Object-oriented modelling of information systems : the INCA conceptual object model

Object-Oriented Modelling of Information Systems

The INCA Conceptual Object Model

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Bakker, Harm

Object-oriented modelling of information systems:
the INCA conceptual object model / Harm Bakker. - [S.l. :
s.n.]. - Ill., fig., tab.
Proefschrift Maastricht. - Met index, lit. opg. - Met
samenvatting in het Nederlands.
ISBN 90-9007906-8 geb.
NUGI 855
Trefw.: object-georiënteerd programmeren / informatiesystemen / computers.

Cover design: Mariëlle Bakker-Tromp

Voor Mariëlle, Maaike en Thomas

Object-Oriented Modelling of Information Systems

The INCA Conceptual Object Model

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Rijksuniversiteit Limburg te Maastricht,
op gezag van de Rector Magnificus, Prof. dr. H. Philipsen,
volgens het besluit van het College van Dekanen,
in het openbaar te verdedigen
op woensdag 18 januari 1995 om 16.00 uur

door

Harm Bakker

Promotor: Prof. dr. H. J. van den Herik
Copromotor: Dr. P. J. Braspenning

Leden van de beoordelingscommissie:
Prof. dr. ir. J. L. G. Dietz
Prof. dr. ing. D. K. Hammer (Technische Universiteit Eindhoven)
Prof. dr. ir. A. Hasman
Prof. dr. P. T. W. Hudson
Prof. dr. A. Ollongren (Rijksuniversiteit Leiden)

Preface

The research for a thesis is impossible without the help of many persons, as is the writing of it. I intend to let all my helpers have due recognition. The research part would have been impossible without the incessant support of Jaap van den Herik. We first met in Delft in 1987 during a programming contest, which became the starting point for a Master's thesis in Delft, and then developed into a doctoral thesis at the University of Limburg, Maastricht. I thank Jaap for his trust and patience. With Bob Herschberg prompting, he provided me with sufficient examples of fine English. I aimed at their level of proficiency, but am aware that I have fallen short, for which the responsibility must be mine.

Peter Braspenning has been the main force behind the INCA project. Six years ago we became acquainted in Maastricht. We combined our ideas on modelling, object orientation and information systems. Our numerous working sessions during the past years have been a great source of inspiration. Peter's enthusiasm and positive thinking helped to overcome my doubts and hesitations.

The final version of this thesis has benefitted greatly from the valuable suggestions from my professorial teachers of the thesis committee. It took many an extra evening to grasp their suggestions, let alone to incorporate them. My gratitude is not diminished by the fact that they caused me more work.

My former colleagues in the Department of Computer Science of the University of Limburg always have made me feel at home. I especially thank Hans Henseler and Eric Postma for their support, good humour and for the experiences they were ready to share. Cooperating with Luc van Leeuwen and Jos Uiterwijk in the INCA project has been a source of inspiration I wish to put on record.

Keeping up my good spirits was greatly aided through the interest shown by my parents, parents-in-law, brothers, sister and friends. I certainly wish to

express my gratefulness towards my colleagues at the Telematics Research Centre in Enschede. In the past year, their interest and ceaseless queries have accelerated the process of finishing this thesis. If proof were needed this shows the importance of being part of a social 'infrastructure'.

One person never stopped supporting me.
One person never stopped being patient.
One person never stopped caring.
One person never *objected*.
Mariëlle, thank you for keeping up with me.

In the last few months, Maaike's "Papa werken!" has gradually changed into a timid "Papa stoeien?", which, I'm sure, will change for the better into an imperative "Papa stoeien!". I hope Thomas will soon echo the inviting words of his elder sister.

Harm Bakker
Enschede, November 1994

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 INCA

This thesis is the result of a five-year period of research at the University of Limburg in Maastricht, the Netherlands. In September 1988 the INCA (INtelligent CASE) project was launched in the computer-science department, with the long-term objective of developing an INtelligent computer-aided systems engineering (CASE) environment for systems development.

Early 1989, the research field of computer science witnessed the rebirth of *object orientation*. In fact, commercial objectives led the way to make object orientation the new *hype-of-the-year*. Object orientation, it was proclaimed, held the promise of systems development in a controlled manner, enabling large-scale software reuse through the distinguished feature, *inheritance*. The label of object orientation has since become synonymous with good. However, the inheritance of object orientation as a hype-of-the-year has also caused confusion on what it really means: today one can read advertisements for object-oriented drawing tools, referring to the capability of treating manipulated items as objects.

In our opinion, object orientation, when treated adequately and seriously, really keeps a promise for future software development. The arguments are twofold: (1) object orientation allows a natural view of the world, and (2) object orientation offers good concepts for software engineering.

Object orientation as a natural view of the world The use of objects for representing a domain goes back to Minsky's seminal paper on frames

(Minsky, 1975; Minsky, 1981). The frames were used to represent particular objects in a domain, with slots describing the different properties of an object. Since the introduction of frames, the notion of object orientation has had several adaptations and extensions in knowledge representation and knowledge engineering.

Object orientation for software engineering Considering the development of software, object orientation has a great importance for software engineering. Object orientation combines important ideas from the software-engineering field, such as encapsulation (combining and shielding various items within one structure), data hiding (focus on the external workings of an entity, not its internal realization), modularization (dividing a program into distinct parts), and inheritance (defining new types in terms of existing ones).

The above examples aided in choosing the concepts of object orientation for the realization of an INtelligent CASE (INCA) environment. The two main goals of an INCA environment are (1) to furnish a representation system for capturing the real world in which a (future) information system is to be employed, and (2) to offer concepts for modelling and constructing information systems, in terms of the representation of (1). As we stated above, object orientation combines concepts for the representation system of an INCA environment with concepts for software engineering. This combination of concepts led us early 1989 to choose object orientation as a research issue for the development of our INCA environment.

In this thesis, we describe the concepts of a *conceptual object model*, capable of capturing the *semantics* of real-world objects and their relations, occurring in a complex domain such as CASE.

1.2 Structure of the thesis

After the Introduction in Chapter 1, Chapter 2 describes the evolution of hardware and the so-called software crisis. It forms the background of this thesis. We then formulate the problem statement describing the subjects to be addressed. Software engineering is introduced as a possible answer to the software crisis, and the concept of CASE is presented as a particular way of assisting software engineers in performing their tasks. As an intermediate station along the way to automatic programming, INtelligent CASE is proposed as the next goal to be accomplished.

In Chapter 3 we introduce object orientation from the programming-language perspective. Subsequently, we treat the different kinds of modelling perspectives which a modeller may have, ranging from empirical models to formal models. The chapter ends with a unifying view on object orientation and modelling by describing object-oriented conceptual modelling.

Chapter 4 presents the main ideas underlying the INCA Conceptual Object Model (INCA-COM). It focusses on the importance of *semantics* in the object model. The object-describing entities distinguished as important for describing a model in an object-oriented fashion are introduced. Subsequently, four different models for modelling according to INCA-COM are presented; they are the information model, the event model, the behaviour model and the communication model.

Chapter 5 deals with structuring principles which are part of INCA-COM. We treat sort hierarchies, specification, composite objects, grouping, and systems.

Since versions are also part of INCA-COM, Chapter 6 discusses the version-management mechanism for composite objects. We believe it to be indispensable in an object-oriented model for complex objects. We clarify the ideas of version management with examples.

In Chapters 7 and 8 we illustrate the modelling concepts of INCA-COM by (1) constructing and describing a model of data-flow diagrams, and (2) showing the application of INCA concepts to the domain of spreadsheets.

Chapter 9 describes INCATOOL, a tool supporting the modelling with INCA-COM. Based on an ontology for object-oriented modelling and a form of language reflection, called descriptor reflection, the architecture of INCATOOL is described.

Chapter 10 concludes the thesis with a review of the main goals set forth for the INCA project, and lists possible directions for future research.

Chapter 2

The software crisis

A general goal of information science is to study the concepts of information processing. To this end information scientists are doing research on information systems, i.e., systems which process and store information. The research has spawned many useful methods and techniques for the actual development of these systems. Yet, there have been numerous accidents in systems development, leading to the so-called *software crisis*: software projects finished too late and costed much more than predicted, software systems were often unreliable and performed rather poorly (Sommerville, 1992). Although methods and techniques have been improved considerably, the software crisis still persists. And with the steady increase in computer capacity, the software crisis is likely to stay around for several years.

In order to reduce the costs of systems development the *level* of specification of an information system plays an important role. In this chapter we address the importance of *raising the level* at which an information system is specified. This leads to our problem statement. Moreover, it provides some indications on how we deal with the problems stated in this thesis.

2.1 Information systems and the software crisis

All organizations employ information to conduct their business. Whether it is a university, a building company, or a bridge club, information plays an important role. Any ensemble of information and information processing agents, humans or machines, is called an *information system*. The *information paradigm* for systems splits any system into a real system and an information system (Brussaard and Tas, 1980). The *real system* is the system being controlled,

and the *information system* is the system controlling the real system. The information paradigm is shown in Figure 2.1. For a full explanation and other details of the information paradigm we refer to Brussaard and Tas (1980).

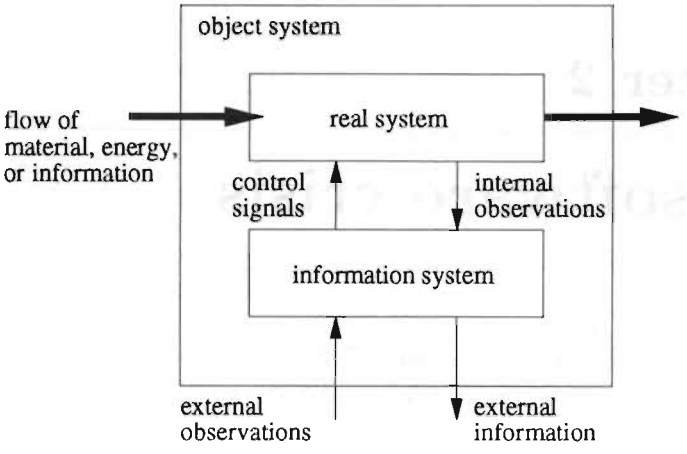


Figure 2.1: The information paradigm.

In order to control *things* in the real system, an information system must have some representation of these things. The things in the real system of which information is contained within an information system are generally termed the *Universe of Discourse* (UOD).

A *Universe of Discourse* (UOD) of an information system consists of the things in a particular domain, about which people in the information system perform a meaningful discourse. This term expresses that a UOD consists of the things upon which a meaningful discourse can be held.

As an example, we consider the domain of a university, giving courses and having students enrolled in courses. For the determination of time schedules, the UOD consists of the university, students, courses, course results, graduations, teachers, course rooms, etc. Not included in the UOD are the bricks of the university buildings, the eye colour of students, the bicycles of the students etc. From another point of view, the UOD might very well contain the latter things. This illustrates that the UOD is determined by the problem under consideration.

Others, such as Van Griethuysen (1982) include in the UOD all things which are of interest, *or might ever be*. However, such a definition is problematic for it forces us to look into the future foreseeing which things might become of interest. In our opinion, the UOD is just the restricted collection of things about which the information system contains information.

Currently, information systems are progressively being automated. The automation of an information system consists of assigning parts of the information system to information-processing machines. The information-processing machines can be dedicated to a particular task, such as vending or weaving, or they can be general-purpose machines (computers), which have been instructed to act as particular information-processing machines. The latter are interesting: they require a special set of instructions (a *program*) to make it behave as the intended information processor. In case the information system is large and a fair part of it is being automated, the resulting programs can also be considered as a system. A set of programs cooperating as an information processor, is called a *software system*.

The increasing demand for automated information systems requires software systems to be built. In combination with the increasing power of computers, the demand for large and complex software systems is increasing. The development of these software systems has been a complex task for developers for over 25 years now. Its complexity causes problems with the software-development process and introduces errors in the software. The problems, such as the specification and implementation of large software systems, have become known as the *software crisis*.

In 1968 the NATO organized a workshop on the software crisis (Naur and Randell, 1968). The increasing computing power of computer hardware had revealed the phenomenon of a crisis in software development. At that time the third-generation computers were introduced. Table 2.1 gives an overview of the evolution of the distinct hardware generations.

We do not treat each hardware generation, but instead focus on the difference between the second and third generation. The transition from the second generation to the third was not a mere *evolution*, but rather a *revolution*. "New hardware, especially the microminiature integrated circuit, enabled internal memory to expand. It also made possible the construction of direct-access (disk) devices, providing mass external storage. The third generation changed the way the computer was perceived. It was no longer an expensive, esoteric device limited to serial processing, but it became a centralized *resource* accessible via terminals throughout organizations" (Evans, 1989, p. 11). Compared to second-generation computers these new computers were orders of magnitude more powerful (see also Table 2.1): (1) integrated circuits on computer chips enabled internal memory to expand, (2) the development of disk devices improved the access time of external storage, (3) the total capacity of both

Generation	*Period*	*Characteristics*
1	1950-58	vacuum tubes, machine and assembly language
2	1958-67	transistor, high-level languages, batch processing
3	1968-78	integrated circuit, disks, extended memory, interactivity
4	1978-88	very large-scale integration, microcomputers, decentralization
5	1988-now	workstations, networking, decentralized integration

Table 2.1: The evolution of hardware generations.

internal and external memory became much larger, and (4) the computer became an interactive resource on which several persons could work concurrently (via time sharing).

With the third-generation revolution, automation of information systems which had been impossible to realize so far suddenly became feasible. For the realization (implementation) of such applications, large software systems had to be built. Due to their complexity, these systems could not be built by one person within a limited period of time. Therefore, they were implemented by a group of people, cooperating in a software project. Existing software-development methods, such as the *code-and-fix* model (Boehm, 1988), were used to build the systems, but the methods showed to be inadequate: the existing methods for building small software systems simply could not be scaled up. Hence, large-scale software projects finished too late (sometimes years too late or even finished not at all), and costs were much higher than originally predicted. The products delivered were unreliable and difficult to maintain, and their performance was often rather poor (Sommerville, 1992).

As a result, building large software systems was provocatively called *software engineering*. This reflected the opinion that an engineering approach to software development would enable software developers to deal with the increasing demands for complex and reliable software.

A real-world example may clarify the problems software engineers encountered. When preparing a meal for two persons, one person can very well manage the preparations, the cooking and the serving of the dinner. The details of the meal are known and a

simple time schedule, like "I'd better start with cooking the potatoes because these will take 20 minutes to prepare and the salad only takes 10 minutes" is sufficient.

Assume you have invited ten of your friends to have dinner at your place. For the occasion you will cook the same meal as you did just two days ago for the two of you. For twelve persons you will need two or three pans and somewhat more preparation, but you will manage, although you will have to work quite a bit harder to get dinner ready. You can scale all the quantities needed and adjust your schedule a little to assure everything to be ready in time.

Next, imagine the preparation of a banquet for about 800 persons. Is it possible for one person to manage this? Of course, the answer is no and the reasons will be clear: it is not possible for one person to buy food for 800 persons and bring it home nor is it possible for one person to prepare the food for 800 persons. Extra people are needed to scrap the potatoes, to cook the vegetables and to prepare dessert. It is also necessary to have new equipment, capable of holding the large quantities of food for the banquet. In addition, a schedule should be used to state explicitly time constraints, preventing the food from being served cold. Beyond this, where will all the banquet guests be seated? It is clear that only a scaling of the two-person cooking methods and techniques as suggested above will not work; you need *new* methods and techniques to manage the complexity of the task in front of you.

Returning to the software crisis, the techniques used to develop software for the second-generation computers were not appropriate anymore for the development of software for third-generation computers. The distinction between one-person programming and building large software systems was made clear by DeRemer and Kron (1975). They introduced the terms programming-in-the-small and programming-in-the-large. Programming-in-the-small deals with moderate-sized programs developed within a short period of time by individual programmers, as opposed to programming-in-the-large which deals with the efforts of groups of people to develop large software systems with long (intended) lifetimes. The transition from programming-in-the-small to programming-in-the-large evoked a need for new techniques and methodologies enabling control of the complexity of building large software systems. Already in 1975, Brooks pointed out that scaling of programming-in-the-small to address programming-in-the-large is not feasible (Brooks, 1975).

2.2 Complexity

Though it was believed that complexity would vanish over time, a decade later Brooks (1987) stated that there is no silver bullet as a final answer to complexity problems in software development. The software crisis, still

existing after 25 years, has made clear that complexity is an inherent part of software and software development. In this section we will systematically look at the three different forms of complexity, which make software development inherently difficult: (1) complexity of the *existing information system*; (2) complexity of the *development process*; and (3) complexity of the *resulting software system*.

Complexity of the existing information system Information systems are being used for centuries. They are used for "support or replacement of spoken or written communication in goal-directed activities" (Nijssen, 1989). Thus, an information system supports or replaces human communication. Human communication is based by and large on natural language, which is the conceptually richest language we have. An information system holds information for the real system it controls. Very often, the real system is a complex system, consisting of people, business goals, plans, production machines, and materials, etc. In order to control the real system adequately, the information system must contain information on all relevant objects in the real system. This information is used for decisions regarding the control of the real system. Although an information system contains abstractions of things in the real system, it is often as complex as its real system. Furthermore, to support or replace human communication, an information system must be able to represent the real system in sufficient detail. If such is not the case, the decision processes cannot function adequately. Thus, the complexity of the existing information system is caused by an equally complex real system.

Complexity of the development process The development process of a large software system can, *in principle*, be accomplished by one person. However, if the system is to become operational within a short period of time, the necessary work has to be split into separate pieces. Thus, several people can work on these pieces in parallel. In order to control the development process, an effective work-breakdown structure and a time schedule are needed, enlarging the complexity of the process. Regarding the division of work, *communication* of results between the different project members is crucial. If communication fails, the project will fail. The communication between project members requires a common "language" in which the object of communication, the *intended* software system, can be addressed. Such common languages range from graphical languages (diagrams) to programming languages. Still, the use of such a common development language has its drawbacks. First, it

needs to be learned by the project members. Second, there exist no standards for such languages. Frequently, different projects use different languages for communication. In addition to the problems with a common development language, the number of communication lines between the project members increases non-linearly with the number of project members.

Complexity of the resulting software system The software, i.e., the program for an automated information system is complex in itself, which makes it hard to understand. As Brooks (1987) describes, *complexity*, *conformity*, *changeability* and *invisibility* are the four essential properties of software. Software is inherently complex because of the following three reasons. First, no piece of software is the same; if two pieces of software *are* similar, they are coded as a subroutine or function, making it unique and generic. In this sense, software systems are different from computers, buildings, or cars, where repeated elements abound. Second, the state space of a software system is very large compared to other systems. The conception, description, and testing of such a system is difficult. Third, the different pieces of a software system mostly interact in a non-linear fashion. Scaling of such a system increases its complexity more than linearly.

Conformity, changeability, and invisibility influence complexity in a negative way. *Conformity* requires that a software system has the ability to communicate with other systems. This conformity requires software systems to have multiple interfaces, which enlarge the overall complexity of the system. *Changeability* of software makes it possible to extend a system's functionality by changing the software. Current software systems are indeed frequently updated in order to enhance functionality or to remove errors (bugs). *Invisibility* of software hinders its understanding. Since software is immaterial and abstract, it has no adequate geometrical representation in the way that a house has a construction map, and a electrical system has a diagram. Insufficient understanding of a software system leads to communication problems among project members, which introduces cost overruns, system flaws and schedule delays. Moreover, insufficient understanding leads to management problems of the development process.

Summarizing, the complexity of the existing information system is caused by the complexity of the real system it controls. This complexity makes it difficult to understand the existing information system. The development process of software is complex since it requires the work to be divided among project

members. Moreover, communication between project members requires the use of a common language. Due to the complexity of a software system it is difficult to construct and understand such a system. The insufficient understanding of a software system leads to functional and technical problems and to management problems.

In order to make complexity manageable, software scientists have developed software-development methodologies. The concept of a methodology is the subject of the next section.

2.3 Software-development methodologies

The software crisis and the complexities of software development spawned research activities into *software-development methodologies.* A software-development methodology offers a framework for the ordered and controlled development of a software system. In order to control software development, two aspects of the development need to be addressed: the process and its output. Therefore, a methodology generally consists of two parts: (1) a process model, and (2) a development model. A *process model* describes *which stages* in software development are recognized, and how they relate to each other. Such a model is used for controlling a project, i.e., keeping it within budget and time, using available resources optimally. A *development model* describes *how* the particular stages, distinguished by a process model, should be executed. The development model offers a set of *abstractions* to be used for analyzing an existing information system, designing a software system, and implementing a software system. The development model also offers methods for creating the intended output based on the input of a particular stage. The output of each stage serves as the input for subsequent stages.

To illustrate the process of system development using *stages* and *levels of abstraction*, we give an example of the construction of an accounting system in Figure 2.2. Our example is based on a similar example of Rich and Waters (1986).

Assume that a non-accounting manager faces the need for particular accounting information. He[1] produces a *brief problem description* concerning the information he needs. This brief problem description is forwarded to an accounting manager who elaborates it into a *detailed problem description.* Both the brief problem description

[1] We remark that the personal pronoun *he* and words such as manager, analyst, designer, and programmer refer to both male and female persons.

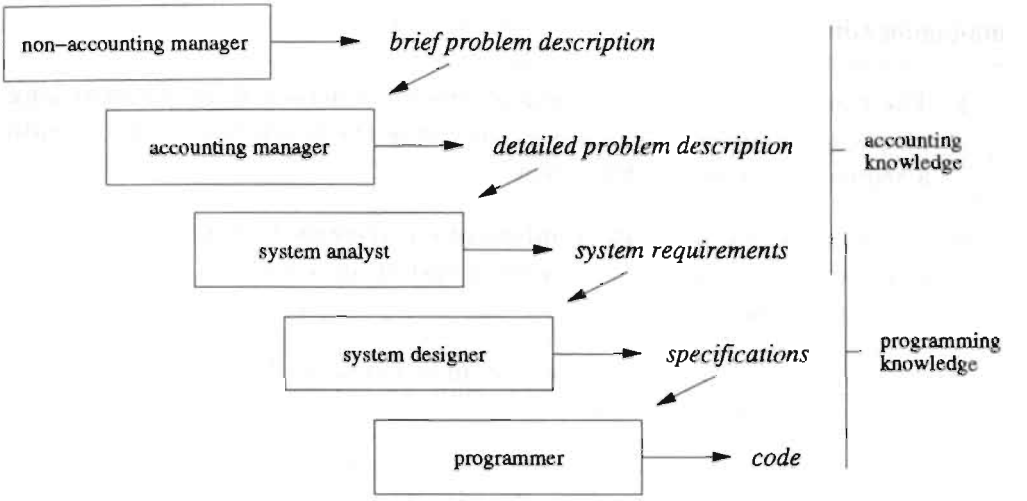


Figure 2.2: Agents necessary to produce an accounting system.

and the detailed problem description are formulated entirely in terms of the basic vocabulary of accounting and normal natural-language statements. Near the top levels accounting knowledge plays a crucial role.

At the next level, the system analyst analyzes the needs and wishes of the managers from a systems perspective. This results in a description of the currently existing information system (automated or not). Based on this analysis, the *system requirements* for a new system are determined. The system analyst must have a clear understanding both of accounting and of automated information systems in order to interface between domain specialists (the managers) and computer specialists (the designers and programmers).

One level lower, the system designer develops the basic architecture of the system. He translates the requirements into detailed *specifications* of the several parts of the software system. The specifications describe how the system can meet its requirements in a machine-independent way.

At the lowest level, the programmer uses the specifications of the designer and translates these into the *code* of a programming language. The code can be translated and subsequently executed by a computer. At this point the correct functioning of the system must be verified against the specification. Finally, the system must be validated, i.e., checked if it meets system requirements and solves the problems by formulated by its users.

The use of a methodology helps in controlling the complexity of software development. In the following three subsections we treat three concepts for

managing complexity.

1. The complexity of the development process is managed by distinguishing *stages* in the development process, dividing the development process into a sequence of manageable pieces.

2. The complexity of existing information systems is managed by using *models*, enabling analysts to view a system in a less complex and more abstracted manner.

3. The complexity of a software system is managed by the *decomposition* of a system into components.

2.3.1 Stages

The process model of a methodology divides the development of software into *stages* with well-defined inputs and outputs. The output of each stage serves as the input of the next stage. Ideally, the input of the entire process is a problem statement, which, through several transformations, results in a software system. All process models are based on the *software life cycle*, which describes the stages any piece of software will go through during its life. The software life cycle consists of six stages, which are commonly recognized nowadays (see, e.g., Sommerville (1992)). Below we provide a short characterization of each stage.

1. **Analysis** During analysis the needs and wishes of the users of the software system are analyzed. The result of this stage is a description of the currently existing information system (automated or not).

2. **Requirements specification** The software requirements, i.e., the system's functionality and operational constraints, must be specified. At this stage, *what* the system is expected to do is described.

3. **Design** Following the requirements specification, the system designer describes *how* the requirements may be accomplished in a machine-independent way.

4. **Implementation** During implementation the design is gradually converted into a machine-executable form, i.e., a program for a particular kind of computer system.

5. **Testing** After the implementation, the correct working of the system must be checked, i.e., it must be checked whether the system meets the requirements.

6. **Operation and maintenance** The system must be installed and used. If errors are discovered they must be corrected. The maintenance of a working system comprises two components:

   (a) tracking and fixing of bugs that show up during the operation of the system;

   (b) modifying the system due to changes in the environment.

Different varieties of process models arise because of different trajectories through the life-cycle stages. In the *stagewise process model* the stages are ordered linearly. Despite the linear order, it will often be necessary to go back to a previous stage since errors or inconsistencies, introduced during a previous stage, are discovered. Royce (1970) recognized the existence of such iterations or recurrent entrances , but in his opinion it was important to restrict them to successive stages of the life cycle in order to eliminate development risks. It has resulted in the *waterfall model* for software development, shown in Figure 2.3 (Royce, 1970).

This process model is most frequently used nowadays. The importance of iteration in the software-development process has been acknowledged by Boehm (1988), who proposed a *spiral process model* of software development.

2.3.2 Models

Whereas a process model, such as the waterfall model, describes what steps (including their order) have to be taken in software development, a development model describes what *kinds of model* are to be produced in a particular stage of the development process. During the distinct stages different kinds of 'things' are modelled.

A model is an abstract representation of reality that excludes much of the world's intricate detail. The purpose of a model is to gain understanding of the nature and behaviour of a phenomenon. A model reduces complexity by leaving out details that do not interfere with a phenomenon's relevant features.

The essence of building a model is abstraction, i.e., the formation of concepts apart from concrete things. In general, abstraction is the human way to deal

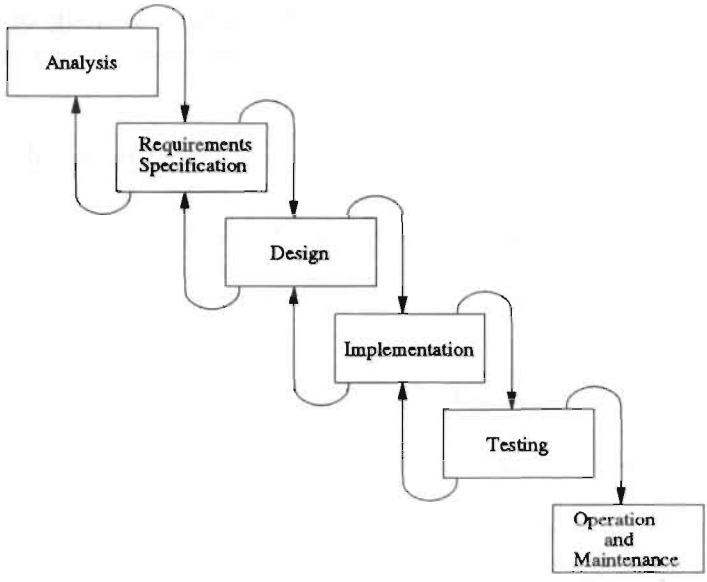


Figure 2.3: The waterfall model for software development.

with complexity by not taking into account too many details at one time. The essential characteristic of abstraction is that it enables the modeller to leave out unimportant detail, while retaining important information. The abstraction of an entity depends on the view the modeller has towards the entity and the purpose of the model being formed. A thing can thus be abstracted in different ways for different purposes.

A general definition of *model* in Webster's dictionary reads: "*a representation of a thing to be constructed or an object that already exists*". Clearly, this definition gives two different purposes for the construction of any model. First, a model is used to represent a thing to be constructed. In that way, the desired features of the 'thing' can be checked or verified. Such models are common in manufacturing of, e.g., cars and airplanes, where a model is used to examine the vehicle's essential features, such as its air resistance. Second, a model is a representation of an object that already exists. Thus, a model is used to represent some piece of a particular domain so that the modeller can subsequently gain understanding. These kinds of model are used in experiments to determine the influence different parameters of the model (should) have in reality. Such models are used in, e.g., economics, to acquire knowledge of financial markets and to make analyses accordingly.

During the distinct stages of software development, the kind of model being used varies. The purpose of analysis is to model an *existing* information system. An analysis model thus describes what is the "state of affairs" in the information system. Stated differently, an analysis model is a *descriptive* model. The purpose of design is to define the architecture of an automated part of an information system. Thus a design model is a *defining* model. Finally, a program implementing a design is an *executable* model. The level of each type of model indicates whether it contains domain-oriented concepts (high-level model) or computer-oriented concepts (low-level model). The interaction of stages and levels is shown in Figure 2.4.

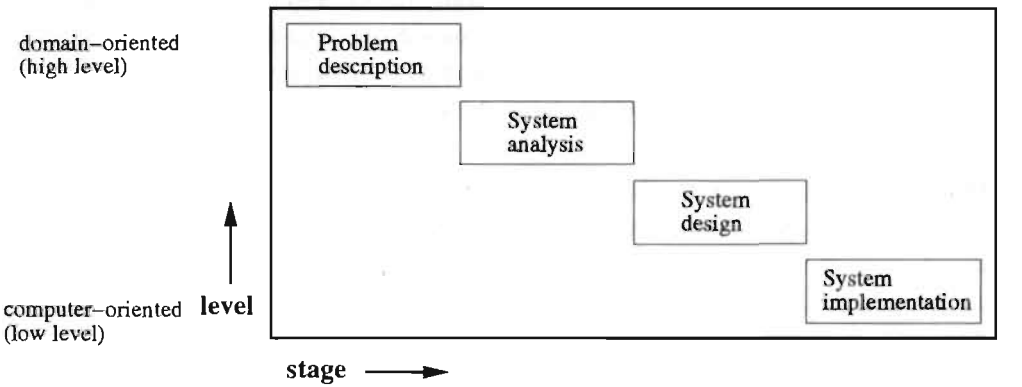


Figure 2.4: Stages and levels in software development.

A model is a conceptual thing. In order for a model to be communicated, it has to be represented in a *language*. Languages for modelling can be either *graphical* or *textual* (or both), depending on the purpose of the model. Analysis models are often drawn in a graphical language, since they must be comprehensible by domain specialists (viz. the accounting managers). Implementation models (i.e., programs) are written in formal languages so as to be understood by computers.

The representation of a model in some language is not merely a *syntactic* tool for communication. Although the representation consists of a well-defined combination of *tokens*, communication requires that the participants of communication (sender and receiver) share the common knowledge of the *concepts* expressed by syntactic tokens. At this point, the *semantics* of a model and its associated language becomes important. For effective communication, both sender and receiver must have the same conceptual knowledge of the semantics of the model.

Different methodologies use different models. We will not provide an extensive overview of existing models and methods. Instead, we focus on particular interesting ones in the scope of this thesis. The 'structured' modelling tripod addresses the stages analysis, design and implementation. Structured Analysis (DeMarco, 1979) models an information system by describing the *data flow* through different *processes*. This kind of model is represented in *data-flow diagrams* (DFDs). Structured Design (Yourdon and Constantine, 1978) is based on the division of a system into a *hierarchy of modules*, which communicate by means of parameters and flags. This kind of model is represented in *structure charts*. Structured Programming only uses three basic control structures (sequence, iteration and selection) in order to make programs readable and maintainable. Similar methods based on data-flow diagrams are the Warnier/Orr method (Warnier, 1976), the Gane/Sarson method (Gane and Sarson, 1979). For a short description of these methods, we refer to Orr *et al.* (1989).

Information Engineering (Macdonald, 1986) stresses the importance of an extensive *business model* describing the information within an organization. Based on such a business model several information-manipulating applications can be designed. In a similar line of reasoning, Jackson System Development (Jackson, 1983) stresses the adagium that "model precedes function". That is, before trying to build information systems, it is necessary to model the information being manipulated in these systems. The reason for this is the difference in stability between information on the one hand and functions manipulating this information on the other hand. The information in an organization will be much less subject to change, whereas the functions (read: systems) may vary according to the particular needs.

The entity-relationship approach (Chen, 1976; March, 1988) has been extensively used for the modelling of data, especially for relational database systems. It is based on modelling entities (read: things) in a domain, and the relations between these entities. It is widely used during analysis and design of database applications.

Natural-language Information Analysis Method (NIAM) (Wintraecken, 1985; Nijssen, 1989) is based on the analysis of information expressed in natural-language sentences. The choice for natural language has been made for reasons of communication and conceptual richness. The information in a domain should be phrased by domain specialists, i.e., users. Since natural language is known to everyone it is ideally suited for modelling information in a way which is understandable for other users. The accompanying development model

(ISDM) unifies the *information model* (what), the *process model* (how), and the *impulse model* (when) into a model of information systems.

Dietz (1987, 1992a) uses the concept of natural-language communication based on *speech acts* (Searle, 1969) in a model for information systems. An information system consists of active objects communicating about a UOD. The communication is based on two types of conversation. These are modelled according to speech act theory, which states that any utterance of a proposition has a *propositional content* and an *illocutionary force*. The propositional content is the proposition being uttered, and the illocutionary force describes what force the proposition possesses, such as a command, a directive, a declaration, or a denial, etc. The first type of conversation is the *fact-creating* conversation during which a state of affairs in the UOD is communicated between two active objects. Such a state is called a *fact*. Fact-creating conversations are based on the *statutive* and *acceptive* illocutionary forces. The second type of conversation is the *act-creating* conversation during which an actor proposes another actor to commit itself to execute some future action. These commitments are called *agenda*. Act-creating conversations are based on the *directive* and *commissive* illocutionary forces.

All methods cover parts of the software life cycle. In general there is a trend towards higher-level specification methods. JSD (Jackson, 1983) is one of the early methods recognizing that a model of information should be constructed before trying to model systems. The interesting ideas of Nijssen (1989) and Dietz (1992a) take information systems back to their origins of natural-language communication.

2.3.3 Decomposition

The human means to deal with complexity is abstraction. A well-known way of abstraction of a system is to decompose it into a set of *subsystems* or *components*. The division of a system into its composing subsystems is called its *decomposition*. The decomposition of a software system is often a hierarchy of components, in which each component itself is decomposed until a suitable level of detail has been reached. Sommerville (1992) names the following components in such a hierarchy for system decomposition: system, subsystem, program unit (module, procedure, or function).

The overall organization of a system and its subsystems is generally called the *system architecture*. The decision for a particular architecture can be based

on a theory for specific applications, or on a similarity between previous architectures. As a case in point, we mention the theory of compiler construction. The literature (e.g., Aho *et al.* (1986)) prescribes that a compiler in general consists of several passes, such as lexical analysis, syntax analysis, optimization, and code generation. In such cases it is wise to follow the theory and make use of existing knowledge. Likewise, once a particular architecture has been realized, it may be reused in other similar systems. In such cases it is wise to follow the general architecture, and decompose a similar system in the same manner. For example, operating systems are often designed with subsystems for process control, file management, and device control. New operating systems are often built in a similar way.

Decomposition helps to construct boundaries between different components of a system. The decomposition of a system requires each subsystem or module to have an *interface* through which operations may be invoked or particular data may flow. The specification of an interface, together with a description of the reaction that will be returned upon calling a function is called a *protocol*. An interface is an important concept for hiding internal parts of modules: the interface only allows access through well-defined functions (controlled access). The module also offers a form of independence. As long as the interface and the protocol are obeyed, a module's internal working may be changed without affecting its *clients* (i.e., other modules, which invoke functions). This provides system developers with a form of freedom and a means for division of work among project members.

With respect to the decomposition of a system into modules, the concepts of *cohesion* and *coupling* help a designer to decide where to put component boundaries. Cohesion is a measure of *relatedness*, whereas coupling is a measure of *dependency* between different abstractions. The strategy to define components is to group together related entities (high cohesion), and to make the coupling between components, i.e., the number of dependencies between them, as low as possible (loose coupling).

The decomposition of a system into components can be done in two distinct ways: horizontal *layers* or vertical *partitions*. In a *layered* architecture several layers of functionality are stacked upon each other. Each layer has access to the functions (often called *services*) offered by the next lower layer. A layer only has knowledge of its next lower layer; it has no knowledge of higher layers. Examples of layered architectures are the ISO model for Open Systems Interconnection (OSI) and the X Window System.

In a *partitioned* architecture partitions offer particular packages of function-

ality. Partitions are more or less independent of each other (decoupling). An operating system is often designed in such a way.

Summarizing this section, for managing complexity a software-development methodology comprises the three concepts *stages*, *models*, and *decomposition*. The concepts stages and models influence each other. During analysis a *descriptive model* is used, during design a *defining model*, and during programming an *executable model*. Decomposition is important irrespective of the stage or the kind of model being used. Both an analysis model, such as a data-flow analysis in a data-flow diagram, and an executable model, such as a program, must be decomposed into components.

2.4 Support for systems development

To offer support for systems development, we distinguish between two types of support: programming-language support and tool support. *Programming-language support* is needed to express the appropriate system concepts. *Tool support* helps a systems developer to focus on the creative aspects of development, while tools take over bookkeeping and translation tasks. Both languages and tools show a trend to raise the level of specifications. We first describe programming-language support, and then tool support.

2.4.1 Programming-language support

The trend in programming-language design has been to move away from imperative languages to languages describing the key abstractions in a problem domain. As a case in point, structured programming in the 1970s was soon followed by structured design and analysis, effectively raising the specification level.

As with hardware generations, there have been generations in programming languages. There are two important differences, however, between the evolutionary scenarios of hardware and software. First, new hardware has been invented and constructed by a technology *push*, and new programming languages were designed due to a market (= software developer's) *pull*. Second, today's computers do not use hardware components of thirty years ago, while one of the earliest programming paradigms, procedural programming, is still being used. An important aspect of the complexity of software systems, is the

dominant hardware style used in computers today. The so-called von Neumann architecture encourages a procedural programming style, and successive hardware generations have not changed this. The main attraction of new hardware has been a scaling of the von Neumann architecture, resulting in immense advances in processing power. In order to raise the level of specification, a programming language must support concepts corresponding to concepts in the problem domain. Nevertheless, a program in a programming language must be executable as well, meaning that a suitable translator or interpreter must be available. Translators and interpreters bridge the gap between high-level problem-domain oriented models and low-level bits and bytes of executable programs.

In successive programming-language generations, the kind of abstraction mechanism supported changed in order to support the growing need for more adequate abstraction concepts. Stroustrup (1987) and Booch (1991) give excellent overviews of existing programming paradigms, and their respective merits and shortcomings. We follow Stroustrup's survey of the evolution from machine-language programming to object-oriented programming (see Table 2.2).

Period	*Abstraction*
-54	no abstraction: machine and assembly language
1954-58	mathematical expression
1959-61	procedure
1962-66	module
1966-70	abstract data type
1967-72	object (class + inheritance)

Table 2.2: The evolution of programming abstractions.

Machine and assembly language Machine languages lacked any form of abstraction: a program consisted of statements from a machine's basic instruction set. A program was formed by entering code manually, and executing each statement. Assembly languages introduced names (mnemonics) for each instruction. This helped in memorizing the different types of instructions. We do not view such an introduction of names as a form of abstraction.

Mathematical expression Within a few years, computers were increasingly used for computing simple formulas. This resulted in languages for translating formulas automatically into machine code. A well-known language was FORTRAN, designed for *for*mula *tran*slation.

Procedural programming Procedural programming is probably still the paradigm most commonly used. Procedural programming focusses on the design of processing and the algorithms needed to perform the desired computations. The need for efficient and fast algorithms was caused by the costs of processing time, which were relatively high in the early days of programming. FORTRAN, initially developed for translating formulas into computer language, is the original procedural-programming language, but languages such as Algol60, Algol68, Pascal and C are languages in the same tradition.

Modular programming Over the years the emphasis on processing has shifted towards the organization of the data acted upon. The paradigm known as the *data hiding principle* emphasizes the importance of the data by partitioning a program into *modules*, i.e., sets of related procedures combined with the data they manipulate. Programming with modules leads to the centralization of data controlled by a type-manager module. Compared to the organization of functions and procedures using procedural programming, this is surely an improvement, but "types" created in this fashion are still different from the built-in types used in a programming language. Each of the type-manager modules must have procedures to create and delete variables, and make assignments to them. Variables created in such a way lack the usual scope rules and cannot be passed as arguments in the usual way.

Abstract data types To overcome the problems with modules, in languages such as C++ and Ada it is possible to define types that behave in (nearly) the same way as built-in types. These types are often called *abstract data types*, although a more appropriate name would be *user-defined data types*. The paradigm supporting this style of programming is called the *data-abstraction paradigm*. Abstract data types define a sort of black box. Once specified, a type cannot be adapted to new uses, except by changing the actual definition of the type. A well-known example of an abstract data type is the ubiquitous stack type on which elements can be pushed and from which elements may be popped. The implementation of such a type is hidden from *clients* (users of the stack type), who can only see the externally defined procedures pop and

push. The black box thus defined by the abstract data type, results in a severe shortcoming of programming with this paradigm. Once the data type stack is defined, it can be used. But the only way to extend the stack type is to copy its definition and provide additional functionalities, resulting in two functionally similar data types, which are unrelated in their definition. To eliminate the disadvantage of the black-box approach, it is necessary to include a mechanism for defining new abstract data types in terms of other ones. This latter feature is precisely the distinguishing feature of *object-oriented programming.*

As an illustration, suppose we have certain types of geometric shapes such as squares, circles and triangles. Some routines can be specified handling operations on objects of these types, e.g., routines to move, rotate and draw a shape. A routine must know what kind of shape is acted upon. Its structure will, e.g., consist of a Pascal case or C switch statement, enumerating all known shapes and performing the necessary type-specific actions. Assume now we want to add a new type of shape. Since the new shape must have the possibility of rotating, moving and drawing as well, it is necessary to examine and modify the existing routines on shapes. Every case statement enumerating the possible shapes it acts upon, must be extended to cope with the new shape, with the risk of introducing bugs in the existing code. The problem is that there are no possibilities to distinguish between the properties of *any* shape (a shape has a location, a shape can be drawn) and the properties of a *specific* shape (a circle has a center and a radius, a square has an upper-right corner and a lower-left corner). Operations should not decide which code to execute depending on the type of object. The object itself should be in control to choose what operation it has to perform. This is concisely phrased by Gorlen *et al.* (1990, p. 104) as "Switch statement considered harmful." Expressing and using the commonalities and differences between certain kinds of objects is the essence of object-oriented programming (Wegner, 1987a; Meyer, 1988).

Currently, object-oriented programming is being used progressively during implementation. The possibility to define types in terms of previously defined types enables the *reuse* of software. Object-oriented programming is a stage of an evolution of different programming paradigms. Similar evolutions take place in non-programming domains, such as enterprise-modelling languages, data-modelling languages, and semantic-modelling languages. In order to abstract from the programming domain and to assess the concepts of object orientation, we address *object-oriented modelling* (see Chapter 3).

2.4.2 Tool support

Software tools support a developer in focussing on creative aspects of software development. The lack of appropriate tools leads to analyses and designs which are not up-to-date, because of the trouble people have in modifying them. A data-flow diagram, once drawn by hand, is tedious to change because of its graphical complexity. In the following subsections we treat the evolution of software tools from *programming tools*, via *programming environments* and *computer-aided software-engineering environments*, to the use of *artificial-intelligence techniques* in software development, such as *automatic programming*, and *intelligent assistance*.

Programming tools

Programming tools are focussed on supporting the tasks of a programmer in the software life cycle. Since programmers in the early days of programming were the only interactors with a computer, it was a natural choice to provide them with appropriate tools first. On first-generation computers, programs were written in number code or mnemonic assembler code, which could directly be translated into machine code by an *assembler*. The transition from first-generation to second-generation computers enabled the introduction of high-level programming languages, such as FORTRAN and COBOL. *Compilers* and *interpreters* were introduced to translate programs written in such languages into machine code. The use of compilers also necessitated the use of associated tools, such as link editors and librarians. The set of tools to support a programmer has become known as a *programming environment*.

Programming environments

A programming environment consists of all the supporting software a programmer uses in the course of preparing his program. Compilers, linkers, loaders, librarians, and documentation tools are all part of a programming environment. Because computers in the 1960s were largely batch-oriented, early programming environments were also batch-oriented. The computer gathered a batch of jobs, say FORTRAN programs to be translated, and invoked the compiler for each of the jobs during night time. The next day, a programmer would check the results of his job and make corrections if an error had occurred. During the next night, the corrected version was compiled, and so on. In this way it could take some time to have a program running correctly.

The advent of the third-generation computers, changed the way of working from batch processing to interactive use of computer resources (see Table 2.1). Consequently, programming environments also changed from batch-oriented to *interactive programming environments*, in which a programmer invokes the tools he needs interactively. The effect of interactive programming environments was a gain in programmer productivity, because now it became possible to develop programs in an edit-compile-run fashion.

The tools in an interactive programming environment are the same as in a (batch-oriented) programming environment, extended with some specific interactive tools, such as syntax-driven editors, and debuggers. A characteristic example of an interactive programming environment is the UNIX Programming Environment (Kernighan and Pike, 1984).

Computer-aided software-engineering environments

Batch-oriented and interactive programming environments only supported the *programmer* (i.e., the final stage of the development process) in his task. The use of new concepts in software engineering, such as the stages of the software life cycle, paved the way for software environments for software development. Such environments have become known as *computer-aided software-engineering environments* or CASE environments.

Winograd (1973) recognized the problem of complexity in large software systems from his own experiences with SHRDLU, a program which conversed in English about a blocks world. Complex interactions between SHRDLU's components gave the program much of its power, but at the same time they were a formidable obstacle to understanding and extending it. Winograd suggested that the *complexity barrier* as he called it, could best be broken by building large programming support systems. He proposed a programming-support system, called **A**, with the following characteristics: (1) since **A** may require the user to do extra work (e.g., describe the motivation for a decision), the system must in return provide substantial support to the user; (2) **A** should be coherent and integrated; a system consisting of several tools pasted together would prove insufficient; (3) **A** should be capable of maintaining higher-level descriptions of the program; simply remembering the program at the source-code level is insufficient; (4) **A** should know a great deal of specific knowledge about the process of programming.

The first three characteristics are addressed by current CASE environments (Dart *et al.*, 1987). Characteristic four is still being studied; many papers have

been published on the subject, but the work still has not proceeded beyond the research stage. In the next subsection we treat the use of artificial-intelligence techniques during software development.

CASE environments are designed to support all stages of the software life cycle, from requirements specifications to maintenance. Their support can be confined to one specific methodology, or can encompass several methodologies. Most CASE environments are highly interactive and require high-resolution displays and pointing devices like a mouse or a touch tablet.

The tendency in CASE development is to raise gradually the level of the problems that can be handled by the environment. It is no longer only the programming stage that is being supported. Gibson (1989) signals a functional separation of CASE into upper, middle, and lower CASE. *Upper* CASE supports corporate planning, *middle* CASE supports systems analysis and design, and *lower* CASE supports automated-systems development. By this tripartitioning, a CASE environment is an integrated system of components supporting the stages of the software life cycle.

In the first stages of the life cycle, planners using upper CASE create specifications for the corporate plans, i.e., the planning requirements for the company's fundamental activities. During this phase the focus is on the company's core activities, which means that software systems are not addressed at all. Using middle CASE, system analysts convert the planning requirements to requirements for information systems. System designers refine the system requirements into design specifications. The design specifications finally are used to produce the development specifications. At this stage one enters the domain of lower CASE. System developers elaborate the development specifications to make them more comprehensive. Lower CASE systems use the development specifications to generate the application system and accompanying end-user documentation.

Thus, with CASE technology the computer is used as a development tool to integrate planning with the design and development of computerized information systems.

As we remarked in Section 2.3, model building is an important concept in software-development methods. Since model building is an iterative process, it requires the model builder to revise the various models during the development process. The benefit of CASE is that it encourages an iterative approach of modelling by making the revision of a model as easy as possible. Nevertheless, software development still remains a creative task, and the use of CASE tools will by no means replace the intellectual input from human software

developers. In order to come to intelligent tools, artificial-intelligence techniques have been used to equip software-development tools with intellectual capabilities.

