual handling of instances is placed in the same block as the block that represents the instances itself. In the recycling example we have such a case. Think of *Deposit Item*. This block should receive the stimulus exists, which invokes an operation that should look among all items to find if there exists an item that fits the appropriate parameters. On the other hand, the block should also receive stimuli like incr, which is sent to one specific instance. We thus have encapsulated the handling of instances and the actual instances in the same block. In some object-oriented languages, like C++ or Smalltalk, you can place behavior as class operations. Another solution is to have one class representing the actual instances and one class that implements the management of these instances. Which way to use is more a matter of programming techniques and philosophy, and also of the actual environment.

Sometimes one block's functionality will not need to be implemented as a class and instances of this class. The reason in this case is that there is no need for any instance variables. A bunch of free operations may then be sufficient. This is possible in C++, but in some other object-oriented languages it is not. Then they could be placed as class operations.

Components are important to use in structuring the internals of a block. Sometimes components must be adapted in order to be used and therefore they may also give rise to new classes that act as a shell for the component but, as said before, more about components in a subsequent chapter.

8.3.4 Implementation

Now that we have described the blocks using state transition graphs, we can seamlessly implement them in the programming language. Already today there are systems that automatically translate from

SDL descriptions into source code. Normally, however, we will still need human beings to make this final transition to source code.

The adaptation to the programming language will be made according to a specialization of the construction process to the selected language. The specialization to programming languages will not only occur in this late phase, but we have seen that during the entire design process we will be forced to consider how to solve problems that will appear due to the implementation environment in question.

Note that we have maintained traceability from our analysis model all the way down to the source code. When we read the source code we can also directly trace what gave rise to it in the analysis model. This property of a system is very valuable, and when a system is changed during its life cycle, this property is invaluable for the engineers who work on the maintenance and development of the system.

The specialization to the programming language describes how we translate the terms used in the design into the terms and properties in the implementation language. Although the methods described here are general and largely independent of the actual language, all languages will have their specialities during the implementation. In this discussion we will consider programming languages that are object-oriented. We will also discuss differences relative to languages that are not object-oriented. The ideas described in this book can be used for various languages. The techniques are at a higher structuring level of the system and thus not dependent on particular constructs in any specific language.

Many projects and organizations have **coding rules** concerning how the source code should be written. The purpose of these rules is to obtain a uniform source code that not only facilitates understanding and reading, but also achieves portability between different environments. The specialization described here should not need to influence these rules, but the coding rules should be applied regardless of this specialization.

Object-oriented languages

Let us start by looking at languages that offer classes and inheritance. Here we include among others C++, Smalltalk, Simula, Eiffel and Objective-C. These languages have classes, and these classes will correspond to the object modules that we have designed.

It would be desirable to group these classes in the same manner as they were grouped earlier, in for example blocks and subsystems. The programming languages usually have no further terms for this grouping (except for Simula), and here you will have to use the local standards. Usually you use different files and file directories for this purpose. One block could then correspond to a file and a subsystem to a directory. We have previously seen how one block could be specified in a C++ file However, in modern environments there exists support for this partitioning.

Inheritance between object modules is straightforwardly mapped onto inheritance between classes in the programming language On the other hand, inheritance between blocks can be more complicated. Here we want all the functionality in one object also to be accessible from another. This problem could often be solved by direct inheritance between the main classes in the objects.

Attributes that have been identified during the analysis phase or that appeared during the design work are usually implemented as a variable of a specific type. If the type has a complex data structure, we often have to make a new class of the attribute type, otherwise it could be made of a more primitive type (in, for example, C++ a struct). We saw an example of this in the header files were we wanted InsertedItem to have one attribute totalNumber storing how many of this type the customer had returned:

```
class InsertedItem : public Link {
  private:
    DepositItem *returnedItem;
    int   totalNumber;        // here is the attribute of type int
```

Acquaintance associations are normally implemented in the same way as attributes, namely as ordinary instance references or pointer variables to instances. The same could apply for communication associations, but these could also be represented by ordinary local variables in an operation or by nothing at all if a return value is used as an object reference. For these associations we also want to handle stimuli. Stimuli normally become messages or ordinary routine calls Below is shown how the incr stimulus looks in the code:

```
ii->incr();      // ii is a pointer to the inserted item
```

Note the possibility here of using *virtual* (deferred in Eiffel, subclass responsibility in Smalltalk) operations for classes. This means that in the ancestor we only declare that a certain operation will exist, whereas the implementation is done in a descendant. This enables different implementations for different classes, a facility that is frequently useful

We see that all concepts are directly translatable to properties in the programming language. In Table 8.1 we have summarized the

Table 8.1 Traceability for C++

Analysis	Design	Source code C++
Analysis object	Block	1..N classes
Behavior in object	Operation	Member function
Attribute (Class)	Attribute (Class)	Static variables
Attribute (instance)	Attribute (Instance)	Instance variable
Acquaintance ass	Acquaintance ass.	Instance variable
Communication ass	Communication ass	Reference to a (member) function
Interaction between objects	Stimulus	Call to a (member) function
Use case	Designed Use Case	Sequence of calls
Subsystem	Subsystem	File

traceability for C++ Note that this is one alternative; there may exist other alternatives in each specific case.

For state-controlled blocks, we often need to keep track of the current state. The basis is often a description from a state transition graph One solution is to use CASE or IF statements for this, another is to use polymorphism, and then the virtual procedure frequently is very useful. It should be noted here that we do not use CASE statements to check the type or similar; this should definitely be solved by using polymorphism instead, but only to implement state machines. In, for instance, Smalltalk we could also concatenate the state with the actual stimuli to find the correct operation. An example of this is given in Chapter 14.

Instantiation is handled in the manner offered by the language. The instantiation is normally described in the interaction diagrams. In this context we should also mention that the memory management must be looked for as previously discussed. In languages such as Smalltalk, Simula and Eiffel you normally use automatic garbage collection, whereas this is done by the user in, for example, C++ and Objective-C.

Data hiding is an important property used in object-oriented contexts. All attributes should be made private unless there are special reasons for the contrary Accessing these variables should be made by special operations of the form getX. This is standard in Smalltalk and normally also in Eiffel and C++, but in Simula you must indicate explicitly that they are hidden

We can consequently see that we maintain the traceability all the way to the source code by means of these rules Below is the actual implementation of the member functions specified above

```
// File receiptbasis cc

#include 'receiptbasis h'

void ReceiptBasis::insertItem(DepositItem *di) {
  if (itemList->isEmpty()) {
    InsertedItem *ii = new InsertedItem(di);
    itemList->addLast(ii);
  } else {
  int found = 0;
  InsertedItem *ii = (InsertedItem*)itemList<<first();
  do {
    if (ii->getItem() == di) {
      ii->incr();
      found = 1;
    } else
      ii = (InsertedItem*)ii-nextLink();
    } while (ii && !found);
    if (!found) {
      InsertedItem *ii = new InsertedItem(di);
      itemList->addLast(ii);
    }
  }
 di->incr();
 }

 void ReceiptBasis::printOn(ostream &os) {
   char *name;
   float value;
   float totalValue;
   InsertedItem *ii = (InsertedItem*)itemList->first();
   do {
     name = ii->getItem()->getName();
     value = ii->getItem()->getValue();
     totalValue = value * ii->getTotal();
     os << name << value << totalValue <<\ 'n';
     sum += totalValue;
     ii = (InsertedItem*)ii->nextLink();
   } while (ii);
   os << sum << 'n';
  }
```

Non-object-oriented languages

If our language is not object-oriented, we must translate the fundamental concepts, like inheritance and encapsulation, in another manner. Here we will survey some of these properties and concentrate on what differs from what has been said above about object-oriented languages.

The most common languages used include object-based languages such as Ada and Modula-2 and traditional programming languages such as C, Pascal, Fortran and COBOL. Here we will primarily discuss Ada, but this, together with what has been said earlier, should be adequate to indicate how to transfer the terms to other languages. The great difference compared with object-oriented languages lies above all in the inheritance concept.

An important term in Ada is the package concept. It offers the possibility to express hiding. In Ada we translate each object module into a package. Since packages can import other packages (using the WITH construction) we can also use them here to group blocks and subsystems.

The most difficult problem is how inheritance should be translated. The simplest solution is to get a preprocessor where these terms have been added to the language. Such preprocessors exist for Ada (e.g. Classic Ada), for C (e.g. C++), and so on. With this solution the entire problem is solved, but this is not always possible to do.

Another solution is to simulate inheritance. This can be done in several ways. The easiest way is simply to import the package placed above in the 'inheritance hierarchy', encapsulate its type definition and simulate the inheritance by linking together several calls. This is sometimes called **call-through**. It can cause problems for long inheritance hierarchies (performance), but it could be a solution when the inheritance hierarchies are short. If you use generic packages this linkage can be made by using a certain routine as the relevant (generic) parameter when instantiating the package. In this case linkage is done during compilation and produces no overhead on execution. Another way is to use derived types.

One alternative to simulate inheritance is to use the inheritance hierarchy only as a method of grouping your information and thus avoid having to write slavishly the source code according to it. The result will then be that we will not see the inheritance structure in the source code. The abstract blocks will then have no correspondence in the source code.

Attributes will normally be made into variables, sometimes of a predefined data type but often as a self-defined type, for example a composite data structure. Class and instance variables can also be expressed in Ada, but this may be more difficult in other languages.

Acquaintance associations normally become a reference to the object or a variable of the object's type. Cardinality [1] or [0..1] can be expressed with a simple reference. Cardinality [M] or [1..N] is expressed by a vector of references or as a dynamic structure (e.g. list) of references.

Table 8.2 The analysis concepts are treated in the design model and onward to the source code The example shows this traceability for Ada.

Analysis	Design	Source code Ada
Analysis object	Block	Package
Behavior in object	Operation	Procedure or task
Attribute (class)	Attribute (class)	Variables (global in package body)
Attribute (instance)	Attribute (instance)	Variables (part of private type)
Acquaintance ass.	Acquaintance ass	Variable reference
Communication ass.	Communication ass.	Existence of procedure call
Interaction between objects	Stimulus	Procedure call or entry call
Use case	Designed use case	Sequence of calls
Subsystem	Subsystem	Package

Stimuli normally give rise to a routine call. In Ada communication associations normally give rise to a variable of the associated object's type. Instantiation is made in different ways depending on the nature of the object. For example, blocks can be instantiated by simply declaring new variables of the relevant type. Note that these objects do not survive in the same manner as objects in object-oriented languages. If you want to instantiate for example control objects, this is normally done with one or several procedure calls.

Also in this case we can see that we maintain the traceability all the way to the source code. Table 8.2 summarizes this traceability. Please note that there may be alternatives in specific cases.

8.3.5 Implementation of probes

The explanation why the extends association gives rise to a communication association was given earlier. This association is associated with a probe which specifies where the sequence should be inserted. By a probe we mean a position in a use case description or its accompaning interaction diagram where, when a use case follows this description, another behavior can be inserted. Since the original description must be independent (unknowing) of the new, inserted behavior, all of the control structure should be placed in the block offering the inserted use case.

Unfortunately we cannot solve this with today's programming languages, and so we are forced to deviate from this request. Instead the probe must be implemented in the block where the sequence should be inserted. We thus must add one variable to hold a reference to the inserted functionality and also the stimuli that should be sent to this object. This is illustrated below:

```
class CustomerPanel {
public:

  void stuck();
  void reset();

private:

  alarmist myAlarm;

}

void CustomerPanel::stuck()
{
  myAlarm alarm;
}
```

We here assume that an instance of *Alarmist* (myAlarm) is created in, for instance, the constructor of the class.

In a more traditional language where we do not have the class concept, we usually implement the probe with a specific procedure. In the example this would look something like this:

```
procedure Receive_Item(measure:measure_type);

DO
  Item := getItem;
  probe_Alarm;  \\ here is the probe_procedure
  CASE Item : Item_type OF
    Can:     newCan(Item);
    Bottle:  newBottle(Item);
    Crate:   newCrate(Item);
  ENDCASE
WHILE not Receipt;
```

And the probe procedure would look like this:

```
procedure probe_Alarm;
BEGIN
  IF stuck THEN
    BEGIN
    alarm;
    wait_for_reset;
   END;
 END:
```

In this manner we have hidden what the probe really means. When a change needs to be made to the inserted sequence, we do not have to change the procedure that realizes the *Returning Item* use case, but only the probe procedure.

Today there is no other way of resolving probes in the most common programming languages There are some indications, however, that similar problems have been solved in some systems, for example TMS (Doyle 1979) which is implemented in KEE3, IntelliCorp Inc., for example. Another solution can be found in the programming language LOOPS, see Stefik *et al.* (1986). The problem is also discussed by Jacobson (1986)

8.4 Working with construction

We have now discussed how the ideal analysis model is refined and adapted to the implementation environment to reach, finally, the source code level. This process is seamless as it will lead to the implementation in a straightforward manner We will here discuss some topics that were briefly mentioned in the previous presentation.

As stated several times, our design must be adapted to the real environment in which our system works. All conditions for our implementation emanating from somewhere other than the analysis phase are called the implementation environment. This environment comprises the conditions coming from the target environment in the form of, for example, existing hardware and distribution, and also the requirements that come indirectly from the development environment in the form of the selected programming language and its conditions.

The implementation environment also includes requirements that can be traced to the requirement specification in the form of performance requirements and existing resources. Also requirements such as that the system shall use some other systems are included in the implementation environment. This environment is in short everything disregarded in the analysis phase and that now must be looked at. We say that we **specialize** the construction process to these different environments Specialization thus means adapting the design to the relevant implementation environment.

8.4.1 Existing products

A common requirement from the implementation environment is that the system must use **existing products**. Even if the limit for extending the existing product and thus making a further development is fluctuating, it is still two different problems. In one case the existing product is the same system, but in an older version, and in the other we build another (new) system that only makes use of the existing product because it already exists.

This situation is very similar to that of using completed components, which will be discussed in a chapter of its own, and the difference is above all the size and thus complexity of the existing product. To build with components also differs from this solution; components are used as powerful units that we combine into objects. To use an existing product is different; here we must first make an analysis of the product and adapt our design to it.

An assessment must be made as to whether you want to use an existing product and thus are forced to adapt your design to it, or whether you will not use the existing product and decide to develop this functionality yourself. The advantage of the latter is that you have a homogeneous system that should be easier to maintain since everything is described in a similar manner, rather than having two systems that perhaps are described differently and thus look very different. The advantage of the former is that it requires few resources since you need not devote resources on developing this part; the only thing required is to adapt the interfaces between the two systems. Experience tells us that in the long run it is very expensive to change fundamentally a system architecture, and this should therefore be avoided so far as possible. You should use the same design idea for an entire system, see Lawson (1990).

A common problem is that you want to develop a system that has not been designed according to an object-oriented method and this involves great adaptations to the existing system This problem is difficult and often requires much **re-engineering**. See the discussions on this subject by Dietrich *et al.* (1989) and by Jacobson and Lindström (1991).

One of the (most important) criteria when deciding whether you should use something old or develop something new yourself is to consider the testing costs. It is hoped that an existing product has been used earlier and it should therefore already be well tested, whereas our own system must be tested thoroughly to minimize the number of faults in it. Another important decision is how well you know the existing product and how much resources are needed to get to know it. Perhaps some project members have been involved in its development and it may therefore be rather easy to become familiar with it. Otherwise it may take a long time, especially if the documentation is poor, before you have learned this product and how to use it.

8.4.2 Abstractions

When you develop systems on an industrial scale you must at all time be able to handle the system complexity. This is the important issue for abstractions on different levels. We have seen how we have used the blocks to encapsulate and thus abstract actual implementation details. In this manner we could group several classes in the actual system to one block in the design model.

The block structure used in the example is appropriate for smaller systems, but is by no means satisfiable for larger systems. Then we must find some way to group these blocks into larger units. In a normal system it is not unusual that we have hundreds of blocks. This becomes impossible to handle and assumes a complexity that is hard to manage.

To handle this complexity we use **subsystems** as discussed in Chapter 7. The subsystems defined in the analysis model should serve as the base also for a division into subsystems in the design model and in the implementation. To be flexible for larger systems, we can have subsystems within subsystems.

These subsystems should become the management unit in the organization. A subsystem should thus offer a protocol to other subsystems. We do this by exporting or making certain blocks public to other subsystems as previously discussed. The interface of these blocks then will form the interface of the entire subsystem.

The lowest level of the subsystem, **service packages**, often groups a related functionality. This level is then used as a base for system configuration. Such subsystem will either be delivered entirely or not at all. The subsystems are thus indivisible. The subsystems can therefore have a functionality that is optional in the system. The system can thus be delivered with or without certain subsystems. This means that the subsystems are atomic units used in, for example, discussions with customer and product planning.

The work flow illustrated in this chapter can be applied recursively also to the subsystems Hence it is fully possible to draw interaction diagrams between subsystems on various levels. This is often an appropriate technique when doing further development of a system. Then you could design a subsystem using only the interfaces of related subsystems and you only use the blocks or classes that are made public from these subsystems.

In this manner we have a technique to decrease the complexity on the object level In the same manner there may be times where you also need to reduce the complexity of the stimuli defined. In some of the interaction diagrams presented here we have used this technique Normally a stimuli would be specified exactly as it will look in the source code, but we could, for instance, postpone the definition of actual parameters and define these later when the stimulus picture starts to mature. Another way of reducing the complexity is not to show stimulus interaction encasulated in a block, such as stimuli sent to components or predefined stimuli in a class structure. We could also, but with some doubt, abstract the stimuli to have one abstract stimulus that when implemented will give rise to many stimuli sent to the same block.

8.4.3 Development is incremental

The picture we have given here is that the process of construction is straightforward and an easy road to follow. However, this is not true The development is an incremental process and many iterations must be done before all stimuli are defined and the interfaces can be frozen. Further, when implementation is started, changes of the definitons of stimuli will occur. There you will notice that you must get a reference to an object as a parameter. These iterations will occur and there is no point in rejecting them initially. What is important is to find a way of handling them in the project. We will disuss this topic more in Chapter 16. We will only give some good advice here:

- Start with the construction early, preferably at the same time as you start working on the analysis model. The first step in construction is to identify the implementation environment and this could be done in parallel with analysis so you have it ready when the actual construction starts.
- You have surely noticed that the design model is a refinement of the analysis model with regard to a certain implementation environment. When should you do the transition? You could use the interaction diagrams in the analysis model as well as

state transition diagrams. If you have done a very detailed analysis model, the degree of refinement needed may be very small, unless the implementation environment will give you problems. When to do the transition must be decided from time to time It is important to decide this early, and the decision should be based on the result from the identification of the implementation environment.

- If you are inexperienced in OOSE, try to take one shot through all the models initially. In this way you will get a much better feeling of how the models relate and how output from one process will work as input to another. By working in this manner you will also get some feedback helping you to decide when to make the transition from analysis to design. Additionally, you will deal with potential problems early, that you may have to solve before going into construction with full strength.

Iterations may thus exist on several different levels of granularity. When you work on the design this may give rise to changes in the analysis model. Still smaller iterations can mean that after an inspection meeting you may change something in the design, for example in which object some functionality shall be placed. Iterations are of course also made on the level where two designers perhaps agree to change the type of a parameter in a stimulus. The lowest level of iteration is probably an individual designer who simply makes changes in his or her own classes.

8.4.4 Further issues

We have now described briefly both analysis and construction. The presentation has been an overview to illustrate the central features and the flow of work Among the simplifications we have made we can mention:

- Documentation rules have been entirely ignored. Real situations require such rules. In real projects we use document instructions that specify these rules. There are different document instructions for different types of activities.
- The discussion only reflects the method; process issues are left out. In real development there is of course much more interaction between different parts. This interaction has been formalized, as discussed in Appendix A, and the idea is illustrated in Chapter 15.

- Other activities also take place during these phases, such as configuration management and reviews However, this has been omitted entirely here These issues are extremely important in a real development We will discuss some of these issues in Chapter 15
- We will devote a whole chapter to tests later in this book Here we will only mention that the tests of the blocks and object modules are normally done by the designer who designed them, whereas testing of the use cases work and the system as a whole is done by special testers, often in a special test department. Tests of classes, blocks, use cases and the system will be discussed in Chapter 12.

8.5 Summary

In construction we carry out the design and implemetation of the system. This work is based on the output from analysis. The analysis model describes the system under ideal circumstances where no consideration is given to the actual environment. The purpose with this model is to gain a robust and maintainable structure over the entire system life cycle. In construction we must adapt this ideal model to the prevailing conditions.

Initially, in construction, we must thus identify the actual implementation environment. Here we must investigate all prevailing conditions for the realization of the system This includes considering for instance how the DBMS should be integrated, how processes and distribution should be handled, the constraints from the programming language, what components libraries to use, and the incorporation of any graphical user interface tools. It also includes organizational issues such as distributed development, competence areas of the staff, and market issues such as early delivery of some subsystem. The result of this identification should be a strategy for how to handle all these issues in advance. This work can be done in parallel to analysis.

When the analysis model starts to mature, the implementation strategy should be added to this model. This will give us the first design model. Here we show how the DBMS is incorporated, how the system distribution is solved, and so on This will give us the first real approach to the actual system architecture Since we want this architecture to be stable and robust, it is important to make as few deviations from the analysis model as possible

The design model is then refined using the use cases specified during the analysis. By doing use case design, using the special technique of drawing interaction diagrams, we will develop step by step the complete interface specification of each object. Here we will define each stimulus sent between objects in the system. These specifications can be further refined through a state transition graph of each object showing the object behavior in more detail.

The objects are then structured and implemented using one or several object modules. These object modules correspond to classes in an object-oriented language, but if the language is not object-oriented they will correspond to the module concept in the current language. The use of software components is important in this structuring.

The implementation is straightforward in the language. An object-oriented language is preferable since all important concepts used in OOSE are directly mapped onto these languages. If the language is not object-oriented, some deviations must be made. However, an object-oriented structure is fully possible even for systems implemented in non-OO languages.

To handle the construction for real system development, we need abstraction mechanisms on several levels. The block is such an abstraction mechanism for the classes in the source code. Maybe the most important concept involved in handling the complexity is that of subsystems. These subsystems will partition the design model and can be used to define explicit interfaces for the subsystems in terms of block interfaces or class interfaces.

9 Real-time specialization

9.1 Introduction

The development of advanced industrial real-time systems is one of the major areas of applicability for OOSE. As with all solutions, the problem philosophy established for the first version sets the tone for implementation. OOSE does not bind the solution space of real-time applications at the beginning. That is, the problem can be analyzed, to a large extent, independent of the artifacts (operating systems, languages, etc.) that will be used in its realization. The analysis phase proceeds from the requirements and the functionality to be provided (expressed via the requirements model). From this base, various alternative realizations can be obtained from the selection of 'appropriate' operating system structures, programming languages and so on.

In order to establish a degree of commonality for our discussion of real-time systems, let us first introduce a model for the broad class of industrial real-time systems as portrayed in Figure 9.1

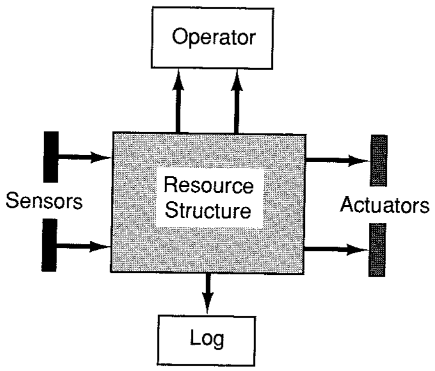


Figure 9.1 Abstract model of industrial real-time systems

In general we can state that the **sensors** and **actuators** provide respectively a view of the application behaviors in the external real-time environment and the means of controlling the external environment behavior(s) The **resource structure** (i.e hardware and system software processes) can be distributed or centralized. By **process** we mean, in this context, a structural and behavioral unit.

The real-time system may (optionally) have some means of observing what is going on (externally and internally) as well as for controlling processing from an **operator** interface Further, there may optionally be some means of keeping a history of what has transpired via some form of **logging** media.

9.2 Classification of real-time systems

In relationship to Figure 9.1, we can identify two major categories of industrial real-time systems, namely those that have **hard** deadlines that must be met, otherwise a catastrophe can occur, and **non-hard** systems where services are provided in real-time and while important, a catastrophe will not occur if immediate service is not provided for all requests by a certain deadline. The control of a modern aircraft belongs to the hard category; whereas a digital telephone exchange is a typical example of the non-hard category.

In the scale of application processes that are to be implemented we find critical processes, essential processes and non-essential processes as identifed by Ramamritham *et al* (1989). The deadlines for **critical processes** must be met **Essential processes** have deadlines, but missing these deadlines will not cause a catastrophe. On the other hand, **non-essential processes** may miss their deadlines without any effect in the near future, but they may have an effect in the long term if not executed (for example, maintenance and bookeeping functions).

We can also categorize processes according to their periodicity. That is, **periodic processes** are executed at regular intervals; whereas **aperiodic processes** are to be executed at any arbitrary point in time.

From these views of processes and based upon the deadline requirements, we can further characterize hard versus non-hard real-time systems.

In **hard real-time systems**, deterministic predictability of processes execution is the essential property. We must be able to guarantee processing of all critical periodic and aperiodic processes. While it is possible to dimension the system for periodic processes, the accommodation of aperiodic processes typically provides a major challenge. Further fault tolerance is essential in this type of system.

For more on hard real-time systems, see Stankovic (1988) and Stankovic and Ramamrithan (1988).

In **non-hard real-time systems**, quality and performance is measured in terms of service provided The execution properties are stochastically distributed based upon the quantities of resources available and the loading of the system While it is quite possible analytically to place bounds on such systems, the development of an optimal scheduling strategy is impossible in the general case.

In applying OOSE to non-hard real-time systems, the specialization is based on the functionality and the degree of service achievable within the resource structure. On the other hand, in applying OOSE to hard real-time systems, a more direct relationship between the execution properties of processes within selected or alternative resource structures and the deadline requirements must be taken into account. The timing requirements as well as the execution property parameters are taken into account during design and implementation.

9.3 Fundamental issues

In developing industrial real-time systems of the hard or non-hard variety there are three fundamental issues that must be taken into account; namely the view of **processes**, the means of **communication** and the method of **synchronization** These issues are typically treated in the behavioral description of the problem as well as in the implementation environment and resource structure.

When strong semantics are provided at the behavioral description level for all of these fundamental issues, they radically affect the selection of appropriate resource structure (hardware and system software processes). All programming languages and most formal methods as well as most contemporary software methods have explicit or implicit means of dealing with these three fundamental issues. When these views are congruent or at least congenial to the views of the implementation environment, all is well. However, when a semantic gap exists between the behavioral description and the implementation environment, typically long lasting complexities are built into the solution that will affect its lifetime and its ability to continue to provide predictable results and satisfactory service as well as maintainability and extendability.

A good example is the usage of Ada, where the semantics of the *rendez-vous* provides for a particular (orthogonal) view of processes (tasks), communication and synchronization. Matching Ada's strong semantics to the problem and implementation environment leads to difficulty, see Lawson (1990)

In the area of software methods, we can for example consider HOOD (see Chapter 16) which is based on Ada In this case, the solution domain that is reasonable from HOOD is affected by its Ada orientation.

While the concepts used in OOSE provide a useful means of identifying processes, it does not provide any *a priori* strong semantics for communication and synchronization. Thus, while the problem is addressed at a structured high level, specialization provides necessary adaptions in order to match the semantics of the problem with those of the implementation environment.

The real-time requirements are passed from the requirements specification to analysis and further are used in selecting potential implementation environments (if there is a choice) Based on the execution properties of the resource structure(s) (hardware and software processes), parameters are extracted which are fed into the construction process and used as a basis for determining whether predictability requirements can be met.

Varying real-time system requirements have an impact on the work to be performed in the various OOSE activities. We will consider the more important aspects of these impacts here. The analysis process primarily involves the collection and structuring of the real-time requirements, while in the construction process the requirements are implemented in varying ways. Based upon the degree to which the actual implementation environment can be influenced, different potential model modifications influence the actual system design. However, it is essential to retain a strong logical coupling between the application problem and the actual implementation environment in order to achieve the goal of traceability. The testing process is possibly the most problematic, since it is often extremely difficult to test a real-time system. Therefore verification methods should be applied in the preceding phases in order to guarantee that (especially hard) real-time deadlines are met.

9.4 Analysis

The use cases, specified in analysis, provide a very strong tool for capturing real-time requirements of various kinds. At this early stage, it is possible to attach both hard and non-hard requirements to use cases. An example of a hard requirement is a control use case that must be completed within 100 ms. An example of a non-hard requirement is that 10 000 subscribers shall be able to use a telephone exchange simultaneously.

By associating a time attribute to a sequence in the use cases, we are able to document hard real-time requirements. These requirements will then naturally become relevant during later phases since the use cases form the thread through all activities of OOSE. We thus have a traceability of these requirements and also a technique to verify that these requirements will be built into the system.

Time attributes may be related to either periodic or aperiodic processes, as discussed earlier. In the periodic case, we also need to attach the period or frequency associated with the time attribute, and in the aperiodic case we may want to indicate the maximum response time. In our previous example of the recycling machine, for example, we may specify that when a customer pushes the receipt button, it should take no longer than 1 s before the printing of the receipt is initiated.

We may wish to attach information on possible concurrency to the use case model. This may occur in two different ways: The first type of concurrency occurs *within* a specific use case, that is multiple activities within a use case may be performed simultaneously. The other type concerns concurrency *between* use cases That is, several use cases may be performed by the system in parallel This form of concurrency actually involves instances of use cases, since it is possible that one type of use case may have several active instances at the same time.

Hence real-time requirements of a system are often naturally attached to use cases, thus making the requirements traceable during all phases of development. Further, the requirement information contained in the use case model can later be exploited in the verification of system behavior with respect to the specification.

In the analysis process, the use cases are further structured in the analysis model Even for real-time systems, this structuring should be done from a logical perspective. That is, real-time requirements should normally *not* affect the structuring. Later, during construction, we may need to modify this structure for these real-time requirements. The reason for this is, as stated several times earlier, that we want the system structure to be stable and robust and this is best achieved by focusing entirely on the problem initially, and not on the circumstances for its realization

When structuring the analysis model, the real-time requirements initially attached to the use cases, when meaningful, may be attached to the objects Thus we provide traceability of the real-time requirements into the objects. This assists in verifying the requirements in later design and implementation phases. However, in some cases, the requirements may be attached to a sequence in the use case and this sequence may be allocated to several objects.

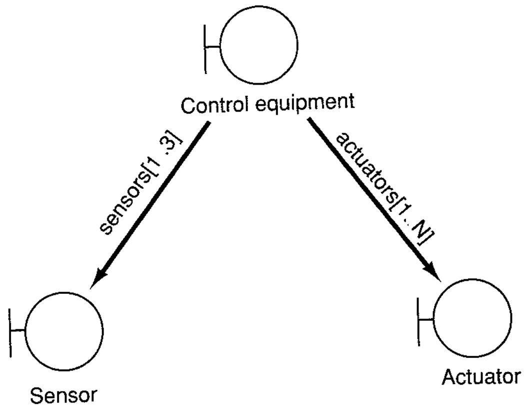


Figure 9.2 Modeling the interface with several associated interface objects.

The sum of the times it takes for the objects to carry out their tasks must then be less than or equal to the time associated with the sequence in the use case.

Real-time requirements often come from the externals of the system. The sensors and actuators of the system are then often tightly connected to these requirements. These will thus normally be the interface to the actors of the system. Note here that we may thus have actors only receiving stimuli from the system in, for instance, a control application.

If the sensors and actuators of the real-time system environment are **external** to the system, we normally model the interface to them as interface objects. We then specify an interaction **protocol** between the system and its sensors/actuators. This protocol may be specified in advance (possibly standardized), but it may need to be specified as a part of the system development Since interface objects may be built using a hierarchic structure of other interface objects, complex interfaces may be built principally as illustrated in Figure 9.2.

Alternatively, sensors and/or actuators as well as other hardware may be viewed as an internal part of the system, this may be modeled using the techniques described in the analysis chapter. For example, we may view a temperature gauge as an entity object. Through this entity object we can read the current temperature value using an operation on the entity object.

9.5 Construction

During construction, we consider the real-time requirements in relationship to the target environment. Although the target environ-

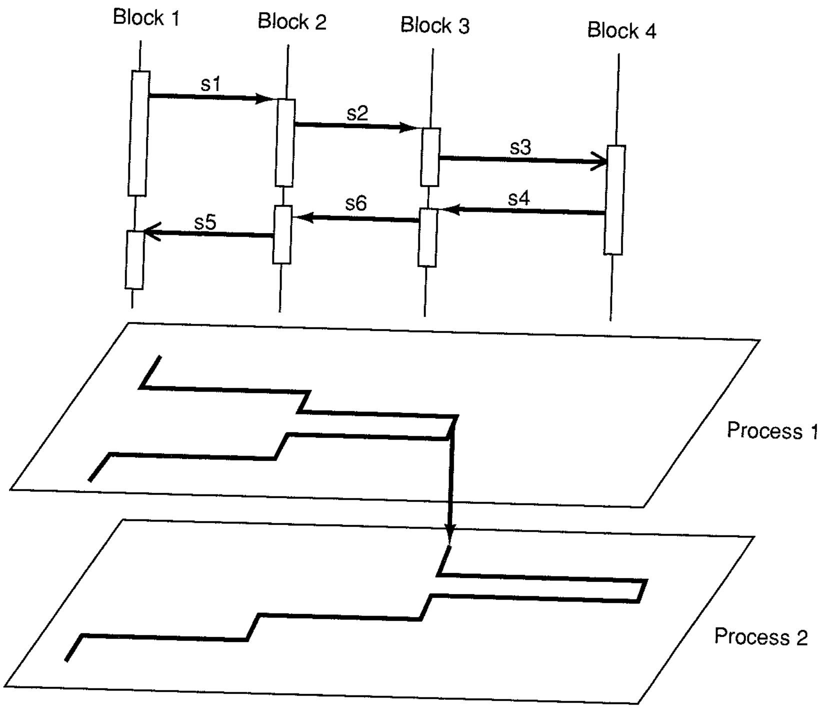


Figure 9.3 The behaviors in the objects are associated to certain processes Note that objects and processes are orthogonal in the general case s3 is a signal, while the other stimuli are messages.

ment is often given *a priori*, it is essential to consider how the target environment can be adapted to fit the object structure, so that the structure is minimally distorted.

Real-time systems do not necessarily include concurrent processes. However, we will discuss processes in this context anyway since traditionally concurrent processes and real-time systems are often treated together. The reason for this is that real-time systems are often based on process concurrency in the target environment. Behaviors in the use cases are then mapped onto the individual concurrent processes. It is essential to note that it is *behavior* that provides the basis for this division and not the objects. In the general case processes are orthogonal to the objects, that is, one process can involve several objects, and one object may be involved in several processes, see Figure 9.3. However, in almost all practical cases one object will be allocated to only one process

To use different processes requires that the target environment supports the process concept. The processes are normally handled by the operating system which manages the process scheduling, synchronization and so on Processes in the operating systems context usually have separate memory address spaces. This means that execution in one process cannot directly access information in another process. Hence references to instances cannot be sent between processes, since only one process has the instance in its memory address space To send an instance you therefore need to copy it and send the copy. Thus in practice you will have only one process handling a specific instance in the system This makes the orthogonality of objects and processes in reality hard to work with.

It is sometimes possible to simulate processes with a shared memory address space. These are often called **threads** or **light-weight processes** The common address space is one of the benefits of such an approach. Additionally, since all threads execute in one 'heavy-weight' process, the inefficiency coming from operating system overheads, for example replacing address translation tables or processor cache memory, is eliminated. Using threads, you handle scheduling and other process-related topics yourself. The use of threads is often an attractive approach in combination with object-orientation since it is possible to share instances between different threads in a straightforward way

If processes execute on one processor they will not be executed in parallel, even if the developer has this view of the execution. Processes may also be allocated to several processors which will give a true concurrent execution However, so long as interprocess communication is handled over several processors, the developer can often view the processes irrespective of which processor they execute on.

Parameters from the target environment often strongly affect the semantics of the concepts of process, communication and synchronization. To ease the transition to implementation, the actual target environment's semantics of these concepts should be used, and therefore OOSE needs to be specialized to use the semantics of the implementation environment If concurrent processes should be used, the operating system must of course support this. Then the semantics used in the operating system also should be adopted in construction. Many operating systems supports different views of processes. Not only may there be support for threads and processes, but also different kinds of these. Which technique(s) to choose should be decided early and then used consequently in the construction.

When introducing the stimuli, we defined two types, messages and signals. The signal type is used for interprocess communcation. The exact semantics of a signal is not specified, but should normally be as used in the current implementation environment. This means that the signal type could be further specialized for expressing different semantics, like timeouts, balking or remote procedure call (RPC).

Process synchronization should also be taken from the semantics used in the actual operating system. The use of properties such as mutual exclusion, semaphores, monitors, locks, different scheduling algorithms, process priority that are available in the operating systems, must be decided upon for each system. Here again, the concepts used should be specialized to fit the semantics of the actual implementation environment, for example *rendez-vous* semantics when working with Ada and UNIX semantics when working with UNIX.

The modeling concepts used in OOSE can be used to **identify the processes** used in the system. We will discuss here some of these approaches Strategies for decomposition of the system into processes in an object-oriented system do not differ essentially from strategies in more traditional systems. Quite the contrary, an object-oriented structure is often more straightforwardly decomposed into processses than a structural decomposition. The reason for this is that processes fit very well into the general ideas of object-orientation: internal states, well specified interfaces, data abstraction, information hiding and the like.

The most obvious reason for introducing processes is that the system is in a *parallel environment.* Since events from the environment can occur at different speeds, frequencies and orders, we often need processes that receive these stimuli. Here we use the behavior of the actors, and also the specifications of the interface objects that receive stimuli from these actors, to identify these processing requirements. We could then assign one process to each interface object or each group of interface objects that must execute in parallel. Possibly one process for incoming stimuli and one process for outgoing stimuli is required. Buffering of stimuli must also be considered in this context.

This identification of processes from the interface objects must then be viewed with regard to the use cases. How should the use case continue its sequence in the system? There may be processing of the incoming stimuli that should not lock the interface processes, and thus *loosen the coupling* between the interface and the processing. Then it may be necessary to assign new processes for this internal processing that can execute simultaneously with the interface process. A concrete example of this approach is the situation in a telephone exchange, where a common solution is having one process receiving digits from a subscriber and another process analyzing them.

Possibly, we must protect *shared objects* The use case may continue and affect, for instance, an entity object that should be accessible in several different use cases and possibly from several different interface objects Such objects must then be protected by some sort of mutual exclusion, that is, we must ensure that only one process can affect the object at a time. This problem can be solved in various ways For instance, monitors or semaphores could be used in the operating system. The introduction of new processes that handle requests to the objects is another possibility. An even better solution is to let the operating system handle atomic transaction with mutual exclusion.

Time-critical functions are typical for hard real-time systems. To guarantee that we will meet these deadlines, use cases that involve them therefore often must run as separate, high-priority processes. The scheduling of processes is done by the operating system and it is not always possible for the programmer to influence this. Additionally, the scheduling algorithms used often make it hard to guarantee that deadlines really will be met. However, **rate-monotonic scheduling**, see Sha and Goodenough (1990), is a scheduling algorithm in which it is possible to verify that processes will meet their deadlines under certain conditions. The processes must be *independent, periodic* processes that execute under a *pre-emptive* scheduler. Additionally, the processes must also have a *fixed* upper limit of *execution time*. The strategy is to set the process priorities in decreasing order of their process execution period. Hence the most frequently executed process will have the highest priority and then less frequently executed processes will have lower priorities.

A *distributed environment* will of course also involve several processes, since we will execute on several connected processors. We must then normally allocate the objects also to different processors. However, there are cases when one specific object must be represented on more than one processor, and thus in more than one process. The principal strategy should then be to encapsulate this distribution inside the block The implementation of this block could be done using one object module at every node representing the object and classes that will handle the distribution, see Figure 9.4.

In the figure 'objectAtHome' represents the object that exists at the mother site. The 'objectAway' represents the object at the distributed node. The 'Interface/Converter' is the functionality that handles the packaging and unpackaging of stimuli between the nodes and also the transfering These latter object modules are components that are used in all distributed blocks. Normally it is necessary to have different object modules at the host and at the distributed sites. An execution should then look as follows A message is sent to the 'objectAtHome'. This decides whether the processing must go to the distributed node If so a request is issued to the 'Interface/Converter' that packs the message and sends it to the appropriate site There it

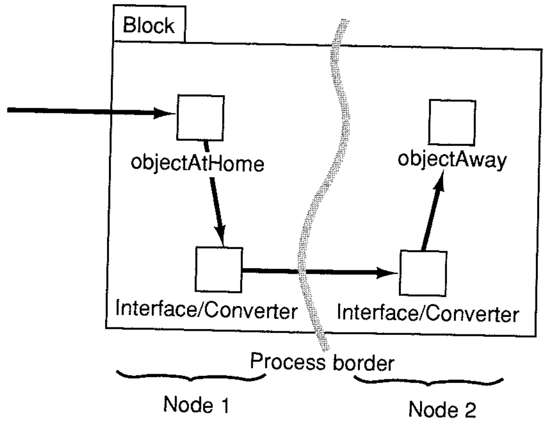


Figure 9.4 A distributed block with object modules at different nodes.

is unpacked and a message is issued to the appropriate instance. When the operation is complete a signal is sent back to the mother site where the return parameters are unpacked and sent to the 'objectAtHome' which, in it's turn, responds to the first message.

Here of course there are several detailed topics that must be solved (how to find the correct instances at the sites, how to block the first message, how to keep track of the transactions issued, what happens when the distributed site does not respond). We have found that the use of threads solves many of these (transaction related) problems since we then have implicit support for transaction identity, blocking messages, finding the correct instances and so on. However, a discussion of all the details will take us too far, and since this matter is not OO-specific we will not discuss it any more here. We will only mention the similarities of this approach to Remote Procedure Call, RPC, see Birrell and Nelson (1984). RPC has been implemented in some (UNIX) operating systems.

Objects in a use case that are *tightly coupled* in the sense of their having a large number of stimuli being sent between each other should preferably be placed in the same process. The reason for this is that signalling between processes will have a high penalty on operating system overhead in process communication. Note that this penalty must not be strong when using threads since we will not then have the overhead of context switching in the processor. However, this coupling between objects indicate that they are closely related and should therefore be placed close to each other anyway.

Use cases that have a *periodic behavior* could also be used as a base for process identification Such periodic behavior can be grouped together and placed in one process that is activated periodically. However, even if we have two behaviors that are not

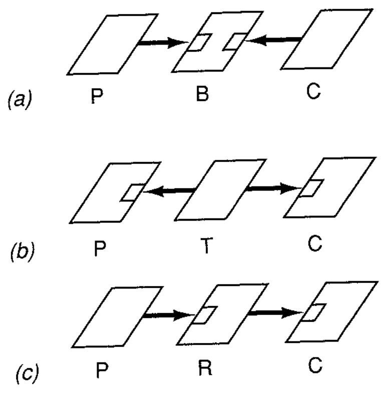


Figure 9.5 The principal functionality of processes for (*a*) buffers (B), (*b*) transporters (T) and (*c*) relays (R) P is the producer process and C is the consumer process

functionally related, but do have the same periodicity, it is not always certain that the best thing is to place them in the same process. The reason is that their periodicity may change over the system's life cycle, and not necessarily in the same way.

Use cases that involve *heavy computation* can also give us ideas of additional processes. Such behavior can normally be placed in low-priority processes and thus consume spare CPU cycles and not interfere with the more normal process execution.

To handle process **communication** and **synchronization** between processes may introduce additional processes. Examples of such processes are buffers, transporters and relays to loosen the coupling between processes, see Figure 9.5.

These different kinds of intermediate processes may also be combined to have a varying level of process coupling. Especially is this interesting in Ada where the *rendez-vous* mechanism between tasks is asymmetric For instance, we may have a sequence like PBTC, see Figure 9 6

This configuration will loosen the coupling to C Here C does not have to wait for *rendez-vous* with B since C will be called by T. When new processes have been introduced, we must decide for each pair of P and C which will be the caller and which will be the callee.

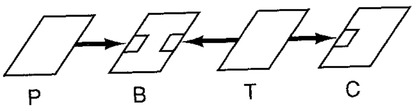


Figure 9.6 A process configuration that loosens the *rendez-vous* coupling between P and C.

For more on this topic, see Buhr (1984).

Owing to the problems of mapping processes onto existing hardware resource structures, an ideal process structure can very seldom be directly implemented The real work involves the merging of the ideal process structure to an acceptable number of processes that can be conveniently implemented. For example, we way need to sequentialize behavior from several processes to a smaller number of processes. Thus it is important to differentiate between what *might* be run in parallel and what *must* be run in parallel Using the guidelines discussed above may yield too many processes Application knowledge is then necessary to decide which processes should really be used in the system.

Many of the process identification approaches discussed above are not unique to OOSE. This topic has also been discussed by Gomaa (1984, 1989), Ward and Mellor (1985) and Nielsen and Shumate (1988).

The introduction of processes will yield many problems of its own like deadlock, starvation and racing between signals. However, these problems are not unique to OOSE, but are well-known problems of the concurrent programming community. We therefore only refer to any ordinary textbook in this area, for example Ben-Ari (1982), Hoare (1985), Peterson and Silberschatz (1985).

Other issues that may give problems when processes are introduced are the topics of *performance* and *memory management*. The introduction of processes will decrease the performance of the system since we have increased overhead by the operating system, for example process scheduling, context switching, interprocess communication and memory spacing We must therefore have a clear picture of the performance and memory requirements of the system and the cost of introducing new processes

The processes introduced may very well change the design model. Normally we will have blocks introduced that handle the interprocess communication and suchlike. Then the communication between the objects will change Since the communications have been found from the logical relationships between objects, this must not be the actual way that the stimuli are implemented Objects in different processes must access each other through objects supporting interprocess communication

The use cases are described by means of **interaction diagrams**. This technique is very powerful since it conveniently maps the real-time requirements of the use cases to specific objects and specific processes. We can thus easily trace the requirements to sequences over the blocks providing assistance in verifying time requirements. An example of this is shown in Figure 9.7 where a requirement of a

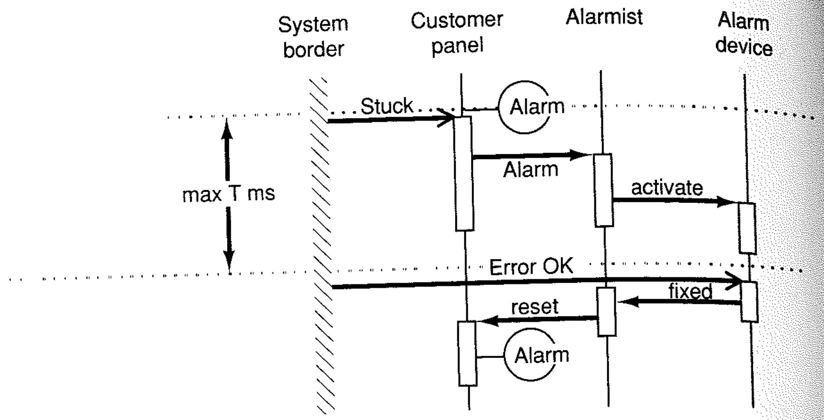


Figure 9.7 A real-time requirement from a use case can be allocated to objects using interaction diagrams

maximum execution time of *T* ms for the alarm in the recycling machine can be allocated to the particiapting objects.

The syntax and semantics of the interaction diagrams should be specialized to the actual implementation language. When we require special semantics, such as for synchronization and communication, notation for this may be used in the interaction diagrams.

The interaction diagrams also assist in identifying behavior that may be executed in parallel. In the interaction diagrams we may specify which parts may be executed at the same time, and which parts must be executed in a specific order. The interaction diagrams provide a visualization of how a certain series of tasks will be executed.

Interaction diagrams may also be drawn for processes only. Then the vertical lines represent processes and all communications are performed with signals

To handle concurrency and racing problems we may use state transition graphs. These can also be drawn for processes. In these we can then see which stimuli can be received at what times. Hence we can use this to detect erroneous stimuli received at the wrong times (states). This should be designed to get a more fault-tolerant system.

When the hardware resource structure provides for parallel/distributed processing, new criteria must be applied for process allocation and scheduling. In addition, due to the availability of redundant resources, the possibilities for developing fault-tolerant responsive system solutions can be exploited For a further discussion of these parallel processing issues for real-time systems see Lawson (1991).

9.6 Testing and verification

The testing of real-time systems is extremely difficult. There are several reasons for this situation To provide verification of hard deadlines we must test the system in the target environment and test it so that the probability of missing the deadline is sufficiently low Often the verification of deadline requirements is impossible by normal testing techniques, since the system may meet the deadline 99 times, but the 100th time it fails, for example when two processes use a common resource. Problems may not always arise since the processes using the common resource might seldom execute simultaneously and thus the problem will not occur. This means that failures can be very hard to reproduce since failures can be time-dependent Another consequence is that when the system is executed with debugging tools or traces on, the system can behave correctly, while the system runs into a failure when these are switched off, or vice versa.

We will discuss testing in a later chapter, but we wish to say here that the ideas of automated testing, regression testing and stress testing are extremely important in real-time systems. Furthermore, since real-time requirements are often attached to use cases, use case testing becomes extremely important. All of these tests should be performed many times since the system may behave differently at different times (which motivates automation of the test).

Verification is not simply a question of the testing of the system, but must be an integrated part of the design process. An interesting approach to the verification of timing properties of real-time systems has been presented by Shaw (1989) where minima and maxima for process execution as well as time for environment overhead are used to determine the predictability and loading of real-time systems. Techniques of this variety should complement the usage of OOSE as a means of verifying timing properties during the development.

9.7 Summary

A typical real-time system detects and/or controls events outside the system under timing constraints If these timing constraints must be met to avoid catastrophes, the system is a hard real-time system. Otherwise it is a non-hard real-time system. Real-time systems add an extra dimension to the system, namely time, which makes them even harder to develop than other systems. Although it is not a rule, the fundamental issues of these systems include processes, synchronization and communication OOSE does not put any *a priori* strong semantics on these concepts since various implementation environments provide various semantics Instead the semantics of the actual implementation environment should be used. Therefore a specialization of OOSE should be done for each specific implementation environment.

During analysis the real-time requirements must be collected and analyzed. This is done mainly by associating these requirements with the use cases in the requirements model. The requirements are then traceable through all activities to implementation for verification at various levels. The development of the analysis model should normally not be affected by these real-time requirements since this model should be independent of the implementation environment.

However, during construction we must design and implement the system under these constraints. This often involves the design of processes, communication and synchronization of processes using the appropriate semantics. OOSE provides a basis for the identification of processes. Since the ideas of object-orientation have many similarities with concurrent processes, it is often straightforward to make the identification The use cases are here a strong tool since they express the execution in the system. Distribution of a specific object over several computer nodes can be encapsulated in the object itself. The interaction diagrams can also be used to handle real-time requirements and to allocate the timing requirements to different objects. Traceability of timing requirements is achieved to aid the timing verification State transition diagrams can also be used for timing constraints.

Testing real-time systems is very hard, mainly because of the extra dimension of time. Since all executions are time-dependent, failures may be almost impossible to reproduce. Therefore it is even more important to have quality assurance and verification procedures during the entire development. The traceability in OOSE is then a fundamental property.

10 Database specialization

10.1 Introduction

The need to use a database often arises from the limited capacity of primary memory. Therefore databases are often stored on **secondary storage**, providing efficient ways to access the objects. Another need that arises is the ability to store objects longer than a program execution. Thus we want the object to survive the execution that created it. This ability is called **persistence**. Persistence often means that objects are copied from a fast and volatile primary memory to a slow and persistent secondary memory, see Figure 10.1.

Since the user of the database does not want to bother about how the storage is done, we often need a database management system, or a DBMS, to handle this. The application programmer only wants a logical view of the database and does not want to take part in the decisions of how the physical storage is done This is an important property of a DBMS. One alternative to this approach is to implement this functionality yourself using the file system and bury it in application code, but this would be a significant amount of work. A DBMS enables one to obtain such functionality in another, usually easier way.

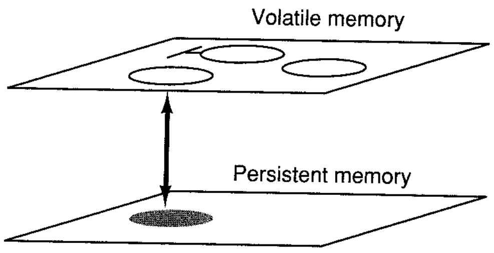


Figure 10.1 An object is copied from/to the volatile memory to/from the persistent memory.

Other features that a DBMS should provide are, see Atkinson *et al.* (1989):

- Concurrency: allows multiple users to work against a common database simultaneously
- Recovery: if a (hardware or software) failure occurs, the DBMS shall be able to bring the database back to a consistent state of the data.
- Query facility: the DBMS should support an easy way to access the data in the database.

Database specialization means that a DBMS product is integrated in the development of a system The specialization of OOSE will vary according to the type of DBMS being used

Often it is obvious that a DBMS should be used in a development and usually it is already in the requirement specification of the system. If such a requirement does not exist, then the need should be discovered by studying the requirement model. Typical properties that indicate a need for a DBMS are:

- Information that needs to be persistent,
- More than one application sharing (part of) the data,
- Information structures with a large number of instances,
- Complex searching in the information structure,
- Advanced generation of reports from stored information,
- Handling of user transactions,
- A log for system restart

Therefore the requirement and analysis models can be studied to decide which objects are to be persistent Since the entity object typically hold information that survives the use cases, the entity object are major candidates for persistent storage. In some applications there is also a need to store other kinds of objects, such as objects to handle interface set-ups Even if the information does not have to survive different executions, there may be a need to use secondary storage devices. This is typically the case if the number of instances (and/or the size of each instance) is very large compared with the primary memory space Ways to solve this are to use the file system or to use a DBMS.

However, the main integration work is done in the Construction process Here we must decide how the DBMS should be incorporated in the design model and the implementation model. Additionally, we must obtain information on how to optimize the storage of the object's state, for example how a table structure in a relational database should be indexed in order to perform faster searches. Our discussion will focus on the design work included in integrating a DBMS in the system

DBMSs have evolved through a number of generations including hierarchic, network, relational and now also object-oriented databases. The dominant type in industrial applications today is the relational model and that is why our discussion here will emphasize relational databases. For a thorough discussion on relational database systems see Date (1986) Object-oriented databases will be discussed briefly.

10.2 Relational DBMS

10.2.1 Problem issues

In a relational database, information is stored in tables. The types to be used in the tables are mostly of primitive types such as characters, integers and the like This yields some problems for our ambition of storing objects. Firstly, only data can be stored and not behavior and secondly, only primitive data types can be stored and not the complex structures of our objects Although many vendors add these capabilities to their RDBMS products, no standard or even consensus exists of how this should be done, see Bloom and Zdonik (1987). We therefore assume that none of these capabilities exist in the products we are to use In our discussion we will assume that our programming language is an object-oriented language, such as C++

When a programming language is connected to a DBMS, a number of problems arise. The first problem is that in our system all information is stored in the objects. We therefore need to transform our object information structure into a table-oriented structure. This problem is sometimes referred to as the **impedance problem** The problem is often that the program has too rich a set of types, including those created by the user. All these types must be converted to the more primitive data types of the RDBMS. The name of the problem has been obtained from similar problems of impedance when using transformers in electrical engineering.

The impedance problem yields yet another problem, it creates a strong coupling between the application and the DBMS. To make the design minimally affected by the DBMS, as few parts of our system as possible should know about the DBMS's interface.

A third problem is how to express inheritance in the database. Thus, if one object is inherited by another, how do we express this?

Since all objects that are persistent have this ability (role) in common, the behavior of persistence can be inherited also. Hence we have similarities both in the data structures and in the ability to be persistent. The latter ability will be expressed in the programming language (we will look at an example later) and the former may be expressed in the database.

Other problems that arise include how to store operations, the transaction view (application view versus DBMS view), locking of the database during longer transactions, and distribution. Integrity problems between primary memory and the database must be handled, for example references between objects might change in the system and hence must be updated in the database.

We will look at solutions to some of the problems mentioned above. The problems we will discuss include the impedance problem, how to isolate the DBMS from the system and the problems of how to handle inheritance and model the ability of being persistent abstractly. The other problems must of course also be solved, but we will not discuss them further here.

10.2.2 Objects into tables

The first thing to do is to decide upon which classes and which variables of the class must be stored in the database. Each one of these classes will then be represented by a table (at least one) in the database. A class is mapped onto tables in the following manner.

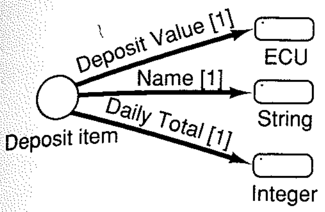
(1) Assign one table for the class.

(2) Each (primitive) attribute will become one column in the table. If the attribute is complex (i.e. must be composed of DBMS types) we either add an additional table for the attribute, or split the attribute over several columns in the table of the class.

(3) The primary key column will be the unique instance identifier, namely the identifier by which the instance is uniquely recognized. The identifier should preferably not be visible to the user, since changes of the keys for administrative reasons should not affect the user; machine generated keys should be used.

(4) Each instance of the class will now be represented by a row in this table.

(5) Each aquaintance association with a cardinality greater than 1 (e.g. [0..N]) will become a new table. This new table will connect the tables representing the objects that are to be



(a)

Deposit item

Name	Deposit value	Daily total
Can33	0.25	142
Can50	0.35	35
Bottle25	0.30	173
Bottle75	0.50	78

(b)

Figure 10.2 In (*a*) the object and its attribute will yield a table as shown in (*b*)

associated. The primary keys of these tables can be used in this 'aquaintance' table. In some cases, however, we may have the relation represented only as a column (attribute) in the associated object's table.

Let us turn to an example In the recycling machine example we have the object yielding a table as illustrated in Figure 10.2. The attribute types are all very simple, and thus will not yield any new tables

This strategy will give us a first logical database design where we have defined tables to store the objects. Now what about normalization of the database? (Normalization concerns the issue of eliminating redundancy in the database and avoiding certain update anomalies In most cases it is enough to reach the third normal form for the database design.) Actually, a database design based on an object model will normally end up in the third normal form from the start The reason for this is quite obvious. The third normal form states, intuitively, that: if, and only if, for all times, each row consists of a unique object identifier together with a number of mutually independent attribute values, then the table is in third normal form, see Date (1986). Hence we will have the third normal form if we have an object model where each object has a unique identifier, and the attributes are mutually independent, that is, none of the attributes is functionally dependent on any of the others.

More intuitively, since object-oriented models are often models of the reality, we tend to identify unique objects and also assign the attributes to the objects where they naturally belong Hence, since the reality is normalized as such, a good object model will also be normalized

Techniques for relational database design based on object-orientation have evolved, see Blaha *et al.* (1988). The experiences have been good and superior to other tachniques and these authors state that: 'Building a data model from a small number of coherent entities object is superior to the traditional approach of collecting all the attributes, ferreting out the functional dependencies, and synthesizing tables '

However, when the database is normalized, the problem of low performance occasionally occurs. Low performance can in many cases be solved by special indexing tables If this is not possible, the real problem is to '*denormalize*' the database to increase performance. The denormalization must be done with consideration of how the database is used. This may take the database design as far as below the first normal form.

In this case we have redundancy problems in the database. However, here an object-oriented view will help us again. An object encapsulates the object's representation in the database. Then the only place where the knowledge of the redundancy exists is within the object, in one place and not spread among several application programmers.

10.2.3 Inheritance

The objects that should be persistent will thus be mapped onto tables in the database. If classes inherit each other, there are principally two different ways to solve this.

(1) The inherited attributes are copied to all the tables that represent the descendant classes. No table will represent the abstract class

(2) The abstract class is in one table of its own, to which the tables of the descendant classes refer

We will illustrate this with some examples from the recycling machine. We want every Deposit Item to be persistent, namely to survive when we turn the machine off every night. In Figure 10.3 we show some of the entity objects with their attributes.

The first alternative is to have only tables for the Can and Bottle objects and copy the attributes from the parent class as shown in Table 10 1

The second alternative is to store the common attributes in a table representing the abstract class Deposit Item as shown in Table 10.2. Please note that in this case the primary key must be unique in the common table Generally, primary keys should be generated by the system to guarantee uniqueness.

So which alternative is to be chosen? There is no best general solution for all cases, but let us illuminate some of the properties of

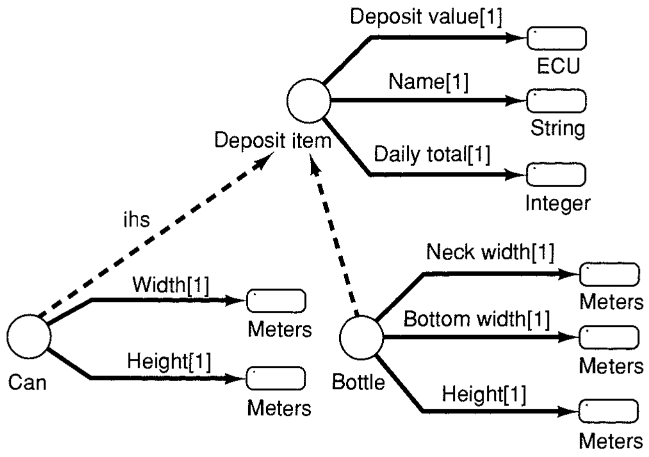


Figure 10.3 Some entity objects with their attributes.

both alternatives The first alternative is normally faster since no joining, or searching in several tables, is necessary to get information about one object. However, the size of the database will increase since the inherited columns must be duplicated. Additionally, if changes occur in the inherited attributes, these changes will affect the tables of all descendant classes In this solution we have no knowledge of what type is represented in the table representing the abstract class. Therefore a specific type table may be necessary. The second alternative includes less redundancy, but may lead to problems

Table 10.1 Only the descendant classes are represented by tables

Can Name	Deposit value	Daily total	Height	Width
Can33	0.25	127	17	7 50
Can50	0 35	283	25	7.50

Bottle Name	Deposit value	Daily total	Height	Bottom width	Neck width
Bottle25	0 30	173	23	6	2
Bottle75	0 50	78	32	9	3

Table 10.2 All classes in this alternative are represented by a table. The Can and Bottle objects refer to a common table where the common attributes are stored

Deposit Item		
Name	Deposit value	Daily total
Can33	0.25	142
Can50	0 35	35
Bottle25	0 30	173
Bottle75	0 50	78

Can Name	Height	Width
Can33	17	7.50
Can50	25	7.50

Bottle Name	Height	Bottom width	Neck width
Bottle25	23	6	2
Bottle75	32	9	3

if the identifiers are common in the table of the abstract class. For instance, if we have an abstract class person and descendants teacher and student, then no person can be both student and teacher. Additionally, if we want to change the primary keys, these changes must be done in several places. Generally, the second alternative (to have tables for the abstract class) is to be recommended. It is then often practical to have the same identifier in all tables. If this is not possible, use system generated unique keys for the shared tables. What really is important is of course how the tables are used in the different use cases Which specific search criteria need to be satisfied? What other requirements need to be considered?

