软件工程系列教材

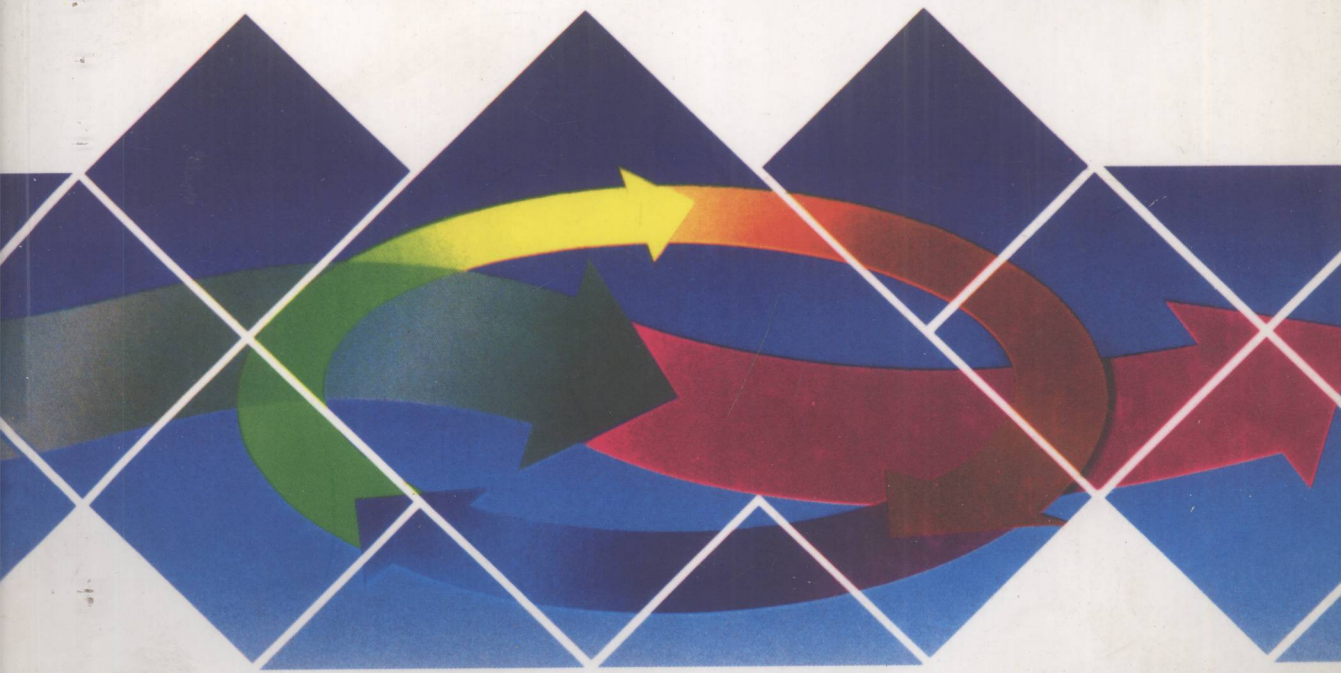

COMPUTER LANGUAGE Productivity Award Winner

# 面向对象软件工程

(修订版) (英文版)

# Object-Oriented Software Engineering

## A Use Case Driven Approach



Ivar Jacobson
Magnus Christerson Patrik Jonsson Gunnar Övergaard 著

人民邮电出版社
POSTS & TELECOM PRESS

软件工程系列教材

# 面向对象软件工程
# （修 订 版）

## Object-Oriented Software Engineering
## A Use Case Driven Approach
## （英 文 版）

Ivar Jacobson

Magnus Christerson　Patrik Jonsson　著
Gunnar Övergaard



人 民 邮 电 出 版 社

**图书在版编目（CIP）数据**

面向对象软件工程 /（ ）雅各布森（Jacobson, I.）编著. —修订本.
—北京：人民邮电出版社，2003.9
（软件工程系列教材）
ISBN 7-115-11163-4

I. 面… II. 雅… III. 面向对象语言—程序设计—教材—英文 IV. TP312

中国版本图书馆 CIP 数据核字（2003）第 071438 号

**版 权 声 明**



软件工程系列教材

**面向对象软件工程（修订版）（英文版）**

◆ 著 Ivar Jacobson Magnus Christerson
Patrik Jonsson Gunnar Overgaard
责任编辑 俞 彬

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress com cn
网址 http //www ptpress com cn
读者热线 010-67132705
北京汉魂图文设计有限公司制作
北京隆昌伟业印刷有限公司印刷
新华书店总店北京发行所经销

◆ 开本 787×1092 1/16
印张 35 25
字数. 798 千字 2003 年 9 月第 1 版
印数 1-3 500 册 2003 年 9 月北京第 1 次印刷

著作权合同登记 图字：01－2003－0391 号

ISBN 7-115-11163-4/TP・3376

定价：49.00 元

**本书如有印装质量问题，请与本社联系 电话：（010）67129223**

# 内容提要

软件开发方法学家 Ivar Jacobson 在本书中提出的 OOSE（面向对象软件工程）方法是面向对象建模语言中最著名的方法之一，本书提出的 Use Case 分析方法在 OO 技术领域具有举足轻重的地位。