Artificial-intelligence techniques

The ultimate goal of applying artificial intelligence to software development has been *automatic programming* (Rich and Waters, 1986). In the limit, automatic programming would allow a user simply to say what is wanted and have a program produced completely automatically (Simon, 1986). However, it is unlikely that this kind of system will be available soon, if at all. As a result, the application of artificial-intelligence techniques has shifted focus towards *intelligent assistance*. In this respect, the goals of automatic-programming research have become directed towards CASE.

Automatic programming The success of compilers, initially called automatic-programming systems (ACM, 1958, p. 8), tempted researchers to believe that automatic programming could indeed fulfil the need for a system translating users' wishes into programs. Automatic programming initially focussed on raising the level at which programs are specified. To this end, Rich and Waters (1986) consider the different levels of Figure 2.2. The phrase "a user simply says what is wanted" is then rephrased into three particular questions: (1) who is the user; (2) what does the user say; and (3) how does the user express himself?

Figure 2.2 shows the agents necessary to produce an accounting system. It also shows the levels at which automatic programming might be applied. At the lowest level, the programmer is the user of an automatic-programming system (a compiler), which converts high-level code into machine-executable code. In this case the user expresses himself in a programming language.

At the next two levels, the system analyst and the system designer would like to use a specification language as input to the automatic-programming system. Such a specification language would resemble a normal programming language with additional support for abstract concepts, such as sets and quantifiers, and systems and modules.

At the second highest level, the accounting manager should be able to input the requirements of the program into the automatic-programming system. At this level, the automatic-programming system replaces both the analyst and

the programmer and therefore should have extensive knowledge of accounting and programming.

At the highest level, the manager uses free-form English to state the problem and the automatic-programming system produces a solution. At this level the system would have to be an accounting expert itself.

Research into automatic programming has focussed on the *specification method* and the *method of operation* of an automatic-programming system. The specification method varies from a programming language to a natural language (viz. the levels in Figure 2.2). Since natural-language translation is still an open problem, it is at least doubtful whether this approach will ever become successful. The method of operation of an automatic-programming system varies from theorem-proving techniques to programming-by-examples. Yet, most efforts have not passed the research stage. Rich and Waters (1986) provide a collection of papers on the different aspects of software engineering and artificial intelligence.

Instead of full automation at some levels, partial automation of several tasks at all levels has shown to be more fruitful. In that way, a form of *intelligent assistance* is realized.

Intelligent assistance Instead of trying to translate natural-language specifications into programs, intelligent assistance is based on taking a slice through the software-development process. It provides partial automation at each level. This kind of system thus performs bookkeeping and takes over several other tasks. The advantage of this approach is that such a system can be used immediately, while incremental improvements are added to it.

The idea of a *plan-based assistant* is that many programs can be viewed as the composition of standard program fragments or "plans". Winograd (1973) proposed a system based on this principle (see also Section 2.4.2). Such a system is the focus of the Programmer's Apprentice project (Rich and Waters, 1990). In the future, the level of this kind of system (programming level) will be lifted to address design and analysis problems: Rich and Waters propose to proceed with similar concepts to construct a Designer's Assistant.

As Rich and Waters (1990) portray (see Figure 2.5), the next generation of CASE environments will have to be based on deep-representation concepts and inspection methods. Deep representation means that tools are not just based on the textual or graphical representation of program code or diagrams, but that they have a deeper knowledge of programs, in terms of composition of

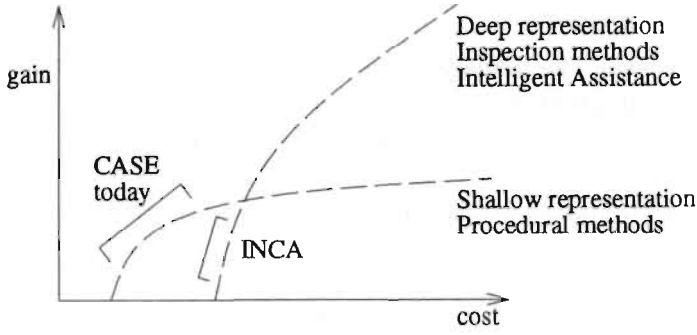


Figure 2.5: The expected influence of *deep representation* on CASE.

(adapted from Rich and Waters (1990))

standard fragments and compositional structures between these fragments. Inspection methods may be used by a tool to recognize fragments and structure of existing programs, thus providing a deeper representation of programs. In their opinion the results of these approaches will possibly be as revolutionary as the transition from assembly programming languages to high-level languages.

We hold the same point of view as Rich and Waters (1990) do in striving for a model of "deep representation" for CASE environments. In that way, CASE can offer full support for the development of information systems.

2.5 Problem statement

The software crisis still persists. Software without errors ("undocumented features") is hard, if not impossible to find. The reason for this lies in the inherent complexity of software and our unability to develop appropriate concepts and tools for the development of automated information systems. Too often, implementation is the only development stage during which formalized descriptions of a system are produced in the form of programs. Errors in specifications, discovered during implementation are costly to fix, since they require previous design and implementation decisions to be reconsidered.

In order to reduce or eliminate the number of errors in software systems, concepts and tools are needed to handle the complexity of systems development

throughout the software life cycle. At the implementation level of information systems, object-oriented programming languages are being used progressively. The reasons for this are the naturalness of concepts in object-oriented programming languages, and the possibility for reuse of specifications. However, current object-oriented programming languages only address the implementation phase of software development. In order to use the concepts of object orientation during analysis and design, the level of (object-oriented) specification should be raised. Currently, methods for object-oriented analysis and object-oriented design are being developed (e.g., Object-Oriented Design (Booch, 1991) and Object Modelling Technique (Rumbaugh *et al.*, 1991)). In order to facilitate the use of object-oriented concepts during analysis and design they should offer abstractions which are meaningful for human beings (i.e., users, analysts and designers) and which at the same time may be used by supporting software tools to assist a modeller during the modelling process. Thus, a set of concepts for object-oriented analysis and design should be developed for modelling information systems at a conceptual (i.e., non-programming) level. What are the requirements and characteristics for such a *conceptual object model*? This is the basis for the problem statement, which we formulate as follows:

> What is the nature of a conceptual object model for the analysis and design of information systems?

Below, five requirements for the conceptual object model (COM) are enumerated. A conceptual object model should offer concepts for:

1. representing the UOD of an information system. Using such concepts, a context for information systems can be created. The environment in which an information system has to become operational can be modelled.

2. modelling generic parts of a UOD. This enables different information systems to share a common context.

3. modelling specific parts of a UOD. This enables different information systems to create a particular context, suited to the purpose of the information system.

4. modelling information systems by means of well-defined (de)composition principles.

5. reuse of specifications to allow cheap systems to be built.

To meet these requirements, we propose a conceptual framework offering concepts for (1) modelling a UOD in terms of generic world knowledge and specific application domains, and (2) concepts for modelling information systems in terms of interacting objects.

Conceptual framework Obviously, object-oriented modelling plays a central role in the conceptual framework. The reason for this is twofold. First, object-oriented modelling is used within artificial intelligence as a knowledge-representation concept in the form of frames. Frames allow for representation of knowledge about a UOD by clustering information around (representation) objects. Second, object-oriented modelling is used within software engineering as a combination of different programming paradigms in the form of object-oriented programming. It allows for encapsulation, modularity, and inheritance. Moreover, object-oriented modelling enhances cheap systems development through reuse of specifications.

Our aim is to extend the object-oriented modelling concepts, so that (1) the semantics of a UOD can be captured easily in a natural way, (2) application domains consisting of application-dependent objects can be modelled, and (3) systems based on a particular UOD can be described. The framework is called the INCA *Conceptual Object Model*, or INCA-COM for short. Figure 2.6 shows the structure of the conceptual framework.

Below, we discuss the different parts of the framework. The core of the framework consists of generic knowledge about the *world*. It is independent of any application. Objects in the core are representations of things that exist in reality. Such world knowledge can be reused in different application domains. The layer of the *application domains* is used to model objects in a particular application domain, i.e., a domain of objects focussing on particular information systems. Application domains are thus constructed around the core of world knowledge. Both the core and the application domains provide an information-oriented view of a UOD based on objects. The layer of *applications* provides a process-oriented (what functions to perform) and impulse-oriented (when to perform) set of concepts. Both the representation of information and the representation of applications are based on object-oriented concepts. We remark that a methodology should guide the modelling process with these concepts. The structure of the conceptual framework already gives an important guideline on how to proceed during modelling.

In Chapter 4 we elaborate on the structure and concepts of INCA-COM. Before

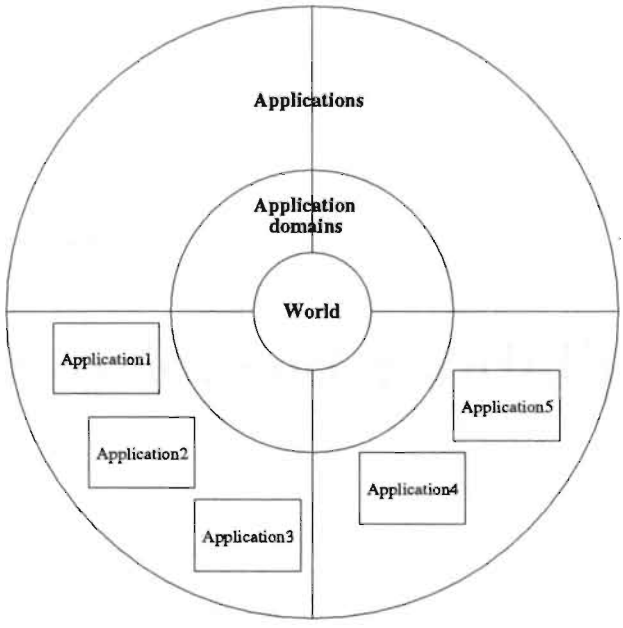


Figure 2.6: The structure of the conceptual framework.

doing so, in Chapter 3 we investigate the nature of object-oriented modelling and determine which concepts are particularly significant for the conceptual framework of INCA-COM.

Chapter 3

Object-oriented modelling

Object-oriented modelling refers to the use of object-oriented concepts for modelling a particular domain. In the literature, object-oriented concepts are often introduced on the basis of object-oriented *programming* (Cox, 1987; Gorlen *et al.*, 1990; Stefik and Bobrow, 1985) . Although useful, such an approach stresses too much the programming issues, and neglects the modelling concepts present in an object-oriented environment. The main purpose of this chapter is to describe, from a modelling perspective, the concepts made available by the object-oriented paradigm. Since programming is also a modelling activity, a brief examination of object-oriented programming languages is appropriate. The examination is followed by a generalization of programming concepts towards modelling concepts.

We first collect a representative set of concepts from the object-oriented programming languages. This provides some insight into the current state of object orientation at the implementation level. Next, we consider the purpose and nature of modelling, and the different kinds of models which can be realized. Subsequently, we take up the issue on the need of raising the level of specification in software-system development. Programming is a way of modelling leading to models to be executed by a computer. Thus, the need to raise the level of specification requires raising the level in modelling, which leads to *conceptual modelling*. Using the concepts of object orientation, conceptual modelling becomes *object-oriented conceptual modelling*. The use of object-oriented modelling in the context of INCA-COM is emphasized. The chapter is concluded with an integrating view on object-oriented conceptual modelling and the description of the set of concepts in an object-oriented environment.

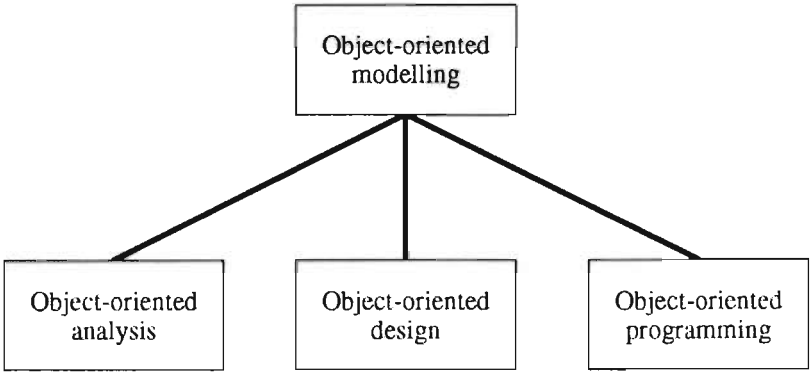


Figure 3.1: Object-oriented modelling as a generalization of various object-oriented techniques.

3.1 Object-oriented programming

Object-oriented programming can be seen as a special case of object-oriented modelling resulting in models to be executed by a computer. In general, object-oriented modelling is a term for a way of modelling, having various specializations, such as object-oriented analysis, object-oriented design, and object-oriented programming. This is illustrated in Figure 3.1.

It is generally accepted that object orientation as a *programming* paradigm started with Simula (Birtwistle *et al.*, 1977). Nygaard (1986) states that the ideas of objects and classes resident in Simula were in fact introduced partly when no appropriate mathematical model could be formulated to simulate an operational-research problem. The popularization of object-oriented programming is associated with Smalltalk-80 (Goldberg and Robson, 1983), and more recently, with languages such as C++ (Stroustrup, 1991), Objective-C (Cox, 1987), Eiffel (Meyer, 1988), and CLOS (Moon, 1989). In this section we treat a representative set of concepts found in such object-oriented programming languages.

3.1.1 Objects

A characteristic notion in object-oriented programming is the *object*, which is an encapsulated combination of data and procedures that act upon the data. An object has its own (possibly persistent) private memory representing its

state, together with its public interface, defining operations other objects may invoke. (Often, the term protocol is used to denote an interface. However, a protocol defines a typical interaction sequence between two or more objects, and thus has a dynamic nature, whereas an interface is static.)

Wegner (1987a) contrasts objects with functions which have no state or memory. In this view, an object is identical to a 'system with memory', able to respond to particular inputs with well-defined outputs: "An object has a set of 'operations' and a 'state' that remembers the effect of operations. Objects may be contrasted with functions, which have no memory. Function values are completely determined by their arguments, being precisely the same for each invocation. In contrast, the value returned by an operation on an object may depend on its state as well as its arguments. An object may learn from experience, its reaction to an operation being determined by its invocation history."

Besides state and operations (behaviour), Booch (1991) and Rumbaugh *et al.* (1991) suggest a definition, which includes object identity: "An object has state, behaviour, and identity; the structure and behaviour of similar objects are defined in their common class." The same notion of identity was already suggested in one of the early papers on INCA-COM (Braspenning *et al.*, 1989b). The *identity* of objects accounts for the fact that two objects which have the same structure and behaviour are counted as different. Thus, two objects representing two similar green apples of 50 grams, which can fall from a tree and can be eaten, are still distinct. The *state* of an object is expressed in the values of its *attributes*, which are representations of the object's features. Each object has its own private collection of attributes. The *behaviour* of an object is the way it responds to interface requests by invoking a *method* (i.e., a procedure or function corresponding to the requested operation).

An object fulfils two important roles with regard to the software-engineering concepts for abstraction and decomposition. First, an object is a natural representation of some entity in the problem domain. An object helps abstraction by offering an encapsulated unit of both state and operations. Second, the object provides a modular unit for decomposition by dividing a program into a set of interacting components. Other objects are not allowed to change the state of an object directly. Instead, objects request each others' services through their interfaces. Interface requests are typically sent via *messages*.

3.1.2 Classes and instances

Although a single object is a useful unit of abstraction, the common features of objects are specified in a *class*. This allows the class to act as the unit of abstraction in the specification of a program. During execution of a program, however, objects are the unit of execution. Classes form a sort of template through which objects can be *instantiated*. Objects belonging to a certain class are often called *instances* of that particular class. A class is similar to an abstract data type: it defines a set of operations that its instances will be able to perform. The main problem of programming with abstract data types (see Section 2.4.1) is the inability to define a new type in terms of a previously defined one. To alleviate this problem, classes in object-oriented programming languages are ordered into a *class hierarchy* which provides a means of *generalization* (going up the hierarchy) and *specialization* (going down the hierarchy). The ability to define classes in terms of more general classes is the essence of an object-oriented programming language (Wegner, 1989).

The top of a class hierarchy is the most generic class of which all other classes in the hierarchy are *subclasses*. At the lower levels of the hierarchy one finds classes that are more specialized. An example of such a class hierarchy of shapes for a graphical editor is presented in Figure 3.2.

The class Shape at the top of the hierarchy is the most general one; this class defines the most general attributes and methods of shapes. For example, Shape defines that each shape has a colour and a location, and additionally has methods to draw, move, cut and copy itself. The lower classes in the hierarchy are the more specialized ones. Specialization of a class into a *subclass* can be done in two distinct ways. First, a subclass can *extend* its *superclass* (i.e., its hierchically higher class) by defining new attributes or methods, in addition to the ones defined in the superclass. Examples of the extending form of subclassing are the classes Line, Polygon, and Oval. They extend the Shape class by defining attributes, such as a begin and ending point (Line), a set of points (Polygon) or two radiuses (Oval). Second, a class can *constrain* the values of some of the attributes defined in its superclass. Examples of the constrained form of subclassing are the Square and the Circle subclasses. The class Square constrains the length of the sides of a Rectangle to be equal, and the class Circle constrains the two radiuses of an Oval to have equal lengths. The two forms of specialization both make the descriptions of subclasses more specific (i.e., the intension of a subclass is greater than the intension of its superclass, meaning that it is applicable to less objects than its superclass).

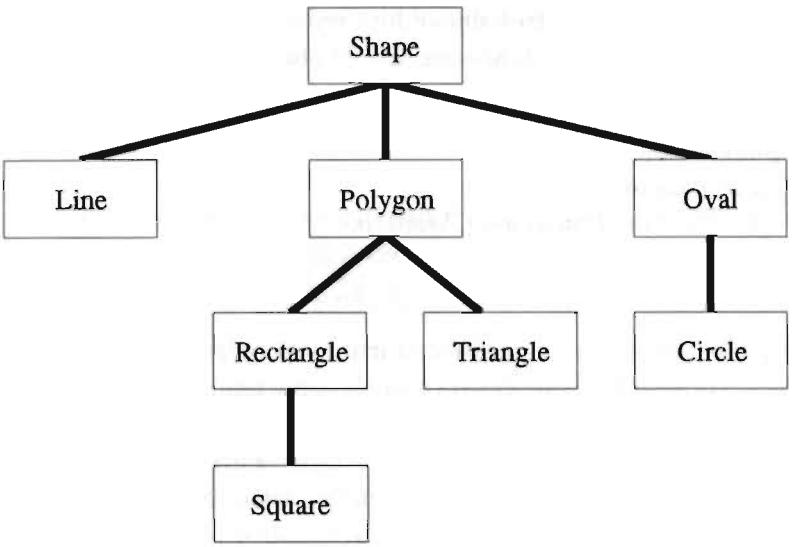


Figure 3.2: An example class hierarchy.

Therefore, both forms of specialization are shown in Figure 3.2 with the same type of line.

Some object-oriented languages allow subclasses to override completely the specification offered by the superclass. This way of (ab)using the generalization/specialization relation between classes should be avoided, since it denies the fact that the subclass IS-A specialization of its superclass.

Classes in object-oriented programming languages have three roles which especially focus on programming. First, a class object has a *methods dictionary* holding the methods for the specific messages that can be sent to its instances. Conceptually each object has methods to respond to incoming messages, but in most implementations of object-oriented programming languages, an object receiving a message uses the methods dictionary of its class. Second, the class acts as an *object factory* or *template*. Every class is equipped with a mechanism for constructing a new instance according to the template (i.e., the structure) specified by the class. Third, the class has a role as an *object warehouse* in the sense that the collection of its instances is in some way attached to it.

The distinction between classes and instances is based on the philosophical distinction between *generic concepts* and *individuals*. However, the distinction

is not always clear, and is often dependent on the context in which the concept is used. Consider, e.g., the following list of classes and instances, taken from Hofstadter (1979, p. 351):

(1) a publication
(2) a newspaper
(3) *The San Francisco Chronicle*
(4) the May 18 edition of the *Chronicle*
(5) my copy of the May 18 edition of the *Chronicle*

As Hofstadter points out, when viewed from *newspaper* (2), *The San Francisco Chronicle* (3) is one of its instances (namely a particular newspaper). However, when viewed from *The Chronicle of May 18* (4), *The San Francisco Chronicle* (3) is a class, one of its instances being (4). This line of reasoning seems to be right. However, there is a difficulty with this view. When talking about *The San Francisco Chronicle* as an instance of a newspaper and, more generic, as an instance of a publication, one is interested in the features of the *The San Francisco Chronicle* as a newspaper, such as its editor-in-chief, its place of publication, and its first year of appearance. However, the May 18 edition of the *Chronicle* is an instance of different class, the class of *San Francisco Chronicle* editions. For this class, features such as the date of issue, the number of pages, the number of articles, and the subjects covered are interesting. In a similar vein, my copy of the May 18 edition of the *Chronicle* is an instance of yet another class, The San Francisco Chronicle edition issue. The latter defines features such as the number of issues, the paper and ink used for printing, and the price of the newspaper. Of course, these classes are strongly related to each other. There might be relations along which default values for features for the different classes are distributed. It is the task of an information analyst to detect problem cases such as sketched above. The ambivalent role of classes often is caused by mixing different UODs, in which classes can have different roles.

In order to handle the ambivalence of classes, in some object-oriented languages, classes are implemented as objects. This, of course, gives rise to the question what the class of a class (viewed as an object) is. For this purpose, the concept of a *metaclass* is introduced. It is a class having classes as its instances. To break an infinite recursion, such a metaclass is often an instance of itself.

The ability to treat classes as objects offers in principle the same means of description as objects have: a class can have attributes and methods. A typical

example of using a class as an object is by sending it an instantation message (usually the message named **new**), by which the class is asked to instantiate a new object. *Class* attributes are typically used to model attributes which are common to all instances of the class.

3.1.3 Inheritance and delegation

The organization of a class hierarchy is based on the relations generalization and specialization between classes. A class A is called a superclass of a class B if A is a generalization of B. A class B is called a subclass of class A if B is a specialization of A. Sometimes the generalization/specialization relation is called the IS-A relation, reminiscent of the similar relation used in semantic networks. The specialization relation is used to express that the subclass *is a* particular case of its superclass. In virtually all object-oriented languages, the class hierarchy is used as a vehicle for a form of code sharing, referred to as *inheritance*. Some people call the generalization/specialization relation between two classes the inheritance relation (e.g., Snyder (1987)), but we avoid this. Generalization/specialization is the relation between classes, whereas inheritance is only a mechanism for the effective reuse of code among classes, valid since the subclass is a specialization of the superclass (see also Chapter 5).

There are two possible ways to share code in object-oriented languages. The first one is the use of *inheritance*, referred to above, and the second is the use of *delegation*.

Inheritance Inheritance structures come in two flavors. Some object-oriented languages organize the class hierarchy as a tree, which allows classes to inherit directly from *one* superclass. This form of inheritance is referred to as *single inheritance*, *hierarchical inheritance* or *linear inheritance*. The second possible form is to allow classes to inherit directly from more than one class. The class hierarchy thus forms a *directed acyclic graph* or *lattice*. This form is called *multiple inheritance*.

Delegation Whereas inheritance is used for code sharing, based on the class hierarchy, Lieberman (1986) presents *delegation* as the means for code sharing among objects (not classes). Delegation is based on message passing to mimic the effects of code sharing using inheritance. The idea is that an object that does not know how to respond to a message should have objects to which the message may be forwarded instead. In the literature, much has been said

about the difference between delegation and inheritance. Lieberman (1986) stated that delegation is a more powerful concept, but Stein (1987) has shown that every concept using delegation can be realized with suitable inheritance mechanisms.

An important difference between the two code-sharing concepts is that in delegation systems a method is executed in the context of the delegate, whereas with inheritance a method is executed in the context of the delegating object itself. Inheritance can thus be seen as copying the method from the superclass to the delegating object. Another difference is that delegation is a relation between two objects, whereas inheritance is based on a relation between classes.

3.1.4 Messages and methods

For communication, objects can send each other a *message*. The message concept in object orientation is used in different ways. First, message passing can be used as a *metaphor* for communication. This corresponds to the use of messages in Smalltalk, where sending a message to an object has the effect of a dynamically bound procedure call (or method invocation). Second, message passing can be used as a way of communication. This form is mostly used in parallel and distributed systems. In such systems, messages are entities, which are sent between objects.

Objects can request other objects to perform some operation by sending it a message. A message generally consists of four possible parts: (1) the *receiver*, i.e., an identification of the object that is to receive the message; (2) the *sender*, i.e., an identification of the object sending the message; (3) the *selector*, i.e., the name of the operation to be performed by the receiver; (4) *parameters* needed to perform the requested operation.

The selector in a message specifies which action is to be performed. The receiver (or some kind of dispatcher) checks its interface to see whether it has a *method* corresponding to the selector, and if so, it invokes this method. A method is thus a procedure that is indirectly invoked by other objects via messages.

The use of messages introduces *polymorphism*. Polymorphism is not a particular charachteristic of object-oriented programming languages. It is also found in procedural languages. In general the term polymorphism means "having or assuming different forms". In the context of object-oriented programming,

polymorphism refers to the capability of variables to contain values of different types. Messages sent to such a variable result in an appropriate, polymorphic response, depending on the actual type of the object the variable is referring to at the moment the message is sent. Using polymorphism a programmer no longer has to provide code for checking the actual type of a variable: the run-time environment of a program checks the type of object and invokes the appropriate method. Polymorphism facilitates the extension of programs since new types may easily be added, by providing a subtype with specific operations. There is no need to update existing code to include type checks for the new subtype: the run-time environment handles such objects appropriately.

Polymorphism allows programs to address uniformly objects that arise from different classes. It extends the notion of modularity by allowing objects to interchange as long as their interfaces are the same. For example, in an object-oriented program classes for Integer numbers and Real numbers often have the same interface, consisting of the definition of operations for adding and subtracting numbers. Thus, instances of both classes can be used in arithmetic expressions such $2 + 3.4 - 0.98$. In Smalltalk, the execution of this expression results in a '+' message sent to the object 2, with argument 3.4, resulting in an instance 5.4 of the class Real, which is subsequently sent the '−' message with argument 0.98. Both classes of Integers and Reals have the same interface, but the implementation of their operations may differ considerably.

Related to polymorphism is *operator overloading*, which means that an operator may be used by different classes to refer to similar operations. An example of overloading, is defining the operator '+' both for the class of numbers, as an arithmetic operator, and for the class of character strings, as a string concatenation operator. The gain of overloading is an enhanced legibility of programs, since operators may be used in different classes to refer to suitable operations.

Polymorphism and operator overloading are not solely features of object-oriented programming languages. Procedural languages may also include them.

3.1.5 Typing and binding

A type is a set of objects that share some properties, notably the operations that can be performed on them (see, e.g., Wegner (1987b)). Programming languages can handle typing in two different ways. The first way is to require that each value is to be annotated by the programmer to be of a particular type. This is commonly referred to as *strong typing*. The main advantage is

that the compiler can check for possible *typing conflicts* in an early stage. The second way of typing is to have each value carry with it typing information during run time, so operations to be performed on a value can check their applicability. This form of typing is called *weak typing*. The drawback is that an operation may find out during run time, that one of its parameters is of an incorrect type, resulting in a run-time type error.

As an advantage of weak typing the ease of programming is mentioned. Indeed, the programmer does not have to specify any typing information, but he still has some sort of information about the usage of certain variables. Making such knowledge explicit, is precisely what typing is about.

The class hierarchy is used often as the source of typing information, but America (1987) points out that subtyping and subclassing should not be mixed. Subtyping is a relation between classes sharing some external interface (i.e., the operations one can perform on instances of the type), whereas subclassing enables reuse of code of the superclass by its subclasses. In other words, subtyping specializes the *interface* of supertypes, whereas subclassing specializes the *description* (or in this case, *implementation*) of operations specified by the interface. Mixing both specialization concepts leads to discussions on the 'right' way of using the subclass relation. An illustration of such a discussion on the subclass relation between a class Square and a class Rectangle is found in ACM (1993, p. 112). In the concept-oriented view, a Square is a special case of a Rectangle; thus, Square should be modelled as a subclass of Rectangle. However, in the program-oriented view a Rectangle is a Square, with additional operations with respect to the operations defined for Squares.

Binding is the process of linking messages or procedure calls to the program code to be executed. There are two forms of binding. The first form of binding is called *static* or *early binding*. In languages like C and Pascal, all procedure and function calls are said to be statically bound, because the compiler can solve at compile time which routine has to be executed. The second form of binding is called *dynamic* or *late binding*. In object-oriented languages like Smalltalk, it is not known at compile time which method is to be called in response to a message, because the (class of the) recipient of the message is not known at compile time. In such languages the actual code to be executed is selected during run time. Typing and binding are two independent concepts. So it is possible to have strong typing, combined with late binding, as, e.g., C++ illustrates.

3.1.6 Concurrency

A source of inspiration of the Smalltalk project at Xerox was Alan Kay (1977). His vision of the future of computing was built around the Dynabook, a powerful computer like today's workstations, with the size of a notebook, and with graphical capabilities as well. In Kay's view, Smalltalk should become the (programming) language for communication between user and system. The objects in the system were to be all little machines, running concurrently and independently, communicating via messages.

The application of concurrency to object orientation has led to the development of object-oriented concurrent languages. An important feature of concurrent languages in general is the means of interaction between concurrent objects. The most important characteristic of the interaction mechanism is, whether it is *synchronous* or *asynchronous*. A synchronous construction inhibits an object to perform any action after it has sent a message. An asynchronous construction allows an object to continue its activity after it has sent a message. Since an interaction in an object-oriented language requires one object sending a message, and another receiving it, there are four possible communication primitives: blocking send, non-blocking send, blocking receive, and non-blocking receive. Using these communication primitives, a language designer may construct different kinds of interaction mechanisms. The character of a parallel language is determined by the choice of interaction mechanisms.

Tomlinson and Scheevel (1989) list the following five interaction mechanisms found in object-oriented programming languages.

Asynchronous message passing With this interaction mechanism, the receiver blocks until a message is available. The sender is non-blocking and hands the message over to the underlying machine, that is responsible for buffering messages. Graphically this form of communication is depicted in Figure 3.3a.

The activity of an object is shown in a straight line, with time continuing downwards. Solid lines indicate activity of an object, whereas a dashed line indicates that an object is inactive, waiting for an external stimulus. In Figure 3.3a object **B** is blocked waiting for a message. As object **A** sends a message **M** to **B**, **B** is deblocked.

Synchronous message passing With this construction both send and receive operations are blocking, as is shown in Figure 3.3b. Synchronous

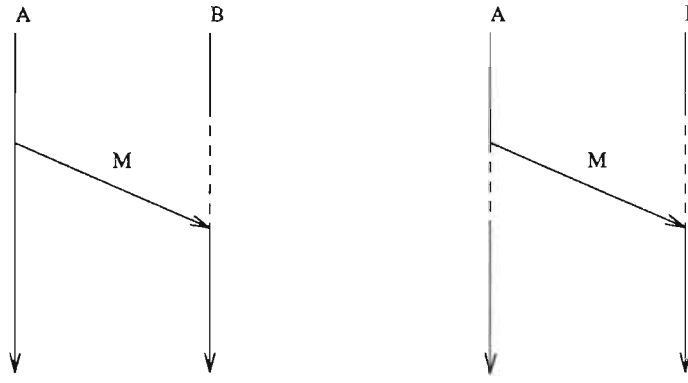


Figure 3.3: (a) Asynchronous message passing and (b) synchronous message passing.

message passing does not need a buffering mechanism for messages, since the objects themselves implicitly take care of buffering.

In *remote procedure call* (rpc) mechanisms, an object has two possible send operations, a *call* and a *reply*, and one receive operation.

Non-blocking rpc Object **A** is blocked from the moment it sends a call to object **B** until **B** returns a reply. **B** is only blocked from the moment it is ready to receive a call, until it actually receives one; **B** is not blocked after sending the reply to **A**. The non-blocking characterization of this interaction style comes from the non-blocking reply of **B**.

Future rpc The future rpc allows the sending object to continue with its activity, until the result of the rpc is needed. A reference to the result will block the sender if the result has not been returned yet.

Blocking rpc The blocking rpc is similar to executing a program serially. Parallelism occurs if object **B** can handle a call of a third object, while object **A** is performing computations not involving **B**. Any type of parallelism can be used for *distribution* of an otherwise serial language over different nodes in a network.

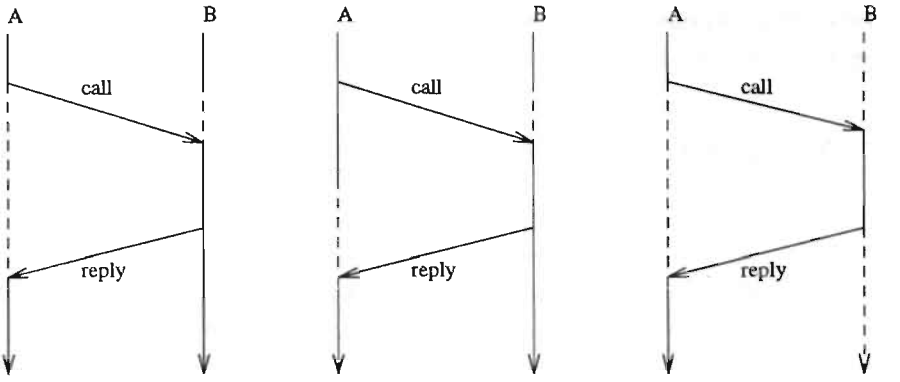


Figure 3.4: Remote procedure call: (a) non-blocking, (b) future, and (c) blocking.

3.1.7 Reuse and reusability

As early as in 1968, in the NATO workshop on the software crisis, McIlroy (1968) advocated mass-produced software components, which should be reused to decrease software-development costs. In analogy with hardware with its catalogs of VLSI devices, Cox (1987) has proposed to develop software ICs. Such components can easily be reused, as the level of reuse in hardware development today illustrates.

There is a clear distinction between software *reuse* and *reusability*. Reuse refers to the act of reusing software and reusability refers to the ability of software to be reused. Reusability is thus a prerequisite to actual software reuse. The most serious reason why software reuse has not succeeded so far is the lack of reusability. Specifically, the mechanisms to package software, such as the common notions of module, are not appropriate (Meyer, 1988). Objects provide a natural way to compose a system of a set of components. An object which has other objects as its *parts* is called a *composite object*.

Decomposition and composition (see also Section 2.3.3) are invaluable concepts both during analysis and design of (software) systems. The analysis of an existing system can be done effectively with objects (Wand and Weber, 1990). Further steps in analysis can decompose particular components into subcomponents, until a suitable level of detail is reached. Such a top-down approach is useful during analysis. Design requires a mixed approach of top-

down decomposition and bottom-up composition. Reuse requires that particular low-level components can be combined into larger components, eventually forming parts of the intended software system.

3.1.8 Memory management and persistence

An important issue in the implementation of object-oriented systems, is *memory management*. In general, two aspects of memory management are considered: (1) object construction and destruction, and (2) garbage collection. Both aspects can either be arranged by the programmer or by the run-time environment of a program.

Object construction and destruction New objects are constructed by specialized object-construction methods often attached to a class of objects. Such methods are called *constructors*. They may be invoked explicitly by a programmer providing a statement calling the constructor, or implicitly by the run-time environment when the scope rules of the language necessitate the creation of an object.

In a similar way, objects are destructed by so-called object-desctruction methods, called *destructors*. They are invoked by the programmer, or by the run-time environment when an object needs to be destructed according to scope rules.

Garbage collection Garbage collection is the process of reclaiming memory space that is not used, or that is used by objects which are no longer being referenced by a process. Garbage collection is often necessary since a process does not have an unlimited amount of memory. A special process, the *garbage collector* is responsible for keeping track of references to objects being used. The garbage collector may be activated explicitly by a programmer, or by the run-time environment when available memory space becomes too small. On activation, the garbage collector reclaims the space occupied by unreferenced objects. Various algorithms for garbage collection are available, such as reference counting or generation scavenging.

Both the explicit and implicit forms of object construction/destruction and garbage collection have their merits and drawbacks. Implicit garbage collection has the advantage of levering the programmer from specifying the details

of when and how memory is to reclaimed. The disadvantage is the unpredictable moment at which the garbage collector is activated, making such an arrangement ill-suited for real-time applications. The latter disadvantage is remedied by making the programmer responsible for deallocating unused memory, allowing detailed control of memory management. The disadvantage of this approach is that it is cumbersome and error-prone. Errors in programs are often caused by incorrect memory (de)allocations. Such errors are hard to reproduce and thus difficult to correct.

In traditional languages the lifetime of data usually is not longer than the duration of a particular program. Databases store data whose persistence transcends that of individual programs. When an object-oriented language is equipped with persistence, it can be used as a basis for an *object-oriented database*. Objects provide an adequate starting point for databases, since they introduce the concept of *state* between the execution of operations. Object-oriented databases are in some important aspects more appropriate for storing large amounts of information than relational databases.

1. An object-oriented database is capable of *handling composite objects*. A relational database only provides tuples, so that composite objects have to be mapped onto lower-level structures, i.e., they have to be decomposed and stored in multiple tuples. A client of a relational database must therefore have extensive knowledge of the database-representation schema in order to retrieve such composite objects.

2. In an object-oriented database, an object has an *identity*, which is independent of its value. In a relational database, there is no concept of identity: an object is represented as a 'bundle' of properties. Two tuples with equal values for their properties, representing two distinct objects, cannot be distinguished as being different in a relational database. Object identity is an important feature for sharing, updating and persistence.

3. A *class hierarchy* in an object-oriented database allows classes/types to be defined in terms of other classes/types. In a relational database, such a mechanism is not available.

4. *Encapsulation of state and behaviour* enables both data and operations to be stored in an object-oriented database. Using a relational database forces a separation between data in the database, and the operations performed on the data by application programs.

In this section, we have presented a representative set of concepts in object-oriented programming languages. From a modelling perspective, programs in an object-oriented programming language are models to be executed by a computer. In the next section we discuss the use of modelling and its role in problem solving. In the sections thereafter, the discussion of object-oriented concepts for modelling is taken up.

3.2 Modelling

In this section we discuss the nature of modelling and examine its usefulness in handling complex problems (a special case of which is software-systems development). When doing so, we seek answers to the following three questions:

1. why is modelling useful?
2. what is modelling? and
3. what is a model?

Problem solving in large-scale, complex systems requires adequate means to handle complexity. In the systems-oriented approach modelling is the central process of representing a problem domain into a model for subsequent analysis. The usefulness of modelling in problem solving is *problem simplification by abstraction.* By representing a system in a model which is simpler than the original system, the complexity of the system is reduced, thus simplifying the problem. Simplification of reality is necessary for the following two reasons: (1) human beings are unable to handle the complexity of reality; and (2) not every detail of reality is important for the problem under consideration. Working with models requires the human modeller to be aware of this simplification. The problem under consideration indicates which aspects of reality are important. It is the task of the modeller to determine the important aspects to be represented in the model of the system. The quality of a model is measured by its purpose for the problem at hand. So, the quality of a model depends on the kind of problem and the ability of the modeller to recognize the important aspects of the system. This implies that there is no single true model for all problems to be solved.

A system is defined by the elements or objects contained in it, together with the relations between these objects. The latter defines the *structure* of a system.

For solving problems using a model, there must be some form of *correspondence* between the original system and the model. In fact, this correspondence between system and model makes it necessary for the model to be a system itself, for how can a system be modelled by something differing from it in nature? Thus, the use of models is always based on at least two systems: S, the system to be modelled, and M, the system used as the model of S. In order for M to be an adequate model of S it has to be *isomorphic* with it. An isomorphic transformation of a system into another is a transformation in which the *structure* of both systems is similar.[1] In other words, the structure of the relations of S must be reflected in the structure of M.

A well-known example of two isomorphic systems is the hydro-electrical isomorphism, by which an electrical system is modelled by a similar waterworks system. In such an isomorphism, pipes correspond to electrical wire, taps to lights, pumps to batteries, valves to resistors and containers to capacitors. The structure of an electrical network, i.e., the electrical wire connections between the elements, should be translated into similar pipe connections between elements in the waterworks system. If done properly, the behaviour of a shining electrical lamp can now be studied by opening the corresponding tap. Since the waterworks system is more familiar, it is helpful to use this model in order to understand the electrical analogy. Even only the mental transformation of an electrical system into a waterworks system is often insightful. Of course, the waterworks system is not useful in all aspects. For example, the electrical effect of a short circuit can be simulated in the waterworks model, but will never lead to fire as it does in the electrical model.

We can now be more precise on the steps a modeller has to make during modelling. First, the modeller has to select a part of the problem domain which is of interest to the problem under consideration. This comes down to selecting a *system* in the problem domain, in which the problem occurs. Second, the modeller has to determine which *aspects* of the system are to be represented in the model. This entails the determination of an *aspect* system of the system. Finally, the resulting *aspect system* has to be represented as a model system by applying an isomorphic transformation on it. All this is condensed in the following definition.

Definition 3.1 Modelling *is the process of (1) determining a system in the problem domain; (2) determining an aspect system of the system; and (3) finding an isomorphic transformation (translation) of the aspect system into a model system.*

[1] The term *transformation* often refers to a relation in which the *form* of the two systems is not similar. In this thesis the term *translation* is used to denote an isomorphic transformation.

We note that in this definition the determination of a system includes (1) finding the relevant entities, and (2) finding the relevant relations between these entities. It is not sufficient to consider a system as a set of related entities, since the nature of the relations would then still be unknown.

Sometimes, the correspondence between a system and its model is claimed to be based on a *homomorphic* transformation. That is, a transformation in which two or more different relations in the original system are transformed into one relation in the model system. However, we think that such a modelling approach based on homomorphism does not distinguish between the three aspects of modelling as given in the definition discerned (i.e., the modelling approach based on a homomorphic relation between system and model system is itself homomorphic).

The definition of modelling leaves implicit what the nature of a model is. It suffices here to say that a model of a system is itself a system. We will come back on the relation between systems and models later in this section.

3.2.1 Representation and interpretation

Models can have a physical nature, such as model cars and model trains. In computer science the nature of models is conceptual, i.e., consisting of (mathematical) concepts, such as data flows, functions, and predicates, etc. Since a conceptual model is often used as a means for communication too (e.g., between people), one needs a mechanism to convey the model to a medium for communication. This in general comes down to making a *formalization* of the model, such that its meaning is preserved in the formal description of the model. In order to communicate (or to visualize) models in computer science, a *language*, i.e., a system of coded signs is used. Here, a language is a collection of agreements on the use of *tokens* that have a particular *meaning*.

Bunge (1974, p. 8) defines signs used for communication in the following way: "An artificial sign, whether written, uttered, or in any other guise, is a physical object which (1) *represents* some other object (physical or conceptual); (2) belongs to a *sign system* (language), within which it can concatenate with other signs to produce further signs, and such that the whole system is used for (3) the *communication* or transmission of information concerning ideas, etc." Languages can be divided into *symbolic* and *non-symbolic* languages. The latter are of no interest for computer science, since they have a physical nature. An example of such a language is the scent female animals release to attract male partners.

Symbolic languages are based on symbols, which by convention designate certain objects. The *syntax* of a symbolic language defines the set of basic symbols or tokens (the alphabet) and the rules (the grammar) according to which the basic symbols can be concatenated to form *expressions* (also called well-formed formulas). The tokens of a language designate certain objects in the real world. The *semantics* of a language is given by adding to its syntax a coding function which associates every token with a set of objects called the *denotatum* of the token. For example, in arithmetic, numbers are designated by numerals: 'number 3' is designated by the numerals "3" or "III". For effective communication using a symbolic language it is necessary for the *sender* and the *receiver* to share a common background in order to understand the tokens of the language. A subset of symbolic languages, so-called *conceptual languages* focus on the expression of *concepts*. Such languages are used in mathematics and computer science. In a conceptual language, tokens designate concepts.

The modelling process consists of making a suitable *representation* of a UOD. The process of representation is based on conceptualization, and subsequently formalization. Both a conceptualization and a formalization assume a basic set of concepts and tokens, respectively. This is shown in Figure 3.5. The dashed boxes in the middle provide the basic concepts and tokens. A formalization may be used for communication. To this end, the sender transmits the tokens of the formalization by means of signals through some channel. The receiver detects these signals and reconstructs the tokens by interpreting the signals. Of course, the receiver also needs to know *which* tokens can be accepted in the transmission process. Subsequently, the tokens are interpreted to find the corresponding concepts. The process of reconstructing tokens, concepts, and eventually the (possible) reality intended by the sender is a process of *interpretation*, shown in Figure 3.5 on the right hand side.

The relations between a token, the concept it designates, and the objects in the world the token denotes are shown in Figure 3.6, which is known as the *meaning triangle* (Sowa, 1983). A concept *refers* to objects in the UOD, which are the objects to which the concept applies. A *generic* concept (e.g., car) is applicable to several objects, while an *individual* concept (e.g., Amsterdam) only refers to one object. The combination of designation and reference is called *denotation*: a token *denotes* an object in the UOD. When the denotatum of a token happens to be conceptual, designation and denotation coincide. Such is the case, e.g., in arithmetic where "3" both designates and denotes 'number 3', since 'number 3' is a conceptual denotatum.

The relations in the meaning triangle are the basis for defining the concepts

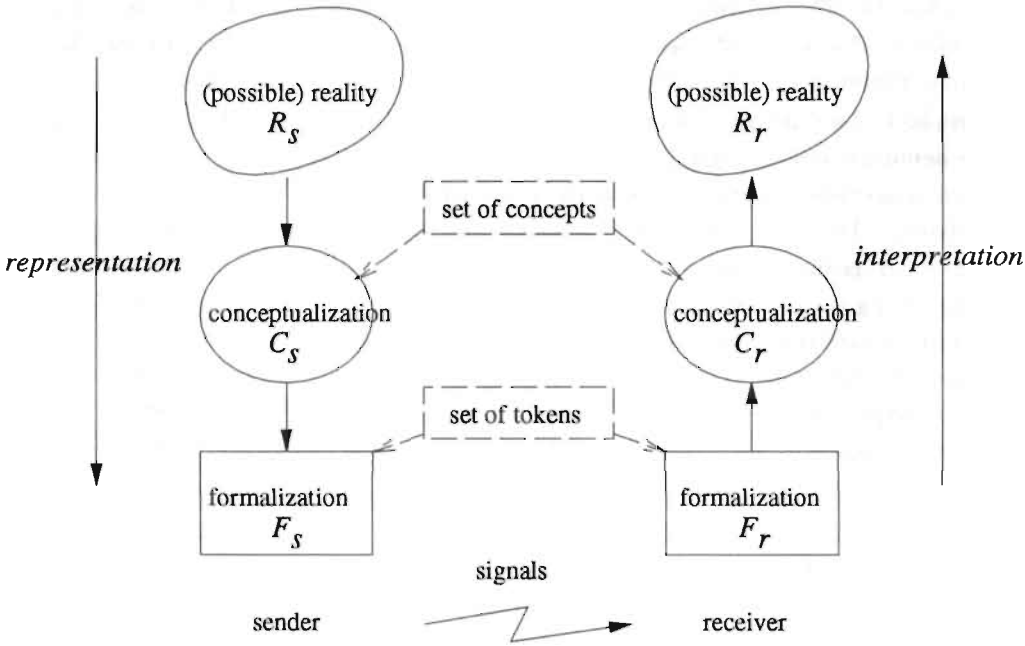


Figure 3.5: Representation and interpretation.

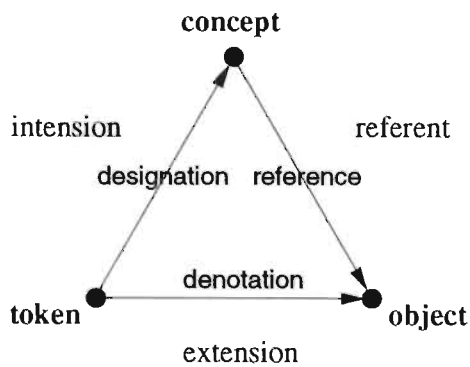


Figure 3.6: The meaning triangle.

intension, *extension*, *referent*, and *population*; the first three are shown in Figure 3.6. The object or set of objects denoted by a token is called the *extension* of the token. The concept, more particular the meaning of the concept, is called the *intension* of the token. The *referent* of the concept is the set of objects in the real world referenced by the concept. The extension of the token and the referent of the token's concept are equal. The extension of a token is the set of *existing* or *possibly existing* objects in reality. In order to discriminate between *possibly* and *actually existing* objects the term *population* is used to refer to the latter. Thus, a token's population is a subset of its extension.