10.2.4 Modeling the persistent object

We will now look at an example of how to handle persistent behavior in an abstract block. Since every object that should be stored in the database must have a persistence functionality in common, we create an abstract block, ObjectToStore, that has this functionality, see Figure 10.4. The classes in this block form a framework to build other blocks from. The classes are not independent, but have a well-defined

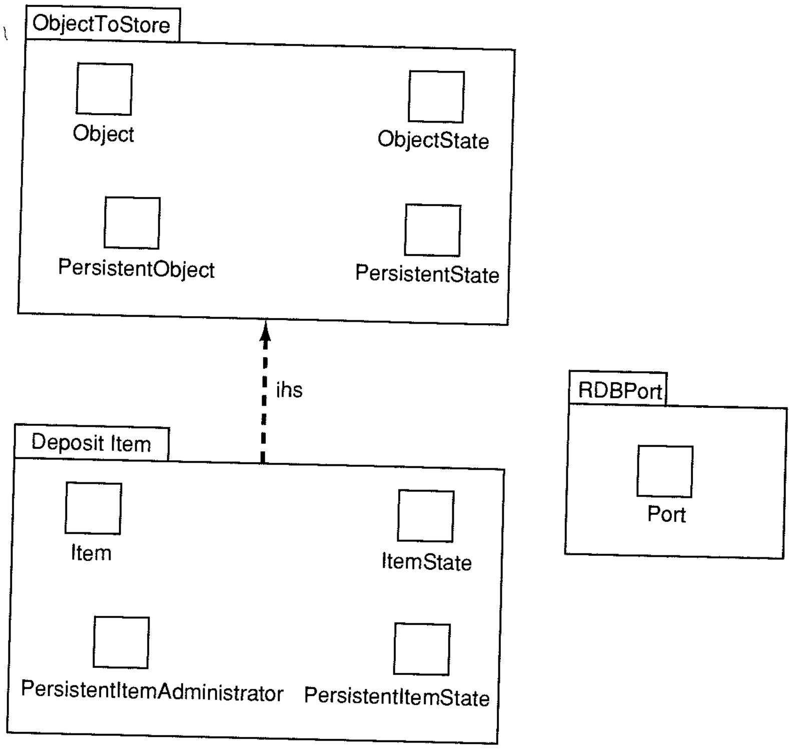


Figure 10.4 Common parts of all blocks that will store its information in the database are put in an abstract block which is inherited by other blocks.

protocol that handles the persistence mechanism of the blocks This block is inherited by all blocks that must be able to store information in the database and the framework is specialized to every particular block, see Deposit Item in the figure. The block Deposit Item thus inherit the block ObjectToStore and we now need only to program what is specific for every Deposit Item namely the functionality of Item, ItemState, PersistentItemAdministrator, and PersistentItemState which differ from the classes we inherit We also add a block RDBPort which includes a component class to handle the interface against the database.

The classes of ObjectToStore are as follows:

- Object represents the object abstractly and Item is the actual object in the system.
- ObjectState represents the attributes of Object to be stored in the database in Object's datastructure, hence ItemState contains the information of Item that is to be stored in the database.

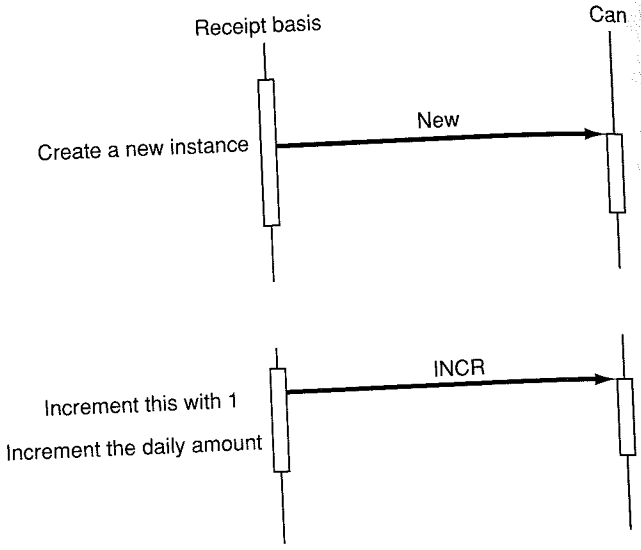


Figure 10.5 The external view of creating and updating a block *Can*.

- PersistentState represents the attributes in ObjectState, but in the type systems used in the DBMS.
- PersistentObject converts ObjectState into PersistentState. Hence, PersistentItemAdministrator converts ItemState to the attributes in the tables and stores it in PersistentItemState.

  PersistentItemAdministrator uses RDBPort to store this information in the database.

Initialization and the creation of tables must of course be done at the beginning We do not discuss this here, but these issues should prefeably be handled by the RDB-port since they are highly DBMS-dependent.

The flow will be as follows When Item is updated, ItemState is also updated. To store this information in the database, Item tells PersistentItemAdministrator to store the information. PersistentItem-Administrator gets the information in ItemState and translates it into PersistentItemState. When this is done it stores PersistentItemState in the database via RDBPort

Figure 10.5 shows how a creation and an update will look externally from a Can block Note that none of the internal behavior in Can to store information in the database is seen from the outside. We have thus encapsulated the function/data paradigm used by the DBMS and can now show an object-oriented interface to the application

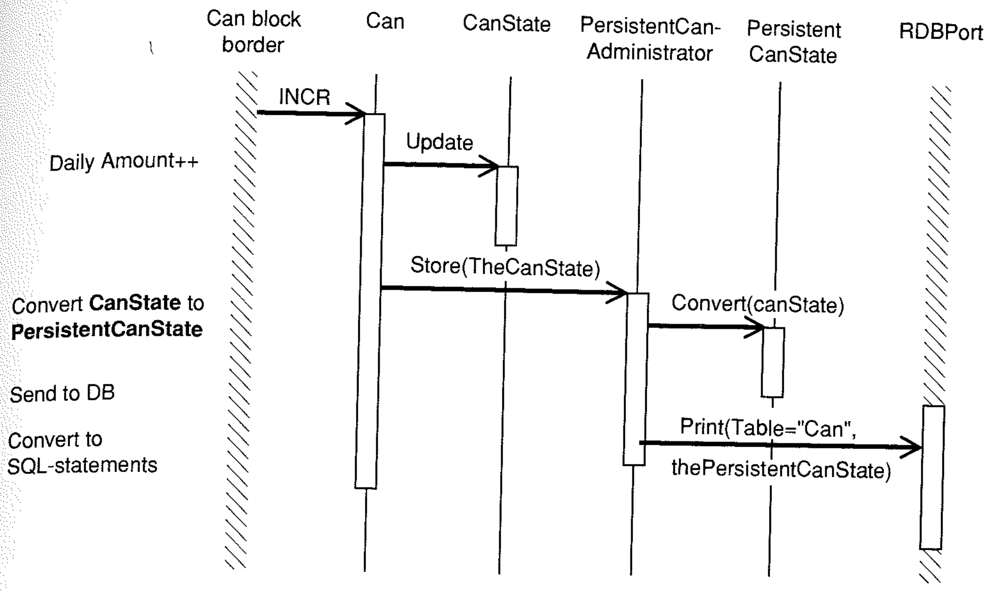

Can block border
Can
CanState
PersistentCan-Administrator
Persistent CanState
RDBPort
INCR
Update
Daily Amount++
Store(TheCanState)
Convert CanState to PersistentCanState
Convert(canState)
Send to DB
Print(Table="Can", thePersistentCanState)
Convert to SQL-statements

Figure 10.6 The behavior inside the *Can* block when updataing the daily amount.

Let us take a look inside the Can block, see Figure 10.6. When the INCR stimulus comes, the update is made. The update is also made in CanState. To store the update in the database a stimulus Store is sent to PersistentCanAdministrator which converts the CanState into PersistentCanState and tells RDBPort to store it in the database. Nothing of this is seen outside the Can block

The impedance problem is handled by instances of ItemState, PersistentItemState and PersistentItemAdministrator. The independence of the DBMS (provided that it is a relational DBMS) is handled by the block RDBPort. The similarities of all persistent blocks are thus described in ObjectToStore.

If we want to use another type of DBMS, which parts must we then change? Obviously RDBPort must be changed This block thus encapsulates the DBMS used. Since PersistentItem and Persistent ItemState only know that we are using a relational database, these do not need to be changed on changing the DBMS However, different DBMSs may have different type systems and possibly also different table semantics Generally, avoid being tied to any specific vendor, use capabilities which are widely recognized only PersistentItem knows the names of the tables which store the object's state and PersistentItemState knows the table structure, so if these topics also change, these need to be changed too

To summarize, to each block we assign a block description that includes descriptions of:

- Classes for the block's data and behavior,
- Classes to handle the persistancy,
- Tables in the database to store the block's information.

A similar technique to encapsulate a RDBMS behind an object-oriented interface is described by Premerlani *et al.* (1990).

10.3 Object DBMS

Relational systems were first introduced in the early 1970s. During the (late) 1980s they have totally dominated the DBMS industry. They serve as a simple and logical view of the data stored in the database. The user views the data as stored in tables which he can manipulate. One important idea is that data be stored in as few places as possible. Different normalization techniques, as we have briefly mentioned, support the process of making the tables as independent of each other as possible.

Simplicity and data independence are the major features of a relational DBMS, but can also be a drawback in some applications. The relational model cannot capture the semantics of complex objects. To model a complex object we often have to split the information into several tables, and this makes each access to such objects slow since the DBMS must join a lot of tables to gather the object's information.

The idea of an object DBMS (ODBMS) is to store the objects as such, and thus bridge the semantic gap all the way to the database. In this way we do not have to perform any slow joins to get access to a specific object. By storing the objects as such instead, it is possible to express the semantics of the objects in a much better way than is possible in relational systems. This implies that ODBMS is very useful in applications where complex objects must be persistent. Typical examples are different kinds of design support such as CAD, CASE, CAE and the like.

Most commercial ODBMSs do not use a specific data manipulation language to store and retrieve information, but use the programming language directly, for example C++ or Smalltalk. This means that a specific interface to the DBMS is not necessary, as is the case for relational systems (that mostly use the language SQL). Hence the impedance problem is eliminated completely. This also means that the design of the database is integrated during the analysis and design of the application.

What criteria are there then for an ODBMS? One of the more important papers written on this topic is by Atkinson *et al.* (1989), although there is far from being a consensus on the contents among the ODBMS community. They identify, besides the criteria discussed above for an ordinary DBMS, the following:

- **Complex objects.** It should support the notion of complex objects.
- **Object identity.** Each object shall have an identity independent of its internal values
- **Encapsulation.** It shall support encapsulation of data and behavior in the objects.
- **Types or classes.** It should support a structuring mechanism, either in the fórm of types or classes.
- **Hierarchies.** It should support the notion of inheritance
- **Late binding.** It should support overriding and late binding.
- **Completeness** The manipulating language should be able to express every computable function
- **Extensibility.** It should be possible to add new types.

The commercially available object DBMSs have evolved from two origins. The established DBMS vendors have built object-oriented features on top of their relational systems. These are often called extended relational DBMSs. Several new vendors instead start from the programming language and often support specific classes or other features to intergrate the ODBMS in the programming environment.

We do not have sufficient experience to discuss fully the use of ODBMS in development using OOSE, but the projects where we have used such systems show that the construction process is much simplified in comparison with using relational systems Mostly this does not affect the remaining construction, but is very straightforward to use during implementation. You just mark the classes of which the instances should be persistent and also define the transactions in some way, and that is all. However, since relational DBMSs have been used for a long time, these products are more mature and also have more extensive support for many complex issues included in the use of databases

Hence the major benefits of using ODBMS include:

- The objects as such can be stored in the database (often the operations are not stored, but are only present in the class library in the primary memory),
- No conversion is needed for the DBMS type system; the user

defined classes are used as types in the DBMS,

- The language of the DBMS can be integrated with an object-oriented programming language The language may even be exactly the same as that used in the application, which does not force the programmer to have two representations of his objects

10.4 Discussion

The ease of incorporation of an ODBMS into the design often makes this an attractive solution. However, different application areas have different requirements. The tendency is therefore that different vendors optimize their product for different requirements, that is application areas. The differences may involve the complexity of the objects, the length and frequency of the transactions (the checking out of information during longer periods (days, weeks), etc). What is obvious, however, is that all ODBMSs focus on more complex data structures than a relational DBMS is able to handle. This also means that for application areas where the information structures are very simple and are best expressed in tables, a relational DBMS may be a better choice than an object DBMS anyway. Typical of such application areas are banking systems where we have customers and their accounts, or a flight reservation system where we have flights and reservations on the flights.

Hence there is no general best choice. A choice must be made from time to time with great respect to the needs of the application to be developed. The best thing is to benchmark different databases with the specific application as such or to simulate the requirements of the applicaions Then it is important to use real volumes of data and real use of the data. If this works, double the amount of data since it is almost impossible to guess the amount of data to be stored in the future. In practice, all queries with a performance requirement must be verified in advance. More about ODBMS, maturity and benchmarking can be found in *JOOP* (1991). More comparisons between RDBMS and ODBMS can be found in Loomis (1990) or Stone and Hentchel (1990)

10.5 Summary

Databases are used to store objects persistently, that is, for longer than a specific execution. To use a DBMS gives support for this persistence and also other issues that raise the abstraction level for the database user The main DBMS type in use today is the relational DBMS which stores data in tables Object DBMSs are now evolving with great speed to support the complete persistence of entire objects.

To use a relational DBMS in combination with an object-oriented system raises many problems. However, many of these problems can be encapsulated; one strategy using a reusable framework is shown schematically in this chapter To do the logical database design from an object-oriented model is straightforward since it normally leads directly to normalized tables. Also, inheritance can be simulated using the tables

Object DBMSs have been developed to store objects as such in the database No standard, or even general consensus, exists on what defines an ODBMS. Different application areas have different requirements, and many vendors optimize their products for a specific application area. To use an ODBMS in the development in OOSE is often easy. Generally, no extensive overheads are needed to incorporate the ODBMS in the application developed.

11 Components

11.1 Introduction

11.1.1 Reusable software engineering

To be able to reuse code has long been a notion in the world of software engineering. This has been one of the major issues to increase dramatically the productivity of the programming community. However, although most people agree upon the importance of reuse, it has been practiced only very scarcely in software development organizations. We will come back to the reasons for this.

When we speak of reuse in software engineering we mean everything that can be reused at a later time. This includes all the information and knowledge that has been developed, including experience from earlier projects, both by humans and by organizations, system architectures that have proved to be good, development methods that proved useful, but also written program code or completed algorithms. This highlights the fact that code reuse is only one small part of reusable software engineering, see Freeman (1987).

We have seen that the models developed in OOSE provide a good base for reuse of information. As an example we see that the use case model is reused for developing all subsequent models. Additionally, the analysis model is developed to get a robust structure with reusable objects for future changes. There we try to identify a structure that is as much reusable in future changes as possible. However, in this chapter we will concentrate on the code reuse level, namely the reuse of **software components**.

11.1.2 Components as a reinforcement mechanism

When working with software components, we should view these as a part of the programming language. Just as we have primitive

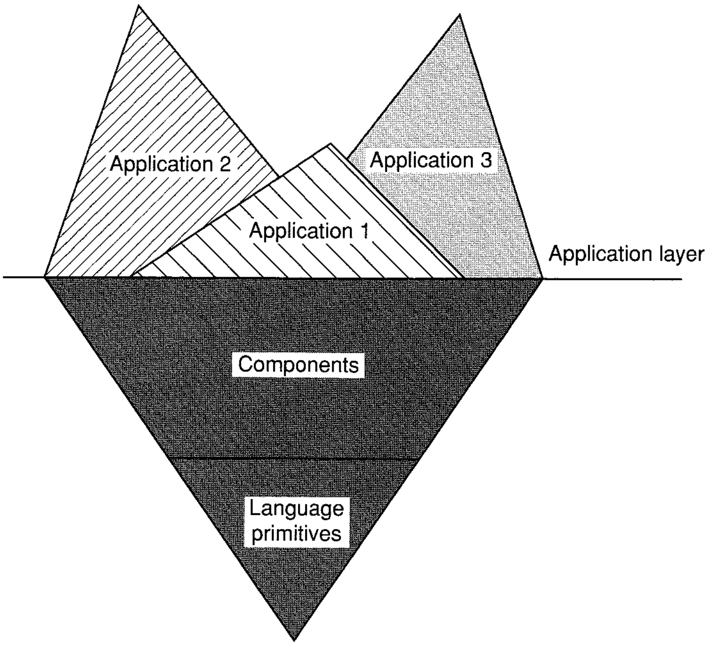


Figure 11.1 Components are used to build bottom-up whereas applications are built top-down.

constructs in the language, we will have primitives at a higher level of abstraction. In this way components constitute a reinforcement mechanism for the implementation language; instead of working with primitive constructs, we will work with constructs of higher level abstractions, like lists and windows. In this manner we can thus leverage the development from primitive constructs to more complex constructs. Having the components built on the programming language we will get a higher level of the application layer on which we will develop our applications, see Figure 11.1.

To build with components and to build with objects are two entirely different activities With objects we start from the use cases and design the objects as our system is developed. To build with components, on the other hand, is a bottom-up activity. Furthermore, blocks will be managing units handled by the application developers and several applications may be developed in one organization. They are documented for this purpose. Components, on the other hand, will be managing units of a component system and are shared among several applications They are normally documented in a completely different way

Having to work with components is not unique for software engineering Exactly the same mode of work is used in all the other engineering disciplines. Electrical engineers, for example, work with integrated circuits for their designs, mechanical engineers have standardized bolts, civil engineers have different types of girders to work with, chemical engineers have different solvents and process devices. Not least, VLSI designers base their work on complete module libraries to build their circuits They only indicate the width and number of registers to have a complete register bank for their design; they do not put in each transistor by hand! Today software development is something of an art in which each programmer is more of an artist than an engineer, as discussed in Chapter 1. You can consequently say that the use of components is also making the software industry into a fully fledged engineering discipline that should work on an industrial scale

Many feel that components are one of the most important solutions to the software crisis The crisis is that as the complexity of our systems grows, they become more expensive and more difficult to develop The problem is that the productivity of the software engineers does not grow at the same pace. If we do not become more productive we will fail in building the systems that we are expected to build

By means of components we thus hope to master the design of large systems. There are above all two important criteria for which components are of great help: to reduce the development time (cost) and to raise the quality. We reduce the development time as we have more powerful components as a basis. This means that we reduce the complexity and will thus have to write less code ourselves. As we use complete components this means that they are also well tested, both because we spend more time on their development and because they have been used in other applications. If we fulfill these two criteria it will lead us to less expensive and better software systems.

11.1.3 Why do we not have components today?

Now, if components are so good why are they not widely used to a greater extent today? We can identify a number of answers.

- Projects are often conducted with a tight budget and a tight time schedule. To design good components takes time. Since it is not profitable to design components for one single project there is no incentive to do so.
- It is safest if I write the code myself; the 'not invented here'

attitude. To trust the code of someone else makes one feel uncertain If there is going to be a fault I want to have control over it and be able to correct it myself

- There is no recognized component standard that is widely used Such standards should specify the functionality and the use of components. To establish a standard for components is a gigantic task.
- Components already exist, but we cannot find them Complete components can be designed in a project but they are not saved.
- You feel productive when writing source code Since the most common measure of software is lines of code, you feel you are progressing when you produce code. This is an abuse of your task, you should solve the hard problems, not the easy ones. Instead you must accept what is available and go forward; you cannot spend time on putting the finishing touches to a component to suit your purposes. This will lead to low productivity and a waste of resources
- If a component market should appear there will be problems as components are copied and distributed for free, which means that no one wants to pay their high costs. Compare this with the same problem in the software industry. Few companies consequently dare to design components on a large scale.

An important reason why good components have not been designed is that engineers have tried to find them from functions and not from the objects When it was discovered that something was to be stored, a function for this purpose was immediately designed; put (e). It quickly became apparent, however, that you had to store it in different ways so a parameter was added: put(e,queue) and the variants proliferated rapidly. Then you wanted different parameters for the data structure: put(e,queue, preemptive, priority, length) and you continued to add parameters for efficiency, . . . You never found the correct abstraction, or to quote Gerry Weinberg (1971): 'Unfortunately, program libraries are unique; everyone wants to put something in, but no one wants to take anything out.'. Tracz (1988) also discusses this topic

However, reuse in the software field *have* been practiced in different forms If we want a sorting algorithm for a specific situation we normally find a good algorithm in the literature and implement it instead of inventing a simple algorithm ourselves This shows that reuse of program code and algorithms is actually nothing new but that it has existed for as long as we have been programming. A standard work that has been a source for many such 'components' is Knuth (1973, 1981)

11.1.4 A new approach

The area in which components has succeeded best is probably mathematics routines in FORTRAN Here we have an abundance of routines to be used in a lot of applications that need some form of more advanced calculation, for example statistical or numerical. The reason why it has succeeded there is probably that it was easy to define the 'objects' that the routines were to operate on; representation of matrices or arrays is easy to standardize. That it was easy to design components where the data structure could be defined easily indicates that this is where the problem actually lies This would indicate that information hiding in general and object-oriented programming in particular is a good road to take when it comes to components. This proves to be true; as we encapsulate the data structure completely and only offer operations on it in an object, the component becomes insensitive to the surrounding world that only knows certain operations In the same manner we have today developed the theory about abstract data types which also provide an excellent basis for implementing components.

This also suggests to another reason why it has been so difficult to design useful components; the development method has not stimulated it. A traditional development method often has the form of functional decomposition or at a least top-down process. This does not at all benefit the use of components; you need luck to end on a component at the very bottom. With object-oriented methods you build both bottom-up and top-down. To build with components is consequently much more natural with an object-oriented method than with a function/data method.

Components are today used in certain areas. An example is the Smalltalk world in which much of the programming is to find the correct class for your application Here we already have the attitude that 'a good class is a reusable class'. Here it is a matter of pride to write such a good class that others will want to use it. It should consequently be positive that you can spend your efforts on solving the difficult problems instead of doing mechanical routine work. The challenge is raised by one level of difficulty

Software components are also used in certain industries; several companies (among them Toshiba, IBM and NobelTech) have today started special software component departments. Their purpose is to provide all other departments with components, and they have no direct profitability requirements. This means that work on designing a good and useful component will not detract from some critical project budget, but rather will be distributed over several different projects. Later in this chapter we will outline how such a department can work.

11.2 What is a component?

As stated before, here we will only deal with reuse consisting of already implemented software components. Once more we wish to emphasize that this is a considerable restriction of reusable software engineering. However, we need to define what we mean by a component and how to work with them. A component is a standard building unit in an organization that is used to develop applications.

To make a component reusable in various applications it is necessary that it be independent of the application for which it was designed. This is not always necessary to achieve; sometimes components that are application-dependent are of interest for reuse We can consequently grade our components on a scale that ranges from complete application-independency (such as a binary tree structure) up to complete application-dependency (such as scanning a bar code reader). The more dependent the component is on a certain application, the more frequently you must adapt it in order to use it Examples of components that may have to be adapted are browsers, windows and file handlers, but they are still very useful

The requirements imposed on a component are much greater than on ordinary software To use a component means that we can save time since we do not need to know how it works on the inside We only need to use it from the outside. A complex component that is easy to use thus raises the *abstraction* level for the developer. It then forms a conceptual simplification of what is implemented

Since the component should be used in various contexts we require it to be a *general* abstraction so that it can be used widely. This means that the component must be designed to be reused. Hence we need to include operations so that all reasonable use of the component is satisfied. However we should not include too many operations, since that makes the component hard to understand and use. Thus the component should capture all operations that make it meaningful to use for any developer and no other operations. Note though, that this does *not* mean that we should give the component a *general implementation*. If this makes it ineffective, it should definitely be avoided. More about this later.

A critical issue is whether the developer trust the component. If the components come with a bad reputation, no one will use the components. Therefore the components must be of an extraordinary *quality*. They thus need to be well tested, efficient and well documented. Components are therefore normally more expensive to develop than ordinary software.

The components should also invite reuse. The components must thus have a well-designed interface, easy retrieval and be accompanied by good, correct and easy to use documentation. The components should thus be *packeted for reuse*

Generally speaking there are two schools of thought on what primitive components should look like. One is that components should be *flexible*, which means that they should be easy to change in order to adapt them to your own needs, see Parnas *et al.* (1983). The arguments for this are reasons of efficiency; general components are often inefficient.

The other viewpoint is quite the opposite, namely that components should already be *specialized* and you should *not* make changes in them (see Booch (1987a) where there are 26 different implementations of 'Set'). Here you instead solve the efficiency problem by having several different types of the same component. We will here mainly follow the latter school for small scale components, but for larger scale the former is of course of interest

To design useful components is not easy. The most well-known components include various data structures and window management systems. Today there are several libraries on the market for these kinds of components. However, more specialized components have been harder to define. Components are often developed gradually from several different examples where you can find a general behavior. From these you can extract something that is useful in several different places, an embryo of a component, that you frequently have to modify in order to make it fully useful. We have then a **white-box component**, that is, you have to look inside it to reuse it, see Johnson and Foote (1988). These often have a tendency to be oriented towards a specific application domain. To develop these white-box components it is therefore necessary to have a good knowledge of the application domain.

When white-box components have been used and further refined it is often posssible to reach an even further level of abstraction. Then it is possible to extract the essence of a component and design it as a **black-box component**. In this case you will not change the inside of it in order to use it in a new context.

Black-box components are often easier to use than white-box components since they are used as plug-in components and no modification is necessary. Additionally, they should be well tested and are often of a higher quality than ordinary software since requirements are stronger on these Typical black-box components are the common abstract data types such as stacks, strings and search trees.

However, white-box components, although they are harder to work with, are of great interest anyway. The reason for this is that there is often reuse of much larger parts and thus they will increase the productivity significantly more than reuse of black-box components. A special form of white-box component is a design **framework** where you have an entire skeleton to build your application in. This is a reusable design. Typical white-box components include a sorting algorithm where you might want to change the 'greater than' operator (this could also be done with a black-box component) and Model/View/Controller in Smalltalk which is a class framework (one of the first) for interface construction.

Many problems still need to be solved for white-box components, although research and experiences are under way. One problem with white-box components is for instance how much should you actually be able to change a white-box component before it no longer counts as the original one, and how should a white-box component be tested to become stable?

11.3 Use of components

11.3.1 Finding places to use components

Components should be viewed as an important part of the implementation language used They should give leverage to increase productivity. This means that, just as the developer must have a good knowledge of and skill in the programming language, she or he must likewise have a good knowledge of the component library in order to use it effectively. This is taken to the extreme in for instance Smalltalk, where the language vocabulary as such is very small, but the actual programming is more a case of managing and using the class library.

Generally, everything could be implemented using components. However, often the right component does not exist, and so is not available for use. We will discuss later how a component library could be built incrementally for more application-oriented components. Normally the 'classical' components exist as a base. These are either obtained on the market and/or developed in house and implement classical abstract data types, and often also offer windowing facilities

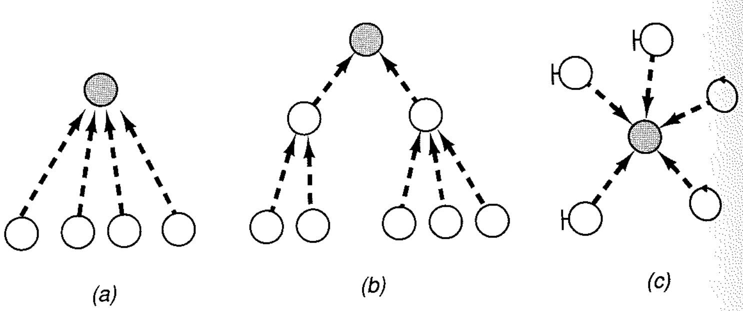


Figure 11.2 Objects that are inherited by many other objects, either directly (*a*) or indirectly (*b*), or are accessed by many others (*c*), could be a base for a component

Today there exist several such libraries for various environments. To give you an idea of how such components could be used, we will discuss some examples of how we can find places to use these components.

The analysis and design models provide a strong framework for finding such places. In particular, objects, associations and attribute types should be reviewed. If the component exists in the library we could use it directly, otherwise proposals for new components are needed. Here are some examples of places where components could be used.

- General **entity objects** that are used to develop other entity object, for example if they are inherited by other entity object or accessed by other entity objects, see Figure 11.2. This is then often an application-oriented component. If a certain kind of object will be implemented in the same way all over the system, such as an entity object whose information should be stored in a database, a general framework could be developed as indicated in Chapter 10.
- **Interface objects** can often be implemented using components for windowing systems. Windows, buttons and scroll bars are then typical components. In the same way, one application area could have similar interfaces which then could be a general framework for all applications in the area. Additionally, general frameworks for windowing systems and tools for graphical user interfaces (GUIs) should also be used when developing the system interfaces. Note here the conflict; if you use a modern UIMS, you normally do not need any components for window management.
- Some **control objects** will have general functions such as logging activities, data collection for statistics and on-line help

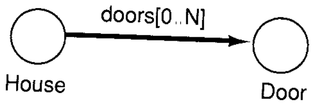


Figure 11.3 An acquaintance association with a cardinality more than one will typically be implemented with a component that holds all the references.

functions. These are often modeled using an extends relation and are used in different places and possibly also in several applications. Then we could have for instance a general on line help function that is a reusable framework in various applications

- An **acquaintance association** is often implemented with a reference. If the cardinality has a fixed interval we use a static structure such as an array or collection to hold these references. If fast searching is required, other structures or storing strategies could be possible such as binary searching and/or tree structures. If the cardinality is variable, see Figure 11.3, we use a dynamic structure such as a list, dictionary or a collection, depending on what is available and which is most appropriate
- Different kinds of **types** will occur at various phases, for example attribute types, types of parameters and local types when implementing the blocks Some of these types may be general and thus can be implemented with components. The criterion is whether the type is a good abstraction, namely a conceptual simplification without involving implementation details If the type only has read and write operations, this indicates that it is not a good abstraction. Then it is not a potential component To make it a component you should raise the conceptual level of the operations.
- During construction we mentioned that our system should be robust for changes in the **implementation environment**. This was achieved by encapsulating the environment. This encapsulation could be done by using components as discussed in Chapter 8. By using these components whenever a service is to be used in the environment, we only need to change these components when changing the underlying system.

To be able to use components, it is essential that it is clear early in the development which components are available. The best case is when the components are available from the beginning. Then the developer can view them as a natural tool during all development work. In some larger projects/organizations it is sometimes appropriate to start with a development of application-oriented white-box components and frameworks Then what is essential is a good application domain knowledge to develop components that will generally be used for different applications. Version handling, configuration management and releasing of such systems to the developing organizations must be managed with care.

11.3.2 Implementing with components

Components are used to construct blocks The use of components is therefore not documented in the same way as design objects. When using components in the design model, various techniques can be used to document the use The use of the most primitive components is normally not documented in the design model; these are used more like programming language primitives. Components are normally introduced as a special case of object modules, namely on the most primitive level in the design. However, for more complex components it is often meaningful to include them in the design model. Especially frameworks and other components that involve major design decisions should be included in the design model Blocks can also be reused, but they are reused to configure the system for different customers. Examples of reusable blocks are Account in a banking system and Line in a communication system.

It is often claimed that inheritance is fundamental for reuse since you can inherit components and thus reuse them. Although inheritance is a strong tool in many contexts, this is a misconception. Inheritance is not a prerequsite for reuse It is fully possible to reuse without inheritance. Actually, inheritance has frequently been misused for reuse issues. Imagine you want to implement a class with a list component. One possiblity is of course to inherit it to use it as shown below:

```
class house:private list{
  head* doors;
  };
```

Here the class house inherits list. This is in most cases an inappropriate use of inheritance as discussed in Chapter 3; a house is not a subtype of list, it only uses list for its implementation, that is, it uses it as a client, and hence from the outside. So when implementing the class you should have an acquaintance association to the component instead, as in the following code segment:

```
class house{
list* doors;
  };
```

In this example it is obvious that inheritance should not be used, but, as we stated when inheritance was first introduced, it is not always obvious how to differentiate between the use of inheritance and acquaintace. However, as a first rule, use composition when implementing with components.

A common need when you use components is that you must adapt or specialize them to the application in question This should normally not be done by making changes in the component, but instead you should either encapsulate the component in another object that acts as an interface between the application and the component, or make a specialization of the component. The previous technique can also be used when you want to solve naming problems or encapsulate components to make the application independent of the component's implementation (language, software or hardware etc). When you encapsulate you should be careful to avoid efficiency losses.

Another problem in using components is that they are written in another language or another environment than the relevant development environment This will probably be of less importance in the future when it becomes easier to bridge such gaps.

The use of components leads to better code. This is a consequence of their frequent use and the fact that they consequently have been thoroughly tested, but also a result of the fact that more work has been put into designing the component than is normal for ordinary program code. Components should be looked upon as 'fault free' by the application developers This means that faults should not initially be looked for in the components

Components should be used in the implementation. It is important for any manager to be serious about the use of components. You should encourage the use of components and question any design or implementation that does not use components. Hence it is obvious that to judge a developer's productivity in terms of the number of lines of source code is the opposite of this idea. The more you reuse the less you will write yourself. Thus a better productivity measure is the number of components used in the design or implementation. This will also increase the quality of the system.

11.4 Component management

A **component system** is normally not profitable for only one project. Such systems should be shared among several projects. The reason is that the development of components is often more expensive than the development of ordinary software and this makes it unprofitable in terms of a project horizon. The real benefits come when a component could be used in several projects and products. Therefore component management should be based on multiple projects.

A special component management department or group is therefore necessary, that is responsible for building a specially made component library for the organization. This department should also be responsible for obtaining components from the market. Additionally, they should encourage and enforce the use of components in projects.

Besides such a component department, and especially if it does not exist, a spontaneous component activity will evolve. We have seen that this is particularly true when working with an object-oriented programming language which encourage the use of components. These activities will be on the individual, group and project level and they are often done as a developer adds some extra effort for the design of a specific class or module that she or he sees will be usable in other contexts too This should be encouraged by the project management since it could form a source of good components.

There are two types of activities for component management; one for the design of a complete component system and one for construction of individual components. We will discuss both of these and we start with the latter

11.4.1 Construction of components

The construction of components is fundamental for their use. Reuse does not come as a side effect. Specification, construction and testing must all be done for reuse. This makes a component more expensive (up to 10 times) to develop than other software.

Several different criteria for what is a good component have been suggested. These criteria can be summarized in the following areas

- The first is *understandability*. The component shall represent an abstraction. It shall have high cohesion and offer only the operations needed to make it useful in an efficient manner. It shall also have a well-defined interface, both syntactically and semantically. If two operations in two different components have the same name, they shall act in a similar manner Their style should be similar to facilitate understanding.

- The component should be *independent* of surrounding entities; it should be loosely connected and thus have low coupling to other units. An object-oriented philosophy leads to this independence.
- The component should be a *general* abstraction which is useful in several applications without having to undergo changes. It is standardized with respect to name, fault handling, structure and so on.

Understandability must not only be external, but also internal. Since good components will have a long life, they will be maintained for a long time. This means that it is especially important that they are written so that they are easy to maintain, even if people should not make changes in them One problem in the maintenance of component systems is the need for (backward) compatibility. Even if we improve our component system, perhaps in performance or interfaces, it must be possible to use the new version or use an earlier version This means that we must also be able to handle versions of components, just as for all other products This requirement is a great problem but must be solved in order to have a working component library.

The most important criterion for reuse consists of the interface of the component. It must be general and complete enough to make the component easy to reuse

A trivial but very common problem is the name Generally it takes a long time to get a good name, a task that is often underestimated but extremely important. The most obvious need here is to have standards. Naming standards can be specific for a project but should apply to larger areas than that to be really useful.

As mentioned before we can use inheritance in our component design. The inheritance mechanism is very useful for building up a powerful component library However, the use of inheritance is also fundamental for the quality of the component system. Booch and Vilot (1990) note that trade-off between different inheritance hierarchies is a fundamental design issue.

Time and/or memory performance of components is also a critical issue. As stated earlier, a general implementation that leads to an inefficient component should be avoided since it will probably not be used. So different implementations for various requirements should be avoided. For instance, Booch (1987) offers several different implementations of each abstract datatype.

Several various criteria for good and reusable component designs have been proposed, see Johnson and Foote (1988), Meyer (1990) and Lieberherr and Holland (1989). These criteria includes:

- Reduce the number of parameters; fewer arguments will lead to more cohesive operations that also imply more primitive operations,
- Avoid using options in parameters; any options should be set by the creation procedure, and special operations should be used to change these options,
- No direct access to instance variables; to access instance variables, special operations should be used, this frees the implementation from the actual interface,
- Naming should be consistent; operations in different components should have similar names if they perform the same task.

11.4.2 The component system

A component library is something that all mature development organizations should have. The construction and evolution of such a library does, however, involve many questions and we will discuss some of these here This activity includes selecting, classifying and managing the included components and also the development of new components. Somebody should also be responsible for making sure that information about the component library is available and spread throughout the development organization and that the components are accessible.

We have stated that a component library should preferably be shared between several different products. This means that the component system should serve several projects. Proposals for new components normally come from the projects that the component system supports. It is important not to be passive in the search for new components by merely waiting for proposals, but you must also actively search for new components that could be suitable. All our experience shows that to have a component system working well, the people working them to be active towards the actual projects. Figure 11.4 proposes an organization for a component system.

The proposals should be reviewed by a group consisting of experienced designers and also someone from the component department forming a software component commitee They shall judge whether the proposed components are to be developed or not.

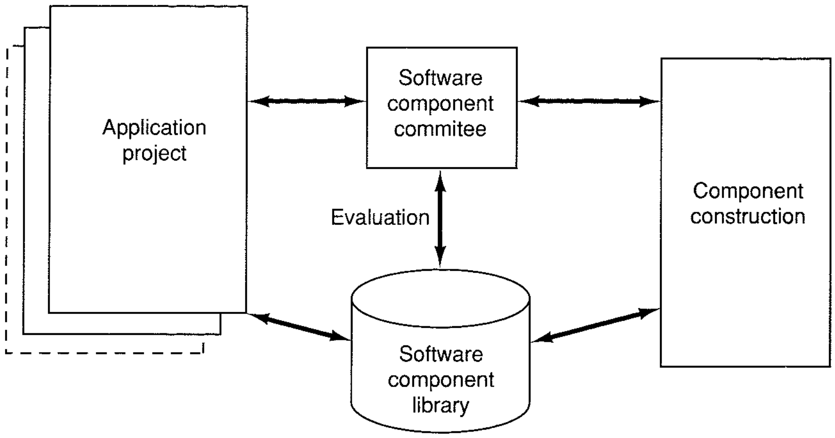


Figure 11.4 An organization of component management.

If it is decided to construct the component, it is forwarded to component construction with a deadline. When ready, it is added to the software component library which then takes a new revision state

As the component system is being used, the software component group should analyze its value; which type of components is used most? Which are not used at all? How much can you gain from using the components? This analysis helps in developing the component system A similar manner of handling the component library is described by Matsumoto (1987)

When you decide to delete components from the library, great care must be taken. Since applications that have been designed can be built on these components it is important not to remove the basis for these designs, thus you must make sure that the documentation will not be removed completely. It is not until all users of a component have changed that a component can be deleted. A component will thus have a life cycle as illustrated in Figure 11.5.

When you start the design of a component system you are faced with some important questions.

- In which programming languages and development environments (eg. compilers) shall the components be offered? How should various environments be handled? Does this work for the target environment?
- Are components already available internally? How are they used? Can we use them for our component system? Should they be integrated?

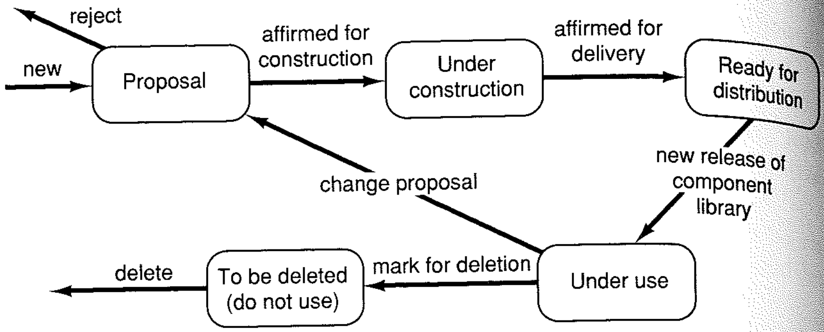


Figure 11.5 The life cycle of a component

- Which external component systems can we use? Should we purchase them or use them as a source of ideas?
- How shall the components be used? Which classification method should we use? How can we simplify the use of components? What should our component survey look like?

When you have obtained a basis for your component system this should not be a static library, but quite the opposite. It should rather evolve continuously and be extended gradually.

Then, which components should be included in the library? Initially, a study of existing, mainly external, libraries should be performed The basic components, such as data structures and graphical facilities, should exist in the library. These are normally bought from an external vendor or accompany the development environment used.

The most essential factor is that the components should be profitable for the organization. Here the most important criteria are the potential reuse of the component and its size. The more places it is used, the more its profitability.

A component proposal consists of the desired functionality (interface and characteristics), its reuse potential (how and where it will be used) and a date for delivery. The proposal can concern development of a new component, purchasing a component or a proposal to change a component.

Whether a new component should be developed or not is determined by the software component commitee. The value of it should then be judged. A technique for this is discussed below.

Purchasing a component is often cheaper in the beginning, but may involve trouble in, for instance, maintenance and support. The purchased component should be incorporated in the structure used and should also be documented in the standardized way used in the component system.

A change proposal could be of three kinds. An *error report* is always accepted. A *change of the implementation* is accepted if it is reasonable, such as an optimization. A *change of the interface* is accepted only if there are very good reasons, since it affects everyone dependent on the component, application developers and other components.

To determine the value of a component, a cross reference table could be used which points out where a certain component will be used. Such a table will help in motivating which components should be developed. This table also serves as a tool for determining when components must be ready for use. Many proposals will not become components. This is essential to note early so that the development of these parts can be carried out on a project basis.

Statistics should continously be collected concerning the use of components. Every use of a component should thus be noted. Then the software component commitee could see which components are reused and use this a a base for the evolution of the component library. As the library grows it may be necessary to restructure the library.

A more active way of finding components has been proposed by Caldiera and Basili (1991). Various metrics of what is typical for a reusable component are used to identify components. A program that calculates these measures for existing source code has been developed and produces candidates for components. Hundreds of thousands of lines of C source code have then been analyzed. The result looks promising. The metrics used include:

- *Size* This affects both reuse cost and quality. If it is too small the benefits will not exceed the cost of managing it. If it is too large, it is hard to have high quality.
- *Complexity*. This affects also reuse cost and quality. A too trivial component is not profitable to reuse and with a too complex component it is hard to have high quality.
- *Reuse frequency*. The number of places where a component is used is of course important too.

The release of a new version of a component library must be timed in cooperation with the projects. The projects should of course also know which components are included in the forthcoming releases. Since many components will be used in different versions, it is essential to keep track of where a certain component is used.

As we have stated earlier, only when no product uses the component can it can be deleted from the library.

One problem that will occur when several different class libraries are to be joined is that they will not be compatible with each other. Today this is a big problem since many different component libraries have components with the same name and they are not disjoint. A possibility is to split them, but since the libraries often use other components in their own library, they may be almost impossible to split to disjoint part and then join. Hence they will not work together. This is a major problem today when a component library grows, and here standards are necessary.

11.4.3 Documentation of components

Although we do not discuss documentation in this book, we think it may be appropiate to mention some in this context since the documentation of components must be made from several views. Since the components are used in various ways we need different kinds of documents for different purposes.

Firstly we have the projects which use the components. A component is documented for use by a **component data** sheet. This sheet has the same purpose as a data sheet for hardware components, namely it describes what a user needs to know in order to use the component. To find the correct component we use a (possibly automated) **component survey**. This document presents the entire component library and the developer searches in this document to find the correct component to use. This document should be generated from characteristic information attached to each component. Different organizations of the component survey will be discussed shortly. For maintenance purpose, we also have a **component description**. This document describes the implementation of the component and is used by the organization maintaining the components. The relationships are illustrated in Figure 11.6.

The thing that differentiates industrial software development from other programming is above all the large scale. To get the real benefits of reuse we therefore need a large number of components. We have previously discussed some problems with a large component library. Another problem is the structuring of the library. To have too many components in a library is not recommendable today; the technology to handle them simply does not exist. The problem of classification is one part of such technology. We can compare this with hardware components. There are many different types of circuits with a certain functionality. They have different sensitivities to

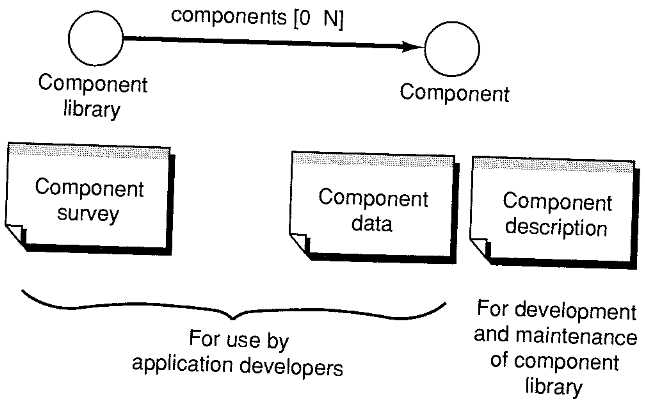


Figure 11.6 Documentation for a component system.

temperature, they have different power consumption, and they can drive different types and numbers of circuits on the output and so on. Here you search for the desired logical function and also for the properties This must also be done for our software components, the question is which criteria are important We should include criteria such as efficiency, speed and storage requirements.

Today's classification methods can be characterized as hierarchical methods and keyword-based methods. In **hierarchical** methods the components are classified in a tree structure where you start searching from the root and as you get down into the tree you indicate which properties the component should have. When you have reached a leaf node in the tree, you have found your component. An example of a hierarchical method is suggested by Booch (1987). His taxonomy starts by first classifying components into *structures*, *tools* and *subsystems*. Structures are components that can be described as abstract data types or abstract state machines. Tools are components that describe an algorithm abstraction. Subsystems are components that are built of structures and tools, see Figure 11 7 Structures are in turn divided into monolithic (the structure must be treated as a whole in order to be meaningful) and polylithic (it is meaningful to speak of parts of the structure). On the next level you find your abstract data types. When you have reached this far, you continue searching to indicate exactly which properties the component shall have. Should the component work in a concurrent environment? Is the size of the structure static? When you have answered all these questions you have found your component in a leaf in this tree.

The other classification method is built on **keyword-based** searching. Here we try to describe our component in descriptive words that we then can search for in a database. It is self evident it

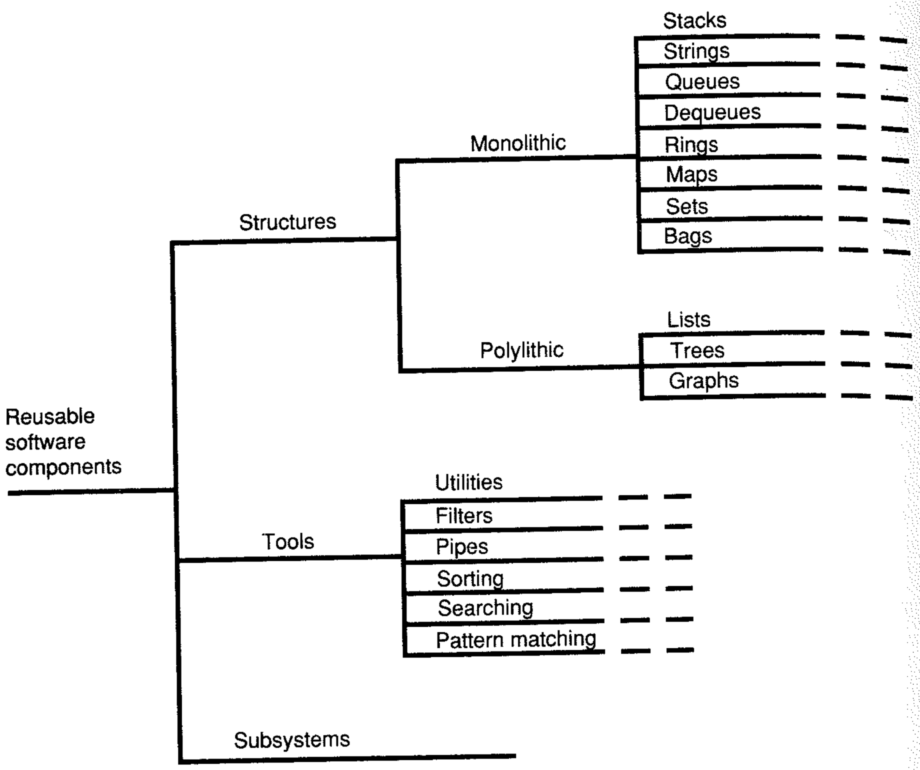


Figure 11.7 Booch taxonomy for classification of components.

is important to have a standardized terminology that everyone can use. Examples of methods to name components have been proposed by Prieto-Diaz and Freeman (1987) and by Cox (1987). The first one is based on the fact that each component can be described in terms of its function, how it is executed and its implementation details. Each component can then be described with a 6-tuple ⟨function, object, medium, system type, application, application field⟩. A word list of synonyms is an aid to using standardized words

Of course both methods have advantages and disadvantages. A hierarchical method is good since it is simple and logical to use, and it is also relatively easy to introduce new components if the structure is good. The problem is that you cannot describe many-to-many relations with it, which means that if you have entered the wrong subtree in the taxonomy you will never find the right component. If you find the taxonomy is not very good, it becomes difficult to work with, and if it appears that the structure should be changed then it is a big job to rebuild your component library. The keyword-based method has its advantages as it is very flexible when it is controlled by ordinary language. You yourself can also select the resolution you want since you can make a selection on different words (indexes). The disadvantages are that you are forced to change or at least extend, the search keys as the library is growing. The method also requires a uniform vocabulary To cope with the deficiencies of both methods, new methods have been suggested in which properties from these two methods have been mixed, see Browne *et al.* (1990).

However, to get a working reusable library, the size of it is not critical For one product using Objectory, the team has developed about a dozen components. Most of them are quite primitive, but some are larger frameworks. These component are very reusable and increase the productivity significantly. The system is built using C++ and in almost all operations (member functions) at least one component is used Additionally, the components are also used to make the code look cleaner and more easy to read than plain C++ code. These components, together with the system structure, help the team to very rapidly do further development. One new use case is typically developed in one or two weeks. The lesson to learn is to start using a small and carefully selected component library today, and add new components as they are needed.

11.5 Summary

Reusability in software engineering is not a new idea and it is a critical issue for a substantial increase in productivity. Still, it is not widely applied in the software engineering community and there are several reasons for this. Object-oriented technology is a promising approach to enhance reuse. The reusability not only concerns code, although this is the main topic of this chapter. Reuse of components should raise the abstraction level of our programming efforts and thus increase the productivity.

A component is an implemented abstraction that is general and has a high quality, and also is developed and packaged with the aim of reuse. A component could either be white-box, which means that you have to modify its internals to reuse it, or black-box, which means that you do not change the inside of it. A special kind of white-box component is a framework, which is a larger skeleton of a design that is reuseable.

The models used in OOSE provide a strong tool to find places to use components. Components should be used widely in the implementation. A construction not using components should be reviewed with skepticism and discriminated against in favor of constructions that use components widely

Component management should be based on multiple projects. Since the construction of components does have other requirements in terms of quality and documentation than ordinary software, the construction of components should not burden a project. For component construction the fundamental issues are understandability, independence and generality. All of these issues should be highlighted in the component interface.

Component management includes management of the component library. This should evolve in an ordered fashion and support increasing reuse in the organization Proposals for new components come from the projects; these proposals are then implemented by the component department It is of great importance that the component department is active and actively promotes the component library. Documentation of components should satisfy both the use and the maintenance of components. Various methods exist to structure the component library for easy retrieval.

12 Testing

12.1 Introduction

To test a product is relatively independent of the development method used. This chapter describes testing in a manner comparatively independent from the method, but we will nevertheless see that OOSE provides some new possibilities, but also some new problems. The chapter does not claim to give a full description of testing activities, but is included to give a complete picture of the development and in order to indicate what differs for object-oriented systems. The standard work on testing is still Myers (1979) and a more comprehensive description has been made by Sommerville (1989)

The test activities are normally divided into **verification** and **validation**. Verification is the work involved in checking whether the result agrees with the specification, whereas validation is the work necessary to check whether the end result is what was actually wanted The two terms are usually summarized as two different questions:

- Verification: are we building the system correctly?
- Validation: are we building the correct system?

Here we will only discuss the verification part, but internally we also do validation on specific blocks. Nor is testing the only activity necessary to obtain a qualitative product, but activities like reviews and code inspections must also be performed We will discuss some of these issues in Chapter 15.

The fact that we discuss testing in a chapter of its own does not mean that the test activities commence only when the analysis and construction have been completed. Testing is actually integrated into all activities and we will see that the test work can, in fact, begin at the same time as the analysis work is started. The more the testing is integrated, the better Software development is an incremental

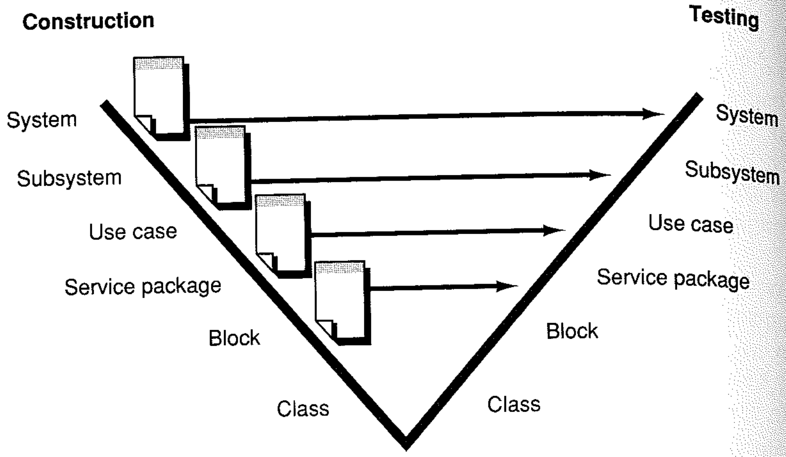


Figure 12.1 The test activities are performed in reverse order relative to the construction activities. Documents prepared on one level are used on the corresponding test level.

activity consisting of analysis, construction and testing that what is done is done correctly.

In this chapter we focus on the testing activities that concern testing the final product by executing the program code. This is sometimes called **certification** to differ from other types of quality activities. Testing can also be done on, for instance, document level by verifying that the analysis as well as design are done appropriately. We will only mention some approaches of these kinds of testing activities, but not go into them in detail.

The most important aspect of testing is actually the attitude towards it. You must be aware that testing takes time and the cost must be allowed for. The test activities take up perhaps 30% of the entire development cost, but may sometimes exceed 50%. Testing is consequently a large part of the development and must of course be planned in the same manner as analysis and construction. Testing should thus be included in the project plan. Much effort must be spent on planning the testing; it should not be something that is scrambled together quickly at the end of a development phase. We must obtain the best possible result for our invested resources.

We will first discuss testing in general and different types of tests. Then we illustrate how the testing of individual modules is done, and how their integration is done. We will see that the use cases give us much help in the integration. This kind of testing is normally done in reverse order to design, see Figure 12.1.

12.2 On testing

12.2.1 The purpose of testing

Let us first define some important concepts. We follow here the IEEE (1983) standard. A **failure** occurs when a program misbehaves. Thus a failure is a (statistical) property of the system in execution. A **fault** exists in program code. When the code is wrong, the fault can be fixed by changing the code. A fault, if enountered, may cause a failure. There is no fault if the program cannot fail. An **error** is a human action that results in software containing a fault. Thus an error may lead to the inclusion of a fault in the system, making the system fail.

The first lesson learned about testing is that you can never prove that the program will never fail; one can only show that it contains faults. We should therefore consider a test that has found many faults as a successful test and not the opposite. As a curiosity we can note that we have a paradox here; if we have made a successful construction, we will have an unsuccessful test, and if our construction was deficient, then we will have a successful test. For completeness, we will mention that active research is continuing for proving program correctness, but even if such techniques are used on a small scale today, we believe that it will (unfortunately) be a long time, if ever, before they can be fully applied to the systems discussed here. For more on such formal techniques, see Wing (1990) or Hall (1990).

The purpose of testing is to find faults. The testing is thus a destructive process to some extent; we must show that something is incorrect. It is therefore unsuitable to test your own construction yourself since it does not feel natural to work only to prove that you yourself have made an error. Unfortunately, however, it becomes very expensive to have someone else test all you have done in detail, and usually you can only afford to test major parts in a special test department. This is normally done during the integration test. The developers should of course be given the opportunity of identifying these faults, however.

An alternative view of quality in software is to do it right the first time. Then any faults introduced should be eliminated as fast as possible. This is one of the major ideas in an approach called **Cleanroom Software Engineering**, see Mills *et al.* (1987). However, space does not allow us to discuss this further here.

The above discussion indicates that the testing is generally done bottom-up. We start by testing the smallest modules and as their construction is completed, and we cannot find any more defects in them, we put them together and test the next level, and so on.

There are also methods in which top-down testing is used. These methods are then only based on the fact that the system is designed top-down and that it is tested continuously. The underlying modules consist only of stubs from the beginning which may return random values, and they are then replaced with the real code.

It is a well-known phenomenon that when you correct detected faults, you introduce new faults in the system. When a fault has been corrected, you may therefore have to retest your system from the beginning, including the blocks that use the corrected block. It is to be hoped that you introduce fewer faults than you correct. Levendel (1990) has noted that you generally introduce one new fault for every third fault you correct. He also notes that 'defect avoidance is more powerful than defect removal.' Defect avoidance, he claims, is achieved with a structured development method, and this is something we strive for in OOSE.

12.2.2 Test types

There are many types of tests. To avoid misunderstandings we will discuss the most common types here. They are not independent of one another and when you test you should use several of them in combination.

- **Unit testing** means that one and only one unit is tested as such. This test requires that the unit is independent of other units. The unit can be a procedure or a class, but it can also be a module or a group of modules. We include classes, blocks and service packages in these units.
- **Integration testing** involves tests with the purpose of verifying that the units are working together correctly. Integration and unit tests can be the same tests (eg. for a block), it is the testing method that differs. We use use cases for this type of test. We will see that use cases are an excellent tool for this type of test. Blocks, service packages, subsystems and the entire system are tested in this manner.
- A **regression test** is made when you have made changes in the system, for example corrected a fault, and the test's purpose is to verify that the old functionality remains. This test is one reason why you should use automated testing which will be discussed later. When a tested unit is changed, we may need to change the test specifications also, since any changes of the

code may lead to that old test cases do not test what their original purpose was.

- The **operation test** is the most common large-scale test. Here the system is tested in normal operation for a longer time. The system is used in the intended manner. Only normal mistakes are made, that is, mistakes that the normal user may be expected to make. If the system is to be reconfigured during operation, then this should be tested also. This type of test also measures the reliability of the system, and thus statistical measures may be used, such as mean-time-to-failure (MTTF)
- A **full-scale test** is the natural continuation of an operation test. The test means that we run the system on its maximum scale. All the parameters of the system approach their limit values, all types of equipment are connected, many simultaneous users are present in the system, and many use cases are in operation at the same time. This test requires a lot of the system, but these requirements must be managed by the system. It requires that a full-scale plant is available and the test is therefore often expensive, but experience shows that new failures arise with full scale
- A **performance test** or **capacity test** has the purpose of measuring the processing ability of the system. The test shall be designed so that you can measure the performance with different loads What you want to measure may include store allocation, CPU utilization or perhaps only the speed in a specific use case The measured values shall be compared with the required values.
- **Stress testing** means that the extreme limits of the system are tested. An **overload test** is a special type of stress test and is also related to the performance test Its purpose is to see how the system behaves when it is subjected to overload and consequently goes one step further than the full-scale tests. We cannot expect the system to manage the processing of these tests, but it should perform well and must not stop. The system should survive occasional load peaks. How much the system's performance drops is interesting to measure and we consequently also have a performance test in this case. Checking should be done so that no catastrophes occur.
- A **negative test** is a type of stress test intended to subject the system to stresses beyond what it has been built for. If there are any special barriers they should be tested and verified and use cases that normally will not be invoked simultaneously, should be executed at the same time, and so on. The system

shall intentionally and systematically be used in an incorrect manner. This maltreatment should be carefully planned so that especially critical places are tested

- **Tests based on requirement specifications** are those that can be traced directly to the requirement specification. They can be one of the earlier tests, for example a performance test or full-scale test, but they can also be a particular requirement that you want to check explicitly.
- **Ergonomic tests** are becoming increasingly important as computer systems come to maturity. If the system has a man–machine interface then the ergonomic aspects should be tested. Is the interface consistent in a use case? Is the interface consistent between several interfaces? Are the menus readable? Are system messages visible? Can you understand the failure messages? Does the system provide a logical picture?
- **Testing of the user documentation** is a type of ergonomic test in which the system's documentation is tested Both the user manual and the documentation for maintenance and service should be tested. Far too often manuals and system behavior are not consistent. Its readability should also be tested, the documents should be checked for language, the balance between chapters and the balance between text and pictures should be assessed, and so on
- **Acceptance testing** is normally performed by the organization ordering the system and it is the final checking by the customer. The system is now tested with real data. This is often also the validation of the system. This checking is often done against the original requirement specification. Sometimes the requirement specification is deficient and consequently this checking should not be done slavishly. This type of testing is often also called **alpha testing**. This test can be done for a longer time when the system is working in the environment for which it has been developed. When the testing has been done, the decision is made as to whether the product shall be accepted or not. If there is no specific orderer, for example a compiler product, **beta testing** is often used. This means that the product is tested by specially selected customers or potential customers who use the system and report the faults they detect. Beta testing is done before the product is released on the market and is a form of prerelease.

Of course there are several variants of these test types; some have been discussed here but there are also several other types of

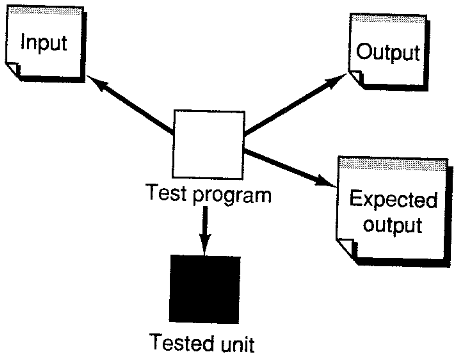


Figure 12.2 Schematic illustration of automated testing.

tests. **Code inspection** is, for example, a method that can be used. Its purpose is that someone inspects the developed code and assesses its quality. This is a very expensive method, but it often produces a qualitative code.

Since testing is an expensive activity it must be efficient. When the test resources are restricted, each test should lead to the detection of a failure. We should strive to find the major failures first. One consequence of this efficiency requirement is that we must be able to do our tests again without any great effort. This means that all tests should be saved, including ergonomic tests, in order to perform regression tests in a simple manner.

It was stated earlier that it is important to plan the testing; testing is not something that is done on an *ad hoc* basis. Each test to be done, with the exception of the most low-level unit tests, should be documented. The conditions for the test should be specified, for example whether the test should be made in a development or target environment, how the test should be performed, expected outcome, actual outcome, and so on. Later on we will discuss what this test specification can look like. The test process is a process that, to a great extent, runs in parallel with the other processes.

12.2.3 Testing techniques

The goal should be to **automate** as much as possible of the testing. This can be done through special test programs with the associated test data. It is then simple to repeat tests, that is, to perform regression tests, in new versions or when some part has been corrected since the preceding test. The principle is illustrated in Figure 12.2.

The **test program** fetches sequences and data from the input data. Then the unit/system is fed with the sequence and the tested system's response is observed by the test program This output can be stored directly on an output file or be compared with some expected output. Hence a (primitive) failure analysis can also be done automatically. The **test data** consists of sequences of stimuli to be sent. An example of such an instruction could be *'send stimuli X with parameters A B C, receive response Y'*. The sequences indicate which data shall be sent and received.

The test program and data are placed in separate files, and can in this manner be reused with several different test combinations. The test program is usually bound to the environment where it is executing, but this does not apply to the test data It is easy to modify a test; we only change the test data files. The goal is to have the test program as general as possible and independent of the test data. We should be able to reuse it for several different tests, and if we can reuse it for several different products it is still better. Test data, on the other hand, can seldom be reused in other tests, but it should be possible to reuse it if we want to make a regression test. We can modularize our test data, however, so that it can be combined in different ways and thus obtain different tests For example, use cases with extends can of course use the test data for the use cases that are to be extended

Other issues that should be considered include the automatic generation of test data. For example, test programs to test different classes will have similar structures. It may be worthwhile to write a specific program that takes a given class and generates a skeleton for a test program or test data for that class

Often object-oriented systems are highly integrated and certain objects are dependent on other objects. Therefore it may be necesary to develop object simulators that simulate the behavior of adjacent objects. These supporting objects may be implemented with only operation stubs

When you test the entire system you need several different test programs To obtain a simple interface to the system and to make the test program independent of the system, you normally build an **interface simulator** or test driver, see Figure 12.3. The test may now be performed in the target environment. This simulator often saves work since the handling of the interface of the system does not have to be described in each test program.