本书全面介绍了软件工程学科的概念和方法。本书分为三部分，第一部分——简介，内容涵盖了软件工程学科的背景知识，共 5 章，包括系统开发简介，工业过程需求概述，系统生命周期，此外还介绍了面向对象概念以及如何将其应用到系统开发中。第二部分——概念，包括 7 章内容，第 6 章介绍了 OOSE 基础概念，并解释了为什么选择这些概念。接下的章节论述了分析和构建方法。接下来的两章论述了这些方法是如何被采纳到实时系统和数据库管理系统中的。第 11 章组件，介绍了什么是组件，如何将组件应用到开发过程。第 12 章介绍了测试。第三部分——应用，介绍了 OOSE 应用以及作为一种新的开发过程是如何被组织和管理的。本部分的最后论述了其他面向对象方法。本书附录介绍了对象技术的发展史。

本书荣获 1992 年度软件开发杂志效率大奖（Productivity Award），同时也是软件工程领域中享有盛誉的经典著作。本书是计算机及相关专业本科生和研究生的优秀教材，也可作为软件工程领域技术人员的参考资料。

# 序

这是一部国际公认的名著。

Ivar Jacobson 是一位来自工业界的学者，一位软件开发方法论的大师。他从 1967 年便开始致力于软件的工业化开发技术与理论的研究。他把在爱立信等公司进行大型系统开发的丰富经验加以总结和升华，取得了一系列原创性的研究成果，如：序列图（Sequence diagram）、协同图（Collaboration diagram）、电信领域的规格说明语言 SDL、基于组件的软件开发过程、用例驱动的开发方法等，而 Objectory 则是集中体现了这些研究成果的一个产品，Jacobson 将这个产品中所体现的技术称为 OOSE（面向对象软件工程）。从 20 世纪 90 年代中期开始，他与 Grady Booch、James Rumbaugh 一起，设计了著名的统一建模语言（UML）和统一开发过程（UP），OOSE 则是 UML 和 UP 的重要组成部分。

本书是对 OOSE 的全面描述。应当指出，由于历史条件的限制，OOSE 既是一种开发过程，也是一种开发方法。从 UML 和 UP 开始，人们趋于将开发方法、开发过程、建模语言进行明确的分离。

OOSE 的出发点，是继承传统工业中已经行之有效的工业化手段，在软件系统的开发中采用构件化技术；OOSE 的目标，则是为实现高度自动化的、采用可重用组件来制造软件的工厂，来提供技术与管理方面的支持。

组件化已经成为当前软件开发的主流技术之一，但在 20 多年前就认识到这一点、并且不遗余力地坚持到现在，这便是 Jacobson 的过人之处。坦率地讲，有些人将软件模块加以并不高明的包装，便声称提出了一种新的组件化技术，而 Jacobson 则是来自工业界的一位务实的工程师。他在本书中关于组件的朴实无华的论述，十分接近于从 1993 年开始备受关注的模式（Pattern）。另外，从工业化和实用性的角度来看，构件的正确性与可靠性对于有效的重用是至关重要的，因此本书中专门有一章来论述构件的测试。实际上，本书全面论述了基于构件的软件工程技术，而不仅仅是组件开发技术，在今天仍然具有普遍的指导意义。

OOSE 的一个重要特点是采用反射（Reflection）体系结构的理念来建立软件开发过程，这使得它成为一种可扩充、可定制的开发过程，可以针对开发组织和应用领域的特征进行相应的特殊化处理，以构成适用的开发过程。从这个意义上来看，OOSE 不仅能够用于软件系统的开发，而且能够用于特定软件过程的建立。现在，反射已经是一种被普遍接受的理念，在主流的语言（如 UML 和 Java）和软件开发过程（如 OOSE）中都得到了应用。

OOSE 的一个重要创造是用例（Use case）与用例驱动。用例是软件需求的一种表示手段，促使开发人员将注意力集中于系统的业务功能需求。然而，用例又不仅仅是需求的表示手段，

它贯穿了系统生存周期中的所有阶段，这就是用例驱动，从而为实现需求的可追踪性提供了一种有效的支持手段。系统需求将必然发生变更，这是一条客观规律。由于需求的变更可以被直接映射到对应一组用例的变更上，因此OOSE为快速原型开发以及其他形式的递增开发，提供了一个坚实的方法论基础。从我个人的研究、开发与教学经验来看，将用例作为需求表示手段，在理解和掌握上一般是不困难的，而要将用例驱动技术自如地应用于递增开发与需求变更处理，则要困难得多。实际上，本书的大部分章节都是从不同的侧面在描述用例驱动技术，读者应当注意这一点。

Jacobson有着开发大型软件系统、特别是实时系统的丰富经验，本书中分两章给出的两个范例都是作者亲身经历的项目，值得我们仔细考察和分析。

我谨向致力于面向对象和（或）软件工程领域研究与应用工作的读者们推荐本书，从它出版后的11年里所产生的巨大影响来看，的确是一部值得反复阅读与学习的名著。

陈　平
西安电子科技大学
2003年仲夏于西安

# 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 practising 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 keeps 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 practised 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 practise 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 by 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 of 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 industrialization 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. Introducing the process into an organization needs careful planning and dedication. It also requires that the process be adapted to the unique needs of the organization. Such process adaptations must of course be carefully specified, which can be done in a development case description, as will later be explained.