As an example, consider the word "car" which represents the concept car. The concept car is the intension of the word "car". The set of all cars (existing or possibly existing) is the extension of the word "car", and the referent of the concept car. The population is the actual set of existing cars. The meaning of the concept car consists of the properties which discern a car from other vehicles, such as bicycles, boats and airplanes, and the relations the car concept has with other concepts, such as persons, roads, and traffic. In general, a concept's meaning consists of its discerning properties with respect to other concepts and relations with other concepts.

3.2.2 Types of models

In systems theory a model of a system is itself a system. In fact, Apostel (1960) and Dietz (1987) define a model as a role a system can play with respect to another system: "someone, who uses a system M, which is independent of a system S, in order to understand system S, uses system M as a model of system S." In 't Veld (1981) states that, since a model is a simplification of the problem domain it models, a model is a system at a 'lower level of aggregation' than the system being modelled. This is identical to our definition that a model system is a representation of an aspect system in the problem domain.

The distinction between concept, token, and object in the meaning triangle of Figure 3.6 is also applicable to the realm of systems. In fact, it gives rise to three kinds of systems:

- a *concrete* system: a system in which the elements are concrete things;
- a *conceptual* system: a system in which the elements are concepts; and
- a *formal* system: a system, in which the elements are uninterpreted syntactic symbols (also called symbolic system).

Since a model is also a system of one of these three types, there are three *model types* as well. In fact, a system of type x playing the role of model with respect to some other system, is called a model of type x. Thus, the three model types are called a *concrete model* (also called empirical model), a *conceptual model*, and a *formal model*, respectively. Each model type can play the role of model with respect to the other system types, but also to systems of the same type. This gives rise to nine types of relations between system and model. These are shown in Figure 3.7 as the model triangle (adapted from Dietz (1987)), following the similarity with the meaning triangle. Kramer and

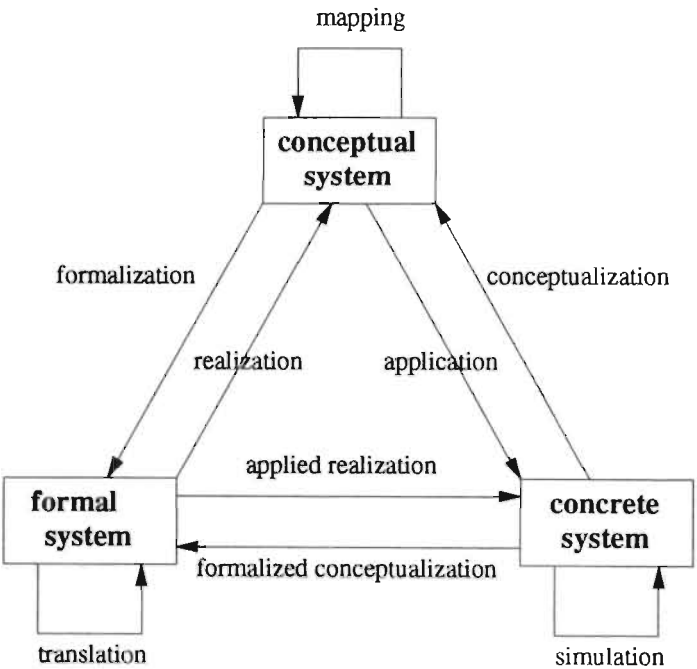


Figure 3.7: The model triangle.

De Smit (1987) provide examples of each type of model. We present these in tabular form in Table 3.1, in which the rows identify the type of system modelled, and the columns the type of the model used. Each entry in the table gives an example of the particular combination of model and system.

An empirical model of a concrete system is called a *simulation*; it is used to obtain knowledge about the dynamical behaviour of the modelled system. An example of such a model is a simulation of a logistics system by a special

	empirical model	*conceptual model*	*formal model*
concrete system	simulation of a logistics system	conceptualization of an information system as a data–flow model	formalized conceptualization
conceptual system	application of a data–flow model in an organization	mapping of a data–flow model onto a data–flow diagram	formalization of mathematical logic
formal system	applied realization	(true) interpretation of a logic formula	translation of Euclidean geometry into algebra

Table 3.1: Model types.

computer system showing the flow of materials between different parts of the logistics system.

An empirical model of a conceptual system is called an *application* of the conceptual system. Sometimes, application is also called *implementation*. An example is the application of a data-flow diagram in an organization, by instructing the persons in the organization to behave according to the relations (i.e., data flows) in the data-flow diagram.

A conceptual model of a concrete system is a *conceptualization* of the concrete system. An example of such a model is a conceptualization of an organization in terms of the data that flows between different elements of the organization. Such a conceptualization can be expressed in the aforementioned data-flow diagram.

A conceptual model of a conceptual system is a *mapping* of the conceptual system. A mapping consists of relating concepts in the system to concepts in the model. An example is the mapping of a data-flow diagram onto a structure chart (which consists of modules calling each other).

A conceptual model of a formal system is called a *realization* of the formal system. In logic a realization is often simply called a model of a formal system. A realization in logic is an *interpretation* of a formal system (a logic formula) which expresses a true fact. An example of such a realization is the interpretation of the logic formula $\forall x \exists y | x < y$ by taking x and y to be natural numbers, and $<$ the binary relation 'smaller than'.

A formal model of a conceptual system is called a *formalization* of the conceptual system. A formalization focusses on the symbolic representation of concepts in uninterpreted symbols. Examples are (1) the formalization of mathematical logic, allowing the inference of true propositions by using formalized reasoning schemes such as modus ponens, and (2) the axiomatic systems of mathematics, in which symbol manipulation is performed without referring to the interpretation of these symbols.

A formal model of a formal system is called a *translation* of the formal system. Examples of such formal models are (1) the translation of Euclidean geometry into algebra and vice versa, (2) the translation of Cartesian coordinates into polar coordinates, and (3) the translation of ASCII symbols into EBCDIC symbols.

The two direct relations between a formal system and a concrete system do not exist directly. They only exist through the intermediate step of a conceptual system. This is reflected in the naming of these relations: one can speak of an applied realization (application and realization) and a formalized conceptualization (formalization and conceptualization). This illustrates the central role conceptual models play within the modelling process. In fact, conceptual models allow us to *describe* and *circumscribe* the empirical reality using concepts. Such conceptualizations can be used for formalizations, but, more importantly, they also allow us to conceptualize reality *as we would like it to be*. This twofold role of conceptual models is used in software development for analysis and design purposes. The result of analysis is a model describing what is the case in the UOD, whereas design focusses on the description of what is intended to be the case in the UOD, i.e., a description of a system of which a part may be an automated system. Next, we look at the importance of linguistic specification in modelling.

3.2.3 Linguistic specification

In the model triangle, a conceptual model refers to a UOD. In order to communicate a conceptual model to another person, one has to express the model linguistically. Such a representation is called a *conceptual-model specification* (or *conceptual-model schema*). The language a modeller uses is generally called a *conceptual-model specification language*. It is important to see that any conceptualization requires some language. Such a language can be formal or informal, graphical or textual, etc. The use of a language in the representation of a conceptual model presupposes some basic set of concepts which can

be expressed in a conceptual model. In order to communicate a conceptual model M to a person p, the model is expressed in a language agreed upon by sender and receiver. The resultant representation R is transferred via some medium to the person p, who interprets the tokens of the language according to the concepts the language is meant to express. Communication succeeds if p is able to interpret R and reconstruct M.

A program in a programming language is nothing else than a representation of a conceptual model, in terms of the concepts on which the programming language is based. However, a program is both a conceptual-model specification and a formal specification. A program in Pascal, e.g., is a conceptual-model specification, since it can be understood by a person familiar with the basic Pascal concepts, such as program, procedure, function, variable, type, statement etc. However, the same Pascal program is also a formal specification of an algorithm. Such a formal specification can be interpreted by a suitable processor, either directly, or with an intermediate *translation* by means of a compiler into a similar formal specification (such as machine code). The process of interpretation of such a formal specification gives rise to a machine, i.e., a concrete system, which plays the role of empirical model with respect to the conceptual model specified by the Pascal program (cf. the model triangle). Thus, with reference to the model triangle, the resulting concrete system is an application of the conceptual model, and an applied realization of the formal-model specification. The machine interpreting the formal specification is, however, not aware of the concepts expressed in the formal specification; it only knows which tokens to recognize, and what to do with these recognized tokens. In fact, conceptual interpretation is the distinguishing feature which human beings do and machines do not possess.

To sum up, any model specification presupposes an interpretation mechanism. Such an interpretation mechanism can either be at a low level, such as a computer interpreting zeros and ones. Or it can be at a high level, such as natural language or diagrams in human communication. The question remains which concepts are useful for human communication, and yet can also be interpreted by a suitable automated interpreter. This is in fact the focus of conceptual modelling in a more narrow sense than the one we sketched here. The next section on conceptual modelling sheds light on this more restricted view of conceptual modelling, focussing on the aspects of *understanding* and *communication* among human beings.

3.3 Conceptual modelling

The development of information systems also deals with automating parts of existing information systems. To this end, every automated information system must contain an 'image' of the UOD it controls. Such an image is a conceptualization of a part of the world. Therefore, the problem of developing information systems can be viewed as a problem of conceptual-model development. Models are developed as part of the two most important phases in the software life cycle, namely *analysis* and *design*. During analysis, the current information system has to be modelled; this is followed by the specification of the requirements of a new information system. During design, a model of the new information system has to be developed.

The complexity of software development, treated in Chapter 2, makes it important to have a clear understanding of both the UOD and the information system to be realized. For this purpose, conceptual modelling can be used to model information systems at a conceptual level, i.e., a level allowing concepts to be expressed. Thus, conceptual modelling addresses two important problems in systems development: (1) it allows to make a description of an existing information system and its UOD in terms of abstract concepts; and (2) it allows to make a description of an automated information system at an abstract conceptual level. Referring to the model triangle, a conceptual model plays a central role in system development. First, a conceptual model of the existing information system and its UOD is the final 'product' of analysis of the UOD (conceptualization). Second, the conceptual model of the new information system is the output of the design process, which can be used for an *application* or *implementation* in the UOD.

The focus of conceptual modelling is *understanding* and *communication*, as Mylopoulos (1992) defines it: *"Conceptual modelling is the activity of formally describing some aspects of the physical and social world around us for purposes of understanding and communication."* The description resulting from conceptual modelling is often called a *conceptual-model schema*. Referring to the model triangle, a conceptual-model schema is a formalization of a conceptual model. Often, the term conceptual model is used to denote the set of language elements a modeller may use when constructing a conceptual-model schema. Using such terminology, one may think of the relation between a conceptual-model schema and the conceptual model in a similar way as the relation between a database schema, describing the schema of a particular

database, and a data model, being the set of language elements to construct database schemas.

Conceptual modelling is the emerging result of streams in artificial intelligence (knowledge representation), databases (semantic data modelling), and software engineering (object-oriented programming languages) (Brodie *et al.*, 1984; Loucopoulos, 1992). We now consider the differences of conceptual modelling with respect to knowledge representation, semantic data models, and programming languages. The introduction of *frames* in the field of artificial intelligence has led to the development of frame-based knowledge-representation languages, such as KL-ONE, KEE, and SRL (Brachman and Schmolze, 1985). Traditionally, knowledge representation focusses on interesting reasoning patterns and their computational representation. This assumes that the knowledge bases resulting from this representation will be used by some computer program to perform an 'intelligent' task. Conceptual modelling, however, focusses on representation for understanding and communication among human beings (Mylopoulos, 1992).

Semantic data models in database theory introduced the concepts of *data independence* and *abstraction forms* (Rolland and Cauvet, 1992). The idea of data independence is that the conceptual schema of a database should be free from the physical structure of the database, allowing a conceptual view of the UOD. Different abstraction forms aid during conceptual modelling. The most important abstraction forms are *classification*, *composition*, *generalization*, and *grouping*. A large variety of semantic data models have been proposed, such as the ER model (Chen, 1976), RM/T (Codd, 1979), TAXIS (Borgida *et al.*, 1984), SDM (Hammer and McLeod, 1981), DAPLEX (Shipman, 1981). For an overview and comparison of the field of semantic data models, we refer to Peckham and Maryanski (1988).

The shortcoming of semantic data models is their orientation towards the structure of passive data: there is no way to represent behaviour of entities in the UOD. In order to address this issue, *object-oriented data models* have been developed, such as GemStone (Maier and Stein, 1987), Orion (Kim *et al.*, 1987; Kim *et al.*, 1989; Banerjee and Kim, 1987), Ontos (Andrews and Harris, 1987), IRIS (Fishman *et al.*, 1987) and O_2 (Bancilhon *et al.*, 1987). For an overview of object-oriented data models we refer to, e.g., ACM (1991). Semantic data modelling shares with conceptual modelling the objective of representing the UOD with appropriate abstraction mechanisms. However, semantic data modelling is by nature oriented towards the static aspects of the

UOD. Furthermore, it introduces assumptions about the way the conceptual schema will eventually be realized in a database. Semantic data modelling is thus more data-oriented, whereas conceptual modelling is concept-oriented.

The evolution of abstraction mechanisms in *programming languages* has resulted in object-oriented programming (see also Section 2.4.1 and Section 3.1). In contrast with semantic data models, object-oriented programming languages focus on the representation of objects with behaviour. The execution of an object-oriented program then consists of a collection of objects invoking each other's operations via messages. The concept of an object has been introduced in Simula (Birtwistle *et al.*, 1977), followed in numerous other languages. There are at least over eighty different object-oriented programming languages, as reported by Saunders (1989). The advantage of object-oriented programming languages is the natural orientation towards the UOD. An object is an abstraction of some entity in the UOD, including its operations. The dichotomy between data structures and algorithms is thus addressed by objects, combining both a data perspective and an operational perspective. The disadvantage of object-oriented programming languages is the inability to model *processes* in an adequate way. In most object-oriented programming languages, the description of a process is 'hidden' in the description of the various behaviours of participating objects. Object-oriented programming languages are oriented towards both representation of a the UOD and a program operating with the abstracted UOD. The representational aspects and the processing aspects are combined in the object, which makes an object-oriented programming language ill-suited for representation and communication of the UOD, since the representation is cluttered with implementation aspects.

3.3.1 Direction of fit

Wieringa (1990) observes that conceptual models can have different roles. He refers to a *direction of fit* between a conceptual model and the UOD it models, also described by Searle and Vanderveken (1985). The *direction of fit* between two entities is an arrow which points to the entity to which the other entity must adjust itself in case there is a mismatch between the two. Three different kind of models resulting from their direction of fit with the UOD are shown in Figure 3.8.

The direction of fit determines the role of the conceptual model. If the model should be adjusted in case there is a mismatch between model and UOD, such a model is called a *descriptive model* (Figure 3.8a). If the UOD should be

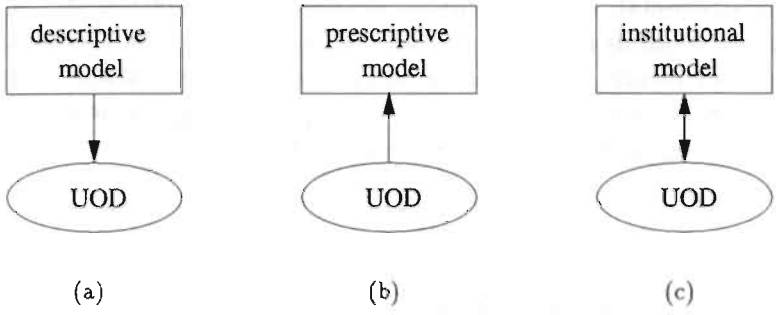


Figure 3.8: Three kinds of models and their direction of fit: (a) descriptive model; (b) prescriptive model; (c) institutional model.

adjusted in case there is a mismatch between model and UOD, the model is called a *prescriptive* or *normative model* (Figure 3.8b). The third kind of model is the *institutional model* which combines both aspects of the descriptive and prescriptive models (Figure 3.8c). The descriptive aspect of an institutional model is that the model itself, by its existence, institutes the basic elements of the world to be modelled. The prescriptive part then assigns the elements in the UOD to the elements instituted by the descriptive part. Thus, elements in the UOD only exist because the model exists, and the elements in the UOD are said to count as particular elements in the model. This aspect of institutional models introduces the concept of *roles* in a UOD. Since an institutional model creates elements just by the fact that it presents them to exist, it has a double direction of fit with the UOD.

As an example of an institutional model, consider modelling a soccer match. In this model, some people have roles as players and goalkeepers, while others play the roles of referee and lines men. The model institutes what should be counted as a goal: the net fact that the ball passes between the two posts will count as a goal. The roles and the fact that a goal has been scored only exist because the institutional model of a soccer match exists. Often in sports, institutional models are sources of conflict, since there has to be an impartial person such as a referee creating the institutional facts: the scoring of a goal depends on the observing capabilities of the referee and will often be questioned by the players.

The direction of fit is not only important in conceptual modelling; it also is

applicable to programs. Conceptual models and programs can contain descriptive, institutional, and prescriptive elements. In a conceptual model, e.g., *types* describe the kinds of entities in the UOD and *laws* can be used to prescribe how the entities should behave. In a computer program, *classes* describe the common structure of objects with *assertions* prescribing particular invariants to be true in every state of the program execution.

3.3.2 Object-oriented modelling

Conceptual modelling is based on the existence of a set of concepts and structures that may be represented in a conceptual model. Different varieties of conceptual modelling arise by taking different sets of concepts and structures. Thus, object-oriented modelling is a specialization of conceptual modelling with concepts of object orientation. Based on the earlier definition of modelling, we define object-oriented modelling in a similar way.

Definition 3.2 Object-oriented modelling *uses the concepts of object orientation to produce an object-oriented model.*

The term object model could be used to denote a model resulting from the object-oriented modelling process. However, we use it here in the sense of the set of basic concepts to model any UOD or application domain.

Definition 3.3 *An* object model *(OM) defines the conceptual entities that can be used for the object-oriented modelling of an application domain or* UOD.

We remark that within a particular modelling environment there is always *one* object model, which is the base on which any model within that particular context is founded. Thus, it is normal to speak about *the* Smalltalk object model or *the* C++ object model. When it is clear from the context which object model we are referring to, we usually refer to it as *the* object model. The object model thus refers to a class of models based on the same concepts. Similar terminology is also used in, e.g., Booch (1991), who presents the framework for object-oriented design as *the object model*. Analogously, *the* relational model in database research stands for the class of all possible logical models of all possible specifications in a relational language. Any modelling activity presupposes some basic set of concepts. This comes readily apparent if we compare the different kinds of programming styles that are used today, together with their basic abstractions: procedure-oriented modelling uses

algorithms, object-oriented modelling uses classes and objects, rule-oriented modelling uses if-then rules etc. In short, we adhere to the common practice of calling the concepts for object-oriented modelling the object model, which will be apparent from its context as well. In case of ambiguity we will make the distinction between object model and model explicit.

The result of modelling a particular domain in an object-oriented way is called an *object-oriented model.*

Definition 3.4 *An* object-oriented model *(*OOM*) is the result of modelling a particular domain using the concepts offered by an object model.*

A final remark should be made on the use of the term *meta model*, which should be distinguished clearly from the term object model. A meta model describes the generic structure of models that can be constructed. So, a meta model is a model for models. A meta model is also expressed in some language, which is not necessarily the same for expressing 'normal' models. An illustration of a meta model is the use of the Backus-Naur formalism to specify the syntax of a programming language. The specification of, e.g., the syntax of Pascal, is the meta model for all syntactically correct Pascal programs. In this example, the meta model is specified in terms of the Backus-Naur formalism, whereas a Pascal program is defined in terms of a program, functions, procedures, and variables. A meta model which is defined in similar terms as the models it refers to introduces the important concept of *reflection*, which we come back to later in this thesis.

3.3.3 Object-oriented modelling and INCA-COM

There is no consensus about a commonly accepted definition of the object-oriented approach. As Wand (1989) premises, much of this confusion surrounds objects because they emerged as programming concepts and, therefore, were driven by implementation considerations. He suggests that the perspective of the object-oriented paradigm should move from implementation-driven to modelling-driven. This is in line with the earlier observations of, e.g., Cox (1987) and Abbott (1987), viz. that there is a continuous evolution in programming, away from computer-oriented principles towards problem- or knowledge-oriented principles. Indeed, the Smalltalk *programming* environment focusses on implementation, and its class hierarchy reflects this by offering a large set of classes for this purpose, such as lists and bitmaps. However, describing a

problem domain using domain concepts is gaining importance over implementation techniques. This requires an adequate set of concepts for modelling complex object domains.

The multitude of object-oriented programming languages shows that there is a great interest in adopting new concepts. The focus of the past years has been directed towards the application of object-oriented concepts during the implementation of software systems. The attraction of this style of working has been an improvement of program code, and consequently, a gain in effectiveness of software maintenance. Using an object-oriented programming language does not imply better designs: it is still possible, when using object-oriented techniques, to make a system which is difficult to maintain. After all, designing is and will remain a creative task, requiring human or artificial intelligence.

The concepts of object orientation are by no means new. They are natural to human thinking as illustrated by the ontologic conceptualization of reality in terms of interacting objects. It is worthwhile to search for means of adopting the object-oriented modelling paradigm at higher levels than solely during implementation, i.e., adopting it during the analysis and design of a system. Efforts in this direction are geared towards the development of object-oriented *analysis* and object-oriented *design*.

Among the current methodologies for object-oriented analysis are Object-Oriented Requirements Specification (Bailin, 1989) and Object-Oriented Analysis (Coad and Yourdon, 1991; Shlaer and Mellor, 1988). Fichman and Kemerer (1992) give a clear exposition on today's methodologies for object-oriented analysis and design. They conclude that the current methodologies for object-oriented analysis are revolutionary compared to the *process-oriented* (i.e., data flow-oriented) structured methodologies, such as Structured Analysis (DeMarco, 1979). Compared to the *data-oriented* methodologies, such as Information Engineering (Martin, 1990) and even JSD (Jackson, 1983), object-oriented analysis methodologies are more akin. From this point of view, object-oriented analysis methods are an evolutionary step.

The purpose of object-oriented design is to define the *architecture* of a software system. It is important during design to use software-engineering concepts, in order to make the resulting design understandable and easy to modify. Encapsulation and modularity introduce two levels of partitioning within object-oriented design: they are based on (1) the object, and (2) groups of related objects. Currently, Object-Oriented Design (Booch, 1991) and Object Modelling

Technique (Rumbaugh *et al.*, 1991) are the most widely used object-oriented design methodologies.

Object-oriented modelling at higher levels The way to object-oriented analysis and design is a bottom-up approach. The first models built with object-oriented concepts were computer programs. Since the object-oriented approach to programming proved to be successful, it was adopted also for design and analysis. We note that a similar bottom-up approach has taken place with structured programming. As soon as structured programming was being used, researchers tried to exploit similar concepts for design and eventually analysis. The 1970s have become known as the structured era, with Dijkstra (1968) (structured programming), Page-Jones (1980) (structured design), and Yourdon and Constantine (1978) (structured analysis) making up the structured tripode.

Analysis and design models should be defined independently of any implementation considerations. For the purposes of *understanding* a domain and *communication* with other people one needs to construct models which are close to a human being's perception of the real world. Such modelling activity is called *conceptual modelling* and the models resulting from conceptual modelling are called *conceptual models*. Within the context of object orientation, a conceptual model is called an *object-oriented conceptual model* (OOCM). The conceptual framework containing the concepts a modeller may use for the construction of a OOCM is called a *conceptual object model* (COM).

Following the same terminology for the object model in the context of object-oriented modelling, the *conceptual* object model is used in the context of object-oriented *conceptual* modelling. Below we give two relevant definitions.

Definition 3.5 *A* conceptual object model *(COM) defines the concepts to be used for the object-oriented conceptual modelling of an application domain or* UOD.

Analogous to an object-oriented model we define the object-oriented conceptual model.

Definition 3.6 *An* object-oriented conceptual model *(OOCM) is the result of modelling a particular domain using the concepts offered by a conceptual object model.*

For the construction of OOCMs, one needs a COM offering the designer the conceptual tools to construct an object-oriented model of an interesting slice

of reality. Such an OOCM can then be used for implementing the system in a programming language. Ultimately, the OOCM can be used as a "program" itself by making the objects in the OOCM alive, thus creating a simulation of the real world.

The complexity of current systems is so high that existing COMs are not adequate in addressing it. Their most serious drawback is their semantical poverty, i.e., the concepts used do not refer to the concepts human beings use. This observation has been made too by Lenat *et al.* (1990) as the major source of *software brittleness*: "Programs often use names for concepts such as predicates, variables, etc., that are meaningful to humans examining the code; however, only a shadow of that rich meaning is accessible to the program itself." The shortcoming of existing COMs is their inadequacy to model the essential meaning of objects, caused by a too small set of descriptive means: existing COMs only offer a modeller attributes and methods to describe objects (Braspenning *et al.*, 1989b). The use of such COMs has serious drawbacks. First, the modeller has to translate the things found in the real world to model concepts (representation). Such a translation often becomes a *transformation*, since existing COMs do not have enough concepts necessary for describing real-world entities. Second, users of such a model have to transform the model concepts back again to their real-world equivalents (interpretation). Both representation and interpretation implicitly suppose a common background shared by modeller and model user. In complex domains this background is complex as well, and leaving it implicit is risky since the modeller and model user may understand the model differently. Such misunderstandings can have great impacts when discovered too late. Boehm (1981) has shown that the costs of corrections in software systems increase significantly as a function of the stage in which the change is made. Object-oriented conceptual modelling addresses the issues of representation and interpretation of a UOD. It is based on object-oriented concepts, the meaning of which can be enhanced by using semantically motivated means for description and by meaningful associations between objects.

3.3.4 Object-oriented conceptual modelling

Having described modelling, object-oriented modelling and conceptual modelling, we conclude this chapter with an integrating view on *object-oriented conceptual modelling*. This will provide the basis for INCA-COM and a validation of using object-oriented concepts in INCA-COM.

We stress that INCA-COM will offer concepts for object-oriented models during analysis and design. It is frequently advocated that the object-oriented paradigm is the solution to the software crisis through its unified use of objects during the entire software life cycle. However, analysis and design are different stages and should not be intermingled (although iteration between them is useful). Analysis strives to a clear view of *what* is currently the case in the UOD. Using the analysis, a decision has to be made regarding the part of the UOD to be automated. This results in a requirements specification of a system to be developed. Based on the requirements specification, a system must be designed, describing the system in a machine and language-independent way. Finally, the design should be implemented and tested. (See also Figure 2.3 for the steps in the software life cycle.)

So, the two stages analysis and design are intimately bound, but the analysis describes the UOD and the design describes the system. Both analysis and design add knowledge to the models, but their nature differs: analysis produces domain knowledge about the UOD, whereas design results in application knowledge. In our opinion, the distinction between domain knowledge and application knowledge should be reflected in INCA-COM. The role of INCA-COM with respect to the different roles of OOCMs is shown in Figure 3.9. The phases of the software life cycle are shown in bold type face (see also Figure 2.3). Every life-cycle activity results in a particular model, such as an analysis model, a design model, and a computer program. The analysis results in a description of the current real and information system, whereas the computer program model resulting from the implementation describes the automated information system, which is a part of the larger information system. A completed design model can be used for *simulation* purposes to check the validity of the automated information system.

Why should one adopt object-oriented concepts for conceptual modelling? By answering this question we give insight into the usefulness of the object-oriented approach for conceptual modelling in software-system development. There are three main reasons to use object-oriented concepts during conceptual modelling. Each will be treated below. For similar reasons see also, e.g., Fiadeiro *et al.* (1992).

Natural view of the world

The development of automated information systems deals with automating parts of existing information systems. Before automating, a clear description

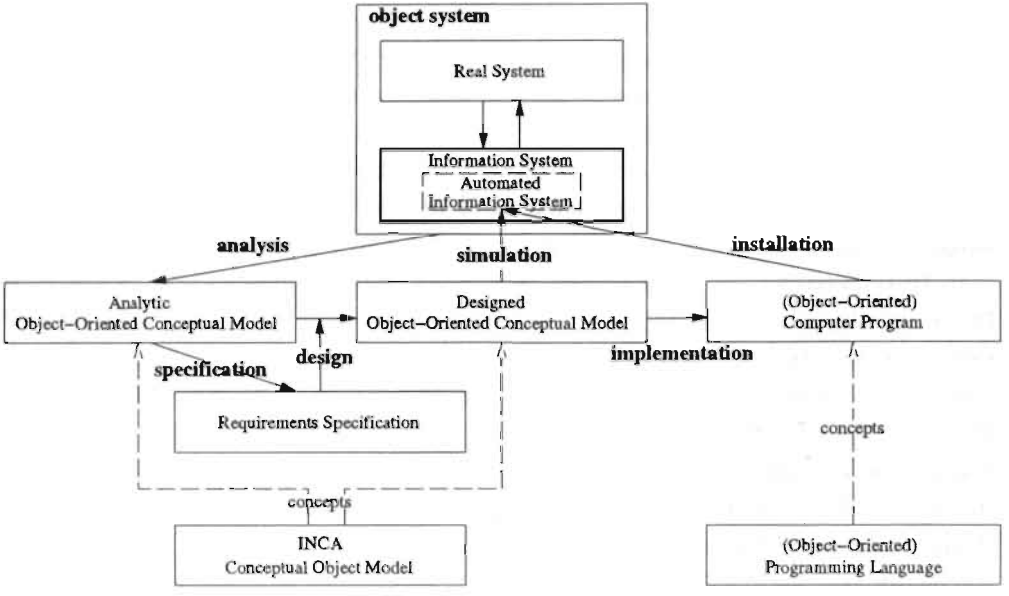


Figure 3.9: INCA-COM and object-oriented conceptual models.

of what is currently the case in the UOD should be available. This comes down to analyzing an organization in order to discover its structure. Objects provide a suitable means to describe a UOD in terms of objects and their interactions. Similar to things in the real world, some objects may be active, while others are passive, i.e., undergoing actions of active objects. An analysis based on objects gives insight into the current state of affairs in a UOD. Subsequently, some objects may be selected for being automated, while others are not. This selection is the essential activity of a requirements specification.

Combined modelling of static and dynamic aspects

Early semantic data models focussed on modelling the static part of a UOD. Application programs were still written in a procedural language, leading to applications consisting of a static part (the database) and a dynamic part (the program). An object combines static and dynamic aspects by encapsulating both state and operations. It must also be possible to distinguish between active and passive objects.

Unit for modularity and reuse

An analytic OOCM is used as the basis for the development of an automated information system. Such an information system can be seen as a collection of interacting objects. The development process starts with the OOCM, by automating particular objects. An object provides a unit for modularizing the resulting information system and dividing the development process. Furthermore, since objects are abstractions of particular entities in the UOD, it is likely that such automated objects may be reused in different information systems. Of course, implementation reuse is not new, since software (function) libraries provide similar facilities. However, reuse based on objects with state and operations, provides a better unit for reuse.

Finally, one question remains to be answered: which concepts are useful in object-oriented modelling?

3.4 Object-oriented concepts

This section concludes the chapter by introducing concepts which a COM should be able to represent. The concepts are based on ontological principles. In ontology there is a distinction between (1) things and (2) generic concepts (classes). Things are entities in the UOD having an identity and a description. We call the representation of a thing an object. Classes are abstractions of concrete things. For the semantics of a UOD, the structure of a UOD should be expressed in the structure of the resulting OOCM. Therefore, objects and classes can be related to each other in meaningful ways. Three of such semantics enhancing relations are classification, generalization, and composition; their dual relations are instantiation, specialization, and decomposition (see also, e.g., Kristensen and Østerbye (1991)). Below we treat the concept *thing* and the three relations mentioned above, together with their duals. In Chapter 4 they are used for the concepts in INCA-COM.

3.4.1 Thing

In Bunge's ontology the concept of "thing" is introduced as the basic element the world is composed of (Bunge, 1977). A thing is a combination of *substance* and *form*, which will be introduced below.

Substance The basic principle of Bunge's ontology is the existence of substantial individuals (also called *entities*). A substantial individual is the result of stripping a thing of its properties. Thus, a substantial individual is a *bare individual.*

Form Substance expresses the fact that there exist individuals, which we can study apart from their form. However, this is a simplification. In reality, substance is bound to form: there are no bare individuals (formless substance) except in our imagination, nor are there pure forms apart from matter. Bunge (1977) makes a distinction between different types of properties. Substantial properties are features of substantial individuals, and conceptual properties are features of conceptual individuals (conceptual properties are also called attributes). Because a model of substantial individuals is built with concepts, it contains attributes, representing substantial properties. It is important to note, that the main purpose of Bunge's ontology is *de*scriptive modelling, i.e., modelling the existing world through description. The distinction between properties and attributes is based on descriptive modelling: an (existing) object *possesses* properties, which are captured within a descriptive model through attributes. Some properties are represented in the model, while others are not.

Thing A *thing* is a substantial individual endowed with all its properties. Thus, the concepts of substance and form are combined in the concept of a thing: a thing is something that exists (substance) and about which something can be said (form). A substantial individual possesses properties, which become known to us via attributes.

3.4.2 Classification and instantiation

Classification and instantiation are relations between generic concepts and individuals. Classification consists of assigning an individual to a concept. The concept to which an individual belongs is also called its class. The inverse relation of classification is instantiation. It consists of the formation of an individual based on a concept.

According to Wieringa (1990), the goal of classification is to formulate laws that obtain in the UOD. That is, the classes in a classification should be such that all and only objects of a class obey a law. Laws may be empirical (humans have an age), analytical (the age of a human is a positive natural number),

normative (each person should have a name and address), or institutional (persons older than 18 are voters).

Although both classification and instantiation relate concepts and concrete things, their application domain is quite different. Classification is important in *conceptual* models, in which the description of a particular domain is essential. Classification is thus a means of identifying objects. To identify an object with respect to a classification is to determine, given a few properties of the object, of which class(es) it is a member (Wieringa, 1990, p. 68). This kind of classification is important in qualitative sciences, such as biology and chemistry. It represents knowledge about the classified objects, which can be used to infer new knowledge about the objects.

Instantiation is used in *programs* to create new instances, based on the intension of the class. Once the common underlying structure of a set of objects has been determined (i.e., finding the generic concept and forming a class), the class can be used as a 'factory' creating distinct objects, which share the common structure. The objects belonging to a class are called the *instances* of the class. The term instantiation is used primarily in programming languages, which are implementation languages. We remark that the term class is used in these languages both to designate a concept and to act as a run-time implementation mechanism.

3.4.3 Generalization and specialization

The concept of a hierarchy plays an important role in handling the various abstractions distinguished in a particular domain. A hierarchy is used to *order* different abstractions. A hierarchical organization is a simplification of a particular domain, that helps human beings to organize it. The two most important appearances of hierarchy are the generalization/specialization hierarchy, and the part/whole hierarchy.

Whereas classification organizes a UOD with respect to the difference between concrete things and concepts, generalization and specialization order different concepts. Generalization and specialization are relations between generic concepts. The resulting network of relations is called a taxonomy, since it imposes an order on the different concepts.

A taxonomy is the result of the classical way of specifying the *essence* of objects in a class (the intension of the class). This method is used, e.g., in biology and is based on the method of *genus* and *difference*. A class is specified by giving its genus and its specific differences. As Wieringa (1990, p. 80) describes

it: "(1) The *genus* is that part of essence which is predicable also of other things differing from it in kind; (2) The *specific differences* are those parts of the essence which distinguish it from other subclasses of the same genus. These subclasses are called *species*." The definition by genus and difference is by nature hierarchical. For this reason, taxonomies of classes are also called *class hierarchies*. Each definition of a class in a taxonomy defines a general concept, which is the intension of the defined class. Since the definitions form a hierarchy by the method of genus and difference, the concepts form a hierarchy as well. And since the concepts are class intensions, the corresponding class extensions are also hierarchically ordered.

The definition method of genus and difference makes the species more specific than its genus: the definition of a species consists of its genus, and any additional specific characteristics. Since the intension of the species is more specific, it is predicable of a smaller number of things. The intension of the genus is less specific and thus predicable of more things than the intension(s) of its species. As a result, the extension of a genus is larger than the extension of its species. The hierarchical definition of genus and difference thus gives rise to a subset relation on the extensions of the classes. This subset relation is contrary to the intension of the classes defined. A class G generalizes a class S if G's intension is less specific than S's intension, and G's extension includes S's extension. G is called a generalization of S, and S is called a specialization of G.

3.4.4 Composition and decomposition

Besides the hierarchical ordering of classes by means of generalization and specialization, a second form of hierarchy is the use of composite objects. A composite object is an object which has other objects as its *parts*. Whereas generalization and specialization structure a domain by relating different *concepts*, composition and decomposition order object instances by relating them in terms of *wholes* and *parts*.

There are several applications of composition and decomposition. The well-known methodology of *stepwise refinement* is based on the algorithmic decomposition of programs into smaller program units. This process leads to an organization of a program as a composition of functions and procedures at different *levels of abstraction*. Structured programming is based on this methodology.

A level concept is a natural way of structuring a domain. Human beings

employ levels of abstraction when considering, e.g., a book as a hierarchy of chapters, sections, subsections, paragraphs, subparagraphs, sentences, words, and characters. This example illustrates that for each decomposition, their is some basic level, serving as the *atomic* level of decomposition. The level of characters, e.g., is the atomic level for decomposition of books.

Frequently, composition as a structuring relation between part objects and composite objects is not supported explicitly by object-oriented languages. The usual way of constructing a composite object is using the values of attributes to refer to the parts of the composite object. The disadvantage of this lack of support is that the semantics of composition are not supported by the programming language, but must be guarded by the programmer. For example, the deletion of a composite object should ideally result in the deletion of its part objects. Normally, it is not sufficient to delete only the references to part objects because then these may continue to exist. For effective reuse, a programming language should support composite objects by offering special facilities for describing compositional structures.

This chapter has set the stage for the discussion of INCA-COM. We have investigated several object-oriented programming languages, and extracted the concepts found in such languages. Object-oriented programming is defined as a specialization of object-oriented modelling, which is a form of conceptual modelling. For use in INCA-COM we have looked at the importance of object-oriented concepts in high-level conceptual modelling. The three main reasons to adopt an object-oriented approach in high-level conceptual modelling are: (1) object-oriented concepts allow a natural view of the world; (2) objects provide a unified concept for modelling both static and dynamic aspects of the world; and (3) objects are an adequate unit for modularity, enhancing their reuse. The next chapter defines the concepts of INCA-COM.

Chapter 4

The INCA Conceptual Object Model

> If you tell the truth, you don't have to remember anything.
> *Mark Twain*

In this chapter we resume the description of INCA-COM outlined at the end of Chapter 2. In Chapter 3 we discussed the necessary concepts of object orientation for INCA-COM. The description of INCA-COM is started with an introduction of the basic concepts of object, description, and sort. It is followed by an elaboration of the conceptual framework of Figure 2.6. Next, we describe the four different kinds of models of objects and systems. The chapter is concluded with a comparison of INCA-COM and related work.

4.1 Basic concepts

Before treating the four different models used in INCA-COM, we turn to the basic concepts, i.e., the concepts underlying the models. These are essentially the following three concepts: object, description, and sort.

4.1.1 Object

An object is defined as an individual concept having an own identity. The form of an object is captured in its descriptions. Thus, a (conceptual) object reflects Bunge's ontological concept of thing (see Section 3.4). Additionally, an object can be distinguished from other objects by means of its name , i.e, the

object concept includes that each object has an identifier. The object name is unique so that the object may be referred to by its name. This is captured in the following definition.

Definition 4.1 *An* object *is an individual concept with an own identity. An object has a set of descriptions. Every object has a unique name. An object o is formalized as a tuple* $(n, D(n))$*, in which n is the object name, and* $D(n)$ *the set of descriptions of n.*

Objects are introduced into INCA-COM's object world by descriptions, annotating the form of a (conceptual) object. An object's identity is independent of its descriptions, meaning that we make an analogous distinction as the substance/form distinction in, e.g., Bunge's ontology. At this point a remark about the existence of objects is appropriate. INCA-COM focusses on the construction of a conceptual object world. Thus, the objects being modelled only 'exist' in a conceptual way (insofar it is legitimate to speak of conceptual existence). Such objects can be made to exist in reality by giving the object descriptions to a machine with the ability to interpret them, and create real objects accordingly. Of course, such objects are not real in a substantial way, but in an 'informational way', meaning that they exist within a computer's memory.

4.1.2 Description

The result of our modelling activity is a set of descriptions of an object. A description is a partial annotation of the form of an object, and generally there will be several descriptions of an object.

Definition 4.2 *A* description *is the result of a modelling activity, by which the form of an object is annotated. A description consists of a descriptor and a value for the descriptor.*

A description thus is a (descriptor, value) pair. When dealing with an object and its descriptions, we must have some means to refer to them. For this purpose, the object name is introduced (see above). Additionally, we need a way to refer to a description of an object.

Definition 4.3 *A* descriptor *designates a description.*

Thus, a descriptor is an element of a description, and can be used for the identification of the description. In order to enhance the semantics of a description, a descriptor belongs to a *descriptor type*. Such descriptor types allow a modeller to classify the descriptions of objects in different semantic categories. The different types represent the distinctions between features and associations, and essential and role-dependent features (e.g., property, attribute, relation, and link). A descriptor often can have some value from a well-defined *value domain*. Combining descriptors and values into descriptions yields a description which takes the form of a ⟨descriptor, value⟩ pair. In such a description both descriptor and value are typed. A descriptor is sometimes also referred to as an *object describing entity*.

The modelling concepts of INCA-COM are illustrated with examples of object descriptions, given in the format shown in Figure 4.1. The object exampleOb-

```
Object exampleObject {
  <description>
  <description>
  ...
}

where <description> has the form:

<descriptor_type> :: <descriptor> : <value> :: <value_type>
```

Figure 4.1: The format of object descriptions.

ject is the start of the object specification, which leads to the creation of an undescribed identifier exampleObject. The following descriptions annotate the form of exampleObject. The set of descriptions can be as large as necessary for a sufficient annotation of the object structure, indicated by the dots (···). Each description has a form as given in the lower part of Figure 4.1. The two elements of a description, the descriptor and the value, are at the center of the description, and are flanked by their respective *type* specifications. The form of descriptions is only a notational convention. The important point is that the combination of descriptors and values yields a description. Figure 4.2

shows the graphical way of showing an object.

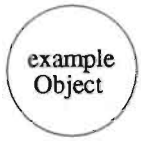


Figure 4.2: Graphical notation of an object.

4.1.3 Sort

The division between things and concepts is represented in INCA-COM by the introduction of *sorts*. Every object in INCA-COM belongs to a sort. Although the relation between a sort and its instances partly is based on set-theoretic notions, we stress that our notion of a sort goes beyond that of a set. A set is nothing more than a collection of things, i.e., a set only has an extension. A sort in INCA-COM has both an intension and an extension. By assigning an object to a sort, the place of an object in the object world is determined. The sort assignment enables the transfer of generic descriptions from a sort to its instances (i.e., the objects belonging to the sort).

Definition 4.4 *A sort represents a concept which can be instantiated. A sort acts as a semantical primitive, and allows its instances to receive an initial set of descriptions.*

Sorts themselves can be considered as useful individuals. Therefore, we have decided to give sorts the status of an object. In INCA-COM we discern two aspects of a sort, namely its intension and its extension, and thus we have an *intensional description* and an *extensional description*. A sort's intensional description annotates the role the sort has as an object, whereas its extensional description annotates the description of its instances. Figure 4.3 shows the graphical notation we use for sorts, together with one of its instances. The similarity of the round form of both sort and instance stresses the object status both have, whereas the difference in line width indicates the difference between a sort and a simple object. We remark that an object's name gives a clue whether the object is a sort: a name beginning with a capital letter names a sort, and a name not beginning with a capital letter names a simple object. An object name may be composed of several words. For readibility the

constituent words of the name are capitalized. Thus, in Figure 4.3 ExampleSort is a sort, and exampleObject is a simple object.

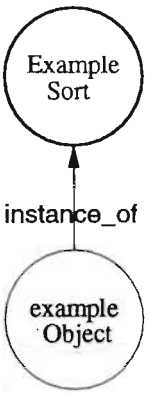


Figure 4.3: Graphical notation of a sort.

An elaborate discussion on sorts, specification, and sort hierarchies follows in Chapter 5.

4.2 The framework

The purpose of INCA-COM is to extend the object-oriented modelling concepts in such a way that (1) the semantics of world objects can be captured adequately, (2) application domains can be modelled, and (3) systems based on an application domain can be described. Figure 4.4 shows the structure of the conceptual framework of Chapter 2 in more detail. The core of the framework contains world objects. These are the basis for modelling particular application domains. Applications are based on these application domains.

In the following three subsections we elaborate on the different parts of INCA-COM. These are modelling of the *world*, of *application domains*, and of *applications*.

4.2.1 World

An ontological perspective sheds light on what there is (things). However, in order to study things, they have to be represented in a way enabling us to perform a discourse and to formalize them. For this reason Bunge (1977)

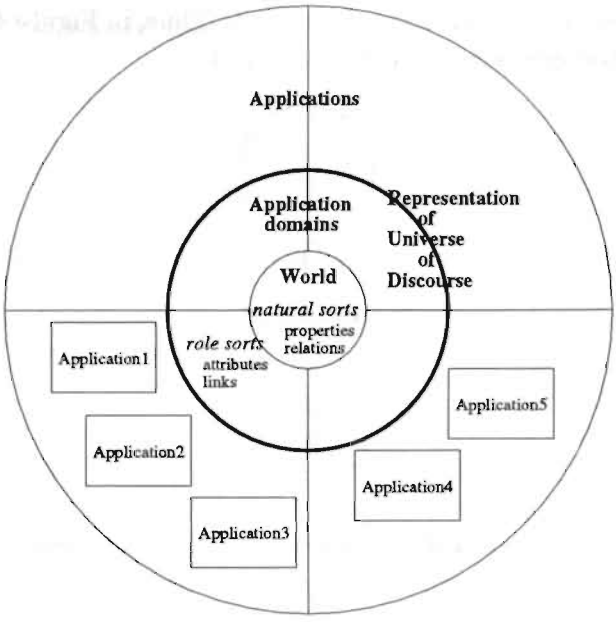


Figure 4.4: The structure of the conceptual framework.

rightfully makes a distinction between on the one hand things and their properties, and on the other hand their respective representations as model things and attributes. This distinction is so essential that a picture may illustrate it.

The left-hand side of Figure 4.5 shows a thing in a UOD. Ontology studies what kinds of things there are in a UOD and how these may be conceptualized and formalized. A conceptualization of the thing in the UOD is shown on the right-hand side of Figure 4.5. The distinction between UOD and conceptualization introduces a *conceptual world*, consisting of model things and their attributes. Such a conceptual world is the focus of INCA-COM. Since it is based on the concepts of object orientation, we call our conceptual world an *object world*.

An object world consists of a core of world objects. They are modelled with the descriptor type *property*. The objects represent things that occur naturally in the world, such as human beings, trees, cars, etc. The core of the object world is shown in the centre of Figure 4.4.

The meaning of objects and the structure of an object world is enhanced by using associations between objects. Associations between objects in the real

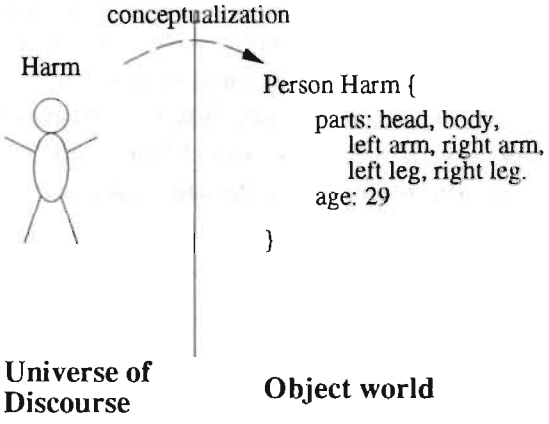


Figure 4.5: The distinction between UOD and object world.

world are represented by the descriptor type *relation*. Similar to properties, relations model *essential* associations between objects. An example is a human being having some parent association with another human being. Since the parent association is an essential one, without which something cannot possibly be a human being, it is modelled as a relation.

The special relation instance_of associates an object with its sort. Relating an object with its sort determines the 'place' of the object in the object world. A sort can have relations with other objects (or sorts). The relation subsort_of determines the location of the sort with respect to other sorts. In order to express that sorts in the world denote concepts in the real world, we call them *natural sorts*. Objects which are instances of natural sorts are called *natural objects*.