When you develop the test programs and test data you should of course use the same techniques used for all other development. If we have a high level of ambition for our test programs, then we will

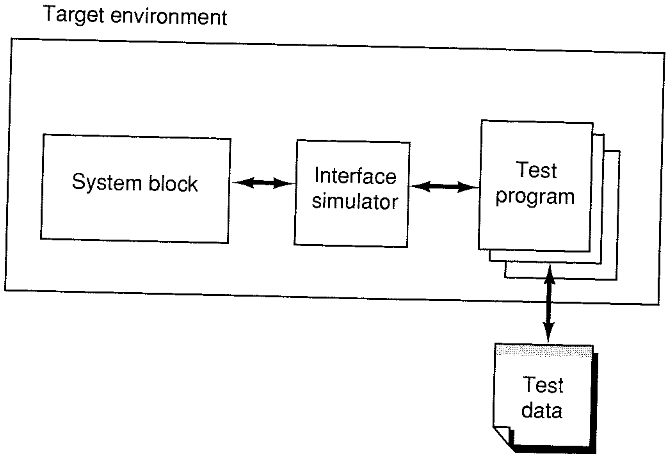


Figure 12.3 Use of an interface simulator

also have products with assured quality. Modularity and reusability shall be the guidelines. By writing good test programs we can build up a store of reusable tests, both test programs and test data, in the same manner as we do for components. These test programs, however, must be developed when our system is ready for testing. Once more we see that the testing must be part of the regular development process.

A question that quickly arises is whether the test programs are part of the tested product or whether they are a separate product. We should regard them as part of the product and they should be placed in a separate block. This block can be used for maintenance of the system and it also simplifies failure finding and fault localization when the system has been installed. By adding new subtests it is simple to perform regression tests.

If the system is to work in critical areas, for example military or banking applications, the inclusion of test programs in the product should be avoided since it may be easy to evade the security routines through a test block. Another disadvantage of this is that the test block may impair the system's performance or create other problems. However, this can be solved by having a separate module that is only loaded if required and has a special place in the memory, for example buffer areas or on special memory expansion boards. Another way is to have the test tools in special hardware separated from the system. Supervisory units can also be placed in this hardware and can thus be protected from the system.

We will now discuss the different test levels, namely unit test

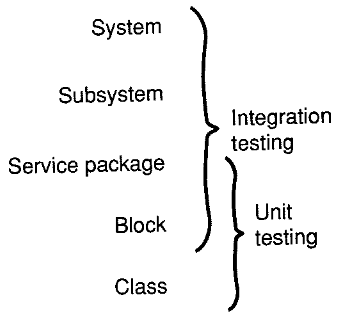


Figure 12.4 Unit testing and integration testing are made on different levels. Note that service packages and blocks are tested in both manners.

and integration test, see Figure 12.4. Service packages and blocks are tested as units, but also as an integration of included blocks and classes. The system test is the last part of the testing. Acceptance test is not described here since it normally is done outside the development activities. The reason is that it is often tightly coupled to the requirements, and this is enterprise modeling. Therefore acceptance testing is a part of the last phase of enterprise development.

12.3 Unit testing

12.3.1 What is a unit test?

A unit test is the most primitive form of testing and it is normally done by the developer himself or herself (see the earlier discussion about costs). To avoid misunderstandings about what is a unit: in unit tests we include the testing of classes, blocks and of service packages. The larger the unit, the more formal the testing will be. In a traditional system a unit test is often a test of procedures. Thus unit tests in object-oriented systems are often made on a higher level.

To do a unit test of object-oriented code is more complex than to test ordinary (procedural) code. This is a result of the object-oriented approach; the program has a flat structure which makes the program flow and the program state distributed. It is difficult for the developer who is dependent on the objects of other designers to test his or her own. This is at the same time a great advantage; the integration test is normally much smoother than for ordinary code. Additionally, concepts like inheritance and polymorphism leads to additional complexity of the testing. We will come back to this later.

Despite these difficulties we must often prepare special test

beds for our testing. They are used as for traditional programming, namely to simulate the surrounding world of the tested unit. Often you are forced to introduce special test operations in your classes to instantiate the test bed. In the extreme case you must build up an entire shadow system that only simulates all the objects of the surrounding system. Normally, though, it should be enough to have a test bed that is one magnitude smaller than what you are testing; one test class for each block, one test block for each service package. Since object-oriented development is a strongly incremental development, it is especially important to save the test bed. As the system is being developed, you also develop your test bed and the testing can comprise more and more activities.

The requirements for debugging tools are also greater for object-oriented systems. Normally the environments contain support to inspect the object structure during execution and other service packages. This is (usually) a standard in environments for Smalltalk, C++, Simula and Eiffel. These tools are invaluable for fault finding. For more on debugging tools of object-oriented systems, see Purchase and Winder (1991).

It is also common that special statements are inserted for printout only for the debugging. These statements are then usually connected with a condition that indicates whether you want these help printouts or not. This condition is checked either during execution or, better still, at compile time. If you can check this at compilation, you will not have to include these statements on delivery (unless this is desired). Different levels for the printout can also be selected, for example how often you want a printout and what it shall contain. Eiffel has additional support for failure detection, namely through assertions, that is, conditions for when a routine can be executed and what the result of the routine should be, see Meyer (1988).

When you test a unit there are generally speaking two methods. **Structural testing** (or **white-box testing**) means that we use our knowledge of how the unit is designed internally for the testing. **Specification testing** (or **black-box testing**) means the opposite; we test without any knowledge of what the unit looks like internally and the only thing you have is a specification of the unit. Normally both are needed since they complement each other. Often you start with white-box testing and, when this has been completed, you continue with black-box testing. An important requirement when these tests are developed is, as for all tests, that they should be designed with a view to future regression tests that are made when new versions are developed.

12.3.2 Structural testing

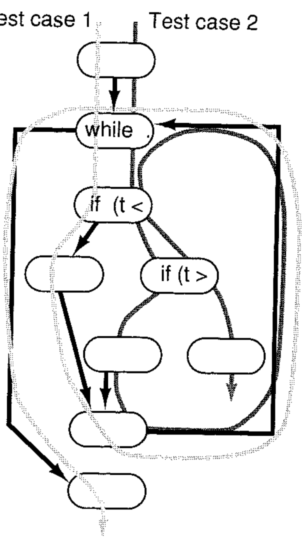
The purpose of structural testing is to test that the internal structure is correct This means that you use your knowledge of how the unit is implemented when you test it It would be desirable to cover all possible combinations of parameters, variable values and paths in the code during the testing, but this is almost always impossible since we would have an enormous number of test cases. Structural testing is sometimes also called program-based testing, white-box testing or glass-box testing To examine the effectiveness of our test cases we can use measures of **test coverage**. The least coverage is to exercise each decision-to-decision path (DD path) at least once. A decision is typically an IF-statement (i e a DD path is a path between two decision statements) This coverage leads to the situation that all statements are executed and thus also that each decision outcome has been tested. The minimum requirement should be that all statements have been executed. Normally this is a reasonable goal for test coverage. However, most problems arise on odd path *combinations* A more ambitious goal is that we exercise all pairs of DD paths. We can then increase our ambition to cover a significant number of DD paths in combination. However, the number of test cases increases very rapidly. Almost never is it possible to test all the possible paths in the system, regardless of parameters or variable values Complete path testing can often only be made locally. Especially critical passages such as loops must be given extra care. To use debuggers is often a great help in these tests since they permit the user to inspect statements that are executed and run-time values of variables.

Consider the example shown in Figure 12 5. This describes a member function in C++ for a class intset to check whether a certain integer is a member of the set. This member function can be tested so that each DD path is tested at least once The test cases are illustrated in the flow graph. Express the test cases in code!

Theoretically, all possible paths in the code should be executed to test an operation completely. In the example we have three alternative paths inside the while loop. The loop can be executed zero or more times. If the loop is executed zero times we have only one path through the function If the loop is executed once we have three alternative paths. If it is executed twice we have six paths, and so on. The total number of paths through the code is thus $1+3+6+9+12+\ldots$, or $1+\Sigma 3^n\ ;n>0$. In practice, this is far too many different cases We must therefore choose an appropriate number of test cases. In this case two test cases will lead to us executing all statements.

```
boolean intset::member (int  t)
(
  int l = 0;
  int u = cursize – 1;
        // Binary search
  while (l < = u)  (
    int m = (l+u)/2;
    if (t < x[m])
       u = m – 1;
    else if (t < x[m])
       l = m+1;
    else return true;
 )
 return false;
)
```

(a)



(b)

Figure 12.5 The member function in (*a*) can be tested with only two test cases illustrated in (*b*) to do a full cover of each DD path.

Polymorphism helps us to isolate the tests needed when changes are introduced. Polymorphism is a very strong tool which often make changes easier to incorporate If you add a descendant class and test this class on its own, you do not need to test client classes again, so long as you have used inheritance for sub-typing, see Chapter 3. In procedural code, however, you not only need to add and test the new functionality, but client codes must be changed and tested also, as we will indicate below.

One of the major strengths of polymorphism is that it hides the complexity of the code However, this makes testing of the code more complex. Consider the class hierarchy shown in Figure 12.6.

We want every shape object to be able to be drawn. Hence we have a virtual operation in Shape called Draw. This means that whenever a user of shapes wants to show the object on the screen, he can use the program indicated below:

```
figure : Shape;

figure Draw;
```

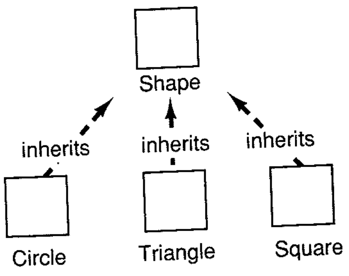


Figure 12.6 A class hierarchy of shapes.

Here we first declare a variable figure of type Shape. Then we can assign figure to an object of any of Shape's descendants. Since they all have a Draw operation, although differently implemented, we do not need to know which class the objects belong to. This is the polymorphism mechanism. In a traditional language we would need to know exactly which type the object belonged to, since we need to call different procedures for different shapes. The code fragment would look something like:

```
figure: Shape;

CASE figure type IS
  Triangle:  DrawTriangle(figure);
  Circle:    DrawCircle(figure);
  Square:    DrawSquare(figure);
ENDCASE;
```

In this latter case we can see that we have three different paths to test In the object-oriented code fragment this complexity was not obvious. Test coverage based upon DD paths, discussed above, is thus much harder because of the polymorphism. You do not see in the code which unit is actually invoked, not even in a strongly typed language. It looks like a stimulus is sent to an instance (of a specific type in a strongly typed language), but during execution, an instance of any class (of the descendants in a strongly typed language) can actually receive the stimulus. In traditional language the operation to be invoked is made explicit. Hence every stimulus in OO-code corresponds to a CASE-statement in procedural code.

The discussion above about test coverage and paths to be tested illustrates that these concepts make testing of OO software much harder. However, in the discussion we have taken a traditional view based upon code being sequences to execute. This view, however, is not the view traditionally taken in the object-oriented community The focus is instead on communicating objects that are decentralized. The sequences are still there, refer to the use cases, but they are implicit. The interfaces are explicit though, and this is one of the major strengths of object-orientation. The implicit sequences are tested during integration testing. Hence integration testing is an important activity, and the integration occurs in all development, as the object interfaces are focused upon early.

Another issue specific to OOSE testing is the inheritance mechanism. This often leads to less code, as operations defined in ancestors are also used in descendants, but the amount of testing of such code is not necessarily less To test abstract classes we must focus on two properties. Firstly, we must test that we can inherit the class and that we can create instances of the descendants. Secondly, we must test that any stimuli sent to the object itself (this in C++, self in Smalltalk) work properly The testing program should thus create instances of the descendants and then test this instance.

When a class inherits another class, the inherited operation may also need to be retested in the new context. That an operation worked well in a class does not necessarily mean that it will work well when it is inherited, since it now executes in a new context This is actually an issue for integration since it concerns the integration of classes We have at least two reasons for an inherited operation not to function in a descendant:

- If the descendant class modifies instance variables which the inherited operation assumes certain values for,
- If operations in the ancestor invoke operations implemented in the descendant.

The first case can be avoided by only using the ancestor class in such a way that it is always consistent, namely by not changing instance variables in an improper way. This can be accomplished in C++ by defining certain operation or instance variables as private, that is, they are not accessible in the descendants, otherwise this must be controlled by coding rules. The second case can be avoided by testing the operation in the descendants that are invoked to be sure that it really fulfills its specifications. If the new descendant does not interact in any way with the instance variables or the operations inherited, we do not need to retest these in the descendant.

Overriding of operations in the inheritance hierarchy is also crucial for testing, as we may conclude from the above discussion. Of course we must retest the overrided operation. We may then also need to develop new test cases as the implementation of the operation is changed. We can view this as the descendant class using the ancestor class just as a module uses another module

Hence inheritance may lead to more extensive testing. Not only when we modify an ancestor class must we retest the descendants, but also when we add a new descendant we may need to retest the inherited operations. In the worst case, we may need to develop unique test cases for every level in the inheritance hierarchy. By designing the test cases optimally, several of the discussed test principles can be covered in the same test. Perry and Kaiser (1990) have also discussed the problems of unit testing object-oriented programs.

12.3.3 Specification testing

Specification testing, or black-box testing, has the purpose of verifying the input/output relations of a unit The goal is to verify the specified behavior of the unit, that is, *what* the unit does, but we are now not interested in *how* the unit solves this We send stimuli with different parameters to the unit and as output we receive new stimuli or perhaps we see a change in some variable If it is essential we must also test the input/output relation in different states of the unit.

The selection of test data for procedural code is based on the parameters of the procedure For object-oriented code this is not sufficient, here we most also take into consideration the state of the object to be tested Thus you cannot view the operations of an object as independent, to be tested separately We must also see how the objects are to be used, their life cycle. This is especially important for state-controlled objects

Equivalence partitioning is a technique to reduce the number of tests that we need to perform. The objective is to select a reasonably small number of test cases, out of a large number of possible test cases, so that the probability of finding faults is high An **equivalence class** is thus a set of values for which an object is supposed to behave similarly The idea is then to write test cases to cover all equivalence classes. For example, if we want to test an object Stack, we may write test cases for when the Stack is empty, loaded and full. Often it is at the boundary values of these equivalence classes that failures occur

The technique of equivalence partitioning should be used in specification testing. For every operation we identify the equivalence classes of the parameters and the object state. Then we select typical test data for each of the equivalence classes. Most important of course

Table 12.1 The state matrix for testing All combinations of states and stimuli can be tested on the basis of this matrix

Stim \ State	s0	s1	s2	s3	s4
Stimulus1	ok	ok	wrong response	ok	fail
Stimulus2		fail	ok		
Stimulus3		ok	slow – check it		
Stimulus4					

is that the unit works with normal input values. Ordinarily it works for them, whereas limit values tend to cause problems, for example, when a queue is empty or if a parameter is on the verge of forbidden values. If our unit expects values in the range 0 to 100, then we should test, for example, with 0, 1, 56, 99, and 100. We should also test with values outside the range, for example −1 and 110, to see that the unit will not break down The selection of test cases should be made so that failures do occur.

Since variables in object-oriented languages are often references to objects, these references may refer to no object (0 in C++ or nil in Smalltalk). This means that every parameter may be a reference to no object and this thus forms an equivalence class. In untyped languages like Smalltalk, we may also have the equivalence class that the object belongs to the wrong class.

A technique to find equivalence classes from only object pointers is to 'flatten' the object. We then instead view all attributes of the object, and find equivalence classes for the combination of the values of these attributes

Equivalence classes for the output should also be made. From these we can select test data (input) so that we will verify all output equivalence classes.

As our units only communicate through defined stimuli, specification testing is rather straightforward. We have defined which stimuli the unit can receive and which behavior it shall show for each stimulus

A very good tool for this type of testing is a state matrix, see Table 12 1. The matrix indicates all the states that can be adopted by the unit and all the stimuli that the unit is expected to receive in the various states. In reality it is impossible to include all states (all possible variable values!) and all variants of stimuli (different parameters!) Normally the states identified when describing state transition graphs of the objects are sufficient. As the testing is performed you can fill in this matrix. One of the great advantages of this type of matrix is that it draws the attention of the designer to state/stimulus combinations that may have been neglected during the design

States that must not be overlooked are different configurations of instantiations of classes, but also of different configuration of the final system. Above all we must test so that no block is dependent on other blocks that are not certain to be delivered. Here we have a lot of configurations that may be overlooked in the design, but that may arise in the real system

A common mistake when you test the code is not to check the output data. Just to receive output data is not sufficient, we must also make sure that it is correct. This may involve much work, for example in mathematical applications when you must verify the calculation manually, or with another system that you know will produce correct results.

12.3.4 Testing of blocks and service packages

Tests of service packages and blocks mean both unit tests and integration tests. Unit testing has been described, and integration testing will be discussed in the next section. Here we will show some aspects that are specific for blocks and service packages.

This test activity is special in that it is normally performed by the designer himself or herself, but it should be treated formally (at least for the service package), as a test department would do. To test service packages and large blocks you should specify the test in a test specification. The specification indicates what shall be tested and how this is to be done.

Tests of blocks are normally only made by using the above principles and without writing any test specification. The rule that usually can be applied is that the highest block level that can be connected with one single programmer (normally a service package) should be tested with formal documentation The programmer normally does this test himself or herself and it is usually done in the development environment. The testing comprises normal cases, odd cases and any tests that can be traced to the requirement specification.

The testing of several blocks should be done in steps and you should consequently increase the test complexity When a block has been tested, you test it together with another block. When these two are working together, you add another, and so on. When all appear to work, we can consequently change to black-box testing of the service package.

As mentioned earlier we use special test blocks (to test service packages) or test classes (to test blocks). They work in the same manner as test programs that were described earlier.

12.4 Integration testing

12.4.1 What is integration testing?

The purpose of integration testing is to test whether different units that have been developed are working together properly. In traditional development this usually involves a lot of work and comes as a 'big bang' event at the end of development. In OOSE integration testing comes smoothly and is introduced early in the development.

If we test all units extensively, is there a need for integration testing, since all units will be correct as they are? Yes, definitely. When units are combined new failures will be detected. The combinaton of units will increase the number of paths possible exponentially Therefore, on testing the combination of units, failures may be detected that were impossible to detect when testing only a single unit

In integration testing we include testing use cases, subsystems and the entire system. Service package and block tests can also be included to a certain extent, since they also integrate units. There is consequently not one integration test in a development, but rather the test is performed several times on different levels. Normally integration testing is made by a special testing team in the project. Here the documentation is frequently more formal than in the unit tests. The testing done by the test team should usually be performed in the target environment, that is, the environment in which the system shall execute when in operation.

Before we start the integration test the included blocks must of course be completely designed, tested and approved. This does not mean that we must wait with the integration test until the entire system has been designed On the contrary we can start the integration test already when we have only a few blocks, we can, for example, test use cases when those blocks that shall participate are completely designed and approved.

Integration testing is an activity that we can describe as shown in Figure 12 7. One starts work by planning the testing. This plan is

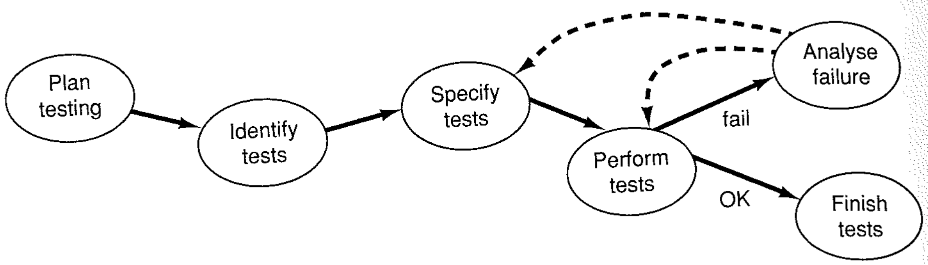


Figure 12.7 Activities in integration testing.

the basis for identifying what shall be tested and then specified in more detail This specification is the basis of the actual performance.

12.4.2 Test planning

As mentioned before, the activity of testing begins early in the development process. The planning can begin when we start the development. We can work on it in general during the analysis phase, but we cannot seriously start preparing the testing until we start construction.

Since the integration test is normally performed in the target environment, the need for testing equipment can be identified at an early stage This equipment can then be purchased or developed within the company and can be installed in good time Usually this picture becomes clear with time, but if we have identified the testing needs, then we also have increased the probability that the appropriate equipment is available when we need it.

The testing guidelines are established early. By determining the method and level of ambition we have created the foundation of the testing. It should be determined whether the testing shall be made automatically or manually, and we can also make an early estimate of the resources that will be required.

When our requirements model is ready we can start the planning in earnest We study whether the existing test programs and data can be used, whether they must be modified or whether we must develop them anew. Using the guidelines as a basis we can determine what degree of coverage our tests should have. We can also define the surroundings for the test By looking at which parts will be ready first, we can also decide which tests can be made first. Never start the integration test until the unit test is ready, deviations will prove crucial! Faults on underlying levels will usually prevent further tests! You can test incrementally, however, by adding new functionality to the blocks as you go along.

The test planning must also consider any standards for the testing and the resources required for each subtest. The plan should not control the testing in detail, but only function as a basis for the test activities

A **test log** is kept during the entire test work. A log should be connected to a version of the system. The purpose of the log is to give a brief survey history of all the test activities, both successes and setbacks Setbacks often require a longer explanatory text indicating the reason for the failure and explaining what action has been taken. The log is filed after the testing and is the basis of the refinement of the test process and the planning of new tests.

12.4.3 Test identification

The identification is to find out what shall be tested For the first time several classes, blocks, service packages and subsystems are brought together and therefore the testing should concentrate on this. Each use case is initially tested separately The use cases constitute an excellent tool for the integration test since they explicitly interconnect several classes and blocks. When all use cases have been tested (at various levels) the system is tested in its entirety Then several use cases are executed in parallel and the system is subjected to different loads.

When we identify what should be tested, we can also estimate the required resources. This is a more detailed estimate than that done earlier and it now acts as the guiding principle for the specification and execution of the test

The test identification also means that we start sketching the test specification During the identification we divide the testing into different test types (the terms were explained in Section 2 of this chapter). For a use case we may have the following tests:

(1) Basic course tests,

(2) Odd course tests,

(3) Tests based on the requirement specification,

(4) Test of user documentation.

In the use case test you perform an operation test of the **basic courses** of the use case, that is, the expected flow of events. Since this is the first time the service packages are integrated, these tests should aim at stress the simultaneousness of the service packages. The tests for **odd courses** are all other flows of events, that is, the odd cases of the use case. If we can trace something from this use case to the requirement specification, it is also identified during the **tests based on the requirement specification**. Similarly, if user documentation is connected with this use case we also make these tests during the **test of the user documentation**. Here we also include tests of the user interface to the system in precisely this use case.

Normally we test all use cases one by one. Those use cases with an extends association to other use cases are tested after testing the use case where it is to be inserted. We thus first test the original use case, and when it is approved, we test what shall be extended by means of the original use case. When we test the use cases, the machine could very well execute other tasks, it is actually an advantage with background noise during these tests. When we test the use cases it may happen that one of them perhaps cannot be tested alone, but that it requires several other use cases to be meaningful (e.g. a use case for supervision). Then special attention must be given to this when we test the system block

The system tests may be divided into the following tests:

(1) Operation tests,

(2) Full-scale tests,

(3) Negative tests,

(4) Tests based on the requirement specification,

(5) Test of the user documentation.

When we test the system the use cases should be tested in parallel, both in step and out of step. We should also stress the system by running several use cases at the same time. All these divisions appear from our division into subtests; the **operation test** is a run of the entire system, the **full-scale test** increases the parameters of the system to their specified limits, and the **negative test** has the purpose of breaking the entire system by breaking through these limits. Tests based on the requirement specification and the test of the user documentation have been discussed earlier.

The performance test, which may be of extra interest for object-oriented systems, is placed among those tests based on the requirement specification if this is mentioned in the requirements specification, otherwise it is placed under the operation tests. In the same way we place the overload test under those tests based on the requirements specification or under the negative test

Each test type is now divided into subtests with different conditions. We can describe these subtests according to Figure 12.8

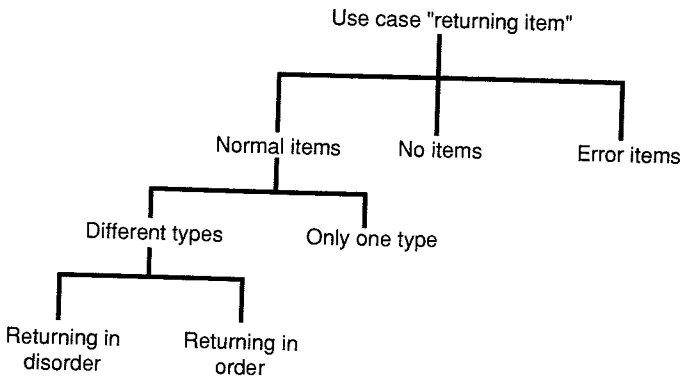


Figure 12.8 Each subtest is divided according to different conditions.

and consequently decompose the test hierarchically. Figure 12.8 illustrates an example of how equivalence classes are used to do this decomposition for the use case *Returning Item* in the recycling system. Each node indicates some condition. The tests that are specified and performed are those placed in the leaves of this tree.

12.4.4 Test specification

When we have identified which subtests are to be made, they are specified on a functional level where we describe the test in survey form and its purpose, and also on a detailed level where we describe exactly how the test is to be made. The latter part includes a complete procedure description of each step in the test. Here again the use cases are a very powerful tool. The purpose of this description is to give a detailed run instruction so that persons not familiar with the application or even the system, can execute the test.

The test specification also contains the test conditions that apply; test beds, system software, hardware, test equipment, and versions of them. The specification also includes the criteria for an approved test.

As previously discussed, each test should aim at detecting a failure. If we have a view that we should detect all faults present in the system, we may aim at trying to minimize the number of faults per 1000 lines of code or something like that. This view implies that we may have defect-free software, that is, that all faults may be found. However, all experience shows that we are far from this situation. What is actually more interesting is to view the system as having faults, and instead focus on how many hours of operation is needed to detect new failures. Thus, at least operation tests should also be performed with this statistical view in mind, to measure metrics like mean time between failures (MTBF).

The basis of the specification emanates from the interaction diagrams. In them we can see all the stimuli that are sent between the user and the system and between the objects in the system. If the testing is done manually this is often enough, and if it is performed automatically then the diagram is the basis of the test data (and to a certain extent the test program)

The test specification is used for the planning and execution of the tests. From the specification we can identify the need for equipment together with the test programs and test data that must be prepared. What we prepare and what we do not prepare is written down, with reasons, in the log

When test specifications are written, you penetrate the system during operation Then weaknesses may be discovered in the design, and these should of course be forwarded to the designers. The earlier a design error can be detected, the cheaper it is to correct it.

The test specification contains the conditions for the test and which tests are to be made in which order. For each subtest a detailed description is made of how the test shall be executed. We specify also the expected result and the criteria for an approved test.

When you write the test specification you also prepare the test reports required to report the tests. The reports are used for notes when the test is being made. Since the report skeletons are prepared prior to the testing, they control the testing instead of the test outcome controlling how the reports are written

12.4.5 Test execution

When performing tests we use the test specification and the prepared test reports. When the test bed has been installed we start the tests of the use cases, which are tested one by one As soon as some use cases have been approved, the use case tests of the system can be started The strategy is to test as much as possible in parallel, even though this may be difficult.

The testing is done by executing the automatic tests and by doing manual tests according to the directions. The test specification indicates the expected result If some subtest should fail, the subtest is interrupted and the result is noted, the defect is analyzed and corrected if possible. Then the subtest is performed again.

By means of a **decision table**, see Table 12.2, we can get an assessment of the result of the test. The table includes all the subtests,

Table 12.2 An example of a decision table

Test nr	Importance	Outcome	Evaluation
1	5	1	5
2	4	1	4
3	2	0	0
..	..		..
n	5	0	5

Σ 53 > 50 ⇒ Test is OK

weighted according to their importance, and we can determine whether the test is approved or not by its outcome. The *importance* denotes the weight of the test The *outcome* is 1 for OK and 0 for fail. *Evaluation* is calculated from *Importance*Outcome*. The evaluation is summed up and compared to a *Limit* (in this case 50). If the sum exceeds this value, the test has approved the test object, otherwise not. This table should of course be prepared in good time so that the result will not influence the weighting The table shows a survey of the total test result, both for the use cases and for the system block The evaluation of the entire test depends upon which tests are approved and their weight.

When the use case tests have been made, we analyze the result. If the use cases are approved, we can continue and test the entire system. In the same way we analyze this result. Is it approved or not? These analyses result in test reports

A test report contains a summary of the test and is also the final report from the use case and (sub)system tests. It consists of a summary and the result of the individual subtests The summary should be brief and must contain conclusions; the spent resources, and whether the test is approved or rejected. The result of each subtest is also shown in this report with the result, resources spent and the action taken, if any. Any bottlenecks that have been discovered are also written down and shown in the log.

12.4.6 Error analysis

If faults are detected when the test is made then the test must be analyzed and the reason for the fault identified. The fault need not be due to the system, but may have other causes:

- Has the test been performed correctly?

- Is there a fault in the test data or the test program?
- Is the failure caused by the test bed?
- Does the underlying system software behave properly?

If the failure was not due to the system it should be corrected and the test done again If it was caused by the system, the basic principle is that deficient blocks should be returned to the designers. However, there can be some fault finding in the test activities, for example to identify the defect block. This fault finding is helped if the blocks offer built-in facilities for fault finding, for example fault counters for stimuli or a fault log.

12.4.7 Test completion

When all the testing has been completed, the equipment and the test bed should be restored so that they can be used again for later testing. All the documentation prepared for the test should also be saved; the test documentation is just as natural to save as the source code and all other documentation.

The experiences of the testing are collected and discussed in order to learn for future test activities. Since experience is always highly coveted and always costs so much, it is important to make use of it. Concluding notes are made in the log which is also filed.

12.5 Summary

Testing is an activity to verify that a correct system is being built. Testing is traditionally an expensive activity due to the fact that many faults are not detected until late in the development. A qualitative and well-organized approach to system development is necessary in order to increase the quality of the system and to decrease testing costs. To do effective testing we must have as a goal that every test should detect a fault. Therefore we must have a disciplined approach to testing. There are several different testing types and testing techniques of which some have been discussed in this chapter.

Unit testing is performed to test a specific unit where a unit can be of varying size from a class up to a specific service package. The unit is initially tested structurally This means that we use our knowledge of the inside of the unit to test the unit. We then have different coverage criteria for the test whereby the least ambition is to cover all statements. However, these coverage criteria may be hard to define since many branches are made implicit in an object-oriented system thanks to polymorphism. The polymorphism also emphasizes the independence of each object, making them easier to test as stand alone units. The use of inheritance also makes the testing harder, since we may need to retest operations at different levels in the inheritance hierarchy. On the other hand, since we normally have less code, we do not need to test as much code. We do specification testing of a unit primarily from the object protocol. Here we use equivalence partitioning to find appropriate test cases.

Integration testing starts early in OOSE. Here we integrate instances of different classes continously through-out the development. Testing activities is performed throughout the development. The test planning should be done early and also the identification and specification of the tests.

Part III
Applications

1

1

1

13 Case study: warehouse management system

13.1 Introduction to the examples

In this chapter we will give an example of using OOSE in a larger system. Our goal is to illustrate the usage of the concepts we have discussed earlier. Of course, the entire process can not be fully illustrated within these few pages (the documentation alone would cover more pages than this entire book) We will focus on certain especially interesting things and we will go fairly deeply into some details and skip others, rather than give a complete overview.

The discussion will be more detailed in the earlier phases and less complete in the later phases. This is due to the fact that the complexity increases fast during construction, and thus needs more explanation in some areas. The implementation environment in particular needs much discussion that is unique for every development. However, we hope to give a feeling for the usage of OOSE in large and complex systems. Our coverage of examples will initially give a brief introduction to the problem domain and the requirements of the system, and then we will discuss the different activities applied in this domain.

13.2 ACME Warehouse Management Inc.

The system will support warehouse management. The company ordering the system, ACME Warehouse Management Inc., is specialized in supporting its customers with warehouse spaces all over the nation. Examples of customers are companies that need space to store their products to be shipped or companies that need local warehouses without having local offices. ACME is already a specialist in storing different kinds of items and in the use of trucks to redistribute the items. ACME plans to grow and now needs an automatic information system which they are able to grow with. The idea is to offer the

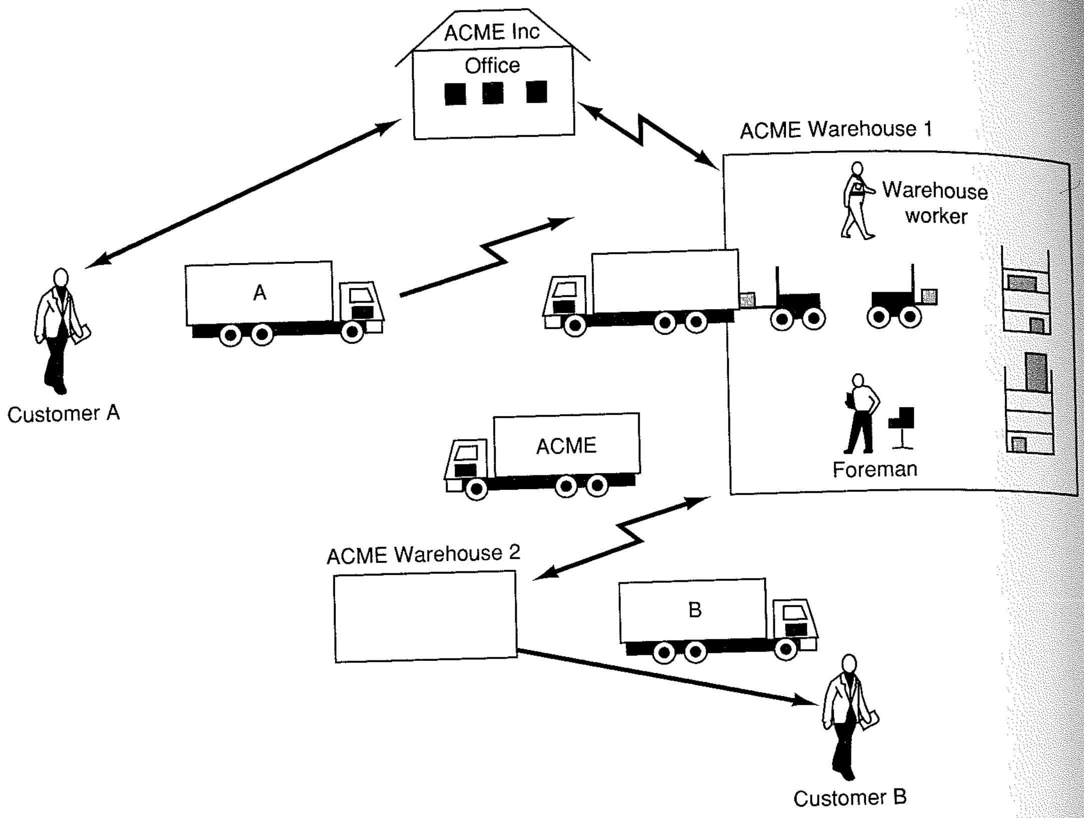


Figure 13.1 Overview of the ACME Warehouse Management System. The system consists of several different warehouses where the customers can store their items

customers warehouse space and redistribution services between different warehouses with full computer support. The service includes redistribution both within a warehouse and between warehouses, all dictated by customer needs. All kinds of items may be stored in the warehouses, which means that it is important to differentiate between certain kinds of items, for example some items must not come in contact with other items (such as industrial chemicals and foods). In Figure 13.1 the idea is illustrated schematically.

The following people will be using the system in some way or another

- Foreman, responsible for one warehouse,
- Warehouse worker, works in a warehouse, loading and unloading,
- Truck driver, drives a truck between different warehouses,

- Forklift operator, drives a forklift in one warehouse,
- Office personnel, receive orders and requests from customers,
- Customers, own the items in the warehouses and give instructions as to where and when they want the items.

It is fundamental for the ACME system that the system should be as decentralized as possible and that all persons involved should be reachable at all times. Therefore the truck drivers should have communication devices by which they get their orders and they must be able to communicate with the foreman or the office. This means that we also need a radio communication network, a system that should not be developed by us but is bought separately. Further, the warehouse workers loading and unloading should use a barcode reader when handling the items in order to be as efficient as possible. This means that all items must be marked when inserted in the warehouse system by some warehouse worker, this marking must at the same time give information of the item to the information system. The foremen should be able to work with several items at the same time, so they will probably need some window-based terminal. They are responsible for effecting the redistribution orders from the office.

When a customer wants to do something with his items, he will contact the office, which in turn submits redistribution orders to the system. Further, if the system is to work well, ACME plans to give their customers terminals so that they can interact directly with the system.

The system should use a relational database (since ACME has all its information in relational databases already and does not want to change) and the application should be coded in C++. Further the system should have a distributed implementation.

13.3 The requirement model

The first model developed is the requirement model. This model is to gain a better understanding of the system and analyze the requirements on it. In Section 13.2 we described the ACME warehouse management system. Although requirements specifications are usually more thorough than that description, they are very seldom complete and consistent. Generally, it is very hard to identify directly the right objects from the initial requirements specification. Usually, a far better understanding of the system is needed before objects can be identified to get a robust system that is maintainable.

The actors may be identified as roles played by people (or other systems) interacting with our system. An often appropriate

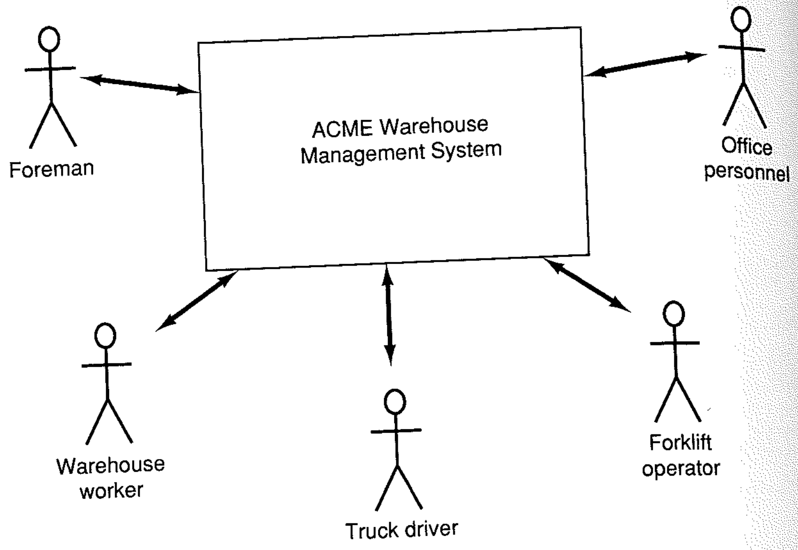


Figure 13.2 The initial system with interacting actors.

strategy is to identify organizational roles in the system domain. We know that we have foremen, employees (warehouse workers, truck drivers and forklift operators) and office personnel. In the future we may also have customers as actors. Our picture of the system will thus initially look like that in Figure 13.2.

Thus we will have the following actors interacting with the system

- Foreman,
- Warehouse worker,
- Truck driver,
- Forklift operator
- Office Personel.

To start identifying the use cases, and especially when specifying them later, it is often appropriate to have a first picture of the system in terms of some problem domain objects. This picture should only function as a support for identifying and specifying the use cases. If too much detail is put into this sketch, it can later be hard to liberate ourselves from this model. When looking at the description of the system to be built with the accompanied picture of it, we can directly find the problem domain objects.

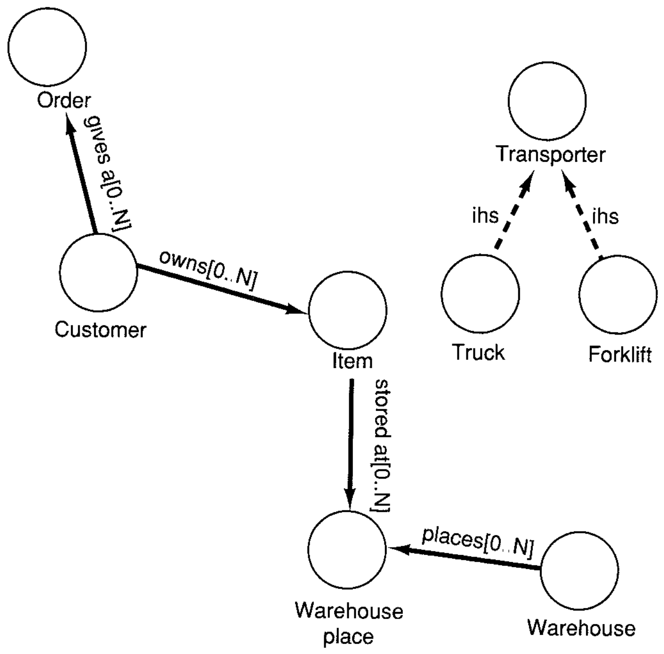


Figure 13.3 The first intuitive picture of the objects in the system

The system should manage items stored in certain places in the warehouses. Every item has an owner who is a customer of ACME. The transporters can be either a truck or a forklift. They are transporting some items from a customer order and they should all be reachable over the radio. The initial picture of the internal view of the system may thus be as shown in Figure 13.3. Here we have chosen not to devise an exhaustive object model with behavior and all, but rather a support to understand and formulate the problem.

To identify the use cases, we look at each actor and investigate what that actor wants to do to the system. We will here only look at some of the primary use cases (use cases that support the main functionality of the system). In this system there will also be secondary use cases (use cases that support the primary use cases). When investigating these we will probably find at least one new actor, a system manager, that will administrate the system as such.

We will from here concentrate on the actors Foreman and Warehouse worker.

A Foreman has to be able to move items between warehouses with or without a customer order. We call this use case 'Manual

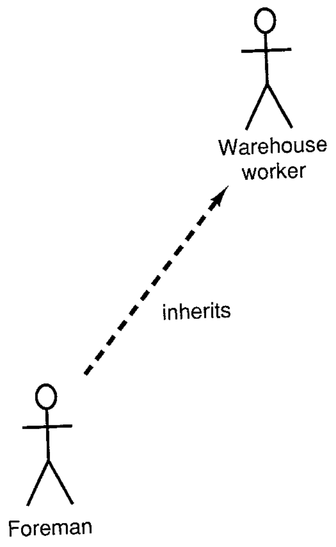


Figure 13.4 Actor *Foreman* inherits actor *Warehouse worker*.

redistribution between warehouses'. He also wants to check how far a customer order has been handled. This use case we may call 'Check status of customer order'. Other use cases that might be interesting are:

- Manual redistribution within a warehouse,
- Insertion of a new item into a warehouse,
- Check status of a truck driver,
- Check items in a warehouse,

The actor Warehouse worker is concerned with performing the following use cases:

- Customer ordered withdrawal of items,
- Check items in a warehouse,

among others. We see that some use cases can be performed by both actors. This is an indication of a separate role that is common to both actors. Are they in fact identical? No, they are not. We want the Foreman to have more privileges than a Warehouse worker, thus he should be able to do more. To avoid specifying the same use case twice, as the above example indicates, we might identify a common abstract actor that both Foreman and Warehouse worker inherit. When thinking further on this abstract actor, we realize that it will actually be identical to the Warehouse worker; everything that the Warehouse

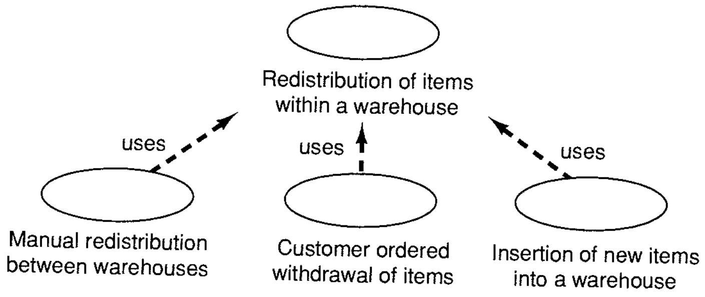

Redistribution of items within a warehouse
uses
uses
uses
Manual redistribution between warehouses
Customer ordered withdrawal of items
Insertion of new items into a warehouse

Figure 13.5 An abstract use case of redistribution of items within a warehouse have been identified.

worker should be able to do the Foreman should also be able to do. Thus we might instead let the Foreman inherit the Warehouse worker, see Figure 13.4. This is actually a technique used when modeling different levels of privileges for the actors

Before concentrating on some of the use cases, we might do a review of the use cases mentioned. We see that many of them will have a sequence when moving an item inside a warehouse. This sequence will show up when moving items between warehouses (to move the item from the loading platform to a place in the warehouse), when carrying out a customer withdrawal (to move the item from the warehouse to the loading platform) and for instance when performing a manual redistribution within a warehouse. We here might identify an abstract use case that redistributes an item within a warehouse that is used by use cases that might need this sequence as a sub-sequence, refer to Figure 13 5

We will from now on focus on two use cases for a more detailed discussion:

- Manual redistribution between warehouses,
- Customer ordered withdrawal from warehouse

The first use case is performed by the Foreman when he wants to redistribute an item in one warehouse to a place in another warehouse. The second is performed when a customer wants to withdraw some items from the warehouse

13.3.1 Manual redistribution between warehouses

A common question when identifying use cases is the extent of a use case. A use case should be a logical cohesive sequence of events: but what does this mean? If we analyze what will happen when a

Foreman performs a manual redistribution of items between warehouses, we see that it has at least four phases:

(1) Initialization, when the Foreman gives the request to do the redistribution,

(2) Planning, when the system does planning to coordinate the transports and issues transport requests,

(3) Loading, when a truck fetches the items from the warehouse,

(4) Unloading, when a truck delivers the items to the new warehouse.

All four of these phases may very well be seen as use cases of their own, possibly excluding the 'Planning' part, just as they may be viewed as parts of one complete use case. This question is thus quite common when identifying use cases and there is no general best answer. The arguments in favor of having the sequence as one complete use case are:

- When specifying the use case, we may follow a complete flow through the entire system,
- From the orderer's view it is a logical cohesive flow of events in the system,
- It may be more effective when testing the use case since it covers more logical cohesive events in the system and failures can be found more easily,
- It is easier to synchronize the use case since it is one sequence that starts different events in chronological order.

The arguments in favor of separating the use case into several different use cases are:

- It may be troublesome to find the right instance of a use case that of large extent since the use case may very well last for several days (as in the current case),
- From a potential actor's view it is more logical to have use cases that the actor starts,
- It may be easier to test the use case since every use case starts from external events and not by internal system events.

In this case we choose to keep the use case as one coherent use case, principally because it is more logical when specifying the use case. The use case starts when a foreman issues a redistribution request and ends when the transportation is done and the foreman has been notified. The use case also involves the coordination of the transportation resources so as to use them as effectively as possible.

When specifying a use case which should be performed in a window management system, it is often appropriate to do sketches, or even better prototypes, of the view the users will have. Our experience shows that to do effective user interface prototyping you need first to have identified the major use cases and also to have developed a first intuitive picture of the objects in the system. This is because you need first to have thought about the system a while, what information it should hold (the intuitive objects) and how it should be used (the major use cases), before you can do effective user interface prototyping. Starting with the user interfaces may end with having the prototypes developed determining what the system should do instead of the other way around.

We will here use interface descriptions as seen by the users when specifying the use case. In the use case description we have numbered each line to refer to it in the text later. With a tool support this can be appropriate. We have also used headlines to indicate the different phases in the use case; a method often appropriate also in real cases.

Initialization

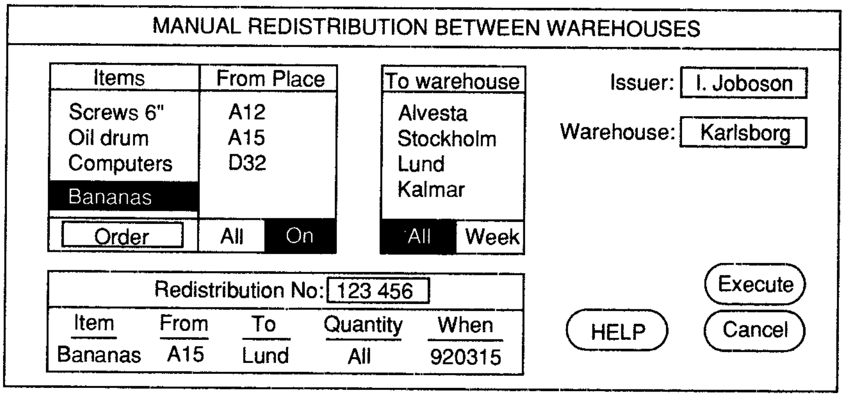
(Words in italic refer to objects in the initial problem domain)

(1) The Foreman gives a command for redistribution between warehouses

(2) The window in Figure 13.6 is presented to the foreman.

(3) The *items* can be ordered in a number of ways This is selected with the ORDER menu The following orders are possible:

(*a*) alphabetical order


MANUAL REDISTRIBUTION BETWEEN WAREHOUSES

Items	From Place
Screws 6"	A12
Oil drum	A15
Computers	D32
Bananas	
Order	All On

To warehouse
Alvesta
Stockholm
Lund
Kalmar
All Week

Issuer: I. Joboson

Warehouse: Karlsborg

Redistribution No: 123 456

Item	From	To	Quantity	When
Bananas	A15	Lund	All	920315

Figure 13.6 Manual distribution window.

(*b*) index order (each item has a unique number)

(*c*) turnover of the items

(*d*) storing order.

(4) In the 'From place' table we might choose to view either all *places* in the current *warehouse* or, if we have selected an *item*, the *places* where that *item* exists.

(5) In the 'To warehouse' table we might select all *warehouses* or the *warehouses* that we have a transport to this week.

(6) The 'Issuer' and 'Warehouse' fields are automatically filled when the window pops up, but they might be changed (This is the way to do a redistribution from another *warehouse* to our own *warehouse*.)

(7) The foreman selects an *item* by pointing to it and dragging it to the Redistribution form. He then selects from which *place* to take the *items* and to which *warehouse* to transport them This information is automatically shown in the form

(8) The foreman then gives the quantity to be moved, a possibility is to give 'ALL', and then the foreman gives a date by when it must be done

(9) It is possible to change the information when the form has been edited When the foreman EXECUTES the redistribution, the transport is being planned It is also possible to CANCEL the redistribution as such When selecting HELP a window is shown that gives information about the current window

Planning

(1) When the redistribution is executed the *items* to be moved are marked as move-pending

(2) The planning should minimize the use of *trucks* with the condition that all delivery dates should be held and the *trucks* should be compatible with any delivery requirements for the *items* (e g. in size). The minimization should be done by adjusting the new redistribution requirements with existing, already planned redistributions

(3) This may render new transport requests and may also change existing transport requests already in the system.

(4) The transport requests are connected to a specific *truck's* transportation plan

Loading

(1) A *truck* driver asks for a transportation request. The request is marked as ongoing. He or she also gives the time when he or she will be at the warehouse

(2) Give appropriate request to the *forklift* operators to have the items ready when and where the *truck* is expected

(3) When a warehouse worker gets a request to fetch *items* he or she, at the appropriate time, orders *forklift* operators to move the *items* to the loading platform

(4) When the *truck* driver arrives the *items* are loaded. The *truck* driver tells the system when the *truck* is loaded and when it is expected to be at the new *warehouse*

(5) Decrease the number of *items* in this *warehouse* Mark the transport request as on transport.

Unloading

(1) When the *truck* has arrived at the new *warehouse*, the *items* are unloaded.

(2) The *truck* driver tells the system that the transport to this *warehouse* has been done

(3) The warehouse workers receive the *items* and determine a place for them in the *warehouse*.

(4) *Forklift* operators are told to move the *items* to the new *place* in the new *warehouse*.

(5) When the *truck* driver confirms the insertion, the system updates the new *place* for the *items*

(6) The transportation time is recorded and stored in the system

(7) The Redistribution and the transport request are marked as performed (It is deleted by a foreman or the system manager later)

Alternative courses

A request is not executable

The execution is interrupted and the foreman issuing the request is informed

Redistribution is wrong

When the redistribution is filled and executed by the foreman, the appropriateness is checked immediately and returned to the foreman Possible errors may be:

(*a*) The *warehouse place* does not have enough *items* to move,
(*b*) The *warehouse* to move the *items* to is not appropriate to the *item* (e g. because of size or storing circumstances).

No truck available

When performing the planning, there may not be any *trucks* available at an appropriate time. Then notify the foreman who should either delete the request or change it

The use case has some alternative courses that may change the flow of the use case. As many as possible of these alternative courses, often erroneous courses, should be noted. Although many of them will not be noticed or even identified, the more of these alternative courses that are noted in advance, the more robust the system will be. It is valuable to have these courses noticed at this early stage since they may be considered in the following work of the system.

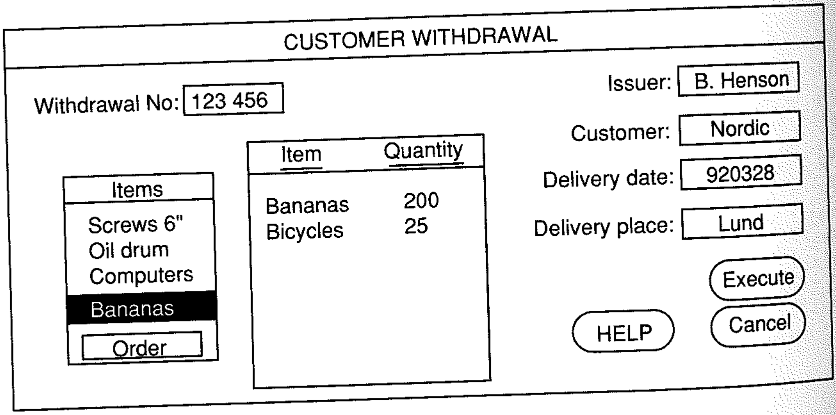
Before discussing use cases more generally we will first look at the other use case, 'Customer withdrawal from warehouse', which is smaller than the first one.

13.3.2 Customer withdrawal from warehouse

This use case covers the case when a warehouse worker does a withdrawal at the request of a customer. The use case starts when the customer request is inserted into the system by the office personnel. It lasts until the customer has signed out his items from the warehouse.

(1) The office personnel insert a request for a customer withdrawal at a certain date and a certain warehouse The window in Figure 13.7 is shown to the office personnel when giving the command issuing a customer withdrawal.

(2) The information filled by the office personnel is Customer, delivery date and delivery place The name of the office personnel person is



Withdrawal No: 123 456

Item	Quantity
Bananas	200
Bicycles	25

Issuer: B. Henson
Customer: Nordic
Delivery date: 920328
Delivery place: Lund

Figure 13.7 Customer withdrawal window

filled from the beginning but it is possible to change it The office personnel person selects items from the browser and adds the quantity to be withdrawn from the warehouses The browser can here show only the items of the current customer

(3) The following criteria are checked instantly:

(*a*) The customer is registered,

(*b*) The number of items ordered exist in any warehouse

(*c*) The customer has the right to withdraw the items

(4) The system initiates a plan to have the items at the appropriate warehouse at the given date. If necessary transport requests are issued The items are reserved three days in advance (The longest possible time for a warehouse redistribution has been measured to take three days.)

(5) At the date of delivery a warehouse worker is notified of the withdrawal

(6) The warehouse worker issues requests to the forklift operators to get the items to the loading platform The forklift operator executes the transportation

(7) When the customer has fetched his items the warehouse workers marks the withdrawal as ready The items are removed (decreased) from the system

Alternative courses

There are not enough items in the warehouses

The office personnel are notified and the withdrawal cannot be executed

Customer has no right to withdraw an item or Customer is not registered

Notify the office personnel The withdrawal cannot be executed.

When writing use cases, an active form should be used from the system perspective. When mentioning actors in the use case the formal names of the actors should be used This means that when changing the formal name of an actor, it should be changed everywhere the name is mentioned.

It is the basic course that should form the basis when specifying a use case and this should be described first. The basic course is the course which gives the best understanding of the system, not necessarily the most often executed course. The reason for this is that the use case aims at bringing understanding of the system.

Alternative courses are described separately. These are often written under a special headline '*Alternative courses*' Here references are made to the basic course where the alternative course may happen. The reason for this is to build as much as possible on what has already been specified and to avoid redundancy in the descriptions. An example of this is in the Customer withdrawal use case when the system notices that there are not enough items in the warehouses.

When the requirement model has been specified, it is possible to check the model against the orderer or the end users. We see that actually there is no need to understanding system development in order to understand the use cases. We have often seen in real projects that use cases are an appropriate basis for discussion when analyzing the system for requirements. Thus the requirement model as a whole may therefore be a part in the contract between the orderer of the system and the deliverer – the use cases are then the commitment made by the deliverer for the system.

13.3.3 Abstract use cases

We noted earlier that several use cases have one thing in common, namely that they should move an item from one place to another inside a warehouse. This is also seen in the two use case descriptions above. In Manual redistribution between warehouses we see it in Loading step 3 and Unloading step 4, and in the Customer withdrawal use case we see it in step 6. This sequence may be described in an abstract use case 'Redistribution of items within a warehouse' which covers the sequence relevant when a forklift operator receives an order to move an item from one warehouse place to another (including the loading platforms).

In the use case descriptions, we see that this step is not formulated in exactly the same way. This is very common since often several people are involved in the use case descriptions and it often takes some time to write the use case descriptions. Therefore it is essential that someone with a very good overview of the system performs an analysis of common parts in the use cases. The reason to separate this part out to form an abstract use case is so that we need only design this part once and this part can then be reused in several use cases.

Experience shows that you should wait to identify abstract use cases until you have described several concrete use cases. The reason for this is that the abstract use cases should evolve fom the concrete ones and not the other way around.

13.4 The analysis model

When the requirement model has been developed, a larger milestone has been reached The requirement model has been checked by external reviewers, maybe the future users, and is now approved. Our aim is now to develop the first preliminary structure of the system. This should be a logical structure that is stable over the system life cycle with no consideration yet to the actual implementation environment.

The use cases will now be broken down into the analysis model and we will thus identify the objects that offer the use cases. When we have done this structuring of the system, we will identify the subsystems that we may divide the system into. Hence firstly the interface objects, entity objects and control objects will be identified These objects are normally identified in an iterative manner over a use case, and then over all use cases offered by them

13.4.1 Analysis objects

When identifying the **interface objects** we focus on where actors interact with the use case (i.e. the system). Everything in the use cases that implies some interaction with the system should be offered by an interface object.

In our example we do not have any examples of communication with other information systems. This version, at least, will be a stand-alone system that does not interact with any other system This means that all interface objects will be interfaces to humans. These interfaces will be of three kinds: firstly, it is the window systems whereby the actors interact by a user interface based on a window management system; secondly, hardware devices like push buttons and barcode readers used by the warehouse workers; and thirdly, the radio communication network Let us discuss the last one first.

The radio communication network is bought from another vendor. The network will thus be *outside* our system. This means that we may view it as an existing product that will be connected to our system. Therefore we should not describe the network in detail. What we need to model though is an interface between the network management system and our system. Hence, in the analysis model we will model the interface to the network, in only one interface object. If we were to develop the network also, this would not be enough. Then we should analyse the entire network too. We have thus now identified the first interface objects which we may call *Truck radio* and *Forklift radio*. They are modeled by two objects since they will be different networks: inside a warehouse for the forklift operators; and outside the warehouse for the truck drivers. If we were to have the network inside our system, we would not have an interface object to it.

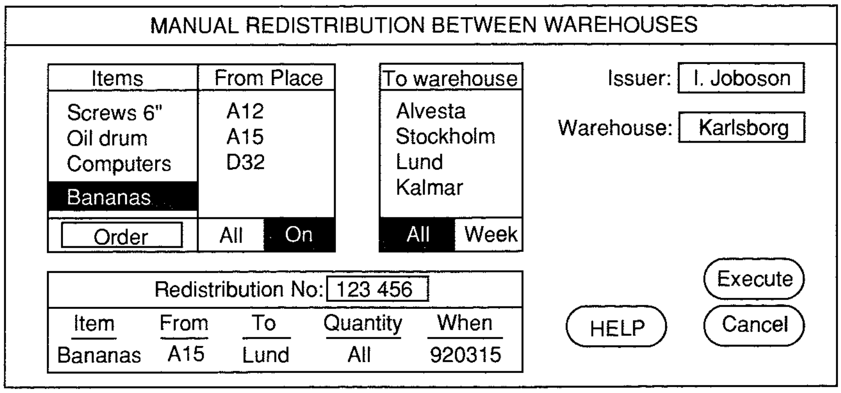
To model the hardware devices in the warehouse, we model each different device by an interface object. This encapsulates everything in the device and converts external signal to stimuli inside the system.

The window systems and terminals will also be modeled by interface objects. Three actors will have such interfaces to the systems. These are the Office personel, the Foreman and the Warehouse worker. When regarding which use cases each of them will be able to perform we have already noticed that the foreman will inherit all capabilities of the Warehouse worker. The Office personnel will perform quite different use cases than the Foreman and the Warehouse worker. User interface prototypes or sketches of the interfaces have been made during the requirement modeling. Since they are part of the requirement model which forms the contract between user and developer, many of the problems with user interface design are avoided. Users can now give their opinions on the interfaces before the actual development work starts and thus avoid much of the changing of user interfaces in late phases of the system development. This is one of the major strengths of OOSE since the development is driven from the actors' needs and the users have an important role to play during the requirement modeling when defining the tasks of the system. The sketches and/or prototypes form a rigorous base from which the interface objects are now to be modeled.

We see from the use cases that many of them start with one window which is filled with information and then executed. New windows may also show up at later stages in the use case. Hence we can use these to identify the interface objects.

The modeling of the interface objects can be done at several different levels of granularity. The usual level is to model each window that should be displayed by an interface object. When doing this we use the sketches or prototypes already developed. Refer to Figure 13.8. This was the first window to be shown to the Foreman when he wanted to perform the Manual redistribution between warehouses.

To model the structure of the window we use the widgets that are needed. How these really look in the user interface management systems (UIMS), or other class libraries which we shall use when implementing, does not show here. That will be taken care of during construction. The structural description describes only what the user interface should be composed of. We use a central interface object



MANUAL REDISTRIBUTION BETWEEN WAREHOUSES

Items	From Place
Screws 6"	A12
Oil drum	A15
Computers	D32
Bananas	
Order	All On

To warehouse
Alvesta
Stockholm
Lund
Kalmar
All Week

Issuer: I. Joboson

Warehouse: Karlsborg

Redistribution No: 123 456

Item	From	To	Quantity	When
Bananas	A15	Lund	All	920315

Figure 13.8 The window from the use case *Manual redistribution between warehouses.*

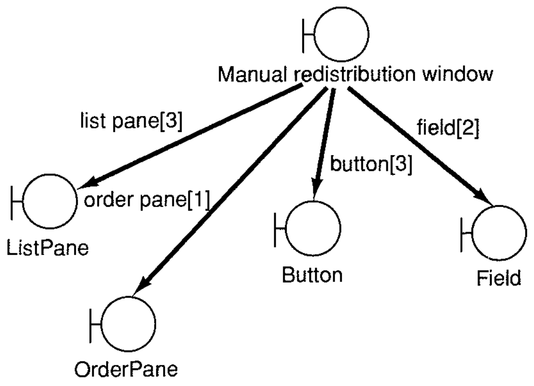


Figure 13.9 A structural model of the window for *Manual redistribution between warehouses.*

which represents the entire window and other interface objects to model the widgets. A structural view of the window is shown in Figure 13.9.