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 in a way which will make it accessible both to practitioners in the field and to 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 covers the background, and contains 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 of 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 contains 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 discuss the method of analysis and construction. The next two chapters discuss 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 comprises:

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

**Appendix.** Finally we have an appendix which comments on our development of Objectory.

So, how should you read this book? Of course, to get a complete overview, the whole book should be read, including the appendix. 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.

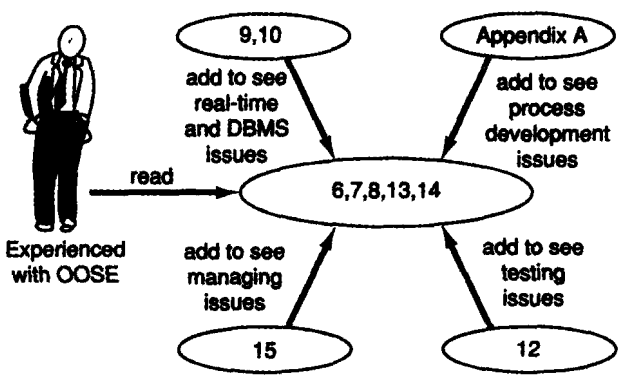



**Figure P.1**

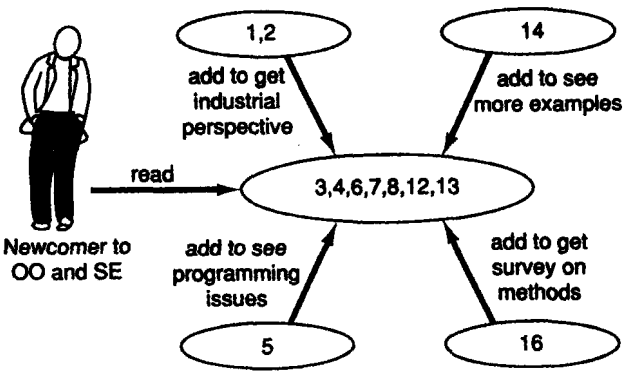



Figure P.2

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.

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

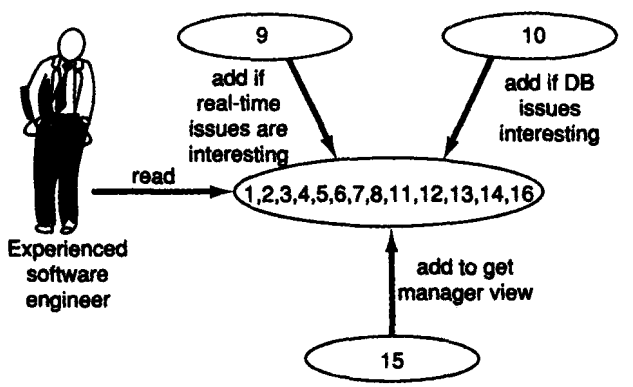



Figure P.3

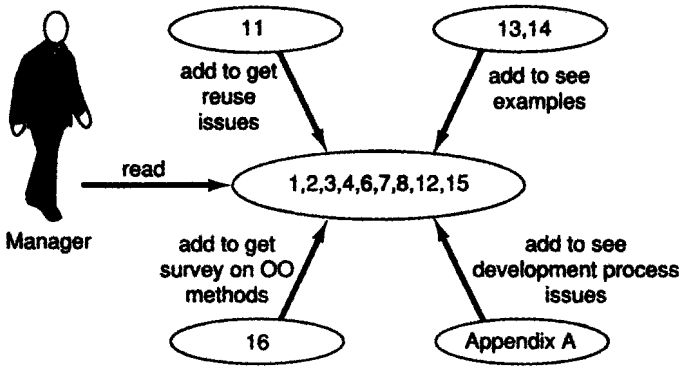



Figure P.4

## 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 alone. 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 signals 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 receipt of an input signal, performing internal actions, 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 of 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 the skeleton of a new design method, the use of which was first demonstrated in the development of the AKE system put into service in Rotterdam in 1971, and more completely demonstrated in the AKE system put into service in Fredhall, 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 done 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-scale' and 'small-scale'.

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 object and two types of communication mechanism, send/wait and send/no-wait semantics. The language supported concurrency with atomic transactions and a special semantic construction for the handling 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, taking a new approach, develop the method. The principles of Objectory were developed in 1985–7. I then further refined and simplified the ideas, generalized the technique used in the telecom applications, extended it with the inheritance concept and other important constructs like extensions, and coupled to it an analysis technique and object-oriented programming.