4.2.2 Application domains

An application domain contains objects and sorts that are only interesting in a particular domain. In addition to properties, an object may have *contingent* features, which are only of interest within a particular application domain. Contingent features are called *attributes*, since they are attributed to the object. An example is a human being in an application domain of banking who has an attribute named creditworthy. Structuring an application domain is done by using *links*, which are contingent associations. Natural objects may receive attributes when they begin to play a *role* within an application do-

main. During their life time, objects may start or cease to play various roles. A natural object may play different roles within an application domain, or different roles within different application domains. When an object ceases to play a role, the role object disappears, but the natural object temporarily playing that role persists. If a natural object 'dies', all its roles also cease to exist. The sorts in an application domain are called *application sorts* or *role sorts*. The representation of a UOD consists of the combination of a (slice of the) world and an application domain.

4.2.3 Applications

For modelling applications, the information paradigm (see Figure 2.1) serves as the conceptual model for an information system. An information system consists of (1) active objects performing discourse about a UOD, and (2) a representation of the UOD supporting the discourse of active objects. The representation of a UOD has been described above; it consists of a (slice of the) world and an application domain. For the discourse of active objects we adopt *speech act theory* (Searle, 1969; Searle and Vanderveken, 1985). This enables us to model the communication between active objects in a way close to human communication. Thus, an information system contains a representation of a UOD in terms of *passive objects*, about which *active objects* perform a discourse. This approach is similar to, e.g., Dietz (1992b) who proposes a model for information systems analysis consisting of communicating *actors*, based on speech act theory.

The division between active objects and passive objects seems counterintuitive to the general idea of object orientation that state and behaviour should be combined. Thus, one might argue that the division between active and passive objects is similar to the classical program/data division: active objects are the sources of activity, while passive objects are recipients of the actions of active objects. However, we believe that such a division is necessary for the following reason. An information system contains a *representation* of a UOD. Although the represented things can be active (in their own 'real world'), their *representation* is under the control of active objects within the information system. The division between active objects and passive objects is not a step back to a procedural way of modelling. After all, both active and passive objects can be described in an *object-oriented* way, using the dichotomies of classification/instantiation, generalization/specialization, and composition/decomposition introduced in Section 3.4.

The difference between essential features (properties and relations) and contingent role features (attributes and links) is used as follows. Active objects acting on passive objects may only change attributes and links within the application domain under consideration. Properties and relations used in an application domain are *read-only* features. The latter does not mean that a property or relation has a constant value. Its value can change under influence of natural processes, e.g., age. It just cannot be changed by an application. Thus, the structure of the framework implies that there is a core of essential features, called properties and relations, and several application domains introducing attributes and links.

Supporting the division of an information system into a passive representation part and an active processing part, we introduce four different INCA models. The representation part is based on passive objects, modelled in an information model and an event model. The processing part is based on active objects, modelled in a behavioural model and a communication model.

4.3 INCA models

It is interesting to see that both an object-oriented design method, such as OMT (Rumbaugh *et al.*, 1991), and an information-systems oriented method, such as NIAM-ISDM (Nijssen, 1989) incorporate analogous models for information systems. OMT and NIAM-ISDM divide (a model of) an information system into three basic models: (1) an *object* or *information* model describing the structure of information; (2) a *functional* or *process* model describing what processes act on the information; and (3) a *dynamic* or *impulse* model describing when the functions/processes are executed. Rolland and Cauvet (1992) add to these models a *behaviour* or *life-cycle model*, accounting for the definition and ordering of events which may occur during the life cycle of objects.

In summary, the four perspectives in modelling information systems are: (1) the *static* perspective: what is the structure of information and what relationships are there? (2) the *life cycle* perspective: which events may occur to objects and what is their order? (3) the *process* perspective: what is the structure of processes and how are objects used in processes? and (4) the *impulse* perspective: when are the processes activated? In INCA-COM, we introduce four different models to capture a UOD from these perspectives.

1. The *information model* describes the structure of the information and

the relations that are valid between various elements of the descriptions. In addition, the information model offers a role concept, which allows for the description of roles within different application domains. The different aspects of object descriptions are semantically categorized by introducing different *types* of object-describing entities. The description of an object is the basis for defining different *states* in which an object can be. The information model is discussed in Section 4.4.

2. The *event model* describes which events can take place in the object world, and which order is imposed on these events. An event is defined as a state transition of an object (the cause of which is not relevant in the event model). The event model describes the interface of passive objects, since it models the possible events an object may suffer. Thus, the event model provides a shopping list view of events which might happen in the object world, similar to the ideas expressed by Meyer (1988). The event model is discussed in Section 4.5.

3. The *behavioural model* describes which objects are active (i.e., perform actions) by displaying particular behaviour. The behaviour of such active objects consists of particular actions, which give rise to events in the object world. The relation between behavioural model and event model is thus a correspondence between a particular action and a possible event. The goal of an active object is to accomplish the actions it has committed to. Following Dietz (1992b), the commitments constitute the *agenda* of an active object. The behavioural model is discussed in Section 4.6.

4. The *communication model* describes what interactions take place between active objects. Communication between active objects is based on speech act theory, similar to the application of speech acts as given in Dietz (1992b). The communication model is discussed in Section 4.7.

4.4 Information model

The information model is used to describe the structure of information and the relations between different objects. This is done by combining descriptors and values, thus forming descriptions. INCA-COM offers the following six types of descriptors:

1. properties,

2. attributes,
3. relations,
4. links,
5. displays, and
6. versions.

In the following subsections we treat the different types of descriptors and their usage in modelling. For clarity's sake, we treat properties and attributes, and relations and links in one subsection each. After the sections on displays and version, we close with a section on state and state space.

4.4.1 Properties and attributes

To describe a feature of an object, INCA-COM introduces two descriptor types, namely *properties* and *attributes*. Although these two entities are frequently used in other COMs to denote a feature of an object, we make a clear distinction between the two types. An analogous distinction as our property/ attribute distinction, is the distinction between *essential* and *non-essential* (or *contingent*) features of objects (Wieringa, 1990). By saying that an object has *essential* features, we mean that the object will have those features in every possible state or application, i.e, it is inherent to the object's being. By saying that an object has *contingent* features, we mean that the object has features, which it may fail to have in some other state or application.

A property represents a feature of a real-world thing. It is a descriptor which apart from the object does not possess any descriptive value. It *belongs* to the object and describes nothing in isolation. The real-world thing's identity implies that it has the property. This gives an important clue to decide if something should be modelled as a property: if a feature is essential to a thing's identity, it is modelled as a property. In the framework of Figure 4.4 properties are descriptors for objects in the 'world core', and are thus independent of any application domain.

Definition 4.5 *A* property *is an essential feature of an object, meaning that the object has the property in every possible application.*

As an example, in the domain of university, we might model researchers, who are writing scientific articles to be sent to conferences. A property of a scientific

article is its title: we cannot see a title separate from an article object, i.e., every article has a title. Thus, the description of an article object takes the form as shown in Figure 4.6. The description follows the format of Figure 4.1. In Figure 4.6 the object myArticle is described by the property (denoted by P)

```
Object myArticle {
  P :: title : "Object-oriented modelling" :: String
}
```

Figure 4.6: Description of an article object.

title. The value of this descriptor is Object-oriented modelling, which is of type String.

In contrast with a property, an *attribute* is a feature of an object, when the object is playing a certain role within the UOD. In the framework of Figure 4.4, attributes are located with role objects in application domains. Thus, an object playing a role within an application domain may acquire attributes, which are application-dependent features. Hence, attributes are *institutional* features, since roles and role sorts are *application-dependent* or *institutional* sorts.

Definition 4.6 *An* attribute *is a contingent feature of an object when it plays a certain role.*

In the university domain, an article may be sent to a conference, by which the article starts its role as a (possible) conference article. At the moment it starts playing this role, it may have attributes (denoted by A), such as its valuation, meaning a referee's valuation of the article (Figure 4.7).

Based on the distinction between natural and contingent features, Wieringa (1990) introduces (1) natural kinds as the natural classes, which have essential features, and (2) role kinds as contingent classes, which have contingent features. Analogously, the sorts in the core of the object world are called *natural sorts*, and sorts in application domains are called *role sorts*.

Since an object can exist without playing a particular role, there must be some means by which objects come to play roles. We model this by including

```
Object myArticle {
  P :: title : "Object-oriented modelling" :: String
  A :: valuation : 8 :: Integer
}
```

Figure 4.7: Attribute of an article.

a description specifying the role sort to be a role of a natural sort. During operation of the information system, it means that the role sort can be instantiated and that the resulting role instance r can be associated with an object o playing that particular role. Thus, the object myArticle may play a role as a ConferenceArticle by associating myArticle with an instance of ConferenceArticle, e.g., conferenceArticle01. In Section 4.4.2 the role association between these two objects is illustrated, and Chapter 5 contains a more elaborate example in Section 5.4.

The process of role association may be repeated for each role an object plays within different application domains. In this way, an object may play several roles by instantiating different role sorts, and associating each role instance to the object. In fact, the use of roles and role sorts introduces the concept of *institutional kinds*, which can be instantiated by particular, authorized objects.

4.4.2 Relations and links

Associations between objects are of great importance for a good structuring of a world and for giving coherence to an object model. In our view they should *not* be expressed by means of properties or attributes. That is, an association between two or more objects should be modelled by a separate descriptive entity. INCA-COM offers two modelling entities for this purpose: *relations* and *links*. The distinction between a relation and a link is based on analogous principles as the distinction between a property and an attribute. A relation is used to describe an *essential* association between things, whereas a link describes an *application-dependent* association. Again, this distinction is made for semantical reasons. We believe that this distinction adds more knowledge to models. Woods (1975) criticized the use of one kind of *links* for

various and quite different purposes, e.g., for knowledge representation and for implementation. In our opinion, distinguishing two types of associations makes clear to a modeller that there are different *kinds* of associations.

Definition 4.7 *A* relation *is an essential association between objects, without which the description of the objects is not complete.*

An example of a relation in the university domain is the association between an article and its author. The association is essential, since any article has an author. Hence the association between an article and its author is modelled as a relation (denoted by R in Figure 4.8). Another example of a relation is the

```
Object myArticle {
  P :: title : "Object-oriented modelling" :: String
  A :: valuation : 8 :: Integer
  R :: author : harmBakker :: HumanBeing
}
```

Figure 4.8: Relation of an article.

instance_of association between an instance and its sort. It is also a relation, since an instance depends on the association to its sort. The instance_of relation is so important, that we assign it a special graphical notation, consisting of an *arrow* from an instance to its sort. Henceforth, we will suppress its name in figures.

Definition 4.8 *A* link *is a non-essential association between two objects, when the objects are playing a certain role. The objects may exist without having the role association.*

In the university example, an article may play a role as a conference article. Within the conference domain, during the refereeing process the conference article may be assigned to a referee. This can be modelled by a relation between the conference article and a referee. However, when viewed from the article playing the role as conference article, the association between conference article and referee should be seen as a link, since it is an association of the

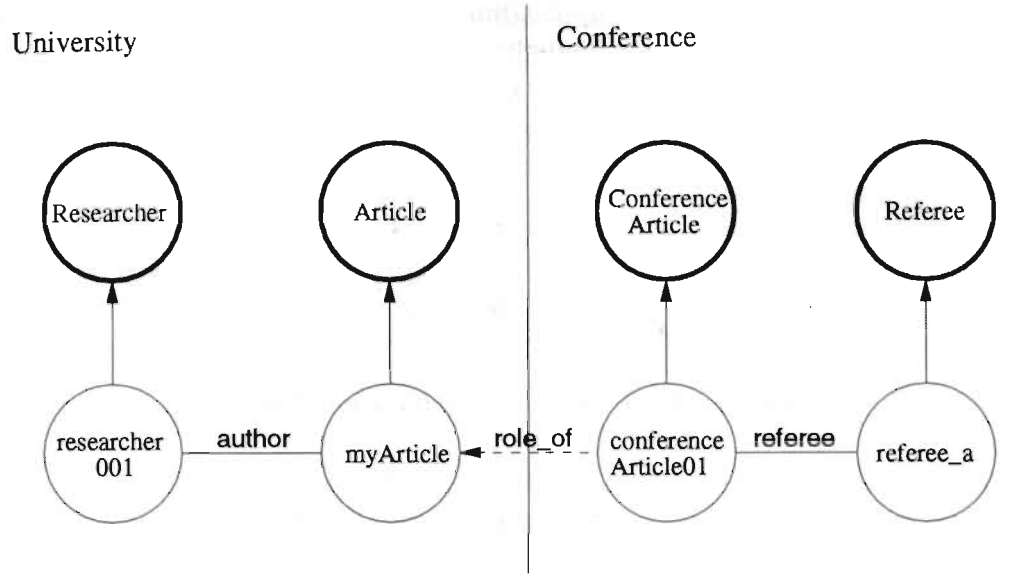


Figure 4.9: University domain.

article *when it is playing a particular role.* Thus, the article in its role as a conference article has a *link* to a referee. This example illustrates that the distinction between the descriptor types relation and link depends on the application domain, i.e., the viewpoint of the modeller. The examples given so far are shown graphically in Figure 4.9.

There are two domains: the university domain, with the sorts Article and Researcher, and the conference domain, with the sorts Referee and ConferenceArticle. The university domain contains two instances of Article and Researcher, called myArticle and researcher001, respectively. By means of the author relation myArticle is related to researcher001. The conference domain contains two instances of Referee and Article, viz. referee_a and conferenceArticle01, which are related to each other by the referee relation. The instance conferenceArticle01 is a role of myArticle.

Once again, as remarked earlier, the application domain in question determines what should be characterized as essential or as role dependent. The structure of the modelling framework introduces a *nesting* of application domains, as shown in Figure 4.10. Each application domain introduces its domain-specific sorts, which are meaningful in the particular domain. Other application do-

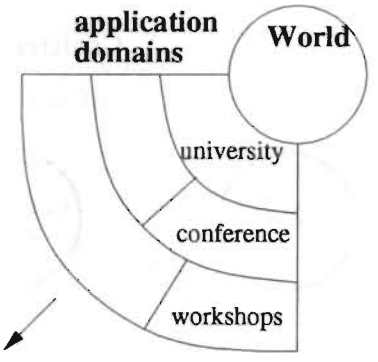


Figure 4.10: Nesting of application domains.

mains may be based on earlier-defined domains. For example, the conference domain is based on the university domain. The meaning of such a nesting is that each application domain introduces objects which may be a role of an object in a nested application domain. The properties and relations introduced in the inner application domain are also meaningful in the outer application domain, with the restriction, that the outer application domain may *use* the inner properties and relations without actually changing their values. Thus, the descriptors defined in inner application domains become a kind of *read-only* descriptors in outer application domains.

As an example, the article myArticle in the university domain has a title. The conference article conferenceArticle01, which is a role of myArticle will have the property title of myArticle. Its value may be used within the conference domain, but it may only be changed in the university domain, e.g., by its author researcher001.

The instantiation of application-domain sorts may introduce roles for objects in the inner application domains. However, such role linking is not obliged. In the conference example, a conference article may be instantiated without a role link to an article. For such unlinked instances it may be necessary to duplicate particular inner-domain descriptors, since they may be needed in the outer domain. As a case in point, consider the creation of a conference article c, without associating c as a role to an article instance a. As a result of this, the conference article c will not have the property title from article instance a. Thus, the modeller may be required to specify a title for c, since a conference article should have a title.

We remark that the distinctions we make between properties/attributes, and relations/links depend on the chosen domain. When viewed from the university domain, myArticle has a *link* with a referee, whereas conferenceArticle01 has a *relation* with a referee. The different views express that within the conference domain the article-referee association is essential, whereas in the university domain this association is role dependent. Additionally, it may even be the case that an article from a university domain plays several roles as a conference article, although conference organizers often only accept original articles for presentation at their conference.

4.4.3 Displays

The descriptor type of *displays* specifies how an object will manifest itself to the outside world. Although a display is usually associated with a mere visual style of presentation, we stress that a display is used to describe how the object exhibits itself via one or another medium (such as sound, vision, or otherwise). One could specify, e.g., that a C program modelled as an object displays itself visually as a program structure diagram, or that an error condition displays itself as a bell on a terminal. In Chapter 7 we shall address displays of objects more extensively.

An object can have several displays, depending on the roles it plays in different application domains. In one application domain, a person may be presented by means of a bitmapped photograph, whereas in another domain it may happen by means of its name.

4.4.4 Versions

Versions of an object are also descriptors, analogous to the descriptor types discussed above. With a version we mean a variation on a theme (in this case the theme being the object). Semantically, such a variation differs not enough from the object to be modelled as a new object. Nevertheless, a version may, purely syntactically speaking, differ considerably from the original object.

For instance, consider the modelling of program sources as objects. One such object might be a program for sorting numbers according to the quicksort algorithm. Within the world of source objects there could exist many versions of quicksort in different programming languages, which from a semantical point of view all refer to the same object.

We remark that every object in INCA-COM can have versions. We note, however, that a version in INCA is *not* an object, although in special cases versions can be modelled as so-called *version objects*. For a more profound analysis of versions, we refer to Chapter 6.

4.4.5 State and state space

In order to introduce a concept of event as a transition from one state to another, it is necessary to have a concept for the *state* of an object. The descriptors introduced in the information model provide the basis for a conceptualization of the state and the state space of an object.

In ontology, it is hypothesized that at a given time every object is in some state. In order to illustrate the concept of *state*, we follow Bunge (1977). The description of an object o is a functional schema, which we call a *description schema*, i.e., the set of descriptions of an object is a n-component function $\mathcal{D}$, which is defined as $\mathcal{D} = \{D_i | 1 \leq i \leq n\}$. Each $D_i(o)$ is a descriptor of object o. The domain of the function D_i is the extension of the sort S to which o belongs and the range is VD_i, the value domain of descriptor D_i. Each component $D_i : S \rightarrow VD_i$ of $\mathcal{D}$ is called a *description* of the sort S to which o belongs. The description schema $\mathcal{D}$ is called the *set of descriptions* for o, and its value $\mathcal{D}_o = \langle D_1, \ldots, D_n \rangle(o) = \langle D_1(o), \ldots, D_n(o) \rangle$ for $o \in \Omega$ represents the state of o. Ω is the object world, i.e., the population of objects at a particular time.

The various descriptions of an object are in general not independent. Often descriptors and their values are somehow related to each other by constraining the values each descriptor can have. Any such constraint on the values of descriptors is called a *law statement*, for it describes in which lawful states an object can be. As an example of such a law statement in the university domain, consider the statement that a researcher should have a salary which is not below a particular minimum.

The description schema as a representation of an object's state gives rise to a *state space* for objects. If we take the Cartesian product of the ranges of the components of $\mathcal{D}$ the range VD of $\mathcal{D}$ itself is formed. The range VD is called the *conceivable state space* $S(o)$ for the object represented. However, not every value in the range VD is permitted, as expressed by the law statements of the object. Thus, the states in which an object may be, is a constrained subset of the conceivable state space $S(o)$, which is called the *lawful state space* $S_L(o)$. Figure 4.11 shows the state space of a university employee o consisting of two

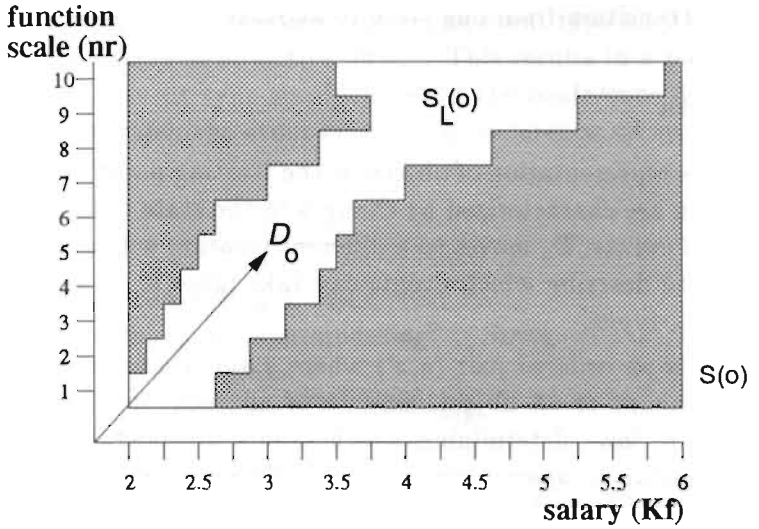


Figure 4.11: The conceivable state space $S(o)$ and the lawful state space $S_L(o)$ of an object o.

descriptors, salary and function scale. The rectangle consisting of the white area and the shaded areas, is the conceivable state space $S(o)$ of employees in scales 1 up to and including 10. However, by the constraints the university imposes on the minimum and maximum salary for each function scale, only combinations of function/salary in the white area are permitted. The white area thus depicts the lawful state space $S_L(o)$ of an object o, which is a subset of the Cartesian product of the ranges of its descriptor function. The state of object o is represented by the value of the total descriptor function $\mathcal{D}_o$, shown as the tip of the arrow in Figure 4.11. It points at a lawful combination of salary and function scale (salary: 3Kf, scale: 5). For ease of presentation, we have shown a total descriptor function with only two components. We remark that in general the total descriptor function can contain any natural number n of descriptors, giving rise to an n-dimensional conceivable and lawful state space. For the specification of laws we refer to Section 4.5.3.

4.5 Event model

Assuming that every object is in some state, implies that objects can be in different states at different time instants. Otherwise, an object world would not change, and thus be uninteresting. In the event model, *events* are introduced

as an object's transition from one state to another.

4.5.1 Events

The state-space representation of objects is the starting point for the definition of events, which are characterized by changes in the state of an object. Under the influence of events, $\mathcal{D}_o$ moves to a different location within its lawful state space. Below we describe which events can take place in an object world by describing event types.

An event type is an ordered pair $\langle s, s' \rangle$ where s and s' are states of an object o. The event space of an object consists of all such ordered pairs, but in general there are laws determining which events are lawful and which not. Consider, for instance, again a university employee o with state $\mathcal{D}_o$ equal to (salary: 3Kf, scale: 5). All ordered pairs $\langle \mathcal{D}_o, \mathcal{D}'_o \rangle$, where $\mathcal{D}'_o$ is a lawful state, can be considered as a conceivable event. However, by constraints from the university, the event ⟨(salary: 3Kf, scale: 5), (salary: 2Kf, scale: 1)⟩ is not a lawful event. Thus, similar to the lawful state space, there exists a subset of the event space with events which are permitted by the laws for the domain under consideration. The event space of permitted events is called the *lawful event space.*

For the modelling of a UOD it is not sufficient to treat events as mere state transitions. The reason is that the representational purpose of objects requires that state transitions of things in the UOD are represented within the information system's representation of the UOD. This, in turn, requires the state transitions of objects to be invoked by *active* objects. Thus, in INCA-COM each event type is given a name by which it can be invoked. Naturally, each event type represents a meaningful event type in the UOD. The modelling of event types results in a set of applicable operations, which transfer the object's state from one lawful state to another. The specification of an event type includes four aspects:

1. the name;
2. names and types/sorts of the parameters (also called the event signature);
3. preconditions, stating the requirements for the event's applicability; and
4. postconditions, describing the state of the object after the event.

During execution of an information system, an event type may be instantiated as a result of an action of an active object. This results in a particular event. An example of an event type specification in the conference domain is given in Figure 4.12, in which the event types for a conference article are described.

```
Object conferenceArticle01 {
  P :: title : "Object-oriented modelling" :: String
  A :: valuation : 8 :: Integer
  R :: author : HarmBakker :: HumanBeing
events:
  create
  admit
  assign-to-referee (r : Referee)
  referee
  reject
  accept
  assign-to-session (s : Session)
}
```

Figure 4.12: Events for a conference article.

The specification of events in Figure 4.12 does not include the specification of the preconditions and postconditions. In order to be able to specify these, we need a law concept. The concept of a law is important in the specification of (1) lawful states and (2) lawful events. In the next subsection we examine law statements.

4.5.2 Law statements

Law statements pose constraints on an object's set of possible states and events. In order to represent law statements, we first consider their nature. Investigation shows that three distinct divisions of laws are possible, depending on the viewpoint.

1. As described above, there is a division of laws into *static laws* and *dynamic laws*. A static law constrains the conceivable state space of an object to its lawful state space. Similarly, a dynamic law constrains the set of possible state transitions. Additionally, laws may put contraints on the *existence* of objects without necessarily referring to their state.

2. Orthogonal to the division of laws into static, dynamic, and existence laws, laws can either be *local* or *global*. A local law constrains the possible states and state transitions of a single object, whereas a global law constrains the possible states of two or more objects.

3. A third division of laws is suggested, among others, by Wieringa (1990), in which three kinds of laws are distinguished: (1) analytical laws, which hold because of the propositional content of the law; (2) empirical laws, which hold because of empirical evidence; and (3) deontic laws, which express obligations or permissions, that can be violated by things in the UOD.

   An example of an analytic law, is the law statement 'age is a positive number'. This law holds since the concept of age includes that its value is positive. An empirical law is 'the age of a human being is less than 200', which holds since no human being lives more than 200 years. It might some day be invalidated by a person living longer than 200 years, in which case such a law should be redefined. A deontic law constrains the state space of objects, by imposing institutional constraints. Deontic laws can be, and frequently are, violated by the objects to which they apply. As an example, consider the deontic law in a library stating that 'Every book should be returned within four weeks'. Such a law statement is frequently violated, and it is the task of some agent in the library system to take corrective action in order to ensure that the book is returned (by, e.g., sending a person a reminder).

The three viewpoints on laws give rise to 18 different types of laws. Table 4.1 shows the different kinds of laws, with an example of each kind. We remark that analytical existence laws (both global and local) cannot be formulated, because of their nature.

4.5.3 Modelling of laws

The various kinds of laws described in the previous section can all be modelled in INCA-COM. Below we describe the modelling process.

		analytical	*empirical*	*deontic*
local	*static*	age is a positive number	human age is less than 200	the balance of a bank account should not be negative
	dynamic	age can only increase	every human dies	a library book should be returned within four weeks
	existence	–	there is an object of type Manager	there should be an object of type Manager
global	*static*	the age of a father is larger than the age of his children	a father is at least 10 years older than his children	an employee's salary should never exceed that of its manager
	dynamic	if a football team loses, its opponent wins	if a football team loses, its manager is fired	no more than \$10,000 may be transferred from a savings account to a current account
	existence	–	the number of employees is larger than 10	the number of employees should be smaller than 100

Table 4.1: Eighteen kinds of laws.

Analytical and empirical laws are so-called *necessary truths* (Wieringa, 1990). By definition, analytical laws cannot be violated by things in the UOD. They can be used to verify if an object's state represents the state of a possible thing in the UOD. An object with, e.g., a negative value for the age descriptor does not represent a possible state of a thing in the UOD. Empirical laws must be formulated in such a (weak) way, that they also cannot be violated by things in the UOD. For instance, the empirical law stating that a human being's age has an upper limit of 200 years is formulated weakly enough to be a necessary truth.

Necessary truths on the local level (cf. Table 4.1) are modelled in the following way.

A static law constrains the value of a descriptor to some well-defined subset of values. This is modelled by specifying an appropriate value domain, or by describing a *descriptor value constraint*. For instance, age can be represented by a natural number (value domain), and human age can be represented by a subrange of natural numbers with an upper bound of 200.

A dynamic law imposes appropriate conditions on an event. This is modelled by event pre- and postconditions, formulated in terms of an equation between the descriptor values before and after the event under consideration. For instance, a birthday event in a human being's life cycle should only be allowed to increase age. This can be modelled with an event postcondition expressing that $age' = age + 1$. We use the prime character "'" to denote the value of a descriptor after the application of an event. age' is the value of the age descriptor after a birthday event.

An existence law is a constraint on a sort's extension. For the specification of existence constraints we introduce the symbol Ext referring to the extension of sort. For instance, to express that there is an object of sort Manager, the description includes an existence constraint such as $Ext(Manager) > 0$.

Necessary truths on the global level are modelled in the following way.

A static law is a global descriptor-value constraint, relating two descriptors of different objects. For instance, stating that a mother's age is larger than the age of her children is expressed by a global descriptor value constraint, specifying the age descriptor of the mother to be at least as large as that of her children.

A dynamic law is a common event. This means that two (or more) objects are subjected to one event. For instance, if a football team loses, its opponent wins.

An existence law is a constraint on a sort's extension. This is similar to a local existence law (see above).

Deontic laws express obligations or permissions of things in the UOD. They are useful in the specification of institutional elements of object-oriented conceptual models (see Section 3.3). Deontic laws are modelled in the same way as analytical and empirical laws, with the additional specification that it is a law that may be violated by things in the UOD. The difference between on the one hand analytical and empirical laws, and on the other hand deontic laws is that all objects obey analytical and empirical laws, whereas there may be objects that do not obey deontic laws.

Every deontic law requires that an active object guards its violation and takes corrective actions to suppress the violation. Examples of such active objects

are (1) a librarian sending a reminder to a borrower who has forgotten to return a book within due time; and (2) a police officer fining someone for speeding. Therefore, each deontic constraint should be checked, either continuously or periodically.

Table 4.2 gives an overview of the way different kinds of laws are being modelled.

		analytical	*empirical*	*deontic*
local	*static*	value domain or descriptor constraint	value domain or descriptor constraint	deontic constraint
	dynamic	event post–condition	event post–condition	deontic constraint
	existence	–	sort extension constraint	deontic constraint
global	*static*	global descriptor constraint	global descriptor constraint	deontic constraint
	dynamic	common event	common event	deontic constraint
	existence	–	sort extension constraint	deontic constraint

Table 4.2: Modelling different kinds of laws.

The *lawful event space* enables us to describe a *life cycle* of objects. The concept of a life cycle combines the concepts of state and event; it is described by an object's states and transitions between these states. A life cycle can, e.g., be described as a finite state machine. It is preferred to show the life cycle of an object in a graphical specification. Figure 4.13 is a specification of a paper at a conference with states (nodes) representing lawful states and state transitions (arrows) representing lawful events.

In a textual specification, the life cycle is expressed as a regular expression. The life cycle of a conference paper is specified as shown in Figure 4.14.

In information-systems development there exists an inherent dichotomy between things in the UOD and their representation as objects in the information system. Since our conceptualization is only an abstraction of the UOD, not ev-

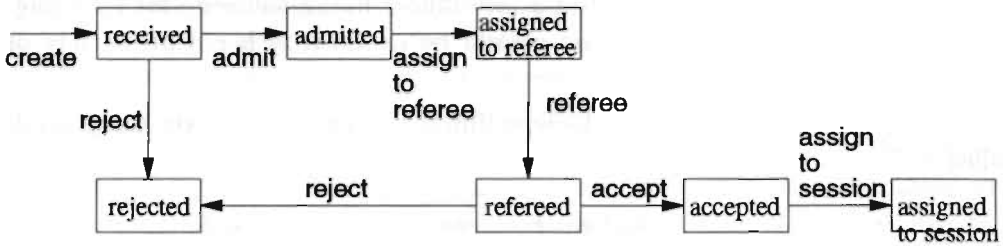


Figure 4.13: Life cycle of a conference paper specified by a state transition diagram.

```
Object conferenceArticle01 {
  events:
    create
    admit
    assign-to-referee (r : Referee)
    referee
    reject
    accept
    assign-to-session (s : Session)
  life cycle:
    create . (reject +
              admit . assign-to-referee . referee . (reject +
                                                     accept . assign-to-session))
}
```

Figure 4.14: Life cycle for a conference paper specified as a regular expression.

ery (type of) event in the UOD is representable in the object world. Obviously, as explained in Chapter 3, modelling implies leaving out unimportant detail. Representable events *can* always be represented in the information system. However, this requires that some active object makes the event known inside the information system.

Consider for instance returning a book to a library. This event will only be represented if one asks a receptionist to accept the book, and wait for the receptionist to acknowledge the return. The receptionist in the library is an active object, making the return of a book known inside the information system. Thus, the event of a book return to a library will always be the result of some *action* taken by some active object. Furthermore, the return of a book is only acceptable if the book has been lent out earlier (i.e., the state of the (representation of the) book is on loan).

Disregarding the difference between active objects and passive objects, leads to a collapsed object world in which the control is spread among both active and passive objects. Sometimes, additional coordinator models are introduced to alleviate the problems in modelling UODs with collapsed object worlds (Van de Weg and Engmann, 1991). In our opinion, the division between active objects and passive objects has been neglected so far; yet it seems one of the important issues to be addressed in object-oriented modelling.

In the event model we have introduced the concept of an event. The state transition of an object from state s_i to state s_j is an event. An object representing a thing in a UOD is *passive*, undergoing events. A remaining issue is the cause of events, since objects representing things in the UOD cannot change states spontaneously. An event is caused by an action of an active object. The nature of active objects and their actions is treated in the next section.

4.6 Behavioural model

The INCA modelling framework introduces a division between objects representing things in a UOD, and applications using objects. This is also to be seen as a division between passive objects and active applications, as illustrated in Figure 4.15. Objects in the world and in application domains represent things in the UOD, and applications represent the active objects within an information system, creating and manipulating objects. Figure 4.15 shows two different applications, application1 and application2.

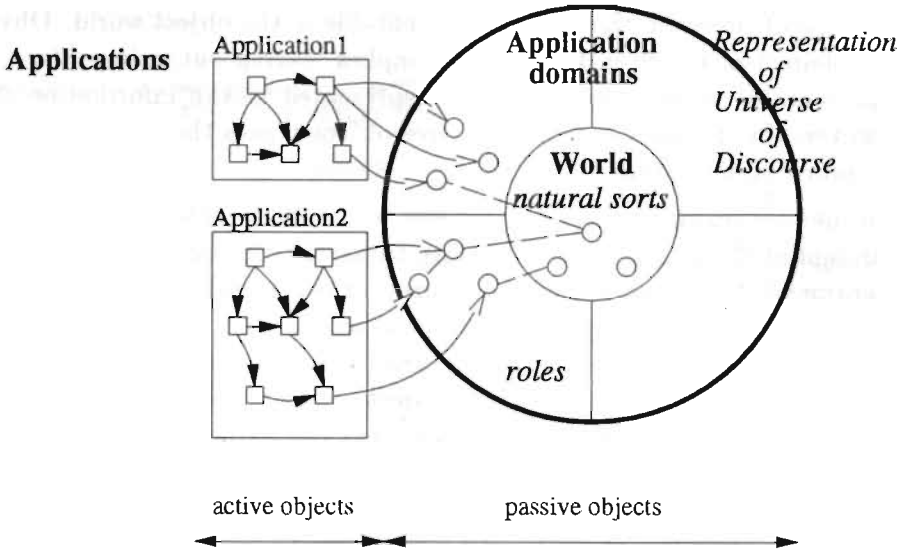


Figure 4.15: The division between passive and active objects.

4.6.1 Behaviour

An application consists of active objects performing actions on passive objects. Following Dietz (1992b), a sequence of actions is called the *behaviour* of an active-object system. In addition to the actions active objects perform on passive objects, they influence each other by means of communication, to be discussed in Section 4.7. In Figure 4.15 the actions of active objects on passive objects are indicated as open arrows. The communication between active objects is shown as solid arrows. The dashed lines between objects in the object world indicate that certain objects are playing different roles.

The division between active and passive objects is reflected in the division between actions and events. Events are suffered by passive objects, and are caused by actions of active objects. The relation between a process, i.e., a sequence of events, suffered by passive objects and the actions performed by active objects is as follows. A process in an object world is caused by a sequence of actions, which is called a *behaviour* of an active-object system. So, every process in a passive-object world is the result of a behaviour in a system of active objects. The unit for defining behaviour is an action.

4.6.2 Actions

A potential action of an active object is described by an entity called an *action*. Each action of an object results in a state change of a passive object or an active object. Therefore, each action corresponds to an event either defined for a passive object or an active object. Describing for an active object which actions it may perform, offers two ways for checking consistency between the event model and the behavioural model. First, any action performed by an active object should be a possible (lawful) event in the event model. Thus, when the actions of an active object have been specified, a modeller or automated tool may check for the occurrence of every possible action in the event model. Second, every event in the event model should be an action of some active object. Since an object world may be used by different applications, each event in the event model corresponds to some action in the collection of applications for the object world under consideration. This allows objects to be modelled with a shopping list of possible events in the various applications in which an object may be used, as suggested by Meyer (1988). This is reflected in the structure of the framework, showing different applications based on the same representation of a UOD. The assignment of actions is also a way of *authorizing* active objects to perform particular actions.

4.6.3 The state of active objects

The state of active objects consists of the actions they still have to perform. Such an action-to-be-performed is called an *agendum*. An agendum usually includes a time interval during which the required action has to performed. The collection of these agendums is called the *agenda* of an active object, in concordance with the normal meaning of the word agenda. The agenda of an active object can be seen as the state of the object. During operation of the information system, each active object tries to perform the actions on its agenda. In order to perform these actions, the object may need information about the state of other passive objects, or other active objects. Thus, both object world and active objects are a source of information for the performance of actions.

The information an active object needs for executing its actions can be divided into two different categories. First, an active object may need information about the state of objects in the object world. For instance, in order to execute the action accept for accepting a conference paper, a conference organizer

needs information from the referees, represented in the object world as referee-reports. Second, an active object may need information about the state of other active objects. For instance, a conference organizer may need to know the number of papers a referee still has to judge, before asking the referee to judge an additional paper. As a final result of executing an action, the active object can do two things. First, it creates the state representing the action it has performed. This results in a state transition in the object world. Second, it directs another active object to perform one of its actions. An active object may also direct itself to perform some action.

As an example, we give the specification of the action admit-paper of a conference-organizer. The action in Figure 4.16 is executed whenever a paper is admitted to the refereeing process. The reception of a paper is taken as a directive to accepting the paper for the conference. The specification of Fig-

```
admit-paper (p: Conference-paper) {
  IF p.date-of-reception > submission_deadline THEN
    p.reject
  ELSE IF p.no_words > upper_limit OR
    p.no_words < lower_limit THEN
    p.reject
  ELSE
    STATE p.admit
    <take care that p is assigned to a referee>
}
```

Figure 4.16: The action admit-paper.

ure 4.16 is given below. The name of the action is admit-paper, and it takes a parameter p of the sort Conference-paper, which stands for the paper to be admitted to the refereeing process of the conference. The specification of the action is done in a procedural language, containing the normal control structures, such as if-then-else and while statements (not shown). The admittance procedure consists of checking two constraints: (1) the date of reception of p should not be later than the submission deadline, and (2) the number of words of p should be in the interval (lower_limit, upper_limit). If these contraints

are met p is admitted to the conference. This is specified by the line STATE p.admit, causing the event admit suffered by p. After p has been admitted, it should be assigned to a referee. This is expressed as a comment; in the next section we come back to the issue of communication between active objects.

By describing the actions of an object and relating the active object to the information it needs, one effectively constructs a process model of a system, similar to data-flow diagrams used in Structured Analysis. The information an active object needs is similar to the use of data flows in a DFD. In addition, the action specification contains a specification of the state change it has performed (p.admit), by which the connection between behavioural model and event model is established.

4.7 Communication model

The communication model specifies which active objects communicate with each other, and what the object of their communication is. The communication model closely follows the application of *speech act theory* (Searle, 1969; Searle and Vanderveken, 1985) as presented in Dietz (1992b).

4.7.1 Communication

Actions specified in the behavioural model are the actions of active objects. For such an action to be represented within the object world, the result has to be *communicated* to some other active object. The latter one checks whether the result of the action is in concordance with the expected result. In the conference example, a researcher can perform an action, such as submitting an article to the conference. For such an action to be successful, first the user must send the article to the conference organizer, and second the conference organizer has to admit the paper to the refereeing process. If the latter is not done, the article formally has not been submitted.

Thus, actions performed by active objects must be communicated before they can be represented *as a fact* in the object world. Of course, an active object can itself be the target for communication, but such self-communication is in general not interesting to be modelled (if an object has different roles, these might very well communicate; such communication takes place between the role objects).

4.7.2 Speech acts

Speech act theory is a theory of communication between people, stating that communication is not only a means for spreading information, but also a way of acting (Searle, 1969; Searle and Vanderveken, 1985). The minimal units of human communication are speech acts of a type called *illocutionary acts*. In speech act theory any utterance of a sentence has a *propositional content* and an *illocutionary force*. The propositional content is the proposition being uttered, and the illocutionary force is the intention the speaker has in uttering the proposition.

As an example, consider the proposition 'Peter is present'. Different kinds of illocutionary forces can be attached to this proposition. The utterance can be, e.g., a statement (Peter is present), a question (Is Peter present?), a command (Peter must be present), a promise (Peter will be present) or an apology (I'm sorry that Peter is present). As the examples show, illocutionary forces are expressed in natural language by means of syntactic features, such as word order, stress, and intonation. However, not only natural languages implement speech acts. Any language used for communication serves as a means for performing speech acts, and so it can be applied usefully in communication within information systems, including automated parts of such information systems.

An illocutionary force in general consists of seven components, which we will not treat here. The most important for our purposes is the *illocutionary point*, which expresses the goal the speaker has in uttering a proposition. The following five categories of illocutionary points are recognized by Searle and Vanderveken (1985).

1. *Assertive or descriptive point*
   An illocutionary force has the assertive or descriptive point if the proposition expressed is presented as representing an actual state in the UOD. Examples of such *assertive* verbs are: *inform*, *state*, *deny*, and *respond*.

2. *Commissive point*
   An illocutionary force has the commissive point if a speaker commits himself to execute the future course of action specified by the proposition. Examples of such *commissive* verbs are: *commit*, *promise*, *guarantee*.

3. *Directive point*
   An illocutionary force has the directive point if the speaker tries to get

the hearer to carry out the future course of action specified by the proposition. Examples of such *directive* verbs are: *order*, *request*, *suggest*, and *pray*.

4. *Declarative point*
   An illocutionary force has the declarative point if the speaker brings about the state of affairs represented by the proposition by his utterance. Examples of such *declarative* verbs are: *declare*, *approve*, *excommunicate*, and *bless*.

5. *Expressive point*
   An illocutionary force has the expressive point if the speaker expresses his feelings about the state of affairs represented by the proposition. Examples of such *expressive* verbs are: *thank*, *regret*, and *apologize*.

Communication is a form of acting by which information is transferred from one active object to another. In this context, Dietz (1992b) proposes a model, DEMO (Dynamical Essential MOdelling), for communication between active objects based on speech acts. In DEMO, active objects perform so-called *essential actions*. These are actions which result in a change in the UOD. However, the result of such an action has to be represented in the representation of the UOD, so that it may be shared with other objects. Hence, communication is used as the means by which the results of actions are communicated, and by which active objects influence each other by proposing one another to perform essential actions.

4.7.3 Conversations

The communication between two active objects takes place by means of *conversations*. A conversation is a combination of two different types of speech acts. There are three different types of conversation, listed below.

1. directive conversation, by which an active object asks another to perform an action; directive conversations consist of a directive and a commissive illocutionary force;

2. statutive conversation, by which an active object informs another object of a state fact in the UOD; statutive conversations consist of a assertive/ declarative and an expressive illocutionary force;

3. informative conversation, by which an object asks information about the state of another (active or passive) object; an informative conversation is similar to a *query*.

The first type of conversation is the *directive* conversation during which an active object S_1 directs another active object S_2 to execute some action. Subject S_2 may respond to S_1 that it will execute the proposed action, which results in the creation of a *commitment* of S_2 to execute the action within some future time range. These commitments are called *agenda*, and they consist of the proposed action, together with the time interval within which the action is to be performed. Directive conversations are based on the *directive* and *commissive* illocutionary forces. An example of directive conversation between two active objects S_1 and S_2 is:

S_1 : Please referee my paper for your conference.
S_2 : Ok, we will referee your paper within four weeks.

S_1 directs S_2 to referee his paper for acceptance at a conference. S_2 responds to this directive by committing to do so within a period of four weeks. After this conversation has finished, S_2's agenda will contain the particular refereeing action with the particular period of time.

The second type of conversation is the *statutive* conversation, which is focussed on the establishment of a state change in the UOD. An active object proposes another active object to accept a state change. The result of such a state change is called a *state fact*. A statutive speech act is either an assertive speech act, i.e., describing an actual state in the UOD, or a declarative speech act, i.e., creating an actual state in the UOD by means of declaring it as being the case. Statutive conversations are based on the *assertive/declarative* and *expressive* illocutionary forces. An example of such a statutive conversation between two active objects S_2 and S_1 is:

S_2 : Your paper has been accepted at our conference.
S_1 : That's great.

S_2 states (to S_1) that his paper has been accepted for the conference to which S_1 sent his article. Prior to this notification, S_2 has *decided* to accept the article, thus creating the state of affairs being expressed in the statutive speech act. Author S_1 is happy about this state of affairs, which is shown in the final speech act of the above conversation. The difference we make here with respect

to Dietz (1992b) is the following. In Dietz' view, the fact that the paper has been accepted is only created after S_1 accepts the fact as being the case. Dietz therefore introduces a type of illocutionary force called the acceptive illocutionary force, which is not mentioned originally by Searle. In the above example, it is unnatural to assume that the paper is only accepted after S_1 has accepted it to be the case. After all, S_2's decision to accept the paper already creates the corresponding fact, which was the objective of S_1. Thus, in our opinion S_2's decision creates a state, of which S_1 is informed.

In Chapter 5, Section 5.4 the concepts of communication by means of speech-act conversation will be illustrated by an example of a conference-registration system.

History and future

As shown earlier, (passive) objects are at a certain time in a particular state. This also holds for active objects, although the elements of their state is different from that of passive objects. The state of a passive object is the result of state-changing operations performed on it. These operations are executed by active objects. The collection of facts, i.e., the state of all objects in the representation of a UOD is a factual representation of the *history*, i.e., a representation of all facts which hold in the UOD. The state of an active object consists of the agenda it still has to perform. These are the results of *directive* conversations with other active objects. The collection of agenda is a list of the states of all active objects. They represent the operations that have to be performed, accounting for the *future*.

The use of speech acts for modelling conversations between active objects introduces an *object interface* for communication. In the behavioural model, active objects are described by the actions they may perform, e.g., which permissions they have. The object interface specifies how an object will respond to communications from other objects, and which agenda will be created. The influence of active objects among each other reflects the purpose of the information system.

The conceptual division of an information system into an object world and active objects communicating about the object world gives rise to two different system concepts. First, each represented object is a system, consisting of descriptions and laws/constraints between these descriptions. We call such a system a *descriptor system*. An object world thus consists of a set of descriptor

systems. Second, each application is a system of active objects which influence each other by means of communicative actions. We call such a system an *active-object system.*

Since the concept of a *system* plays an important role in INCA-COM, we will treat composite objects and systems as one of the important structuring principles (Chapter 5).

Automation and implementation

The division into active objects performing speech acts on passive objects is helpful in automating (part of) an information system. A modeller may automate certain active objects, by deciding which speech acts may be performed by automated objects. Automation consists of granting automated objects the possibility to perform certain speech acts. Such a decision has to followed by a realization of the automated objects by means of an implementation. An implementation generally consists of a description of automated objects, to be executed by a suitable machine or interpreter. The conversations between human active objects and automated active objects can be used then as a model of user-interface dialogues, which can be realized in various ways.

4.8 Comparison with other approaches

The models in INCA-COM are inspired by and partly based on the following sources:

- ontological concepts from philosophy of science by Bunge (1977) and Bunge (1979) on objects and systems;
- the thesis by Wieringa (1990) on the formal modelling of dynamic objects, including the concepts of natural kinds, roles, and different roles of conceptual models;
- the article by Wand (1989) on the formalization of an object model, especially the concept of communication by means of laws;
- the work by Dietz (1992a) on the distinction between active objects and passive objects;

- the work by Van de Weg and Engmann (1991) on the life cycle of objects, and the development of an object-oriented method for analysis and design.

Bunge (1977, 1979) furnishes an interesting ontology of objects (things) and systems. He also makes a distinction between properties and attributes, but puts it at another conceptual location than INCA-COM. In Bunge's ontology a natural object has properties, which become known to us as attributes in a model. So properties may or may not be represented in a model as attributes, and even if they are represented, it may be in an inaccurate way. We use the distinction between properties and attributes to distinguish between essential features and contingent (role) features. In addition, Bunge (1979) gives a clean concept for systems. This ontologic notion of a system is, however, focussed on natural systems, such as societies, families, molecules, etc. The concept for information systems is not extensively addressed.

Dietz (1992b) uses Bunge's system model in his methodology *Dynamical Essential MOdelling* (DEMO) in a subject-oriented model for information systems. The communication model in INCA-COM is for a large part based on DEMO. However, the information-oriented part of DEMO is not object-oriented, but based on NIAM's fact-oriented approach and predicate calculus.