Many modern UIMSs support the modeling and creation of windows from the external view shown above. Some of them also support the adding of dynamic behavior behind the windows. In some of the projects we have been involved in, such UIMSs have been used in a successful manner, but since the above description with the interface objects composed of other interface objects has much redundancy with the external view of the UIMSs, these projects have used another technique to model interfaces. The interface objects are then lifted one step. The central interface objects then represent

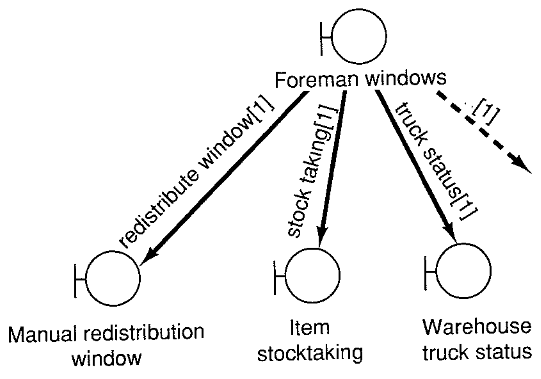


Figure 13.10 The possible windows of the foreman.

all the windows that a certain actor will be able to view, while each window is represented by an interface object beneath it. An example for the Foreman view is shown in Figure 13 10.

Here we suppose that we have a modern UIMS and thus follow the second strategy. With this technique we may thus have one central interface object corresponding to each actor using a window system. To model the fact that the Foreman should be able to do everything that the Warehouse worker is able to do, we used inheritance between both actors – the Foreman inherited the Warehouse worker. When modeling the interface objects we may also reuse all the windows defined for the Warehouse worker when defining the foreman's interface object. This may be expressed with inheritance between the interface object descriptions as showed in Figure 13.11.

To summarize, we will thus have the following central interface objects:

- *Warehouse worker windows*
- *Foreman windows*
- *Office personnel windows*
- *Forklift radio*
- *Truck radio*
- *Barcode reader*

The functionality of the use cases is thus placed in these interface objects. If we later want to change any of these functionalities, such as where the foreman should pick up icons instead and drop these icons at the new location, such changes should only affect the interface objects associated with this functionality. Hence the objects 'behind' will then not be affected.

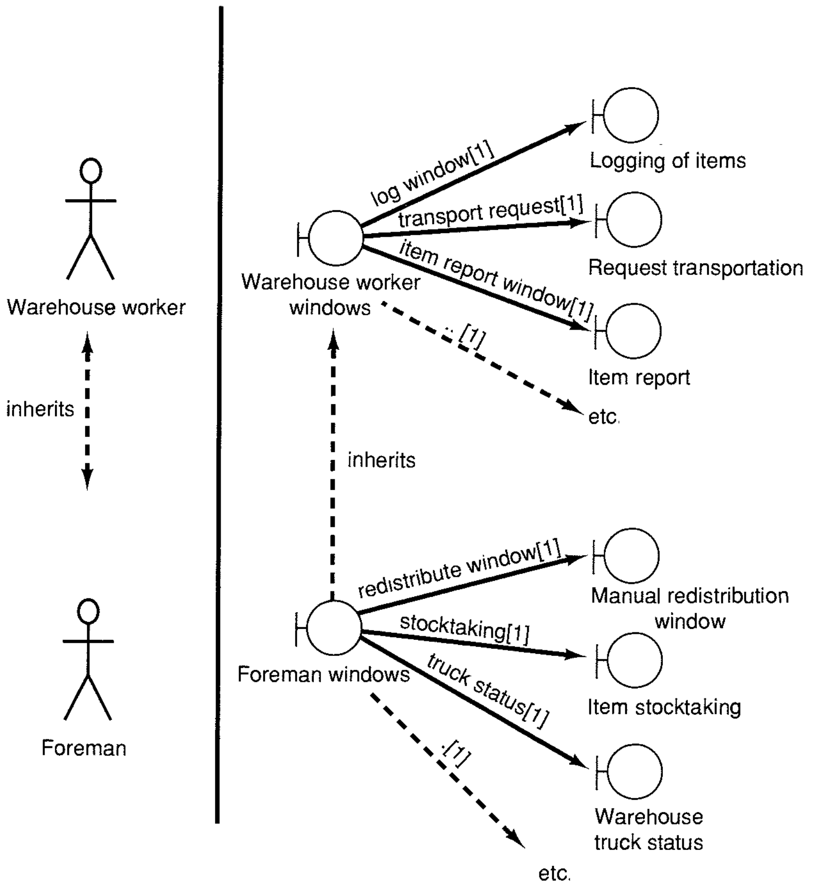


Figure 13.11 Inheritance of interface objects to reuse interface descriptions

Although the identification of the objects is normally iteratively, it is often easiest to start investigating the interface objects, since they are often evident in the use case specifications (pictures). Thereafter the **entity objects** often come naturally. We will now discuss the identification of the entity objects involved in the use cases previously discussed.

Basically we find the entity objects by reading the use case descriptions and looking for information that needs to be stored over a longer period of time. The problem domain objects are very good candidates for this, but are seldom enough. Entities are often quite easy to find and even if different people perform the identification, we have seen that the result is often very similar. We see that the system should manage information about *Customer, Items, Warehouses,* Truck and *Warehouse places* and so on. This is also confirmed when reading the use cases.

From the Manual redistribution between warehouses use case we also see that we need to store information about the new redistribution and the transport request (either already existing or a new one). A Redistribution entity object is thus identified in Initialization steps 2, 7–9, Planning steps 1–2 and in Unloading step 7. The transport request is noted in Planning steps 2–4, Loading steps 1 and 5 and Unloading step 7. In Planning step 4 a transportation plan is mentioned which should contain the transport requests planned for a certain truck. One possibility is to have this as an attribute of the Truck entity object, but since the Planning part will use all these transportation plans to coordinate the transportation we choose to have it as a separate object. Hence we need some entity object to hold this information also. This is a typical example of what we discussed in Chapter 6 about an object model. Objects must be identified in the light of how they shall be used, not as a stand alone process. In another system the Transportation plan could very well be an attribute in the Truck entity object. Now transportation plan is a complete list of all planned transportations relating Trucks with Transport Requests.

When we identify the entity object we notice that they have certain relations to each other. These relations are static in the sense that they are stored for a longer time. These static relations are modeled by acquaintance associations. Every such association has a name and a cardinality. An example of such an association is that we may say that a Redistribution should be performed by one Transport request. The entity object and static associations identified in this use case are shown in Figure 13.12. Note that all associations are one way only. If an acquaintance is needed the other way around too, a new association is needed. This is a significant, and often subtle difference compared to many kinds of data modeling techniques.

Attributes cannot be attached to associations. Therefore we must model these attributes in some other way. In this example we have such examples. Refer to the association Item–place [0 M]–Warehouse place. How many of the items have we stored at the Warehouse place? We cannot have it as an attribute in the Item entity object since Items may be stored at several different places. Neither can we have it as an attribute in the Warehouse place entity object since several different kinds of Items may be stored at the same Warehouse place (the requirements have not defined how big the warehouse places are). So where should we store the information on the number of Items in a specific place?

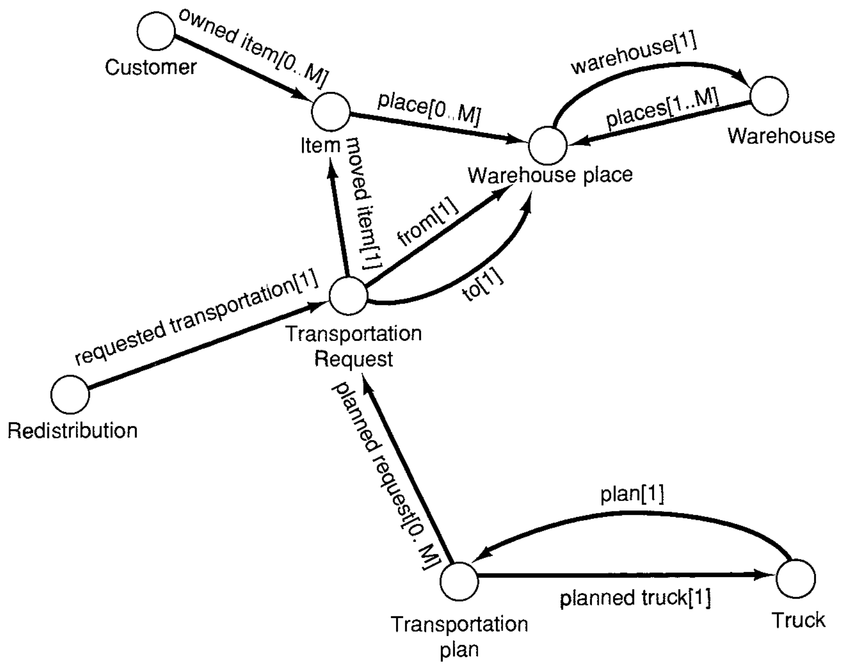


Figure 13.12 A first attempt of the entity object model for the use case *Manual redistribution* between warehouses

There are two possibilities. If we think how this should be coded later we would probably have a list to store association to the warehouse places. In this list we could add an attribute to each element to store the number of items in a specific place. If we consider the relational database and how the information should be stored, we will have one extra table for the association since it is of cardinality [1 .M] (see Chapter 10). In this relation we may add an attribute to hold the number of Items at a specific place. Hence, when later doing the design, it will be no problem to realize this attribute. The first alternative is therefore just to note down the attribute in the Item entity object description when describing the association.

If we also want to be able to check how many items are stored at a specific Warehouse place, this numerical information needs to be accessible from the Warehouse place entity object. The above solution would then not be appropriate, since we would need to store the number information also in the Warehouse place entity object. Then we would have redundant information in the system and thus have

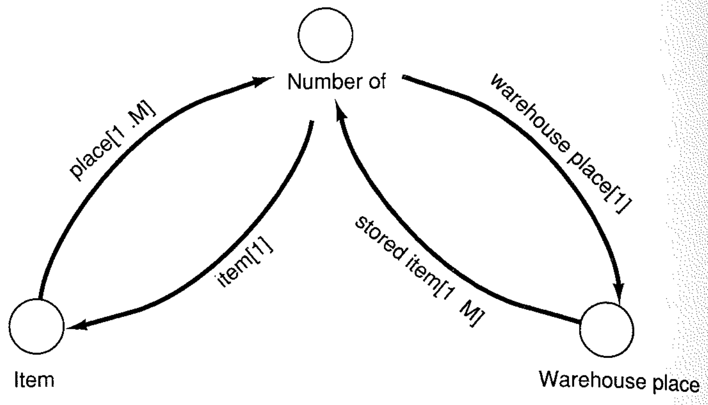


Figure 13.13 The entity object Number Of represents the attributes of the relation between Item and Warehouse place

difficulties to maintain and update the information. Therefore we should only store the information in one place and we then have to add a new entity object Number of to store this information, which the other two refer to as shown in Figure 13.13.

We choose to use the latter technique since we would probably want to know how many Items are stored at a specific Warehouse place in another use case. The problem of connecting information to associations also occurs in another place in the model. Between Tranportation request and Item we need to store how many items this request covers. But since we only have a cardinality of [1] and we do not need the inverse relation, there is no need to add an extra entity object in this case. Instead we note that this information should be stored at Transportation request. In this example there are no more cases of this problem. Thus the final entity object model will look like Figure 13.14.

The entity object are thus:

- Item The entity object holds information on its identity (name), any special storing or transport requirements and the total number in all warehouses (this may be calculated). It also has acquaintance with the owner (Customer) and where it is stored and how many are stored at that place.
- Number of. This holds information on how many are stored at a specific Warehouse place.
- Warehouse place. This holds information on the name of the place and any special characteristics of it and knows also which warehouse it belongs to

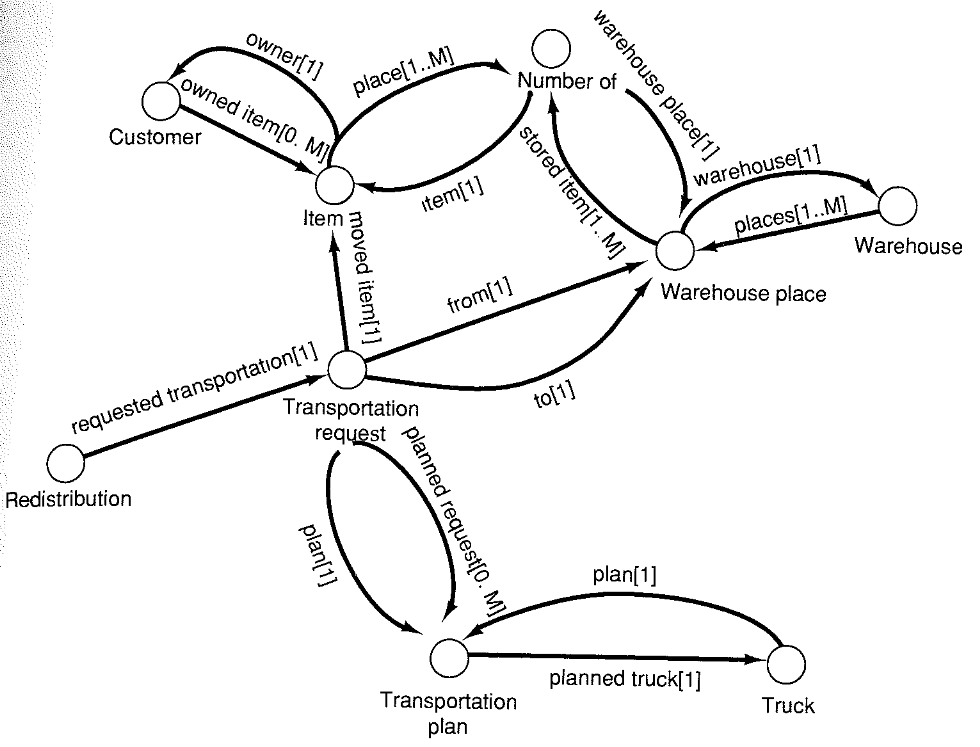


Figure 13.14 The final static entity object model covering the current use cases

- Warehouse This holds information on the identification (name) of the warehouse and its address and knows its associated warehouse places.
- Customer. The address and the customer number are attributes, it associates the items that the customer owns.
- Transportation request. This keeps information on the number of items to be moved, delivery dates and the status of the transport and associates what item to move with a source and destination for it. Concers only one Item.
- Transportation plan This is a list of transportation requests that are planned for a certain Truck.
- Truck. This has information on the identification of the truck, its radio address, the driver of the truck, its maximum load and any other special characteristics of the truck, and associates its transportation plan.

- Redistribution. This handles an order of redistribution with the issuer and a redistribution log number and is handled by a Transportation request.

The description of an entity object includes its attributes, associations and possibly also identified operations identified of the entity object (i.e. the dynamic behavior of the entity object). The operations are used to express how entity object perform their task. They can also use operations on other entity object. An example is when the truck wants to know from where to fetch an item. This may be an operation on Transportation plan which asks Transportation request to get a reference to the current warehouse. This may be fetched from the Warehouse place entity object. A reference to the Warehouse may thus be returned to the Transportation plan which asks the Warehouse for its identification and then returns this to the Truck.

The acquaintance associations are used to model static references between the entity object. The dynamic relations between them are modeled by communication associations. In the previous example we see that we have a communication association from Transportation plan to the Warehouse (to get its name), but we do not need a static relation between them.

The acquaintance associations are also used to show how certain entity object play different **roles** in the model. An example of this can be seen in the associations from the Transportation request to a Warehouse place. A specific Warehouse place may act as both the source and the destination of a transportation. To use associations to indicate different roles played by some objects is very effective since it significantly decreases the number of objects in the model and thus decreases the complexity of the model.

The control object is the third and last type of analysis object types. Their main purpose is to get a maintainable system and to increase the reusability potential of the model.

The control objects are also identified from the use cases. It is mainly behavior from the use cases that we do not want to place in the entity object, so as to get a looser coupling between different parts of the system. One reason for this is that the behavior may change in a different way from the behavior associated with the entity object.

The control objects are generated in two phases. The first is a rough attempt and generates one control object for each (abstract and concrete) use case. The reason for this is that every concrete use case represents behavior that should be tied together and the abstract ones represent behavior that will be shared by different use cases.

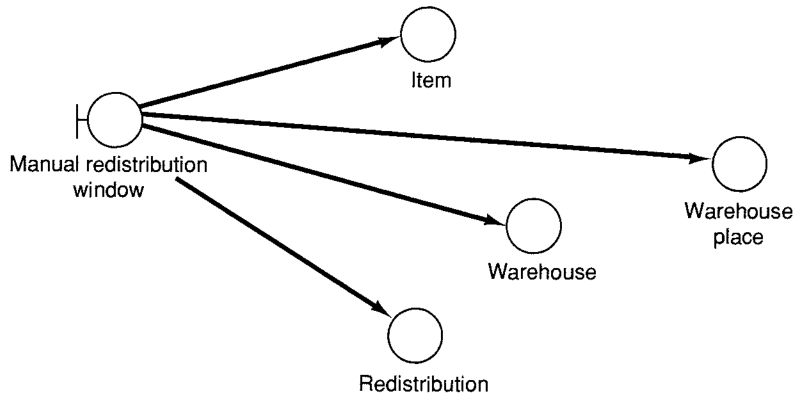


Figure 13.15 The *Initialization* part illustrated schematically.

In the first phase we name the control objects and it is often easiest to name them exactly as the names of the use cases. In our example the initial control objects will thus be:

- Manual redistribution between warehouses,
- Customer withdrawal of items,
- Redistribution of items within a warehouse.

When this first attempt has been made, which can be done mechanically, we will allocate the behavior from the use cases to the objects identified so far. This forms the second phase identification of the control objects and is much harder than the first phase. Now we see whether a control object is really needed or maybe a new one is necessary. Often trade-offs must be made between potential reusability of the model and the effort you want to put into it. In the second phase a better and more appropriate name should be given to them. This is often more noun-like

Let us look more at the use case Manual redistribution between warehouses. The Initialization part will be handled completely by the interface object which will directly access the entity object that have the information to be shown. Here a control object will not be needed since the presentation and the information have a simple correspondence. The interface object will also create a Redistribution object when the actor executes the redistribution, see Figure 13.15.

The Planning part of the use case is more complex though. Planning step 1 will be performed by the Redistribution entity object. Steps 2–4 are a coordination activity of several entity object not naturally associated with any special object in the model. One possibility is to assign it to one of the already introduced control objects, namely Manual redistribution, but it seems more like a special activity that can be used in several different use cases Therefore we introduce a new control object, Planner, that performs this planning activity.

The Loading part of the use case starts when a Truck driver gets his current Transport request (step 1). He then starts an activity to coordinate the withdrawal, transportation and insertion of items. This activity is naturally performed by the control object identified for this concrete use case. We see now that there are actually two activities going on here – one inside one warehouse and one to coordinate the entire transportation to the other warehouse. So it seems that our first attempt at the control objects will fit very well into this use case. Hence the control object Redistribution of items inside a warehouse will handle all activities inside a warehouse (i.e steps 2 and 3) This will now be a general, and reusable control object. that coordinates one transportation request inside a warehouse. This means that the first idea of having it as an abstract reusable use case was right. It will be a special part of the system that can be used in several different use cases. To have a shorter, more noun-like name of the object, we rename it to Local warehouse transporter. The other steps of this part of the use case (i e steps 1, 4 and 5) can be handled by the other control object. (Possibly the behavior of step 1 will not need any control objects. This we will see more obviously later.) Thus the control object initially called Manual redistribution will handle all activities that coordinate the transport between warehouses. This may now be renamed Interwarehouse Transporter.

The Unloading part of the use case will be manageable with these objects Steps 1,2,5,6 and 7 will be managed by Interwarehouse transporter while steps 3 and 4 can reuse the control object Local Warehouse Transporter

Thus the following control objects have been identified as participating in the use case Manual redistribution between warehouses:

- *Interwarehouse Transporter*, which handles a transportation of items between warehouses including some of the coordination of trucks and forklifts,
- *Local Warehouse Transporter*, which manages transportation inside a warehouse including some part of the coordination of trucks and forklifts,
- *Planner*, which plans and issues transport requests and inserts them in the appropriate truck's transportation plan.

We have stressed the coordination of the trucks and the forklifts. This is because two different control objects (actually three since

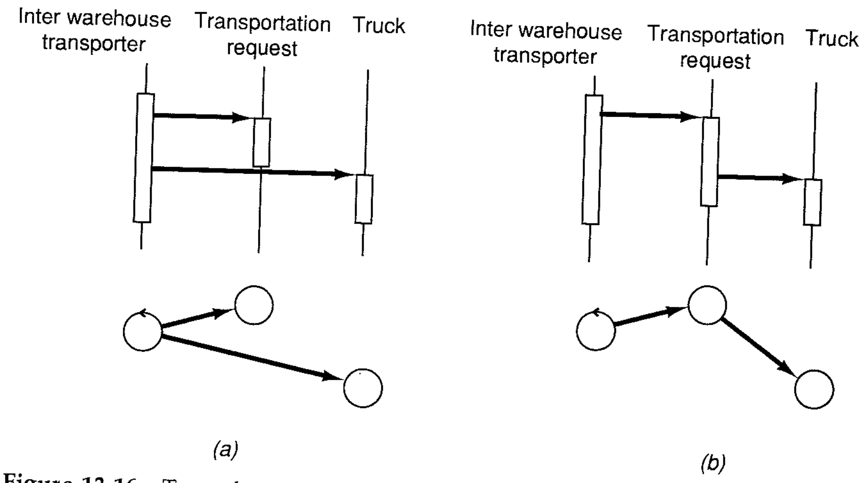


Figure 13.16 Two alternative ways of allocating the use case behavior to simple objects

Local Warehouse Transporter will be instantiated twice in every use case) must interact with each other, which is quite complicated. In the design they will probably be distributed and thus communicating over some computer network. We will focus more on this during the construction.

When **allocating behavior to the simple objects**, important decisions are made concerning how the objects will depend on each other. Refer to Figure 13.16 where we have used the interaction diagram technique to illustrate the relationships between objects. In case A the control object Interwarehouse Transporter knows the existence and the protocols of both the entity object Transportation request and Truck. The control object here accesses the Transportation request to get a reference to the right Truck. The Transportation Request knows about the existence of Truck (actually, indirectly since it uses its Transportation Plan to get the reference, see the entity object model), but not about the Truck's protocol since it doesn't access it. In case B the control object does not need to access the Truck directly, hence it is not dependent on its protocol or even its existence. When to use which approach was discussed in the analysis and construction chapters.

To illustrate schematically how the behavior of a use case is distributed over the objects, **use case views** over the objects are often drawn. In these views we include the objects and associations that participate in the specific use case. The union of all such views should become the entire model. Since, for instance, only the acquaintance associations that are used in the specific use case exist in these views, it is essential not to become confused when two

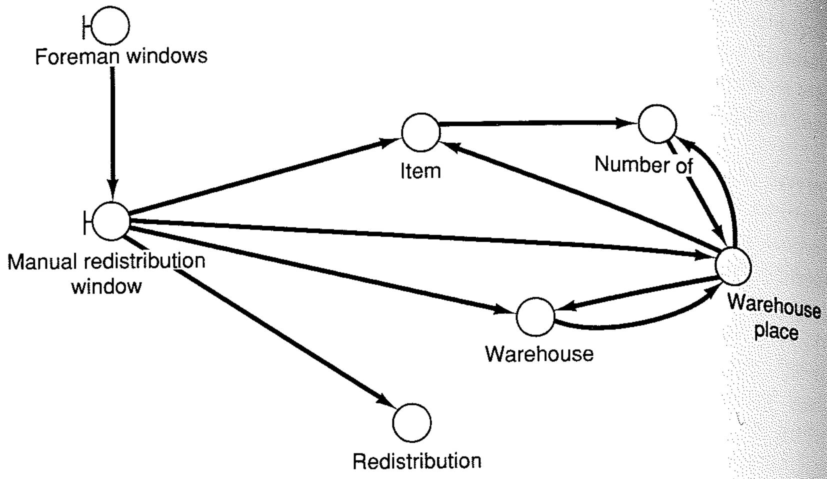


Figure 13.17 The initialization part of the Manual redistribution between warehouses use case We have used communication associations to illustrate the dynamic flow over the objects

objects that apparently should be associated are not associated in the view; they may very well be associated, but the association is not used in this particular use case.

We have drawn such use case views over objects for the use case Manual redistribution between Warehouses. Since it is a large use case, we have drawn one view for every part of the use case instead of one for the entire use case. The Initialization part of the use case is shown in Figure 13.17, Planning in Figure 13.18, Loading in

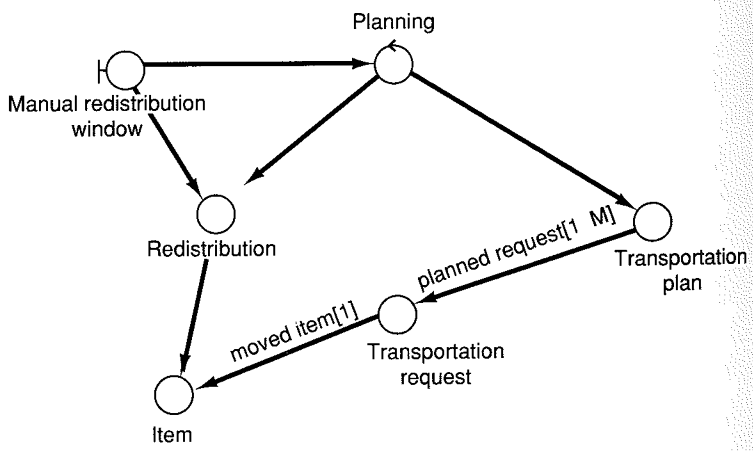


Figure 13.18 The Planning part of the use case.

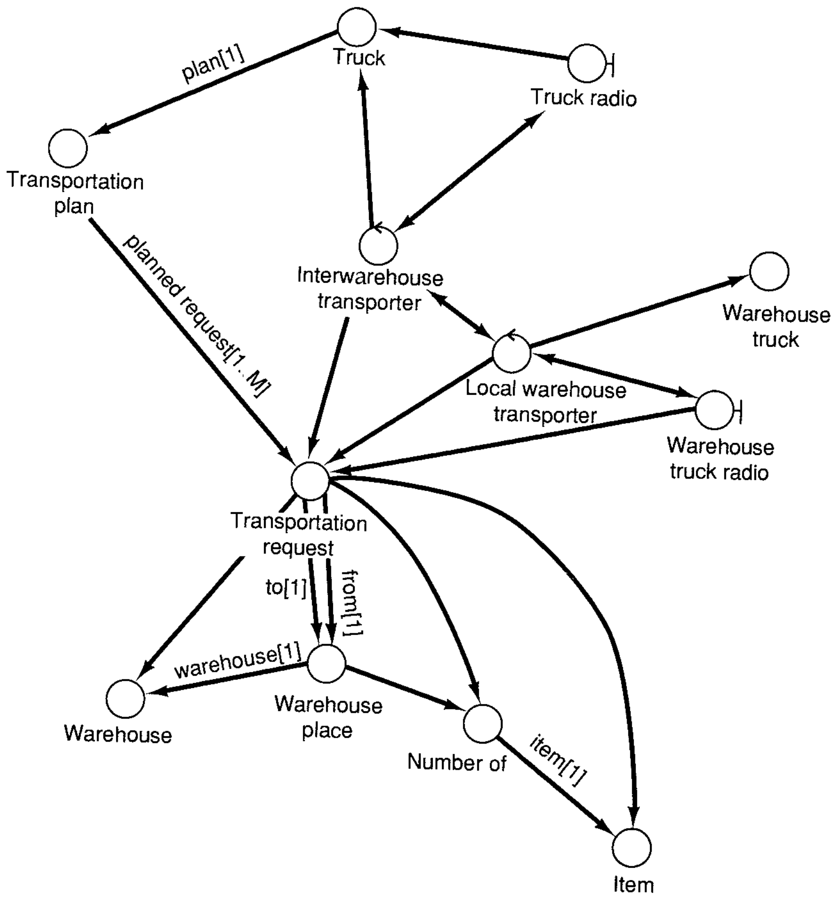


Figure 13.19 The Loading part of the use case.

Figure 13.19 and Unloading in Figure 13.20. In some views only communications associations are shown.

We will describe the Loading part in more detail to explain how we have come to this picture. The loading part starts when a truck driver asks for a Transportation request. The Interwarehouse Transporter asks the Truck for its next Transportation request which is found via the Transportation plan. The request is marked as ongoing. It also returns the Warehouse name The time that the truck driver estimates he will be at the new place is attached to the Transportation request. The Interwarehouse Transporter then initiates the Local Warehouse Transporter which finds an unoccupied Forklift. This truck also gets information on what items to move, from where and also how many. When all the prerequisites are ready, the commands are

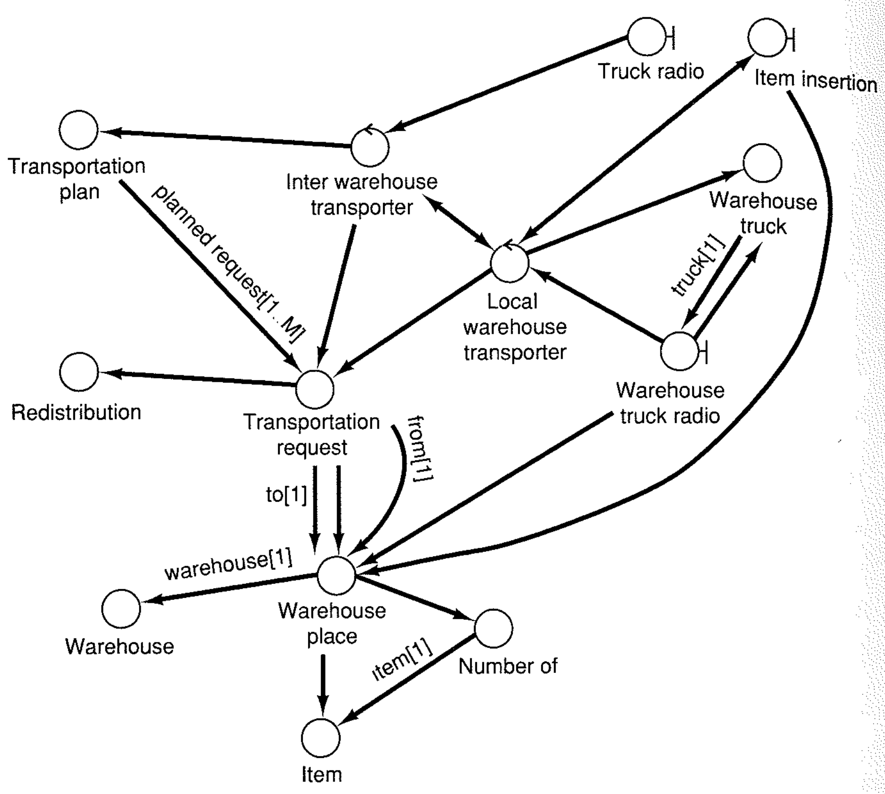


Figure 13.20 The Unloading part of the use case.

returned to the Interwarehouse Transporter. That object waits for the truck to arrive. When the truck driver signals that the items are stored, the expected arrival time to the destination warehouse is stored in the Transportation Request. The number of items shipped is then decreased at this warehouse.

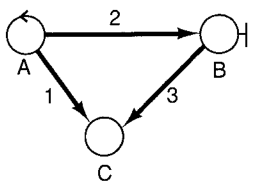


Figure 13.21 A common structure to facilitate reuse is when an object A gives a reference of an object C to another object B Then B accesses C.

The use case views are quite extensive It can be quite hard to find all the dynamic associations.

A structure that often appears is shown in Figure 13.21. A control object or an entity object hands a reference to an object over to another object. This object then accesses the referenced object. This structure often encourages reuse since the accessing object does not need to know which object it sends messages to; the only requirement is that the accessed object can respond to these messages. For instance, the interface object Forklift radio wants to ask for 'radioNetworkAdress' and is not interested in which kind of object answers

13.4.2 Subsystems

The lowest level of subsystem is a unit of configuration designed for managing purposes. It contains a group of objects that represent a self-contained piece of functionality. It also represents a piece of the system that a customer either wants all or nothing of As an example, a subsystem of a wordprocessing program could be a tool for spelling correction or hyphenation

Subsystems are also used as a basis for changes; each change proposal should ideally influence only one subsystem. Furthermore, they are used as a basis for the transition from analysis to construction. All these different usages of subsystems complicate the task of deriving them from the objects.

From the objects, we can see quite clearly that some of the objects are needed to transport between warehouses while some of the objects are only needed within one specific warehouse. When regarding ordering criteria we may have the case that we want to be able to deliver a part when adding one more warehouse in the system. Additional changes in the system can come from changes inside a warehouse or changes that affect interwarehouse communication. Hence there seem to be good reasons for having this as a first basic criterion when dividing the system.

When further reviewing the objects (also including other use cases), we see that some objects are used mainly to manage the system information while some objects are used to perform the transportation. Since these two areas are very much 'self-contained pieces of functionality' and also may have a different frequency in their changes, we should also use this as a criterion for the division. Thus we now have four subsystems, namely:

- Single Warehouse Management
- Multiple Warehouse Management
- Local Warehouse Transportation
- InterWarehouse Transportation

To attach the objects to these four subsystems is now quite easy, but when reviewing the objects once more in view of potential changes in the system, we see one object that will probably be changed quite often initially That is the Planning object. We have not discussed the planning strategy here, but there are many ways of getting a fast and effective planner. One possibility is to use techniques from artificial intelligence to do the planning. We will probably need to simulate and prototype the planner to get it effective, but even then we will probably want to optimize it when we have had some experience with the system in operation. Additionally, since the system will execute in a dynamic environment, many of the plans will be inexecutable since something has happened in the environment. Then we may need to perform a replanning of the current plans to adapt to the new circumstances. The areas of planning and replanning are currently evolving fast in the AI community. Hence there are many reasons to isolate the planning object so as to handle future changes more easily. Therefore we add one susbsystem object to handle the planning.

When attaching the objects to a specific subsystem, we must be aware of one thing. If the system is to be used only within a single warehouse (a quite reasonable guess), we will only deliver the subsystems that are local to a warehouse. Therefore all objects that are basically needed in the system should be placed in the single warehouse subsystem to be sure that they will always be delivered.

Hence the subsystems will be the following, with its contained objects

(1) Single Warehouse Management
  (a) Item
  (b) Number Of
  (c) Warehouse Place
  (d) Transportation request
  (e) Item insertion.

(2) Multiple Warehouse Management.
  (a) Warehouse
  (b) Redistribution
  (c) Manual redistribution window

(3) Interwarehouse Transportation
   (a) Interwarehouse transporter
   (b) Transportation plan
   (c) Truck
   (d) Truck radio.
(4) Local Warehouse Transportation
   (a) Local Warehouse Transporter
   (b) Forklift
   (c) Forklift radio.
(5) Planning
   (a) Planning.

Almost all the other subsystems use objects in the subsystem Single Warehouse Management. It contains the basics of what we need to run the warehouse management system (from the perspective of the use case Manual redistribution between warehouses). All different configurations of the system must have at least this subsystem.

Different subsystems have different purposes. The subsystem Interwarehouse Transportation contains the objects needed for transportation between warehouses. On the other hand, Local Warehouse Transportation contains the objects needed for transportation within a warehouse. Multiple Warehouse Management's purpose is to handle the functionality that makes it possible to manage many warehouses. Finally, Planning takes care of using the trucks as economically as possible.

A subsystem is said to be *dependent* on another subsystem if at least one of its objects is dependent on an object (i.e. associates it in some way) in the other subsystem. In Figure 13.22 we see an example of a dependency between two subsystems. The dependency originates from the association between the two objects.

Dependencies between subsystems limit the number of possible configurations of the system. The dependencies between the subsystems of the example are shown in Figure 13.23, they are derived from the associations in the use case Manual redistribution between warehouses.

Figure 13.24 shows one of the possible configurations of the system.

We have here identified subsystems when the object model has been completed. Sometimes it is more appropriate to identify them earlier, often before you identify the objects.

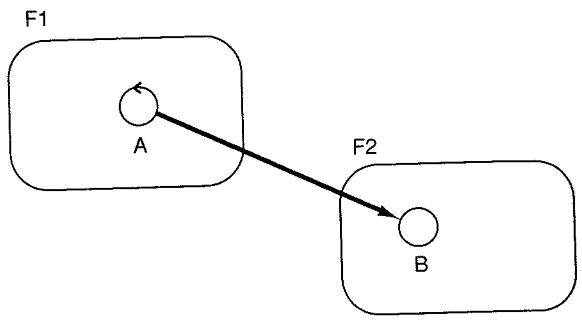


Figure 13.22 An example of a dependency between subsystems The control object A accesses the entity object B This leads to a dependency between the subsystems F1 and F2.

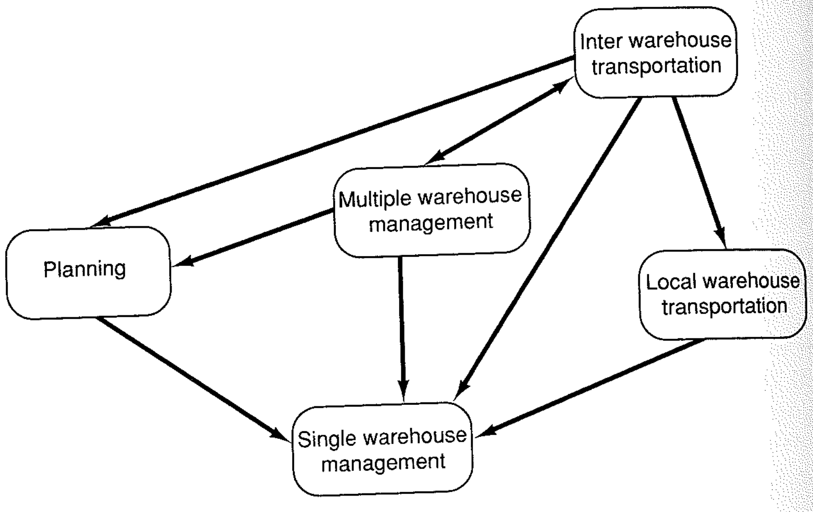

Inter warehouse transportation
Multiple warehouse management
Planning
Local warehouse transportation
Single warehouse management

Figure 13.23 Dependencies between the subsystems of the example system.

13.5 Construction

The analysis model has now been developed under ideal conditions. It is mainly a logical model of the system without regard to the current implementation environment. The main purpose of the construction and implementation process is to customize this logical model to a specific implementation environment to finally implement the system

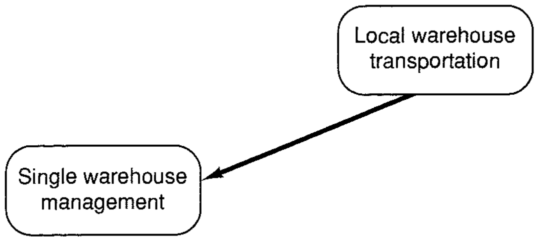


Figure 13.24 A possible configuration given the dependencies between the subsystems

13.5.1 Identify the implementation environment

To do this customizing we need to analyse the current implementation environment. This identification and analysis should aim at clarifying strategic questions for the implementation and drawing conclusions on how to implement certain tasks. In our example we need to clarify how to handle the distributed environment, the relational database, distribution of the database, the radio network, and incorporation of results from the UIMS, for example. Many of these questions have several answers and it is important to choose one before starting the actual design. This is to avoid redoing things later and having different solutions by different developers

Ideally, the implementation environment should be specified from requirements that have evolved from the analysis work and the implementation analysis. Then we will have the implementation environment that is ideal from our perspective. However, this is seldom the case in practice. In our example we had the prerequisite of using C++ and a relational database. These prerequisites are actually quite good. C++ is one of the more efficient and widespread OO languages and is also appropriate here A relational database is actually also quite appropriate in this case. The data structures in the example are quite simple and thus a RDBMS will probably have better performance than an ODBMS. However, we would need a support for a distributed database

How to handle all the specific parts in the implementation environment is not something that OOSE controls. Normal software engineering practice should be used and techniques used elsewhere are often also appropriate here. In this book we have illustrated some examples of how to handle these issues (e g databases, programming language and real time systems). However, our experience shows that you should strive for encapsulating consequences of the implementation environment as far as possible. This is to spoil the logical structure as little as possible and also to encapsulate potential changes in the implementation environment. In this example we will not do an entire analysis of the actual implementation environment, but rather give some examples of how to reason when analyzing and handling it.

When analyzing a system like this it is often appropriate to start from the major structure of the system, namely the system **hardware configuration**. In the example we see that the system will be spread over several sites, namely the warehouses and the office. We can then either have one central mainframe to serve all these users or have a decentralized structure of computers. Since there is actually no natural central site, but rather the system has a very decentralized nature, we choose to have a decentralized structure. Thus the office needs one server and each warehouse also need a server. The office personnel and the foremen will be clients to these servers. The warehouse workers and the forklift operators will also be clients of the servers located at the warehouse site. The truck drivers will communicate with the system from several different places. To choose one specific server for them is harder since they will not be tied to any specific warehouse. One possibility is to give them the office computer as a server, another is to allow them to be distributed and to connect to the closest server at any particular moment. Since many network facilities today support this latter facility, and also because the future in telecommunications is going in this direction, we choose to allow the truck driver to use the closest site as his server In the future the customer should also be able to use the system as a client Then it should be possible for him or her to use the closest site as his server. Thus it is appropriate now to prepare for a dynamic use of the servers. Our client/server model and structure will thus look like Figure 13.25.

The distributed nature of the system must be supported by some kind of network. This network would probably be the public telecommunication network. There are products existing to handle this distribution which can very well be used here. In our system we use such facilities and thus do not develop this part by ourselves, but rather use an existing product. What we do need, however, is to design an interface to this product so that our system will not be too dependent on the chosen product. It should also be possible to change the communication product with as few changes as possible to the system.

In larger systems it is also essential to consider how the system should be configured, managed and delivered. When identifying the subsystems we did the first analysis of this. If it is not done before,

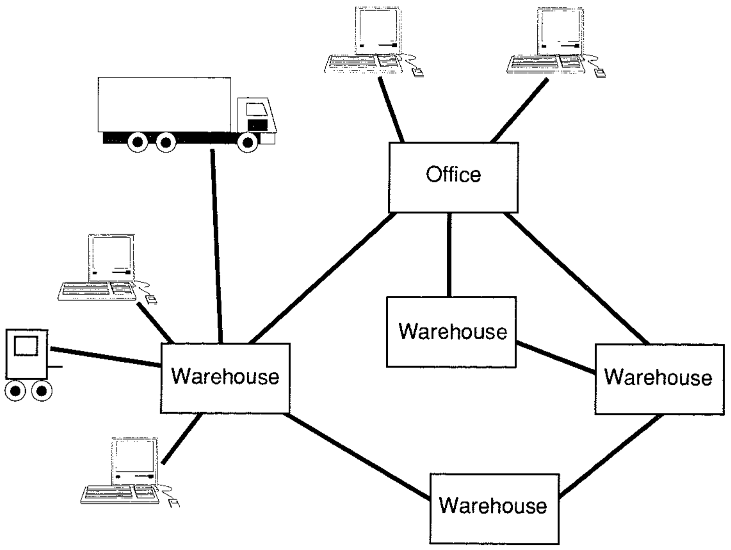


Figure 13.25 An example of the physical configuration of the system. The boxes represents servers in the systems while the trucks and terminals represent clients to these servers

it is essential to do a final analysis and decide upon the product structure here In our example we see that the subsystems identified earlier are appropriate also here. The subsystem Single Warehouse Management must be delivered to each specific warehouse. The other warehouse subsystems must also be delivered to each warehouse with the capability to communicate with other warehouses. Although the office should actually only need the subsystems that are needed for the distribution, we see from the dependency between them (see Figure 13.23) that the single warehouse is also needed. We have not identified subsystems especially for the office, but some subsystems will only be needed at the office site

When discussing the distributed nature of our system let us also discuss how to distribute the database; first the realization of it. There exist commercial *distributed databases*. The best solution is to use one of them to avoid developing and maintaining this facility yourself. It is often best to build a homogeneous database initially, that is, one with all sites running the same DBMS; but during further development, when extending the network, it is seldom possible to maintain this situation. Then we will have a heterogeneous system, that is, we might have different kinds of DBMSs in the network. This is a likely change of the system and we should therefore prepare for it in the design.

A major benefit of a distributed database is that data is stored close to the place where it is most frequently used, thus giving high performance at the same time as having an increased accessibility of information in the entire system. One of the major drawbacks when using a distributed database is that the network is often far too slow. Therefore how you structure the database is very important. The goal is to minimize the communication over the network. We must therefore analyze how frequently the objects should be used at different sites. When regarding the analysis model, we see that it is not a direct map from a particular object to a specific site where the objects will be used. For instance, the Item object will be used from almost all sites. The same goes for almost all objects in the system. This means that almost all objects in the system must be distributed and thus dependent on the communication product discussed earlier. We will discuss shortly how to handle this.

To incorporate the relational database in the object-oriented structure of the system we will use the guidelines given earlier in Chapter 10. For a discussion on the topic, please refer to that chapter.

To communicate with the truck drivers, a *radio network* will be used. This network is also an existing product that we should use. Products exist that can be programmed to adapt for the messages to be sent and displayed. When designing, the interface objects should therefore be designed as interfaces to this network. This means that we will not have any problems with the distributed nature of the network – the network will handle that – what we need though is to cope with unique radio addresses, but that will surely be a part in the interface to the network at every site.

This is also true for the forklifts; they will also be handled by a commercially available radio network This means that a truck does not only need a unique identification within a certain site, but within all warehouses and offices connected to the system.

13.5.2 The block structure

We have discussed some of the preconditions to consider when developing the preliminary block structure We will now look at some examples of ways to handle the implementation environment to keep a logical, and thus stable, structure as much as possible.

The ideal block structure would be to take the complete analysis model and map it directly onto a block structure: but is that possible in this case? Let us look at the consequences from the implementation environment analysis. We start with the distribution. One example of blocks that should be spread over several sites is the block corresponding to the Item object. Since the Item objects each belong to a specific warehouse it is natural to place the particular instances at the site where the item is stored: but how do we solve the distribution? The first intuitive picture is to have one block at every site, having the blocks communicating with each other when they need to do so: but this means that when another block needs to communicate with the Item block it needs to know at which site the block exists. This knowledge is not natural for other blocks, but should belong to the block Item itself. Hence we should encapsulate the distribution of the block as discussed in Chapter 9. One possibility for doing this is to use the classes in the blocks. We then have instances of one class that represents the existence of the items at this site. This instance receives stimuli from other parts of the system at this particular site. If the information does not exist at this site, we have to go out on the network and fetch the information. This means that we should have one class that looks up a particular instance of the Item, either at this site or on the network; one class that represents the existence of the Item at this site; one class that works as the interface to the network; and one class that receives signals from the network to this particular Item, see Figure 13.26.

We see that this design idea can be equivalent to all blocks that share the same property of being distributed. Therefore we may very well form an abstract block that all distributed blocks may inherit. Such a reusable design is often called a **framework** The distributed block thus consists of the network classes and a general LookUp class, possibly having only virtual member functions (in C++). The network classes will then be the only classes dependent on the current network; a change in the network will only affect these two classes. This block, called the Distributed block is shown in Figure 13 27.

By designing the distribution in this manner we can design the use cases without having to think about how to find a particlar instance in the network. All this behavior will be encapsulated inside the block

To incorporate the relational database in the system we will use the design discussed in Chapter 10. This gives us a block which will function as the interface to the DBMS. The design framework used also has an abstract block that specifies how the conversion from the OO system to the RDBMS functions. This block should be inherited by all blocks that will have the ability to store their information in the database. This means that blocks that should be both distributed and stored in the database need to inherit both the persistent block and the distributed block. Can we solve this with only single inheritance? Yes, if we have an inheritance hierarchy as

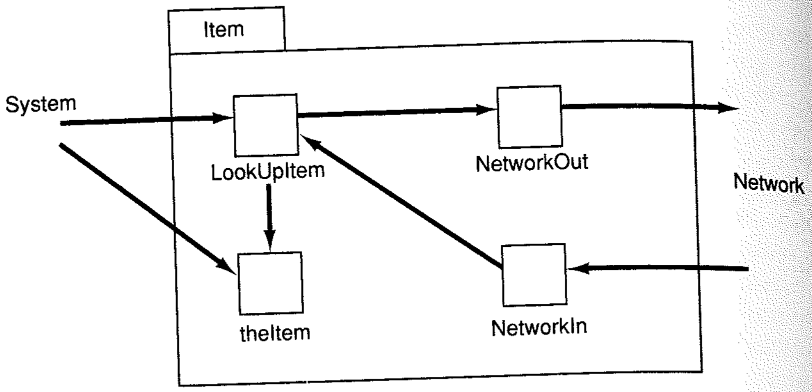


Figure 13.26 The initial design of the Item block. Classes are used to encapsulate the actual distribution of the block.

shown in Figure 13.28(*a*) or (*b*). Here we must choose one of them giving unwanted consequences: in case (*a*) all Distributed blocks must be persistent; and in case (*b*) all persistent blocks must be distributed. In case (*c*) we can have these two abilities independent of each other. Case (*c*) is an example of subtyping (see Chapter 3) with multiple inheritance. Here it is two roles we want to reuse in several blocks, the roles of distribution and persistence. We choose to use the last alternative here.

Since the distributed block and the persistent block may have a general integration which can be made independently of the actual

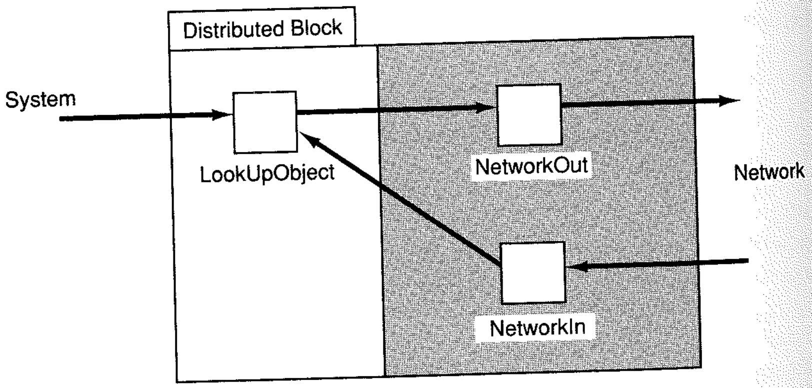


Figure 13.27 An abstract distributed block that will be inherited by all blocks that should be distributed in the system. This block forms a framework for the design of other blocks. Note that some classes are private.

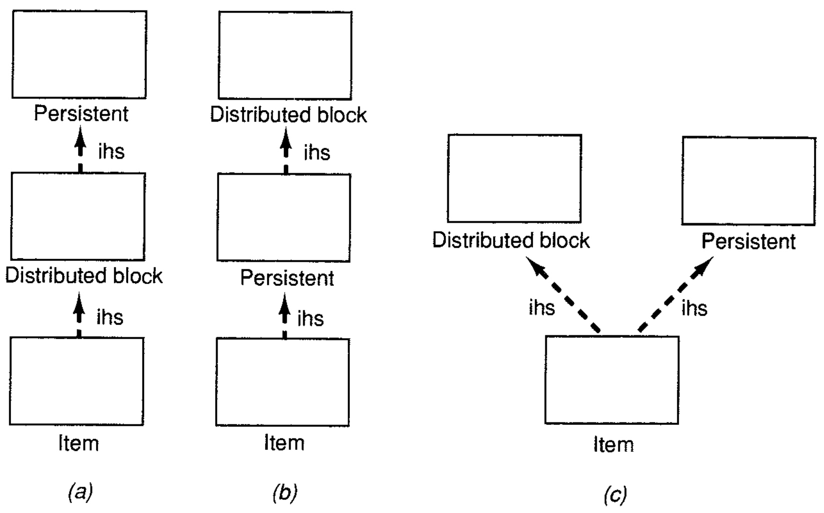


Figure 13.28 Inheritance hierarchies to get both the ability to be stored in the database and the ability to be distributed. In case (*a*) all distributed blocks must be persistent while some blocks (only inheriting Persistent) only may be persistent. In case (*b*) all persistent blocks must be distributed since persistent inherits the Distributed block. In case (*c*) it is possible to have both abilities independent of each other.

concrete block, we add a new block which incorporates these two blocks. This block is called the Distributed Persistent block and it inherits the two blocks Distributed block and Persistent, as shown in Figure 13.29. Since some blocks may be only persisent or distributed but not both, we also need to retain the two blocks just as they are. This will also be a benefit when we want to modify the persistence or the distribution; the modification will be local and easy to localize.

We also need to cooperate with the radio network. Since the main functionality from an existing network system will be used in this system we need to design the interface to the network system. This may very well be designed as a component used in the network block. The actual network will thus be encapsulated in the radio network block.

The block structure now grows to a first attempt where we have considered some of the consequences from the implementation environment. All blocks that should be distributed and persistent will inherit from the Distributed Persistence block. This accounts for practically all blocks that come from entity object.

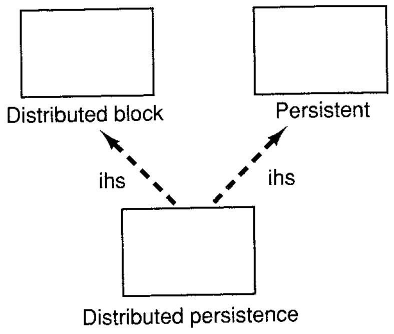


Figure 13.29 Since many blocks will both be distributed and persistent, we add a new block that integrates these two features. All blocks that need these two abilities will inherit from this particular block

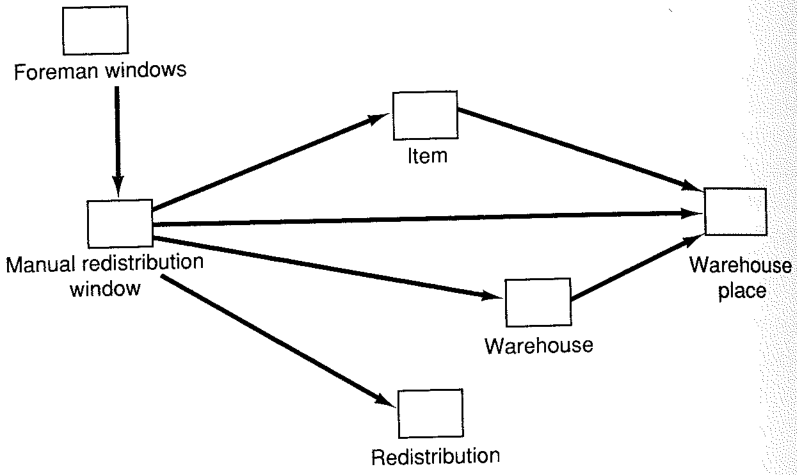


Figure 13.30 Block diagram for the first part of the use case *Manual redistribution between warehouses*

We will here continue to focus on the use case *Manual Redistribution between warehouses*. The block diagram for the first part of the use case will look like Figure 13.30. Here we have taken the analysis model with very small modifications; the only change is that we have removed the Number of entity object as a way to express how many Items are stored at a specific Warehouse place. Instead we have drawn the communication associations as bidirectional since both blocks will initiate stimulus sending. The reason that the Number Of block is excluded is that the attribute will be a column in the table connecting Item and Warehouse place since the association between them is [0..M]. In that table we can include the information. The information can also be stored attached to the references between the classes in the blocks in C++. Thus it would be overkill to have a block for this information.

Block diagrams over the other part of the use case will be discussed when we design the use cases over the blocks.

13.5.3 Use case design

We will now look at the design of the use case *Manual Redistribution between warehouses.* We will then use interaction diagrams. The division into four parts will also be used here.

Initialization

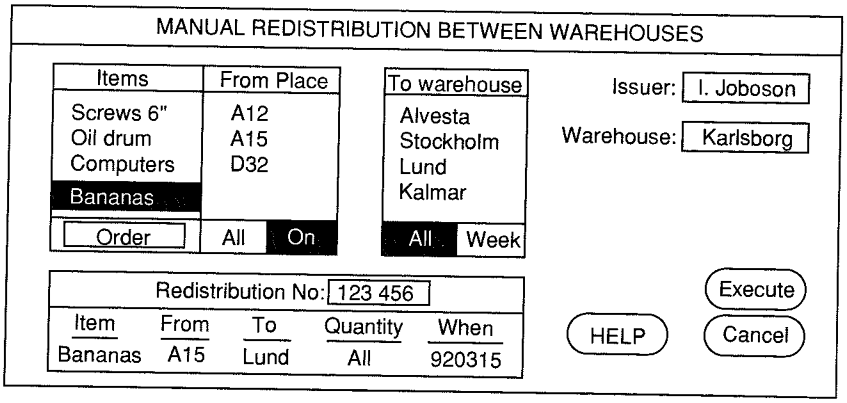
(1) The Foreman gives a command for redistribution between warehouses

(2) The window in Figure 13.31 is presented to the foreman

(3) The items can be ordered in a number of ways. This is selected with the ORDER menu. The following orders are possible:

(*a*) Alphabetical order,
(*b*) Index order (each item has a unique number),
(*c*) Turnover of the items,
(*d*) Storing order

(4) In the 'From place' table we might select to view either all places in the current warehouse or, if we have selected an item, the places where that item exists


MANUAL REDISTRIBUTION BETWEEN WAREHOUSES

Items	From Place
Screws 6"	A12
Oil drum	A15
Computers	D32
Bananas	

To warehouse
Alvesta
Stockholm
Lund
Kalmar

Issuer: I. Joboson

Warehouse: Karlsborg

Redistribution No: 123 456

Item	From	To	Quantity	When
Bananas	A15	Lund	All	920315

Figure 13.31 Manual redistribution window.

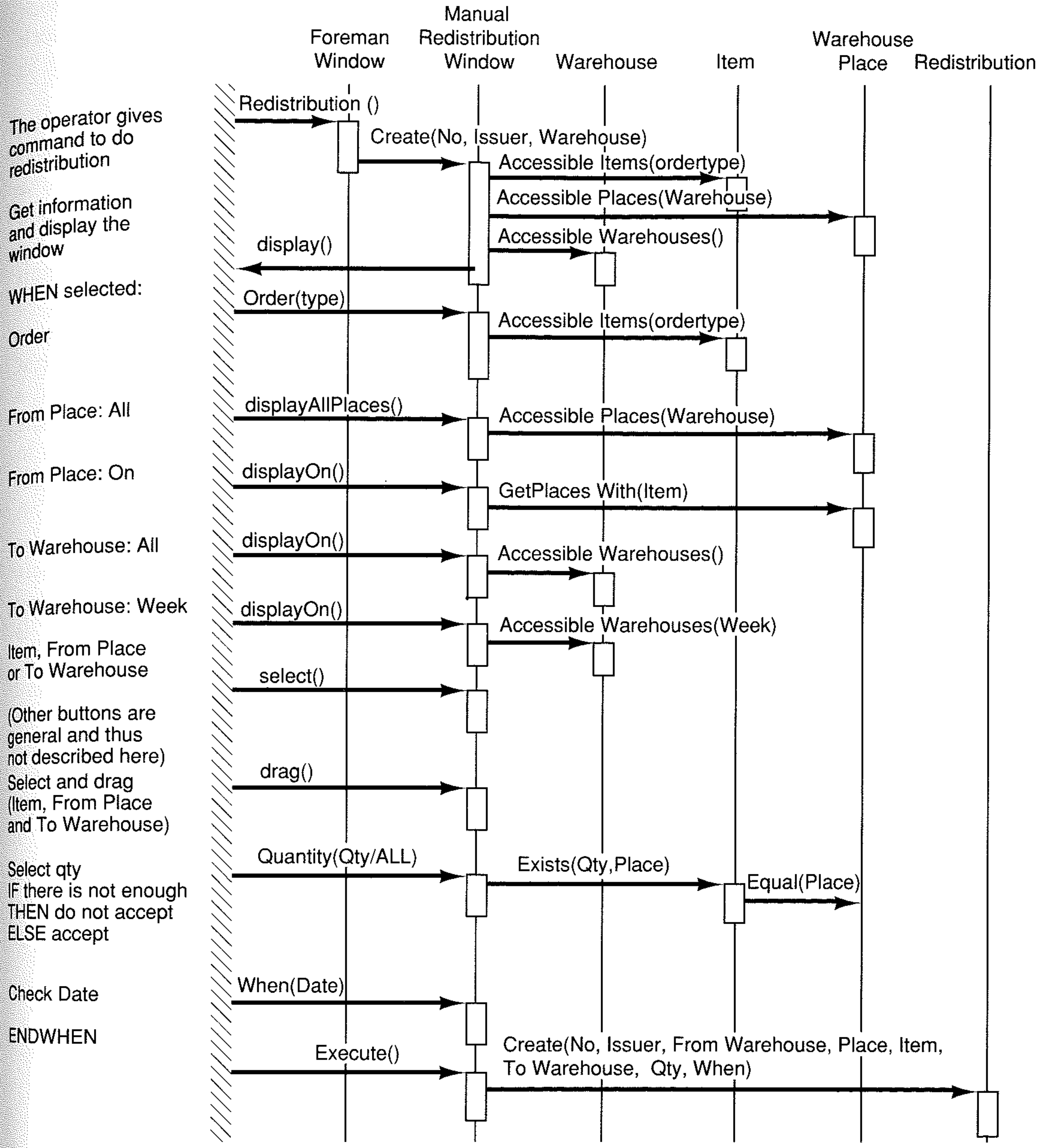
(5) In the 'To warehouse' table we might select all warehouses or the warehouses we have to transport to this week.
(6) The 'Issuer' and 'Warehouse' field are automatically filled when the window pops up, but they might be changed. (This is the way to do a redistribution from another warehouse to our own warehouse.)
(7) The foreman selects an item by pointing to it and dragging it to the Redistribution form. He then selects from which place to take the items and to which warehouse to transport them. This information is automatically shown in the form.
(8) The foreman then gives the quantity to be moved, a possibility is to give 'ALL', and then the foreman gives a date by when it must be done.
(9) It is possible to change the information when the form has been edited. When the foreman EXECUTES the redistribution, the transports are beeing planned. It is also possible to CANCEL the redistribution as such. When selecting HELP a window is shown that gives information about the current window.

This sequence will now be distributed over the blocks participating in the use case and we will now define the stimuli sent between the blocks. To recall the Initialization part of the use case, we give it here again. The corresponding interaction diagram is shown in Figure 13.32. This diagram has the typical behavior of external events. Many events may happen in an arbitrary order. Every external event gives rise to some action inside the system.

In the interaction diagram we see that almost all events are directly triggered from an operation on the window. We thus have no complex sequences in this diagram. This we could also have guessed, since we do not have any blocks originating from a control object, which are typical for more complex sequences. Let us continue by looking at the next part of the use case, namely the Planning part. The first use case description looked like this:

Planning

(1) When the redistribution is executed, the items to be moved are marked as reserved.
(2) The planning should minimize the use of the trucks with the condition that all delivery dates should be held and the trucks should fit any delivery requirements of the articles (e.g. in size). The minimization should be done by adjusting the new redistribution requirements with existing, already planned redistributions
(3) This may give rise to new transport requests and may also change existing transport requests already in the system
(4) The transport requests are connected to a specific truck's transportation plan.



Foreman Window
Manual Redistribution Window
Warehouse
Item
Warehouse Place
Redistribution
The operator gives command to do redistribution
Redistribution ()
Create(No, Issuer, Warehouse)
Get information and display the window
Accessible Items(ordertype)
Accessible Places(Warehouse)
display()
Accessible Warehouses()
WHEN selected:
Order
Order(type)
Accessible Items(ordertype)
From Place: All
displayAllPlaces()
Accessible Places(Warehouse)
From Place: On
displayOn()
GetPlaces With(Item)
To Warehouse: All
displayOn()
Accessible Warehouses()
To Warehouse: Week
displayOn()
Accessible Warehouses(Week)
Item, From Place or To Warehouse
select()
(Other buttons are general and thus not described here)
Select and drag (Item, From Place and To Warehouse)
drag()
Select qty IF there is not enough THEN do not accept ELSE accept
Quantity(Qty/ALL)
Exists(Qty,Place)
Equal(Place)
Check Date
When(Date)
ENDWHEN
Execute()
Create(No, Issuer, From Warehouse, Place, Item, To Warehouse, Qty, When)

Figure 13.32 The interaction diagram for the Initialization part of the use case. Note the typical structuring from external events coming into the window block

To draw the interaction diagram we first look at the block diagram. Since all objects in this model will be substantial, the block diagram will be a direct map of the analysis model. The diagram is shown in Figure 13.33.

The interaction diagram of this part of the use case is shown in Figure 13.34. Note that much of the behavior has been centralized in the Planning block. The reason for this is the discussion of the changeability of the planning routine; we believed that it would change and be optimized quite frequently. Therefore the behavior is not distributed, but rather held together. Note also how the stimuli have been designed to be reusable. The Add( ) stimulus to Transportation Request has been used twice. It would be possible to include this behavior in the Create( ) stimuli, but since we might like to do creation separately from the addition of information in a request, we have designed two stimuli for this. Additionally, if the Create( ) stimuli involved the Add( ) information, a change in this information would affect two stimuli instead of one. The other parts of this use case are designed in the same way.

In this fashion all use cases are designed and the behavior is distributed over the blocks and the interfaces between the blocks are specified. This is an iterative process where use cases are designed and the block structure and stimuli are modified. In practice it is very hard to design the use cases in detail without having any simulation tools. One possibility is to design a use case and simultaneously implement the interfaces of the classes. In C++ this means that the header files will be developed during the use case design. The body files may be implemented only with stubs in this

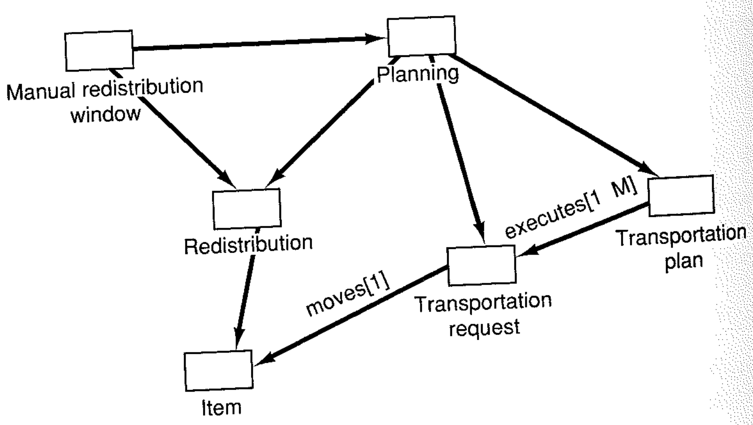
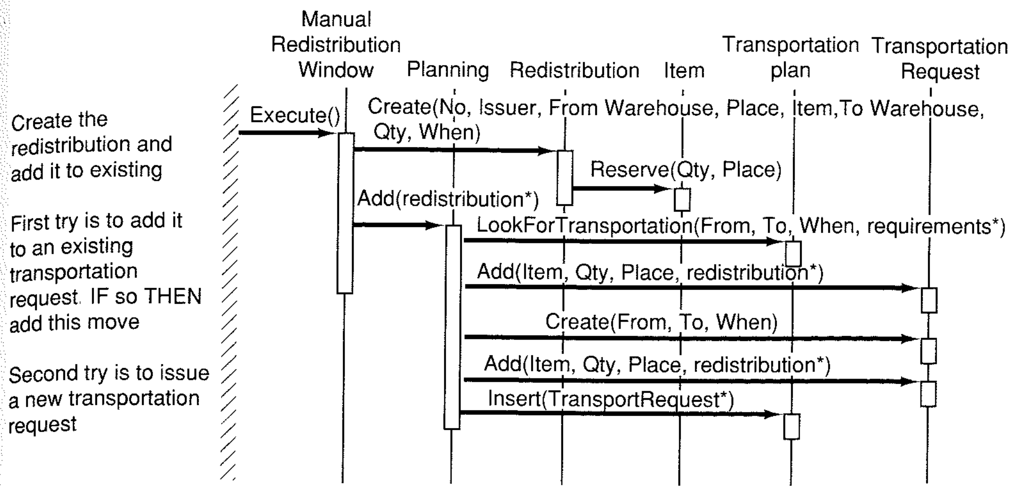


Figure 13.33 The Block diagram for the Planning part of the use case. This is a direct map from the analysis model.



Manual Redistribution Window
Planning
Redistribution
Item
Transportation plan
Transportation Request
Create the redistribution and add it to existing
First try is to add it to an existing transportation request. IF so THEN add this move
Second try is to issue a new transportation request
Execute()
Create(No, Issuer, From Warehouse, Place, Item,To Warehouse, Qty, When)
Reserve(Qty, Place)
Add(redistribution*)
LookForTransportation(From, To, When, requirements*)
Add(Item, Qty, Place, redistribution*)
Create(From, To, When)
Add(Item, Qty, Place, redistribution*)
Insert(TransportRequest*)

Figure 13.34 The Planning part of the use case distributed over the blocks involved in the sequence

phase. Then you are directly able to simulate your use case design and thus develop the interaction diagrams close to the code in an incremental manner. The use case design then helps you to find the interfaces to the classes while the classes help you to verify your design incrementally. We will now continue to look at how the blocks are designed from this.

13.5.4 Block design

The designing of the block may start when the design of the use cases becomes stable. From the use cases we will have detailed specifications of the protocols of the block and the behavior associated with this protocol. The task is now to decide upon an implementation of the block. The first sketch has already been made when making the first attempt at a block structure under consideration of the implementation environment. This design will now be refined and finally implemented.

We will here focus on the Item block and only take into consideration the protocol identified in the use case design described above. When doing the extensive design of all use cases we would have a much larger protocol, but the idea should be clear. The rest is a straightforward realization in the same manner as described

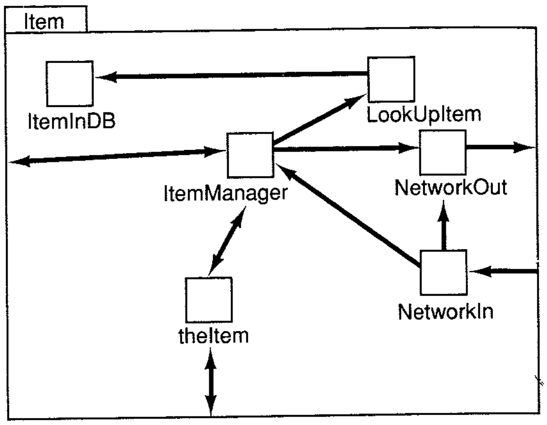


Figure 13.35 The detailed structure of the *Item* block.

here. From the use case design described above we can identify the following signatures of the Item block:

AccessibleItems(ordertype)
Exists(Qty, Place)
Reserve(Qty, Place)

This identification is done straightforwardly merely by looking at the pile representing the block Item. This is a mechanical extraction which can be automated.

To realize the block let us look at the first attempt at the structuring of the block. Since Item will be both persistent and distributed, the block will inherit from the Distributed persistent block. We will thus have a structure that looks like Figure 13.35.

The classes are explained below:

- ItemManager manages all Item instances at this site and also can give requests on the network to other sites. It inherits from the abstract class Manager.
- theItem is the actual Item object with state and operations. It inherits from the class Persistent.
- LookUpItem, from a specific key, finds the item in the database.
- ItemInDB: when LookUpItem finds an Item it creates instances of this class. When the Item is needed in the system an instance of theItem is created by ItemManager and information from this instance is converted to instances of theItem. The reason for having two classes for this is that possibly not all instance

variables of theItem will be stored in the database. It inherits from the class ObjectInDB.

- NetworkIn and Out are interface components to handle the network.

When looking at the protocol we can assign the signatures to specific classes. The AccessibleItems() and Exists() function will handle several instances of the Item while the Reserve() function will be called on a specific instance of theItem. We thus have the following header files for the classes of ItemManager and theItem:

```
enum ordertype {alphabetical, index, turnover, storing};

class ItemManager: private Manager {
public:
  list AccessibleItems(ordertype order);
  boolean Exists(int Qty, storePlace Place);
  //
private:

  };

class theItem: public Persistent {
public:
  status Reserve(int Qty, storePlace Place);
  //
private:

  };
```

This is a direct mapping onto C++ member function definitions from the interfaces extracted from the interaction diagrams. The implementation of these functions may initially be done using stubs as discussed above, and later refined into the actual code. The benefit then is that you are able to simulate the use cases in the code simultaneously as you develop them. The final implementation may then evolve from these stubs.

When implementing the blocks, new stimuli will arise from sequences not described in any interaction diagrams. In particular, flows that are internal to the block and not shown outside will be added. These flows are internal and thus hidden from the outside, that is, they describe the internal implementation of the block and are up to the block designer to implement. Other additional flows that often arise are error flows, namely flows that will arise when errors occur. These will also be added later to make the system robust.

Here we have only outlined the analysis and construction of the system The real system includes many more use cases and interaction diagrams. The use of existing products has also only been outlined. Since this is nothing unique for OOSE, but rather is quite common to all larger system developments, we have not described it in detail. Additionally, we have been very sketchy in the implementation. This is due to the fact that this book does not cover programming techniques; there are other excellent books on this topic We hope, though, that the strategy to reach the code is clear. Neither have we mentioned anything about how to organize the system into files and file structures. This must also be handled. Components have been mentioned briefly while the testing is omitted. Quality assurance, metrics and bad design decisions have also been left out here. The reader should understand by now that development of real software systems needs extensive support from a well-defined development process, something that is only surveyed in this book. For real system development a far more comprehensive process description is therefore needed

14 Case study: telecom

14.1 Introduction

This chapter will describe the development of a telecom switching system. The focus will be on the parts handling a local phone call between two subscribers connected to the same switch. We first discuss the functionality of switching systems in general and the specific requirements of parts on which we are to focus. We then develop the different models concerning this functionality. We use Smalltalk to illustrate the implementation, although Smalltalk is not (yet) widely used in this application area.

14.2 Telecommunication switching systems

Before looking at the requirements on the functionality to be developed, we will give a very short and simplified introduction to the world of telephone exchanges. An exchange connects subscribers with each other. The subscriber that calls is called the **A-subscriber**. As she or he dials the number, the exchange analyzes the digits and looks up the line to the subscriber to be called. This subscriber is called the **B-subscriber** The exchange then connects the lines of the two subscribers so that they can talk to each other. When they are finished, they put their telephone handsets down (called on-hook in the telecommunications world; picking up the phone is called off-hook) and this makes the exchange disconnect the two lines. Normally, a subscriber can be both an A-subscriber and a B-subscriber, but in a specific phone call he or she can only play one of the two roles.

Every unit connected to the switching system is called a **device**. A line to or from a subscriber for example is a device. This implies that each specific subscriber is connected to a certain exchange. Each subscriber has two associated physical devices, an **A-subscriber line** and a **B-subscriber line**. These devices are used to make outgoing

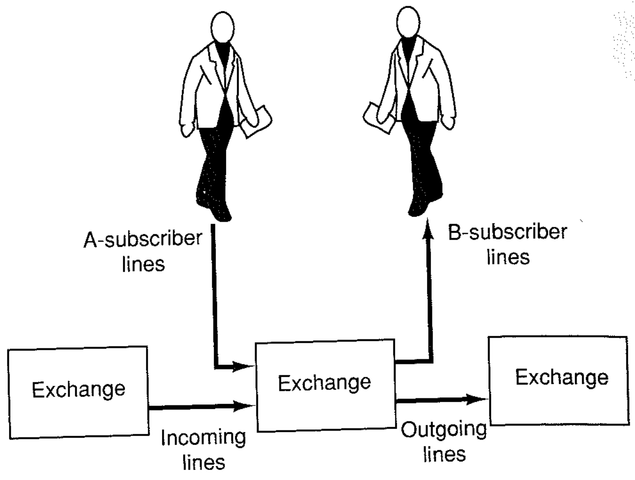


Figure 14.1 A schematic picture of a telecommunications network.

calls and to accept incoming calls. Since a larger telecommunications network cannot be handled by one exchange only, devices can also be **incoming lines** or **outgoing lines** to other exchanges, see Figure 14.1. An incoming line is connected to an A-subscriber in another exchange, while an outgoing line directs the call to a B-subscriber in another exchange. When a subscriber wants to call a subscriber connected to another switching system, she or he uses outgoing lines on the system to connect to the other system's incoming lines.

A **route** is a collection of outgoing lines, all in one specific direction, such as all lines from one exchange to another. To communicate with the other system, our system will choose one outgoing line from the route in that direction. It is of no interest which line in the route, so long as it is not busy, the system can use any free line in the route.

Before we start defining the requirements of the functionality, we will make some general assumptions:

- A subscriber is assumed to have two identities: a subscriber number known among subscribers and found in the directory, and a physical external identity used by the maintenance personnel,
- It is assumed that hardware associated with each subscriber will recognize whether an off-hook is performed as the beginning of a new call or as an answer to a call,
- All devices have an external physical identity,

- Each route has a unique identity,
- In our example system we will, for simplicity, not directly include the application hardware, but instead consider the hardware as part of the underlying system.

Now let us turn to the actual functional requirements. The requirements specification is separated into two groups:

- Call handling
- Operation and maintenance

Call handling

Call between A-subscriber Line and B-subscriber Line. When an A-subscriber lifts his handset, it results in the following actions by the system. First the subscriber category is checked to see whether the subscriber is allowed to make outgoing calls. If so, mark the subscriber busy for incoming calls, and return the dial tone. The dialling tone indicates that the system is ready to receive digits The dialling tone is disconnected when the first digit is received. After dialling all digits, the A-subscriber will be connected to the B-subscriber.

If the B-subscriber is idle and if his category allows incoming calls, he will be marked busy and the A-subscriber and the B-subscriber will be connected When the connection is performed, the B-subscriber will get a ringing signal and the A-subscriber will receive a ring tone. Both these signals will be interrupted when the B-subscriber answers In a normal call, the two subscribers talk to each other for some while before disconnection. The call will be disconnected when both the A-subscriber and the B-subscriber have cleared. Then both subscribers are marked idle and they are disconnected.

If one of the parties clears, the call is continued if the party that has cleared lifts his handset again. The A-subscriber may clear the call at any moment during the call set-up.

Call between A-subscriber Line and Outgoing Line. This function is similar to the *Call between A-subscriber line and B-subscriber line* with a few exceptions. The direction of the call will now be to an Outgoing line This will be chosen among the free lines in the route between this switching system and the other system. The line will be marked busy during the call in the same way as the B-subscriber line. The system does not need to check the category of the outgoing line since it is always allowed to receive calls.

Call between Incoming Line and B-subscriber Line This function is similar to the *Call between A-subscriber line and B-subscriber line* with a few exceptions. The function will be started when another switching system starts communicating with this system. In this case all the digits will be received directly and there is no need to check whether calls are allowed to originate from this line. The function will then behave like the function *Call between the A-subscriber and B-subscriber lines.*

Call between Incoming Line and Outgoing Line. This function is similar to the previous functions but in this case the line from where the call originates in this switching system is an Incoming line and the direction of the call is to an Outgoing line.

Operation and maintenance

Subscription Changes. An operator can connect a new subscriber to the system or disconnect an existing subscriber. When a new subscriber is connected to the system, he or she is informed of the phone number of the subscriber, its external identity object and its category.

Changes of Digit Information. An operator can change the digit information for the system. This information is used for each call in the system so that proper action can be taken. The information is normally structured as a tree where each node represents a digit and a leaf node represents a direction.

The result of a digit analysis is associated with a leaf node. The result consists of three components: the name of the outgoing function to which the call is directed, an external route identity object (not required for a call to a B-subscriber line) and the expected number length (dependent on the direction of the call). The number length is used to decide when all digits for a call have been received.

Connection of Devices. This function is used to connect new devices and to delete existing devices from the system. For each connected device the system is informed of the external identity of the device and the external identity of the route to which the device belongs. The external route identity is used outside the system and for communication with the system.

14.2.1 An overview of the system

Before going into the development work we will now give a brief overview of how the system works. When a new subscriber is to be connected to the system, an operator uses the function described as *Subscription Changes.* The operator gives the subscriber's number, external identity and category Then the operator uses *Changes of Digit Information* and the system is informed about the new subscriber, the number of the subscriber, the direction of the call and the length of the number. The number of the subscriber is used to find the other information, such as the direction of the call. It is now possible to make a call to and from this subscriber if the subscriber is physically connected to the telecommunication switching system. This should be done before the operator connects the subscriber to the exchange

When the subscriber lifts the handset and is going to make a phone call, the external identity of the subscriber is sent to the system. Then, depending on the destination of the call, the subscriber uses one of the functions described as *Call between A-subscriber Line and B-subscriber Line* or *Call between A-subscriber Line and Outgoing Line*

An operator removes a subscription by giving the number of the subscriber whose subscription should be removed, or for another type of device by giving the external identity of the device to the system.

14.3 The requirements model

In the use case model we will describe what functionality each actor should be able to perform. Therefore we start by identifying the actors. From the above description, we see that the system is primarily developed to serve subscribers. This is thus a candidate actor. However, we see also that the subscriber can play either of two roles; an A-subscriber or a B-subscriber. We have previously discussed the importance of describing roles played to the system. These two roles will therefore be perfect to model with two actors; A-subscriber and B-subscriber. To model incoming and outgoing lines we also use actors since they will show behavior external to our system. Here we see an example where other systems (in this case even the same kind of systems!) are modeled with actors. Let us call these actors A-remote and B-remote. In this example these four actors will be the primary actors. We also have secondary actors, namely actors that will service the functionality for the primary actors. From the description above we see such actors also. We will have an Operator to perform the functionality under Operation and Maintenance above. In this example we will not have any more actors. We thus have the picture in Figure 14.2.

From these actors we will now identify and specify the use cases. In this case we have a much more extensive requirements specification than in the example on the warehouse management system. The requirements specification is here quite detailed and can actually work well as a base for the use cases.

To understand the requirements specification, we will draw a conceptual picture of the system using the domain object model. We

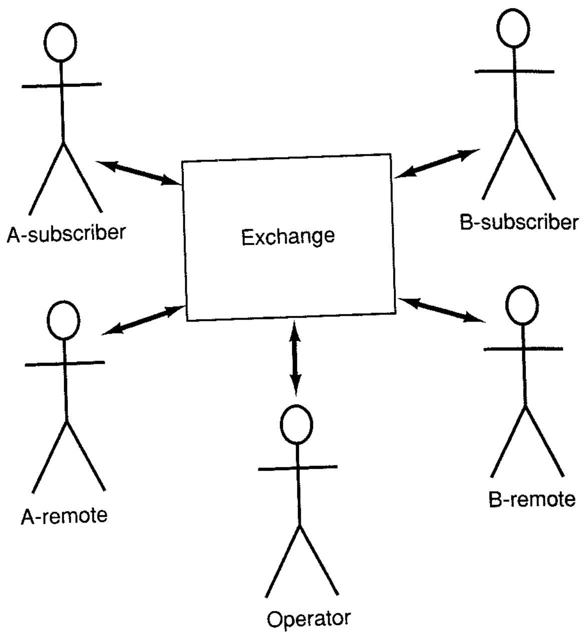


Figure 14.2 The actors of the switching system.

have seen that Subscriber is quite essential in everything that will be done. Additionally, we notice that we have different kinds of devices; A- and B-subscriber lines, incoming and outgoing lines. These are all devices. We also need to group outgoing devices in a Route. To direct a certain call we also need some sort of Directory to keep track of all the Subscribers and Routes We thus have now a domain object model as shown in Figure 14.3.

Some of the objects and their attributes are as follows.

• Device:	physical id
• A-line:	–
• B-line	–
• Incoming line:	–
• Outgoing line:	–
• Subscriber:	number, category, busy
• Directory:	–
• Route.	external id

This domain object model could be further refined to include operations and stimulus paths also. We could even use interaction

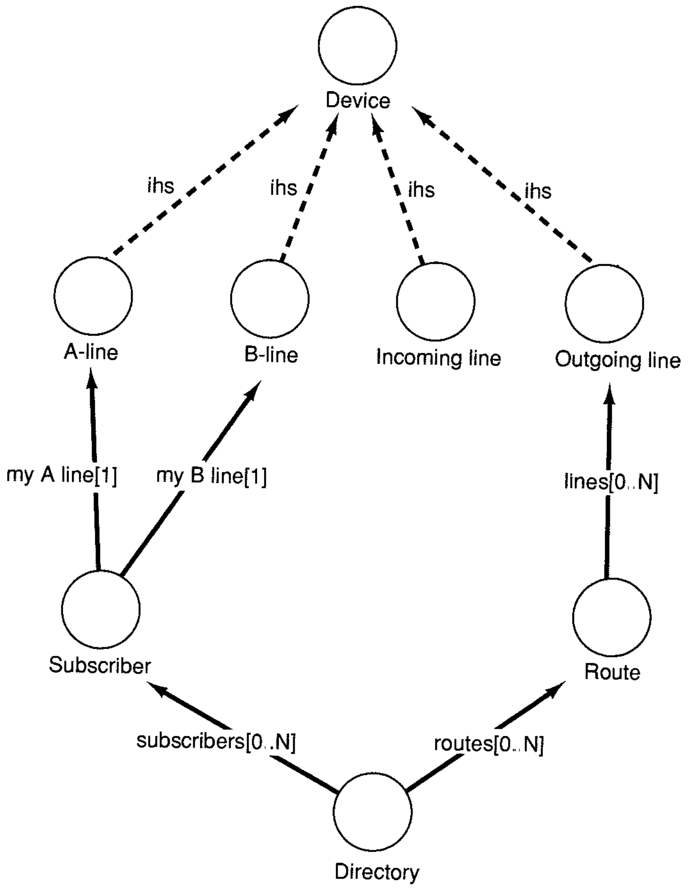


Figure 14.3 The domain object model of the telecom example

diagrams to show how they interact. However, we believe that normally this is overworking the model since this model will not form the direct base for implementation. The above model is for now at a sufficient level of detail to ease the understanding of the domain. However, there may be cases when it is worth refining this model. For instance, if you just want to do a system specification and not implement the system based on this model. For very simple systems, where robustness is not a major concern, it may also be worthwhile to base the implementation on such a model. We will later show that it is possible to execute an entire use case using only these objects.

The call handling functionalities described above have great similarities in having one described in terms of the others. Here there should be a great potential of reuse However, we will postpone the reuse issue until we look at the abstract use cases. We now focus on the actors.

The A-subscriber should be able to call a B-subscriber and an outgoing line. This yields us two use cases; Local Call and Outgoing Call. The A-remote actor should in the same way be able to call the B-subscriber and an outgoing line giving us another two use cases; Incoming Call and Transit Call. The B-subscriber and B-remote will not initiate any use cases and thus not give us any more use cases. The four use cases were the same four alternatives for Call Handling as we found in the requirements. Such a direct correspondence between a requirements specification and use cases is not uncommon, quite the opposite, often use cases can be found directly from the requirements.

In the requirements we do not have anything about how to charge the phone calls. However, it is likely that the calls should be charged. To be able to charge the calls we therefore need to specify how the charging should be performed. Since it is not in the requirements, this question would have to iterate back to the people responsible for the requirements. Let us say that we want the caller to be charged for the phone calls he or she is making. That means that charging should only be included in the use cases Local Call and the Outgoing Call. Incoming Call and Transit Call will not be charged in this switching system, but in the system that the A-subscriber initiating these use cases is connected to. Charging could be described in the use case that should be described, which would mean that we have to define it twice. Another possibility is to describe it as an extends use case to these two use cases This alternative is better since we only have to describe it once and also it is separate from the other two use cases. Additionally, these two use cases can be described independently of how they should be charged. That is good since they are now independent; a change in one of the use cases will not affect the other. Changes in how the charging should be performed will now be local to only one use case We thus identify the use case Charging that can extend Local Call and Outgoing Call. We have the use cases shown in Figure 14.4.

The secondary actor Operator will have to perform use cases to manage the functionality offered to the primary actors. Turning again to the requirements we see that we need a use case to add and delete subscribers; this we call Subscription changes. Additionally, we need a use case to manage and change the digit information; this use case is called Change Digit Information. Furthermore, to handle the devices we need a use case; Device Connection. These use cases are directly found in the requirements. The Charging functionality must also be able to be managed by the operator The operator must be

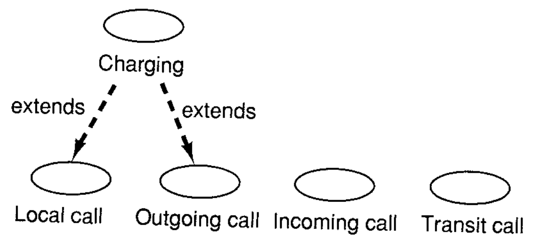


Figure 14.4 The use cases in the exchange system when the primary actors have been analyzed.

able to change a charging factor and also must be able to investigate the amount a subscriber has been charged. We thus have two new use cases; Changing the Charging Factor and Reading the Charging Register. From the requirements we cannot find any more potential use cases in the functionality specified We thus have the use case model shown in Figure 14.5.

The use cases are summarized below:

- Local Call: A call between an A-subscriber and a B-subscriber both connected to the same switching system,
- Outgoing Call: A call between an A-subscriber connected to this switching system and a subscriber connected to another system,
- Incoming Call: A call from another switching system to a B-subscriber connected to this system,
- Transit Call: A call between two subscribers connected to other switching systems using this system for transiting,
- Charging: The use case includes the behavior related to charging a subscriber, namely time measurement, deciding which charging factor that should be used and increasing the subscriber's account.
- Subscription Changes: Connection or disconnection of a subscriber to the system,
- Change Digit Information: The use case changes the associated digit result for a direction in the system,
- Device Connection: The system is informed of a device that is connected or disconnected to the system and its corresponding route,
- Changing the Charging Factor: The system is informed of a new charging factor for a certain direction,

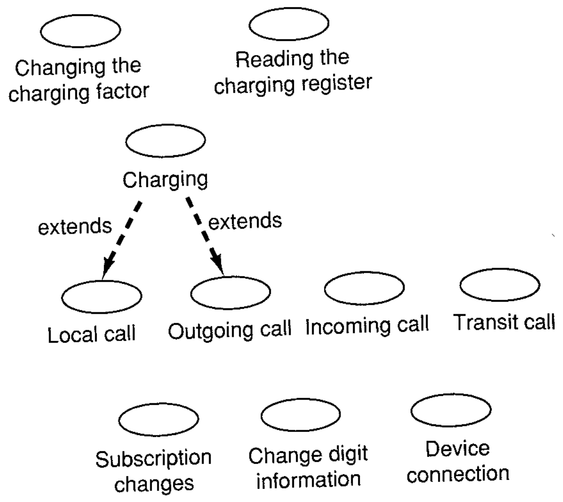


Figure 14.5 The use case model of the switching system.

- Reading the Charging Register: The amount a subscriber has been charged is presented for an Operator together with some additional statistics.

We will from now on focus on the two use cases Local Call and Subscription Changes.

14.3.1 The use case *Local Call*

Local Call is started by the A-subscriber when he or she off-hooks. The B-subscriber will also be involved in the use case even if he does not answer. Specific interface descriptions are not very interesting here since they will mainly be handled by the hardware which we do not discuss. Below is the detailed description of the Local Call use case. Note how the domain objects are used as predefined concepts.

Basic course

When a Subscriber makes a call, namely lifts his handset, a stimulus is sent to the system carrying the external identity object of the calling Subscriber (the A-subscriber) which is used for identifying the Subscriber in the system. The system checks if the Subscriber is permitted to make calls. If so, the Subscriber is marked as busy to prevent other calls being connected to him or her. Finally, a dial tone is sent to the A-subscriber.

A digit from the A-subscriber results in the disconnection of the dial tone and storing the digit. Then the system checks in the Directory if enough digits have been received for analysis, and, if so, requests the outgoing function. In this case the outgoing function request must return a call to a B-subscriber.

Furthermore, if all digits have been received, the system uses the number for translating the number of the B-subscriber to the corresponding internal Subscriber representation. This is used to check in turn if the B-subscriber is idle and if he may receive incoming calls. If so the Subscriber is marked busy and his external identity object is found. Then the system operates the switches between the A- and the B-subscriber and sends a ring tone to the A-subscriber and starts ringing on the B-subscriber.

When the B-subscriber answers the call, the ringing signal to the B-subscriber and the ringing tone to the A-subscriber are disconnected. The two Subscribers are now in conversation mode.

Finally, when the B-subscriber clears the call, the new event is registered in the system. Then, when also the A-subscriber clears the call, the system marks both Subscribers idle and terminates the call by releasing the switches between the two subscribers.

Alternative courses

Termination order

If the call is first cleared by the A-subscriber then the system waits for the B-subscriber to clear. When the B-subscriber clears the call, the use case continues as in the basic course when both subscribers have cleared the call.

Termination point

At any time when the B-subscriber has not answered, the A-subscriber can clear the call. The system marks the calling subscriber idle and terminates the call in case connection has taken place and releases the switches between the two subscribers.

In this use case we also specify two alternative courses; if the A-subscriber on-hooks before the B-subscriber and if the B-subscriber does not answer when he is called.

We stated earlier that it is possible to express the computation as having the domain objects interacting. We will here show that this is possible, although we do not recommend it in the normal case. The reason is that this work will not be used in subsequent phases. However, we could draw a figure like Figure 14.6.

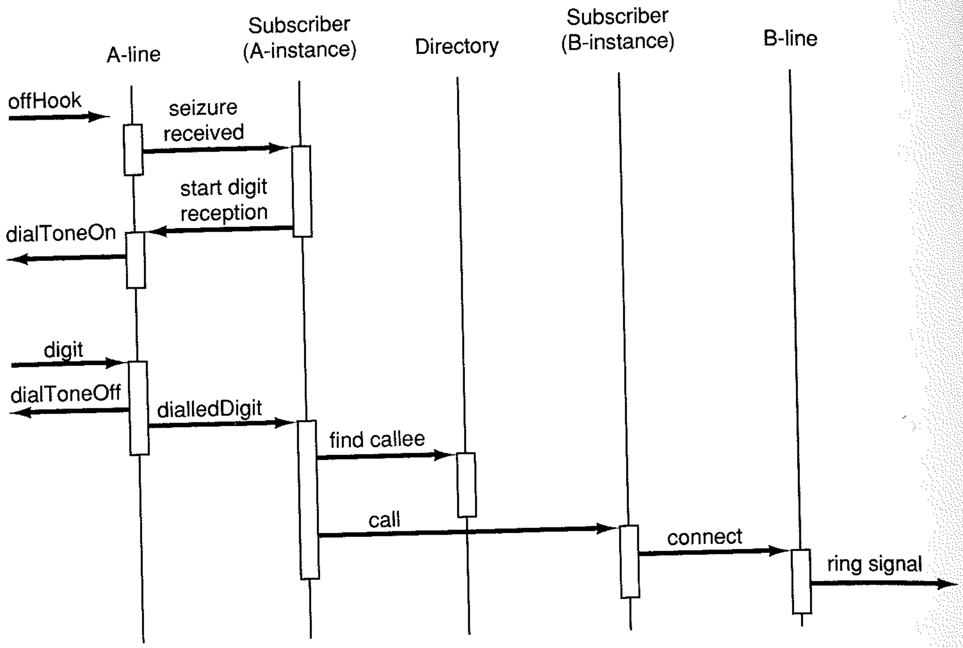


Figure 14.6 The first part of the use case Local Call using the domain objects.

The use case starts when a subscriber off-hooks. This will invoke an off-hook operation in the object A-line. A-line tries to make a call using the appropriate Subscriber object. Then a dial tone is sent. When digits are dialled, they are analyzed and the Directory is used to find the B-Subscriber. When the B-subscriber is found the call is connected. Hence it is fully possible to describe the use case flow over these domain objects, but, we repeat, normally this should not be done since this model will not be further used in the development and it forces the developer to get used to a structure which is not made for robustness.

How the objects are related by the stimulus paths is shown in Figure 14.7.

14.3.2 The use case Subscription Changes

This use case is performed by the Operator when he or she wants to connect or disconnect a subscriber to the system. The specification is as follows.

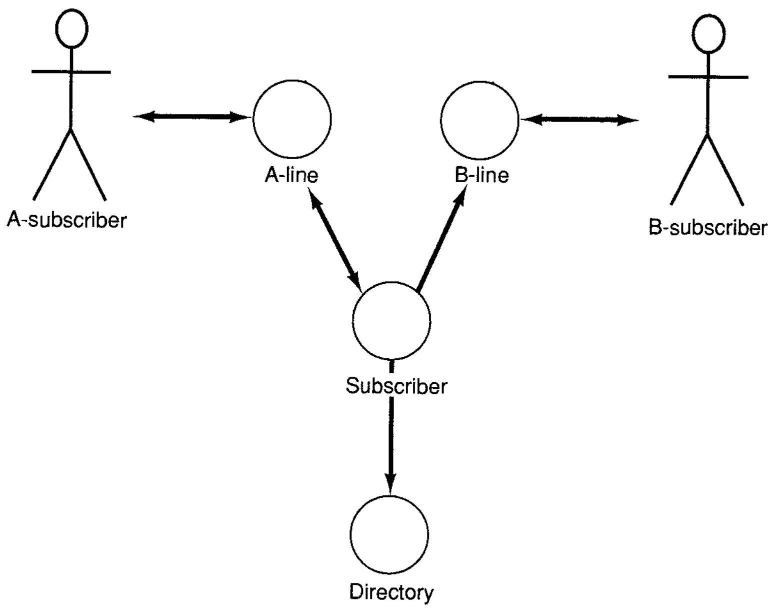


Figure 14.7 The stimulus paths in the domain object model.

Basic course

A new Subscriber is connected by an Operator through a command resulting in a form presented on the screen The Operator completes the information requested, namely a directory number, the subscriber's category and the external identity object of the new Subscriber. The information is stored in the Directory. Finally, the operator is informed of the success of the subscription

Alternative course

Disconnection of Subscriber

Disconnection of a connected Subscriber follows a similar procedure as the basic course. First the information representing the Subscriber is found by the Directory number of the Subscriber. If the Subscriber is idle, its internal information is removed and deleted. The Operator is informed of the success of the disconnection If the Subscriber is not idle the Operator is informed and the disconnection is not performed.

We do not describe here the details of what the window presented to the operator looks like. For a more detailed discussion of window systems see Chapter 13.

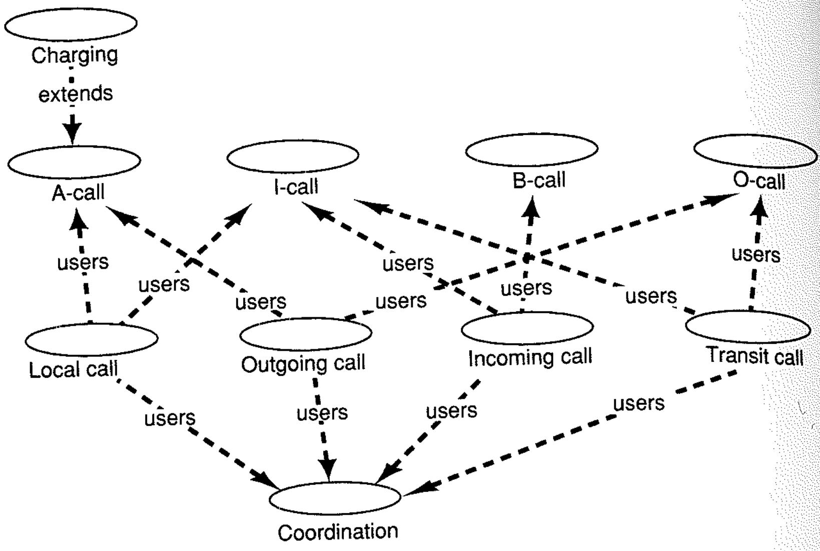


Figure 14.8 The use case model with abstract use cases.

Before going into the analysis model we will briefly discuss the topic of abstract use cases in this example.

14.3.3 Abstract use cases

When specifying the rest of the use cases we notice similarities between them. Especially in the call use cases we will have this situation. In the Local Call use case we see that there are functionalities to take care of the A-subscriber and also the B-subscriber. The A-subscriber behavior (and thus functionality) will be similar whether the A-subscriber makes an Outgoing call or a Local call. Likewise will the B-subscriber functionality be similar when the call is local and when it is an incoming call. A Transit call can thus be composed of an incoming part and an outgoing part. All use cases also have a common part where the system analyzes the digits that is received and determines how to connect the call This is also a potential abstract use case. We thus have five abstract use cases for calling, as shown in Figure 14.8. The Charging use case should only be inserted in those use cases where the A-subscriber participates. This can be modeled by having this use case extends to the A-call use case as shown in Figure 14.8.

The abstract use cases are:

- A-call: This use case includes the behavior related to an A-subscriber, that is, communicating with the calling subscriber; receiving its hook signals and dialled digits, sending a dial tone and ringing signals,
- B-call: This use case includes the behavior related to a B-subscriber, namely receiving its hook signals and ringing signal,
- I-call: This use case includes the behavior related to communication with an incoming trunk line.
- O-call: This use case includes the behavior related to communication with an outgoing trunk line.
- Coordination: This use case includes the behavior related to the analysis of the direction of the call and for the connection of the subscribers.

The reason for identifying abstract use cases is actually threefold. Firstly, it gives us a tool to reuse descriptions of use cases. Secondly, we can guarantee that parts of several different use cases will execute similarly, in this example, we do not want the A-subscriber to bother whether it is a local call or an outgoing call; his behavior should be similar. Thirdly, it will be a tool when developing the analysis model; especially when identifying the control objects.

14.4 The analysis model

We will here discuss how the analysis model is developed from the use cases. We then take one part of a use case at a time and discuss the objects needed for this part.

14.4.1 The use case Local Call

Based on the use case descriptions, we will now develop the objects in the analysis model. Here we will start looking at the use case Local Call. To identify the analysis objects we will illustrate more how a real development works. Then we go through the use case and incrementally identify the objects independently of their type. Let us take a closer look at the first section.

When a subscriber makes a call, namely lifts his handset, a stimulus is sent to the system carrying the external identity object of the calling

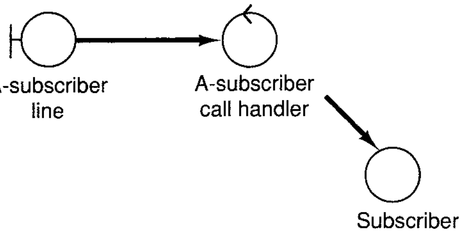


Figure 14.9 The analysis object identified from the first section of the use case Local Call.

> subscriber (the A-subscriber) which is used for identifying the subscriber in the system. The system checks whether the subscriber is permitted to make calls. If so, the subscriber is marked as busy to prevent other calls being connected to him or her Finally, a dial tone is sent to the A-subscriber

From this we see directly that some sort of interface object is needed for the A-subscriber, let us call this interface object A-subscriber line. The external identity object of the subscriber comes to this object. We then need to process this information and find the correct subscriber information in the system. Information about a subscriber is something that should be kept and handled in the system. Therefore we identify an entity object Subscriber to handle this. We now have an interface object and an entity object. There are some processing (like finding the right subscriber instance, checking whether that subscriber is allowed to make phone calls, mark him busy etc.) that should be done on the subscriber, but it does not seem suitable that the interface object should be responsible for performing these operations, since this should be done independently of the kind of interface object being used (eg. a payphone). New use cases would then require changes to the interface object. An alternative is to allocate this processing to the Subscriber object instead. However, the Subscriber object should not know which decisions are taken as the result of a test. Thus far in the use case it could be an alternative, but we will see later that there will be more processing like this. To handle this processing, we therefore initially identify a control object A-subscriber Call Handler. This control object could also be found from the abstract use cases; we there had an abstract use case A-call which corresponds very well to the control objects just identified. From the first section we thus have the objects shown in Figure 14.9.

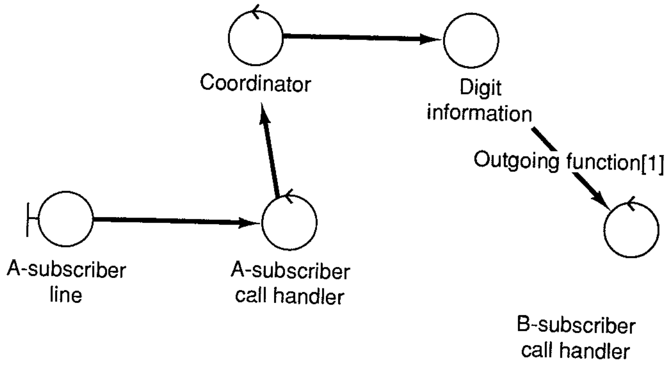


Figure 14.10 The analysis object identified from the second section of the use case Local Call

The next section in the use case was as follows.

> A digit from the A-subscriber results in the disconnection of the dial tone and storing the digit. Then the system checks in the Directory if enough digits have been received for analysis, and, if so, requests the outgoing function. In this case the outgoing function request must return a call to a B-subscriber

Here we start to analyze the digits and try to find the direction of the call. The direction of the call must be determined from some stored information concerning how to interpret the digits dialled by the subscriber. The processing of the digits is something that is common to the different kinds of call use cases we have, that is, the abstract use case Coordination. Let us therefore assign a control object Coordinator. The Coordinator receives the digits entered and checks them against a directory about how to interpret digits. This information should be handled by an entity object, we call it Digit Information. Whenever a new digit is received the digits are checked against Digit Information to see whether more digits are necessary. Digit Information then identifies that this call is a call to a B-subscriber and thus can return this to the Coordinator. How to handle a call to a B-subscriber is part of the use case B-call, thus we will have a corresponding control object for the B-subscriber also. Hence there is a control object B-subscriber Call Handler. In this section we thus have the objects shown in Figure 14.10.

The third section of the use case is as follows.

> Furthermore, if all digits have been received, the system uses the number for translating the number of the B-subscriber to the corresponding internal subscriber representation This is used to check in turn if the B-subscriber is idle and if he may receive incoming calls. If so the subscriber is marked busy and his external identity object is found Then the system operates the switches between the A- and the B-subscriber and sends a ring tone to the A-subscriber and starts ringing on the B-subscriber.

Initially we must find the internal representation of the B-subscriber. This is stored in a similar way to that of the A-subscriber, thus in an instance of the entity object Subscriber. The handling of finding, checking and allocating the B-suscriber is performed as operations on Subscriber done by B-subscriber Call Handler. From the external identity we have now found which devices (lines) should be connected for the call. Now we should find a free path and connect these two devices. This can be represented in the system by an entity object Connection that connects subscribers. All connections in the system are aggregated and handled by another entity object Network. When the connection is up, a ringing tone should be sent to both the A- and B-subscribers. This part of the use cases can now be offered by the objects shown in Figure 14.11.

The fourth section of the use case reads as follows.

> When the B-subscriber answers the call, the ringing signal to the B-subscriber and the ringing tone to the A-subscriber are disconnected.

This part can be offered by the objects already identified and will thus not yield any new objects. Hence, from this use case, we have identified the following *Interface Objects*:

- A-subscriber Line: The A-subscriber Line communicates with an A-subscriber, transforming the subscriber's actions, to stimuli recognizable to the interior of the system and vice versa,
- B-subscriber Line: The B-subscriber Line communicates with a B-subscriber, transforming the subscriber's actions to stimuli recognizable to the interior of the system and vice versa.

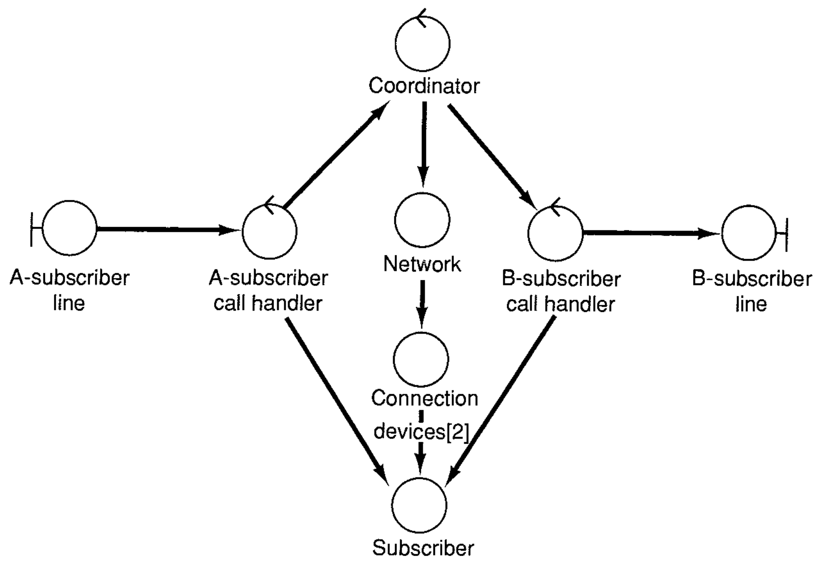


Figure 14.11 The objects offering the third section in the use case

We have also identified the following *control objects*:

- A-subscriber Call Handler: The A-subscriber Call Handler communicates with the calling subscriber; receives its hook signals and dialled digits, sends a dial tone and ringing signals and cooperates with the Coordinator.
- B-subscriber Call Handler: The B-subscriber Call Handler communicates with the called subscriber; receives its hook signals, sends ringing signals and cooperates with the Coordinator.
- Coordinator: The Coordinator cooperates with the Caller, analyzes the dialled digits and decides the direction of the call. It also cooperates with the Callee. Finally it communicates with the network for the physical connection of the subscribers.

Finally, we have the following *Entity objects*:

- Subscriber: The Subscriber entity object represents subscribers internally to the system. For each subscriber there is an instance of the entity object keeping the information of the subscriber's state, the directory number, the external physical identity and the category.
- Digit Information: The Digit Information entity object keeps the information about what kind of outgoing function there is for a number, how many digits there are in a complete number

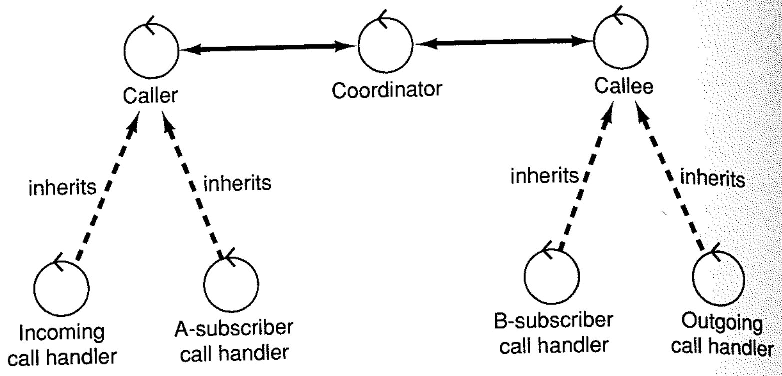


Figure 14.12 The control objects participating in the different kinds of call use cases

and what route the corresponding line belongs to. It can only exist as one instance of this object in the system.

- Network: The Network entity object includes the network switches and retains information about what devices are for the moment connected to each other. It can only exist as one instance of this object in the system.
- Connection: The Connection is an aggregation of Devices connected together in the network.

14.4.2 Further refinements

Now, we have only focused on objects in one calling use case. As the reader may guess, many of these objects can be reused in other use cases also. Furthermore, these objects will have similarities with objects participating in other use cases. This means that the object model developed here will be refined and enhanced when investigating the other use cases as well. We will not go into a detailed discussion on this here, but only give some results of such an analysis.

Similar to the control object representing the A- and B-subscriber Call Handler, we will identify corresponding control objects for the other use cases as well, giving us four different kinds of Call Handler. When regarding these further we will notice that with the caller party there are great similarities and also with the callee party. It is mainly the interaction with the Coordinator object that is similar. We will therefore get a control object view as shown in Figure 14.12.

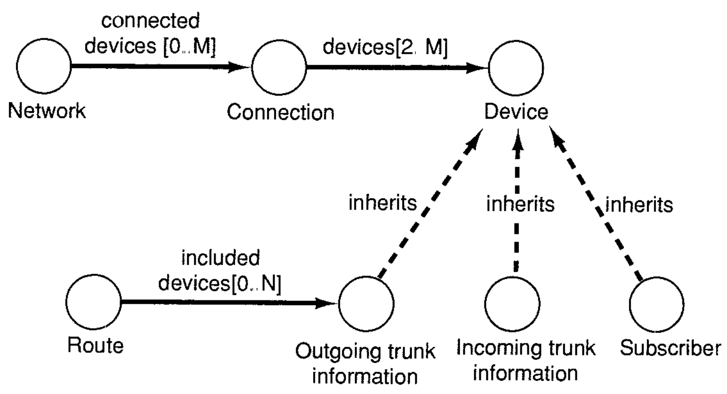


Figure 14.13 Entities participating in the different kinds of call use cases

Likewise, when analyzing the devices connected to other switching systems we will have a similar situation. We will have entity object representing these devices that are similar to the devices for subscribers, as shown in Figure 14.13. (The reason that we have the cardinality [2..M] on the aqcuaintance association between Connection and Device is that we actually may allow more than two parties to be involved in a call, a multiparty call.) The knowledge of what the Outgoing Trunk Information objects is also held by a Route object.

14.4.3 The use case Subscription Changes

The use case Subscription changes is far more simple than the Local Call use case The basic course in the description reads:

> A new subscriber is connected by an Operator through a command resulting in a form presented on the screen. The Operator completes the information requested, namely a directory number, the subscriber's category and the external identity of the new subscriber. The information is stored in the system Finally, the operator is informed of the success of the subscription.

This part thus creates instances of the object Subscriber as identified in the Local Call use case. We also have a form on which to fill in the information. This form will be a direct mapping onto the entity object, which is why there is no need for a control object

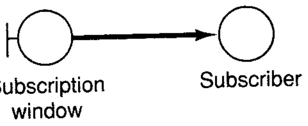


Figure 14.14 The objects offering the use case Subscription Changes.

to handle this part. We thus will have the object shown in Figure 14.14. These objects will also be sufficient for the alternative course, namely when a subscriber should be disconnected to the system.

14.4.4 Subsystems

In this manner all objects required to offer the use cases are identified. When all objects have been identified and described we will have a large number of objects that we are not able to handle at one level. These objects are therefore grouped into different subsystems.

From these use cases we have reviewed here we will have the objects shown in Figure 14.15. Note that we have a direct pair of interface objects and entity object; one interface object for Incoming Trunk Line and one entity object for Incoming Trunk Information. The reader may ask, why not also place the entity object functionality in the interface object? This is fully possible, but again we have the robustness motive for modeling as we do. There are certain kinds of information for each associated line that are independent of what type of line it is. However, there are several different types of lines and thus many different types of interface objects to interface the lines. All of these interface objects will have the same information functionality, and so this is modeled in the entity object. In fact we have an association between each pair of interface object and entity object, an association not shown in the Figure 14.15.

We see that this gives a large number of objects and also a large number of associations. For true systems, the number of objects must be grouped in order to get a manageable structure Subsystems are the normal grouping concept of OOSE. Subsystems should group objects with related functionality. We see that some objects concern traffic control. Those will form the Traffic Control Subsystem, which includes the following objects: Coordinator, Caller, Callee, Digit Information, Network and Connection. Some objects are involved with the lines to the subscribers and to other switches. These constitute the Trunk Subsystem which includes Outgoing Trunk Line, Outgoing Call Handler, Outgoing Trunk Information, Incoming Trunk Line, Incoming Call

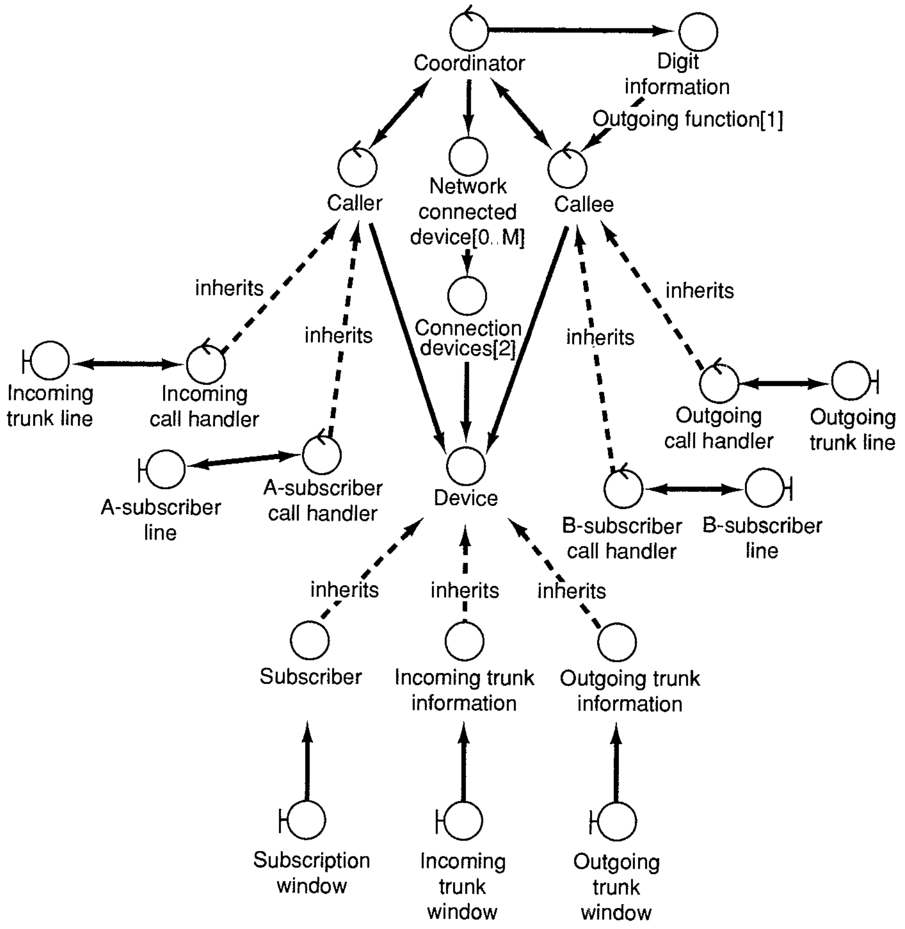


Figure 14.15 Part of the analysis model for the telecom example.

Handler, Incoming Trunk Information, A-subscriber Line, A-subscriber Call Handler, B-subscriber Line, B-subscriber Call Handler and Subscriber. We will also have one subsystem for Operation and Maintenance which includes objects like Subscription Window, Incoming Trunk Window and Outgoing Trunk Window.

14.5 The design model

In the design model we will adapt and refine the analysis model so that we can implement the system in a seamless and straightforward way. The adaption will be made from the current implementation

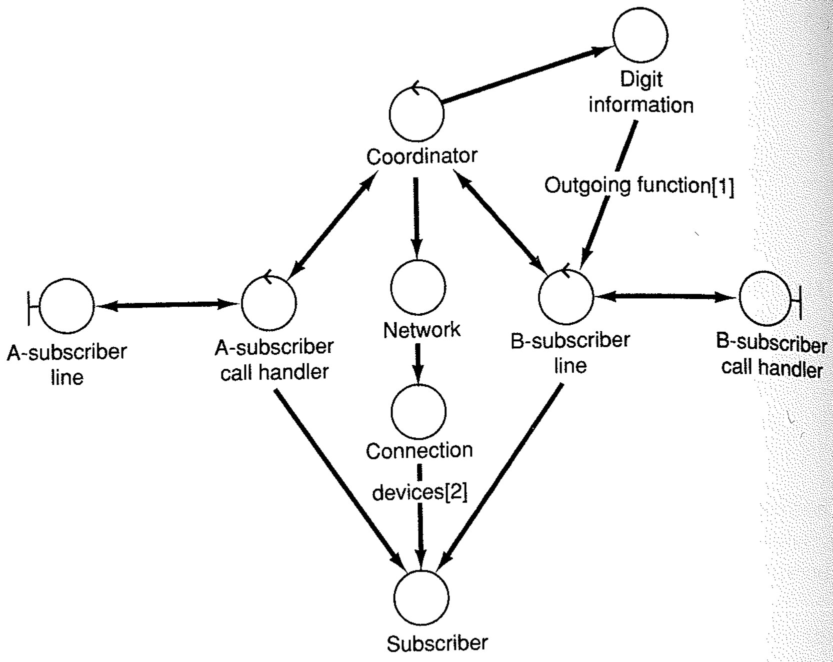


Figure 14.16 The use case view over analysis object in the Local Call use case.

environment and the refinement will be made by explicitly expressing each stimulus sent between the objects.

We will continue to focus on the use case Local Call and see how we incorporate the implementation environment in the object model offering this use case. We will then continue to refine this object model and also look at how the implementation is done for some objects. We will use Smalltalk as our implementation language.

14.5.1 The block structure

Let us start with the block structure. The first attempt at such a structure is to do a one-to-one map from the analysis model, that is, for each analysis object we assign one block. The use case view of the analysis model for this use case looks like Figure 14.16.

The ideal design model would be to start to convert the analysis objects into blocks. This means that the first design model would be as shown in Figure 14.17.

Let us discuss some of the blocks. The interfaces A- and B-subscriber lines are actually interfaces against the hardware (lines)

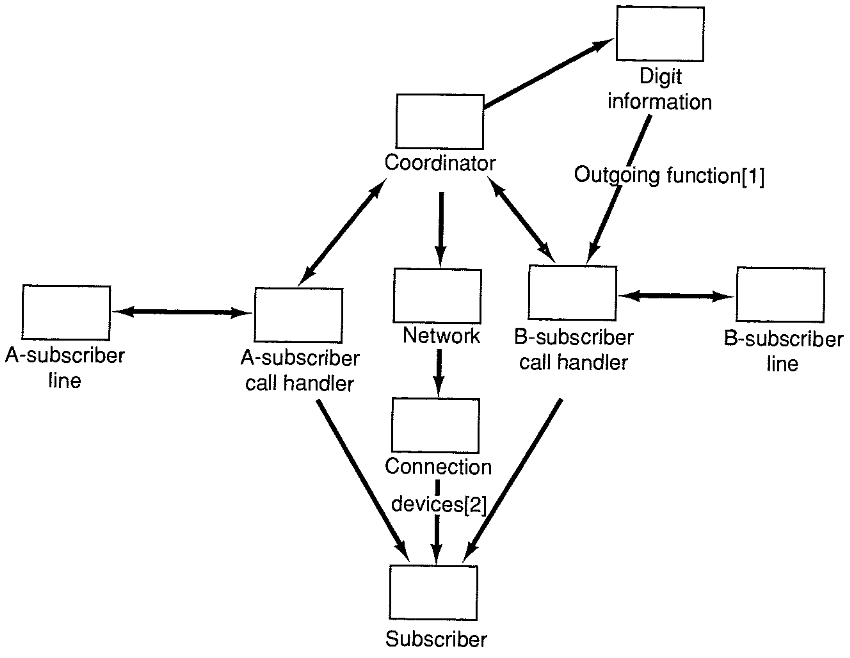


Figure 14.17 The first attempt at a design model is a direct conversion of analysis objects into blocks.

going to every subscriber attached to the system. We do not discuss the hardware part here, but in a real switching system there is only one circuit handling I/O to each subscriber. Therefore we must develop software that handles this interface. It is thus not interesting to differentiate here between different kinds of lines when preparing for the implementation. We therefore only have one block Subscriber Line. This means that we merge the A- and B-subscriber line to only one block. However, in every use case there will be two different instances of this class participating in the use case. It can therefore be interesting to have both these instances shown in the design model when approporiate.

The different kinds of Call Handlers will have different behavior, though. Therefore it is not possible to join these to only one block, both are needed. Recall that during analysis we discussed whether to place this behavior in the interface objects or in the entity object Subscriber. If we had done the latter, it would be harder to join the interfaces in only one block or we would have an unnecessarily complex Subscriber.

The network block will have control over all switches connected in the system. There will only exist one instance of this class in the

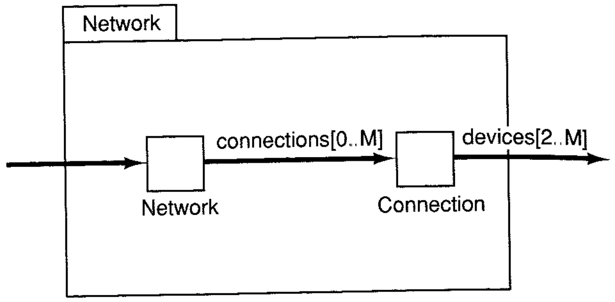


Figure 14.18 A first sketch of the inside of the Network block.

system. However, the connection through the switches was modeled by an entity object Connection in the analysis model, which gave rise to a corresponding block. The network can very well be implemented with, for instance, an array or a list that, in each element, has two (or more) references to a Subscriber object. Thus Connection can actually be encapsulated in the Network block to hide the actual implementation of the representation. We can therefore eliminate Connection from the Design model and have a block in terms of classes of Network as outlined in Figure 14.18.

The Digit Information, which exists in one instance only in the system, will match the number dialled and find which outgoing function should be offered. It will also check that the number dialled is a valid number and, for outgoing calls, also find an outgoing route. The finding of the outgoing function can be done after only a couple of digits (usually the first three digits in a telephone number determine which switch to turn to), while we may need the entire number to determine whether it is a valid number. To handle this matching, a tree structure is appropriate, where each level is determined by a digit. That is, we start in the root and when the first digit is dialled we can descend one level in the tree to the appropriate node and from there analyze the next digit. At some node we will encounter the fact that it is a local call or a call to another exchange. When a leaf is reached we can determine how to handle this call, namely what outgoing function to use. In the leaf we thus have information on the outgoing function, the length of the entire catalog number and how to direct the call The outgoing function refers to a B-subscriber Call Handler if the call is internal to the exchange, or to an Outgoing Call Handler if it is to a remote B-subscriber. If the call is internal we have a Subscriber object, if the call is outgoing we are refered to a Route object. The Digit Information block can thus be designed in a similar manner to the Network block,

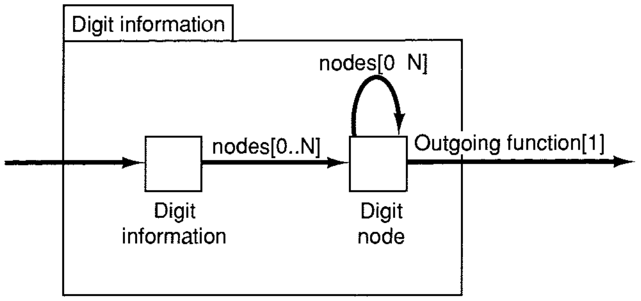


Figure 14.19 A first sketch of the interior of the Digit Information block

but with a tree structure of Digit Nodes. A node could thus possibly refer to an outgoing function and also a route (not used in the use case studied). The Digit Information block design thus looks like Figure 14.19.

The matching of the received digits to the digit tree is also worth discussing a while. The Coordinator block will receive the digits from the A-Subscriber Call Handler. These digits should be stored at the Coordinator and when a new digit has been received, a new check should be made against the Digit Information. Additionally, the entire number dialled should then be passed to the B-subscriber Call Handler. This indicates that we need some form of unit to store the dialled number in. We thus identify a class Register to store this information. We could here use some kind of string, collection or some other class from the library, but we will need a little more functionality from this unit when it is sent to different objects as a parameter. The Register can however be encapsulated in the Coordinator block, since it is mainly this block handling it. The Coordinator will thus have a first sketch as shown in Figure 14.20.

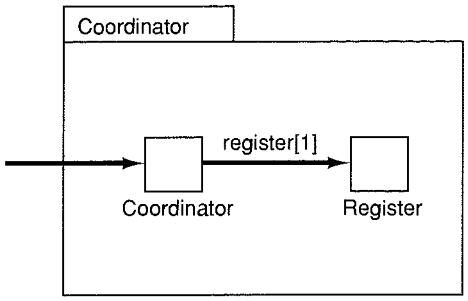


Figure 14.20 A first sketch of the design of the Coordinator block

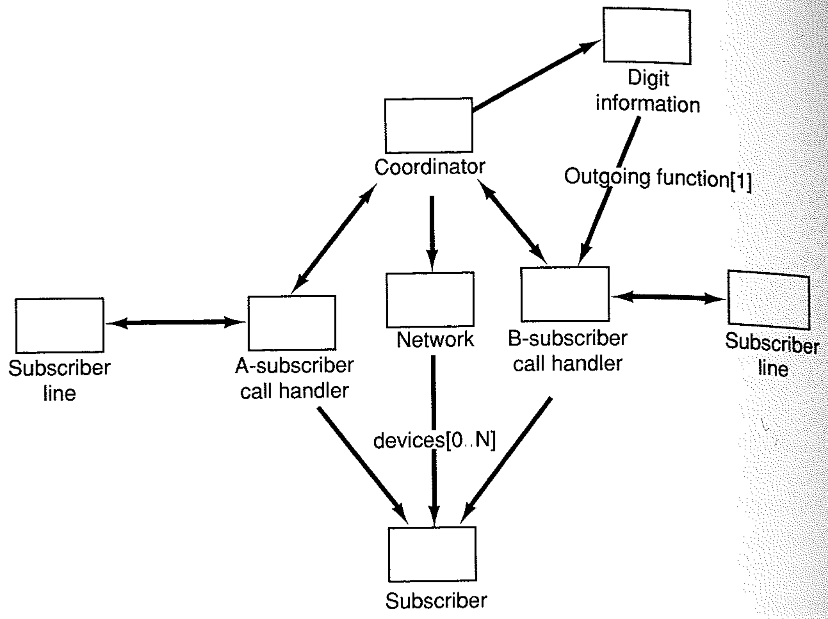


Figure 14.21 The refined design model.