Other approaches are, e.g., Semantical Object Model (SOM) (Velho and Carapuça, 1992), which focusses on semantical modelling of objects. It does not include a concept for systems. It also uses a role (phase) concept. Wand (1989) uses Bunge's (1977) ontological object model for the formalization of objects. However, this formal model also lacks a system concept. Wieringa(1990, 1991) describes a method for algebraic specifications of dynamic conceptual models. His model includes a role concept, but these roles are to be subclasses of the class of objects playing the role. So, an object playing several roles in parallel must necessarily belong to several subclasses (one for each role) in the same hierarchy. Van de Weg and Engmann (1991) do not differentiate between active objects and passive objects. The result is a mix of active and passive descriptions of objects. An object paper in their model contains behaviours such as accept, which change the state of the paper into accepted. However, it is not clear who activates this behaviour. Conceptually a paper should not have such an active behaviour, since it is in principle a passive object.

In this chapter we have described the modelling concepts of INCA-COM. Four models, (1) the information model, (2) the event model, (3) the behavioural

model, and (4) the communication model have been introduced. In the next chapter, we treat the principles for structuring object worlds.

Chapter 5

Structuring principles

In addition to the descriptional structuring of objects, INCA-COM offers four structuring principles on the domain of objects: *sort hierarchies*, *specification*, *composition*, and *grouping*. They offer a modeller meaningful relations among objects, in order to bring structure into an object world.

Sort hierarchies act as *taxonomical structuring devices*. A sort hierarchy outlines a UOD according to some point of view. A UOD can be viewed from different view points, each requiring a corresponding sort hierarchy.

Specification allows assigning one or more *sorts* to an object. Hence, the 'place' of an object with respect to other objects in the object world is determined. Sort hierarchies and specification are treated in Section 5.1.
The structures and relations, spanned by sort hierarchies and specification are the basis for defining three types of inheritance, which we treat in Section 5.2.

Composition allows an object to be composed of other objects forming a so-called *composite object*. A composite object is more than a mere set of objects. A set is just a collection of objects. A composite object has the status of an object. Additionally, a composite object may have a particular *structure*, relating component objects, by which the component objects influence each other. A composite object with an additional structure is a *system*.

Grouping can be used to collect useful and intensionally-defined (meaningful) sets of objects. The composition of an object is an example of an intensionally-defined set of objects. Composition and grouping are treated in Section 5.3.

Section 5.4 describes in detail the modelling of the UOD of a conference, and a conference-registration system. It illustrates the concepts of INCA-COM.

During the description of the modelling concepts introduced in this chapter,

we provide several heuristics and guidelines for modelling with INCA-COM. They are presented as numbered modelling requirements.

5.1 Sort hierarchies and specification[1]

Modelling a new application starts with an outline of the structure of the UOD. Structuring a UOD consists of finding the semantic primitives, and finding relations between these primitives. The semantic primitives are called *sorts*, and the relations between sorts give rise to so-called *sort hierarchies*. *Specification* allows a modeller to relate objects to their sorts, thus determining the place of an object with respect to other objects.

In this section we subsequently treat sorts, sort hierarchies, and the naming of objects. Moreover, when treating the sort hierarchies, two example specifications are given.

5.1.1 Sorts

In INCA-COM, *sorts* are the semantic primitives to model a UOD. Within many COMs every object is thought to belong to a specific *type* or *class*. Often such a type is only relevant during the modelling of objects, but it loses its relevance during execution of the object world. In case the type is also a meaningful run-time entity, it appears to be identical with the *set of objects* of that type. The notion of a sort superficially resembles the concepts of a type or a class, but it is not identical with these.

We distinguish two aspects of a sort, namely its *extension*, denoting the potential set of its instances, and its *intension* or particular *meaning* as expressed by means of a description. In this respect our notion of sort is more like the classical notion, in Latin termed *species*. Hence, it is more than merely a set or class of instances (i.e., the extension).

We have assigned the full *object status* to sorts, thus making it possible to describe sorts in the same manner as primary objects (see also Section 4.1.3). Another reason for granting sorts the full object status is that sorts often play an ambiguous role, depending on the point of view. See also Section 3.1.2 on the different points of view on a news paper and its instances. To solve

[1] The ideas of this section stem from cooperation with J. W. H. M. Uiterwijk and P. J. Braspenning and have been published as a conference paper *Specification and Inheritance in* INCA-COM (Uiterwijk and Braspenning, 1990).

this ambiguity, sorts in INCA-COM are full-blown objects, thereby making it possible to view sorts both as an instance of some sort, and as a class for other objects (instances).

By the introduction of sorts, each primary object is associated with at least one sort by means of a special relational descriptor (the sort descriptor). Since this must also be valid for sorts when viewed as instances themselves, one enters into a possibly infinite recursion. Therefore, the convention is that a sort also *can* have sorts it belongs to. Such sorts are called *metasorts*, and as far as a modeller wishes to specify them, metasorts can (again) belong to metametasorts, and so on. In this way the entire object world is fundamentally partitioned into two parts:

1. primary objects, i.e., objects which do not have instances, and
2. sort objects (sorts), i.e., objects which can have instances.

The partitioning is illustrated in Figure 5.1. The closed line represents the division between sort objects and primary objects; the dashed lines represent the division between sorts and metasorts. An advantage of the object status of sorts is that there exists no conceptual gap between instances and sorts as in, e.g., Smalltalk's class system. Treating sorts as objects leads to a homogeneous object world, consisting of objects, some of which are sorts, and some of which are primary objects. For the same principal reason the authors of ObjVlisp also have chosen to treat their classes as "first-class citizens" (Cointe, 1987), although their new class concept differs from our concept of a sort.

A sort object in its role of sort determines (partly) the structure of its instances (via its *extensional* structure), and additionally also behaves as a dynamical (*run-time*) object (via its *intensional* structure). This distinction in intensional and extensional structure of sorts is a valuable modelling aid of INCA-COM. In languages like Smalltalk and Loops *class variables* are both used for describing features common to all instances of a class (i.e., *extensional features*) and for features of classes themselves (i.e., features, which do not belong to separate instances, but to their classes: *intensional features*). This leads to somewhat confusing discussions on whether information should be stored in class variables of a class or, instead, in instance variables of its metaclass (Briot and Cointe, 1987). In our model such information distribution is clear, as will become evident in Section 5.2.

To support our reasoning about sorts and objects we use the following definitions.

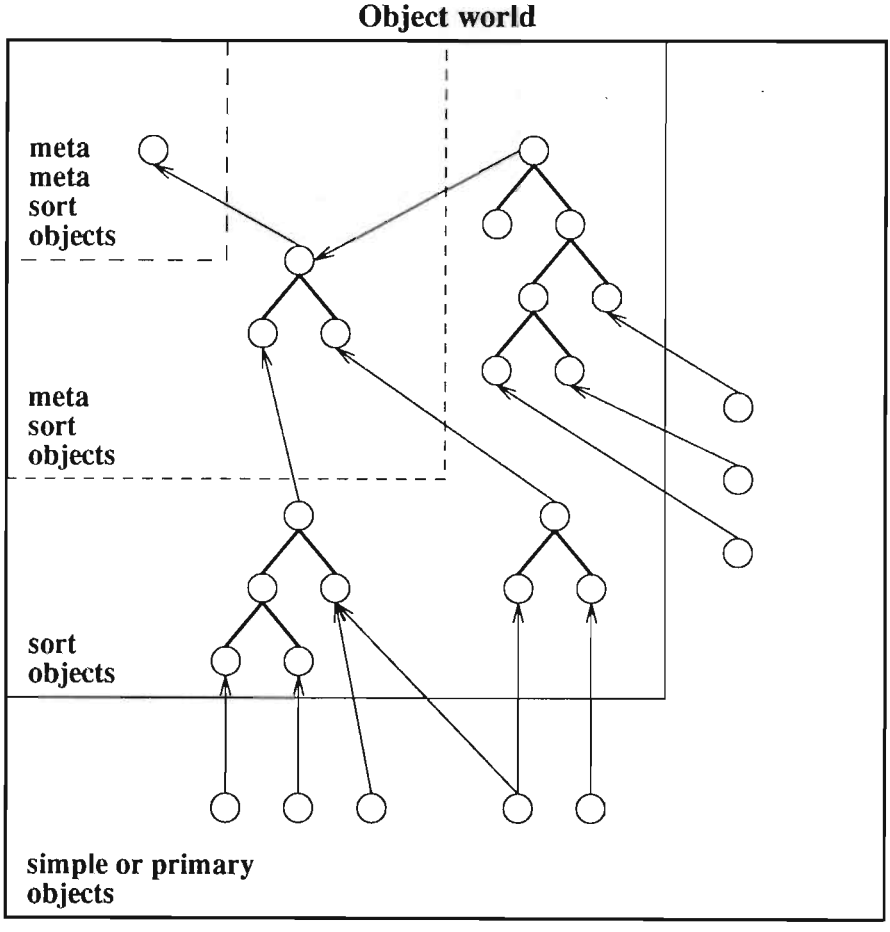


Figure 5.1: Basic partitioning of the object world.

Definition 5.1 *A* sort object *is a sort addressed as an object.*

Definition 5.2 *A* primary object *is any object not being a sort object.*

A basic modelling requirement is the following.

Requirement 5.1 *Every object is allotted (at least) one sort denoting its taxonomic place in the object world, after which every object partakes in (at least) one sort.*

Often the term classification is used to denote the structuring of object domains. However, we prefer the term specification to emphasize the difference among sorts in the INCA model and classes or types in many others (while referring again to the notion of *species* used in biological classification, see also Chapter 3).

In the literature (e.g., Stroustrup (1991)), normal types are usually distinguished from so-called abstract types, the latter being types without instances (and therefore only useful as a pass-through for shared characteristics). In the INCA model such a distinction is not semantically justified, since every sort object has an existence of its own (either with or without associated instances). In its role as sort *object* it may have its own properties, attributes, etc. Therefore, any difference between normal and abstract makes no sense in our object model.

5.1.2 Sort hierarchies

Within INCA-COM, *sort hierarchies* act as taxonomical structuring devices. A sort hierarchy outlines a UOD according to some point of view. Sort hierarchies have a *tree* structure, in which subsequent layers lower in the tree denote levels of increasing specialization according to the point of view expressed by the hierarchy. Sort hierarchies make it possible to specify certain sorts as subsorts (*specialization*) or supersorts (*generalization*) of other sorts.

Definition 5.3 *Let A and B be two sorts. B is a* subsort *of A in some sort hierarchy if and only if B is a specialization of A according to the particular point of view associated with the hierarchy. If B is a subsort of A, then A is a* supersort *of B.*

We note that, in this definition, B being a subsort of A depends on the sort hierarchy in which both sorts partake. As the specialization relation is transitive, all sorts above a particular sort in a hierarchy are to be considered as supersorts. But then, all (direct) instances of a sort (by specification) are also (indirect) instances of all sorts in the hierarchy above that sort. A pragmatic requirement, which ensures the largest set of such possible *inferences*, is expressed in the following modelling requirement.

Requirement 5.2 *Every object is modelled as a (direct) instance of the most specific sort(s) in one or more sort hierarchies.*

The specification of a domain by means of a sort hierarchy makes subsorts *more specific* than their supersorts. A (sub)sort's definition consists of the definition of its supersorts, and any additional descriptions, which are *specific* for the subsort with respect to its supersorts. Thus, on the one hand the intension of a subsort is more specific than the intension of its supersorts, hence it is applicable to a smaller number of objects. On the other hand, the intension of a supersort is less specific, and thus applicable to a larger number of objects than the intension of its subsorts. The order relation between intensions of sorts gives rise to a subset relation on the sorts' extensions. This is captured in the following requirement.

Requirement 5.3 *Let G and S be two sorts, and S be a subsort of G. Then the following relations hold: (1) The intension of S is more specific than the intension of G: $I_S > I_G$, where I_X denotes the intension of sort X and $>$ denotes the order relation between intensions; (2) The extension of S is a subset of the extension of G: $E_S \subset E_G$, where E_X denotes the extension of sort X and $\subset$ is the set inclusion.*

We stress that the specification of a domain by sort hierarchies introduces a subtyping mechanism. The implication of the previous proposition is that if sort S is a subsort of sort G, then an instance of sort S may be used wherever an instance of sort G is needed, since an instance of sort S is also an instance of sort G. The assignment of a sort G to an object o furnishes an *initial* set of descriptions for object o. The initial set of descriptions furnished by G is applicable to all its direct and indirect instances, which makes it possible to use an instance of sort S wherever an instance of sort G is needed.

Nevertheless, we do not hold the view that a *particular* instance of sort G, say g_1 is replaceable by another (direct or indirect) instance of sort G. The reason for this is that g_1 may have received additional descriptions, besides those already specified by its sort G. Those descriptions are not necessarily shared by other (direct or indirect) instances of G. Therefore, an instance of sort G is *in principle* not replaceable by an (indirect) instance of sort G, although an instance of subsort S remains an indirect instance of sort G.

With respect to *multiple inheritance* (see Section 5.2.2), we do not agree with the rather wide-spread opinion that it is meaningful to use a sort which is a direct subsort of more than one supersort (multiple supersorting). The reason is that the division of a domain into several (sub)sorts must be semantically relevant, i.e., any sort to be distinguished within a particular domain must

have its own meaning, different from other sorts *within the taxonomical point of view*. A sort which is a direct subsort of several other sorts entangles the points of view of its supersorts.

The point of view of a sort hierarchy acts as a *fundamentum divisionis*, used in classical hierarchical taxonomies (see, e.g., Joseph (1916)), according to which each sort in the hierarchy is different from its supersort. Several points of view can be modelled by using several sort hierarchies. We stress, that these different points of view must not be intermingled: different points of view on an application domain should be represented by strictly *disjoint* sort hierarchies. This view of sort hierarchies differs from, e.g., Lenat *et al.* (1990), who allow multiple supersorting, and additionally divide categories into mutually disjoint subsets. Such a division is expressed in INCA-COM by constructing two or more sort hierarchies expressing these points of view. Disjoint sort hierarchies on a domain span together a *forest* of hierarchy trees. The previous observations are condensed in the following requirement.

Requirement 5.4 *Specification of an object domain is performed by using strictly disjoint sort hierarchy trees.*

In contrast with sorts, which are not allowed to be the direct subsort of more than one sort, we do allow primary objects to partake in two or more sorts. These sorts are required to belong to different sort hierarchies, i.e., we do not allow objects to be instances of different sorts within one and the same sort hierarchy. The reason is similar to the one for not allowing multiple supersorting, namely that particularizing objects as instances of multiple sorts in one and the same sort hierarchy would deny the semantical difference between those sorts. However, this does not apply to sorts in different hierarchies, since they denote alternate taxonomic views on the same domain: they are not meant to be distinguished *from* each other, but they are meant to be *orthogonal to* each other. This gives rise to a further modelling requirement.

Requirement 5.5 *All (most specific) sorts of an object should belong to different (i.e., strictly disjoint) sort hierarchies.*

Thus, a primary object is allowed to participate (as an instance of several sorts) in different sort hierarchies. This is *not* an entanglement of points of view on the structure of the domain, but a multi-dimensional allocation of the object along sort(-hierarchical) dimensions. A similar notion of point of view is the notion of *frame of reference* (Bunge, 1977). A frame of reference M

is analogous to our point of view, introducing a set of functions on M which represent properties of objects within the frame of reference. We address this issue in Chapter 7 when describing the extensional features of objects belonging to a sort.

Example specifications

Below, the previous definitions and propositions are illustrated by two examples. The first example is on figure modelling, and the second on employee modelling. The latter illustrates how multiple supersorting can be avoided.

Example 1: Figure modelling

Figure 5.2 provides an object domain of figures, viewed from two different points of view (Shape and FillPattern). The thick lines in the figure indicate the subsort/supersort relation between two sorts. The sort hierarchy at the top specifies the object domain with respect to the form each object has; the root sort of this hierarchy expresses this by its name, Shape. The sort hierarchy on the left specifies the same domain, with respect to the fill pattern each figure has; again, the root sort FillPattern indicates the associated point of view. The orientation of the two sort hierarchies expresses the different points of view on the domain of figures. Particular objects can be an *instance* of the sorts in both hierarchies, which is indicated by the arrows pointing from objects to sorts. Of course, such a specification is only applicable if the point of view is appropriate. As a case in point, we mention that the instance of Point in the lower right corner has no sort in the FillPattern sort hierarchy, since a point has no fill pattern.

In this way it may appear as if we have fully described the sample domain. However, in addition to the descriptional structure of objects due to their specification, our model allows instance objects to have their own descriptions, such as the length of a particular square, the radius of some circle, etc. Such (instance) descriptions are not indicated in Figure 5.2.

We do not require subsorts taken together to *exhaust* their immediate supersort (by which we mean that the unions of the extensions of the subsorts are equal the extension of the supersort). For instance, in many cases it will not be useful to subdivide the sort Rectangle into the sorts Square and NotSquare, since the latter sort has probably no relevant additional description compared with the sort Rectangle. In INCA-COM such an irrelevant sort is unnecessary, since we

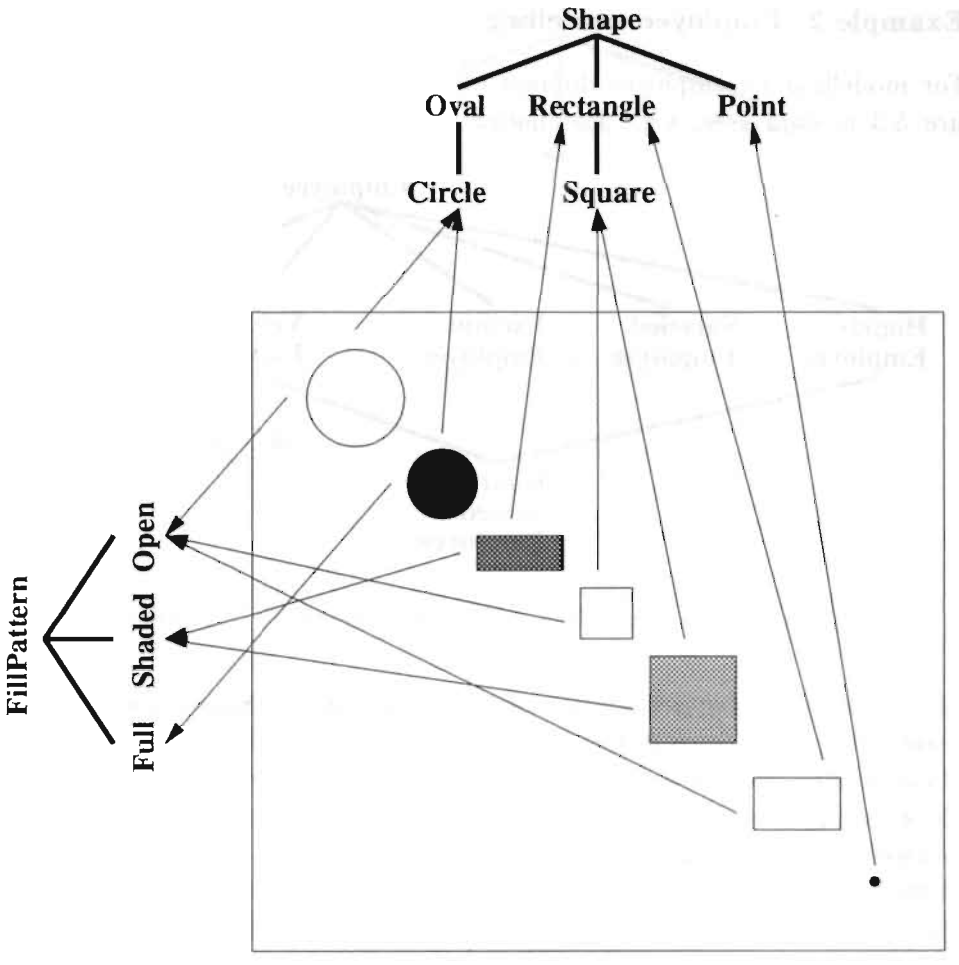


Figure 5.2: Sort hierarchies expressing different view points.

do not require that sorts in a sort hierarchy are instantiated only at the lowest level of the tree. This is illustrated in Figure 5.2 by square objects as instances of Square, and rectangles as instances of its direct supersort Rectangle. Thus, all squares are specified to belong to the sort Square, whereas rectangles which are not necessarily squares, are specified to belong to the sort Rectangle. Every figure can thus be allotted its most specific sorts.

Example 2: Employee modelling

For modelling an employee domain often a sort hierarchy as shown in Figure 5.3 is used (see, e.g., Rumbaugh *et al.* (1991, p. 66)). The problem

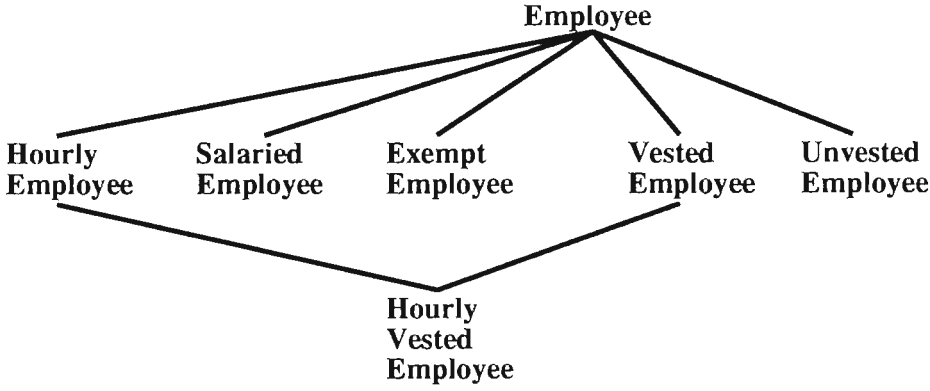


Figure 5.3: A multiple inheritance hierarchy of employees.

of such a hierarchy is that the specialization of Employee into its five subsorts is done with two different dividing principles. The first three subsorts, HourlyEmployee, SalariedEmployee, and ExemptEmployee, are discriminated by their paying status, whereas the next two subsorts, VestedEmployee and UnvestedEmployee, are discriminated by their pension status. Rumbaugh *et al.* (1991) suggest that a sort should never have two supersorts within the same dividing principle, e.g., a subsort with both HourlyEmployee and SalariedEmployee as its supersorts would be illegal. However, in their opinion the division of Employee into its five subsorts is still valid. In INCA-COM, we propose to specify such a domain by using two disjoint hierarchies. Each hierarchy has as its top sort the name of the dividing principle. This is shown in Figure 5.4. We note also that the names of the subsorts do not include the word Employee. We assume that these subsorts are introduced for specification of a domain of employees. The names of the subsorts should be chosen in such a way that the dividing principle becomes clear. Using the name Employee makes this less clear, and inhibits the reuse of a specification in other domains.

Here, we end the discussion of the two specification examples.

The additional principle described in the example specifications is formulated in the following modelling requirement.

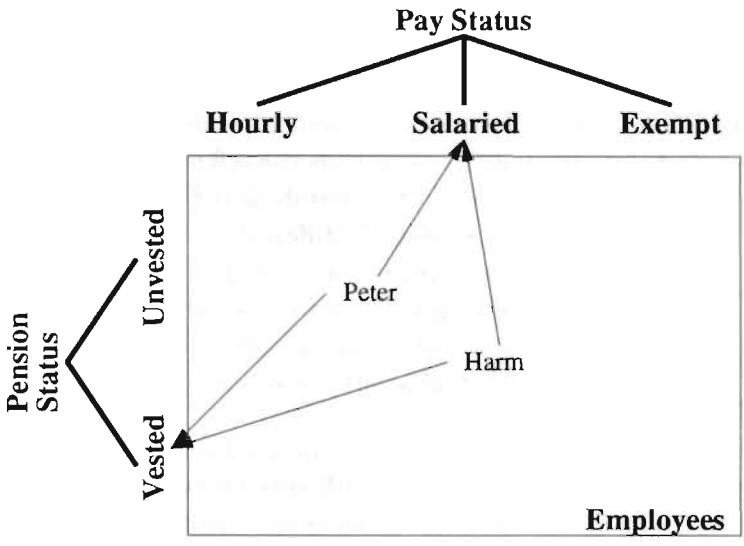


Figure 5.4: The INCA way of specifying the employee domain.

Requirement 5.6 *Only* relevant *sorts are used in the specification of a domain, i.e., those that specialize their supersorts with* relevant *information.*

This heuristic requirement may seem trivial at first blush. However, experience has taught otherwise; the use of rest classes is in fact common practice in many object-oriented application models. In some cases, the use of a rest sort *may be useful*, and should therefore not be strictly forbidden. For example, sometimes one wants to express that an instance is definitely *not* belonging to some sort at the same taxonomic level. However, in the majority of cases we view such a practice (e.g., creating a sort NotSquare as subsort of Rectangle) as not in agreement with the semantical notion of sort and, moreover, as unneeded.

5.1.3 Naming of objects

In order to avoid ambiguity in denoting sort hierarchies the names of root sorts are required to be unique. This is not only a matter of convenience, but is also natural, since different sort hierarchies should model different points of view on a domain, and the root sort of such a hierarchy should reflect this point of view.

Requirement 5.7 *Names of the sorts at the top of all sort hierarchies must be unique and should reflect the point of view associated with the sort hierarchy.*

We note that the name of a root sort in some sort hierarchy may thus equally well be used to denote this particular sort as the full sort hierarchy in question. As an example, we consider the hierarchies depicted in Figure 5.2. Although the same primary objects are modelled, different specifications are discerned in both hierarchies. The names suggest (as it should be) that in one view the modeller concentrates on the forms of the figures (Shape), i.e., only considering their silhouette, whereas the other specification structures the domain of figures as just filling patterns (FillPattern), not having any concrete form.

Definition 5.4 *Every sort has a* sort name *and a* full sort name. *The* sort name *is denoted by a text string. The* full sort name *of a sort consists of its sort name followed by the full sort name of its supersort, if any, separated by a dot.*

According to this definition the name space of all sort objects is treated by a *hierarchical naming scheme*. By virtue of Requirement 5.7 and Definition 5.4 full names of sorts are always unique. If no ambiguity exists for some sort (i.e., if its sort name is unique), the use of its full name is allowed, but not obliged. For instance, in the hierarchies of Figure 5.2 the leftmost and downmost sort may equally well be denoted by Circle and Circle.Oval.Shape; sometimes the use of full names clarifies the meaning of the sorts specified (Full.FillPattern instead of Full). Further, in the context of a particular sort hierarchy the use of full names is not necessary.

This naming scheme is analogous to the well-known directory structure of filenames. One of the differences is the order of the sort names constituting a full sort name, from more special to more general. In this way it is easier to memorize the intended meaning of sorts from their names (if well-chosen); for instance, compare names such as Large.Disk or Point.Shape with their reversely-ordered names.

In Chapter 3, an object was defined as having a unique name. The definition of sort names already introduces a naming scheme in which each sort has a unique (full) name. The requirement that every object (both sort objects and primary objects) have a unique identifier, and hence a unique name can now be realized by requiring each sort to have uniquely-named instances.

Requirement 5.8 *All direct instances of a sort must have unique names.*

To assure that equally-named instances of different sorts can be identified by their name, we define the instance name of an object in the following way.

Definition 5.5 *Each instance has an* instance name *and a* full instance name. *The* instance name *of an object is its object name preceded by the sort name of its sort, separated by the '::' sign. The* full instance name *of an object is its object name preceded by the full sort name of its sort, separated by the '::' sign.*

By this definition, each object can be identified by using its (full) instance name. Examples of the usage of this naming scheme are the instance names Square::mysquare (an instance name), and Circle.Oval.Shape::yourCircle (a full instance name).

We note that by Definition 5.4 and Definition 5.5 a sort can be named in two distinct ways: it has a (full) sort name, and it has a (full) instance name. For example, the sort named Point has the full sort name Point.Shape and the full instance name Metasort::Point (assuming Metasort is the name of the metasort in the particular domain). These names correspond (1) to the role a sort has an instantiable semantic primitive, and (2) to the role a sort has a sort object, being an instance of a metasort. Thus, a sort can be addressed as a sort by using its sort name, or it can be addressed as a sort object by using its instance name.

Furthermore, we require that each sort has an associated *name sort*, used to generate names for instances of the sort.

Requirement 5.9 *A sort has an associated name sort to generate names for its instances.*

In the next section we present the inheritance mechanisms in INCA-COM, which are based on sort hierarchies and specification.

5.2 Inheritance mechanisms

Sort hierarchies are used as a vehicle for inheritance, though they are quite useful even without such a mechanism for reasoning about the place of an object. For example, an instance of a sort in a sort hierarchy can be reasoned to be an instance of any of the supersorts of that particular sort.

In most literature inheritance is somewhat derisively called "the ability to create classes that will automatically model themselves on other classes" (Tello,

1989, p. 7). Snyder (1987, p. 166) states that "inheritance can be used to define a class in terms of one or more other classes" and Meyer (1988, p. 59) defines it as follows: "A class is a descendant of one or more other ones when it is designed as an extension or specialization of these classes. This is the powerful notion of (multiple) inheritance." Although specification and inheritance are intimately bound, we do not adhere to these viewpoints, because they deny the difference between specification and inheritance. In our view, specification is an *organizational tool* for structuring object worlds, whereas inheritance is a *programming tool*, used to obtain less coding effort. The view that inheritance is a programming tool is shared among others by Meyer (1988) and Cox (1987, p. 83) stating that "inheritance is a technique for bringing generic code into play when producing new code." Thus, generalization and specialization are used the denote the relations between sorts, whereas inheritance refers to a mechanism for sharing descriptions (Rumbaugh *et al.*, 1991, p .42).

Nevertheless, the specification of an object domain is useful for guiding the use of inheritance. INCA-COM supports three different types of inheritance (all three precisely defined below): *common inheritance*, *normal inheritance*, and *structural inheritance*.

Definition 5.6 Common inheritance *indicates the conceptual transfer of descriptor/value pairs from (the extensional description of) a sort to its instances. The inherited descriptions will be strictly common to all instances, i.e., all instances shall possess the descriptors with the associated values.*

The term strictly should be taken literally, i.e., if only one instance of a sort lacks some property then this property should *not* be modelled as common. We use the term 'common descriptions' (CDs) as a shortcut for descriptions to be obtained by common inheritance. The same holds for 'normal descriptions' (NDs) and 'structural descriptions' (SDs) to be defined below. An obvious example is modelling humans as instances of some sort HumanBeing with the common property of being mortal.

The use of common inheritance enables universally quantified statements on instances. As such, common descriptions can be conceived as partially defining the objects (namely partially determining *and fixing* their descriptive structure). We note that the inheritance of common descriptions, being universally valid, may be described by first-order predicate logic. The category of descriptors most appropriate for common inheritance is the category of properties, but also relations (e.g., between all instances of one sort and all instances of a second sort) can often profitably be modelled as common descriptors.

Definition 5.7 Normal inheritance *indicates the conceptual transfer of descriptor/value pairs from a sort to its instances. The difference with common inheritance is the possibility to overrule the inherited values of the descriptors at the level of instances. The inherited values are default values for instances, for which reason normal inheritance can be seen as* default inheritance.

Normal inheritance allows *ab*normality to be described. Normal inheritance is analogous to common inheritance except that the descriptive values prescribed by the sort to inherit from can be overruled. This is similar to the model of exception handling (see, e.g., Touretzky (1986) and references found there). The sort from which descriptions are inherited only provides *default* values for the descriptors, which may be overwritten either by whole subsorts (*exceptional subsorts*) or by particular instances (*exceptional instances*).

An instructive example for this kind of inheritance may be the modelling of animals using a sort Mammal of which the extension is described, for instance, with normal property No_of_legs = 4 and subsort HumanBeing with overwriting property No_of_legs = 2 (exceptional subsort), whereas some disabled (one-legged) soldier would be modelled as an instance of HumanBeing with the overwriting property No_of_legs = 1 (exceptional instance).

In contrast with common descriptions, the inheritance of normal descriptions corresponds to some form of non-monotonic logic, e.g., default logic (Etherington and Reiter, 1985).

We remark that the treatment of exceptions is the task (and responsibility) of the modeller. This means that, whereas the model offers the tools to overwrite default values of normal descriptions, it does not prescribe *when* and *how* overwriting has to be done. As such, normal inheritance yields tools for the management of *normality*, but not of *abnormality*.

Definition 5.8 Structural inheritance *deals with the conceptual transfer of descriptors (without any particular values) from a sort to its instances. The values of the descriptors are to be specified by the modeller at the level of instances.*

In this way only a (partial) descriptional structure (i.e., without values) is imposed on the instances. Although the statements implied by structurally inherited descriptors say nothing in particular about an instance itself, they are still meaningful, since they *partially* determine the set of descriptions of the instance: they give a partial framework within which meaningful assertions on an object can be stated. As such, these statements do not denote

propositions on the domain, but **meta**propositions about the way objects are to be described (partially).

The use of structural inheritance will often be apparent in those cases where using normal inheritance would lead to overwriting of the inherited values for all or most instances. For instance, let us assume that persons are modelled as objects having a property Length. Since we feel it semantically unjustified to impose lengths on persons by using a sort Person with a normal property Length = 'Unknown' (or even worse Length = '180 cm'), followed by yielding new length specifications to all persons instantiated, we prefer to equip Person with "only" a structural property Length. Thus, by the use of structural inheritance we are saying, in effect, that our Persons should be described by, e.g., Length.

Since structural descriptions only impose certain descriptors on instances and since these descriptors are universally valid at the description level of all instances of a particular sort, structural inheritance may be modelled, like common inheritance, by first-order predicate logic.

We emphasize that, with respect to the inheritance modes discussed, all categories of descriptors are viewed as similar. A distinction with Smalltalk and LOOPS is the clearer notion of semantics-driven inheritance modes. Instance variables are subject only to structural inheritance in Smalltalk (values are only specified at the instance level) or normal inheritance in LOOPS (default values specified at the class level), but lack the possibility of common inheritance. Although the use of class variables in Smalltalk or LOOPS enables some form of *shared* information between instances, this is not *common* in the sense we use it, since each instance has access to the contents of the class variables of its class, which it can also modify.

Most other object-oriented languages, such as CLOS (Moon, 1989) and C++ (Stroustrup, 1991), show analogous restrictions on method inheritance and on shared descriptions as Smalltalk and LOOPS. To our knowledge no COM supports inheritance equally for all descriptions and as fully as INCA-COM.

5.2.1 Inheritance and the modelling process

The distinction between CDs, NDs, and SDs introduced above (after Definition 5.6) may be used as a modelling aid. This is expressed in the following requirement.

Requirement 5.10 *If descriptions are partially defining for* all *instances of a sort, they should be placed with the appropriate sort as* common *descriptions. If descriptions are meant merely to supply a descriptional structure to instances, they should be modelled as* structural *descriptions of the sort. If both cases do not apply, descriptions are best modelled as* normal *descriptions.*

The three kinds of descriptions used may correspond with three phases in the object modelling process. Since common descriptions *define* (partially) the objects, they are suited for a first structuring activity of the sorts in the object domain, the *identification of sorts.* Normal descriptions can be viewed as *assertions* on objects. Therefore, the modelling of these descriptions is most appropriate for a second phase of structuring, consisting of the descriptive *refinement of sorts.* Finally, the structural descriptions impose *assertional templates* on the instances, i.e., prescribing what kind of assertions should be possible. This corresponds mainly with a third phase of modelling, namely the description of objects that will belong to the different sorts. These three phases exhibit a gradual shift from the emphasis on sorts to instances. Summarizing, they may guide the modelling process as follows.

Requirement 5.11 *The structural specification of an object domain should be divided into three phases, corresponding to identifying common descriptions (partial definitions), normal descriptions (sort-based assertions) and structural descriptions (object-directed assertional templates) consecutively.*

Since inheritance uses the sort hierarchies for information distribution and since sort hierarchies are built from specializations, the distinction between the three kinds of descriptions entails a natural enhancement of the semantics of specialization. Therefore, specialization not only adds information to objects or restricts the value domain for descriptors, but may also change the *status* of some descriptions to be inherited in a well-defined manner. The allowed transformations are from SDs to NDs (adding normal values of descriptions) and from NDs or SDs to CDs (in both cases stating common values of descriptions by definition).

5.2.2 Multiple inheritance

As mentioned in Section 5.1.2 we do not allow sorts to have multiple (direct) supersorts. Instances, however, may have more than one sort. Therefore, instances are in principle subject to some form of *multiple inheritance*

in INCA-COM. Since we require such multiple sorts to be part of strictly disjoint hierarchies and since we consider sort hierarchies to represent particular points of view on the object domain, multiple inheritance seldomly results in *conflicting* inherited descriptions. This is due to the fact that descriptions are often fully dependent on a particular point of view and as such unlikely to appear in more than one sort hierarchy. Therefore, although the INCA-COM *in principle* does allow multiple inheritance, there will be rarely conflicting situations *in practice*.

We note that in the case of *conflicting multiple inheritance of* CDs the following position is possible. Since CDs are by definition universally valid for all (direct and indirect) instances of a sort, it follows that if an instance inherits a common descriptor along two paths with different values, both values must be valid, i.e., that the descriptor in question is a *multi-valued* description. We note that this point of view does not follow from the INCA-COM described so far, since it is also possible to forbid the use of homonymous common descriptors in different sort hierarchies as an illegal modelling activity, thereby avoiding conflicting multiple inheritance of common descriptions. The consequences of whatever point of view is held are not elaborated here. Also other details of multiple inheritance mechanisms (for normal descriptions) await further elaboration.

We have elaborated upon the structuring of a UOD by means of sorts, sort hierarchies, and specification. Within the framework of Chapter 4, sorts and sort hierarchies are particularly used for modelling the core of the framework consisting of natural sorts, and for modelling application domains consisting of role sorts. Specification allows objects to be assigned to sorts determining their place in the object world.

Having described sort hierarchies, specification, and their associated inheritance mechanisms, in the next section we address the concepts of composition and grouping in INCA-COM, which are necessary for describing composite objects and systems of interacting components.

5.3 Composition and grouping

Besides the structuring of an object domain by sort hierarchies and specification, INCA-COM supports objects composed of other objects. Most object conceptions describe these *composite objects* using a hierarchical-decomposition principle (e.g., LOOPS (Bobrow and Stefik, 1983; Stefik and Bobrow, 1985)

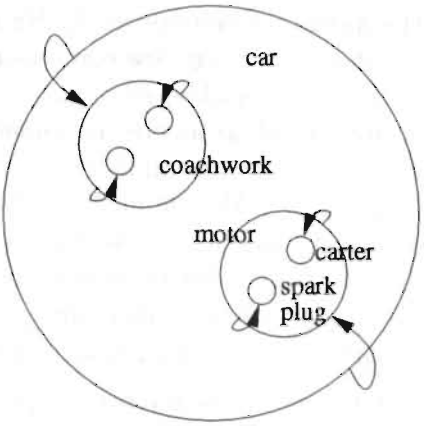


Figure 5.5: Graphical view of a composite object.

and ThingLab (Borning, 1981)) in such a way that the object composition can be represented by a *tree* (Kim *et al.*, 1987). By such a *part-whole hierarchy* the composition seems to be ordered quite satisfactorily.

However, semantically it is then impossible to decompose, e.g., a car object into *just two subobjects* like a coachwork and a *specific part* of the motor object, when a car is considered to be composed of at least a coachwork and a motor (by itself a composition of a collection of motor pieces). Within a particular goal-directed focus we may consider the decomposition of a car to consist of precisely *this* coachwork and *that* motor piece (Figure 5.5). Therefore, we argue in favour of the possibility to represent the (de)composition of an object by a general *set* and not by a tree. Such a set semantically bounds the subobjects that can be focussed at the same time without being obliged to consider larger, embracing objects, for which we may have no interest within the present focus. Formulated slightly differently: such a set only sets limits on the meaningfulness of considering certain decompositions of the object.

Bunge (1977) postulates that "the world is composed of things" and that "things are grouped into systems or aggregates of interacting components" (Postulates $M2$ and $M4$ in Bunge (1977, p. 16)). The first postulate is the basis of the naturalness of object-oriented modelling, assuring that object-oriented conceptual modelling is indeed a way of accurately modelling a human being's perception of the real world. The second postulate sheds light

on the question what the nature of systems is. In Bunge's view the world is a large system, of which all other things are components. Of course, such a view is legitimate, but it is hardly usable, since the world or universe should thus be seen as a composition of all atoms in the universe. Bunge's final postulate states however that "there are several levels of organization" (Postulate 10 (Bunge, 1977, p. 17)), which makes modelling and describing the world easier. In Chapter 2 we already mentioned the importance of a level concept in software development. Although software development entails the creation of artificial systems, the ontological principles are applicable, since ontology studies both natural and artificial (i.e., man-made) systems.

In this section, we elaborate the composition concept in INCA-COM. A composite object is not a simple bundle of component objects. Rather, we focus on the systemic aspect of composite objects, which allows us to contribute holistic features to them. We stress the important distinction between a composite object and a system. A composite object is an object with a composition. A *system* is a composite object, enhanced with a structure. As an example of the difference between a composite object and a system, consider again a car, composed of a coachwork, a motor, a carter, a sparkplug etc. A working car is not just a composite of components, but a system of components which interact in a well-defined way. If we decompose a car, lining up all the components, we do not have a car: we only have the collection of components. If we combine the components in such a way that the wheels are connected to the coachwork, and the motor forces the wheels to rotate, the result is a car. However, if we combine the components in some other way, the resulting system might not even come close to a car. So, a system is a composite object with additional value. The additional value of a system is a particular *structure*, consisting of the relations and interactions between the components.

5.3.1 Composite objects

Using the concept of an object as defined in Chapter 4, we follow Bunge (1977) for the conceptualization of composite objects. Bunge's basis for defining composite objects is the assumption that there is an *association theory*. This allows objects to associate with each other to form other objects. Such an association theory is formally captured in the definition of a commutative monoid of idempotents, including an association operation (Bunge, 1977, p. 113). The important parts of the definition are presented below (following Wand (1989)).

Definition 5.9 *Let Ω be the object world. The binary operation $\cdot$ called* association *is defined in the following way.*

1. *If x and y are objects, then $x \cdot y$ is also an object;*
2. *$x \cdot x = x$ (objects are idempotent under association);*
3. *$\cdot$ is commutative and associative.*

In other words, the above association theory states that an association of objects is itself an object, that the order of association is not important and that an object cannot be associated with itself to form a new object. Using the above association theory, an object association $x_1 \cdot x_2 \cdot \ldots \cdot x_n$ represents an object composed of other objects.

Definition 5.10 *An object o is* composite *iff it is composed of objects other than itself. I.e., $o \in \Omega$ is composite iff there exist objects $y, z \in \Omega$ such that $o = y \cdot z$ and each differs from o. Otherwise the object is* simple.

The composite-object definition boils down to considering an object as composite if it can be described as consisting of (at least) two other objects. For example, if a computer can be decomposed into a system unit and a monitor (each of which might be decomposed further), then it is a composite object. The composition principle for objects is the basis for the definition of the *part_of* relation.

Definition 5.11 *If x and $y \in \Omega$ are objects, then y is a* part_of *x iff $y \cdot x = x$. The symbol for the part_of relation is: $\sqsubset$, so $y \sqsubset x$.*

We can now define the *composition* of an object using the *part_of* relation. The composition of an object o is the set of objects which are part of o.

Definition 5.12 *The* composition *C of an object o is the set of its parts: $C(o) = \{y \in \Omega | y \sqsubset o\}$ for any $o \in \Omega$.*

The interesting thing to remark about the above definition is that it defines the composition of an object as the *set* of all part objects. The latter set is intensionally defined by the function C, which consists of applying the predicate $\sqsubset$ to all objects to find an object's composition. Such a *meaningful set* of objects is called a *group* in INCA-COM(see Section 5.3.3).

The difference between the composite object $o = x \cdot y$ and its composition $\{x, y\}$ is, that the former is an object, while the latter is a group. Furthermore, the part_of relation is a partial order relation, meaning that it is (1) reflexive ($x \sqsubset x$), (2) asymmetric ($x \sqsubset y \rightarrow \neg(y \sqsubset x)$), and (3) transitive ($x \sqsubset y \wedge y \sqsubset z \rightarrow x \sqsubset z$). The set of composite objects is denoted by the symbol Γ.

A composite object is the result of associating two or more objects. The features of such a composite object are divided into *hereditary* and *emergent* features. First, features of a composite object may be related to the features of objects in its composition. Such features are thus inherited by the composite object by composing it of other objects, and are called hereditary descriptors. Second, a composite object may have features which emerge as a result of combining the parts in a particular way. Such features can be modelled and described for the composite object because of its object status. Hence, a composite object can have its own descriptors, which emerge through composition. This is captured in the following definition.

Definition 5.13 *Let F be a feature of an object o with composition $C(o)$. Then*

1. *F is a* hereditary feature *of o iff there exists $y \in C(o), y \neq o$, such that F is a feature of y.*

2. *F is an* emergent feature *of o iff there is no $y \in C(o), y \neq o$, such that F is a feature of y.*

As an example consider again a composite object car. The colour of a car is usually taken to be the colour of its coachwork. Since the coachwork of the car is an object in its composition, the colour of a car is an hereditary feature. Analogously, a car's ability to drive is not a feature possessed by any of its parts. It emerges because the parts are put together in a particular way, so that the motor propels the wheels, the motor is attached to the coachwork etc. The car's ability to drive is an *emergent* feature.

We note that the concept of an hereditary feature is not related to the concept of inheritance in object-oriented programming languages. In such programming languages, inheritance is a mechanism for distributing features along class hierarchies, in order to equip the instances of classes with the appropriate set of features. In object-oriented modelling, classification is performed on the basis of the features an object has. Inheritance of features in the sense of furnished by a class or superclass plays no role in object-oriented modelling.

However, having hereditary features as the result of composition is useful in modelling. It is sometimes implemented in programming languages using delegation, i.e., a car can be made to drive by delegating behaviour to the motor and the wheels. Inheritance is not a modelling tool, but a text-distribution tool.

The introduced general notion of composition is rather simple. Consider the example given in Figure 5.6. As the figure illustrates, the composite object

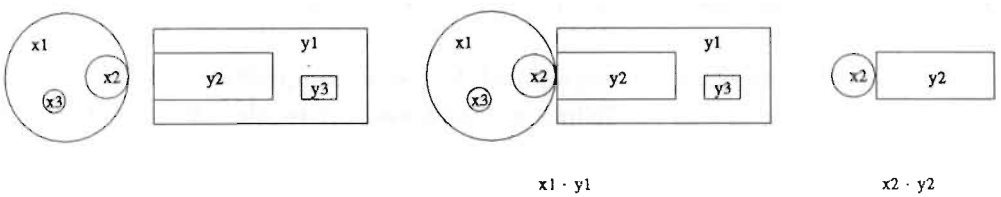


Figure 5.6: The objects x_1 and y_1, and the composite object $x_1 \cdot y_1$.

$x_1 \cdot y_1$ has a part, $x_2 \cdot y_2$, which is not a part of any of the constituent objects x_1 or y_1. It is better to have a composition principle in which the composition of a composite object equals the union of the composition of its components, i.e., $C(x_1 \cdot y_1) = C(x_1) \cup C(y_1)$ in the above example. Furthermore, since the part-of relation is transitive, every part object is considered to be in the composition of an object. Thus in the case of an animal society, the composition of the society is not only the set of animals, but also the organs of these animals, and even the atoms making up the organs. The introduction of a refined composition principle, called *atomic composition* (or *A-composition* for short) proves to be more useful in the decomposition of composite objects into their components.

5.3.2 A-composition

The idea behind A-composition is that we want to be able to decompose an object or system into components which belong to a certain sort. The *A-composition* (or *composition at the A-level*) of an object x is the set of parts of x that belong to sort A. Symbolically: let A be a sort and let o be an object (i.e., $o \in \Omega$). The *A-composition* of o is the set of its A-parts:

$$C_A(o) = C(o) \cap A = \{y \in A | y \sqsubset o\}.$$

The previously defined composition is sometimes also referred to as the *absolute composition* in order to distinguish it clearly from the A-composition. The concept of A-composition has the property that the A-composition of a composite object $x_1 \cdot y_1$ equals the union of the A-compositions of the components: $C_A(x_1 \cdot y_1) = C_A(x_1) \cup C_A(y_1)$.

As an example consider the composition of a body of water. The molecular composition of a body of water is the set of its H_2O molecules. The atomic composition of the same body is the set of H and O atoms composing it; and the elementary-particle composition is its total collection of electrons, neutrons, and protons.

A-composition introduces a concept of *levels* in composition. Such a level concept is a fundamental modelling aid with respect to abstraction. It allows a composite object to be decomposed in particular components of a certain sort. A computer program can be decomposed into modules, modules can be decomposed into functions, and functions can be decomposed into statements, and statements are composed of characters. During the development of complex software systems (i.e., man-made artefacts) we are not interested in the characters of the final statements making up the program. Instead, we like to focus on architectural issues such as the gross organization of modules and the interaction between them. The atomic level of specifying computer programs is the statement.