This analysis of how to handle different aspects of the design can, of course, not all be decided at once Further, many aspects will not be noticed until the model has been further detailed. This work thus has a strongly iterative character We have here only given some examples of what such a refinement can invoke for the design model. We have thus far reached a design model as shown in Figure 14.21. Note that we have here included the Subscriber Line block twice to make Figure 14.21 easier to draw. This should be done if it clarifies things. Semantically, however, it is the same Subscriber Line block.

14.5.2 The process structure

A telecommunication switching system is a typical example of a real-time system in which processes are a natural ingredient. Let us now, from the design model developed, discuss how processes should be identified and handled in this switching system. Taking the guidelines given in Chapter 9, we will start identifying candidate processes. There it was stated that one should initially focus on the interface blocks and see which of these could receive events in a non-deterministic way. In this example, and now we only focus on the Local Call use case, we see that actually both the A- and B-subscriber can generate events possibly independent of each other. We can therefore allocate one process to handle each party of the Local Call. Let us call these processes the A- and B-processes. In these processes we allocate the behavior of the appropriate instances of Subscriber Line, and of A-subscriber Call Handler and B-Subscriber Call Handler respectively Additionally, we need something to handle the processing of these events independently of the events, typically handled by the Coordinator. This is also a process candidate including the Coordinator block. Instances of these processes will be created for each Local Call. (How this is actually done is dependent on the operating system used, and since it is not a subject of this book, we will not go into detail on this.) The blocks Network, Digit Information and Subscriber will be shared by several instances of the processes just discussed. This was also discussed in the chapter about real-time, where typically entity object must often be protected from simultaneous access by some form of mutual exclusion. This can typically be implemented with semaphores, monitors or even separate processes, or some other feature existing in the programming language or operating system. Here we assume that they will be protected by some kind of semaphore.

A process view of the design model may thus look like Figure 14.22. Note that here we have been able to fully allocate each block (instance) to a certain process These processes, as we will see, are explicit in the interaction diagrams.

14.5.3 Use case design

We are now ready to describe how the interaction between the different objects will look in the Local Call use case. All of the blocks previously discussed will participate in this use case. We will thus specify each stimulus sent between the objects. Since different processes are involved, we must also differentiate as to what type of stimulus is sent; messages for normal operation invocation (closed arrow) and signals for interprocess communication (open arrows).

To show explicitly how the use case descriptions are used, we will partition the interaction diagram as for the section in the descriptions. This is not normally done in real systems, but is done here for clarity

Let us start with the first section of the use case description.

When a subscriber makes a call, namely lifts his handset, a stimulus is sent to the system carrying the external identity object of the calling

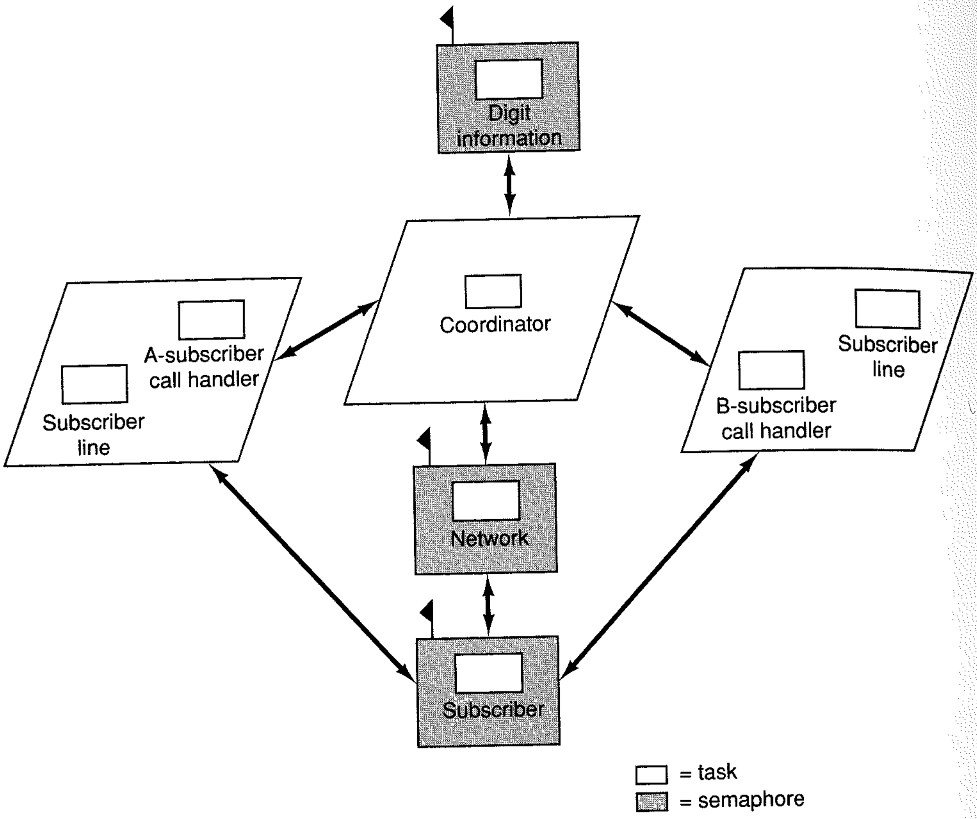


Figure 14.22 A process view of the design model involved in the Local Call use case.

> subscriber (the A-subscriber) which is used for identifying the subscriber in the system. The system checks if the subscriber is permitted to make calls. If so, the subscriber is marked as busy to prevent other calls being connected to him or her Finally, a dial tone is sent to the A-subscriber.

From this we will be able to define the stimulus sequence as shown in Figure 14.23. The correct subscriber will normally be stored in the database. We have not explicitly discussed the database issues in this example. However, we have here different possibilities. We could add a new object which represents the database to which we would send a stimulus createObjectFromDBref aDBref. This would then return an instance of the correct subscriber. However, the handling should rather be encapsulated in the Subscriber object since we do not want other objects to be dependent on how it is actually done, for reasons of robustness. This encapsulation is implemented using the framework discussed in Chapter 10.

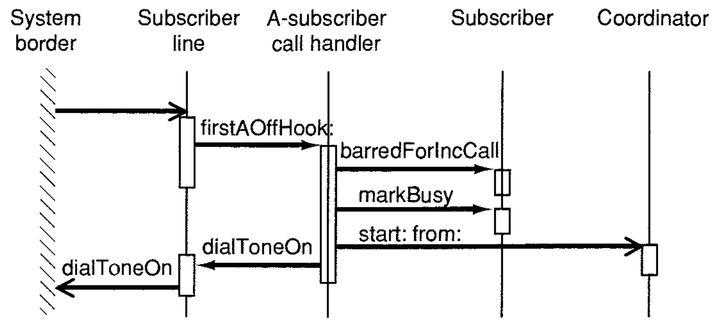


Figure 14.23 Interaction diagram for the first part of the use case.

We thus have defined several stimuli sent between the blocks. Note that when we specified the use case we did not have any idea which blocks should participate in the use case. The domain objects were then our logical view of the system. Neverthless, it is quite easy to allocate the behavior over the objects and the stimuli come naturally. Let us continue with the next part of the use case (see Figure 14.24).

A digit from the A-subscriber results in the disconnection of the dial tone and storing the digit. Then the system checks in the **Directory** if enough digits have been received for analysis, and, if so, requests the outgoing function. In this case the outgoing function request must return a call to a B-subscriber.

We thus have now found the correct outgoing function, in this case a B-subscriber Call Handler The stimulus enough: checks whether enough digits have come to determine which outgoing function is to be selected. The colon indicates that the stimulus carries a parameter (we use the notation used in the current programming language, which is Smalltalk in this case). The parameter for enough: is the current instance of Register where the received digits are stored Digit Information uses this instance to check the number against the digit tree. Now we are just waiting for the final digits before the call can be connected. This is described in the next section.

Furthermore, if all digits have been received, the system uses the number for translating the number of the B-subscriber to the

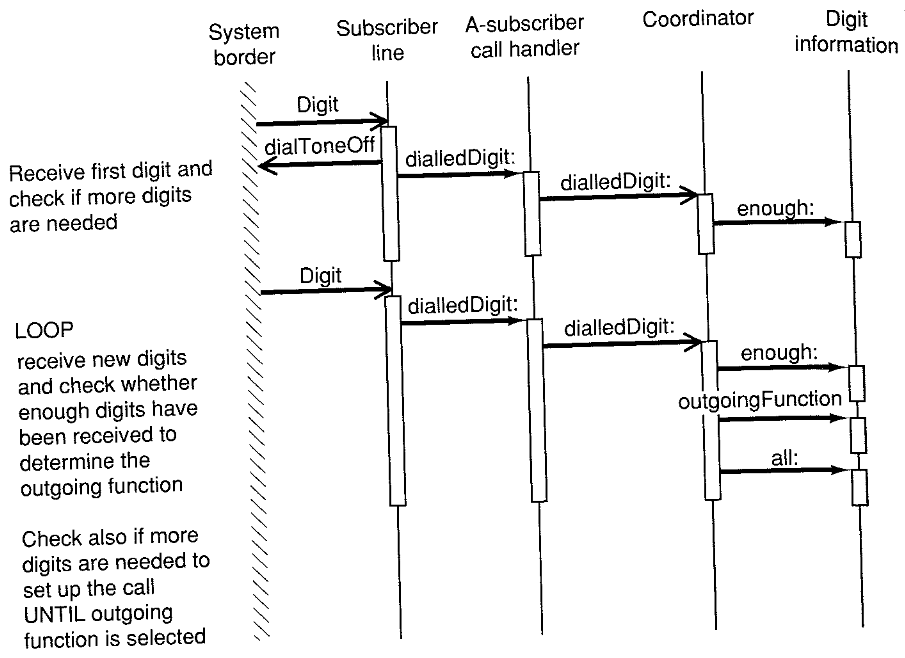

System border
Subscriber line
A-subscriber call handler
Coordinator
Digit information
Digit
dialToneOff
dialledDigit:
dialledDigit:
enough:
Receive first digit and check if more digits are needed
Digit
dialledDigit:
dialledDigit:
enough:
outgoingFunction
all:
LOOP
receive new digits and check whether enough digits have been received to determine the outgoing function
Check also if more digits are needed to set up the call
UNTIL outgoing function is selected

Figure 14.24 Continuation of the interaction diagram for Local Call.

> corresponding internal subscriber representation. This is used to check in turn if the B-subscriber is idle and if he may receive incoming calls. If so the subscriber is marked busy and his external identity object is found. Then the system operates the switches between the A- and B-subscribers and sends a ring tone to the A-subscriber and starts ringing on the B-subscriber

Here the *Coordinator* initiates the processing of handling the B-subscriber (see Figure 14.25). He is marked as busy and then the connection between A- and B-subscribers is connected and dial tones are given at both ends. In the next section of the use case the B-subscriber answers his phone.

> When the B-subscriber answers the call, the ringing signal to the B-subscriber and the ringing tone to the A-subscriber are disconnected.

Here we have used a notation where we actually have two lines representing the system border (see Figure 14.26). One represents the A-subscriber and one represents the B-subscriber. With this

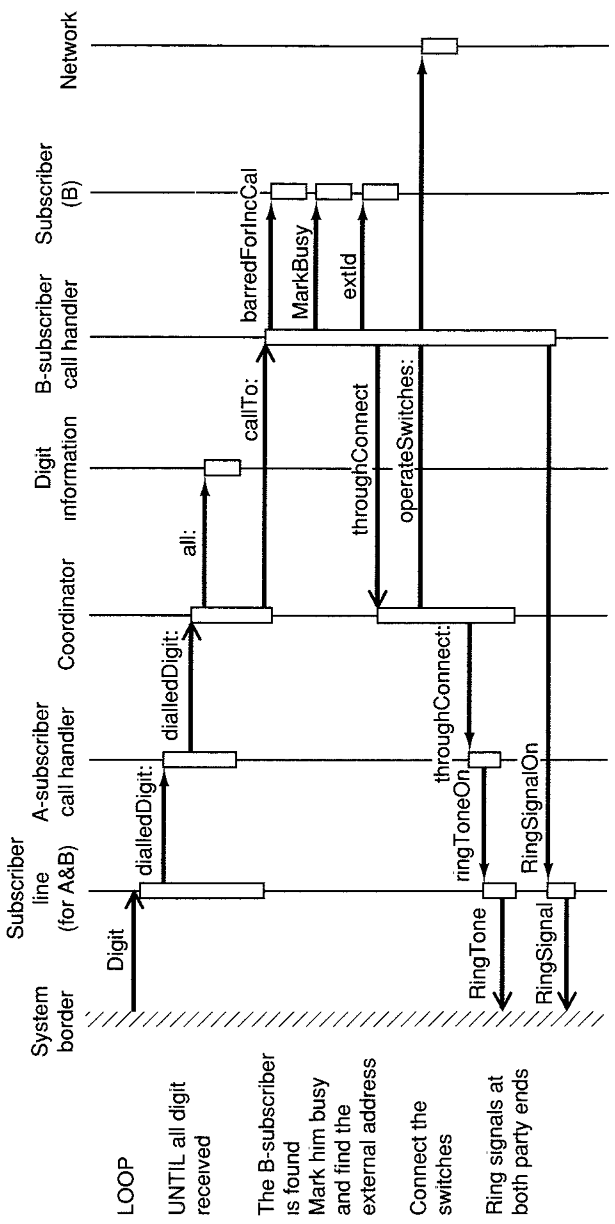


Figure 14.25 Continuation of the interaction diagram.

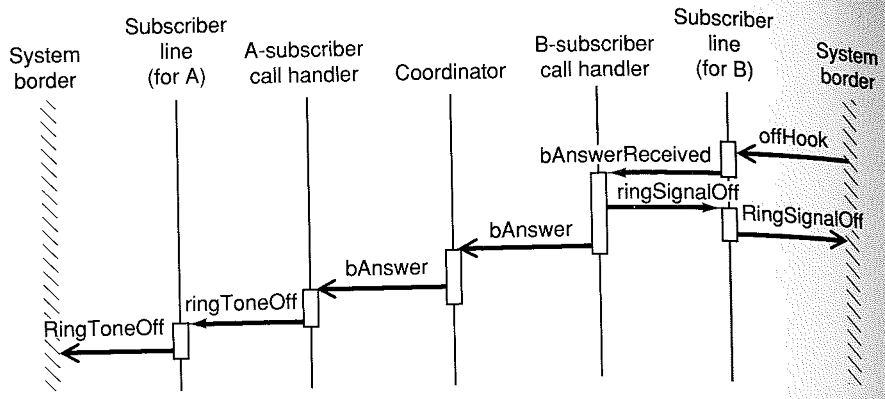


Figure 14.26 The part of the use case when the B-subscriber answers the telephone call

notation we can more clearly see where and to whom the stimuli are really sent. In the same spirit we also have two lines for Subscriber Line. One represents the instance of the A-subscriber and one represents the instance of the B-subscriber. This notation is also used in order to gain more clarity in the interaction diagram. However, remember that the aim for doing the interaction diagrams is to obtain the interfaces of the objects. Since both Subscriber Lines belongs to the same class, we will define stimuli on the same class. Therefore this notation will not give us any more (or less) information, and the only reason is for clarity in the diagrams. Let us look at the last part of the use case (see Figure 14.27).

Finally, when the B-subscriber clears the call, the new event is registered in the system Then, when the A-subscriber also clears the call, the system marks both subscribers idle and terminates the call by releasing the switches between the two subscribers.

The basic course of the use case has thus been designed. The alternative courses are designed in the same manner This will give us the stimuli sent in this use case

14.5.4 Block interfaces

When all use cases have been designed we will thus also have the interfaces to each block. We will here focus on the Coordinator block.

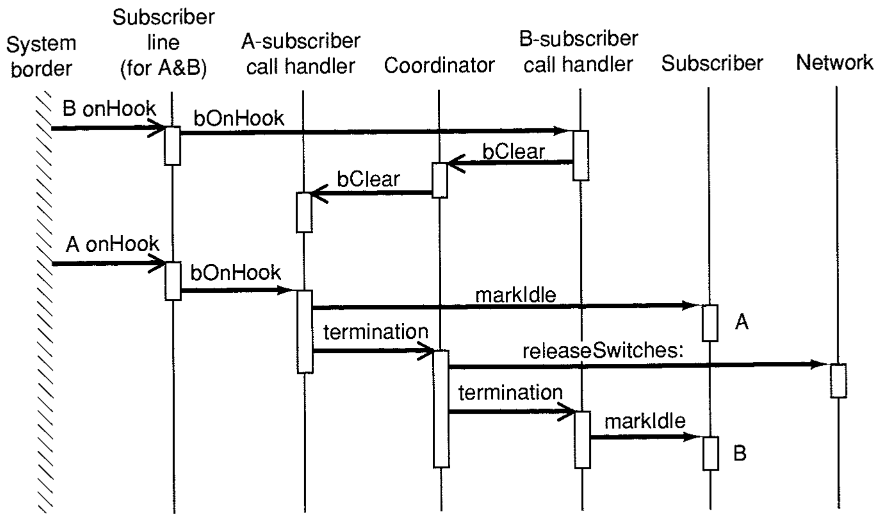


Figure 14.27 The final part of the use case Local Call.

To get the complete interface of the object we investigate all interaction diagrams in which Coordinator participates. From the use case just designed we will have the following interface for the Coordinator object:

start aId from aCaller	'aId – ShortNumber, aCaller – Caller'
dialledDigit: aDigit	'aDigit – ShortNumber'
throughConnect: bId	'bId – ShortNumber'
bAnswer	
bClear	
termination	

Here ShortNumber and Caller denote the intended types of parameters. This is not the only information that we can extract from the interaction diagrams. We also have a sense of the objects' states and what will happen when these stimuli are received. Thus, before going to implementation, we can describe the objects with a state transition graph, describing the dynamic behavior of the object. We continue to focus on the Coordinator block. From the rules given in Chapter 8, we know that each input stimulus is a potential state transition With the notation described earlier we will have the state-transition diagram shown in Figure 14.28.

The state transition diagram gives a picture of the behavior of an object. The figure only includes the basic course for one use case As the other courses and more use cases are analyzed we will refine

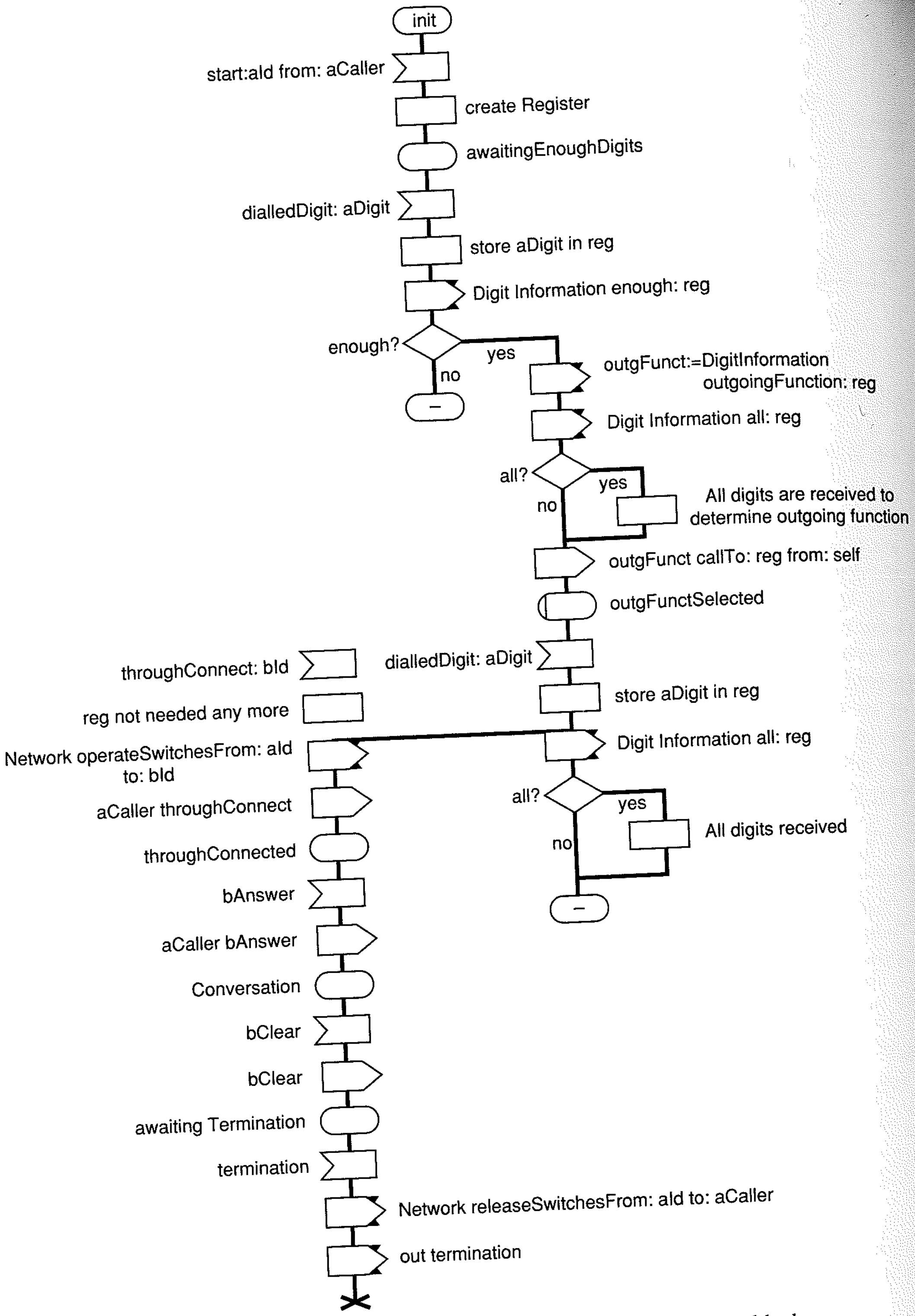


Figure 14.28 A state transition diagram for the Coordinator block.

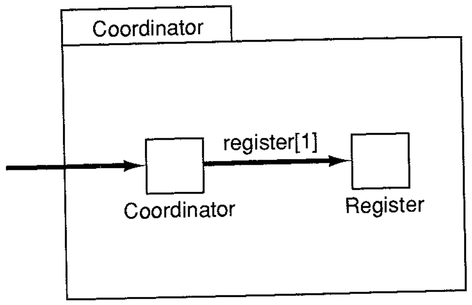


Figure 14.29 The overall design of the Coordinator block

this diagram. Hence the diagram will iteratively be further refined as more specified behavior is incorporated in the diagram. In this case the basic structure will, however, remain as above; some more alternatives will be added though. The reader can try also to design the alternative courses of the use case and incorporate this behavior in the diagram.

We will now continue to discuss the implementation of the Coordinator block.

14.6 The implementation model

Since the system should be implemented in Smalltalk we have used Smalltalk syntax in the design model. This will ease the transition to the actual code, as we will see. We will here focus on the Coordinator block.

When viewing the state transition diagram in Figure 14.28, the first question is how to implement the states. In most object-oriented language, this must be implemented explicitly by the programmer. However, there exists class libraries to handle state machines. We will not here assume any such predefined classes, but instead illustrate two alternatives for such an explicit implementation.

The first design of the Coordinator block looked like Figure 14.29. This design seems to be appropriate still; we have not encountered anything that will make this design implausible. In the state transition diagram above, the behavior specified should mainly be implemented by the Coordinator class. To implement and handle the different states, we can do this in either of two ways. The first and most obvious solution is to have an instance variable for the state that we test upon whenever we receive a stimulus. The second is to implement different operations for different states and, when a stimulus is received, a concatenation of the state and stimuli name will give us the correct method to invoke. We look at both these solutions. In the code we have also included the alternative courses. The fork message is used to handle processes and signals in Smalltalk. Signals do not return anything in Smalltalk and this is why we return nil everywhere this is used.

14.6.1 If-clause implementation

```
class:  Coordinator
superclass:  Object
instance variable names:  'in out inId outId reg outgFunct state'
```

The instance variables used are

● in,	refers to the caller,
● out,	refers to the callee,
● inId,	external id for the caller,
● outId,	external id for the callee,
● reg,	stores the digits received,
● outgFunct,	holds the current outgoing function,
● state,	the state of the call.

```
class methods
start: aId from: aCaller
  'Create a new Coordinator and connect it with the corresponding
  Caller.'
  ↑ super new start: aId from: aCaller

instance methods
start: aId from: aCaller
  'Initialize the coordinator.'

  inId := aId
  in := aCaller
  reg := Register new
  state := 'awaitingEnoughDigits'
dialledDigit: aDigit
  'Receive a dialled digit, store it and and analyze the received digits.
  If enough digits have been received, create an instance of the
  Callee.'

  state = 'awaitingEnoughDigits'
    ifTrue:
      [reg store: aDigit
```

```
        (DigitInformation enough: reg)
          ifTrue:
            [outgFunct := DigitInformation
                   outgoingFunction: reg
            (DigitInformation all: reg) ifTrue: [reg allReceived ]
            [out := outgFunct callTo: reg from: self] fork
            state := 'outgFunctSelected']
        ↑ nil]
    state = 'outgFunctSelected'
      ifTrue:
        [reg store: aDigit
        (DigitInformation all: reg) ifTrue: [reg allReceived]
        ↑ nil]
```

throughConnect: bId

'If the call is successfully through-connected with the Callee, the network switches are connected and the Caller is notified '

```
    state = 'outgFunctSelected'
      ifTrue:
        [outId := bId
        reg notNeededAnyMore
        Network operateSwitchesFrom: inId to: outId
        [in throughConnect] fork
        state := 'throughConnected'
        ↑ nil]
```

bAnswer

'The Caller is notified when the Callee has answered the call '

```
    state = 'throughConnected' | (state = 'awaitingTermination')
      ifTrue:
        [[in bAnswer] fork
        state := 'conversation'
        ↑ nil]
```

bClear

'The Caller is notified when the Callee has cleared the call '

```
    state = 'conversation'
      ifTrue:
        [[in bClear] fork
        state := 'awaitingTermination'
        ↑ nil]
```

termination

'When the call is terminated the network switches are disconnected and the Callee is notified '

```
    state = 'awaitingEnoughDigits'
      ifTrue:
        [state := 'terminated'
        ↑ nil]
    state = 'outgFunctSelected'
    ifTrue:
        [reg notNeededAnyMore
        [out termination] fork
        state := 'terminated'
        ↑ nil]
state = 'throughConnected'   (state = 'awaitingTermination')
    ifTrue:
        [[out termination] fork
        Network releaseSwitchesFrom: inId to: outId
        state := 'terminated'
        ↑ nil]
```

14.6.2 Perform: clause implementation

```
class:   Coordinator
superclass:   Object
instance variable names:   'in out inId outId reg outgFunct state'
```

class methods

start: aId from: aCaller

```
  'Create a new Coordinator and connect it with the corresponding
  Caller '

↑ super new start: aId from: aCaller
```

instance methods

start: aId from: aCaller

```
  inId := aId
  in := aCaller
  reg := Register new
  state := 'awaitingEnoughDigits'
```

dialledDigit: aDigit

```
  'Receive a dialled digit '

  self perform: state , 'dialledDigit:' with: aDigit
  ↑ nil
```

throughConnect: bId

```
  'The call has been successfully through-connected with the Callee.'

  self perform: state , 'throughConnect:' with: bId
  ↑ nil
```

bAnswer

```
  'The Callee has answered the call '

  self perform: state , 'bAnswer'
  ↑ nil
```

bClear

```
  'The Callee hs cleared the call '
  self perform: state , 'bClear'
  ↑nil
termination
  'The call has been terminated '

  self perform: state , 'termination'
  ↑nil

awaitingEnoughDigitsdialledDigit: aDigit
  'Store the received digit  If enough digits have been received, create
  an instance of the Callee '

  reg store: aDigit
  (DigitInformation enough: reg)
    ifTrue:
      [outgFunct := DigitInformation outgoingFunction: reg
      (DigitInformation all: reg) ifTrue: [reg allReceived]
      [out := outgFunct callTo: reg from: self] fork
      state := 'outgFunctSelected']
awaitingEnoughDigitstermination
  'The call has been terminated before enough digits had been
  received '

  state := 'terminated'
outgFunctSelecteddialledDigit: aDigit
  'A new digit has been received after the outgoing function has been
  selected (the Callee) '

  reg store: aDigit
  (DigitInformation all: reg) ifTrue: [reg allReceived]
outgFunctSelectedthroughConnect: bId
  'If the call is successfully through-connected with the Callee, the
  network switches are connected and the Caller is notified '

  outId := bId

  reg notNeededAnyMore
  Network operateSwitchesFrom: inId to: outId
  [in throughConnect] fork
  state := 'throughConnected'
outgFunctSelectedtermination
  'The call has been terminated before it had been through-connected '
  reg notNeededAnyMore
  [out termination] fork
  state := 'terminated'
throughConnectedbAnswer
  'The Caller is notified when the Callee has answered the call '

  [in bAnswer] fork
  state := 'conversation'
throughConnectedtermination
  'The call has been terminated before the Callee had answered '
```

```
    [out termination] fork
    Network releaseSwitchesFrom: inId to: outId
    state := 'terminated'

conversationbClear
    'The Callee has cleared the call '

    [in bClear] fork
    state := 'awaitingTermination'
awaitingTerminationbAnswer
   'The Callee has lifted his handset again '

    [in bAnswer] fork
    state := 'conversation'
awaitingTerminationtermination
    'The call is terminated '

    [out termination] fork
    Network releaseSwitchesFrom: inId to: outId
    state := 'terminated'
```

A comparison of these two implementations show that the first gives fewer but longer methods. The second implementation gives more methods and this solution also has a more direct map from the state transition diagram to the code, being more structured with respect to the possible use case. Note that the second solution is not possible in a language where it is not directly possible to evaluate strings or values, like, for instance, in C++ or Ada. Which of these alternatives should be chosen is up to the developer. We have included both alternatives here to illustrate that one design model can be implemented in various ways. Hence several decisions remain for when the actual implementation is done.

We see here that the transitions between the different models have been seamless and do not involve any major distortions. This is a typical feature of OOSE, namely to develop incrementally the system from initial informal and ideal sketches to a more formal model, ending the transition with the actual code.

15 Managing object-oriented software engineering

15.1 Introduction

To introduce a new development process into an organization is seldom painless. To get all the people involved to accept all the ideas in the new process involves a lot of work from pedagogical methodologists.

It is thus essential to get the development staff adopting an organized way of working and thinking, and this as fast and painlessly as possible. No one will accept any delays caused by a new process being introduced. On the whole it is the standard problem of getting an old organization to adopt a new way of working. The organization is often, at least initially, a specific project. That is often the best form for system development.

In this chapter we will discuss some of the experiences we have had introducing and working with Objectory in real projects during the last few years. To date (Summer of 1991) we have used Objectory in about 15 projects of varying size (3–50 man years). We will first discuss some of the preparations necessary for introducing a new process and then performing projects.

15.2 Project selection and preparation

15.2.1 Introducing a new development process

Most organizations we have worked with that have chosen to use a new development process already have a significant method maturity. Most have worked with one or several development methods earlier, and they have also developed a sound skepticism against introducing new technology or new ways of working too fast.

Five levels of process maturity of a software development organization have been defined by people at the Software Engineering

Institute, see Humphrey (1989) or Yourdon (1990). This classification is gaining wide acceptance. The levels are as follows.

(1) **Initial level** No documented method is being used. Every software developer is doing it his or her own way.

(2) **Repeatable level** A method exists, but has not been formalized or written down However, there exists a consensus of 'the way we do things around here'. Often this level is reached by long experience of system development. However, when new methods and tools are introduced, the organization can very well be thrown back to level (1).

(3) **Defined level**. A formal, documented process of developing systems exists The process is continuously refined by a software process group.

(4) **Managed level**. Formal measurements of different characteristics of process and product are continuously performed. Not only time and cost are measured, but also productivity, effectivity, quality and so on.

(5) **Optimizing level**. The measurements from level (4) are systematically used as feedback to optimize the process.

There also seems to be general agreement that an organization is not ready to adopt new methods or tools effectively unless it is at or above level (3). Investigations in the late 1980s show that 85% of the large software development organizations in the USA are still at level (1), 10–12% are at level (2) and only about 3% were found at level (3). In 1990 no organizations were found at levels (4) or (5) in the USA.

These observations from the USA perspective are very interesting. However, from our (European?) perspective, they seem a bit odd. Method maturity, we believe, seems greater in Europe. In practice, all of the projects that we have been involved in are with organizations at levels (2) or (3). About half of the projects have been in organizations at level (3), (which is said to be only 3% of the large USA data processing organizations late 1980). Our observations are also that certain organizations at level (2) have been matured to adopt the new technology. An especially important observation that we have made is that there can be a significant difference between the organization maturity and the maturity of the individuals.

Hence a big effort and much preparation are needed when introducing a new way of working. The new way means that the organization needs retraining and needs to develop new routines. This will, in a period of transition, give a lower productivity. If an organization has maintenance responsibility for several systems which have evolved during their life cycle, a consideration of each and every system should be made as to whether it really pays to change technology for that particular system. With a matured system, which requires limited resources for maintenance, and that in the near future will be replaced by a new system, there is probably no reason for changing the maintenance strategy.

A practical way is often to introduce the new way of working stepwise. A suitable way is to select a smaller project for new development or a limited re-engineering of an existing system. We will talk more of how to select the right project in the next section.

There are certain factors that increase the possibility of making the transistion to the new way of working successful.

(1) The selection of a new development method is a very important strategic decision which must be supported by upper management. The entire organization should be aware of the importance of the decision

(2) The first development project using the new method will be exposed to much attention. Thus there is a great need for success. There will always be critics who will exploit every sign of failure to discredit the new method. The first project must thus be selected with much care and must have all the attention and resources needed to guarantee an appropriate result.

(3) The people working in the selected project must have a feeling of a positive change. This requires, for instance that they must have sufficient training that they feel comfortable with the new situation. Further, they should have tools (e.g. CASE support) which stimulate the new way of working.

(4) Introduce the method prior to any CASE tool supporting the method. Method and tool are distinct, but both are desirable. You can use a method without a tool, but not a tool without a method. Far too many people have a hard time in realizing the difference between methods and tools.

(5) The new way of working must be integrated with other routines, for example project and product management. This integration should be ready when the new order of working is introduced widely.

(6) Have reasonable expectations of the first project. It will take some years to reach a significant increase in productivity, but increase in quality will usually come from the start. OOSE will normally not be profitable in one project; the first project will be more expensive than traditional technologies, but the quality

will be better. The profits will come in subsequent projects with experienced people and also in maintenance of the first system.

(7) Do not have high expectations of components and reuse initially. The benefits will come in two or three years.

15.2.2 Selecting the first project

The first project using a new working method is often an evaluation project. It is not only a matter of evaluating the new method as such – there may be well documented experiences from other organizations – it is also a matter of evaluating the method used in this particular organization. This involves examining how inclined the staff are to changing their working method and what is involved in doing it effectively. Therefore how the first project is selected and which staff should be involved in it are essential considerations.

We suppose that the overall decisions are already made. By this we mean that upper management believes that the selection of method is a crucial and a strategic decision which they are ready to support both financially and with careful attention. There should be a person in the upper management who has a special interest in following and supporting the project. This person we call a **sponsor**. He or she should have a good professional reputation and be respected in the organization.

When selecting the first project there are certain things to consider. Generally, it should have as optimal conditions as possible for the project, to evaluate the method without any unneeded disturbances such as from shortage of staff or problems in defining the system responsibility. We summarize this in the following recommendations.

(1) Select a real project that is important, but not with a tight time schedule or any other hard constraints.

(2) Select a problem domain that is well known and well defined.

(3) Select people with experience from system development who have a positive view of changes. The management should have confidence in them.

(4) Select a project manager with a high degree of interest in the task

(5) The staff should work full time within the project and not be disturbed by other projects.

Table 15.1 The need for education when introducing a new development process: × = necessary, + = preferably

role / issue	Project manager	Analyst	Constructor	Tester	QA	Upper management
Concepts	×	×	×	+	×	+
Project management	×					
Overview	×	+	+	+	×	+
Analysis	+	×	×	+	+	
Construction	+	+	×	+	+	
Testing	+		+	×	+	

(6) Base your work on a detailed plan developed in advance. Perform evaluation at all stages with criteria established in advance.

15.2.3 Education and training

All personnel involved in the new order of work need education and training. When a method and process have been strictly defined, more emphasis can be put on formal education and training. Therefore more material can be taught, not needing several projects to learn working in an orderly way. In a less formalized method, the staff need several projects as learning projects before getting familiar and highly productive with the method.

The scope and amount of education needed varies depending on each person's role in the project. Everyone should have a basic education. The purpose is to get a common basis of concepts and way of working. Additionally, more specialized education and training is needed.

In the Table 15.1 we have summarized our experiences from the need of education.

An example of contents of the courses is given below.

- Concepts – Basic concepts of object-orientation and the fundamental concepts of the method. Chapters 3–6 in this book may form a basis. 1–2 days.
- Project management – Specific characteristics for the new method.

– Appropriate metrics for the management within the new method 1 day.

- Overview – An overview of the method using examples to get familiar and acquainted with the entire method in the system life cycle and the underlying ideas. This entire book may form a basis for this course Attendees should be familiar with object-orientation. 3–4 days
- Analysis – A thorough and detailed study of the entire analysis process. Emphasis should be on applying the method and process on a larger example, possibly on the current system. 3–6 days
- Construction – A thorough and detailed study of the entire design and implementation process. Emphasis should be on applying the method and process on a larger example, preferably the system to be built. The course does not cover the specific language, but rather how the language is incorporated within the process Those attending should be familiar with the programming language used. 3–6 days.
- Testing – A thorough and detailed study of the testing activities and principles. 2–3 days.

If CASE tools are to be used in the project, additional training is needed. Our experience is that the CASE tool should be well-delimited from the process and method issues. The attenders will otherwise have difficulties in differing in method, process and tool and too much time is spent on getting acquainted with the practices of the tool.

Any other technical aids that are new and should be used may also need additional education and training. Examples are the programming language, DBMSs and operating systems. Other examples of new areas that also require education and training are component management and usage and quality assurance. These other areas should also be supported by education and training.

Besides these courses we also need training to get familiar with the new technology. This can be done by developing a part of the system or introducing new people for simpler tasks in projects with experienced developers. All together about three months should be allocated for a developer to become productive in the new environment. Hence, it is quite expensive to introduce a new development process, but it normally pays off in a few years.

15.2.4 Risk analysis

Introducing new technologies always brings potential risks. The number of new technologies introduced simultaneously increases risks exponentially The introduction of a new development process, often involves other changes as well; a new point of view – object-orientation; a new programming language – OOP; a change of documentation techniques; new CASE tools, new development environments, new QA techniques and so on. Not only does the technology introduce risks, but the maturity of the developing organization could also be a large risk. Are we mature enough to treat system development as an industrial process? Can we introduce the discipline needed? All of these changes at the same time do involve a large risk. It is therefore essential to be aware of these risks and to have a way of handling them. To have progress one must take risks, and awareness of the risks is a prerequisite to manage them.

We will here give a simple technique for how to detect and manage risks in introducing OOSE. The method is divided into three steps.

(1) Risk identification
(2) Risk valuation
(3) Managing the risks.

For **risk identification**, we define the potential and foreseeable risks of the project that, if they occur, may seriously injure the project. The starting point should be the goal of the project. What risks may occur that will make us fail to reach the goal? These risk areas could involve OOSE specific risks, but also other risks. Examples of risk areas involved in OOSE projects are:

- *Paradigm shift*
  - Are we mature enough to adapt a new development strategy?
  - Will the team be mature enough to not start coding too early?
  - Is the project manager familiar with the new paradigm?
  - Are the team members familiar with the new paradigm?
  - Are we familiar with the programming language to be used?
- *Process*
  - Is our process well defined and well documented, and can we work along it?
  - Is the process mature?
  - Does the documentation produced fit our purpose?
  - Do we have sufficient training capabilities?

- *Tools* (analysis, design, implementation, DBMSs, configuration tools, purchased component libraries etc.)
  - Are we familiar with the tools to be used? With the learning threshold?
  - Are the tools to be used mature and stable?
  - Will we have the tools when they are needed?
  - Are the tools compatible? Are they integrated?
- *The system*
  - Are we familiar with the application domain?
  - Are the requirements clear, consistent and stable?
  - Can or must we integrate existing systems? Is it possible?
- *Organization*
  - Do we have a tight schedule?
  - Will time-to-market be critical? Will the pressure from the market change?
  - Does the organization have realistic expectations for the project?

These are just a few examples of potential risks. There are of course many more, and the risks must be identified from each project's circumstances. About 15–20 risk factors could be identified in a normal sized project (5–15 persons). Risk identification is best done on a group basis, and it is better to identify too many risks than too few. The risks should be broken down to concrete situations or events.

The next step is **risk valuation** Now the potential risks should be evaluated to assess the consequences (disturbance or damage) if they occur We then initially judge the *probability* that the risk will occur. A scale from 1 (very improbable) to 5 (very probable) could be used. Then we assess the *consequences* of each risk. A scale from 1 (neglegible) to 5 (catastrophic) could be used. Now multiply the probability and the consequence factor for each risk, see Table 15.2.

We will now have a relative measure of the size and potential threats of the risks.

The third step is **managing the risks**. From these measures we establish a prioritized table of the most serious threats to the project. For each of these serious threats we propose active actions to prevent them from occurring or to reduce their consequences. This could be done by decreasing the probability of the risk occurring or decreasing the consequences, or both. In this example we see that the major threats are that the project team are not used to OOSE and we are not sure about the delivery of the DBMSs. The measures to avoid these threats could be to issue an education program for the team members and not to use the DBMS proposed.

Table 15.2 A table for risk valuation

Risk	**Probability to occur (P)**	**Consequence (C)**	**P*C**
Development tools not mature	3	4	12
Not familiar with application domain	2	4	8
Project team not used to OOSE	5	5	**25**
Weak support from upper management	1	4	4
Delayed delivery of stable DBMS	4	5	**20**
..	.	..	..

Since some risks may be hard to detect initially, we should identify warning signals that we should look for during the project. We could also initially issue alternative plans for if any of the risks should occur.

Risk analysis should be done initially in the project, but it is important to do it also throughout the project, since new risks may show up or their probability/consequence may vary during the project.

Introducing OOSE in an organization involves risks. It normally also involves a large paradigm shift with far-reaching consequences. It is important to be aware of this, and, as stated, not have too high expectations of the first project, especially not in the reuse issue. As previously mentioned, OOSE stimulates reuse, but to be able to reuse, we need something to reuse. Depending on existing reusable software, the first project will normally have varying levels of reuse. However, the first project will usually yield reusable software for future projects.

This simple method of risk analysis could be used if no other method exists in the organization. The important thing is to think about potential risks prior to the project and to prepare activities to manage potential problems in the project. Please note that we cannot compare projects with this method. Neither should we mechanically conclude that any project with high risk threats should be stopped. As stated, preparing mentally for potential risks enforces the project. A more advanced method for risk analysis is the **Lichtenberg** method, see Glahn and Meland (1985).

Since a project introducing a new working method involves large risks, a risk analysis often signals red. To balance such a risk analysis, a **benefit analysis** could also be made. What are the advantages of introducing this new working method? We will not go into details here, but just mention some major benefits of OOSE. Firstly, typically more than 50% of total development time involves information search and retrieval and coordination of activities, thus not directly efficient development time. An ordered way of working, introducing a defined method and tools, typically decreases the information search and retrieval time. Additionally, a well-defined process decreases the coordination time overhead. Hence more time can be spent on actual development. Secondly, since OOSE models the way people think, the models are much easier to understand by non-OOSE practitioners than are models produced by other classes of methods. Thus managers and customers can take a more active part in the actual development. These are just two benfits of OOSE, and there are many more like higher quality and decreasing future maintenance costs.

15.3 Product development organization

When developing a product, the basic organization of a project should be built around the product and the activities associated with the development of the product. Since the product (one hopes) will be developed in several versions, we must have the **product life cycle** in mind when discussing this topic. Here we see the important difference between method and process as discussed in Chapter 1. Thus let us discuss briefly the processes that exist to develop a product.

Product development with OOSE is built around different models that are developed in sequence. All these models must be maintained during the entire product life cycle; every further development should be accomplished by modifying these models. This means that all changes in the requirements specification should first be analyzed in the requirements model and inserted there. These should lead to modification in the analysis model and later in the design, implementation and testing. Hence all of these models must be up to date at all times during the product's life cycle. In these models, all objects will be documented on their own. The whole model will hold all of these documents together. We have earlier discussed the importance of the traceability between these models and the documents. It is important to be able to do this continuous revision smoothly.

The first model to be developed in OOSE is the requirement model. This model is developed by the activities shown in Figure 15.1.

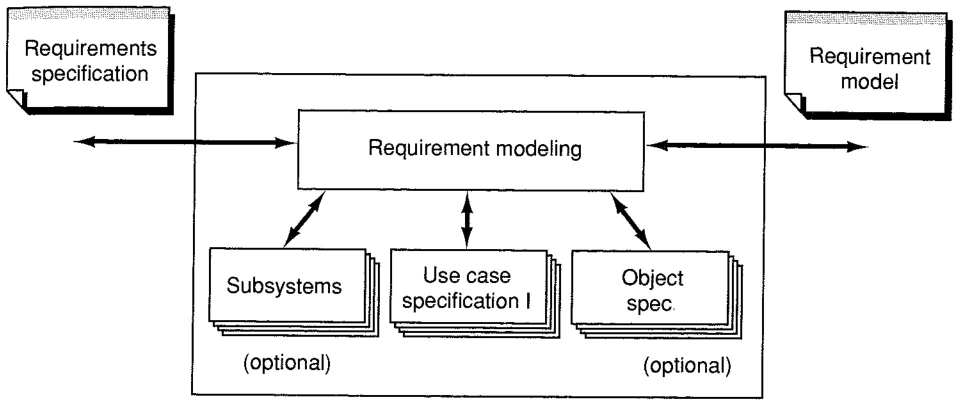


Figure 15.1 The processes of requirements analysis.

Here we have one coordinating process that lays the foundation and one process for each use case that specifies this use case. The identification of domain objects, actors and use cases is done in the coordination process while the specification of the use cases, objects and subsystems and their interfaces is done in the specification processes.

The requirements analysis process delivers a well-defined result; the requirement model with the use case specifications. This forms the input to the object modeling for the Analysis process, see Figure 15.2. The main process here coordinates three different kinds of activities: the identification of analysis objects from the use cases; the specification of each object; and the specification of each subsystem. All of these subprocesses have one subprocess instance for every such specific activity.

The analysis model, forms the well-defined result after this process. This forms the input to the construction process, see Figure 15.3. Here it is the system construction process that is the coordinating process. It has three classes of subprocesses. The first is the design of the use cases, where each use case is designed over the blocks. These activities will result in the interfaces of the blocks. The second class of process is the block construction process where each block is designed, implemented and unit tested. The third is an optional subsystem construction process for a top-down approach. Note that, as you will have noticed, we keep the design and the implementation encapsulated in the block construction process.

The well-defined result delivered by the construction process is the design model and the source code for the unit tested blocks.

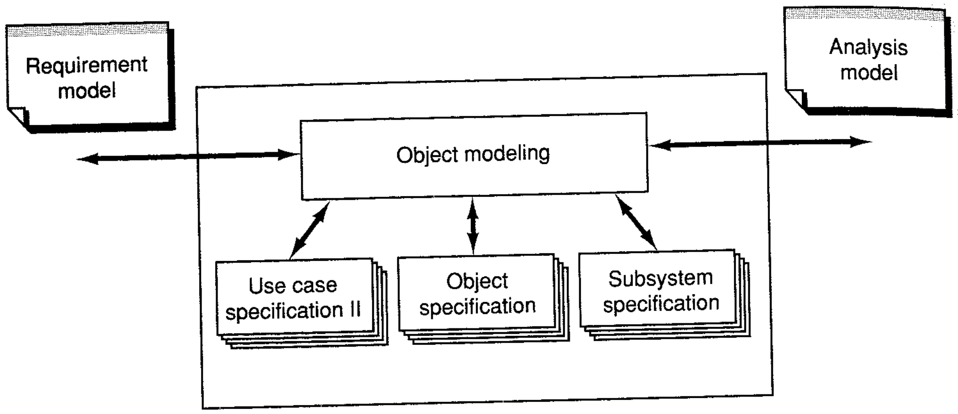


Figure 15.2 The processes of robustness analysis.

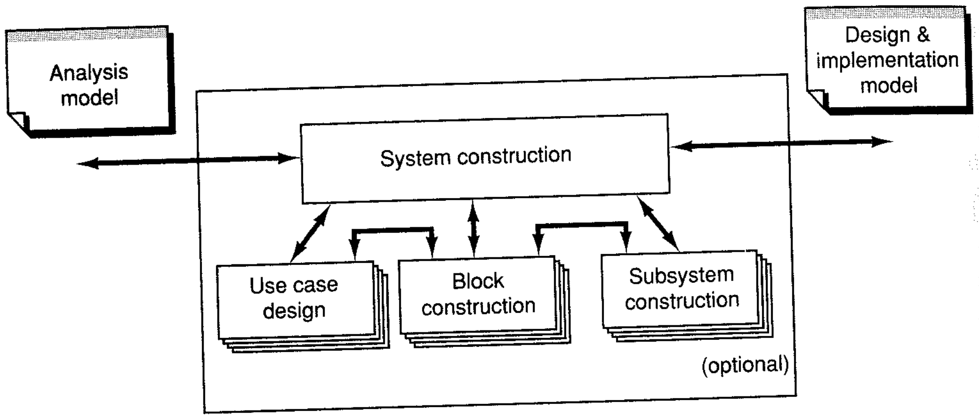


Figure 15.3 The process of construction.

This result is the input, together with the requirement model, to the testing process, see Figure 15.4. In the testing process the system is integration tested and system block tested. Here the coordinating process is the system test itself. The integration testing is performed by two subprocesses, use case tests and subsystem tests.

In addition to these three main processes we also have the component process. This process is normally not coupled to any specific product, but is a multiproduct process, that is, it is shared by several product processes. It is mainly the construction processes that it interacts with. The component process with its subprocesses is shown in Figure 15.5. The coordination process here is the

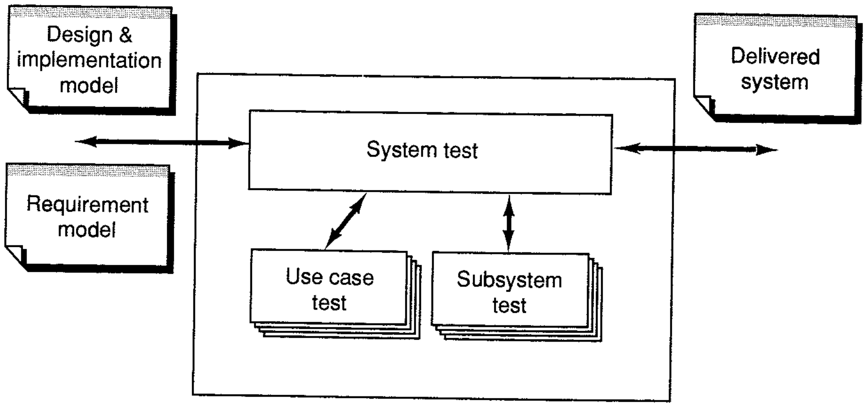


Figure 15.4 The testing process.

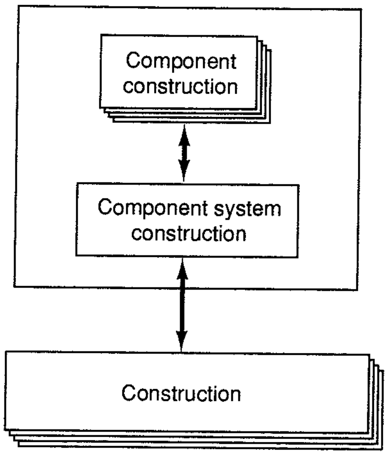


Figure 15.5 The component process interacts with several construction processes, one per product being developed.

component system construction process and the actual design, implementation and unit testing are done in a component construction process one for each component.

We have now discussed the main activities as processes in the product life cycle. The processes start when the development of the first product version starts. They then last as long as the product is maintained. Note that these processes will be unmanned when no development is being done on the product, but when a further development is to be performed the processes will be manned again.

The processes may also be concurrent activities. In this fashion we can define responsibilities for these processes in terms of, for instance, documents to be maintained Additionally, all work that is being done on a product development can be associated with one of these processes. Note here that we have generalized the development to interacting, durable processes and that we will have very similar behavior in new development as in further development. These processes do not only participate in pure development projects, but also in tendering projects, error handling projects etc

The **project organization** problem is now reduced to the question of manning these processes. Note also that the processes are similar during the entire product life cycle, that is, each new project for changing the product also uses these processes. The first version of the product will thus have the responsibility for initiating the processes, while subsequent projects just activate them by manning them. From these processes we can also identify roles played by different people during the development, and we shall discuss the importance of this later. Another benefit of this process model is that it is easier to express different kinds of development. Incremental development, which we shall discuss more later, is a very sound and common strategy, and is easily expressed in this process model. Also, the iterative nature of development is supported by the process model. Let us now turn to the discussion of allocating development staff members to these processes, that is, the actual project organization and management

15.4 Project organization and management

Managing and organizing any software development project requires a thorough understanding of the pitfalls in the trade. A necessary, but not sufficient condition for sucessful software development is good **project management**. There is an abundance of literature about how to manage software development projects. This is however not the subject of the current book and we therefore refer the reader who wants a comprehensive treatment to any of the standard works, such as Metzger (1981). Instead we will concentrate on how to combine common project management practice with OOSE and how to organize the project with respect to the product processes discussed in the previous section We will also consider what implication object-orientation and OOSE will have on project management and what project management should especially consider when working with this new technology.

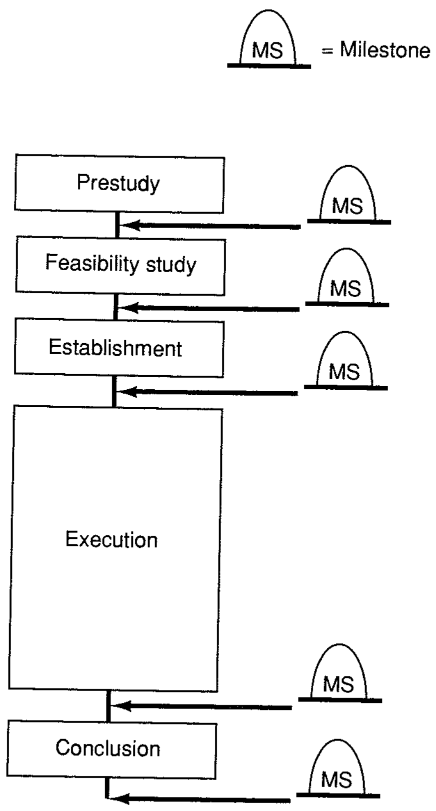


Figure 15.6 The general management part of a project

All projects will have a technical aspect and a management aspect. The purpose of the management aspect is to control, steer and follow up the project. The technical aspect covers what and how you should work to develop the current system or product. It is here that we will find the process model discussed in Section 15.3.

The management and technical aspects of a project must, however, fit together and it is common to achieve this by a number of **milestones** that should be achieved. A milestone is a concrete, objectively defined or determinable event or precisely defined deliverable. The milestones are often combined with **reviews** and **audits** of the work done so far. Between these milestones the work is performed. This division aims to give better control of the project.

In Figure 15.6 is shown a typical project model for the overall management of a project. Each specific phase is delimited by a well defined milestone.

We will here give a short overview of each phase:

- **Prestudy**. Aims at defining the task by developing and evaluating different kinds of requirements, needs and ideas to judge, technically and economically, whether the project is practicable.
- **Feasibility study**. Different technical alternatives and their consequences are investigated. A main time and resource schedule is planned and also an evaluation of potential risks in the project.
- **Establishment**. The project is organized, planned and quality assured. Detailed time and resource plans are developed.
- **Execution**. The project as such is executed in accordance with the plans previously developed.
- **Conclusion** The project is settled and proposals to improve the project and development methods used are summarized.

To this project model we should add the technical aspects of the project These include what to do in the specific phases. OOSE will be used mainly during the execution phase, although it is also possible to run the prestudy as a surveyable execution and thus also use OOSE there. However, work will also be done in the other phases that are part of the technical aspect. One example of this is **prototyping**. It is very important to achieve an understanding early of the system to be developed. The first two phases typically use prototypes. During the prestudy the purpose is mainly to evaluate technical aspects of the system to be built, often in the form of simulating certain critical parts. During the feasibility study, the prototyping is often more focused in its purpose to investigate certain technical alternatives or to support the requirements specification to be written. The purpose of these prototypes is to increase the precision and quality of the requirements, not to skip or short circuit later phases. Often prototypes can result in new prototypes to investigate certain tasks further. This is shown schematically in Figure 15.7

The prototyping technique may also be used during later phases to improve the quality of the system. It is extremely important to be aware of the purpose of the prototype; should the prototype be further refined to the system or does it just aim at investigating certain questions? Both aims are good, but far too often a good experimental prototype 'becomes' the real product, and is not what was aimed for originally Prototyping should aim at increasing the quality of the product; not decrease it.

The models developed in OOSE are good for supporting the steering of the project. All models have a well-defined result and it

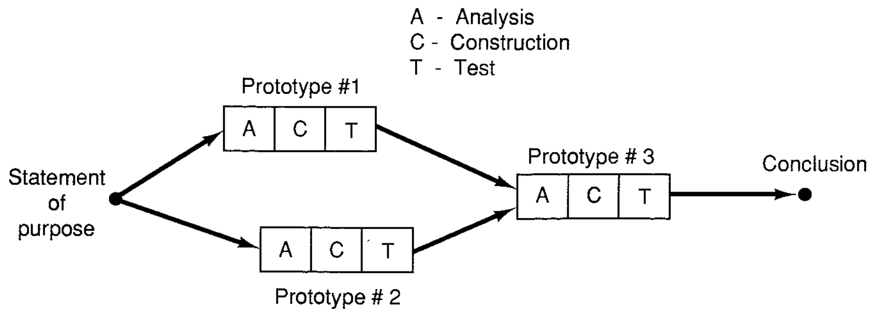


Figure 15.7 Prototypes are often an effective way of testing the validity of product ideas and preliminary system requirements They may very well result in new prototypes to refine the ideas

is appropriate to use them in combination with milestones Hence the models can be mapped onto the project model. An ideal mapping of the models is shown in Figure 15.8.

This ideal mapping gives a sense of an early waterfall model where the entire analysis model should be developed, reviewed and frozen before starting work on the design model. It is essential to realize, though, that the models will be modified when work is started on subsequent models. It is therefore essential to understand that there is no point in thinking that there will be no changes, rather it is important to have a way to handle these changes. A model which is not updated will be out of date and thus not show a consistent picture of the system. An appropriate way often is to give each model a new version at the different milestones In the following we have used an alphabetical versioning technique starting with A, B, ... Any technique could be used, we just want to illustrate the idea here. In Figure 15.9 an example of how to version the models at different milestones is shown. To handle this versioning a tool is often needed. Exemples of different kinds of documents that evolve during the project are also shown in Figure 15.9. It is also notable that the actual coding normally starts quite late in the project. We have seen that this is often an uneasy time for the team members since progress is often measured in terms of lines of code produced. Additionally, since the first project is often an evaluation project, several parties are interested in the progress of the project. However, we have also seen that when the actual coding starts, it is done quite fast, often yielding a high productivity in the overall project. Larger projects often need a more incremental development by developing the system in layers. Incremental development is further discussed in section 15.4.1.

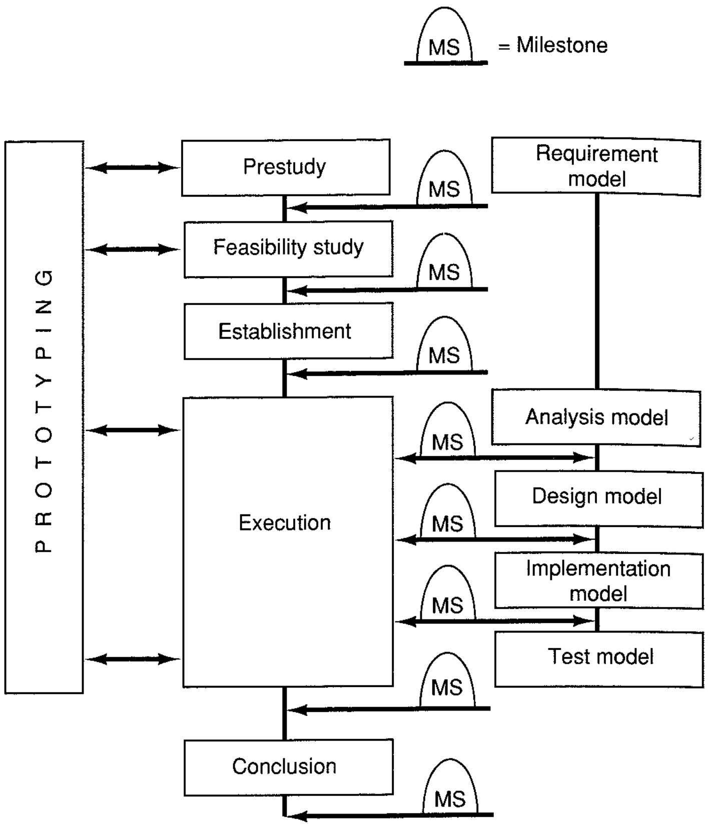
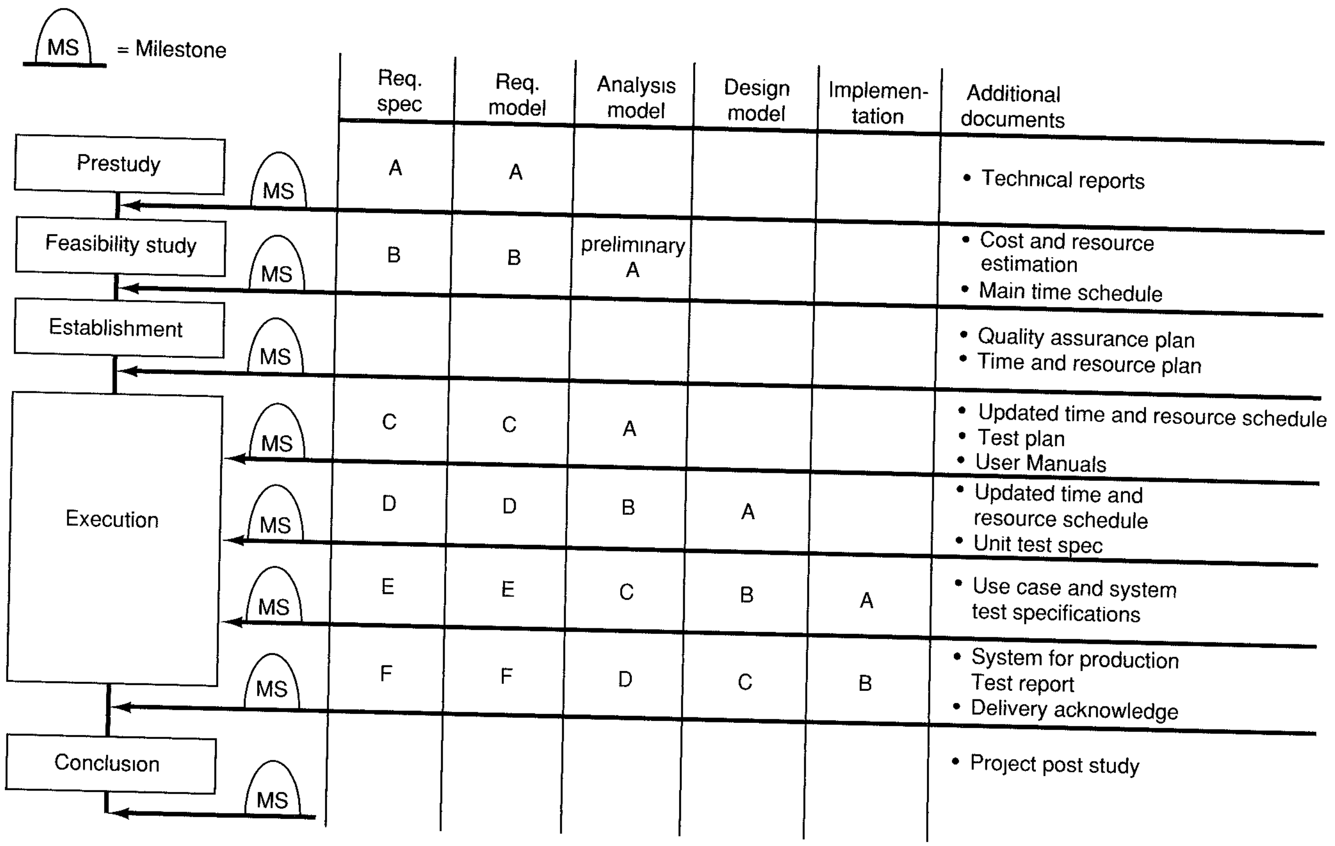


Figure 15.8 Prototyping may be used during many of the phases in the project The models in OOSE may be used as milestones that should be achieved.

The requirements specifications should be approved by both supplier and orderer and should contain a complete list of all requirements on the system. The requirements should be ranked by the orderer and the cost should be estimated by the supplier. Additional requirements may involve delivery date, resources and quality. The quality of requirements specifications varies tremendously. It is usual in technical systems to have detailed specifications while information systems more seldomly have such detailed specifications as discussed in Chapter 1. The requirements model could very well work as the final requirements specification.

Since the requirements specification is something that a contract may be written on the basis of, it may seem odd that the document will be updated and modified as the version table indicates. The hardest part when defining the requirements specification is



Req. spec	Req. model	Analysis model	Design model	Implemen-tation	Additional documents
A	A				• Technical reports
B	B	preliminary A			• Cost and resource estimation • Main time schedule
					• Quality assurance plan • Time and resource plan
C	C	A			• Updated time and resource schedule • Test plan • User Manuals
D	D	B	A		• Updated time and resource schedule • Unit test spec
E	E	C	B	A	• Use case and system test specifications
F	F	D	C	B	• System for production Test report • Delivery acknowledge
					• Project post study

Figure 15.9 An example of how versions of the models may arrive at different milestones. Examples of documents that may be appropriate at each milestone are also shown.

often to formulate it so that the orderer and the end users can understand what will be delivered, at the same time as the developers will have a complete and well-defined support for the coming development. This document will therefore almost always be updated as the development is ongoing and hence the requirements specification (version A) that is written early on will have to be changed during the project. These changes are often aimed at eliminating uncertainties that are noticed and also incorporating additional functionality. Since the contract is of tenbased on this specification, it is important that all modifications of it are approved by both parties. This also enables the orderer to follow the development and to check that no misunderstandings occur. There are three main reasons to modify the requirements specification during later phases. The first is that the requirements are not distinct enough and need to be clarified The second is that the project notices that the time or resources available will not be enough to fulfill all requirements and thus wants to delete certain requirements. Finally, certain user groups will wake up when the system is soon to be delivered and then want to insert requirements that actually should have been in the first version of the requirements specification.

A point that is too often forgotten is that new or changed requirements will inevitably generate new costs and possibly also delay the project. Remember also that it gets more expensive to add new requirements the further you have come in the development cycle. A requirement that was estimated at 200 manhours in the specification phase, but was left out, may very well cost 1000 manhours if introduced in the design phase

When talking about timing we will only mention an obvious fact that is too often forgotten. If you delay one phase of the project, it will inevitably delay the delivery date also. The subsequent phases will not suddenly be done faster just because one phase took longer. Here we see the importance of following up the project plan and schedule.

15.4.1 Incremental development

Although it is possible to execute a development as discussed above, experience shows that an incremental approach is often more appropriate Since the models are developed in a very seamless way, it often feels natural to 'investigate' what will happen to one object in one model when continuing to the next model. Object-oriented modeling is often connected to an iterative way of working, probably because of the strong traceability between different models; a problem domain object may exist all the way to code. However, to work too iteratively is not good either. Then you will use later models to correct earlier models; much the way compilers often are used to debug programs. The solution lies of course in between: first develop one part of one model and then continue this part to the subsequent models and then continue with the next part of the first model.

When using OOSE it is often appropriate to develop the requirement model quite extensively as a first step. One reasonable goal is at least having all use cases *identified* for the entire application. The reason for this is that it is important to have an understanding of the entire system before starting to structure it. It is then possible to start with a couple of use cases and specify them and then continue with these use cases into the analysis model and also the design, implementation and testing phases. In this way we take the use cases of highest **rank** not designed yet and refine the latter models with the new use cases. Here it is evident that the use case is the red thread through all the activities. Testing accumulates the use cases, and when all use cases have been tested, the final system test is performed. This development is illustrated in Figure 15.10.

Each increment should be of reasonable size, not too small and not too large. Increments of about 5–20 use cases are often appropriate. It is also important that each increment covers a limited time; often 3–6 months is a reasonable turnaround time and preferably not more than 12 months. We have seen in projects that project members are often unsure how much work is to be done in one model and how that work is to be used in later models. When this is the case, it is often appropriate to have a fast turnaround, to gain experience from work in all models early. Working this way, the developers will early on have a better understanding of the whole development process and can thus optimize the process as such.

In this way the different phases may also overlap For instance, when doing construction, analysis can be performed in the next increment, see Figure 15.11. Since one increment may modify results from previous phases in the same increment, it is important to have a very controlled way of handling different versions of the models. Additionally, you must be able to handle the modification and updating of models in a dynamic way, for example modifications in the analysis model in the first increment should be incorporated smoothly into the analysis model for the second increment.

It is essential to minimize the work needed to deliver results from one phase to another. The deliverables from each phase are defined by the process outputs A way to solve this is to have some developers follow one increment through all phases and other

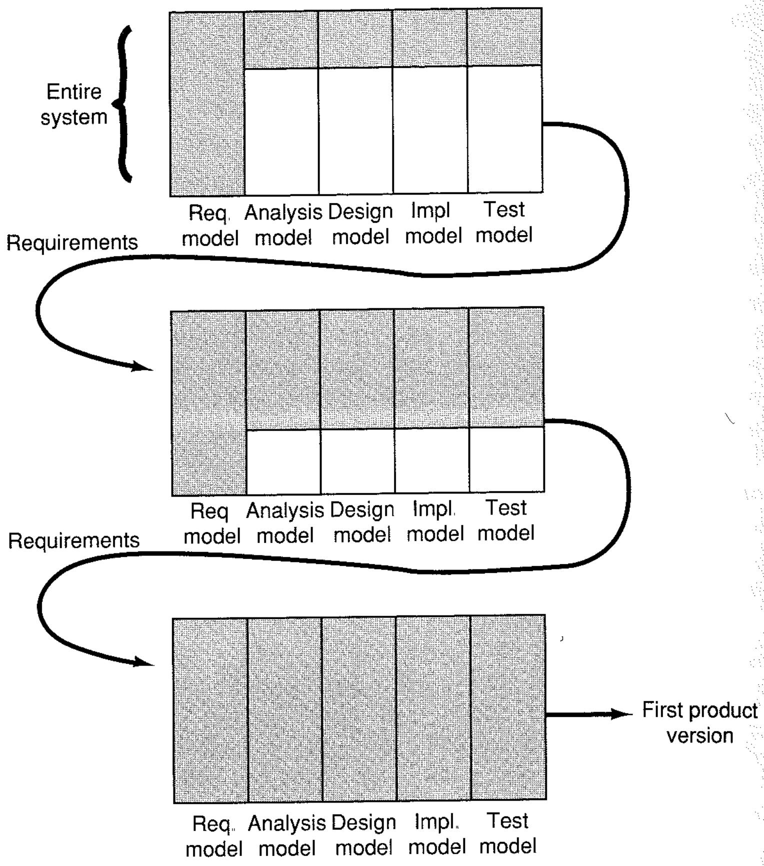


Figure 15.10 Incremental development in OOSE.

developers follow one process over several increments, that is, to have some specializing in a couple of use cases and others specializing in for instance analysis

15.5 Project staffing

The number of people involved in a project varies highly in different phases. Different phases also need different competences. Additionally, the organization needs to be changed over these phases to manage the project. This is due to the fact that in some phases work can be done in parallel while this is harder in other phases, but also since some phases require more resources than others.

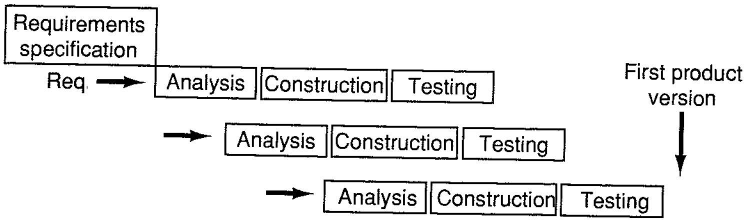


Figure 15.11 Work in different phases may very overlap Then it is essential to have support for handling different versions of the models.

Generally, it is valuable to have people participate both in analysis and construction and thus ease the transition between different models. Therefore a development core of staff can be the core of the entire development. However, we may have people specializing in analysis, construction and testing. Especially (integration and system) testing is often performed by a special group. The test specifications, though, are developed from the use case description from design and are based on which blocks are involved in the use case and when those blocks will be delivered to testing. Design and implementation should thus be planned so that the blocks can be delivered to testing when they are needed in testing. In this way much of the work can be done in parallel. This means that even if other people are involved during testing, they must collaborate intensively with the construction personnel

We will here briefly discuss some typical ways to organize a project and also illustrate some specific points in an object-oriented development. We here assume a medium sized project which involves 5–20 people during analysis, construction and testing. Staffing in other phases than the ones we have discussed in this book will not be covered.

When assigning development staff tasks in a project, we use the processes discussed earlier After all, since the processes describe what is to be done, a specific project is actually a flow over these processes. The problem is thus to man the processes for a specific project Let us start with the coordinating processes. These are summarized in Figure 15.12.

These main processes coordinate the entire development. The identification of use cases is performed in requirement analysis, while the specification of them is done in specific subprocesses for each use case specification. These main processes will structure the

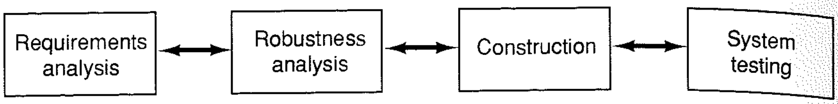


Figure 15.12 The coordination processes in OOSE.

system robustly (in robustness analysis) and make the final architecture (in construction) Hence the important structuring of the system is performed within these processes. The people staffing these processes should therefore be the core of the development and thus should be the same in at least the first three processes We may call this group the **system architecture group**. Their responsibility is to make sure that the system architecture, and its coherent idea, is maintained during the entire development. It is essential that these people are highly qualified and have a strong influence on the project members and that the rest of the project members have confidence in these people. The project manager should be tightly coupled to this group.

The initial surveyable analysis (mainly requirement analysis) should be made by quite a small group with much interaction with the end users. All people involved in requirements analysis do not need to know the technology used in later phases; the concepts used are quite intuitive. These people should have a close contact with end users, customers, marketing people, experienced developers and so on.

The subprocesses for the more detailed work should be manned by development personnel that have special skills for the activity. Here it is often good to have the same person responsible for the same group of objects in all activities. For instance, the person specifying a specific use case should also specify the objects that offer the use case, design that use case and implement the corresponding blocks. However, since the complexity (i.e. work) increases, especially during implementation, this is not always possible. Nevertheless, the person responsible for the specification in analysis should work as a senior designer when implementing the blocks. Similarly there are often people involved in modeling the requirement model who will not participate in the subsequent work.

The main reason to have the same person in all subprocesses that manage some objects is to minimize the work of collecting and understanding information. Additionally, you will avoid the conflict of not consenting with the specification (the 'not invented here' problem). However, this may also involve drawbacks. When doing it this way, reviews will be more important Other persons must also be able to understand the documentation for further development.

In larger projects it is often appropriate to divide the development into several project groups Each group is then responsible for one subsystem and/or a specific task in the development. It is far better to have one group responsible for one part of the system than to have one group responsible for a specific phase. The reason for this is that the domain knowledge, for example the knowledge of the subsystems task, is more important for the result than is detailed knowledge of a specific phase. In each project group, the activities are divided further. All these activities are cooperating subprocesses in the development.

When reaching construction, typically more people are involved This may be solved by adding more people to existing groups or to add new groups in the project organization. When you add new groups you will also lessen the responsibility of each group. It is often more appropriate to have smaller subsystems assigned to each group instead and to keep the initial groups. Generally you should not let the resources affect the system structure. The new people can have other knowledge profiles than the people in the analysis phase. Block testing is done during construction and thus is usually an activity within each and every project group. These phases also need a coordinating architecture group, preferably the same as during analysis.

The component activity is responsible for maintaining and developing components. This is an activity that should be shared among several projects and thus not a part of the project. It is essential that upper management realizes this important distinction.

Integration testing is done in a separate testing phase, often by a separate group. This group can very well start its activites by writing test specifications when use cases are specified during analysis. When all use case tests are performed the final activity within the development is the system test This test involves testing the functionality of the system, documentation and tutorial material. The system test cannot start before all the parts in this version have been delivered from construction.

During or after the testing phase, it is suitable to do a poststudy of the project where positive and negative experiences are collected and documented It is better to do this study directly after the project when the members are available and their picture of the project is still clear and to complement this study with a study of operation experiences of the system after about 3–6 months, than to do the whole poststudy then. The risk then is that the poststudy will never be done and hence valuable experiences are never documented.

Besides the actual developing groups there may be a need for other roles or groups in the project. Examples of this are listed below:

- **Methodologist**. A person or group responsible for the method used. He or she or they should be experts in the method and support the development team in applying the method. It is often useful for these people to understand 'both sides', namely to be able to explain the new techniques in terms of the techniques already used. This means that the methodologists must be experts, not only in the new method, but also in the language, operating system, product structure and development organization of the team where the new method should be introduced.
- **Quality assurance (QA)**. People responsible for both the product and the process to develop the product so that it is of high quality. This involves guaranteeing that all software delivered is of high quality and that documenataion is consistent. Reviews are here a useful tool and we will discuss this issue more later.
- **Documentation, manuals and education**. Documentation of the system should be made by the developers. It should of course be consistent with the system developed. Manuals, both for maintenance and for users, should be written by people with special skills for this. Planning for education on the system must also be done. People who should be trained include users, maintenance and operation people and sales staff.
- **Reuse coordinator**. This person or group is responsible both for encouraging and evaluating how much the project is reusing and also investigating the reuse potential of the code and designs developed. Code, designs, documents and models may be reused. This person or group should work intimately with the architecture and QA group. Reuse might not give a payback in one specific project; the real gain comes in subsequent projects that reuse what has been developed in this project. Therefore the cost of this function should not burden the project cost, but rather be considered as a multiple-project cost. Note that the coordination and management of the reuse library should be interproject, see Chapter 11.
- **Prototyper**. This role is necessary to investigate different solutions at an early stage to prepare for later development. Typically user interfaces are interesting to prototype in early phases, but simulation of certain designs in later phases may also be interesting.

- **Support environment** This is to function as a service to the project as a whole Typically system managers play this role.
- **Staff**. To help the project manager, staff may be needed. One example is a special project administrator responsible for following up cost and time schedules. This is often needed in larger projects. Some of the above roles may be part of the staff.

It is often effective to have people with these special roles also working in the project groups. If so, it is important to give them time to fulfill their double roles.

15.6 Software quality assurance

Software quality assurance (SQA) aims at ensuring that the final product will have an acceptable quality. It is mainly a management activity to identify quality problems early in the development. Cost and time schedules are often tracked in the early stages, as opposed to quality, but quality problems appearing late in development involve large risks for any project. Therefore it is just as important as tracking cost and time to track the quality.

Quality assurance focuses on both the product and the process. The **product**-oriented part of SQA (often called Software Quality Control) should strive to ensure that the software delivered has a minimum number of faults and satisfies the users' need. The **process**-oriented part (often called Software Quality Engineering) should institute and implement procedures, techniques and tools that promote the fault-free and efficient development of software products.

What then are the characteristics for high quality in a product? In Figure 15.13 some of the characteristics are shown.

These characteristics are not exhaustive and not even independent of each other. Additionally, they often tend to conflict in a development. Therefore, when starting a development, a good point is often to decide what characteristics are the most important for this specific product and then focus on these throughout the development. In OOSE the focus is on maintainability characteristics. As we have discussed, the maintenance of the product is the major objective when developing the structure of the system. However, this will of course also have effects on the suitability criteria; if it is easy to introduce changes to the system, it will also decrease the number of faults introduced when modified and thus give the product a higher reliability.

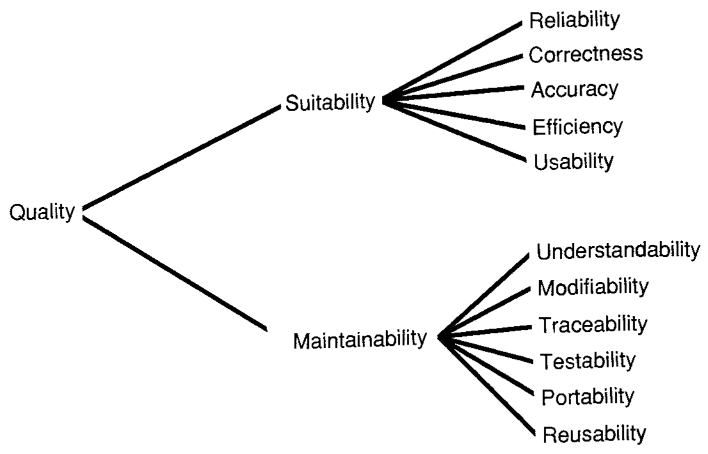


Figure 15.13 Some characteristics of software product quality

The material to work with when doing SQA is mainly the documentation produced during development. No new documents should be needed for SQA. Therefore it is essential that everything important that is done should also be documented. The (far too) common picture in software development is first to do the work and then to document it. The right way is first to document what should be done and *then* do the work. When working with OOSE this is permeated. The requirement model will be developed first followed by the analysis and design models, before writing the actual code. Each of these models will be developed and documented concurrently. Therefore OOSE gives a good platform for carrying out quality assurance in an accurate way.

The main tools for quality assurance are the development process itself, reviews and audits, testing and also metrics. The development processes of OOSE, including testing, have been surveyed in this book. Metrics will be discussed in the next section. Here we will discuss very briefly the integration of reviews in OOSE. First some terminology: a **formal** review's objective is to decide whether or not to proceed to the next phase. Such a review is held at every major project milestone. A quite large review team is often involved and also customers or orderers participate. An **informal** review's objective is to discover errors that have been made. These reviews can be held at any time during development, such as when something is completed that ought to be checked before continuing the development. Informal reviews often have a quite limited participation, typically some of the developers.

Where to use different kinds of reviews when working with OOSE depends on the size of the project. In a small-to-medium sized project, typical formal review points are between the main activities, that is, each model when it has reached its first version. Informal reviews may be used after each subprocess, possibly grouping some subprocesses in one review.

Different kinds of reviews have also been defined by IEEE (1983) in a standard glossary. Three different kinds of reviews are then given:

- **Review** A formal meeting at which a model is presented to the user, customer or other interested parties for comments and approval,
- **Inspection**. A formal evaluation technique in which models are examined in detail by a person or group other than the author to detect errors, violations of development standards and other problems,
- **Walkthrough**. A review process in which a developer leads one or more other members of the development team through a segment of a model that he or she has written while the other members ask questions and make comments about technique, style, possible error, violation of development standards and other problems

Of these we may characterize review as formal and inspection and walkthrough as informal. Although every review is unique and focuses on a specific model or object, they have some points in common We will here not give exhaustive lists of what to review in the OOSE models, but rather just highlight some examples.

Common to all reviews is to check things like consistency with requirements, completeness of model, redundancy, structure, naming, correct associations, understandability, versioning of documents, standards and views. This is reviewed more or less dependent on the purpose of the review

When reviewing each model there are specific things to focus on We shall here only give some examples of points to review in the Requirement model.

- Is the system delimitation appropriate?
- Do the use cases match the requirements specification?
- Have the requirements specification been updated?
- Is it possible to understand the use cases?
- Are all roles interacting with the system identified as actors?
- Do all actors have the right set of use cases?

- Has the requirements specification been covered?
- Are the interfaces described in a satisfied way?
- Are the use case flows correct and complete?
- Have enough alternative flows and error flows been described?

Of course the actual questions may vary from time to time, but the intention should be clear from the above. When reviewing the other models the intention should be the same; to find errors as early as possible in the development process and similarly to guarantee a high quality in the product.

When performing the review, different methods and techniques can be used. A systematic approach to this is a necessity to achieve a high quality of the delivered system. Methods and techniques for this are described in the literature, see Weinberg and Freedman (1982), Myers (1987) and Yourdon (1989b).

When performing reviews, management has an important role to play. It is important that the management show commitment to the process and results and also budget time for performing reviews. More than 5% of the overall development time is not unusual. Likewise it is important to get good people as reviewers to guarantee a high product quality and high confidence with the review process from the development staff. The manager should also reward good reviews of poor products and punish bad reviews of any products. There will always be errors to detect. Some hard facts about defects are the following:

- A defect introduced during requirements specifiation will cost 100–1000 times more to correct when the system is in the testing phase than it would cost to fix it during requirements specification.
- Between different programmers the number of faults introduced may differ by as much as a factor of 10 when producing 1000 lines of code.
- During normal testing only about 50% of the faults will be detected.

To achieve a good quality discipline, and a high quality awareness, an independent quality group responsible for quality assurance in the development department may be needed. This group can do reviews of how well the project members follow the given process of development and can also make an assessment of the project's possibilities of achieving its goals and illustrate potential risks. However, the QA group should not function as policemen, but rather, together with the development team, increase the quality of what is being done

Finally, we want to give some quality advice:

- Follow the development process thoroughly, and note what goes wrong,
- Eliminate the faults as quickly as possible by revieweing all specifications thoroughly,
- See that the review groups have the right composition of people,
- Note in the review protocols the number of pages reviewed and the number of faults of different types found,
- Follow up the review protocols and identify, and possibly rewrite, extremely errorprone objects. Try also to identify any individuals who seem to produce many defects in specifications or code,
- Have an independent testing group testing the system and also writing the test report,
- It is always cheapest to *do it right the first time.*

15.7 Software metrics

'If you can't measure it, it's not engineering'.

A necessary way of controlling a development is to use metrics. The metrics can measure either the process of development or measure various aspects of the product. Metrics in software engineering have been discussed for a long time, but not used widely as a way to increase quality of the product or process The real problem is that we cannot measure exactly what we would like to measure; we must assume that there are relations between what we can measure and what we would like to measure. Process-related metrics have been used much more than product-related metrics, so let's start by discussing this.

Process-related metrics measure things like manmonths, schedule time and number of faults found during testing. To learn to handle and manage a development process such as OOSE it is important to start collecting data on these measures as methodically as possible Below are examples of a number of process-related metrics that it is proposed to collect when working with OOSE.

- Total development time,
- Development time in each process and subprocess,

- Time spent to modify models from previous processes,
- Time spent in all kinds of subprocesses, such as use case specification, object specification, use case design, block design, block testing and use case testing for each particular object,
- Number of different kinds of faults found during reviews,
- Number of change proposals on previous models,
- Cost for quality assurance,
- Cost for introducing new development process and tools.

These measures may form a basis for future planning of development projects. For instance, if we know the average time to specify a use case, we can then predict the time to specify all use cases when we know the number to be specified. Statistical measures (such as averages) should always be accompanied by the certainty of the measures (such as the standard deviation). Otherwise you will have no sense of the accuracy of the prediction. We have also noted that these measures may vary greatly between different projects, organization, application and staffing. Therefore it is dangerous to draw general conclusions on existing data without looking at the circumstances. For instance, in one project a typical complete use case design including all alternative courses and error courses could take about five days. In another project where the use cases are smaller a typical use case design may only take two days.

For **product-related metrics,** several different kinds have been proposed. None of these have been demonstrated to be generally useful as overall quality predictor. However, as we discussed in the previous section, some quality criteria can be used to predict a certain quality property. One example to measure traceability is to measure how many of the original requirements are directly traceable to the use case model.

Traditional metrics on products (including code) may to some extent be used also in object-oriented software. However the most common metric, lines of code, is actually even less interesting to measure for object-oriented software. The less code you have written the more you have reused and that often (but not always!) gives your product a higher quality. To get a feeling of the actual code the following are examples of metrics that are more appropriate for object-oriented software

- Total number of classes,
- Number of classes reused or newly developed,
- Total number of operations,
- Number of operations reused or newly developed,

- Total number of stimuli sent.

Some metrics that are more specific are:

- Number, width and height of the inheritance hierarchies,
- Number of classes inheriting (or using) a specific operation,
- Number of classes that a specific class is dependent on,
- Number of classes that are dependent on a specific class,
- Number of direct users of a class or operation (the highest scored are candidates for components).

It is often also interesting to measure more statistical metrics. When this is the case you should measure both the average and the deviation. Some examples of such metrics are:

- Average number of operations in class,
- Length of operations (in statements),
- Stimuli sent from each operation,
- Average number of descendants for a class,
- Average number of inherited operations.

When measuring code it is important that things like comments do not disturb the metric. Therefore it is often appropriate to differentiate between lines of code, lines of comments and lines of documented code (the sum of the two previous items).

Other common source code metrics can be used with various degrees of usefulness in object-oriented software. For instance, the **McCabe cyclomatic complexity**, see McCabe (1976), measures the complexity of a graph, see Figure 15.14 The idea is to draw the sequence a program may take as a graph with all possible paths. The complexity, calculated as connections−nodes+2, will give you a number denoting how complex your program (sequence) is. Since complexity will increase the possibility of errors, a too high McCabe number should be avoided. Some standards require that no module should have a higher McCabe number than 10. Note also that the McCabe number also gives you the number of test cases to do path testing.

In object-oriented software the McCabe number as defined above will be of less interest. There are several reasons for this. The first is due to the polymorphism As we noticed in Chapter 12, every stimulus sent is a potential CASE statement in a procedural language. Since we do not know about the receivers class, any stimulus statement could hide a various number of operations – in an untyped language, in principle as many as there are classes in the system.

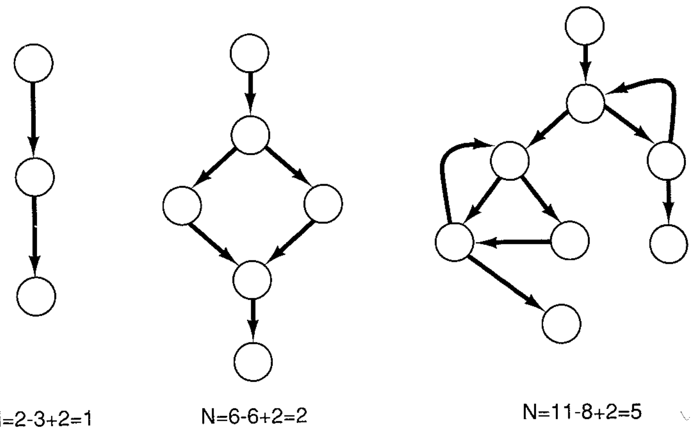


Figure 15.14 The McCabe complexity metric. N = Connection—Nodes +2.

Typically CASE statements increase the McCabe number rapidly. Thus we must decide how to handle polymorphism when using the McCabe metric; in its traditional application it is usually not very interesting

Another issue is that you would only measure the complexity of an operation since that is where you have a program sequence. It would be very unusual to have operations with a higher McCabe number, and definitely not over 10 (unless in very special circumstances), if we do not count polymorphic statements.

However, the McCabe number could be used in OOSE. The use cases connect together several objects in a specific sequence. This sequence will have a complexity that is of great interest. To calculate the McCabe number for a use case gives a complexity measure of that use case. Here the interaction diagrams are used as a tool to calculate the metric.

We have thus far mainly discussed metrics of code. What is often more interesting (and much harder) is to develop metrics to measure the quality of the design and analysis. Today we do not have any such generally applicable metrics. However, it is interesting to collect data and measures even for these models The main reason for this is for project management, where you are interested in quantitative metrics to be able to do project planning and control. Here we will give some suggestions of what to measure.

- Number of requirements,
- Number of use cases and actors,
- Numbers of objects divided on entity objects, interface objects and control objects,

- Number of subsystems,
- Number of blocks,
- Number of classes.

Besides these absolute metrics, it is also interesting to measure:

- The number of objects offering a use case,
- The correlation between analysis objects and blocks,
- The number of blocks participating in a use case,
- The number of classes in a block,
- The number of operations per block,
- The number of stimuli sent in one use case,
- The number of parameters in every stimulus,
- The locality of requirements expressed for subsystems and/or use cases

The metrics that we have discussed above should not be viewed as the only interesting measures to be taken, quite the opposite, they are only proposed metrics and should inspire you to develop your own metrics. We are at the moment not ready to give clear recommendations as to which metrics should be used when working with OOSE for quality assurance and project management.

The number of different kinds of metrics is very large, and which to choose must be decided from time to time. It is important, though, to use metrics and collect data in an organized way Actually the real problem with metrics is that they are not used. And since they are not used we will not collect any data and thus cannot validate or calibrate the metrics and therefore we do not have any really useful metrics To start to break this vicious circle we must start to collect metrics and then refine the metrics as we learn about them. However, even if we have metrics it will not be the final answer. To quote Albert Einstein: 'Not everything that counts can be counted and not everything that can be counted counts ' More on software metrics can be found in Boehm (1981) and Grady and Caswell (1987).

15.8 Summary

The introduction of a new development process into an organization must be done with great care. Such change are normally part of a long term plan for the development organization. It is essential to introduce the new process smoothly, and often a pilot project is selected to try the new process in the organization. To give the organization and the project a fair chance, a special education and training program should be planned. Risk analysis should also be done prior to starting the project to increase awareness of the risks involved, and also to take steps to manage these risks.

The focus of each project must be on the product developed. The subprocesses of the development process describe this handling and these are therefore appropriate to use for the organization of the project. The project organization is thus mainly a matter of manning all the subprocess instances. A project model to steer the project should be used. This should support the idea of milestones to follow up the project and to check that the project is on the right track all along. Prototyping is normally an integrated part of all development to increase the quality of the final product. It is essential to keep track of the documents produced and a versioning strategy should be coupled to the development process. A more complex, but often better strategy is to use incremental development. Then we have shorter turnaround time for each phase, and we also have the possibility to gain experience early.

Project staffing for OOSE involves some new roles, but also several roles that are also important in traditional development. It is essential to have a core group following the entire development that is responsible for the overall architecture and philosophy of the system. It is also important to have knowledge about the system in all phases. The use cases support such a common thread in all phases. It is therefore often appropriate to have team members following some use cases throughout development. Other roles in OOSE involve methodologists, reuse coordinators and prototypers. The team responsible for reusable components is shared among several projects and thus is a concern for upper management.

Software quality assurance should focus both on the development process and on the product developed. The fundamental techniques used include reviews, testing and metrics. Reviews is the best known technique today that gives the highest increase in quality early in development. Metrics should also be used to start collecting data to increase quality. However, there are today no well-known direct metrics that we can use to increase the quality in advance. The indirect metrics include the number of faults identified or the number of changes in a review. However, to gain quality in an ordered way tomorrow, we must start using metrics today.

16 Other object-oriented methods

16.1 Introduction

Most of the methods used in the industry today, for both information and technical system development, are based on a functional and/or data-driven decomposition of the system. These approaches differ in many ways from the approach taken by object-oriented methods where data and functions are highly integrated. In Chapter 4 we discussed the problems with a function/data approach.

To combine function/data methods with object-oriented methods in the same development cycle has been discussed, see for instance Ward (1989). In a panel discussion at OOPSLA'90 on this topic, see OOPSLA (1990), the conclusion was that any such combination should be avoided. In general, we share this view. However, some of the diagramming techniques used in function/data methods may also be used in object-oriented methods. One example of this is state transition graphs which can be used to model objects. As we discussed in Chapter 4, a shift of paradigm occurs when converting from one point of view into another. Such a paradigm shift is very complex and should in general be avoided.

In this chapter we discuss on a survey level some other object-oriented development methods. We make no claim to completeness. All of the methods aim at modeling systems in terms of objects which will form the basis of the system realization. The central problem in all object modeling techniques is to find an appropriate object structure.

It is very hard in a systematic comparison to do justice to all methods. There are so many aspects that have to be compared and real comparisons should involve parallel projects with equal members and on equal conditions. What is really interesting is to compare how the method has helped to achieve a better and more competitive product in terms of quality, productivity, modifyability and so on. What is much easier to compare is notation, concepts and easy-to-define strategies, and most comparisons are made on this level, including this one. We will only be able to give some *ad hoc* comments here on the methods.

The techniques discussed here are all what we defined as methods in Chapter 2. They are all described as step-by-step procedures. Although we have not discussed it substantially in this book, Objectory focuses on the product life cycle instead of on projects only. The Objectory process is thus actually described in terms of process descriptions where we focus on how the product is changed during its life cycle and the activities connected to these changes. Large scale industrialization is thus essential To compare methods with processes is, in our view, to compare an industrial process with craftsmanship For instance: an industrial car manufacturer has a completely different approach than that of a craftsman car developer. If we only compare the fact that both put a steering wheel on, and both put seats in the car, we have missed the point. Then we do not understand the difference in properties of the cars given by a process on the one hand and the craftsman on the other. Hence other properties than concepts are the most important.

As to the methods we will highlight in this chapter, there are some fundamental differences between OOSE and these methods. We will here emphasize some of these

(1) To work with *three object types* in the analysis model to help get a robust structure, which we have called robustness analysis, is unique None of the methods discussed here use different object types to aid the robustness of the system. Some methods use several object types, but they are not primarily intended to find the actual structure of the system.

(2) The formalization of *models* In OOSE we develop different models that are related. Our experience shows that different models are needed for different purposes, such as to communicate the requirements with the customers, for the designer and testing need certain models. Additionally, to maintain the system, models of the system are also needed These models are documented and revised separately. They are all under configuration management and version control. The other methods discussed in this chapter only work with one model, or, in some cases, with different views of the same model.

(3) The need for a *use case concept* The use case concept is central to OOSE. It is used for many different purposes If not earlier, the need for such a concept comes when the system is to be tested or when the manuals are to be written. Some of the methods discussed here have started to see the need for a use

case concept, but none have formalized it as clearly as in OOSE.

(4) The difference between *process and method*. How appropriately a method fits into a larger organization and the scalability of a method are important. None of the other methods described here claim to have a supporting process.

(5) The degree of *formalization* Software engineering is as yet not very formal. However, there is a clear degree of difference between the formalization of the methods discussed here. To formalize the concepts is essential, especially to develop CASE support that are more than a documentation tool, for example consistency control, dynamic and automatic relations between models, automatic generation of information or 'intelligent' support

Most of the comparisons between methods compare only concepts and notations and thus do not cover some of the points above. Some also compare the actual advice and rules given by the methods In this chapter we too will perform a discussion at the easiest level, namely discuss concepts and ideas of how to work with the methods. As stated, however, we believe that the 'industrialness' of the method is the most important consideration when selecting a method for an industrial organization. For academic purposes, however, the conceptual aspects may be the most interesting.

First we shall give a very brief overview of some object-oriented methods, and then we will focus on some of them. The basis of this overview is publicly available references Most methods used in the industry are not publicly available. The references we use may be old and new material which extends the methods may exist. Furthermore, several other methods do exist

16.2 A summary of object-oriented methods

There are several object-oriented development methods around. Some are given generous attention in textbooks while others have only been described in articles. Still others are in-house developments that are only used internally to an organization. We know of several different OO methods that are not easily accessible for public use. It is natural to expect that the number of methods will increase as a result of the current high interest in object-oriented techniques

Many of the earlier attempts were object-based methods, that is supporting objects but not inheritance nor classes One of the early pioneers was Grady Booch (1983) whose early method was based on the ideas of Abbot (1983) and evoloved in several steps, see Booch (1986, 1987), until his latest version **Object-Oriented Design (OOD)**, see Booch (1991). The method has evolved continously and in the latest version, earlier versions are criticized since 'it definitely does not scale well to anything beyond fairly trivial problems', see Booch (1991). We shall discuss OOD in more depth later.

Other early attempts include **Object-Oriented Systems Analysis (OOSA)** by Shlaer and Mellor (1988). This is essentially information analysis based on data modeling. OOSA fails to capture behavior and does not contain inheritance or classification. Rather, the focus is on describing relationships between objects.

The **Object Modeling Technique (OMT)** by Rumbaugh *et al.* (1991) (earlier drafts were presented by Loomis, Shah and Rumbaugh (1987)) is based on entity/relationship modeling with extensions to modeling classes, inheritance and behavior. The technique has been extended to focus on relational database design by Blaha, Premerlani and Rumbaugh (1988). We will discuss this method in more depth later.

Object-Oriented Analysis (OOA) by Peter Coad and Ed Yourdon (1991a) is a step by step method for developing object-oriented system models. OOA will be discussed in more detail later. A design method following the analysis approach has also been published, see Coad and Yourdon (1991b).

Hierarchical Object Oriented Design (HOOD), see HOOD (1989a,b) was initially developed for the European Space Agency (ESA) by a consortium consisting of CISI Ingéniere, CRI A/S and Matra Space. It has been further developed by the HOOD Working Group. HOOD has been selected by ESA as the design method for architectural design in projects like Columbus and Hermes, that is, for large real-time systems. HOOD assumes that the software will be coded in Ada. HOOD will be discussed in more detail later.

Object-Oriented Structured Design (OOSD) by Wasserman *et al.* (1989, 1990) is actually only a notation for object-oriented designs, and is an extension to the structure charts used in structured design and Booch notation techniques for Ada packages. OOSD does not include a method of its own, hence designers are expected to develop and use their own techniques.

Responsibility-Driven Design (RDD) by Wirfs-Brock *et al.* (1990) is a method which models an application in terms of classes, their responsibilities and their collaborations. Initially the systems objects and classes are identified. The system's responsibilities are analyzed and partitioned to the classes of the system. Finally, the collaborations between classes of objects that must occur to fulfill the responsibilities are defined. This gives a preliminary design which is further explored through hierarchies, subsystems and protocols.

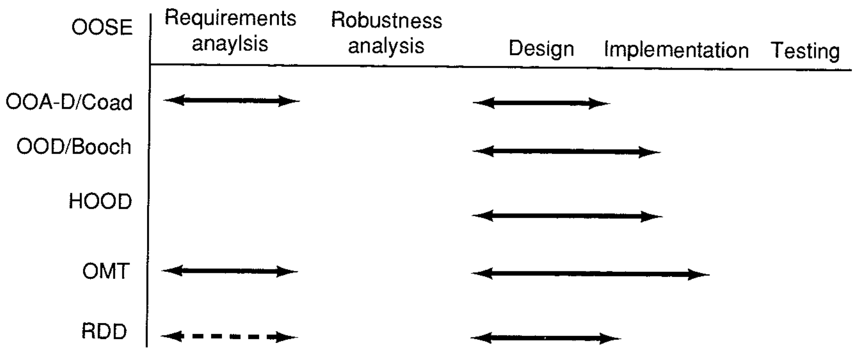


Figure 16.1 The activities of OOSE related to the methods discussed in this chapter

The method uses CRC (Class, Responsibilities and Collaboration) cards as described by Beck and Cunningham (1989). We will discuss RDD in more depth later.

The **Object Oriented Role Analysis, Synthesis and Structuring (OORASS)** method by Reenskaug *et al.* (described by Wirfs-Brock and Johnson (1990)) involves several steps in the systems development cycle. *Analysis* focuses on different **roles** played by objects in the system. *Synthesis* defines new objects by inheriting behavior from several simpler ones. *Structuring* uses a meta model to specify how objects may be bound to each other during system instantiation.

Other, more or less fully worked out, methods that are based on ideas from object-orientation include **Object-oriented Systems Analysis (OSA)** by Embley *et al.* (1992), **Object Behavior Analysis (OBA)** by Gibson (1990), **Synthesis** by Page-Jones and Weiss (1989) and Buhr (1991).

We will now discuss some of these methods in more detail. The methods we will focus on are:

- Object Oriented Analysis (OOA), by Coad and Yourdon,
- Object Oriented Design (OOD), by Booch,
- Hierarchical Object Oriented Design (HOOD),
- Object Modeling Technique (OMT),
- Responsibility Driven Design (RDD).

For each of these methods we will discuss the architecture, method and deliverables, and we also compare the approach with OOSE. We will focus on the properties that differ from OOSE Note that the comparisons are only approximate Many of the relations are not clear. We have then chosen a close concept and also used parentheses where the relation is not clear.

In Figure 16.1 we have tried to compare what phases these methods cover as compared with OOSE Note that to compare phases like this hides very many important aspects. For instance, in the requirement analysis phases the methods discussed here only use domain object models and no use case model, which is central in OOSE. Additionally, the depth of the methods also varies tremendously. Some method descriptions only have a couple of pages of substance in each phase, while others only provide tools for the phases. The ambitions of the methods also vary greatly; we will comment on some of these issues below.

16.3 Object Oriented Analysis (OOA/Coad–Yourdon)

16.3.1 Architecture

In OOA an analysis model is developed to describe the functionality of the sysem. In Table 16 1 we have related some concepts used in OOA to OOSE. The idea in the Coad–Yourdon design is to extend this model with respect to processes (tasks), human interfaces and DBMS issues. We do not consider the design technique to be worth further discussion here.

The term **Class&Objects** is introduced to mean the class and the objects in that class. The word object in OOSE is sometimes used to mean a class or a specific instance of a class. It should be obvious from the context of each occurrence what is meant Additionally, in views we draw objects This usually means the class and all instances of this class We differ between the class' association (dashed lines) and the instance's associations (drawn as full lines). This is to draw

Table 16.1 The concepts of OOA related to OOSE concepts

OOA	OOSE
Class	Class
Object	Instance
Class&Object	(Object)
Gen-Spec-Structure	Inheritance
Whole-Part-Structure	consists-of
Instance Connection	acquaintance
Message	Stimuli
Message Connection	Communication
Attribute	Attribute
Service	Operation
Subject	~View, (Subsystem)

one view instead of three. Otherwise there exists a simple mapping of concepts from OOA to OOSE. The term **subject** means a specific grouping of Class&Objects to help and guide the reader of a model. No direct correspondence to OOSE exists. Instead, we use different views for this purpose. Additionally, subjects may also be used as subsystems in OOSE, but this is not the normal case since subjects may overlap. Subjects are not treated as formally as subsystems.

16.3.2 Method

OOA uses basic structuring principles and joins them with an object-oriented point of view. The method consists of five steps of which it is proposed that the following order be followed.

(1) Finding Class & Objects,
(2) Identifying Structures,
(3) Defining Subjects,
(4) Defining Attributes,
(5) Defining Services.

Finding Class&Objects specifies how classes and objects should be found. The first approach is given by starting with the *application domain* and identifying the classes and objects forming the basis of the entire application and, in the light of this, the *system's responsibilities* in this domain are analyzed. Investigating the system environment may render tangible things that the system should know. Notes are made of information that needs to be saved about each object and what behavior each object must provide.

Identifying structures is done in two principally different ways. The first, the *generalization-specialization* structure, captures the inheritance hierarchy among the identified classes. The other structure, the *Whole–Part* structure, is used to model how an object is part of another object and how objects are composed into larger categories.

Identifying subjects is done by partitioning the Class&Object model into larger units. Subjects are groups of Class&Objects. The size of each subject is selected to help a reader to understand the system through the model. It is appropriate to use the structures identified earlier to define subjects. For example, a gen–spec structure can be grouped into one subject. When useful in guiding the reader, subjects may overlap.

Defining attributes is done by identifying information and the associations that should be associated with each and every instance. For each object you identify the attributes needed to characterize the object. The identified attributes are placed in the correct level of the inheritance hierarchy. Any instance connections are also identified by checking previous OOA results or by mapping problem domain relationships. The attributes are specified by names and descriptions. Any special constraints on the attributes are also specified.

Defining services means defining the operations of the classes. This is done by identifying the object states and defining services such as Create, Access, Connect, Calculate, Monitor an external system and so on. How the objects communicate with messages is identified using message connections, which are similar to the communication associations of OOSE. These are used to specify each operation. Here 'threads of execution' is used as a technique to follow a message sequence. Finally, services are specified by a graphical notation similar to a flow chart.

16.3.3 Deliverables

The outcome of the OOA method is documented in a special graphical notation, see Figure 16.2, and special templates for the textual documentation of classes and objects. The model is presented and reviewed in a top-down manner in the following order:

- Subject layer (only subjects are presented),
- Class&Object layer (Class&Objects and subjects are shown),
- Structure layer (structures are added to the previous layer),
- Attribute layer (attributes are added to the previous layer),
- Service layer (services are added to the previous layer).

16.3.4 Discussion

OOA is essentially an object-oriented approach and concepts like class, instance, inheritance, encapsulation and communication between objects are here essential ingredients. The Class&Object concept is introduced to avoid ambiguities that sometimes arise when there is an unclear combination of classes and objects.

The techniques of finding the objects are very heuristic. There is a lack of a distinct method to follow step by step to identify the Class&Objects of a system. OOA is designed for small systems, even though today it is being used even for large systems. The objects found using OOA appear to be the initial problem domain objects found when using OOSE. There is nothing in the method that tackles

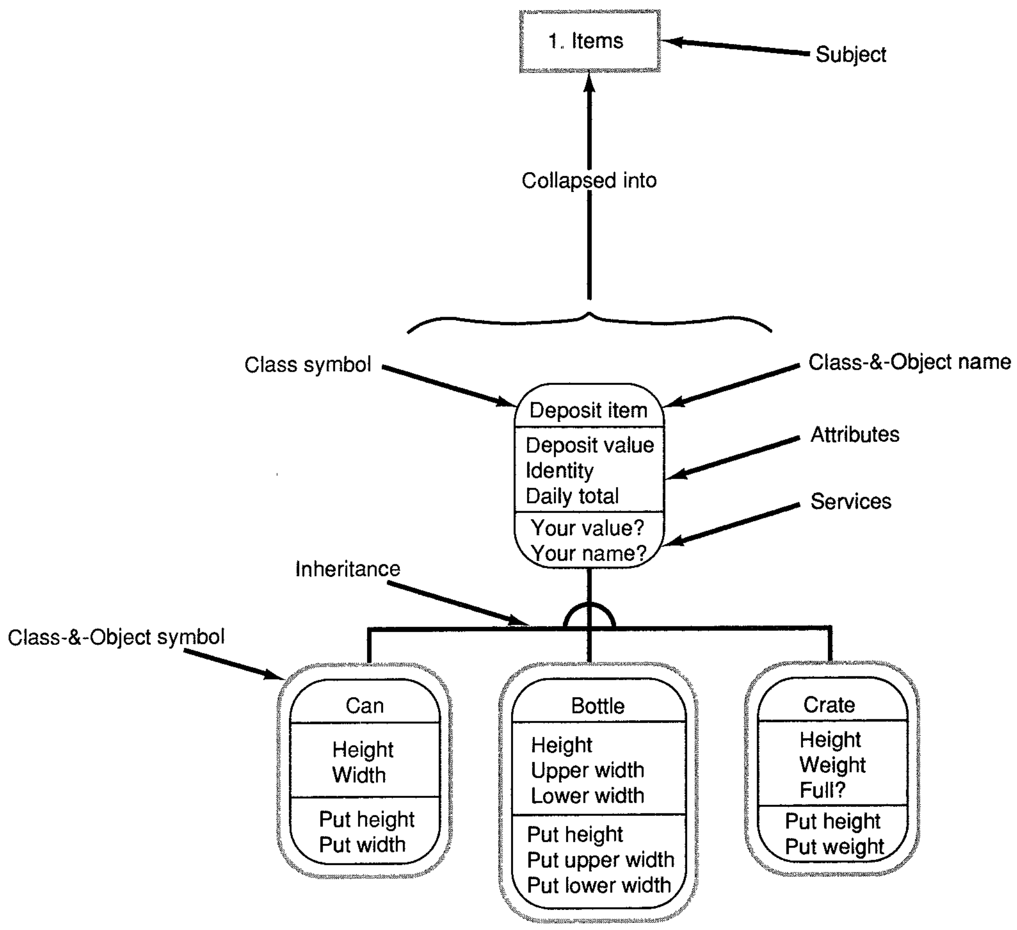


Figure 16.2 Graphical notation of OOA applied to part of the recycling system described earlier

the presentation of user interfaces of the system during analysis; this is handled by the design part.

Neither has the method any specific support to capture systematically all the dynamic roles played by the objects (c.f. the use cases in OOSE). 'Threads of execution' is one approach to this, but this is done late, to verify that the correct operations have been identified. Here use cases and an interaction diagram can be supportive tools. In larger systems a tremendeous effort is needed to capture the total interface of an object by an *ad hoc* strategy. Additionally, from our experiences it is hard to define exactly which objects are essential in the particular system without going through,

Table 16.2 The concepts of OOSE related to OOA

OOSE	OOA
Class	Class
Object	Object
Inherits	Gen-Spec structure
Acquaintance	Instance conn./Whole-Part
Communication	Message connection
Stimuli	Message
Operation	Service
Attribute	Attribute
Actor	(User)
Use Case	(threads of execution)
Subsystem	(subjects)
Service Package	—
Block	—
Object Module	—
Public Object Module	—

in detail, how the system is to be used. For example, in one registration system it may be essential to keep track of the pilot of an aircraft, but in another it may be totally irrelevant to have that information. We also think that it is extremely hard to tell which attribute relations an object should have without knowing how the object will be used. Thus it is natural to model the attributes and the relations *after* (or while) the operations are defined. Otherwise the result will be that you find a lot of attributes initially which later must be removed when you find that they are not being used.

The subject concept is similar to the views or subsystems used in OOSE. However, the subsystems are a configuration unit and a way to manage large scale systems. In particular, the service packages used in OOSE are used to make changes of a system local. A change in one functionality of the system should only affect one service package. This is not the intention of the subjects in OOA, where they are more a guide to a reader of the system model. Additionally, subjects may overlap, whereas this is not the case of the subsystems in OOSE. Views in OOSE are used to partition the model into different perspectives to guide the reader. In Table 16.2 we have related OOSE concepts to OOA

16.4 Object Oriented Design (OOD/Booch)

16.4.1 Architecture

Although the OOD/Booch method has great similarities to the OOA/Coad method in finding objects, the aim is instead to establish a grounding for implementation. In Table 16 3 we have summarily related some OOD concepts to OOSE concepts.

The concepts and techniques used in OOD are quite a few. We will here only mention the more important aspects. Booch suggest a large number of concepts, but suggests that only a number of these should be used when appropriate. What is interesting to note is that the relationships **uses** and **instantiates** (which can be further specialized) are drawn between classes, namely this is a static view of the system. Between the dynamic objects there are only lines with the messages sent between the objects attached to these lines The technique to work with meta-classes has not been described in OOD. A meta-class is a specific class' class. In OOSE these are only used by the people defining the architecture of the method, and thus not normally directly available to the practitioners of the method. Booch also focuses on the interfaces at various stages. He uses different kinds of visibility of various grouping concepts The **class category** is a grouping of classes for presentation and abstraction purposes, very similar to the subjects in OOA The physical grouping is done by using **modules**. In OOSE we use blocks for these purposes and

Table 16.3 The concepts of OOD related to OOSE

OOD	OOSE
Class	Class
Object	Instance/Object
Uses	(Communication)
Instantiates	(Communication)
Inherits	Inheritance
Meta class	(used by methodologists only)
Class category	(Block)
Message	Stimuli
Field	Attribute
Operation	Operation
Mechanism	(~use case/skeletons)
Module	Block
Subsystem	Subsystem
Process	Process

views to guide the reader. The concept of mechanism is introduced as 'structures whereby sets of objects work together to provide the behaviors that satisfy the requirements of a problem' It is not clear exactly what is meant, but the idea has some similarities to the idea of use cases. However, it seems more of a kind of framework of classes where classes can cooperate, similar to the skeletons presented in this book for RDBMS specialization.

16.4.2 The method

Booch strongly emphasizes the iterative process and the creativity of the developer as essential components in object-oriented design. The method is more a set of heuristics and good advice regarding this creative process. No strict baselines or order of work exist, but 'the process of OOD generally tracks the following order of events':

- Identify classes and objects at a given level of abstraction,
- Identify the semantics of these classes and objects,
- Identify the relationships among these classes and objects,
- Implement these classes and objects

This process is recursive and Booch also states: 'The process of object oriented design starts with the discovery of the classes and objects that form the vocabulary of our problem domain; it stops whenever we find that there are no new primitive abstractions and mechanisms or when the classes and objects we have already discovered may be implemented by composing them from existing reusable software components.' (Booch (1991)).

Identifying classes and objects involves finding key abstractions in the problem space and important mechanisms that offer the dynamic behavior over several such objects. These key abstractions are found by learning the terminology of the problem domain. This may be achieved by talking to domain experts.

Identifying the semantics involves establishing the meanings of the classes and objects identified earlier. The developer should view the objects from the outside and define the object protocol. To investigate how each object may be used by other objects is also an essential part of identifying its semantics. This is described as the hard part of OOD and may require the most iterations.

Identifying relationships involves extending the previous activities to include also the relationships between classes and objects and to identify how these interact with each other. Different types of associations are used such as inheritance, instantiation and uses

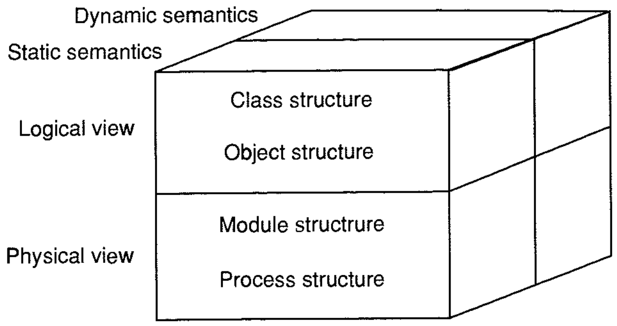


Figure 16.3 Documentation aspects in OOD

between the classes. The static and dynamic semantics of the mechanisms between the objects are also defined. The visibility between classes and objects is decided upon. This step also renders iterations in the design process.

Implementing classes and objects involves delving into the classes and objects and determining of how to implement them. A decision is made of how to use the particular programming language to implement the classes. This is also the step where components are used. The classes and objects are structured into modules. How this structuring is done is not made very clear by Booch (1991). This step may lead to the point where the whole process is performed again, but now only inside a specific class and on a lower level. We may thus end up with a hierarchy of classes or modules. This seems to result in a hierarchichal structure similar to HOOD (see below) for each class or object.

16.4.3 Deliverables

The major strength in OOD is maybe the richness of the diagramming techniques offered to the developer (is it too rich!?). By viewing the model developed from different views, a rich set of perspectives is proposed that expresses different things about the model. The perspectives are summarized in Figure 16.3.

Firstly, a separation is made between logical and physical views. The **logical view** consists of the **class structure** and the **object structure**. The **physical view** consists of the **modules** of the system and its **process** structure. All these diagrams form the basic notation of OOD, which is a static description of the system. In addition to these static diagrams, Booch uses two dynamic diagrams. The first is a **state transition diagram** that describes the semantics of instances of a certain class. The second is a **timing diagram** that describes how events happen between objects.

16.4.4 Discussion

The method is really not developed into a process, but rather a collection of techniques and heuristics that can be used when developing object-oriented systems. OOD does offer, though, a plenitude of diagramming techniques, although a complementary addition of rules for documentation is necessary; an activity that the project or organization can develop on its own.

The notation techniques have some similarities with those used in OOSE. For instance, the timing diagram has a similar purpose to the interaction diagram in OOSE, but is not yet as formally developed The state transition diagrams are described using traditional notation (Mealy diagrams), whereas OOSE may use a different notation; in this book a notation similar to SDL is described. The operations are described in plain text or in (pseudo) code.

The discussion of OOA is in some parts also applicable here. OOD seems to be an outside-in method; starting from the outside and refining each class and instance until it can be implemented by components and code. Thus it is a kind of divide-and-conquer method. The great lack, as far as we can see, is of techniques for finding the operations of each object and class. How this actual work must be done is up to the developer As to OOA, the use case concepts could be used together with OOD. In Table 16.4 we have related the concepts of OOSE to OOD.

16.5 Hierarchical Object-Oriented Design (HOOD)

16.5.1 Architecture

HOOD develops a model in a stepwise refinement that subsequently may be implemented directly in a target language. In Table 16.5 we have summarily related HOOD concepts to OOSE concepts.

The concepts used in HOOD aims at supporting an abstract view of the design and implementation which are aimed at Ada. This also explains the object-based approach, namely the lack of inheritance and pure classes. The objects are classified to support the designer **Active objects** can execute in parallel, whereas **passive objects** can only execute sequentially at a certain level. The **environ-**

Table 16.4 The concepts of OOSE related to OOD

OOSE	OOD
Class	Class
Object	Object
Inherits	Inherits
Acquaintance	(Uses relationships)
Communication	(Uses/Instantiates rel)
Stimuli	Message
Operation	Operation
Atttribute	Field
Actor	—
Use Case	(≈mechanisms)
Subsystem	Class categories/Subsystems
Service Package	—
Block	Module/Class category
Object Module	class
Public Object Module	(visibility of class categories)

Table 16.5 The concepts of HOOD related to OOSE

HOOD	OOSE
Object	Object
Active object	Block that encapsulates a process
Passive object	Block that do not directly encapsulate a process
Environment object	Actor with specified interface
Class Object	(Generic abstract object)
Virtual node object	—
Control flow	(Use case)
Use relationship	Communication
Include relationship	consists-of
Operation	Operation

ment objects represent other systems which the particular system must interact with **Class objects** are used to specify an object where some types are not fully described, namely a generic package **Virtual node objects** represent a node in a distributed system. The control flows are used to show how different objects at a specific level communicate. Different kinds of relationships between objects are used.

16.5.2 The method

HOOD is geared towards coding in Ada, but has been extended to support C++ coding also This implies that the concepts in HOOD all have Ada semantics, for instance, inheritance is not supported. The method starts with the identification of one object for the entire system, this is called the **root object**. The root object is then divided into several internal objects that are further specified. This subdivision is done recursively inside all objects until a level is reached that can be directly implemented with Ada packages. A hierarchical structure was seen as a necessity in order to distribute the development work between different contractors The order of work can thus be illustrated with Figure 16.4.

The phases are:

(1) Problem definition,
(2) Elaboration of an informal solution strategy,
(3) Formalization of the strategy,
(4) Formalization of the solution.

This is done recursively inside each object until the lowest object level is reached, the **terminal** object.

Problem definition means specifying the problem at the current level. First a statement of the problem is made to provide a context for the current object level. The requirements received from the parent object are then analyzed and structured. The purpose of the analysis is to make sure that the problem has been understood well.

Elaboration of an informal solution strategy means that a solution to the problem defined previously is outlined. This is done using natural language explaining the design for the current level of abstraction. This informal description should describe the design by means of real-world objects associated with their actions which may be performed on it. A maximum of 10 sentences is recommended.

Formalization of the strategy aims at extracting the major concepts of the informal solution strategy in order to formalize a description of the solution. This is performed in five steps:

(1) **Identification of objects** is done by extracting the nouns from the informal solution strategy and selecting the appropriate ones,
(2) **Identification of operations** is done by extracting the verbs from the informal solution strategy,
(3) **Grouping objects and operations** involves attaching each operation to an appropriate object,

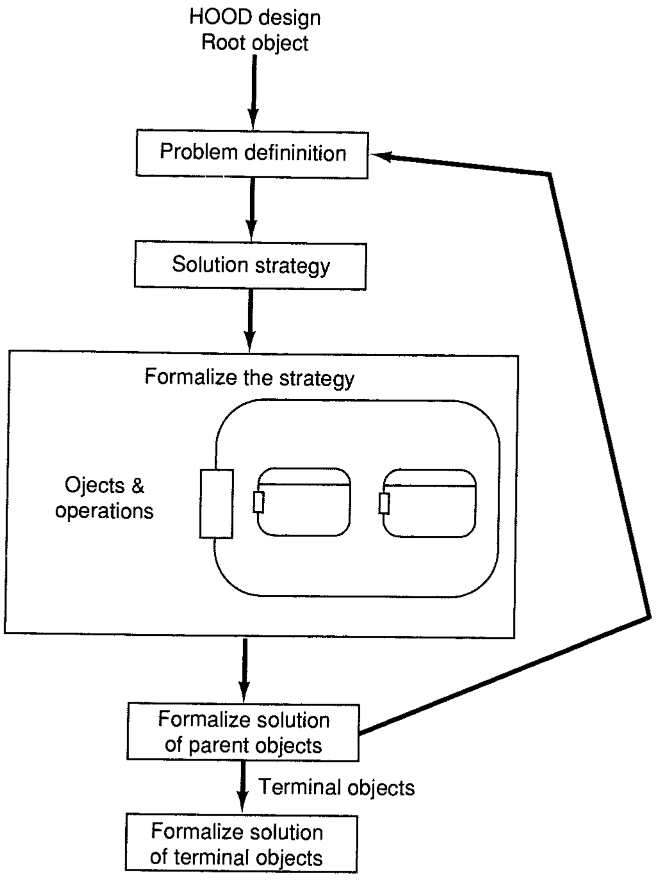


Figure 16.4 Basic design step in HOOD.

(4) **Graphical description** of objects and operation is done using the HOOD graphical formalism,

(5) **Justification of design decisions** are then performed by the designer, who explains the reasons for his decision where they are not obvious.

This entire phase may involve several iterations with the former phase.

Formalization of the solution is done by developing a formal model of each identified object. This model is referred to as the *Object Description Skeleton (ODS)* and it contains interfaces, control structures of objects and operations, together with informal comments.

This sequence is thus performed recursively until the terminal objects are reached. The structure of the system will thus become hierarchical with child objects inside their parent objects

16.5.3 Deliverables

Each object at every hierarchical level is documented in a HOOD chapter. Each HOOD chapter has subsections according to the steps described above. This provides a detailed object description which may be reviewed and further refined. Since the system will have a hierarchical structure, the documentation will also be hierarchical having the root object's documentation as its root document.

The formal description of the object, written during the step 'Formalization of the solution', is written using an Object Description Skeleton (ODS) which is thus a subsection of a HOOD chapter. The ODS is written using a special Programming Design Language (PDL). This language has a similar syntax to Ada and the ODS is also used to generate Ada code (and possibly also C++). In the ODS, pseudo-code or Ada may be used to specify each operation or each control structure (task). Further refinements, in the generated Ada code, of the object operations are performed by the designer Each object generates a package in the Ada code and each child object is included in its parent object by the WITH clause in Ada

The definition of the syntax used in ODS makes it possible for contractors using different development platforms to exchange design documents. The translation of ODS to Ada code can be automated and is normally part of the development tool

16.5.4 Discussion

HOOD is fully geared towards Ada, which has both pros and cons. The advantages are of course that, during the design, Ada syntax and semantics may be used in the descriptions and thus no new language has to be learned by the system developers; this also helps code generation later. The existence of an Ada standard is thus essential since many developing organizations may be involved in a development Support is also given to define process communication and synchronization and the terminology of this is also based on Ada concepts.