5.3.3 Groups of objects

Beyond the structuring principles such as a sort hierarchy and a composite-object graph we introduce another structuring principle, the *group of objects*. This should be a *meaningful set* of objects. Hence, a group of objects is more than just some collection of objects. The importance of the group concept is that it provides the modeller with a tool to model coherent sets of objects without loading the object world with taxonomies or composite objects of limited importance. We require that a group can be defined by a meaningful predicate.

Definition 5.14 *A* **group of objects** *is a set of objects, defined by a meaningful predicate.*

We note that a group, in contrast to a sort or a composite object, has *no* object status. That is, groups are fundamentally different from the objects that are being grouped by them.

In Section 5.3.1, one such meaningful predicate, based on the *part_of* relation between objects, has been used for defining a composite's object composition. Thus, the composition of a composite object is a group, defined by the *part_of* relation.

5.3.4 Systems

The composition is one of the important aspects of a composite object. In order to distinguish a system from a mere composite object, it is necessary to introduce the concept of *interaction* between objects.

Earlier in this section we introduced a system as a composite object with additional value. We can now be more precise about this additional value. The distinguishing feature of a system is that its components *interact*. Interaction is based on a coupling between objects, which is different from a mere relation, such as being older. A coupling (or bond) between objects makes a difference to its relata. Thus, two objects are coupled or bonded in case one of them *acts* upon the other. An object acts upon another object if it modifies the latter's history (although we have not defined the history of an object, it is thought of as the trajectory of its state function in the state space). If both objects act upon each other, they are said to interact. The interaction between components defines the influence each component has on other components. The set of influence relations defines the *structure* of a system. Furthermore, a system has an *environment* consisting of the objects outside the system, which influence (parts of) the system. Thus, a system is defined in the following way.

Definition 5.15 *An object is a* system *iff it is composed of at least two different coupled objects. The* composition *of a system is the set of elements of the system. The* environment *of a system is the set of elements not in the composition of the system, which are influenced by or influence elements of the composition of the system. The* structure *of a system is the set of influence relations between the elements of a system.*

We note that, similar to a composite object, for an object to be called a system, it requires to be composed of at least two other objects. The difference between a composite object and a system is the influence among the components. For the set of systems we use the symbol Σ. According to Definition 5.15 and Definition 5.10 of a composite object, the set of systems Σ is a subset of the set of composite objects Γ, which is a subset of the entire object world Ω: $\Sigma \subset \Gamma \subset \Omega$.

For ease of systems analysis, two organization principles are often used. A first organization principle is that different *categories* of systems are distinguished, each of which has its own type of *atomic element*. The atomic elements of social systems, e.g., are individual persons. Although the brains and other body parts are part of individuals, they do not qualify as elements of social systems, since they do not enter into social relations. Ontology distinguishes the following five system categories; after the category, the category elements are given in parentheses:

1. physical things (atoms and fields);
2. chemical systems (protein-synthesizing units);
3. biosystems (organisms and their organs);
4. sociosystems (societies and their subsystems);
5. artificial or technical systems (man-made artefacts).

A second organization principle is the *level of organization*. Within each system category there are systems composed of elements which are not atomic, but which are themselves systems composed of atomic elements. Thus, in the category of social systems there might exist levels such as persons, which compose into families, which compose into cities, which compose into provinces, which compose into countries etc. Thus, each level of organization introduces a system sort, which decomposes into systems of a lower level.

5.3.5 Systems in INCA-COM

Having introduced a concept for a system, we can now be more precise about the two different types of system in INCA-COM. Following the framework of Chapter 4, an information system consists of two parts: a passive object world, representing a UOD, and a system of active objects communicating about the object world.

The object world is a composite object, i.e., its composition is the set of objects acting as representation of things in the UOD. In the information model and the event model, we have chosen to model such objects by means of descriptors, events, and laws, constraining the state space of objects. Modelling an object world thus has a declarative nature, expressing which state of affairs holds in the UOD and which objects are represented. We remark that the term object

world is chosen deliberately to express that the representation of a UOD is not in itself a system. By this we mean that, although some objects in an object world may influence each other, not every object (directly or indirectly) influences other objects. We reiterate that particular objects in an object world can be a system. The specification of an object world thus consists of the following steps: (1) determination of the things in the UOD to be represented as objects; (2) for each object: describing it in terms of descriptors, events, and laws.

Active objects form a system by communicating on the object world. The specification of an active object is done in the behavioural model, expressing which tasks can be performed by active objects. The structure of an active-object system consists of the set of communication links between the active objects in the system, specified in the communication model. The environment of such a system is the set of objects not in the composition of the system, which communicate with elements in the system. The specification of behaviour has a procedural nature, which is better suited for expressing task-like steps. The specification of an active-object system consists of the following steps: (1) determination of the appropriate object world; (2) determining which active objects are component of the active-object system; (3) determining the environment of the system, and (4) describing the structure of the system in terms of communicative associations (influence relations) between active objects.

5.4 A conceptual model

In this section we describe a conceptual model of a *conference-registration system*, and its UOD. The UOD is divided into a world, and an application domain, the conference-registration domain. The modelling process illustrates the concepts we have introduced in Chapter 4 and the previous sections of this chapter.

The description of the case to be modelled is as follows. The conference is organized by an organizer, who distributes a call for papers. Potential authors who have received or noticed the call for papers, can write and submit papers for presentation at the conference. Each paper is assigned to a referee, who judges the paper and decides to accept or reject it. The result of the refereeing process is communicated to the senior author of the paper.

The specification of a conceptual model always follows the structure of the framework of Figure 4.4. A modeller first specifies the natural sorts of interest, then the application domain by means of role sorts, and finally the application.

In the example we specify

- the world containing the natural sorts person and article (Section 5.4.1);
- the conference-registration domain containing the role sorts researcher, conference organizer, referee and conference paper (Section 5.4.2); and
- the conference-registration system consisting of the active objects organizer, and referee, admitting articles, and refereeing them, respectively (Section 5.4.3).

5.4.1 World

The world contains sorts like HumanBeing and Person. Such *natural* sorts are meant to be used in various application domains (see Section 4.2.1). In order to enhance the reuse of HumanBeing and Person, they are modelled by the least common description, useful for every application wishing to use HumanBeings and Persons. Thus, the sort HumanBeing is described by a date of birth and a place of birth, and the sort Person is described by a name, an address, and a residence. Each sort specified introduces a *name sort*, which is used to generate names for identification of instances for the sort under consideration (see Section 5.1.3). For HumanBeing the name sort generates numbers (nr) as instance names and for Person the name sort constructs instance names consisting of an identification number (ID).

Role sorts, introduced in application domains, are institutionally defined sorts. The instances of such sorts are created within particular application domains. Often, such an institutional instance is a role of an instance of a natural sort. Such an association between a natural instance and a role instance is realized by linking the two instances with a *role link*. The boundary between natural sorts and role sorts seems appropriate to distinguish between naturally occurring objects, and institutionally defined objects. However, as the natural sort Person illustrates, this is not really the case: one may argue that properties such as name, address, and residence are not properties of Persons, but attributes, institutionally defined by societies, and attributed to HumanBeings when playing a role as a Person. Thus, an application domain may be based on other application domains, resulting in a *nesting* of application domains (see Section 4.4). This is illustrated in Figure 5.7, showing the structure of objects playing particular roles. The arrow at the top of the figure shows the direction in which subsequent roles may be assumed.

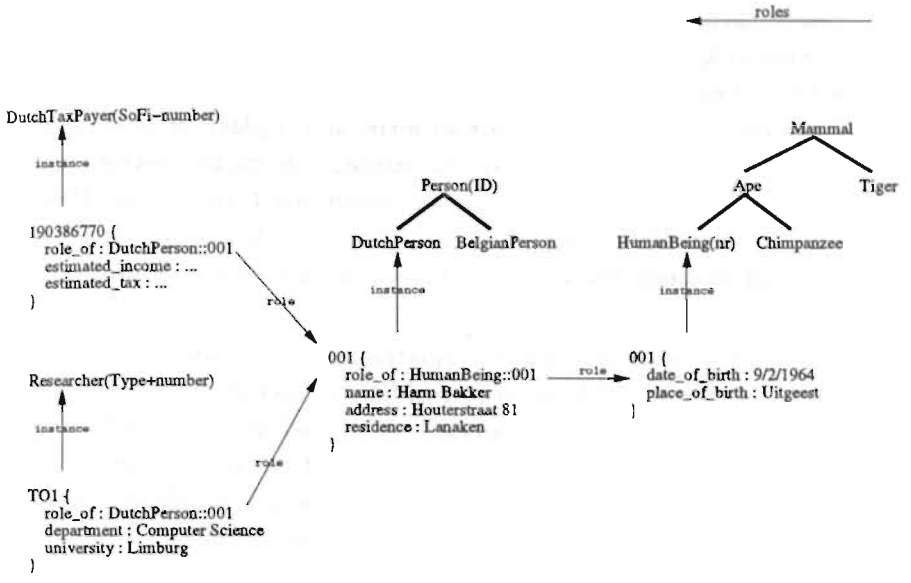


Figure 5.7: The structure of roles.

In Figure 5.7, four instances are described, namely 001, 001, 190386770, and TO1, from right to left respectively. Two of these instances are named equally with the instance name 001. Each instance belongs to a particular sort, indicated by the instance_of relation to its sort. 001 belongs to HumanBeing, 001 belongs to DutchPerson, 190386770 belongs to DutchTaxPayer, and TO1 belongs to Researcher. Shown in parentheses following the sort name, the name sort is given. In order to distinguish equally-named instances from each other (such as the instances named 001 of the sorts HumanBeing and Person), instance names can be preceded by their sort name, separated by a '::' (see Section 5.1.3). This gives rise to the instance names: HumanBeing::001, Person::001, DutchTaxPayer::190386770, and Researcher::TO1. Also shown in Figure 5.7 are two sort hierarchies, namely the sort hierarchy Mammal (with subsorts Ape, HumanBeing, Chimpanzee, and Tiger), and the sort hierarchy Person (with subsorts DutchPerson and BelgianPerson).

The instances in Figure 5.7 are linked to each other by role links. The instance HumanBeing::001 has a role Person::001, which has two subsequent roles, DutchTaxPayer::190386770 and Researcher::TO1. In the figure this is represented by the role_of links from role instance to role playing instance. The structure of the role links between several instances enables particular

sets of descriptors to be located with the sort to which they belong. An instance which is a role of another instance (e.g., the role Researcher::TO1 of Person::001), will have (access to) the complete descriptions of all nested roles. Thus, Researcher::TO1 has (1) a date_of_birth and a place_of_birth descriptor, furnished by its HumanBeing instance, (2) name, address, and residence from its Person instance, and (3) a department, and university from its sort Researcher. When accessed from Researcher::TO1 only the descriptors defined by its sort (i.e., department and university) can be modified. The other descriptors can be used as read-only values.

It is interesting to see that multiple registrations duplicate such descriptors. For example, the name, address, and residence descriptors are duplicated in almost every registration. The disadvantage of such a duplication is obvious: if a person moves to another residence, he will have to change the value of the residence descriptor in every registration. The reason for this has been a lack of connectivity between different registrations. In the future, it may be wise to construct registrations following our proposed role structure. Every descriptor is registered in only one place, which can be accessed by different subroles in order to retrieve its value. Whenever a change is necessary, it only has to be registered in one place.

We remark that the role specification also takes place at the sort description level. The specification at the sort level indicates which sorts of objects can play the role being specified by the sort under consideration. For example, the sort description of DutchTaxPayer would include a role specification with DutchPerson, indicating that DutchPersons can play the role of a DutchTaxPayer.

It is not always necessary for a role object to be associated to another object playing the role. Applications may instantiate role sorts without specifying a 'base' object to play the role. However, such an instance will not be described by the descriptors the base object offers. For example, an application instantiating Person, without giving a HumanBeing to play the role, will not have a date_of_birth and place_of_birth. If these are somehow needed, the modeller must provide these descriptors and values as additional descriptions. An intelligent tool, such as an object editor may assist a modeller in describing objects in such a way.

We have given the basic structure of the modelling framework. The sort HumanBeing is a natural sort, which enables several other sorts to be specified, linked by role associations to the natural sort. We remark that every role sort

is introduced within some application domain, meaning that only applications within the application domain may create instances of the sorts in the application domain. For example, Persons are created by municipal secretaries, who have been authorized to create instances of Person, whereas TaxPayers are created by the taxes department of a country. This illustrates that all role sorts are institutional sorts.

5.4.2 Conference-registration domain

In order to describe the conference-registration system we first have to consider the *conference-registration application domain.* Within the modelling framework, it is another level of role sorts based on previously-described roles, such as Person and Researcher. The sorts in the application domain of the conference-registration system taken into account are: conference paper, author, referee, conference organizer. There might be more sorts, but for brevity's sake we restrict ourselves to the ones mentioned.

Author is a role of researchers. Since researcher is a role of persons, author is a *sub*role of persons. Referee is also a role of researchers (and a subrole of persons). Conference paper is a role of articles. Articles have not been introduced; they are located within the same application domain as researchers. The organization of the framework with the corresponding sorts is shown in Figure 5.8.

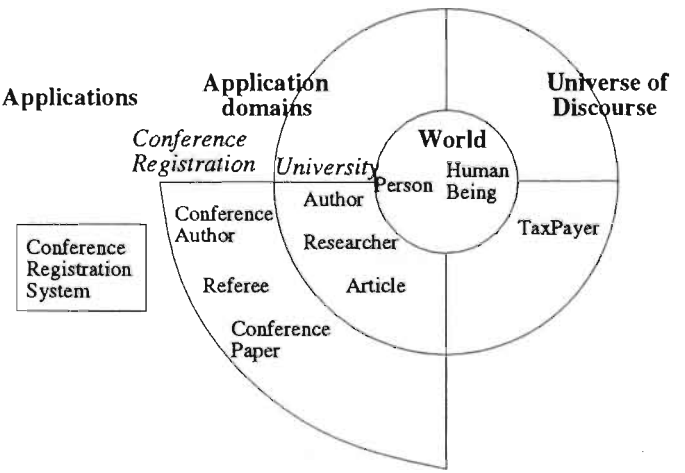


Figure 5.8: The modelling framework for the conference registration.

The description of these objects is shown in Figure 5.9.

```
Sort Author (Name sort: a_number) {
  Role_of: Researcher
}

Sort ConferenceAuthor (Name sort: ca_number) {
  Role_of: Author
}

Sort Referee (Name sort: r_number) {
  Role_of: Researcher
}

Sort ConferencePaper (Name sort: paper_number) {
  descriptors:
    Role_of: Article
    Author : ConferenceAuthor
    Referee : Referee
    Rating : Real
  }
```

Figure 5.9: Roles in the conference-registration domain.

We remark that modelling any (new) application domain should be preceded by finding the proper starting point among available sorts in other application domains. The conference-registration domain illustrates this. The conference-registration domain is based on the university domain. It introduces roles to be played by objects from the university domain. The university domain is based on the natural sorts Person and HumanBeing from the world of the UOD representation.

Having described the conference-registration domain, the final modelling step consists of describing the conference-registration system in the next subsection.

5.4.3 Conference-registration system

The conference-registration system stands for the conference organizer. An author who has written a paper, submits it to the conference organizer. Such a submission initiates a directive conversation (see Section 4.7), in which the author *directs* the conference organizer to admit the paper, and possibly referee it. Upon receipt of the paper, the conference organizer sends an acknowledgement of reception to the author and *commits* himself to execute the procedure for admitting a paper at the conference. The communicative action in response to the directive one is described in Figure 5.10.

```
ON DIRECT receive-paper(p)
DO STATE p.received
    COMMIT admit-paper(p)
```

Figure 5.10: The receipt of a paper.

The result of execution of this communicative action is (1) the creation of the fact that the paper has been received, and (2) the creation of the agendum to execute the admit-paper procedure. This procedure consists of checking the paper requirements, such as the submission deadline, and the number of words. If these are met, the organizer sends the author a message, telling that the paper has been admitted and will be judged by a referee. If the paper requirements are not met, it is rejected. On admittance of the paper, it has to be assigned to a referee. The specification of the behaviour of the conference-registration system is done in a procedural language. Figure 5.11 shows the specification of the admit-paper behaviour.

A conference paper p is rejected if (1) its date of reception is later than the submission deadline, or (2) the number of words is not within the limits imposed. Otherwise, p is admitted, and a subsequent agendum is created to take care that p is being assigned to a referee. For this referee assignment, a separate behaviour must be specified. In an analogous way, the communication and behaviour of a referee may be specified.

```
admit-paper (p: Conference-paper) {
  IF p.date-of-reception > submission_deadline THEN
    STATE p.reject
  ELSE IF p.no_words > upper_limit OR
       p.no_words < lower_limit THEN
    STATE p.reject
  ELSE
    STATE p.admitted
    DIRECT assign-to-referee(p)
}
```

Figure 5.11: The admit-paper behaviour.

5.5 Chapter summary

In this chapter, we have presented four structuring principles in INCA-COM: sort hierarchies, specification, composite objects, and grouping. Sort hierarchies structure an object world by relating sorts to each other by means of generalization and specialization. Sorts can only be a subsort of one supersort. The reason for this is that sorts are introduced within a particular viewpoint, expressed by the name of the top sort of a sort hierarchy. Within such a viewpoint subsorts are introduced according to a dividing principle; relating a sort to more than one supersort would deny this dividing principle. Therefore, INCA-COM does not allow multiple supersorting. Multiple viewpoints on a domain are expressed by multiple sort hierarchies.

Specification of objects assigns to each object a sort, determining the place of objects in the object world. An object can be an instance of only one sort in a sort hierarchy. However, an object can be specified to be an instance of different sorts, provided these sorts are in different sort hierarchies. In this way, an object is assigned sorts according to the points of view expressed by the names of the top sorts of the sort hierarchies.

Sort hierarchies and specification are the basis for defining three types of inheritance. They are common inheritance, normal inheritance, and structural inheritance. Each inheritance type corresponds to a particular phase in the

modelling process.

Composition allows objects to be composed of other objects. The composition of an object is defined as the set of objects being a part of the object. Such an intensionally-defined set of objects is called a group in INCA-COM. A system is a special kind of composite object. In addition to a composition, a system has a structure, consisting of the influence relations between the elements in the composition, and an environment, interacting with elements in the system. An object world in INCA-COM is an example of a composite object. Its composition is the group of objects representing things in the UOD. An object world is modelled in a declarative manner, with objects, descriptors, laws, and events. A composite object consisting of active objects communicating with each other is an example of a system. The structure of such a system consists of the communication links between the active objects. The behaviour of an active-object system is modelled in a procedural way, allowing task-like steps to be specified.

In the final section of this chapter we have illustrated the modelling concepts of INCA-COM in a model of a conference-registration system. The use of sorts for the representation of natural kinds and role kinds introduces a distinction between essential and contingent object features. A natural object may play various roles in different application domains. Each role a natural object assumes within an application domain attributes additional features to the object, which are only interesting within the viewpoint of that particular application domain.

The construction of a conceptual model following the framework of INCA-COM determines precisely (1) the objects to be represented; (2) the roles in an application domain; (3) the active objects; (4) the communication between active objects; and (5) the behaviour of active objects. The resulting model can be used for (1) understanding an application domain; (2) simulating an application domain; or (3) (partly) automating particular active objects.

Chapter 6

Version management[1]

The complexity of a real-world system being modelled often brings about a model with a large amount of objects. The evolution of the objects participating in the model normally introduces even more objects. Close inspection of such new objects makes clear that they are quite often versions of the original objects. To reduce the number of objects, a growing need arises to keep track of different versions of the original objects. In this chapter we describe the version model within INCA-COM and address the specific problems when manipulating versions of composite objects.

6.1 Versions

Versions introduce a notion of time in a domain to be modelled (Björnerstedt and Hultén, 1989). In the literature, there has not been much agreement on the terminology to be used with respect to versions. As a case in point, Perry (1987) distinguishes three kinds of versions: (1) successive versions following each other in time as a result of small corrections or improvements; (2) parallel versions resulting from providing alternate implementations or different functionality; (3) composed versions resulting from constructing objects from separate components. However, Perry (1987) does not make clear when corrections or improvements are to be considered "small" and when they give rise to alternate implementations. In addition, a parallel version can be revised as well, leading to a successive version of a parallel version.

[1] Parts of this chapter have been published as an article for the European Simulation Multiconference 1990 as *Version Management of Composite Instantiated Objects* (Bakker *et al.*, 1990).

Perry (1987) describes three different levels of version management. At the first level there is no automated version management. For instance, at the level of the operating system Unix no facilities for version management are available. The user is responsible for keeping track of the appropriate versions.

The second level of version management (basic version management) consists of managing versions of text files. Such version management is provided by systems such as the Source Code Control System (SCCS) and Revision Control System (RCS),[2] which operate on top of an operating system such as Unix. According to Perry (1987), these systems support all three kinds of versions, although the support for composed versions is questionable. Of course, the text files under control of the version-management system may contain programs.

The third level of version management is used in systems such as the System Version Control Environment (SVCE) described by Kaiser and Habermann (1983). This environment allows for a much more detailed specification of the objects constituting a composite object than the systems of the second level. In SVCE granularity surpasses the level of files: even small granular, syntactic objects such as data structures and functions/procedures can be distinguished.

In INCA-COM, versions are supported at the object level, i.e., version management takes place at the third level mentioned above. Basically, versions are semantic variations of an object, merely denoting the *existence* of an object's variation. Consequently, a version is in principle not an object, as these variations can simply be attached to the object. However, when coping with a complex, real-world modelling problem (e.g., modelling the domain of a CASE environment), one encounters sometimes the need for versions that have their own particular relations with other versions or objects. For this reason, INCA-COM contains two different approaches for modelling versions of objects: *versions as partial objects* and *version as version objects.*

6.1.1 Versions as partial objects

In the first approach for modelling versions, a version is treated as a *partial object*, because its descriptor space is a subspace of the descriptor space of a full-blown object. Some descriptors are applicable, while others are inapplicable to versions. The inapplicability of some descriptors warrant the consistency

[2]Within SCCS and RCS, the terminology used differs from the one used by Perry (1987). For instance, a parallel version is indicated as a *branch* in SCCS, and a version is called a *revision* in RCS.

of the version space, because they are attached only to the original object. For instance, a version as a partial object has no sort descriptor, because if would be semantically unjustified to have a version with its own sort.

6.1.2 Versions as version objects

In the second approach for modelling versions, a version is initially treated as merely existing without descriptive structure. When description of a version is necessary, the version is promoted to a *version object*. A version object is a full-blown object, allowing them to be described using the descriptors for objects. However, the *values* of some of the descriptors are constrained: for instance, a version object has a sort descriptor, the value of which is constrained to be identical to the value of the sort descriptor of its original object.

6.1.3 Comparison of the two version approaches

The partial-object approach differs from the version-object approach in its treatment of versions. In the partial-object approach, every version is treated as a partial object, whereas in the version-object approach a version is only treated as a version object when a meaningful description can be attached to it.

The advantage of the partial-object approach is the smaller number of alternatives. For modelling versions in the partial-object approach, the modeller has to deal with two alternatives: an object and a partial object. In the version-object approach, a modeller has to choose among an object, a version, and a version object, requiring more modelling skill.

The disadvantage of the partial-object approach is the possible proliferation of (partial) objects, making the overall model less transparent. Further, a partial object is a rather rich concept to represent a version, having the risk of stimulating the modeller to introduce unnecessary descriptors for versions.

The advantage of the version-object approach is the possibility for refraining from describing versions, giving a minimum of overhead (e.g., regarding inheritance).

The disadvantage of the version-object approach is the gap between the two different kinds of versions possible, namely the version entity only (seen as an abstract store of the *variation* of the original object), and the version object (describing a version *as an object*).

6.2 Version management of composite objects

As is known from Section 5.3, we have established a way of describing composite objects. Two characteristics of the composition principle in INCA-COM are that (1) objects can only be composed of other objects (i.e., objects cannot be composed of versions), and (2) the composition of an object is a set of objects, limiting the meaningfulness of particular decompositions of a composite object.

Thus, an object can only be part of one composition hierarchy. Objects constituting other objects are referred to as parts. A composite object is described by HAS_PART relations to its parts. All parts have a PART_OF relation with their hierarchically-higher object. The hierarchically-highest object is called the top object. Both HAS_PART and PART_OF are referred to as composition relations.

Since objects may only have other objects as parts, an object cannot be composed of partial objects, since they lack the status of object by definition. An object may, however, be composed of version objects, which are full-blown objects indeed.

The INCA-COM presented so far supports versioned composite objects, since a composite object is a normal object. Since the parts of a composite object are themselves objects (see book in Figure 6.1), INCA-COM allows versions of composite objects to have versions of the parts as components. Consequently, to establish this possibility (see book-v1 in Figure 6.1) we need to extend the notion of a version of a composite object. Below we describe the semantics of such composite versions and the semantical requirements INCA-COM imposes on them.

Requirement 6.1 *A composite version is a version of a composite object denoting which parts or versions of parts constitute it.*

Hence, composite versions have their own PART_OF relations to parts or versions of these parts. We use the generic term *component* to denote an object or a version having a PART_OF relation to another object or version. A group of components connected by composition relations is called a composition. Furthermore, an object or version A is a component of object or version B if there is a path from A to B consisting of PART_OF relations. The object or version that contains a component is called the *whole* of that component.

In INCA-COM's semantics, a composite version should have a reason to exist. This means that the composition of a composite version should be (1) different from the composition of the original object and (2) different from the composition of other composite versions of the original object. The structural difference between composite versions is reflected in our semantics.

Requirement 6.2 *A composite version should not have the same composition as the composite object itself or any of its other composite versions.*

Different composite versions can result from the creation of new versions of parts, as well as from the addition or the deletion of components. Thus, we allow versions of composite objects to have other (or more) components than the object itself has, or to lack any originally participating components. The figures in the Sections 6.3 and 6.4 provide examples of this situation.

Requirement 6.3 *A part of a composite version is not necessarily part of the original composite object.*

Requirement 6.4 *A part of a composite object is not necessarily part of all composite versions of that composite object.*

As stated in this section, we maintain that an object cannot be part of more than one composite object. In contrast with Kim *et al.* (1987), we believe that a version of a composite object is closer to the idea of a configuration, describing which specific versions of the parts constitute this version of the composite object. This is reflected in the following requirement.

Requirement 6.5 *Objects or their versions can be a component of several versions of one and the same composite object.*

Requirement 6.5 is in line with the Reuse Approach advocated by Alderson (1989), based on developing new releases of a software system not by changing the old release of the system, but by reusing parts of older releases of that system. Alderson believes that this approach is not widespread because of the impositions on the performance of the object-management system.

Requirements 6.1 to 6.5 are in accordance with the two version approaches in INCA-COM presented in Section 6.1. In the partial-object version model, a version has a restricted set of descriptors, which may contain compositional descriptors. In the version-object model a version is a descriptor of an object, merely denoting existence of a version. A version can be promoted to a version object, offering all descriptive possibilities for full-blown objects.

6.3 Operational semantics of composite versions

In this section and in the following one we specify the semantics of the operations on composite versions. Our example is a book modelled as a composite object. The book consists of two chapters; the first chapter is made up of two sections. The sections and the second chapter are specified as simple objects (i.e., as non-composite).

A clear exposition of the subject addressed below is not possible without giving a series of illustrated examples. For this purpose we introduce a graphical notation for composite objects and their versions. Figure 6.1 describes the book using the graphical notation.

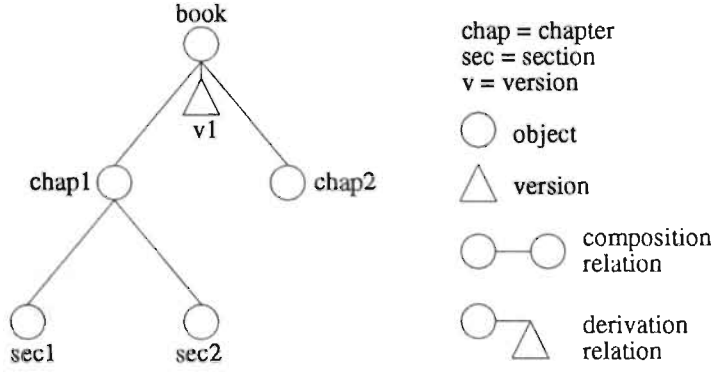


Figure 6.1: Graphical notation of a composition containing versions.

6.3.1 Addition of a version

In general, the creation of a version of an object participating in a composition does not necessarily imply that the version should become a component of a composite version. We state explicitly that this is at the user's decision. Below we treat the addition of a version to (a) a non-composite part, (b) the top object, and (c) a composite part.

(a) Addition of a version to a non-composite part

Suppose a user creates a new version of section 2, identifiable as sec2-v1. When he specifies that this newly-created version should be a component

of a new composition, a new version of chap1, the parent object, should be created (chap1-v1), having sec2-v1 as a component. Recursively, this process is invoked for the new version chap1-v1. Again, the user has to specify whether this new version should be a component of a hierarchically-higher version. This recursive process stops when the user denies inclusion of such a newly-created version in a hierarchically-higher composition or when the process reaches a version of the top object of the composition. The process is depicted in Figure 6.2. A user should also have the opportunity to specify directly the inclusion of a newly-created version in a version of the top object, causing the creation of versions at the intermediate levels.

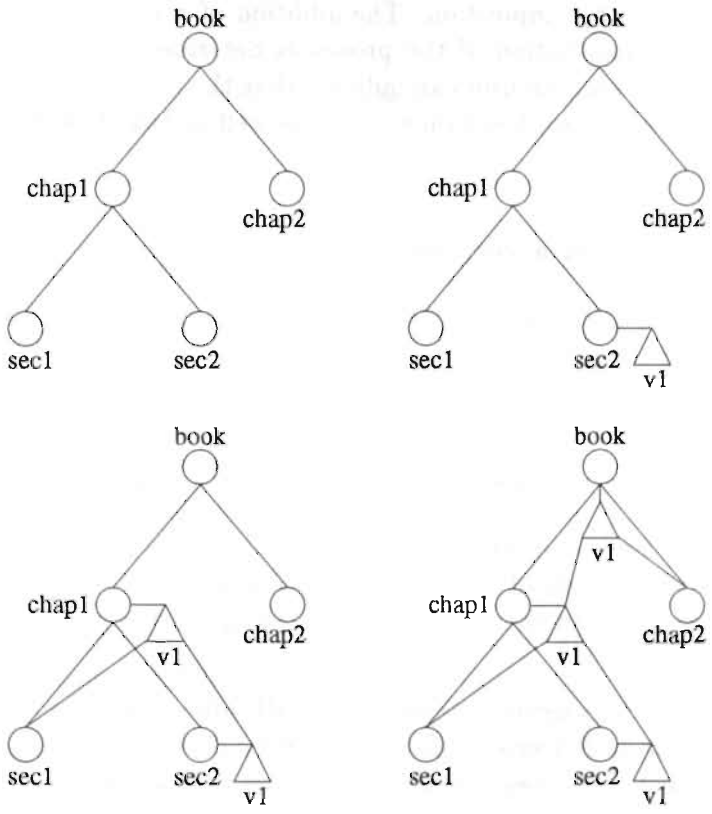


Figure 6.2: Addition of a new version of a component (sec2-v1).

(b) Addition of a version to the top object

New versions of the top object can be created in the same way as versions of parts. When a user specifies that a new version will be composite, he has to indicate which components (objects or versions) will partake in this composite version. Referring to Requirement 6.2, this composite version is different from each of the other existing versions at that moment in time.

(c) Addition of a version to a composite part

The addition of a version to a composite part can be viewed as a combination of two separate cases. First, the composite part has a role as part in a hierarchically-higher composition. Second, the composite part has a role as top object of a (sub)composition. The addition of a version to such a part is governed by a combination of the processes described in Section 6.3.1a and 6.3.1b. Summarizing, the user can indicate that the new version will be a component of a composition (Section 6.3.1a), as well as that it will be composite (Section 6.3.1b).

6.3.2 Deletion of a version

We have divided the treatment of deletion of a version into (a) Deletion of a version of a non-composite part, (b) Deletion of a version of the top object, and (c) Deletion of a version of a composite part.

(a) Deletion of a version of a non-composite part

Specifying this operation, several algorithms are possible. The following algorithms reflect our train of thought, eventually leading to the preferred algorithm, consisting of an enhanced link-redirection mechanism.

Algorithm 1: Recursive deletion of all hierarchically-higher composite versions Because addition of a version of a part may lead to the (recursive) creation of new versions of all hierarchically-higher objects, deletion of a version of a part should cause the (equally recursive) deletion of all versions it is a component of. Further, every component of the deleted versions should be deleted as well, unless it is a multiple component, that is, if it is a component of several wholes (necessarily within one composition).

When this principle is applied, deletion of a newly-added version of a part will result in the original composition (see Figure 6.3). This seems trivial, but further scrutiny of various possible version-management schemes reveals that this is not always the case. A special case occurs when a version is a multiple component, on which occasion the version cannot be deleted directly.

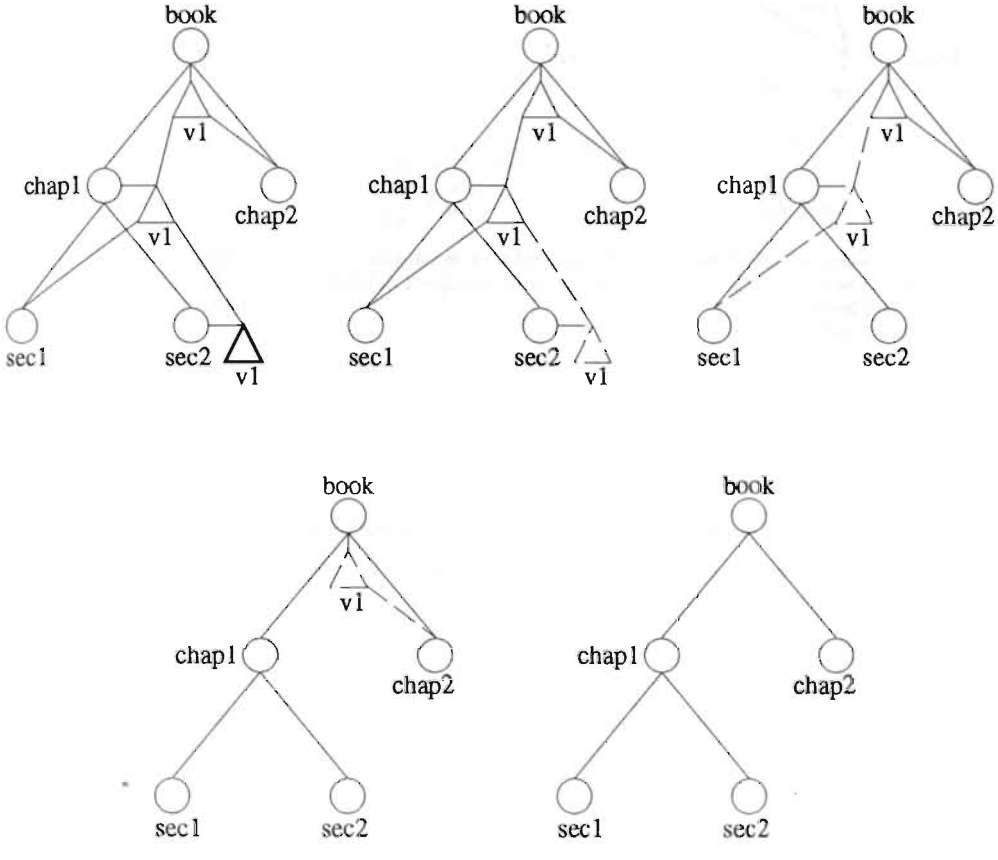
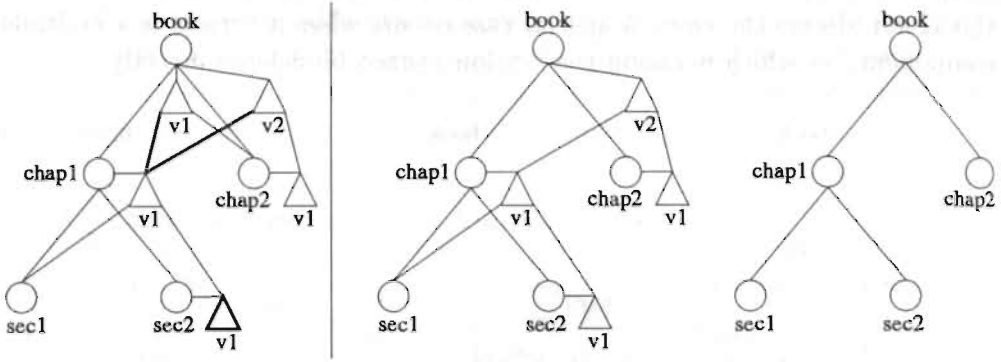


Figure 6.3: Algorithm 1: Recursive deletion of hierarchically-higher composite versions.

Further, the semantics of the PART_OF relation imply that when one of the wholes (that contain the version) is a multiple component, the version itself must also be considered as a (indirect) multiple component. In Figure 6.4a, sec2-v1 is an illustration of an (indirect) multiple component. Following algorithm 1, the only way to delete this component is a one-by-one deletion of all but one composite versions the component partakes in. Finally, the intended

deletion of the component can take place. Figures 6.4b and 6.4c illustrate this process.



a. sec2–v1 as component of two composite versions of book: book–v1 and book–v2. Hence not deletable.

b. sec2–v1 now in only one composition (book–v1 deleted)

c. sec2–v1 deleted book–v2 deleted chap2–v1 also deleted.

Figure 6.4: A component in two composite versions.

Algorithm 2: Deletion of composition relations Figure 6.4 shows that the process of recursive deletion of Algorithm 1 has the drawback of also deleting chap2-v1. The algorithm can be improved. By deleting only the composition relation to this version, the singularly-referenced version remains undisturbed. Figure 6.5 shows the situation in which a version is not deleted, although the composition of which it was a component disappears. The version remains, so it may be used in another version of the composition created afterwards.

Algorithm 2 still has a disadvantage. If sec2-v1 is deleted, chap1-v1 and book-v1 are deleted too, while chap2-v1 is preserved. The deletion is reasonable for chap1-v1, because its right of existence (the presence of sec2-v1) has been deleted. However, the deletion of composite version book-v1 seems unreasonable, because it still has a right of existence, being its component chap2-v1.

Algorithm 3: Enhanced deletion mechanism via redirecting composition links Algorithm 3 does not have the disadvantages mentioned above. Yet, it also has the feature that deletion of a newly-created version results in the previous composition. Looking at Figure 6.4a, there is another possibility

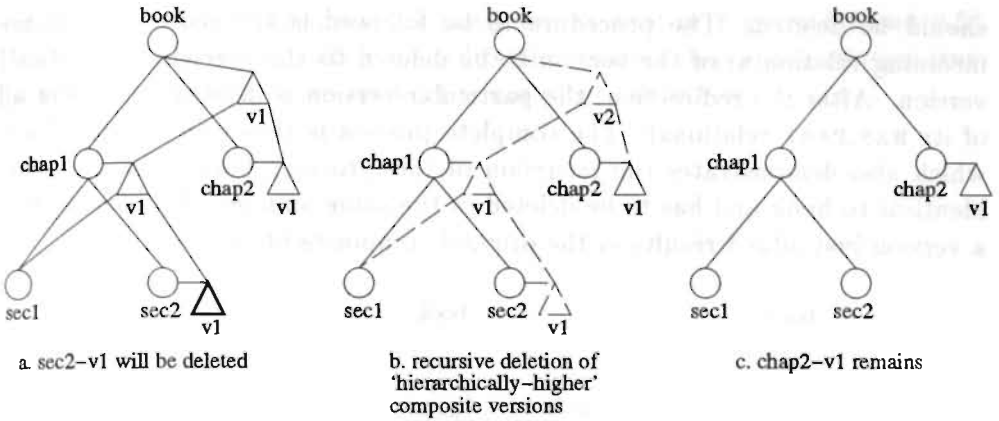


Figure 6.5: Algorithm 2: Upwards deletion of compositions, downwards deletion of composition relations.

to handle the deletion of versions of parts. When, e.g., sec2-v1 is deleted, chap1-v1 is not deleted, but the composition relation to sec2-v1 is redirected to the most recent version of sec2 or sec2 itself, if no other versions are present (as in Figure 6.4). Kim *et al.* (1987) employ an analogous redirection mechanism for the management of composite objects within Orion. Figure 6.6 gives an example of this redirection mechanism.

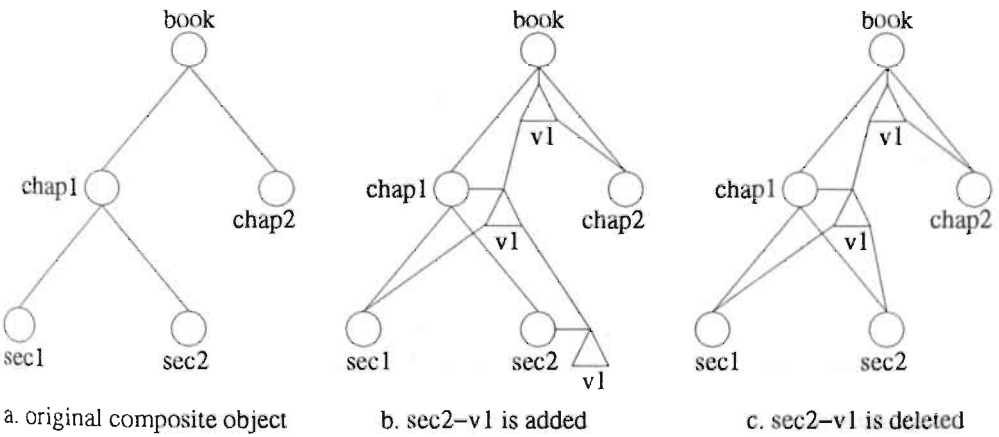
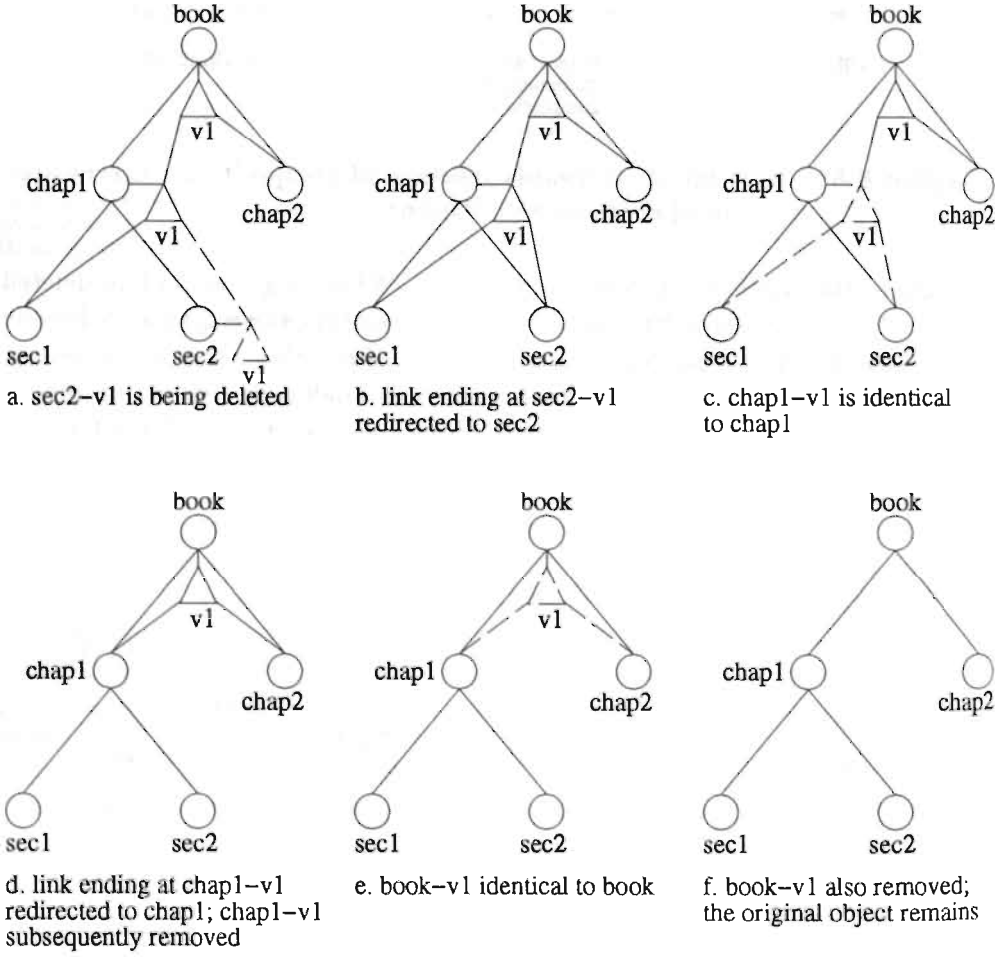


Figure 6.6: Algorithm 3: Redirection of the entering composition relation.

It is clear that the composition chap1-v1 is identical to the composition chap1. For this reason, maintaining chap1-v1 is useless (cf. Requirement 6.2): it

should be deleted. The procedure to be followed is the redirection of the incoming relation(s) of the version to be deleted to the preceding (identical) version. After the redirection, the particular version is deleted (including all of its HAS_PART relations). The complete process is shown in Figure 6.7a–f, which also demonstrates the recursion of this process: book-v1 has become identical to book and has to be deleted in the same manner. Now, deletion of a version just added results in the original composite object.



a. sec2–v1 is being deleted

b. link ending at sec2–v1 redirected to sec2

c. chap1–v1 is identical to chap1

d. link ending at chap1–v1 redirected to chap1; chap1–v1 subsequently removed

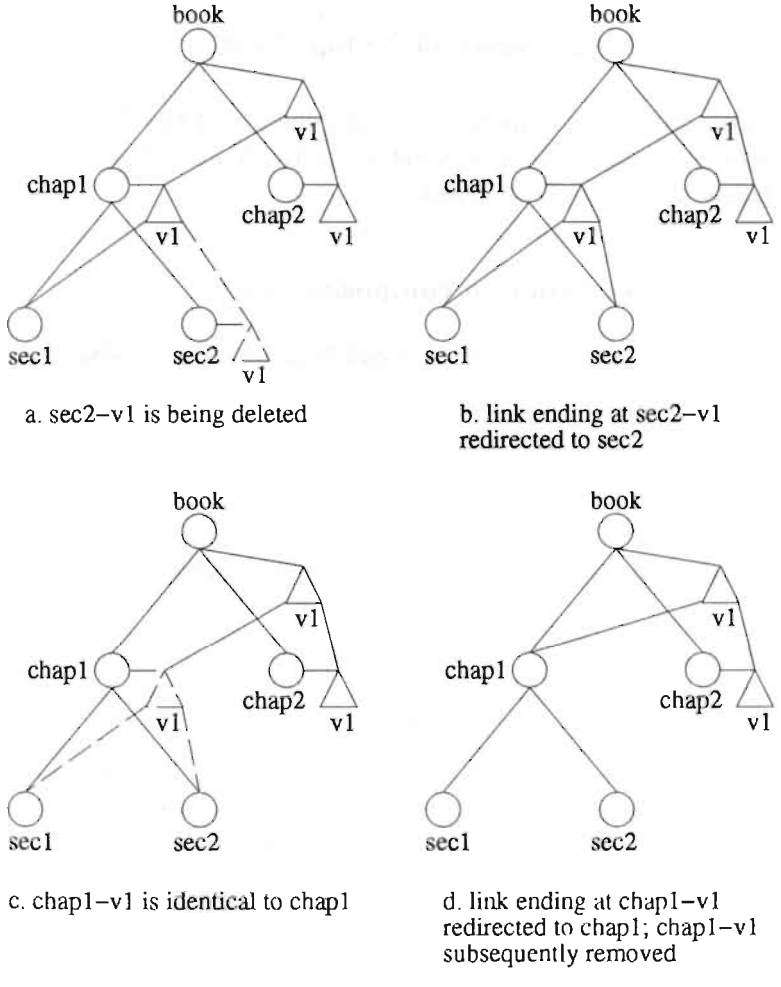
e. book–v1 identical to book

f. book–v1 also removed; the original object remains

Figure 6.7: Algorithm 3: Recovery of the original object after adding and subsequent deleting.

Figure 6.8 shows a more complicated example of the deletion of a version of

a part (note that this example is the same as used in Figures 6.4 and 6.5). Here, the recursion stops after the redirection of the composition relation from book-v1 to chap1-v1, since book-v1 is not identical to book.



a. sec2–v1 is being deleted

b. link ending at sec2–v1 redirected to sec2

c. chap1–v1 is identical to chap1

d. link ending at chap1–v1 redirected to chap1; chap1–v1 subsequently removed

Figure 6.8: Algorithm 3: The recursion stops because book-v1 is not identical to book.

(b) Deletion of a version of the top object

When a version of the top object of a composition has to be deleted, the procedure is as follows:

1. delete the composition relations to its components;
2. delete this particular version of the top object.

From this procedure it is clear that no components of the deleted version are removed, but only the composition relations to them. Therefore we can call such a deletion an "explosive" deletion.

(c) Deletion of a version of a composite part

The deletion of a version of a part is a combination of two separate cases: (1) the version plays a role as a component in a hierarchically-higher composition, and (2) the version plays a role as version of a top object of a (sub)composition. These cases are described in the Sections 6.3.2a and 6.3.2b.

6.4 Operational semantics of composite objects

Here we address the semantics of the operations a user may apply to parts of composite objects. We treat the addition of a new object to a composition, and the deletion of a part of a composite object.

It is the user's responsibility to specify whether the introduction of a new part or the deletion of an existing part is consistent with the nature of the composition. If the nature of the composition is jeopardized, the original composition has to be copied in order to preserve the composite object as it was before.

6.4.1 Addition of an object to a composition

The addition of a new part to a composition leads to the creation of a new composite version of the object it will partake in. This causes one of the procedures described in Section 6.3.1 to be invoked. Figure 6.9 illustrates this situation.

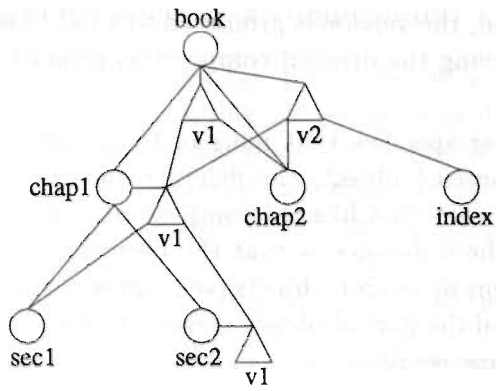


Figure 6.9: Addition of a new part (index) to a composition.

6.4.2 Deletion of an object of a composition

The deletion of an object can be divided into three distinct cases (cf. Section 6.3.2): (a) the deletion of a non-composite part, (b) the deletion of the top object, and (c) the deletion of a composite part.

(a) Deletion of a non-composite part

The deletion of a non-composite part can result in three different situations, depending on the answers to the following two questions:

1. are there partial-object versions or version objects of the object to be deleted?

2. if so, would the user like to impose the role of the object on the oldest of the remaining versions?

First, when there are no partial-object versions or version objects of the part to be deleted, the part and its entering composition relation are straightforwardly deleted (see Figure 6.10a). No recursive deletion of embedding objects takes place, so there is no analogy with the deletion of versions of parts (see Section 6.3.2).

Second, when the part to be deleted has partial-object versions or version objects and the user specifies that one of these versions must play the role of

the part to be deleted, the version is promoted to a full-blown object, replacing the old part and leaving the original composition intact (see Figure 6.10b).

Third, when the user specifies that none of these versions should take over the role of the associated object, the deletion ends up with a composition, without the deleted part, just like the composition remaining in the first case described above. The difference is that the user is still able to promote the partial-object versions or version objects (see Figure 6.10c). If the user chooses not to promote any of the partial-object versions or version objects, a situation similar to the first one results.

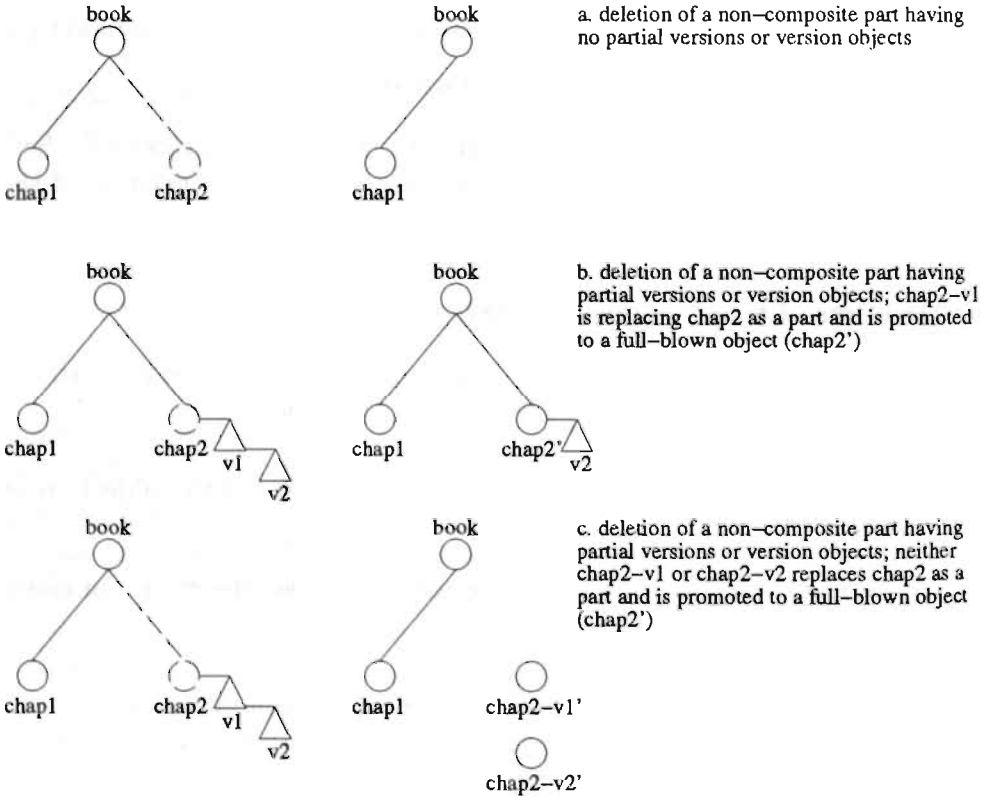


Figure 6.10: Deletion of a non-composite part.

(b) Deletion of the top object of a composition

One might believe that deletion of the top object of a composition should cause the entire composition including all its parts to disappear. However, a composition of objects merely describes the composite structure between these objects, which have an existence of their own. Therefore, the effect of the deletion of a top object is defined as follows:

1. The deletion of the merely existential versions of the top object.
2. The possible copying of the HAS_PART relations to a version that will take the place of the top object. This depends on the same two questions as in Section 6.4.2a. If both questions are answered affirmatively, the composite structure of the top object is not deleted, but copied into the version that will be the new top object. Otherwise, the top object is deleted without its composition relations being copied, and if it has versions as mentioned in the first question, then the user is asked if he wants these versions to be promoted to separate objects. Objects created in this way are not part of any composition at that moment.
3. The deletion of the top object.
4. The parts of the deleted top object are left untouched.

The deletion of composite objects described here only deletes the composition relations between the top object and its parts. The parts themselves are not deleted. We believe there should also be a more destructive operation which deletes an entire composite object, including all its components. The former operation resembles the explosion of an object into pieces and therefore could be called the *explode* operation, while the latter in fact really deletes the composite object and could be called the *delete* operation.

(c) Deletion of a composite part

When dealing with the deletion of composite parts one has to consider two different aspects:

1. the role the part is playing as part of the composition, and
2. the role the part is playing as top object of a (sub)composition.

With respect to the second role, the effects have been described in Section 6.4.2b. One should combine these consequences with the effects the operation has with regard to its first role.

We start with the most simple case: the object to be removed has no partial-object versions nor version objects. As we described in Section 6.4.2b, deletion of such an object results in a situation wherein its parts persist as independent objects, while the object itself is deleted together with its top-object role as mentioned above. Consequently, its role as part should be abandoned, so the object is removed, as well as its PART_OF and HAS_PART relations.

The other cases can be treated in the same way as described above. Therefore, we conclude that the current operation has the same semantics as the operation described in Section 6.4.2b. When one of the versions of the part to be removed takes over the role as top object of the subcomposition, it also takes over the role as part of the embedding composite object.

6.5 Chapter summary

Versions introduce a notion of time in a domain to be modelled. In an object model, versions are variations of objects. Version management is especially important for composite objects, because of the structure of the relations between the composite object and its part objects. The creation of new versions of part objects may induce new versions of the composite object to be created.

INCA-COM contains two approaches for modelling versions. The first approach treats versions as *partial objects*. The descriptor space of a partial object is a subspace of the descriptor space of a full-blown object. The second approach treats versions, when it is necessary to describe them, as *version objects*. A version object is a full-blown object, with contraints on some of the values of the descriptors (e.g., the sort descriptor).

We have focussed on a mechanism for version management of composite objects. We have constructed a composite-version mechanism which gives a natural result when deleting a version just added, namely the original composition. Moreover, the devised version mechanism is capable of coping with more complicated compositions and remains semantically justified. Comparing Figure 6.7 to Figure 6.3 we see that, although the mechanisms of version deletion differ, the results are identical, being the original composite object before addition of sec2-v1.

Chapter 7

Modelling objects using INCA-COM[1]

In this chapter, we illustrate INCA-COM's possibilities for modelling objects in a natural way, i.e., at a conceptual level for communication between human beings. Hence, we describe the development of an object-oriented model of Data-Flow Diagrams (DFDs), introduced in Section 7.1. We model the objects associated with DFDs in Sections 7.2 to 7.4. Section 7.5 provides some conclusions on our modelling effort.

7.1 Data-Flow Diagrams

A Data-Flow Diagram DFD is a presentation of a conceptual model of *processes* and *data flows* between these processes. DFDs play a central role in Structured Analysis, a method for systems analysis stressing the importance of data flowing in an organization (see, e.g., DeMarco (1979)). For the sake of clarity we make a distinction between a conceptual model, which we call a *Data-Flow Model* (DFM), and its visual presentation by a DFD.

The components of a DFD are:

- *circles* representing *processes*,
- *arrows* representing *data flows*,

[1] Parts of this chapter have been published as an article for the conference Technology of Object-Oriented Languages and Systems (TOOLS) (Bakker and Braspenning, 1991b).

- *two parallel horizontal lines* presenting *data stores*, and
- *boxes*, presenting *sources* and *sinks*.

The number of processes modelling an entire information system is often larger than the maximum number of processes a modeller can handle at any one time. To limit the complexity of the presentation, one models large information systems using a levelled set of DFMs. A modeller expands each process in a DFM into a new DFM, thus creating a hierarchy of DFMs. The top-level DFM is called the *context model*, indicating the position of the system with respect to its environment. The DFD in Figure 7.1 shows an example of a DFM consisting of a compiler.

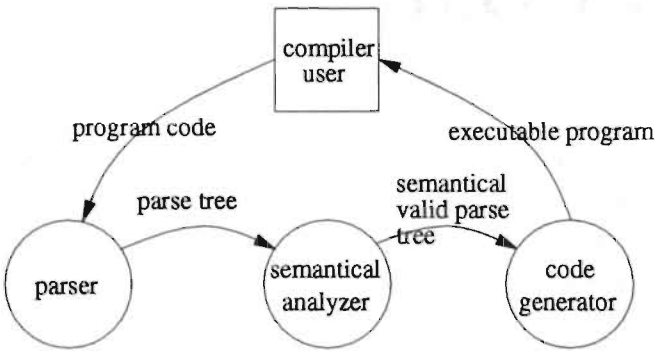


Figure 7.1: An example Data-Flow Diagram of a compiler.

In Figure 7.1, a compiler is modelled as three distinct processes, exchanging data with a compiler user. A compiler user acts both as the source of input, and the sink of output: the user provides the code of a computer program to the compiler, and receives the resulting executable program from the compiler. The compiler is divided into the three processes, parser, semantical analyzer, and code generator. The parser uses program code to produce a parse tree of the code, the semantical analyzer accepts a parse tree to check its semantical correctness, and finally the code generator converts the valid parse tree into an executable program. For brevity, we omit the expansions of the three processes in this model. For a detailed description of Structured Analysis and the use of DFDs, we refer to DeMarco (1979).

A Data-Flow Diagram Editor (DFDE) is a tool to be found in many CASE environments. It is an editor presenting a DFD, enabling its user to create and manipulate the components of the DFD in graphical way.

Our modelling application starts with the description of the DFDE. We then proceed with the modelling of the DFM and its parts. Next, we describe the DFD modelled as a visual display of the DFM. Below we describe these three steps and introduce, when needed, the notation for describing objects.

7.2 Modelling the DFDE

The description of an object always starts with the object's name, which is the identifier of the object. The name of the object is followed by its descriptions, consisting of a set of <*descriptor*, *value*> pairs. The general format for the notation of a <*descriptor*, *value*> is (see also Section 4.1.2):

descriptor_type :: descriptor : value :: value_domain

The descriptor type is one of the descriptor types used in INCA-COM. We use the first letter of each type (in case of ambiguity we use more successive letters), so that we have the following notation for descriptor types: P (property), A (attribute), R (relation), L (link), D (display), and V (version). The description of an object should *at least* specify the sort to which the object belongs, since the sort of an object specifies the place of the object with respect to other objects.
Figure 7.2 gives the partial description of one particular DFDE, named myDFDE. The object myDFDE in Figure 7.2 is an instance of sort DataFlow-

```
Object myDFDE {
 SORT : DataFlowDiagramEditor;
 R :: usedBy : bakker :: User;
}
```

Figure 7.2: A particular DFDE.

DiagramEditor. Further, myDFDE stands in relation (named usedBy) to a particular user (named bakker). This particular user appears as the value of the usedBy relation. The value domain of the relation is User, indicating that the value must be an instance of sort User. We have omitted the description of User.

The descriptor for sorts, such as SORT : DataFlowDiagramEditor is a notational shorthand for the more lengthy R :: instance_of : DataFlowDiagramEditor :: ____, where the value-domain indication '____' stands for all sort objects (i.e., the extension of the domain's metasort, to be explained below). The description of myDFDE relies partly on the description of the sort DataFlowDiagramEditor. It is common practice to model a domain initially by modelling the *sorts* occurring in the domain, rather than modelling particular instances. Because sorts in INCA-COM are objects too, we can describe the sort DataFlowDiagramEditor in the same manner as "normal objects", resulting in the description given in Figure 7.3. The sort descriptor of a sort has as its value

```
Sort DataFlowDiagramEditor {
  SORT : EditorMetaSort;
  SUPERSORT : GraphicalEditor;
}

Sort EditorMetaSort {
  SORT : EditorMetaSort;
}
```

Figure 7.3: The DataFlowDiagramEditor sort.

a metasort (here called EditorMetaSort), which is the sort of all sorts within this particular domain. Because a metasort is a sort, and therefore (again) an object, it has to be an instance of a sort too. To break an infinite recursion, a metasort is an instance of itself. The supersort descriptor is used for the construction of sort hierarchies, which guide the inheritance mechanisms described in Section 5.2. We see that the sort DataFlowDiagramEditor has as supersort the sort GraphicalEditor, which we describe in Section 7.4.

At first sight, a sort in INCA-COM seems similar to the usual notion of a 'type' or 'class' as used in many object-oriented languages. There is, however, an important difference. As we have stated in Chapter 4 and Chapter 5, the whole description of a sort can be divided into an *extensional* part, which describes the structure of potential instances of the sort, and an *intensional* part, which describes the meaning of the sort itself. In Figure 7.3, we have only described (and even incompletely) the intensional part of the sorts. For the

description of the extensional part of a sort we introduce a special attribute with the name instance_descriptors. It is an *attribute*, because in the process of finding the sorts of a domain one is often guided by what some prototypical instances have in common. Such commonality is what makes a sort at first knowledgeable, and attributes are just meant for modelling such entities.

To describe that *an instance* of DataFlowDiagramEditor has a particular relation with the diagram being edited, we extend the description of the sort DataFlowDiagramEditor in the way described in Figure 7.4. The attribute in-

```
Sort DataFlowDiagramEditor {
  SORT : EditorMetaSort;
  SUPERSORT : GraphicalEditor;
  A :: instance_descriptors : {
    R :: editing : _ :: DataFlowDiagram;
  };
}
```

Figure 7.4: The extended DataFlowDiagramEditor sort.

stance_descriptors has as its value a set (indicated by { }) of descriptive entities (possibly with their own values). We have omitted the domain indication for this attribute being the domain of all sets of descriptive entities. The entities listed as the value of the attribute instance_descriptors are inherited by all instances of the sort. For each of the three types of inheritance in INCA-COM (see Section 5.2) there is a different notation for the *value* of the descriptor:

- a description subject to common inheritance is written with an "=" preceding its value,
- a description subject to normal inheritance is written in the normal way,
- a description subject to structural inheritance is written with an "_" as its value.

For example, the relation editing (see Figure 7.4) is structurally inherited by all instances of the sort DataFlowDiagramEditor. Subsequently, one should specify for each instance the value of the relational descriptor, which must be an instance of the sort DataFlowDiagram.

7.3 Modelling the DFM

A DFD is a visual presentation of a conceptual model of processes and data flowing between these processes. To distinguish between a concept and its visual presentation, INCA-COM offers the descriptional type *display* to describe how an object will manifest itself to the outside world. Hence, we describe a Data-Flow Model as given in Figure 7.5.

```
Sort DataFlowModel {
  SORT : ModelMetaSort;
  SUPERSORT : SystemModel;
  R :: usedIn : StructuredAnalysis :: StructuredTechnique;
  A :: instance_descriptors : {
    PARTS : {Process +, DataFlow +, DataStore *,
             Source *, Sink *};
    R :: ExpansionOf : _ :: Process;
    D :: visualization : _ :: DataFlowDiagram;
  };
}

Sort StructuredTechnique {
  SORT : ModelMetaSort;
}
```

Figure 7.5: The DataFlowModel.

Four parts of Figure 7.5 are to be discussed.
First, the relation usedIn indicates a relationship of the *sort* DataFlowModel. This means that the *sort* DataFlowModel is used in the particular technique, not implying, however, that also every instance of DataFlowModel will be used in StructuredAnalysis.

Second, the attribute instance_descriptors contains a parts descriptor, indicating the composite structure of a DFM. The names listed in the value of the parts descriptor are all names of sorts, meaning that an instance of DataFlowModel will have *instances* of the named sorts as parts. The optional specifier after the sort names is used to indicate the cardinality of the specific part: a

'+' indicates that one or more instances are part of the object. Other cardinality specifiers are: '*' for zero or more, a number (e.g., 4) or a subrange of numbers (e.g., 1 . . 4). Hence, in this way one describes a sort by effectively stating that any of its instances has a composite structure, though the sort itself has not.

Third, the relation ExpansionOf indicates that a particular DFM is an expansion of a process on a higher level in the set of DFMs of a system. The ExpansionOf descriptor is structurally inherited by instances of the sort DataFlowModel.

Fourth, we have used a display with the name visualization, indicating that every DFM manifests itself through an instance of the sort DataFlowDiagram. The latter is a visual display of a composite object (a DFM) and therefore an object composed of the *displays* of the particular DataFlowModel parts. Figure 7.6 shows the sort description of two of these parts, Process and DataFlow.

```
Sort Process {
 SORT : ModelMetaSort;
 A :: instance_descriptors : {
  R :: incoming : { _ } :: P(DataFlow);
  R :: outgoing : { _ } :: P(DataFlow);
  D :: visualization : _ :: LabelledCircle;
 };
}

Sort DataFlow {
 SORT : ModelMetaSort;
 A :: instance_descriptors : {
  R :: from : _ :: Process;
  R :: to : _ :: Process;
  D :: visualization : _ :: LabelledArrow;
 };
}
```

Figure 7.6: The sorts Process and DataFlow.

A Process in a DFM generally receives several DataFlows (see Figure 7.1). For

this reason, the value indication of the relations incoming and outgoing is shown as $\{\ \_\ \}$ to indicate that the value is a *set* of DataFlows, drawn from the power set P of DataFlows in the model. A Process and a DataFlow display themselves by means of a LabelledCircle and a LabelledArrow, respectively.

Figure 7.7 shows the three levels (or domains) we have discerned in the modelling process. It is important to see why we have made the distinctions

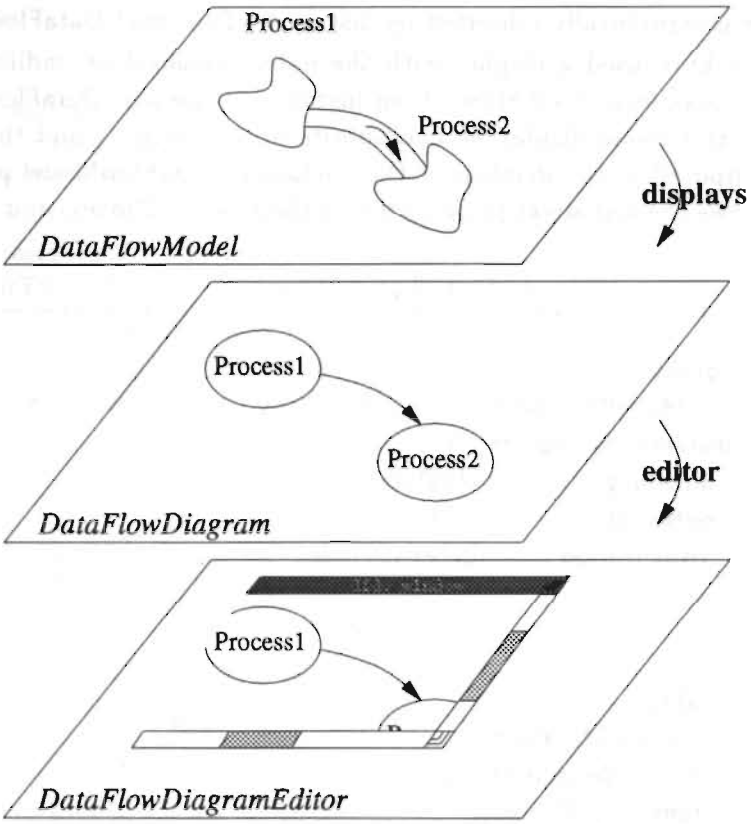


Figure 7.7: Modelling levels.

between a DFDE, a DFD and a DFM. A way of understanding is by looking at the operations one can perform on objects at each level. All operations at a certain level are mapped onto the next *lower* level, e.g., the addition of a process to a DFM is mapped to the addition of a LabelledCircle to a DFD, which is mapped to an operation (or series of operations) for the DFDE. Such a mapping is what makes an editor well-suited: every conceptual operation

should be mapped onto an ordered set of editor operations. However, each level has also operations which are meaningless at the next *higher* level. On the DFDE level, we can scroll the window over a DFD and select circles and arrows; there is no corresponding operation at the DFD level. On the DFD level, we can move or enlarge circles to enhance the lay-out of the DFD; again, there is no corresponding operation at the DFM level.

The operations that can be performed on objects are described by *events*. The distinction in modelling levels is reflected in the possible events of the objects at each level. Figure 7.8 shows some simple events for a DFM. Moreover, a

```
Sort DataFlowModel {
  ...
  A :: instance_descriptors : {
    ...
    events:
      AddProcess
      AddDataFlow
    ...
  };
}
```

Figure 7.8: Events of a DataFlowModel.

DFD has events named MoveCircle and MoveArrow, while the DFDE has events such as SelectObject and DeleteSelection.

7.4 Modelling the DFD

In order to describe the displays within INCA-COM, a set of *visual primitives* of the domain of diagrams is needed. We have examined an existing drawing tool, which allows a user to construct various shapes with relative ease. We have chosen to examine the graphical editor within the desktop-publishing system Interleaf.[2] Investigation showed that Interleaf's basic drawing primitives are:

[2]Interleaf is a trademark of Interleaf Inc.

(1) line; (2) oval; (3) spline; and (4) text. Several other shapes can be composed by a combination of primitives (e.g., a polyline by a number of lines) or by constraining a primitive (e.g., a circle is a constrained oval). As this set of primitives proved sufficient for the construction of a multitude of different shapes, we have adopted it as the basis for the visual-display sort hierarchies. These visual-display sort hierarchies are not to be considered as static structures: the user of INCA-COM can extend them with user-defined displays, in the same manner as introducing subsorts in 'normal' sort hierarchies.

The display visualization of a Process has as value an instance of LabelledCircle. The latter is modelled as a subsort of Circle, with an additional behaviour that displays the name of the Process centered at the origin of the circle. Here we touch an important issue. A LabelledCircle must have some location in order to draw itself. Since this is a generic need of all visual displays, we introduce another metasort, VisualizationMetaSort, which fulfils this need. To make instances of the sorts in this domain inherit from this VisualizationMetaSort, it is modelled as given in Figure 7.9.

```
Sort VisualizationMetaSort {
  SORT : VisualizationMetaSort;
  A :: instance_descriptors : {
    A :: instance_descriptors : {
      P :: origin : (0,0) :: Coordinate;
      B :: draw : ...
    };
  };
}
```

Figure 7.9: The metasort VisualizationMetaSort.

The effect of the description in Figure 7.9 is that all instances of VisualizationMetaSort (i.e., the sorts of the visual-display domain) inherit the value of the attribute instance_descriptors. The value passed to sorts contains a nested attribute instance_descriptors. The value of the nested attribute is inherited by all instances of the sorts.

The introduction of several metasorts divides the entire object world into different *name spaces* corresponding to the different domains (DFM, DFD and

DFDE). Within a name space, there can be no objects with the same name, but within different name spaces this is allowed, as long as it is clear to which name space an object belongs. In the description in Figure 7.9, the value-domain indication Coordinate is a so-called complex type to be specified outside the current model.

Figure 7.10 shows the visual-display sort hierarchies for modelling of visualizations. The gray display sorts are primitive. The white ones indicate visual

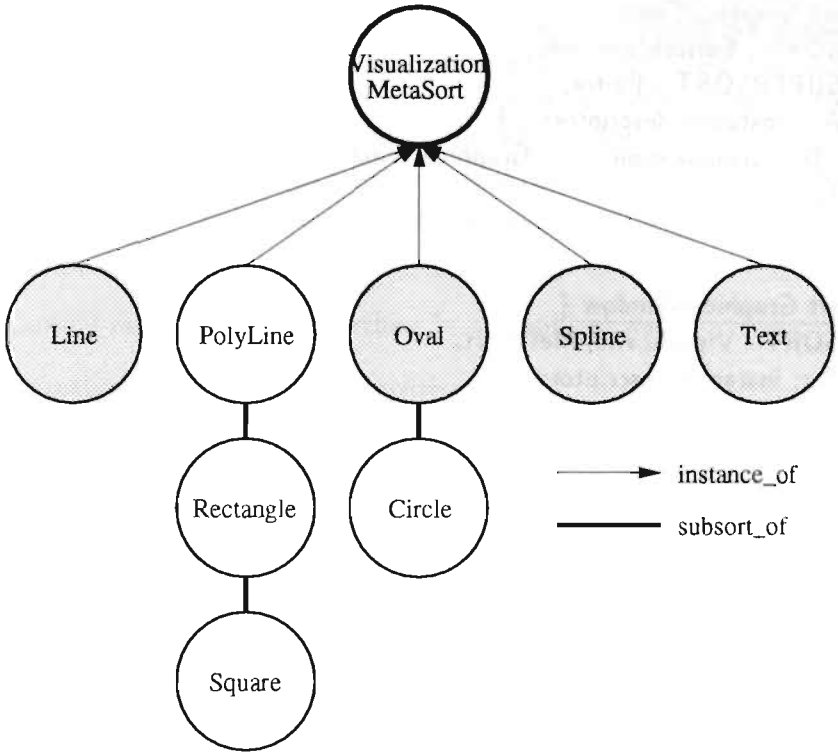


Figure 7.10: Sort hierarchies of visual displays.

displays resulting from either composing or constraining other visual displays. For example, the sort Line is a *primitive* visual display, the sort PolyLine is a *composite* visual display, and the sort Circle is a *constrained* visual display. We note that the sort Square is a constrained composite visual display. For the description of the visual-display sorts, we refer the interested reader to Appendix A.

The display types within INCA-COM can also be used to describe how user-interface objects manifest themselves to the user. An example of such use of a display is given in Figure 7.11, which describes the sort GraphicalEditor (the supersort of the DFDE). The sorts in the parts description of Figure 7.11 must be specified as display sorts (not shown here).

```
Sort GraphicalEditor {
  SORT : EditorMetaSort;
  SUPERSORT : Editor;
  A :: instance_descriptors : {
    D :: visualization : _ :: GraphicalWindow;
  };
}

Sort GraphicalWindow {
  SORT : VisualizationMetaSort;
  A :: instance_descriptors : {
    PARTS : {Border 1, WorkArea 1, Menu 0..1, HorizontalSlider 0..1,
      VerticalSlider 0..1, Closer 0..1, Fuller 0..1, Sizer 0..1
    };
  };
}
```

Figure 7.11: The GraphicalEditor.

7.5 Chapter summary

Designing has always been an important, yet difficult phase in any development process. INCA-COM does not make the modelling process easier; on the contrary, we have seen that the relatively large set of semantically different modelling tools requires a modeller to make much more choices than required when using conventional object models. However, the gain is obvious: a model which is more precise and detailed, and therefore richer in meaning. The effort required to produce such a model is worthwhile: the model is designed

only once, but is usable for many more times. Furthermore, due to the high level of specification and the fact that most object-oriented programming languages adhere to at least part of the same principles as INCA-COM, it should be relatively easy to implement such a design model. The mapping onto a programming language can be automated at least partly.

In our modelling application, the clear distinction between application-domain concepts and their displays enables a clear understanding of the concepts, without being blurred by details for displaying objects. We believe this distinction is beneficial for the addition of knowledge and reasoning mechanisms in order to come to an INtelligent CAse environment. Moreover, the modelling application has guided us in the development of the *basic visual-display sort hierarchies*, which can be used in other application domains too.

In order to specify more complex objects than the ones we have described, one would surely need a sophisticated object design editor. Such an editor should support a modeller in:

- showing multiple sort hierarchies in a graphical way,
- navigating through sort hierarchies,
- creating and editing objects; for instance, as soon as the sort of an object has been specified, the editor should allow to show all the descriptors the new object inherits (with actual and possible values),
- specifying displays in a graphical way; this should be done using a graphical editor.

In Chapter 9 we elaborate on such a tool, called INCATOOL, which is modelled using the INCA principles for application domains and applications.

Chapter 8

An object-oriented model for spreadsheets[1]

We have described INCA-COM as the basis of the next generation of INtelligent CAse environments. Of course, other complex application domains can also profit from the descriptive power of our model for object orientation. In this chapter we treat the application of INCA-COM to the domain of spreadsheets. Through this application we describe a new kind of spreadsheet, and show part of the descriptive power of INCA-COM.

8.1 Existing spreadsheets

The flexibility of spreadsheets is a key factor to its world-wide success, but it is also a well-known disadvantage in maintaining and adapting application models. A spreadsheet modeller possesses some kind of mental model when identifying the information to be represented. Much of this knowledge is left implicit, and thus cannot be used by a spreadsheet program assisting the modeller. Human users can also have trouble identifying the information value in a spreadsheet, if it consists of numbers only.

The usual manner to overcome such limitations is to use cognitive pointers to the intended meaning of the different cells by giving names or text labels to particular rows and columns of the spreadsheet. Although the labels aid the

[1] Parts of this chapter have been published as an article for the European Simulation Symposium 1992 (Bakker and Braspenning, 1992).

user in understanding the meaning of the different speadsheet entries, they are useless for the spreadsheet program.

The new generation of spreadsheets, such as Lotus Improv™, allow the spreadsheet modeller to identify the cells by using *names* instead of tags such as $A1$ or $B2$. Obviously, such a naming scheme for cells is an improvement compared to the conventional scheme, since the need for explanatory labels within the spreadsheet is reduced. However, the application of INCA-COM's object-oriented modelling principles shows that there is more room to adopt new sophisticated models for spreadsheets.

8.2 A new spreadsheet

In order to illustrate how a spreadsheet is modelled we start from the spreadsheet of Figure 8.1. Figure 8.1 contains three regions of interest. We identify the region with Peter and Harm as the *column-label region*, the region with length and age as the *row-label region*, and the lower-right region as the *value region*. Below, we elaborate each of the regions.

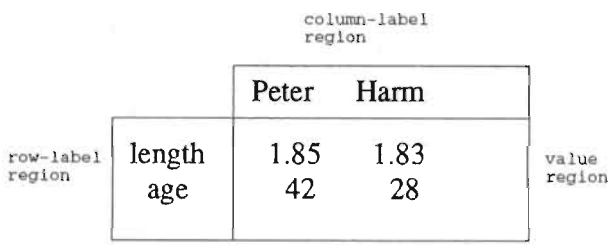


Figure 8.1: The prototype spreadsheet.

Column-label region The region with column labels contains the names Peter and Harm. We consider these names as the names of (simple) objects. Since every object in INCA-COM belongs to at least one sort, the question is: "What are the sorts of these objects?" Both objects are an instance of Human Being, which can be shown by extending the *column-label region* to allow for the *sorts* to be displayed in Figure 8.2. Thus, the objects Peter and Harm are specified as instances of the sort Human Being. Other extensions of our example spreadsheet are also shown in Figure 8.2.

sorts	Animal					
	Human Being		**Dog**		**Cat**	
instances	Peter	Harm	Snoopy	Pluto	Garfield	Tom
length	1.85	1.83	0.95	1.10	0.60	1.15
age	42	28	15	40	10	18

Figure 8.2: The spreadsheet extended with sorts.

The next extension to be made is the introduction of different sorts in the spreadsheet. These sorts are related to each other by means of specialization and generalization relations. This in effect shows the *sort hierarchy* in which Human Being partakes. The supersort of Human Being is Animal, and this is shown by extending the column-label region upward. By its specification, any object is thus connected to at least one sort hierarchy. Continuing in this fashion, and extending the sort hierarchy both upward and downward a spreadsheet like the one in Figure 8.2 is formed. We note that the names in the column-label region, such as Peter, Harm, and Snoopy, denote simple objects (i.e., objects which do not have instances themselves), while the names Animal, Human Being, Dog, and Cat denote sort objects. This distinction is stressed by using bold typeface for sort objects, and normal typeface for simple objects.

Row-label region The next step is to consider the entities along the vertical side of the spreadsheet. The entities in the row-label region are viewed as *descriptors* for the objects along the top side. By means of *descriptional reflection* (elaborately treated in Chapter 9), the row labels can also be extended in a similar way as the extension of the column labels. An example is given in Figure 8.3. It shows the instances of Figure 8.2, which are described using the descriptors length, age, ssn (social-security number), father, and colleague. The names property, attribute, relation and link refer to the descriptor types, used in INCA-COM to specify the semantically different aspects of each object description. Each descriptor is associated with its descriptor type. Thus, property and length are related in the same way as Human Being and Peter are related. Peter is an instance of Human Being, and length is an instance of property.

		Animal					
descriptor types \ sorts / instances / descriptors		Human Being		Dog		Cat	
		Peter	Harm	Snoopy	Pluto	Garfield	Tom
property	length	1.85	1.83	0.95	1.10	0.60	1.15
	age	42	28	15	40	10	18
attribute	ssn	123	456				
relation	father	Joop	Jan				
link	colleague	Harm	Peter				

Figure 8.3: The spreadsheet extended with sorts and descriptor types.

Value region The final step in creating a new spreadsheet consists of modelling the value region. The value region of the extended spreadsheet in Figure 8.3 contains various numbers. These are, of course, the values of the various elements of the description of the objects in the column-label region. Actually, the spreadsheet is nothing more than a way of representing values from an underlying conceptual model, constructed using INCA-COM. The INCA description of the object Peter reads as given in Figure 8.4.

```
object Peter {
  SORT : Human Being;
  P::length : 1.85::RealNumber;
  P::age : 42::NaturalNumber;
  A::ssn : 123::SSN;
  R::father : Joop::HumanBeing;
  L::colleague : Harm::HumanBeing;
}
```

Figure 8.4: The INCA description of Peter.

Depending on the type of descriptor, the value cell in the value region contains a value from a particular *value domain*. For properties and attributes such as length, age and ssn, the value has to be taken from a particular *primitive* value domain (comparable to a *type* in a traditional programming language).

However, the numbers in the value region for the descriptors length, age and ssn are interpreted according to the described value type in the reflexive (again object-oriented) description of the corresponding descriptor.

For relations and links such as father and colleague, the value should be another object in the application domain. Thus the value of, e.g., the relation father of Peter refers to Joop, which is another object in the application domain. An object which acts as the value of a descriptor in the value region is shown with a box around it (see Figure 8.3). Such objects can be made visible, e.g., by clicking on them.

8.3 New concepts

The spreadsheet of Figure 8.3 is a starting point for considering the semantic tools defined in INCA-COM. Not all semantic tools have been shown in this spreadsheet. Below, we introduce them all, illustrating them with an example.

Specification Any cell in the value region contains an element of a particular value domain determined by the corresponding descriptor. Each object (shown along the top side in Figure 8.3) is an instance of at least one sort, which can be shown as well.

Specification is an important new tool in modelling and programming. Besides the usual imperative, functional and logical ways of modelling and programming, the use of specification means the use of a virtual machine attached to the specification hierarchy (i.e., objects belonging to a particular sort). The virtual machine distributes descriptions which are valid for different objects all belonging to a particular sort, thereby enhancing the cognitive economy of the necessary descriptions to be entered by the modeller. The virtual machine may be used also for keeping an eye on application constraints which should be satisfied at all times during a modelling effort.

In INCA-COM each descriptor is itself considered as an object. The descriptors length, age, ssn, father, colleague, shown in the row-label region of Figure 8.3, are objects, with their descriptor types displayed on their left side. The possibility to treat descriptors themselves as objects is a form of *language reflection*, which we treat more elaborately in Chapter 9. Since the descriptors are objects, they can be viewed in the same manner as the objects Peter and Harm. In this way, the descriptors become an application domain on their own, as shown in a separate spreadsheet in Figure 8.5. This application domain is

a basic one, since it reflects the ontology of the modeller, i.e., the basic distinctions which appear to be important for describing any other application domain. A point to note is that in the reflexive spreadsheet the descriptor-

	metasort	PROPERTY			
		Length		Age	
	sorts	HumanLength	DogLength	HumanAge	DogAge
property	minValue	0	0	0	0
	maxValue	3.00	2.00	150	40
	unit	meter	meter	year	year

Figure 8.5: The descriptors treated as objects.

type property is a so-called metasort, having here only two instances, namely Length and Age. These instances are themselves application sorts. They are instantiated during modelling of an application domain.

In Figure 8.5 the metasort named property is shown in bold capitals. (Sorts and instances are shown as previously explained in Section 8.2.) By modelling Length and Age as sorts, the modeller can introduce subsorts, specializing their descriptions. Thus, subsorts, such as HumanLength and DogAge might be introduced, limiting the value the descriptor can have between 0 and 3.00 meters, and 0 and 40 years, respectively (the more generic Length and Age should permit large values to allow generic types of age to be modelled). The result of introducing such subsorts is shown in Figure 8.5 as well.

The possibility to introduce new subsorts in the domain of the descriptors is called *accommodation*. In this way, a (meta)modeller can adapt the medium of representation, ameliorating the *assimilation* of domain knowledge. The terms accommodation and assimilation are adopted from Piaget (1972). For instance, the introduction of the sort HumanAge makes it possible to represent the model of Figure 8.3 more accurately. In this model, all instances have the same descriptor (age), whereas the uses of HumanAge and DogAge for the instances of Human Being and Dog are more specific, allowing for more semantic checks on the model.

Generalization / Specialization The sorts on both sides of the spreadsheet are related to form a so-called sort hierarchy. Such a hierarchy is shown in Figure 8.3, viz. the Animal hierarchy with subsorts Human Being, Dog,

and Cat. Figure 8.5 also contains two sort hierarchies viz. the Length hierarchy with subsorts HumanLength and DogLength, and the Age hierarchy with subsorts HumanAge and DogAge.

Analogously to specification the use of generalization/specialization means the use of a virtual machine which distributes descriptions from sorts to subsorts (when needed), thereby relieving the modeller of the effort to specify identical descriptions too many times. In essence, the sort hierarchy allows a form of reasoning, namely subsumptive reasoning, to be active during the modelling effort. Due to this kind of reasoning the user of such a new spreadsheet is able to reason what the consequences will be of particular changes in the sort hierarchy.

Relation and link The semantic descriptor type *relation* is used to relate two objects. In the spreadsheet the *father* relation in Figure 8.3 illustrates this. The spreadsheet also shows a *link* descriptor (colleague), used for a more *ad-hoc* connection between objects.

The explicit introduction of the types *relation* and *link* prevents the spreadsheet user from entering values that refer to something else than objects of the intended application domain. Hence, again, such explicitness supports the user in capturing the semantics of the object descriptions. In addition, a spreadsheet editor would allow to show the user an overview of already introduced application objects from which a particular value for a relation or link of an object may be chosen.

Display An INCA object has displays, describing how the object is presented to the outside world. In this context, a display models the way the object is shown within the spreadsheet (e.g., iconized, textual, or as a bitmap image). An object may also be made to display itself in an audible way.

Behaviour The possible activity of an object is described by its behaviour. Every behaviour can be shown by a push button. The model user manipulating the spreadsheet can push such a button to activate the corresponding behaviour.

A spreadsheet with displays and behaviours comes near to present-day hypertext possibilities, except that the use of triggers (such as buttons, active words, etc.) is now completely determined by the chosen objects of the application

domain. Thus, all hypertext possibilities are here under the control of a common object-oriented framework providing hypertext-like *reactivity* through the use of particular behaviours attached to any object.

Composition As an example of a composite object, we consider the spreadsheet of Figure 8.6. Analogously to the above, the column-label region contains

	sorts	Company			
	instances	myCompany		yourCompany	
	parts	dept1	dept2	dept1	dept2
property	revenues	42	52	82	94
	costs	20	20	20	34
	profit	22	32	62	60

Figure 8.6: Composition in a spreadsheet.

the sort Company. Two of its instances are myCompany and yourCompany. They are so-called *composite objects*, i.e., objects which have other objects as their parts. The parts of the companies are department1 and department2. The composition of the companies allows them to be viewed by using the department as the focus. The result of viewing the companies by their departments is shown in Figure 8.7. We remark that the information presented in the Figures 8.6 and 8.7 is the same. The difference is the way of presenting it. In Figure 8.6 the numbers for revenues, costs, and profit are presented per company, while in Figure 8.7 they are presented by focussing on the departments (parts) of each company.

Versions An INCA object can have *versions*, which are variations of the object. In the spreadsheet of Figure 8.8, each department is shown to have three versions, namely 1988, 1989, and 1990. Similar to using the composition of objects for changing the focus, the versions of the objects can be used to view the model in distinct ways. Figure 8.9 shows the result of using the versions 1988, 1989, and 1990 as the primary focus on the companies. Thus, the versions are used in Figure 8.9 to view the properties revenues, costs, and profit per department, but even viewing them per company can be accomplished easily.

	sorts	Company			
	parts	department1		department2	
	instances	myCmy	yourCmy	myCmy	yourCmy
property	revenues	42	82	52	94
	costs	20	20	20	34
	profit	22	62	32	60

Figure 8.7: Focus on object parts.

	sorts	Company											
	instances	myCompany						yourCompany					
	parts	department1			department2			department1			department2		
	versions	1988	1989	1990	1988	1989	1990	1988	1989	1990	1988	1989	1990
property	revenues	42	28	30	52	54	56	82	99	98	94	96	54
	costs	20	18	17	20	20	20	20	33	68	34	30	30
	profit	22	10	13	32	34	36	62	66	30	60	66	24

Figure 8.8: Versions in the spreadsheet.

Multiple sort hierarchies INCA-COM does not allow sorts to have multiple supersorts. The reason for disallowing multiple inheritance is that a sort hierarchy should provide one point of view on a domain, according to which subsorts are introduced (see also Section 5.1).

Therefore a sort is not allowed to partake in another hierarchy, since this would be a mixing of different points of view, and would ignore the different points of view each of the hierarchies expresses. Moreover, since every sort

	sorts	Company											
	instances	myCompany						yourCompany					
	versions	1988		1989		1990		1988		1989		1990	
	parts	dept1	dept2	dept1	dept2	dept1	dept2	dept1	dept2	dept1	dept2	dept1	dept2
property	revenues	42	52	28	54	30	56	82	94	99	96	98	54
	costs	20	20	18	20	17	20	20	34	33	30	68	30
	profit	22	32	10	34	13	36	62	60	66	66	30	24

Figure 8.9: Focus on versions.

hierarchy allows for a form of subsumptive reasoning, the points of view should be clearly distinguished. However, it is allowed for an object to have multiple sorts. Thus, an object can be specified from different points of view, which is a quite natural way of modelling.

In the spreadsheet model, different points of view can be shown by using three-dimensional visualization with different planes. Each hierarchy in which an object partakes is shown on a different plane. The planes are ordered in the third dimension, and the user can select the point of view (plane) of his interest.

8.4 Chapter summary

The application of INCA-COM to the general domain of a spreadsheet shows that INCA-COM's semantic tools can be used to express the different aspects and relations of objects. In order to have full benefit of a spreadsheet manipulating program it is necessary to regard the conventional row-column display as merely a *view* on a model. By bringing semantics into this underlying model, the spreadsheet program is able to manipulate the model views in ways not possible with current spreadsheet programs. In this chapter we have shown how INCA-COM is used for the description of the underlying model.

We have divided a spreadsheet into three different regions: the *column-label region*, the *row-label region*, and the *value region*. The column-label region contains the *names of objects*. These objects are assigned to their *sorts*, which are also located in the column-label region. The sorts can be related to each other, thus forming *sort hierarchies*, resulting in additional structure in the object world.

The row-label region contains the names of the *descriptors* for objects in the column-label region. The six descriptor types in INCA-COMare *property*, *attribute*, *relation*, *link*, *display*, and *version*. The relation type comprises at least three presupposedly important relations, viz. the *instance_of*, the *subsort_of*, and the *part_of* relations, which are used for specification, specialization, and composition, respectively. These three relations structure an application domain, so that the spreadsheet program can assist a user with different views on the model.

The value region contains the values of the descriptors in the row-label region. The values have to be taken from a particular value domain. For descriptors from the types property and attribute the values have to be from a primitive

value domain, and for descriptors from the types relation and link the values have to be objects within the application domain.

The semantic categorization of descriptors furthers easy *assimilation* of domain knowledge. In addition, INCA-COM uses descriptional reflection: every descriptor and its type are treated as objects too. By this reflection a modeller may extend interactively the modelling tools of INCA-COM by creating additional descriptor types and associated application descriptors (*accommodation*). The use of reflection will be elaborated in Chapter 9.

In addition to modelling spreadsheets in INCA-COM, the new spreadsheet concept appears to be an effective way of presenting information to a user, and therefore we will adopt a similar INCATOOL interface for presenting any object model in whichever domain to a modeller and to other users.

Chapter 9

INCATOOL[1]

In this chapter we describe the tool INCATOOL, which supports the modelling with INCA-COM. The emphasis of the description is on the architecture. INCATOOL's main purpose is supporting the construction of an INCA model based on the structure of the modelling framework of Chapter 4 and the structuring principles of Chapter 5. Modelling with INCA-COM has an iterative and evolutionary nature, and therefore is best supported by an interactive and open-ended software tool, allowing high-level design decisions to be changed or discarded, continuously showing the consequences of the decisions. Appendix B presents an hypothetical session with INCATOOL illustrating the kinds of operations a modeller may perform during modelling.

We describe the requirements of modelling support by considering modelling as a form of knowledge acquisition, during which assimilation and accommodation play an important role. In order to assist a modeller, INCATOOL uses explicit knowledge about the concepts of INCA-COM and their representation by means of language elements. INCATOOL's architecture is based on the reflective representation of INCA concepts and language elements in terms of INCA-COM leading to the concepts of an ontology and a descriptor domain. The latter two are defined as follows:

- an ontology contains the concepts used when modelling with INCA-COM;

[1]This chapter is based on two articles: *Geautomatiseerde ondersteuning van object-georiënteerd modelleren* (Dutch Conference on Applications of Artificial Intelligence 1991, Bakker and Braspenning (1991a)) and *De ontologie en generieke kenmerken van een object-georiënteerde taal voor kennisgebaseerde analyse en ontwerp* (Dutch Conference on Artificial Intelligence 1992, Bakker *et al.* (1992))

- a descriptor domain contains the language elements used for the description of application objects.

The concept of an ontology is based on considering the INCA concepts as objects. The concept of a descriptor domain is based on treating descriptors as objects.

The chapter also provides an integrating view on the relations between ontology, application domain, and descriptor domain. It ends with a description INCATOOL's architecture.