If any other programming language than Ada is to be used, major changes in documentation techniques and possibly also of the method need to be made. For instance, the inheritance mechanism

Table 16.6 The concepts of OOSE related to HOOD concepts

OOSE	HOOD
Class	(Class Object)
Object	Object
Inherits	—
Acquaintance	(include relationship)
Communication	(Use relationship)
Stimuli	(Stimuli)
Operation	Operation
Attribute	(type)
Actor	(Environment object)
Use Case	(~control flow)
Subsystem	(Root) object
Service Package	—
Block	~Object
Object Module	Terminal object
Public Object Module	Provided interface

is not supported by HOOD. Additionally, there is no analysis view which is independent of the implementation environment. However, of course, HOOD could be extended by an analysis technique

The method gives the basic design step, but does not give any help in finding the appropriate object structure. Actually HOOD does give strong support for containment structures, but not for other structures such as use or inheritance structures. An earlier method proposed by Booch, namely to extract nouns and verbs from a textual description, now abandoned by Booch himself, is recommended to support the findings of objects and operations. It is natural to expect that this recommendation will evolve into other strategies. There is nothing to support dynamic flows over several objects.

The method may be characterized as object decomposition. This is a very complex task and it suffers from several of the flaws of the traditional functional decomposition methods. The major flaw is of course that the most important decisions must be taken very early in the development when dividing the entire system into its components In Table 16.6 we have related OOSE concepts to HOOD concepts.

16.6 Object Modeling Technique (OMT)

16.6.1 Architecture

Object Modeling Technique (OMT) covers analysis, design and implementation In Table 16.7 we have summarily related OMT concepts to OOSE concepts.

The OMT technique probably has the most various concepts of all the methods highlighted in this chapter. Table 16.7 only represents the more important ones. However, the concepts are in most cases well defined and are also related to each other. For instance, we quote, 'a **Link** is a physical or conceptual connection between object instances' whereas 'an **association** describes a group of links with common structure and common semantics'. Link may have attributes, which is an interesting property. In this book we have not defined the acquaintance association in this way, which, however, is sometimes a useful extension. **Aggregation** is used to describe how objects represent assemblies of other objects. **Events** are very similar to the stimuli used in OOSE. **Scenarios** are described in the book in terms of similar ideas to those we have used for use cases in OOSE. However, they are not as formalized and central as in OOSE. Attributes and subsystems are similar to the ones in OOSE, whereas **modules** are used to group classes for manageable purposes. These are also the lowest level of subsystem. There is no special

Table 16.7 The concepts of OMT related to OOSE

OMT	OOSE
Class	Class
Object	Object
Generalization/inheritance	Inheritance
Link	Acquaintance
Link attribute	—
Aggregation	consists-of
Operation	Operation
Event	Stimuli
Scenario	~Use case
Attribute	Attribute
Subsystem	Subsystem
Module	(Block/Service Package)
Sheet	View

notation for modules; they are merely listed on top of each sheet. A **sheet** is a single printed document that shows part of the model

16.6.2 The method

The method consists of four phases: analysis, system design, object design and implementation. Three models of the system are developed initially and then refined in all these phases. The models are:

- The **object model** describes the static structure of the system with classes and their relationships,
- The **dynamic model** captures the temporal aspects of the object model with events and states of the objects,
- The **functional model** describes the computation as how output values are derived from input values, that is, mainly in terms of the operations of the objects.

The purpose of **analysis** is to model the real world so that it can be understood. The analysis model is composed of the three submodels mentioned above. Initially the requirements are stated in a problem statement. From this statement, the classes relevant in the domain are extracted and also their relations and attributes. Together with inheritance and modules this will be the *object model*. The *dynamic model* is developed by looking for events in the object model. The scenarios are used to develop event traces which are very similar to the interaction diagrams in OOSE. From these the events can be identified. From these events state diagrams are developed for the classes. The *functional model* is a data flow diagram of the actual transactions in the system. These diagrams show dependencies between the operations represented by the processes in the diagrams. Additional operations are found from the reading or writing of attributes, from events in the dynamic model and also from actions in the state diagrams. These models are developed in an iterative manner.

For **system design**, a high-level strategy is developed. The system is partitioned into subsystems and also to processors and processes (tasks). Strategical decisions are also made of the use of DBMS, global resources and implementation of control This is basically similar to the first activity in construction in OOSE, namely the identification of the implementation environment and the decisions concerning certain strategic questions.

The **object design** phase aims at defining the objects in detail. This includes defining their interfaces, algorithms and operations.

The objects discovered during analysis serve as skeletons for this activity. Here the three models are integrated to design the objects. New objects are also introduced to store intermediate results. Optimization of the design is also done. The classes are packed into modules.

The last step in OMT is the actual **implementation** of the objects. This is done in terms of a number of guidelines and coding rules for good object-oriented coding style The implementation of both object-oriented and non-object-oriented languages is described.

16.6.3 Deliverables

The actual deliverable consists of the three models:

- Object model
- Dynamic model
- Functional model

which are related to each other. The notation used are mainly conventional techniques such as combinations of entity/relationships diagrams, data-flow diagrams, event traces and state transition diagrams. The overall system design is described in a document of its own together with high level strategy decisions. Additionally, the actual source code is also delivered grouped into modules.

16.6.4 Discussion

The OMT method has very many different techniques and guidelines to support actual development of a system. Additionally, it is the only method presented in this chapter that makes a serious attempt to support both analysis and construction as defined in OOSE

The method is the most developed of the ones presented in this chapter. OMT includes several different concepts, which gives the method a high granularity for expressing different modeling situations. It is thus very powerful, but many concepts require that the notation used, and their semantics, be formally defined to develop a consistent model. Otherwise it will be very difficult to learn and use the appropriate method. It is not very easy to see how all the techniques in OMT can be used to form consistent models, and the relations between the three models are not obvious at all stages.

Table 16.8 The concepts of OOSE related to OMT concepts

OOSE	OMT
Class	Class
Object	Object
Inherits	Generalization
Acquaintance	Link
Communication	(data flow)
Stimuli	Event
Operation	Operation
Attribute	Attribute
Actor	—
Use Case	scenario
Subsystem	Subsystem
Service Package	—
Block	Module
Object Module	Classes
Public Object Module	(Service)

OMT is affected by relational database design, and several concepts essential for such work are introduced. The transition to a relational database design is presented in detail.

The method presents interesting ideas concerning the use of events which have many similarities with those used in OOSE. OMT also uses a technique to abstract states and have states at several different levels.

The approach is thus the most ambitious of the methods presented in this chapter. It also gives a large amount of good and explicit advice that is needed in common situations. In Table 16.8 we have related OOSE concepts to OMT concepts.

16.7 Responsibility-Driven Design (RDD)

16.7.1 Architecture

In Responsibility-Driven Design (RDD) a model is developed from the requirement specification which should be the basis for actual implementation. In Table 16.9 we have summarily related RDD concepts to OOSE concepts

RDD supports the basic concepts of object-orientation, such as classes, objects and inheritance. For each class, different **responsibilities** are defined which specify the responsibilities or the roles of

Table 16.9 The concepts of RDD related to OOSE

RDD	OOSE
Class	Class
Inheritance	Inheritance
Responsibility	(part of a use case associated to one object)
Collaboration	Communication, acquaintance
Contract	Public class
Subsystem	Subsystem

the objects and the actions of the objects. This corresponds to the part of a specific use case that an object should be responsible for. To fulfill these responsibilities, the classes need to collaborate with other classes using **collaborations** which show how the objects interact. The responsibilities are further refined and grouped into **contracts** which define a set of request that objects of the class can support. In OOSE this is similar to defining the public classes of a block. These contracts are further refined into **protocols** which shows the specific signature of each operation, just as the classes in OOSE are refined. To ease the design, subsystems are introduced. A **subsystem** groups a number of classes and subsystems of lower level, and thus abstracts a certain functionality. A subsystem also has contracts which should be supported by some class in the subsystem.

16.7.2 The method

The method comprises a number of phases where each phase is described as a number of activities. The **exploratory phase** consists of:

(1) **Classes**. This is accomplished by reading the specification and extracting the essential nouns. More essential nouns are found from physical objects in the domain, conceptual entities object, external interfaces and larger categories of classes. Find abstract classes that share properties from several other classes, such as from the categories of classes. State the purpose of each class.

(2) **Responsibilities of each class**. By looking for the verbs in the requirement specification we can find the actions of the objects in the system. Wherever information is mentioned this could be allocated to one class. Furthermore, refine the purposes of

each class to specific responsibilities. Look also for relationships between classes. Try to distribute the responsibilities evenly on the classes and decentralize the behavior as much as possible over the classes.

(3) **Collaborations between classes.** The actual collaboration between classes is found by asking questions like 'With what does this class need to collaborate to fulfill its responsibility?', 'What other classes need the result?' and 'What needs to make use of the responsibilities of this class?'. Also the different kinds of relationships, like is-part-of, can yield new collaborations between classes.

The **refining phase** consists of

(1) **Hierarchies between classes.** Here the inheritance hierarchies between classes are further refined. By using *Venn diagrams* over responsibilities, abstract classes can be extracted. It is also emphasized that one must develop a kind-of hierarchy, namely a type hierarchy as discussed in Chaper 3. The classes are marked as abstract or concrete, with no abstract classes inheriting any concrete class. Common responsibilities are placed as high up as possible. Contracts are also identified on the various levels in the hierarchy by grouping responsibilities used by the same clients.

(2) **Subsystems** are groups of classes and are identified to simplify the patterns of collaborations. The use of collaboration graphs is then essential. These show how the different classes and subsystems collaborate to fulfill their responsibilities. All possible paths between classes in the system could be captured in these graphs. We look for frequent or complex collaborations and also try to name the subsystems of a certain functionality. Some contracts are made public to the subsystem. The collaborations internal to and between the subsystems are simplified by minimizing the number of collaborations in the design and also by minimizing the number of classes a subsystem delegates to.

(3) **Protocols.** Here the responsibilities and contracts are further refined to pure protocols of specific signatures of each operation. Emphasis is placed on making the protocols as simple and useful as possible by naming them carefully and giving them reasonable default values for parameters. A design specification of each class, subsystem and contract is written.

Responsibility-Driven Design thus gives a design specification to be implemented. The idea then is to do a straightforward implementation, that is, directly map the design on the implementation. However, new classes will occur and also previously existing classes will be used for the implementation

In all steps, walkthroughs of different scenarios (use cases) are highly recommended to help in understanding the model, but also to check that required behaviors of the system are not omitted.

16.7.3 Deliverables

The output of RDD is a design specification consisting of:

- A graph of each class hierarchy,
- A graph of the collaboration for each subsystem,
- A specification of each class,
- A specification of each subsystem,
- A specification of the contracts supported by each class and subsystem

The result should be appropriate for direct implementation.

16.7.4 Discussion

RDD is a design technique that uses informal techniques and guidelines to develop an appropiate design. The strategy to find classes and their properties relies greatly on the skill of the individual designer. RDD uses the informal, but apparent, technique of CRC cards whereby classes, responsibilities and collaborations are captured in an iterative manner. This will probably make the technique hard to scale to larger developments. However, some new and interesting techniques are introduced, among them the concept of contracts and the use of the subsystem in combination with the contract concept. A thorough use of these concepts should make it possible to implement several objects in parallel.

The technique of identifying classes in RDD is the traditional technique of object-oriented development. We have previously discussed the drawbacks of such a strategy. Additionally, great emphasis is placed on evenly distributing behavior over these classes. Hence the method recommends a decentralized, or stair, structure or delegation of interaction in all cases. We have also discussed our view on this matter

The use of collaboration graphs and contracts has similarities to the technique of interaction diagrams presented in this book. Also,

Table 16.10 The concepts of OOSE related to RDD concepts

OOSE	RDD
Class	Class
Object	Object
Inherits	Inheritance/Hierarchy
Acquaintance	(Collaboration)
Communication	Collaboration
Stimuli	Message
Operation	Method
Attribute	Attribute
Actor	—
Use Case	(scenario)
Subsystem	Subsystem
Service Package	—
Block	—
Object Module	Classes
Public Object Module	Responsibility/Contract

the use case view over the objects is reminiscent of this. Since much emphasis is placed on walkthroughs of scenarios without any formal technique, the introduction of use cases and the interaction diagram in RDD would strengthen the method, we think.

RDD, of the techniques presented here, is the one that most easily can be related to formal techniques in computer science. Here OOSE and RDD are similar. A semiformal computational description technique is used in both methods. The simple and formal techniques are also evident, since RDD has been developed from a Smalltalk background and has many fundamental similarities with Smalltalk. In Table 16.10 we have related OOSE concepts to RDD concepts.

16.8 Summary

There are a large number of object-oriented methods around. Only a few of them are publicly available An increase in the number of methods available can be expected. Many of the methods can be classified as object-oriented in the sense that they fully support the core concepts of object-orientation.

We have only mentioned some of the most well-known methods and studied some of them in a little more detail. Our comparison is very simple and confined to a concept-based comparison between them and OOSE. This is actually not the most interesting

Table 16.11 The concepts of OOSE related to the concepts of the methods discussed in this chapter

OOSE	OOA	OOD	HOOD	OMT	RDD
Class	Class	Class	(Class Object)	Class	Class
Object	Object	Object	Object	Object	Object
Inherits	Gen-spec structure	Inherits	—	Generalization	Inheritance/ hierarchy
Acquaintance	Instance conn./whole-part	(Uses relationships)	(include rel)	Link	(Collaboration)
Communication	Message connection	(Uses/instantiates rel.)	(Use relationship)	(data flow)	
Stimuli	Message	Message	(Stimuli)	Event	
Operation	Service	Operation	Operation	Operation	Method
Attribute	Attribute	Field	(type)	Attribute	Attribute
Actor	(User)	—	(Environment object)	—	—
Use Case	(threads of execution)	(≈mechanisms)	(~control flow)	scenario	(scenario)
Subsystem	(subjects)	Class categories/ subsystems	(Root) object	Subsystem	Subsystem
Service package	—	—	—	—	—
Block	—	Module/class category	~Object	Module	—
Object module	—	class	Terminal object	Classes	Classes
Public object module	—	(visibility of class categories)	Provided interface	(Service)	Responsibility/ contract

comparison from a large scale perspective, but it is the easiest one to make Other areas to consider include:

- How the method enforces production of a robust structure,
- The model concept,
- The use case concept,
- The degree of appropriateness for large scale development,
- The degree of formalization of concepts.

The model developed in OOA/Coad–Yourdon is very similar to the domain object model in OOSE, excluding the use cases that tie them together. Here the model is refined to define operations in a detailed manner using special charts.

The model developed in OOD/Booch is similar to the domain object model in OOSE, but also to the design model. However, there is an unclear mapping from domain objects (logical view in OOD) to the design model (physical view in OOD).

The model developed in HOOD is similar both to the domain object model and also to the design model of OOSE. The domain object(s) are refined in a hierarchical manner into descriptions that can be further translated to code.

The object model developed in OMT/Rumbaugh *et al.* is similar to the domain object model in OOSE. Related to this is also the dynamic model describing the interaction between objects (like interaction diagrams in OOSE) and their internal states (like state transition diagrams in OOSE), and the functional model which describes the actual operations. These models are refined and implemented.

The model developed in RDD/Wirfs-Brock *et al.* corresponds to the domain object model and the design model in OOSE.

Table 16.11 relates the concepts of OOSE to the concepts of the methods highlighted in this chapter.

Appendix A: On the Development of Objectory

A.1 Introduction

Here we briefly describe our work with the development of Objectory, of which this book has described the fundamental techniques. It is by no means complete, but we hope to give you an idea of the way we are thinking.

Underlying Objectory are some basic assumptions and problems.

- System development projects will involve a great number of people who will have to cooperate in an efficient manner. Sometimes as many as several thousand developers may be needed to develop and maintain a single system.
- The software cannot be viewed or touched but is much more abstract than corresponding components in other types of engineering. In the building industry it is always possible to 'see' how the component parts fit together. In a computer system it is very difficult to understand how the different parts interact.
- Software development is a very young branch of industry, only a few decades old. However, in the future it will be necessary to view it as an engineering discipline with industrial techniques.
- System development is a very complex task involving very many significant details

Competence, as well as good tools, is necessary to carry out something complicated. Only poets and dreamers dare 'go for the moon' without years of preparation and training. Obviously this also holds true for other activities of large complexity, such as system development A system development process should offer the aid that the developers need.

The system development techniques that were known to us when this work began (around 1978) were rather simple and limited, at least in comparison with techniques used in more mature disciplines such as civil engineering and electrical engineering. Most development methods, if described at all, are still described *only* in textbooks. To someone who is going to learn the basics of a technique, a textbook, in combination with teaching, will be necessary. A textbook will give a good introduction to the technique; moreover, it can constitute a 'management overview'. This book is structured more or less like a textbook. Below are a few properties that most textbooks have (and should have).

- Description is geared towards new development, that is, it covers only the first part of the life cycle.
- The theoretical substance fills some 50 pages even if the whole text may be up to 500 pages. The rest of the material consists of supporting examples and general information about software engineering.
- Rules and criteria for how the work should be carried out to yield a good analysis, a good design and soon, usually take no more than a few pages.

Anyone can understand that there is an imbalance if the development of a software system, which may cost millions of dollars, is based on instructions whose substance covers only some 50 pages. When the textbook has fulfilled its mission, the developer should therefore have a description, practically at once, of how he or she is to work in practice. He or she will also need a full description of his role in the development work, for example:

- The developer has to know what different tasks he is to perform and with whom he is to cooperate: for each step he will need information about what input will be needed, what output will be required, what is to be done, and how it can be *done well*.
- The developer will also have to learn which tools he can utilize, and find out how these tools should be used in the project.

In other words, the developer will need to have access to a large set of descriptions of different aspects of the system development process. The necessary descriptions will together make a text of some 1000 pages – a volume of considerable size. All the developers do not take part in all the phases of the development, however. All of them need therefore not be familiar with the details of all the steps, even if knowledge of the whole process is stimulating and necessary for efficient work.

A.2 Objectory as an activity

The system development process is therefore a very complicated **activity** Moreover, this activity is continually being developed and improved, just as with any other system. In other words, it must be possible to do maintenance on the process. We have therefore viewed the process as an activity of an enterprise, and used enterprise modeling to describe the process. The enterprise modeling technique is in turn based on Objectory itself. We will here illustrate how we have used these concepts to develop Objectory. Then we will briefly introduce the Objectory enterprise modeling technique (which is conceptually very similar to the system development technique).

Instead of viewing a system, we consider an enterprise or an organization. A specific flow through the organization we call a **handling case**, analogous to a use case. What the enterprise handles we call **handling entity objects**, analogous to domain objects or entity objects. The actual activities in the enterprise we call **processes**, somewhat analogous to control objects. These concepts will not be further defined here. Moreover, this is a simplified description of Objectory enterprise modeling, in which not all the object types are used.

When analyzing a system development process, we use a similar method for our work as in the analysis of any other system. Thus the system development process can be applied to itself.

A.2.1 Handling cases

That which is to be performed by an enterprise is defined through its handling cases. A handling case is a way for the enterprise to function in order to attain a goal. Which handling cases are included in an enterprise is thus determined by the goals of the enterprise.

In an enterprise where computer systems are developed, the two basic handling cases, *New Development* and *Further Development*, will soon be found, and so will the handling cases *Give a tender* and *Handle errors*.

Figure A.1 shows how the system development organization can be modeled as a number of handling cases cooperating with the external entity objects, customer, producer and customer support.

A study of the handling cases *New development* and *Further development* will reveal that these handling cases have many parts in common. An abstract handling case *Development* can therefore be identified which both *New development* and *Further development* will

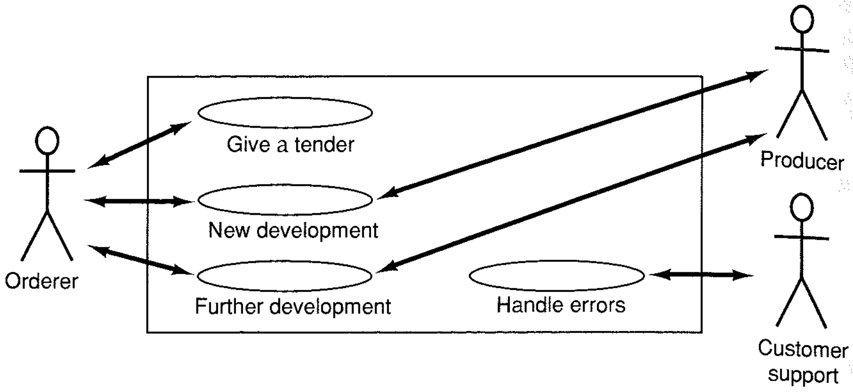


Figure A.1 In system development there will be a number of handling cases that offer their services to external users

use. In it can be described all the common parts, such as how an interaction diagram should be drawn.

Further study of the handling cases will reveal excellent chances to incorporate reviews. Reviews really have nothing to do with the development work as such, but they are vital to achieve a high quality. It is possible to describe reviews as a handling case with an extends association to the handling case *Development*, see Figure A.2. *Reviews* will then be performed at a number of well-defined points in the development.

In the handling case description it is possible to read how the activity will attain its goals, but to understand how the handling

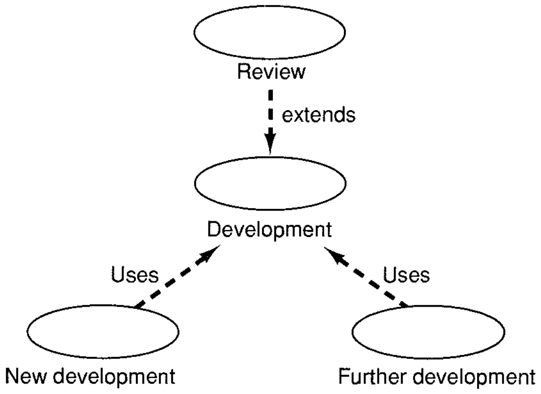


Figure A.2 *Reviews* is performed as extension to *Development*, which is used by both *New Development* and *Further Development*.

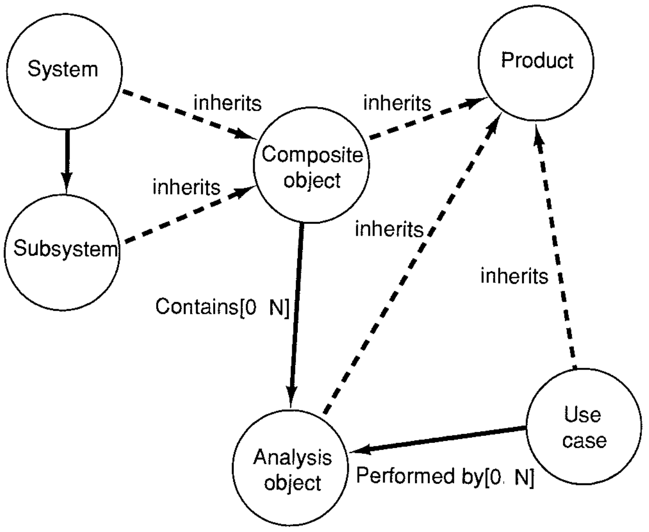


Figure A.3 A handling entity object can be a primitive or composite object

case is performed, it is necessary to see how the handling case is performed by cooperating processes operating on handling entity objects. We will return to this after we have discussed handling entity objects and processes.

A.2.2 Handling entity objects

In system development, entity objects are primarily used to model entity objects for information. This also holds for handling entity objects used in enterprise development. The handling entity objects required to model a system development process are principally the concepts used in the system development. Thus the following handling entity objects are therefore found used in analysis: **use cases, systems, subsystems** and **objects**. Actors are not a handling entity object since we do not describe them The handling entity objects are either simple or composite, see Figure A.3. Composite objects are packaging objects which contain other handling entity objects. This containment hierarchy ends with the analysis objects (entity objects, interface objects and control objects). Each use case is expressed in terms of the objects that cooperate to perform the use case.

All of these concepts will be handled as **products** in an organization. All the handling entity objects have certain parts in

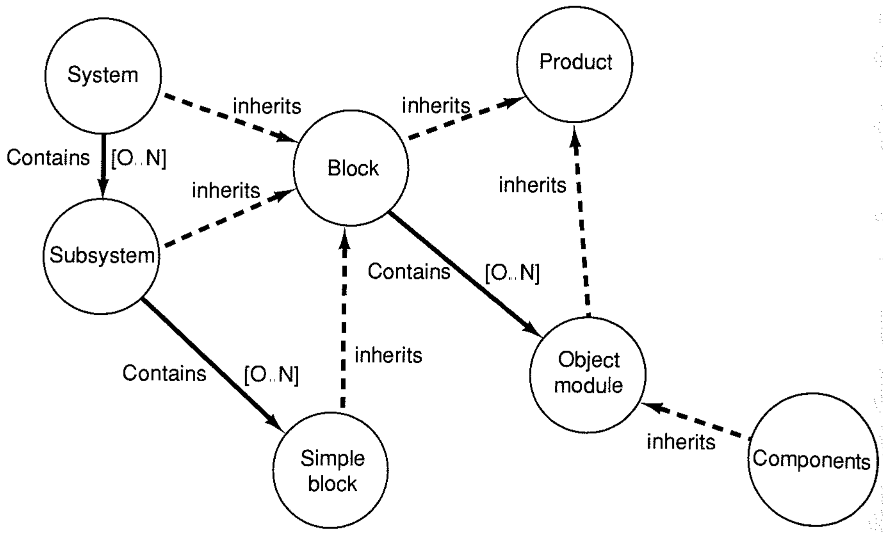


Figure A.4 All types of blocks can contain classes

common. There is one person or position responsible for each handling entity object, for instance. Only this person is allowed to make changes in the handling entity object. Others can hand in suggested changes, but it will always be the responsible person who sees that the changes are made. Such common properties and rules can be formulated in the handling entity object Product, which all the other handling entity objects will inherit.

In construction, a system will be modeled as a hierarchy of blocks and object modules. Here we also have composite objects that can contain blocks and object modules. Normally, however, it is the low level blocks (simple blocks in Objectory) that contain object modules, but it is perfectly possible to have, for instance, subsystems containing object modules directly. Components are a special type of object module which can be used in several blocks supporting the application object modules of the block, see Figure A.4.

When a system is to be delivered, we say that an **instantiation** will be made of a subset of the blocks in this designed system (a customer may not want all the parts of a system). Which blocks will be instantiated will depend on what **service packages** the customer has ordered. The delivered system thus consists of a set of block instances which contain those classes, namely the executable code, which have been received from the corresponding descriptions.

Once the installed system is performing well, the classes that form part of the block instances will be **instantiated**. Then the objects

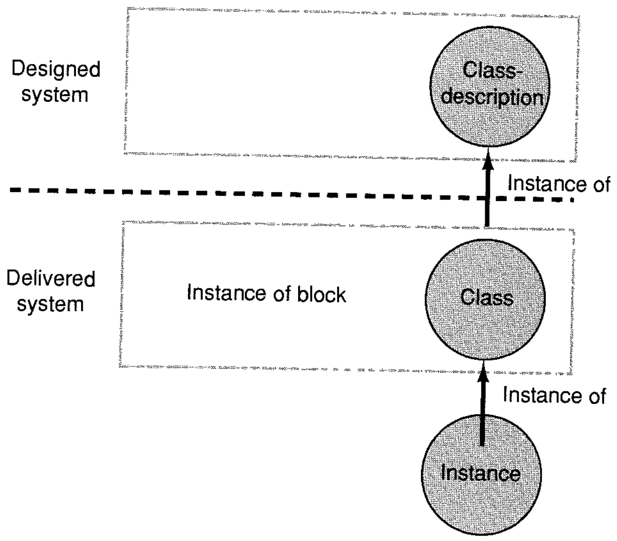


Figure A.5 Instantiation takes place at different levels: the delivered system will be instantiated from the designed system, and the executing object instances will be instantiated from the classes

that really perform the handling of the system will be created (instantiated). This is what instantiating means in traditional object-oriented programming. The relationship is shown in Figure A.5. We thus use the instantiation concept on two levels.

Like entity objects, handling entity objects have associations, attributes and operations. Some associations have already been illustrated above. All the information that a handling entity object must contain can be modeled by means of the attributes. Information stored in the attributes should be homogenized so that there is no redundancy between the attributes.

The most common way to view the attributes of a handling entity object is in some form of **document**. It would be easy to make the mistake of saying that document types are equivalent to attribute types, or even worse, to handling entity objects, but that would be wrong because the content of different document types can and should show redundancy. The name of a certain subsystem, for instance, will be found in at least three document types: in the description of the subsystem, the overview of all the subsystem and in any diagram containing the subsystem. A document is rather a *compilation* of information from the model of the handling entity objects, a compilation adapted for a particular type of reader. The same information can therefore be presented in different ways to

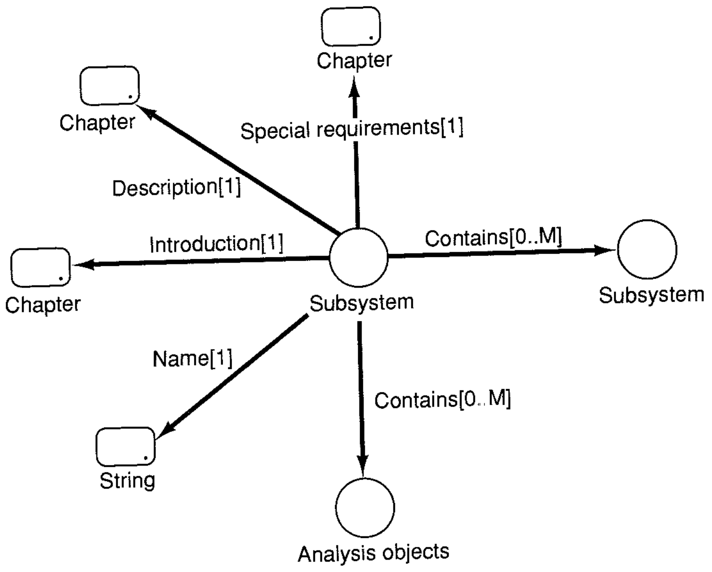


Figure A.6 The handling entity object subsystem with attributes and associations.

different categories of readers, that is, we can define semantic views or perspectives on the model.

A subsystem, for instance, must have at least four attributes: Name, Introduction, Description, and Special requirements, see Figure A.6. The attribute Name is only a String while the other attributes consist of a text that can contain arbitrary subsections, namely chapters. The subsystem handling entity object contains objects (which the subsystem, so to speak, consists of) and it can have at least one association to another subsystem object, namely contains.

When a subsystem is presented in the **survey** of all the subsystems, only the Name and the Introduction of the object will be presented. This document is meant for those who only need a summary of the subsystems. In the **description** documents of the subsystem, meant for those who are developing the subsystem, all the information will be found, as in the model above, including the names of the associated objects. When the subsystem is shown in certain **diagrams**, only its name and the contained objects will be shown The essential point here is that a document is only a kind of report with certain information from a set of handling entity objects.

Each document has a unique identity. There will always be an appointed person who is responsible for the document. These are typical further attributes of a subsystem. There are, moreover, established rules for how changes of a document are to be handled. Documents will have to be changed whenever the state of the handling entity object is changed, because the information of the attributes will change then.

A system thus consists of a number of handling entity objects. These form a structure with the handling entity objects as nodes. A set of documents goes with the handling entity objects, and together with the structures they give a total view of the system. A system can therefore be said to be described by a number of documents that constitute part of a document structure.

The handling entity objects' counterparts to operations are called **handlings** A handling thus defines some work on a handling entity object. For instance it must be possible to *create*, *change*, and *remove* handling entity objects. Any change must be *registered* and form part of a new revision state for the system as a whole. In other words, there must be handlings which register state changes in the documents belonging to a handling entity object. There are also handlings which apply to groups of handling entity objects; it should, for instance, be possible to produce all the documentation that concerns a particular revision state of a system. In addition, there are handlings that are specific for a particular handling entity object. For the handling entity object subsystem it must be possible to add new objects which the subsystem is to comprise.

A.2.3 Processes

Well-defined activities within an enterprise can be described in terms of **processes**. Each type of process can then be made responsible for any work related to a particular type of handling entity object. Detailed specification of an interface object, for instance, will be performed by a process responsible for that particular object

Each process describes how to perform the activities that the process is responsible for. The description of a process can therefore be said to be a kind of work description.

In principle, a process is needed to specify each handling entity object. In this way there will be a pair of objects that cooperate through the whole activity, a process and a handling entity object.

A.2.4 Handling cases in terms of handling entity objects and processes

Each handling case is performed through a number of interacting processes and handling entity objects This implies that a handling

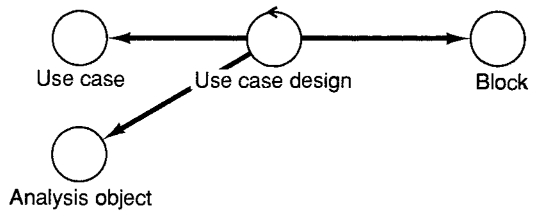


Figure A.7 The subcourse 'design of a use case' in terms of processes and handling entity objects.

case is distributed among the different objects. Normally, the main part of the 'intelligence' of the handling case will lie in the process. This is where it is described how the work should be carried out and how decisions are to be made about various steps.

In the handling case *New development*, a number of subcourses can be identified. Each one of these subcourses can be described separately. One such subcourse comprises the steps that are to be taken when describing how a particular use case is to be realized in terms of blocks. That course is driven by a process for use case design. The process for use case design starts from the analysis description of a handling entity object use case and the involved objects, and it transforms this into a description of how blocks are to cooperate in the use case. What happens, in practice, is that a designer, starting from the analysis description, draws a set of interaction diagrams for the blocks in the use case. In other words, this activity formulates requirements on the blocks participating in the use case, see Figure A.7.

A.2.5 Enterprise construction

With an enterprise analysis of system development behind us, we can now understand the activity at a logical level. In other words, we know what needs to be done in the activity for the goals of the activity to be attained. We also have an idea of how these tasks should be split up into well-defined activities. Now this should be implemented, that is, the logical picture should be translated into an organizationally functioning activity. We call this phase of the enterprise development **enterprise construction**.

We will now introduce the term **activity block**, which corresponds to the term block in construction. An activity block is a well-defined part of the activity, often corresponding to organizations or

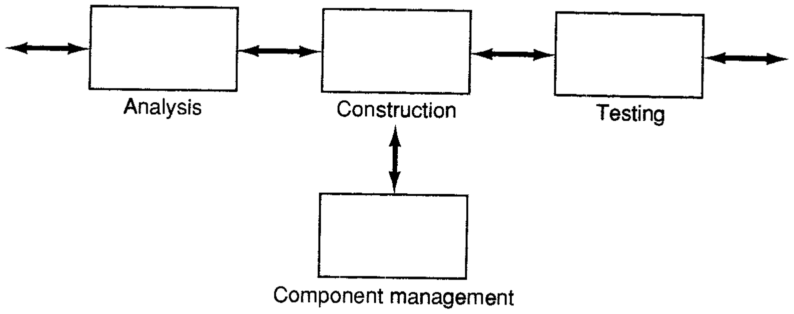


Figure A.8 System development can be divided roughly into four subsystem activity blocks.

groups within a corporation. When we define an activity, we start from the handling entity objects and processes we have identified. The different blocks can then be grouped together into larger activity blocks, until the subsystem activity blocks are reached. If Objectory is divided into subsystem activity blocks, four different blocks will be found: analysis, construction, testing and component management, see Figure A.8.

A.2.6 Handling cases in terms of activity blocks

It will now be possible to describe each handling case in terms of the activity blocks. Each handling case will then be realized as a number of cooperating activity blocks. The dynamics in the activity block model will be described by means of interaction diagrams, which show how the activity blocks communicate to realize the handling cases. Figure A.9 shows a rough interaction diagram for the handling case *New development.*

Note that this shows only one iteration through the different steps of a project. In a real case, the steps will be iterated, and the course of events will not be quite so simple.

A.2.7 Realization of the activity blocks

Implementing the separate activity blocks implies first of all describing how the tasks assigned to them are to be carried out. The interfaces between different tasks will be given by the handling cases (interaction diagrams) where the activity block participates. In other words, we

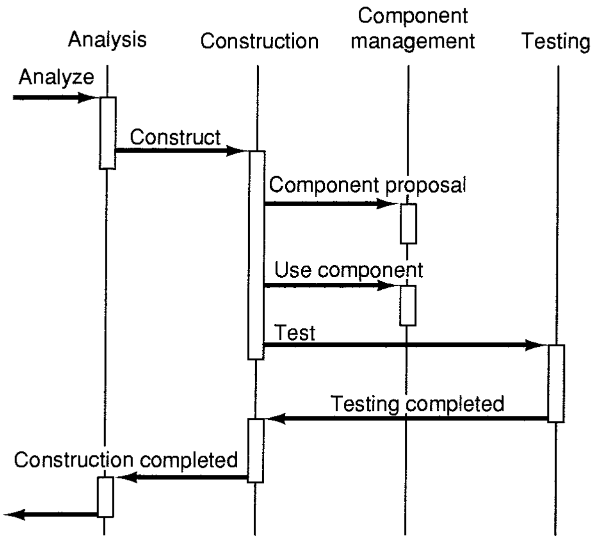


Figure A.9 A global interaction diagram for the handling case *New development.*

are now to define how the work is to be carried out in the different activity blocks and with what resources the work will be performed. Resources are personnel, programming tools and CASE tools. Depending on the resources available, the work in the different activity blocks can be done in different ways. In some activity blocks, a great deal of support will be given by CASE tools, which will simplify the manual work. In other activity blocks most of the work may have to be manual. Owing to the differing amount of resources, the work descriptions of the different activity blocks will therefore be given a different form

By viewing Objectory from an object-oriented perspective, it becomes easy to specialize Objectory for different implementation environments. When adapting Objectory to a specific implementation environment, it is primarily the activity blocks responsible for implementation that will have to be specialized.

A.3 From idea to reality

The problems that system development was faced with when using the prevailing techniques were known even in the 1970s. It was also perfectly clear that the complex system that were desirable would

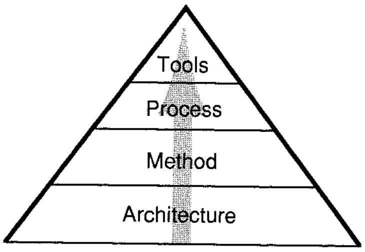


Figure A.10 Objectory consists of four levels, which were developed one on top of the other

place greater demands on system development methods It is from these demands that Objectory has gradually come into being and been developed.

The idea behind Objectory is the creation of a unified process that will support the activity development of software systems throughout the life cycle of the system. Ideally, the process will comprise all the phases of a system, from enterprise modeling, analysis, design, implementation, testing, configuration and installation to maintenance and future changes Most engineers of today lack this global view of system development; they lack this process thinking. At present it is not possible to formulate such a complete technique in one go. As with any complicated system, Objectory has thus evolved into its present version.

In developing the Objectory technique we started with the architecture and continued with the method, which was a prerequisite for the process and tools, see Figure A.10.

The technical foundation (the architecture) that Objectory rests on today were formulated as early as 1985. They are published in Jacobson (1985, 1986). Even if this foundation has been developed over time, the basic ideas remain the same. Thus use cases are still used to describe the essential dynamics of the system. The use case concept has been further developed, however, and its relation to other types of objects has been clarified.

The first version of a coherent method description existed as early as February 1986 and it was presented at OOPSLA'87, see Jacobson (1987). Like the underlying architecture, the method level has been further developed since then, and been supplemented by, among other things, criteria for how objects should be formed.

The process description integrates the method description into a framework stating how the work should be carried out as interacting processes within each phase of the development. The first edition of the process description was ready in November 1988 and covered

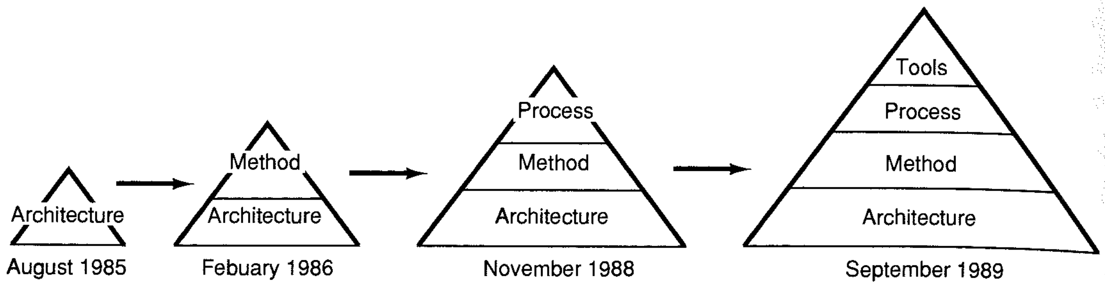


Figure A.11 Objectory has been developed gradually through the addition of new parts and extension of existing parts

analysis, construction and testing. At the same time a new version of the method was ready. Since then, it has been developed considerably in the light of experience gained, and the present third edition comprises some 1000 pages. Since the publication of the first edition, the process descriptions have been used in about 15 projects. The fourth edition of the process description is planned for late 1992.

Even though it may be possible to carry out projects without tool support, the participants in all the projects so far have expressed a strong need for such aid. Above all, they have wanted facilities to keep the information in all documents consistent. Therefore a first version of OrySE (Objectory Support Environment) was ready for customer installation in September 1989. This first version, with support for Objectory analysis, was extended in 1991 to support also Objectory design. OrySE is being developed further (using Objectory and OrySE) and the third version will be delivered late in 1992. The development is illustrated in Figure A.11.

Appendix B: Architecture Summary

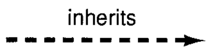
This appendix summarizes the modeling concepts used in this book.

B.1 Associations

The following associations may be used in all models:

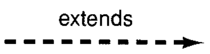
Class associations are drawn with a dashed arrow.

Inheritance



The descendant has all the properties of the ancestor. The arrow is drawn from the descendant and points to the ancestor since the descendant knows of its ancestor, but the ancestor does not know of its descendants.

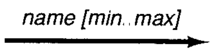
Extends



One object extends another object. The arrow points to the object that will be extended.

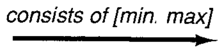
Instance associations are drawn with a full arrow.

Acquaintance association



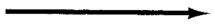
An object holds a reference to another object. The association has a name and a cardinality which tells us how many objects may be associated. We name the association in terms of the responsibility that the object has or which roles it plays to the object that associates it.

Consists-of



An object aggregates a set of other objects. This is a special kind of acquaintance association.

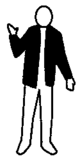
Communication association



An object may send stimuli to another object. The arrow points to the receiver of the stimuli. The association may be bidirectional.

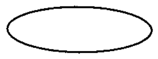
B.2 The use case model

Actor



Models one, or several roles that an interactor to the system can play. The interactor may be either human or machine.

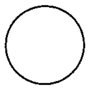
Use Case



A special sequence of transactions in a dialogue between a user and the system. Each use case is thus a specific way of using the system. A use case may have one basic course and several alternative courses.

B.3 Domain object model

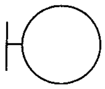
Object



Models a phenomenon in real life that the system needs to be aware of. Aims at understanding the concepts of the problem domain.

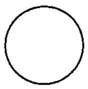
B.4 Analysis model

Interface object



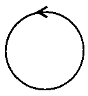
Models objects that are directly dependent on the interface of the system. Should separate the functionality of the application from its presentation to actors.

Entity object



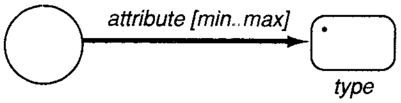
Models information and the behavior associated with this information of the system. Usually a long-lived (persistent) object that survives a use case.

Control object



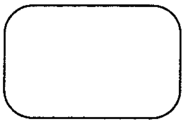
Models objects of the system that are specific to one or a few use cases. Usually ties together the objects that participate in a use case.

Attribute



Models an information element of an object. An attribute has a name which is attribute, a cardinality and a type which is type. Attributes are used for encapsulated information elements of an object.

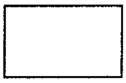
Subsystem



Groups objects and subsystems into managing units. The lowest level is called *service package* and is an atomic managing unit. Subsystems may exist in the analysis model as well as in the design model.

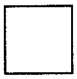
B.5 Design model

Block



Models a part of the implemented system. Normally one analysis object is implemented as one block. A block could be public or private to a subsystem.

Object module



A module in the programming language. In an object-oriented programming language it corresponds to a class. Could be public or private to a block.

References

Abbot R. (1983). Program design by informal English descriptions. *Communications of the ACM* **26**(11).

Abelson H., Sussman G.J. and Sussman J. (1985). *Structure and Interpretation of Computer Programs* MIT Press.

Agresti W. W , ed. (1986). *New Paradigms For Software Development* IEEE Computer Society Press.

Aho A.V., Hopcroft J.E. and Ullman J.D. (1983). *Data Structures and Algorithms* Reading, MA: Addison-Wesley.

Alford M. (1985). SREM at the age of eight: the distributed computing design system *IEEE Computer*, **18**(4), pp. 36–46

Atkinson M , Bancilhon F., DeWitt D., Dittrich K., Maier D. and Zdonik S. (1989). The object-oriented database system manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, Kyoto, Japan, December.

Backus J (1977). Can programming be liberated from the von Neumann Style? A Functional style and its algebra of programs. 1977 Turing Award Lecture Reprinted in *ACM Turing Award Lectures* Reading, MA: Addison-Wesley, pp. 63–130.

Barker R. (1989). *CASE*Method*™ – Entity Relationship Modelling. Wokingham: Addison Wesley.

Barnes J.G.P (1982, 1984) *Programming in Ada*. International Computer Science Series

Barry B. M. (1989) Prototyping a real-time embedded system in Smalltalk. In *Proceedings OOPSLA '89*, New Orleans, USA, October, pp. 255–65.

Beck K. and Cunningham W. (1989). A laboratory for teaching object-oriented thinking *Proceedings of OOPSLA '89*, pp 1–6

Ben-Ari M. (1982) *Principles of Concurrent Programming*. Englewood Cliffs, NJ: Prentice-Hall.

Bergner S-E (1990). A CASE Tool for Object-Oriented System Development in an Industrial Environment. *Proceedings of TOOLS 2*. Paris

Berzins V. and Luqi (1990). An introduction to the Specification Language Spec. *IEEE Software* March, pp. 74–84.

Bird R.J. and Wadler P. (1988). *Introduction to Functional Programming*. Hertfordshire: Prentice Hall.

Birrell A D. and Nelson B J. (1984) Implementing remote procedure calls. *ACM Transaction on Computer Systems* **2**(1), pp. 39–59

Birtwistle G M , Dahl O-J , Myrhaug B and Nygaard K (1979). *Simula Begin*, 2nd ed . Lund (Sweden): Studentlitteratur; Goch: Bratt-Institut für Neues Lernen; Bromley: Chartwell-Bratt

Blaha M R., Premerlani W.J. and Rumbaugh J E (1988) Relational database design using an object-oriented methodology. *Communications of the ACM*, **31**(4), pp. 414–27

Bloom T and Zdonik S B. (1987) Issues in the design of object-oriented database programming languages. *Proceedings of OOPSLA '87*, Orlando, Florida, October, pp 441–51

Boehm B.W (1981). *Software Engineering Economics* Englewood Cliffs, NJ: Prentice-Hall

Boehm B W. (1986) A spiral model of software development and enhancement. *Software Engineering Notes*, **11**(4)

Booch G. (1983) *Software Engineering with Ada* Menlo Park: Benjamin/Cummings.

Booch G. (1986) Object-oriented development. *IEEE Transactions on Software Engineering* **12**(2), pp 211–21

Booch G (1987a) *Software Components with Ada* Benjamin/Cummings.

Booch G (1987b) *Software Engineering with Ada*, 2nd ed Menlo Park: Benjamin/Cummings

Booch G. (1991) *Object-Oriented Design with Applications*. Redwood City: Benjamin/Cummings.

Booch G. and Vilot M. (1990) The design of the C++ Booch components. *Proceedings of OOPSLA '90* Ottawa, Canada, pp 1–11

Brodie M L., Mylopoulos J and Schmidt J. W eds. (1984) *On Conceptual Modelling* New York: Springer

Brooks F (1987) No silver bullet: essence and accidents of software engineering *IEEE Computer* **20**(4), pp. 10–19.

Browne J C, Lee T and Werth J (1990) Experimental evaluation of a reusability-oriented parallel programming environment *IEEE Transactions on Software Engineering* **16**(2), pp 111–20.

Bubenko Jr J and Lindencrona, E (1984) *Conceptual Modeling*–Information Analysis Lund (Sweden): Studentlitteratur

Buhr R.J A. (1984) *System Design with Ada* Englewood Cliffs, NJ: Prentice-Hall.

Buhr R J.A. (1991). *Practical Visual Techniques in System Design: With Applications to Ada* Englewood Cliffs, NJ: Prentice-Hall

Caldiera G and Basili V R (1991) Identifying and qualifying reusable software components. *IEEE Computer* February, pp. 61–70

CCITT (1984) *Fascicle vi 12, CHILL* Recommendation Z.200, Geneva

CCITT (1988) *Specification and Description Language (SDL)*. Recommendation Z 100. Geneva

Coad P and Yourdon E (1991a). *Object-Oriented Analysis*, 2nd ed Englewood Cliffs, NJ: Prentice-Hall.

Coad P and Yourdon E. (1991b). *Object-Oriented Design* Englewood Cliffs, NJ: Prentice-Hall

Constantine L.L. (1990) Object-oriented and function-oriented software structure A revised form of 'Objects, Functions, and Extensibility' in *Computer Language*, **7**, January, pp 34–56

Cox B.J. (1986). *Object Oriented Programming – An Evolutionary Appoach.* Reading, MA: Addison-Wesley

Cox B (1990). There *Is* a Silver Bullet *BYTE Magazine.* October, 1990, pp. 209–18

Dahl O -J and Nygaard K. (1966) SIMULA – an Algol-based simulation language. *Communications of the ACM*, **9**(9)

Date C J. (1986). *An Introduction to Database Systems, Volume 1*, 4th ed Reading, MA: Addison-Wesley.

Dietrich W C , Nackman L R. and Gracer F (1989) Saving a legacy with objects. In *Proceedings OOPSLA '89*, New Orleans, USA, October, pp 77–83.

Doyle J (1979). A Truth Maintenance System. *Artificial Intelligence* **12**(3), pp 231–79

Ellis M.A. and B. Stroustrup (1990) *The Annotated C++ Reference Manual* Reading, MA: Addison-Wesley

Embley D.W. and Woodfield S.N (1988) Assessing the quality of abstract data types written in Ada In *Proceedings of the Tenth International Conference on Software Engineering* IEEE Computer Society Press, pp 144–53

Embley D W , Kurtz B D and Woodfield S N. (1992) *Object-oriented Systems Analysis – A Model-driven Approach* Yourdon Press.

Eriksson G and Holm P (1984). *Programmering i Simula (Programming in Simula)* Lund (Sweden): KF-Sigma

Freeman P ed (1987) *Tutorial: Software Reusability* IEEE Computer Society Press.

Gibson E. (1990). Objects – born and bred *BYTE Magazine* October, pp. 245–54.

Glahn J.E and Meland C (1985). Practical tools which give the decision maker clarity of the future. *Proceedings of 8th INTERNET Congress* Rotterdam.

Goldberg A and Robson D (1983). *Smalltalk-80: The Language and its Implementation* Reading, MA: Addison-Wesley

Gomaa H (1984). A software design method for real-time systems. *Communications of the ACM.* **27**(9), pp 938–49

Gomaa H (1989). Structuring criteria for real-time system design. *Proceedings from International Conference on Software Engineering.* Pittsburgh, 15-18 May, pp 290-301

Gorlen K.E , Orlow S M and Plexico P S (1990) *Data Abstraction and Object-Oriented Programming in C++.* Chichester: Wiley

Grady R.B. and Caswell D.L. (1987) *Software Metrics: Establishing a Company-Wide Program* Englewood Cliffs, NJ: Prentice-Hall

Hall A (1990) Seven myths of formal methods *IEEE Software*, September, pp. 11–19.

Hartson H R. and Hix D. (1989). Human–computer interface development: concepts and systems for its management *ACM Computing Surveys.* **21**

(1), pp 5–92

Hix D (1990) Generations of user-interface management systems *IEEE Software* September, pp. 77–87

Hoare C A.R. (1985) *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice-Hall.

HOOD (1989a). *HOOD User Manual* Issue 3.0. WME/89-353/JB. HOOD Working Group. European Space Agency, December

HOOD (1989b) *HOOD Reference Manual* Issue 3 0. WME/89-173/JB. HOOD Working Group. European Space Agency, September.

Hull R and King R. (1987). Semantic database modeling: survey, applications, and research issues *ACM Computing Surveys*, **19**(3), pp 201–60.

Humphrey W (1989) *Managing the Software Process*. Reading, MA: Addison-Wesley

IEEE (1983). IEEE Std 729-1983 *Standard Glossary of Software Engineering Terminology*

Jackson M (1983). *System Development*. Englewood Cliffs, NJ: Prentice Hall.

Jacky J.P and Kalet I J (1987). An object-oriented programming discipline for standard Pascal. *Communication of the ACM* **30**(9), pp 772–6

Jacobson I. (1985). Concepts for modeling large realtime systems Ph.D thesis. Royal Institute of Technology: Stockholm

Jacobson I. (1986). Language support for changeable large real time systems. In *Proceedings of OOPSLA '86*, Portland, Oregon, USA, September, pp. 377–84.

Jacobson I. (1987) Object-oriented development in an industrial environment. *Proceedings of OOPSLA '87. SIGPLAN Notices*, **22**(12), pp. 183–91

Jacobson I. (1991). Industrial development of software with an object-oriented technique *Journal of Object-Oriented Programming* March/April, pp. 30–41

Jacobson I. and Lindström F. (1991) Re-engineering of Old Systems to an Object-Oriented architecture. *Proceedings of OOPSLA '91* Phoenix AZ, October 1991, pp 340–50.

Johnson R E. and Foote B (1988) Designing reusable classes. *Journal of Object-Oriented Programming*, June/July, 22–35

JOOP (1991) Special issue on databases *Journal of Object-Oriented Programming*, July/August.

Knuth D.E. (1973, 1981) *The Art of Computer Programming*, Volumes 1-3. Reading, MA: Addison-Wesley.

LaLonde W R and Pugh J.R (1990). *Inside Smalltalk*, vol I Englewood Cliffs, NJ: Prentice Hall

LaLonde W R. and Pugh J R (1991a) *Inside Smalltalk*, vol II. Englewood Cliffs, NJ: Prentice Hall.

LaLonde W.R and Pugh J R. (1991b). Subclassing≠Subtyping≠IsA *Journal of Object-Oriented Programming* January, pp 57–62.

LaLonde W R, Thomas D A and Pugh J R (1986). An Exemplar Based Smalltalk *Proceedings of OOPSLA '86*, Portland, OR September–October, pp. 322–30

Lawson H (1990) Philosophies for engineering computer based systems. *IEEE Computer*, **23**(12), pp. 52–63.

Lawson H. (1991). *Parallel Processing in Industrial Real-Time Applications* Englewood Cliffs, NJ: Prentice-Hall

Lehman M M. and Belady L. (1985). *Program Evolution Process of Software Change* London: Academic

Levendel Y. (1990) Reliability analysis of large software systems: defect data modeling. *IEEE Transactions on Software Engineering*, **16**(2), pp 141–52.

Lieberherr K J and Holland I M. (1989) Assuring good style for object-oriented programs *IEEE Software* September, pp 38–48.

Lieberman H (1986). Using prototypical objects to implement shared behavior in object oriented systems. *Proceedings of OOPSLA '86*. Portland, OR, September–October, 214–23

Linowes J.S (1988). It's an attitude *Byte Magazine* August, pp 219–24.

Lippman S (1991) *C++ Primer*, 2nd ed Reading, MA: Addison-Wesley

Loomis M E.S (1990). Several columns discussing issues on ODBMS *Journal of Object-Oriented Programming*, May/June and later

Loomis M E.S., Shah A.V and Rumbaugh J. E. (1987). An object modeling technique for conceptual design *Proceedings of ECOOP '87*, pp. 325–35.

Matsumoto Y (1987) A software factory: an overall approach to software production In Freeman (1987)

McCabe T.J. (1976). A complexity measure *IEEE Transactions on Software Engineering* **2**(4), pp 308–20

McIlroy M D (1976) Mass-Produced Software Components. In *Software Engineering Concepts and Techniques (1968 NATO Conference on Software Engineering)* J.M. Buxton, P. Naur and B. Randell, Eds. Van Nostrand Reinhold, pp 88–98.

McMenamin S M and Palmer J.F. (1984) *Essential Systems Analysis* Englewood Cliffs, NJ: Yourdon Press.

Metzger P.W. (1981). *Managing a Programming Project*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall

Meyer B (1988). *Object-Oriented Software Construction* Englewood Cliffs, NJ: Prentice Hall

Meyer B. (1990) Lessons from the design of the Eiffel libraries. *Communications of the ACM*. **33**(9), pp. 68–88

Mills H D , Dyer M and Linger R.C (1987). Cleanroom software engineering. *IEEE Software* September, pp. 19–24.

Myers, G. J. (1979). *The Art of Software Testing* New York: Wiley

Myers, G J. (1987) *Software Reliability: Principles and Practices* New York: Wiley

Nielsen K. and Shumate K (1988) *Designing Large Real-Time Systems with Ada* New York: McGraw-Hill and Intertext

OOPSLA (1990). Structured Analysis and Object-Oriented Analysis. *Proceedings of OOPSLA '90*, pp 135–9.

Page-Jones M. and Weiss S. (1989). Synthesis/analysis object-oriented method. *DCI Object-Oriented Systems Symposium*, June

Parnas D. (1972) On the criteria to be used in decomposing systems into modules *Communications of the ACM*, **15**(2), 1053–58.

Parnas D.L , Clements P C. and Weiss D.M (1983). Enhancing reusability with information hiding. *ITT Proceedings of the Workshop on Reusability* *in Programming, 1983*, pp 240–47 Also in Freeman (1986)

Partsch H. and Steinbrüggen R. (1983). Program transformation systems. *ACM Computing Surveys*, **15**(3), pp. 109–226.

Peckham J and Maryanski F (1988) Semantic Data Models *ACM Computing Surveys*, **20**(3), pp. 153–89

Perry D E. and Kaiser G E. (1990) Adequate testing and object-oriented programming *Journal of Object-Oriented Programming*, January/ February, pp. 13–19

Peterson J L. and Silberschatz A (1985). *Operating Systems Concepts* Reading, MA: Addison-Wesley

Premerlani W.J , Blaha M.R , Rumbaugh J.E. and Varwig T A (1990). An object-oriented relational database. *Communications of the ACM*, **33**(11), pp. 99–109.

Prieto-Diaz R and Freeman P. (1987). Classifying software for reusability *IEEE Software* January, pp 6–16. Also in Freeman (1987), pp 106–16

Purchase J.A. and Winder R L (1991). Debugging tools for object-oriented programming *Journal of Object-Oriented Programming*, June, pp 10–27.

Ramamritham K , Stankovic J A. and Zhao W. (1989). Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Transactions on Computers*, **38**(9), pp. 938–62

Rochat R. (1986) *In Search of Good Smalltalk Programming Style* Technical Report CR-86-19 Tektronix.

Ross D T. (1985) Applications and Extensions of SADT *IEEE Computer*. April

Rumbaugh J., Blaha M , Premerlani W , Eddy F., Lorensen W (1991) *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall

Scharenberg M E. and Dunsmore H.E (1991) Evolution of classes and objects during object-oriented design and programming. *Journal of Object-Oriented Programming* January, pp. 30–4.

Sha L. and Goodenough J.B (1990). Real-time scheduling theory and Ada. *IEEE Computer* April, pp 53–62.

Shaw A.C. (1989). Reasoning about time in higher-level language software *IEEE Transactions on Software Engineering*, **15**(7), pp 875–89.

Shlaer S and Mellor S J (1988) *Object-oriented Systems Analysis*. Englewood Cliffs, NJ: Prentice-Hall.

Snyder A. (1986) Encapsulation and inheritance in object-oriented programming languages *Proceedings of OOPSLA '86* Portland, OR, September–October, 38–45

Sommerville, I. (1989) *Software Engineering*, 3rd Edn Wokingham: Addison-Wesley

Spector A. and Gifford D. (1986). A Computer Science Perspective of Bridge Design. *Communications of the ACM*, **29**, pp. 267–83

Stankovic J.A (1988). Misconceptions about real-time computing. *IEEE Computer*, October, pp. 10–18.

Stankovic J.A and Ramamrithan K (1988). Tutorial: hard real-time systems. Washington: IEEE Computer Society Press.

Stefik M.J., Bobrow D G and Kahn K.H. 1986. Integrating access-oriented programming into a multiparadigm environment *IEEE Software*.

January, pp. 170–8

Stone C.M and Hentchel D (1990). Database wars revisited. *BYTE Magazine*, October, pp 233–42

Sudkamp T.A. (1988) *Languages and Machines: An Introduction to the Theory of Computer Science* Reading, MA: Addison-Wesley

Taenzer D, Ganti M and Podar S (1989). Object-oriented software reuse: the yoyo problem. *Journal of Object-Oriented Programming* September/October, 30–5

Tracz W (1988) Software reuse maxims. *ACM Software Engineering Notes* **13**(4), pp 28–31

Tsichritzis D C and Lochovsky F. H (1982). *Data Models* Englewood Cliffs, NJ: Prentice Hall.

von Neumann J. (1945) First Draft of a Report on the EDVAC Moore School of Electrical Engineering, University of Pennsylvania

Ward P T. (1989). How to integrate object orientation with structured analysis and design *IEEE Software* March, pp. 74–82.

Ward P T and Mellor S.J (1985) *Structured Development for Real-Time Systems* New York: Yourdon Press

Wasserman A.I, Pircher P A. and Müller R J (1989) Concepts of object-oriented structured design. *Proceedings of TOOLS '89* pp 269–80.

Wasserman A I., Pircher P.A and Müller R J (1990) The object-oriented structured design notation for software design representation *IEEE Computer* March, pp 50–62

Wegner P (1987) Dimensions of Object-Based Language Design. *Proceedings of OOPSLA '87*. Orland Special Issue of *SIGPLAN Notices*, **22**(12), pp. 168–82.

Wegner, P (1989). Learning the Language. *BYTE Magazine*, March, pp. 245–53

Wegner P. and Zdonik S. (1988). Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. *Proceedings of ECOOP '88*. Springer, pp 55–77.

Weinberg G M. (1971) *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold

Weinberg G M. and Freedman D. P (1982) *Handbook of Walkthroughs, Inspections, and Technical Reviews* Boston: Little Brown Computer Systems

Weizenbaum J (1968) *The Fonary Problem Explained*. Unpublished memorandum, MIT Cambridge 1968 as quoted in Allen J (1978) *Anatomy of Lisp* New York: McGraw-Hill

Wing J.M. (1990) A specifier's introduction to formal methods. *IEEE Computer*, September, pp 8–24

Wirfs-Brock R J and Johnson R.E (1990). Surveying current reserach in object-oriented design. *Communications of the ACM* **33**(9), pp. 104–24

Wirfs-Brock R, Wilkerson B and Wiener L. (1990) *Designing Object-Oriented Software* Englewood Cliffs, NJ: Prentice Hall.

Yourdon E. (1989a) *Modern Structured Analysis*, Yourdon Press/Prentice Hall

Yourdon E (1989b). *Structured Walkthroughs*, 4th ed Englewood Cliffs: Prentice-Hall/Yourdon Press.

Yourdon E. (1990). Auld Lang Syne. *BYTE Magazine*, October, pp 257–63.

Yourdon E. and Constantine L.L. (1979). *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* Englewood Cliffs, NJ: Prentice Hall.

Zave P. (1984). The operational versus the conventional approach to software development *Communications of the ACM*, **27**(2), pp. 104–18.

Index