9.1 Requirements of modelling support

Following Piaget (1972), *assimilation* and *accommodation* are the two important aspects of knowledge-acquisition processes as explained in Braspenning (1989a):

- assimilation of fact and rule structures concerns the representation of UOD knowledge by means of partial *matches* with template-like representation structures;
- accommodation of the system's own knowledge-representation capabilities concerns changing the cognitive templates used during assimilation.

If we supply the knowledge about INCA-COM in a reflective manner, i.e., in the form of INCA objects, assimilation of application-domain knowledge and accommodation of the representation language can be performed with INCATOOL. Consequently, we take assimilation and accommodation as requirements for INCATOOL. The two requirements facilitate the construction of an ontology and the determination of a descriptor domain.

Assimilation entails the use of a fixed set of concepts, which act as templates for the modelling of objects in an application domain. In order to support the modeller, INCATOOL must have knowledge about the concepts (and descriptors) available in INCA-COM. This kind of knowledge is required for assisting the modeller during the modelling of a particular application domain and particular application(s).

Accommodation is required to enable a modeller to modify the knowledge about the descriptors *during modelling*. Thus, a modeller can extend (or accommodate) the descriptors used in INCA-COM in order to describe application objects which cannot be represented adequately in in INCA-COM's core.

Stated otherwise, the knowledge base containing explicit knowledge about INCA-COM is used by INCATOOL to support the modelling process (assimilation), and can be extended interactively by the modeller (accommodation).

INCA-COM offers the groundwork for the description of objects in an application domain, such as the conference-registration domain. In INCA-COM a description of an object is usually incomplete: a description should be seen as a *partial* description, which is added to the already existing description of the object. The description process evolves into a final object description. As stated in Chapter 4, each object in INCA-COM is identified by a unique name, taking care of the identity of the object. The object description has the form of an ordered set of ⟨descriptor, value⟩ pairs. A descriptor is identified by a name and is assigned to one of the descriptor types introduced in Section 4.4.

The space of object descriptors and their relations can be seen as an application domain itself, which can be described with INCA-COM as well. The descriptors and their types thus become the subject of description. For the modeller it suffices to specify the semantic type of a descriptor by means of a *descriptor type* indication.

The local description of objects is supported by INCATOOL, offering mechanisms for the efficient distribution of descriptions applicable to various objects. In order to support a modeller, INCATOOL should give immediate feedback by showing the effects of editing operations, such as addition or removal of descriptions. Immediate feedback on modelling decisions is essential for an evolutionary way of modelling.

9.2 The INCA ontology

By introducing an ontology for object-oriented modelling, we do not aim at starting a philosophical discussion on the essence of things or on the study of being. The previous chapters have focussed on primitives such as objects, sorts, properties, attributes, etc. Here, a *technical* concept of ontology, representing a set of concepts to be used in *any* application model is discussed. Based on the concept of an object, our view of an ontology provides the basic distinctions, useful in any modelling effort. After all, modelling presupposes an initial choice of concepts for looking at the world, and expressing a model. The INCA ontology thus does not imply existence of things denoted by its ontological primitives, but it merely provides useful concepts for modelling. The concepts are used for assimilation of application-domain knowledge.

The INCA ontology has been given the form of five sort hierarchies (see Figure 9.1), each of which represents a particular point of view on the basic concept object. The sorts in the ontological sort hierarchies represent concepts, which can be instantiated in any application model. A similar ontological principle is used in KL-ONE by introducing the concept Thing. There, each application concept is either directly or indirectly a *specialization* of Thing. This implies that Thing and its specializations are by definition within the same domain. In INCA-COM, we distinguish between ontological concepts and the application concepts by explicitly creating an ontology.

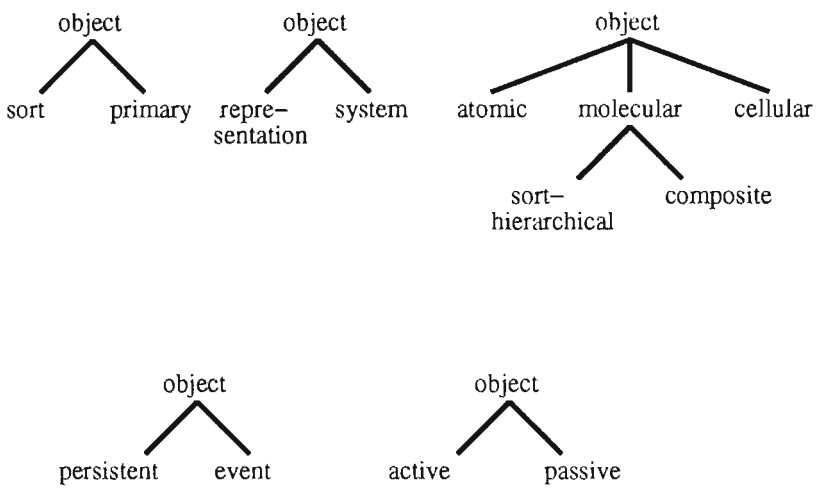


Figure 9.1: The INCA ontology.

The INCA ontology in Figure 9.1 provides distinctions applicable to every object. The ontological distinctions are expressed by means of sort hierarchies. In this respect we follow the guidelines for sort hierarchies set forth in Chapter 5. The names of the ontological sorts are chosen in such a way that they can be read from the leaves to the top of the hierarchy. As a case in point, we mention the hierarchy object with subsorts sort and primary, which are to be read as sort.object and primary.object.

For each object (within an application domain) the modeller is required to specify to which sort in each ontological hierarchy it belongs. This has to be done in such a way that the *most specific* sorts are chosen. Objects for which this specification is not appropriate (e.g., because the modeller wishes to postpone the decision) can be specified to belong to the top sort in the

sort hierarchy under consideration. Later, the object may receive a more specific sort instead of the top sort, i.e., a lower one in the particular hierarchy. Further, the INCA principle that an object may only belong to one sort in the same hierarchy is preserved. This is similar to the use of *partitionings* in the ontology of Cyc (Lenat *et al.*, 1990), which are used to model exclusive divisions of sorts into subsorts. The five ontological distinctions are discussed below.

Sort and primary object The first distinction is made between a sort and a primary object. This distinction results from the decision to describe and treat sorts similar to objects (see Section 5.1). A sort object is a sort with respect to its instances, and an instance with respect to its (meta) sort, but the distinction we make here explicit clearly divides all application objects into primary instances (objects in the application) and application sorts.

Representation and system object The second distinction is made between a representation and a system object. A representation object refers to (is a representation of) an object in the initial application domain, while a system object has been introduced for the description of a representation object. A system object has no reference to an object within the universe of discourse. In a certain way, the distinction between representation and system objects can be seen as the definition of an *application cursor*, which constantly refers to the original application level, while the modeller may be modelling at a different (i.e., higher) level. Such higher-level objects are system objects, while objects at the application level are representation objects.

Atomic, molecular, and cellular object The fact that particular groups of objects are used for constructing composite objects is the basis for a distinction between *atomic*, *molecular*, and *cellular* objects. Atomic objects are objects which are not composite. They are used as the building blocks for more complex objects. Molecular objects are objects consisting of a particular group of objects. A molecular object has the object status. Thus, it has the ability to relate to other objects, and it can be described by means of a set of descriptions.

Two explicit specializations of the ontological sort of molecular objects are the *sort hierarchy* and the *composite object*. A sort hierarchy is defined by a group of sort objects, related by the relation subsort_of. A composite object is defined by a group of objects, related by the relation part_of.

In addition to atomic and molecular objects, another distinction is made into cellular objects. Cellular objects are also defined by a group of objects, related to each other by particular relations. The difference with molecular objects is that a cellular object has the functionality of (sub)system, meaning that there is *interaction* between the parts of the cellular object.

In INCA-COM the group concept is used for modelling sets of objects by means of a predicate (see also Chapter 5). A group is generated by evaluation of the corresponding predicate. Thus, a group is defined by an intensional description of a collection of objects.

A group has no object status, but as we described earlier in Chapter 5 a composite object is an object, described partially by its *composition*. The composition of such an object is in fact a group, defined by a predicate *part_of*: the objects having the *part_of* relation with a composite object are elements of its composition. Because of their lack of object status, groups are not present in the ontology.

Persistent and event object There is no dynamics in an object world, if there is no change. One of the reasons to consider objects is that they are the constant elements within the flux of events in the object world. Objects and events are dual concepts. However, the duality also enables a modeller to consider events as meaningful objects. For instance, in a windows environment, the clicking of a button in a window is an event, resulting in a change in a window object. It may also be useful to consider such an event as an object by describing it in terms of a time instant, a duration, and its relation with previous events. Within the INCA ontology we have therefore made a distinction between persistent and event objects. Persistent objects have a life cycle due to a sequence of events changing their state (see also Sections 4.4 and 4.5). By modelling events as objects, they are considered as instances of particular sorts of events, describing their generic nature.

Active and passive object The fifth ontological distinction concerns the distinction between active and passive objects. Passive objects represent things in the UOD, about which information is represented in the object world. Passive objects are the subjects of discourse of active objects. Events change the state of passive objects. Active objects communicate with each other about passive objects. The actions of active objects correspond to events in the object world. A sequence of actions is called the behaviour of an active-object

system; a sequence of events in an object world is called a process.

9.2.1 INCA-COM as application domain

Since INCATOOL is designed to represent and manipulate objects from a UOD in an application domain, it is appropriate to represent the knowledge about INCA-COM in an object-oriented way. Then, a modeller may use INCATOOL for the accommodation of the medium of representation, in the same way INCATOOL is used in the assimilation of application domains. In fact, this boils down to considering INCA-COM itself as an, albeit particular, application domain. INCA-COM's concepts are thus represented as a collection of objects, which represent the knowledge for using the concepts (Braspenning, 1990).

Figure 9.2 illustrates the idea of representing INCA-COM's concepts as objects in the ontological domain. It shows an application domain containing a sort hierarchy with the application sorts PolyLine, PolyGon, Square, and Triangle. The sorts Square and Triangle have two instances, my_square and my_triangle, respectively. They are related to their sorts by the Instance_of relation, indicated by the solid arrows.

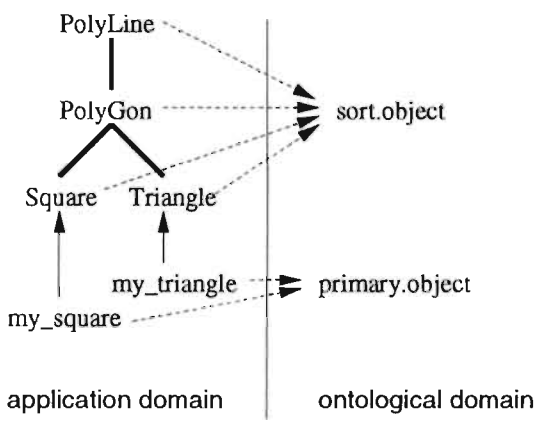


Figure 9.2: Reflection in INCA-COM.

The ontological domain shows the two INCA concepts sort.object and primary.object from the INCA ontology. The dashed lines in Figure 9.2 mark the instantiation relation between the application objects and the modelling concepts in the ontology. The objects my_square and my_triangle are ontologi-

cal instances of primary.object, and PolyLine, PolyGon, Square, and Triangle are ontological instances of sort.object.

Each object in an application domain is an instance of an application sort which is located in a sort hierarchy. Application objects and sorts represent things and concepts in the UOD. The sorts in an application domain have a template-like function, since they may be instantiated in order to represent objects within the UOD. Thus, sorts within an application domain are a conceptual structure, offering a framework of concepts to be instantiated by application objects in order to represent a state of affairs in a UOD.

A similar line of reasoning holds with respect to the relation between INCA models, and a generic (i.e., instantiable) description of INCA concepts. In order to support a modeller during his modelling task, INCATOOL must have some template or generic structure of INCA concepts such that INCATOOL may instantiate these templates to form application models. These generic concepts are located in the ontological domain.

9.3 Descriptor reflection

In order to use the ontological concepts, a language should offer language elements referring to these concepts. Several types of language are possible. A language with a reflective nature (i.e., possessing some reflection, for which see below) is called a reflective language. The common characteristic of reflective languages is the relation between implicit entities in a domain D_n ($n = 0, 1, 2, \ldots$), called the base level, and another domain D_{n+1}, called the meta level (Ferber, 1989). Every domain can serve as a base level for the next higher level, or as the meta level for the next lower level. D_0 can only be used as a base level. We note that reflection has different meanings depending on the context in which it is used. In object-oriented languages the following two forms of reflection can be found (see also Ferber (1989)):

Structural reflection: every entity is considered to be an object; in some object-oriented languages classes are considered to be objects, belonging to a metaclass.

Computational reflection: the specification of the *behaviour* of an object is expressed in the same language used for the description of objects; this approach has been introduced by Maes (1987) and was implemented in 3-KRS, in which language local interpreters (meta-objects) execute the behaviour of each object.

When modelling an application domain with INCA-COM emphasis is on the description of the applition objects and not on the behaviour of the application objects. This implies that computational reflection is not an essential ingredient of our language. As we see it, computational reflection is a specialized form of language reflection, which is defined as follows.

Language reflection: the language elements for describing objects refer to objects of the same type as the objects described by the language.

This definition entails that language reflection contains on the one hand computational reflection and on the other hand descriptor reflection. The focus of INCA-COM is description by means of descriptors. Incorporating language reflection in INCA-COM leads to a specialized form of language reflection, which we call *descriptor reflection*, defined as follows.

Descriptor reflection: the descriptors for describing INCA objects in an application domain refer to INCA objects in a separate descriptor domain.

In INCA-COM structural reflection and descriptor reflection are used in the following way. First, treating a sort as an object is a form of structural reflection. The advantage of this is the possibility to describe sorts in the same way as normal objects. Describing sorts as normal objects has been treated earlier in Chapters 4 and 5. Second, descriptor reflection consists of treating descriptors of objects at the level of an application domain as objects at a higher level application domain, called the descriptor domain. The advantage of descriptor reflection is the possibility to describe descriptors using the same concepts as normal objects and to accommodate the representation language by introducing specialized types of descriptors as explained in Section 9.1.

Figure 9.3 gives an example of a description in which the descriptors of objects in an application domain are treated as objects in the descriptor domain. The description in the application domain contains two object descriptions, namely the objects HumanBeing and Harm. Each object description starts with the indication of the ontological sort(s) to which the object under consideration belongs. The object HumanBeing is specified to be a sort.object. The set of descriptions further indicates that (1) HumanBeing is a subsort of Animal not further described here and (2) its set of InstanceDescriptors contains a descriptor Age meaning that each instance of HumanBeing is at least described by an Age descriptor. Such an instantiated object is the object Harm, specified to be

Descriptor domain

```
sort.object DescriptorType {
  ...
}

sort.object Relation {
  Instance_of : DescriptorType;
}

sort.object Property {
  Instance_of : DescriptorType;
}

sort.object Instance_of {
  Instance_of : Relation;
}

sort.object Subsort_of {
  Instance_of : Relation;
}

sort.object Age {
  Instance_of : Property;
  ValueDomain : HumanAge;
}

sort.object InstanceDescriptors {
  Instance_of : Attribute;
}
```

```
sort.object HumanBeing {
  Subsort_of : Animal;
  InstanceDescriptors : {
    Age
  }
}

primary.object Harm {
  Instance_of : HumanBeing;
  Age : 28;
}
```

Application domain

Figure 9.3: An application domain in which the descriptors are treated as objects in the descriptor domain.

a primary.object. In the application domain it is an instance of HumanBeing, and it has an Age with value 28.

In the descriptor domain, the descriptors of the application domain are further described. The descriptors are Subsort_of, InstanceDescriptors, Age, and Instance_of (note that the values of the relational descriptors Subsort_of and Instance_of (Animal and HumanBeing, respectively), are *objects in the same*

application domain, hence they are not described in the descriptor domain, but in the application domain, which is not shown here). They are described in the descriptor domain as objects. We remark that each of these objects is a sort.object. The reason for this is that they are to be instantiated in the application domain, for every object in whose description they are used. Thus, a descriptor name in the application domain refers to an equally-named descriptor sort in the descriptor domain. The actual descriptional use of such a name leads to the instantiation of the descriptor sort. The instance name of an instantiated descriptor is generated by the underlying naming system; it should not be used by the modeller, but only serves to distinguish between the different instances of the various descriptor types.

The sets of descriptions of the four objects Subsort_of, InstanceDescriptors, Age, and Instance_of contain a description indicating the type of the descriptor. Instance_of is specified to be an instance of Relation, Age is an instance of Property, Subsort_of is an instance of Relation, and InstanceDescriptors is an instance of Attribute. The descriptor domain also shows that the descriptor types, such as Relation and Property, are described as sort objects (in fact, they are meta sorts, since their instances are sort objects), belonging to the sort DescriptorType not further described here.

Referring to the idea of Section 9.1 to consider the space of object descriptions and their relations as application domain we now treat the description of descriptors in the descriptor domain. This enables a modeller to specify the semantics of descriptors. In order to illustrate such a specification, we show how the semantics of the Instance_of descriptor can be specified. In Figure 9.4 the object Instance_of is described. Its ontological sort is sort.object, which means that it is to be instantiated in the application domain, for every object description in which the Instance_of descriptor is used. The type of the Instance_of descriptor is specified by the (self-referential) descriptor Instance_of, relating the descriptor to its sort, Relation. Note that in this special case the description of a descriptor contains a reference to the descriptor being described. This implies, of course, that initially some particular descriptors are present, since it would be impossible to give any description whatsoever without such initial descriptors (bootstrapping problem).

The descriptor type Relation furnishes two descriptors, o and s, which denote the source and target object between which a particular relation holds. These two descriptors are used for the description of the semantics of the Instance_of descriptor.

```
sort.object Instance_of {
  Instance_of : Relation;
  semantics(o,s) : {
    Instance_of(o, primary.object) ∧¬Instance_of(s, sort.object) →
        errorMessage('Only sorts have instances.');
    Instance_of(o, primary.object) ∧ Instance_of(s, sort.object) →
        description(o) += InstanceDescriptors(s);
  }
}
```

Figure 9.4: The semantics of the Instance_of descriptor.

The use of a descriptor implies that its semantics is checked. The semantics of the Instance_of descriptor consists of two rules, which are applied if their preconditions are met:

- if the Instance_of descriptor is not associating an object with a sort object, an appropriate error message is generated;
- if the Instance_of descriptor is actually associating an object o with a sort object s, the description of o is extended with the extensional description of the sort object s (given by the attribute InstanceDescriptors).

This semantics is given in Figure 9.4 as semantics. It takes two parameters o and s specifying the object for which the descriptor is used, and the value specified in the ⟨descriptor, value⟩ pair, respectively.

During the interactive modelling process INCATOOL plays an important role. It should assist the modeller whenever an as yet unspecified descriptor is being used. This can be accomplished by starting a dialogue during which the modeller describes the newly-introduced descriptor. The sort specification of the descriptor object specifies to which descriptor type the descriptor belongs, such as property, attribute, relation, or link.

9.3.1 Descriptors as application domain

The most important modelling activity, is *specification* of objects (see also Chapter 5). By assigning an object to its sort(s), it is assigned its place within the object world. By such a sort assignment, an object receives its initial description given by the extensional description of its sort(s). Besides the primary description of an object by means of sort assignments, INCA-COM contains a set of descriptor types for the semantic categorization of object descriptions. The idea to treat the space of object descriptors and their types as a separate application domain enables a modeller to view the descriptor types as sorts and particular descriptors as instances of these sorts. Earlier, ideas on this subject have been presented in Braspenning (1990); in Chapter 8 we have given an example by showing the modelling of descriptors such as Length and Age. A similar approach has also been employed in the TELOS knowledge-representation system (Mylopoulos, 1992).

The organization of a domain by means of sort hierarchies can be employed to model each semantic descriptor type as the *top sort* of a separate sort hierarchy. Such an approach enables a modeller to refine each type by specialization. The possibility to treat descriptors as objects is an extension to KL-ONE (Brachman and Schmolze, 1985), which does not include a feature to extend the medium of representation. Specializing a descriptor sort or descriptor type into a more specific type, a specialized type should have an extended description: its intension is greater than the intension of its supersort (i.e., specialization is an enhancement of intension). As an example, the descriptor married_to should be assigned to the semantic descriptor type Relation, whereas the descriptor larger_than should be assigned to the type Transitive_Relation, which is a specialization of Relation. By adapting the elements of INCA-COM, a modeller creates a particular specialized, accommodated modelling environment for an adequate representation of an application model in a particular application domain.

The descriptor domain is not located within the original application domain, but is thought of as an application domain on top of it. The descriptors and their types thus become the subject of description: the elements of a (descriptor, value) pair are viewed as consisting of particular instances of generic sorts of descriptors and generic sorts of values, respectively. A description is viewed as a combination of particular instances of these generic sorts, and, similarly, describing objects becomes a way of combining particular (descriptional) objects. Figure 9.5 extends Figure 9.2 by showing the relation

between objects, ontological concepts, and descriptions.

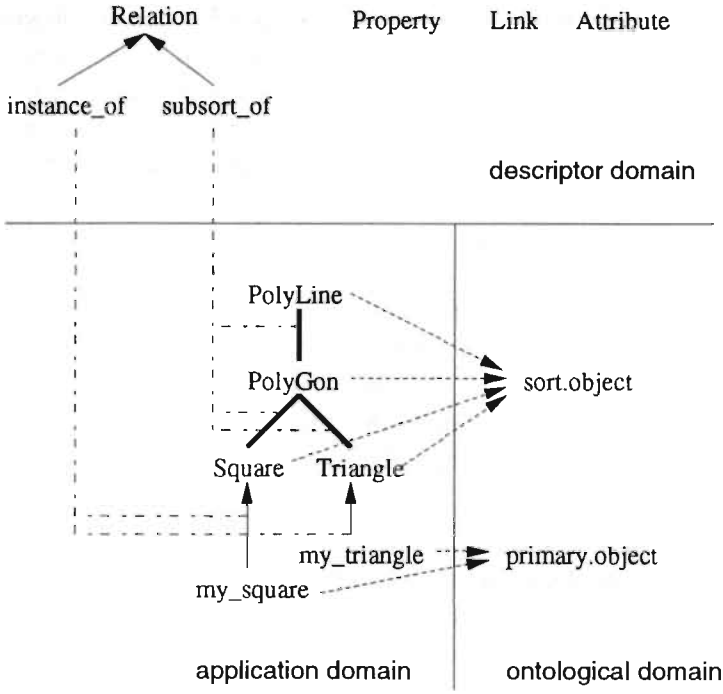


Figure 9.5: The relationship between objects, ontological concepts, and descriptions.

Figure 9.5 illustrates that the descriptions of objects in the application domain are considered as instances of specific descriptor types. The relations between the objects in the application domain are assigned to their descriptor type. The two Instance_of relations between my_square and Square, and my_triangle and Triangle are instances of the descriptor object Instance_of in the descriptor domain. The Instance_of object is an instance of the descriptor type Relation. Similarly, the relations between the sorts in the sort hierarchy in the application domain are considered as instances of the descriptor subsort_of, which is also an instance of the descriptor type Relation.

The other types of descriptors, such as properties, links, and attributes can be assigned to their respective descriptor types (for the sake of clarity this is not shown in Figure 9.5).

The result of the reflective description of INCA descriptors is the possibility to

extend INCA-COM by means of the same abstraction principles used in ordinary application domains. For example, a modeller may introduce a particular subsort TransitiveRelation of Relation, which has the semantics associated with a transitive relation.

With respect to the description of objects, each description is considered as a proposition being added to the set of existing propositions. The description of the INCA concepts is INCATOOL's *knowledge base*. It contains the knowledge INCATOOL uses for supporting the modeller in his task of describing application objects. This knowledge is formulated in an object-oriented manner. As an illustration we give a sample of this knowledge base in an informal language.

1. An INCA object consists of an object name, and a set P of propositions p.

2. The *unique* relation between object name and P is denoted with *object identity*.

3. Each proposition p is characterized by descriptor type DT, descriptor D, value V, and value domain VD, where both DT and D, as well as V and VD are related (see below).

4. Descriptor type DT is the *meta sort* of the declarative sorts *property* P, *attribute* A, *relation* R, and *link* L. Value domain VD is also a particular meta sort, namely of all sorts which during modelling are used as a value domain.

5. Each of the sorts P, A, R, L has in principle an infinite number of instances, in order to describe the application object in an adequate way (i.e., the extension of the sorts P, A, R, and L is infinite).

6. Constraint: the usage of an R or an L for the construction of a proposition p requires to instantiate the VD as an *application* sort, and V as an instance of this application sort.

7. Constraint: the usage of a P or an A for the construction of a proposition p requires to instantiate the VD as a *data type* from the underlying abstract machine, and V as an *exemplar* of this data type.

8. Each of the sorts P, A, R, L is to be conceived as the top of a sort hierarchy, so specializations can be made to introduce more specific *properties*, *attributes*, *relations*, and *links*; a *transitive relation* is an example.

9.4 The location of the ontology

In the Sections 9.2 and 9.3 we have described the use of (1) treating INCA-COM as an application domain in an ontology, and (2) treating the descriptors of objects as an application domain. In this section we show the location of the ontology with respect to both an application domain and a descriptor domain.

The ontological specification of an object is independent from the earlier introduced application-specific specification. The latter consists of application-specific sorts, together with application-specific sort hierarchies. Yet, the ontological specification is in form similar to the application-specific specification: it specifies to which ontological sorts each object belongs. The structure of the resultant specifications is shown in Figure 9.6.

In Figure 9.6 the three different domains of objects are shown, namely the application domain, the ontology (ontological domain), and the descriptor domain. They are shown as the three rectangles in Figure 9.6. The relation between these three domains is as follows.

Ontology Each of the objects in an application domain (both sort objects and primary objects) is an *instance* of several sorts in the ontology. The ontological sorts represent the concepts in INCA-COM, such as the concept of an object, a sort, and a composite object. The five sort hierarchies in the ontology indicate that we use five different viewpoints in the ontology. Each viewpoint is expressed by a separate sort hierarchy, following the specification of an application domain set forth in Section 5.1. The ontology has been discussed in Section 9.2.

For clarity's sake, the relations between the objects in the application domain and sorts in the ontology are shown in the figure as a set of five arrows between the application domain and the ontology. They indicate that each object in the application domain is an instance of a sort in the different five sort hierarchies of the ontology.

Application domain In the application domain each object is an instance of an application-specific sort. This is indicated by the arrow from each object to its sort. The sorts are part of sort hierarchies, shown as the structures of sorts with the fat lines. The division between objects and sorts is shown by the solid line between objects and sorts. The dashed line separates sorts and metasorts, following the basic partitioning of the object world described in Section 5.1. Objects in the application

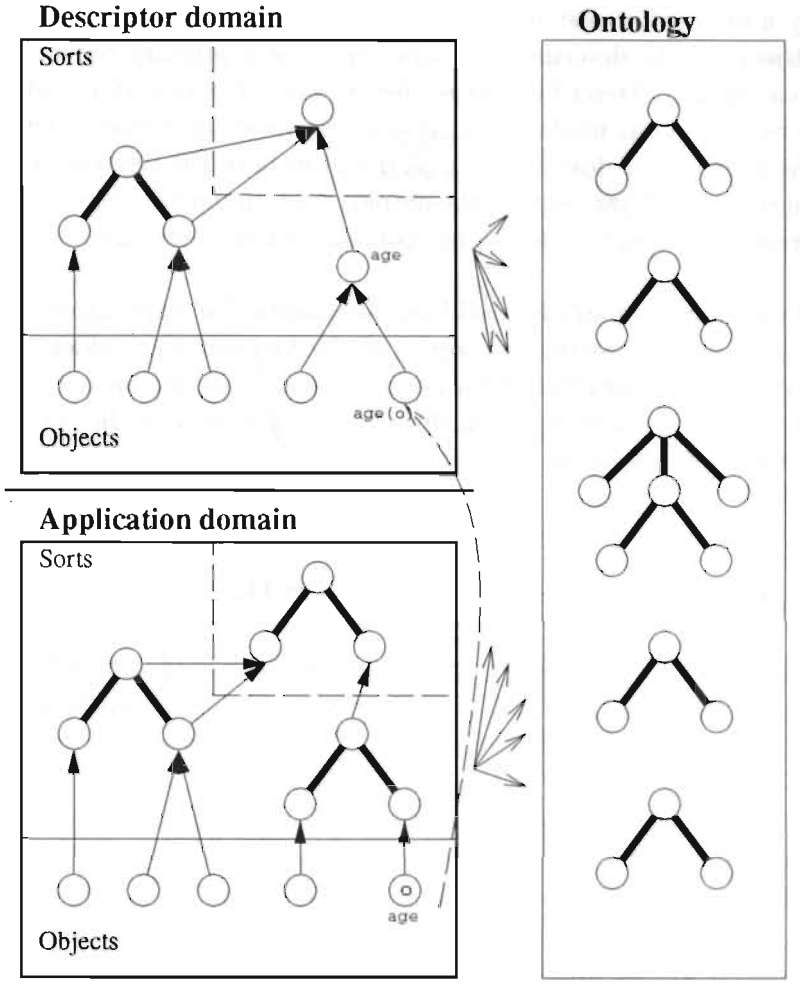


Figure 9.6: The location of the ontology.

domain are described by descriptors. For instance, object o is described by a descriptor *age*.

Descriptor domain The objects in the descriptor domain are the descriptors for objects in the application domain. Thus, the descriptors (and some of the values, in case of relations and links) are viewed as objects, which are located at a higher level, on top of the application domain. The figure illustrates this by object o in the application domain described

by a descriptor *age*: in the descriptor domain, the descriptor *age* is an object. In the descriptor domain, *age(o)* is a primary object, belonging to a sort *age*. Describing object descriptors of level n at a level $n+1$ thus is considered as modelling a separate application domain. Analogous to the application domain, each of the objects in the descriptor domain is an instance of the sorts in the ontology, again indicated by the set of five arrows between the descriptor domain and the ontology.

Each of the three domains is considered as a particular application domain. In Section 9.2 we have treated the sorts and the five sort hierarchies in the INCA ontology. The specification of an application domain by means of sorts and sort hierarchies has already been discussed in Chapter 5. In Section 9.3 we have introduced descriptor reflection, enabling a modeller to treat descriptors as objects.

9.5 Architecture of INCATOOL

The goal of INCA-COM is offering concepts for the modelling of information systems. In INCA-COM, the notion of an information system is broad since any system communicating with its users about a UOD is considered an information system. The generality of this notion of information systems enables us to consider INCATOOL as a particular kind of information system.

Hence, we present the architecture of INCATOOL as an information system following the framework of INCA-COM as presented earlier in Figure 4.15. The purpose of INCATOOL is to assist a modeller during the modelling process. Therefore, INCATOOL must have knowledge of the concepts of INCA-COM and the way applications are modelled. Thus, the UOD of INCATOOL is the complete modelling framework of Chapter 4 (i.e., Figure 4.15). This is indicated in Figure 9.7 with the dotted lines. Analogous to a "normal" application domain offering generic concepts for modelling a particular UOD, the UOD of INCATOOL offers templates for INCA application concepts. We recall that for the description of an application domain we need descriptors. These descriptors are taken from the descriptor domain, which, from INCATOOL's point of view, can also be seen as an application domain. The representation of INCA concepts as an application domain is what we have called an *ontology*. Figure 9.7 presents the relation between the ontology and the modelling framework of Chapter 4. For clarity, we only show the INCA concepts sort.object and primary.object from the ontology.

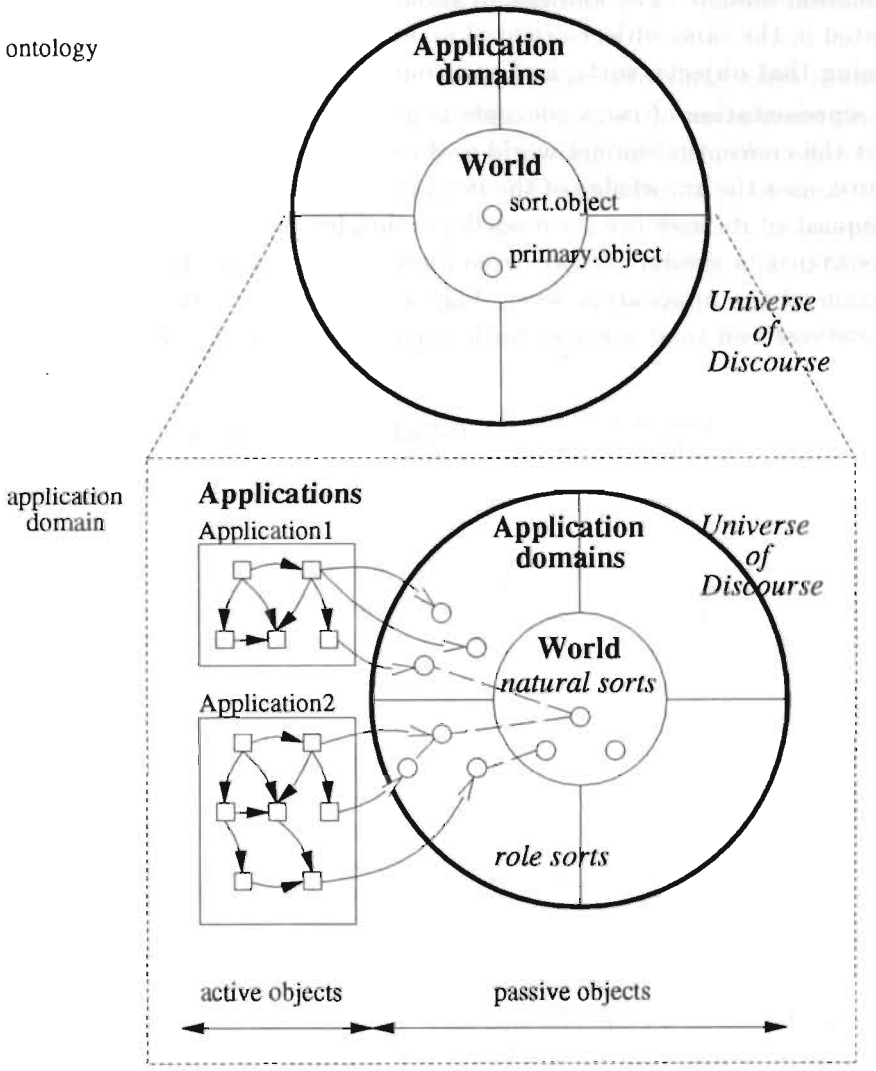


Figure 9.7: Ontology as the representation of INCA concepts.

Figure 9.7 shows that the (representation of the) UOD in the ontology concerns the domain of INCA concepts. This gives the essence of the ontology: it provides the basic concepts and model structures to be instantiated for particular application models. The knowledge about the use of the INCA concepts is represented in the same object-oriented manner as "normal" application domains meaning that objects, sorts, and sort hierarchies are used.

The representation of INCA concepts is used by INCATOOL for communicating about the conceptual-model world and creating particular object models. INCATOOL uses the knowledge of the INCA ontology to perform particular actions on request of its user (i.e., a modeller). Besides viewing a modeller as a user of INCATOOL, a similar relation exists between an application user and an application at the application level. Figure 9.8 illustrates this relation between applications and their users at both application level and INCATOOL level.

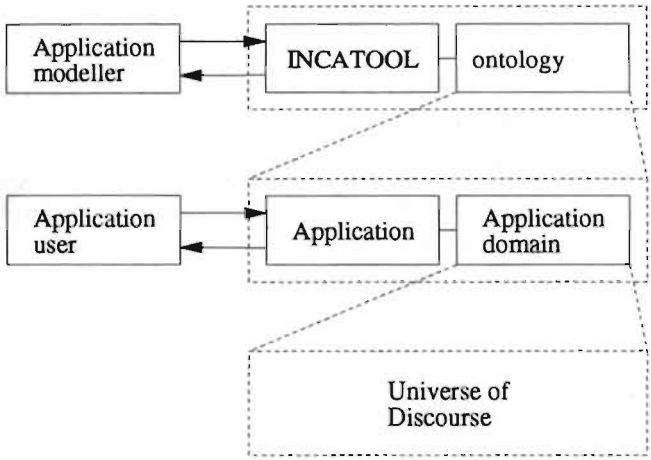


Figure 9.8: Applications and their users.

At the application level, an application user employs an application to create a representation of a UOD. To this end, the application domain offers specific sorts reflecting concepts in the UOD. An application user uses these sorts as templates to instantiate specific objects, corresponding to things in the UOD. By doing so, an application user may also be regarded as a *modeller*, since the use of an application entails transforming or translating facts holding in the UOD into application-domain objects.

In a similar way, at the level of INCATOOL an application modeller uses a specific application, INCATOOL, to create specific application domains, with

associated applications, to be used by application users. Of course, such an application domain must reflect its UOD by an appropriate choice of concepts corresponding to the model of application *users*. This process of finding the right concepts to model a UOD is especially important during analysis in the software life cycle.

The structure of application domains, applications, and their users is similar to the structure proposed by Nijssen (1989). The INCA conceptualization of an application domain, consisting of objects playing specific roles allows particular sorts, i.e., generic UOD structures, to be reused in different application domains. Furthermore, our conceptualization of an application consisting of communicating active objects is more detailed than the *conceptual information processor* proposed by Nijssen (1989). Nevertheless, Nijssen's work is insightful regarding the relation between a meta model and an application-domain model.

9.6 Chapter summary

In this chapter we have described the architecture of INCATOOL. We have taken assimilation and accommodation, two important aspects of knowledge-representation processes, as requirements for INCATOOL. The need for assimilation has facilitated the construction of an ontology of INCA concepts, which furnishes a set of basic concepts to be used in modelling any application domain. The need for accommodation has enabled us to consider the space of object descriptors as an application domain, the descriptor domain. Thus, descriptors are treated as objects, leading to a form of language reflection, called descriptor reflection. By treating descriptors as objects, the descriptor types introduced in Chapter 4 become the sorts of the descriptor domain. Describing objects entails instantiating the descriptor sorts in the descriptor domain. Descriptor reflection enables the modeller to specify the semantics of the descriptors which are used during modelling. Furthermore, descriptor reflection allows a modeller to accommodate the modelling language by introducing new types of descriptors.

In the context of the conceptual framework described in Chapter 4, INCATOOL is an application communicating with a user about changes in the application domain. In the case of INCATOOL, the application domain is the domain of INCA concepts represented in the INCA ontology. It offers the basic concepts to be used in modelling application domains. In interaction with its user,

INCATOOL instantiates INCA concepts from the ontology. The ontology presented furnishes a universal framework for modelling objects. By specifying the sorts and their structure for a particular application, a specific application framework is created. The descriptions of the (conceptual) objects in an application domain are formed by instantiating the sorts in the descriptor domain. Accommodation of the modelling language by introducing new types of descriptors can be performed with INCATOOL since the descriptor domain is also an application domain.

Chapter 10

Evaluation

In this chapter, we summarize the contributions of this thesis and draw some conclusions on the problem statement set forth in Chapter 2. Thereafter, we suggest directions for future research.

10.1 Summary of contributions

The evolution of software-development tools to address the complexity of software-systems development has taken place in two separate domains. First, the automation of tasks in programming has stimulated developments of software-engineering tools, such as compilers and eventually integrated combinations of tools, known as *computer-aided software engineering* (CASE) environments. Second, within artificial intelligence there has been research into automatic programming. The ultimate goal of automatic programming is to arrive at automatically-created computer programs on the basis of a problem description. However, this goal has, by far, not been reached and it remains an open question whether it will ever be. The more promising way to combine the efforts of both software engineering and artificial intelligence leads to the concept of *intelligent* CASE environments. Led by the wish to create the foundations for future intelligent CASE environments, we formulated the following problem statement in Chapter 2.

> What is the nature of a conceptual object model for the analysis and design of information systems?

In the previous chapters we have provided the concepts and structuring principles for the INCA Conceptual Object Model (INCA-COM).

An important decision has been to base INCA-COM on object-oriented concepts. The reason for this is twofold: object orientation has emerged in artificial intelligence as a knowledge-representation tool, and in programming as an evolution of different programming paradigms. The purpose of our research has been the identification of the concepts enabling the level of object-oriented modelling to be raised to the levels of analysis and design, such that (1) a UOD can be represented; (2) application domains can be represented; and (3) systems can be modelled.

For the modelling of information systems we have developed and described a conceptual framework consisting of three distinct parts. First, objects occurring in the real-world outside of an information system are modelled by a core of objects belonging to so-called *natural sorts*. Second, objects in an application domain are modelled by describing *role sorts*, i.e., sorts to be instantiated by applications. Third, the active part of an information system consists of *active objects* communicating with each other about objects in the real world and the roles these objects are playing. Communication between active objects is based on speech-act theory.

The models used in modelling according to the proposed framework are (1) the *information model* describing the structure of information, including roles; (2) the *event model* describing which events can take place in the object world and which order is imposed on these events; (3) the *behavioural model* describing which objects are active by performing particular actions, which give rise to events in the object world; and (4) the *communication model* describing what interactions take place between active objects. The information model and the event model are used for modelling a passive object world, whereas the behavioural model and the communication model are used for modelling a system of active objects. Such a division between a passive-object world and an active-object system may seem counterintuitive to the general idea of object orientation as a unifying and uniform concept. However, we think such a division is essential in modelling information systems: the nature of an information system entails that it contains a representation of a UOD. Although such represented objects can be active in the real world, their *representation* is under the control of active objects within the information system. Modelling an information system with only one type of object does not aid in developing a clear conceptual model.

For the description of objects we have introduced different types of descriptors. They allow a modeller to express different semantic purposes of descriptions. A *property* is used to model an essential feature of an object, which it will have

in every possible application. An *attribute* is a contingent feature of an object when it plays a role. A *relation* is an essential association between objects, and a *link* is a non-essential association between objects, when they are playing a role.

In addition to the descriptional structuring of objects according to the modelling framework, INCA-COM contains four structuring principles on the domain of objects. They are sort hierarchies, specification, composition, and grouping. These structuring principles are based on meaningful relations between objects and sorts. The inheritance mechanisms in INCA-COM are based on sort hierarchies and specification. We have described three types of inheritance, namely common inheritance, normal inheritance, and structural inheritance. A system is a composite object with a structure. In INCA-COM, active objects communicating among each other on an object world form an active-object system. In order to allow objects to evolve over time, a version concept is part of INCA-COM. We have assessed the problems in managing versions of composite objects, and have presented algorithms for addition and deletion of versions and parts of composite objects.

For support during modelling with INCA-COM we have described the architecture of INCATOOL. It supports the assimilation and accommodation of application knowledge. The architecture is based on the model for information systems presented in Chapter 4, leading to an ontology of INCA concepts. Modelling with INCA-COM is supported by INCATOOL, using the ontology as its application domain. INCA-COM has been extended with reflective capabilities. Two types of reflection are provided. First, structural reflection consists of treating sorts are objects, allowing them to be described in the same manner as normal objects. Second, descriptor reflection, a form of language reflection, consists of treating descriptors as objects, allowing the language to be accommodated to enhance the process of assimilation of object descriptions.

10.2 Conclusions

In the previous section we have briefly described the nature of INCA-COM, providing the answer to the problem statement of Section 2.5. Object-oriented modelling is often seen as a panacea for addressing complexity problems during the development of information systems. Especially the concept of an object is mentioned as unifying the behavioural aspect and the representational aspect of an information system. However, for the analysis and the design of informa-

tion systems a division is necessary between (1) active model concepts, taking care for the execution of processes, and (2) passive model concepts, accounting for the representation of a UOD in the information system. By assigning these aspects to a single type of object, combining both procedures and data, the distinction between processing and representation is denied. In INCA-COM the modelling of an information system is based on two different types of object: the active object and the passive object.

In Section 2.5 we mentioned five requirements for the conceptual object model. Below, we repeat each requirement, and describe how it has been fulfilled by INCA-COM.

1. *Representing the* UOD *of an information system.*
   The UOD of an information system is modelled explicitly in INCA-COM using the information model and the event model. These models allow a modeller to express which things in the context of an information system are represented as objects, and which events can take place. The representation of a UOD forms an object world, consisting of passive objects.

2. *Modelling generic parts of a* UOD.
   Generic parts of a UOD are modelled by natural sorts in the core of the object world. They represent sorts of objects which occur naturally in different applications.

3. *Modelling specific parts of a* UOD.
   Specific parts of a UOD are modelled by role sorts in application domains of the object world. They represent roles of natural objects occurring in an application domain.

4. *Modelling information systems by means of well-defined (de)composition principles.*
   An information system or application is modelled by active objects, communicating on an object world. Applications are modelled using the behavioural model and the communication model. The structure of an information system consists of the communication links between the active objects.

5. *Reuse of specifications to allow cheap systems to be built.*
   The use of object-oriented principles for both representation of a UOD

and modelling of an information system enhances the reuse of specifications. The distinction between a passive-object world and an active-object system enhances the understanding and subsequent reuse of different parts of an information system.

Regarding the long-term objective of the INCA project, i.e., realizing an INtelligent CASE environment for systems development, we remark the following. INCA-COM offers concepts for modelling information systems. Since a CASE environment is also a (special) kind of information system, it can be modelled with the INCA concepts. The conceptual architecture of INCATOOL sketched in Chapter 9 gives a starting point for the realization of such a CASE environment. INCATOOL contains knowledge on the concepts in INCA-COM, enabling to assist a modeller in describing the objects in an information system. The realization of INCATOOL should be tackled in future projects.

10.3 Suggestions for future research

In order to achieve full INtelligent CASE, based on the concepts of INCA-COM, the following research is envisaged.

- The language for modelling with INCA-COM should be defined formally. This includes a formal notation for modelling of laws.

- A standardized set of system architectures should be made available. The availability of such a set allows system designers to reuse particular systems, and to implement systems with a predefined architecture, such as a client/server system or a blackboard system.

- The architecture of INCATOOL, based on a reflective metamodel of INCA-COM, should be proven by implementing it in a suitable way. Some experience with this in C++ has shown that implementation of reflection in such a language is non-trivial. The concept of so-called meta CASE tools is interesting in this context.

- The value of the concept of an active-object system communicating on an object world by means of speech acts should be assessed for conceptual models for so-called act-management systems, logistics systems, and, more generally for telematics systems.

We conclude this thesis by stressing that the main point of modelling information systems is the conceptual division between passive objects as representations of things in the UOD of the information system, and active objects as actors, controlling passive objects in a purposeful manner. Object-oriented modelling of information systems should therefore offer concepts for modelling both passive objects and active objects. The concepts of object orientation can be used for both, but they should not be intermingled.

Appendix A

Visual displays

This appendix contains the descriptions of the visual displays, which have been introduced in Chapter 7. Figure A.1 contains the descriptions of the primitive visual displays, and Figure A.2 contains the descriptions of the constrained and/or composite visual displays.

```
Line {
  SORT : VisualizationMetaSort;
  A :: instance_descriptors : {
    P :: endPosition : (0,0) :: Coordinate;
  };
}

Oval {
  SORT : VisualizationMetaSort;
  A :: instance_descriptors : {
    P :: radius1 : 0 :: NaturalNumber;
    P :: radius2 : 0 :: NaturalNumber;
  };
}

Spline {
  SORT : VisualizationMetaSort;
  A :: instance_descriptors : {
    P :: controlPoints : { _ } :: P(Coordinate);
  };
}

Text {
  SORT : VisualizationMetaSort;
  A :: instance_descriptors : {
    P :: font : courier :: Font;
  };
}
```

Figure A.1: Primitive visual displays.

```
PolyLine {
 SORT : VisualizationMetaSort;
 A :: instance_descriptors : {
  PARTS : {Line +};
 };
}

Rectangle {
 SORT : VisualizationMetaSort;
 SUPERSORT : PolyLine;
 A :: instance_descriptors : {
  -- constraint: number of lines is 4
  -- and angle between every subsequent
  -- pair of lines is 90 degrees.
 };
}

Square {
 SORT : VisualizationMetaSort;
 SUPERSORT : Rectangle;
 A :: instance_descriptors : {
  -- constraint: lines have equal length
 };
}

Circle {
 SORT : VisualizationMetaSort;
 SUPERSORT : Oval;
 A :: instance_descriptors : {
  -- constraint : radius1 = radius2
 };
}
```

Figure A.2: Composite and constrained visual displays.

Appendix B

Hypothetical session with INCATOOL

This appendix contains an hypothetical session with INCATOOL, illustrating its use in modelling with INCA-COM. It allows us to gain insight into its manner of operation.

For this purpose we model a domain of various objects and sorts. INCATOOL has to support the modeller in finding the right place to add new sorts, while the object editor can be used to actually describe the new sorts. INCATOOL shows similarities with Smalltalk/V's ClassHierarchyBrowser (Digitalk Inc., 1988), but has (at least) two major differences. First, INCATOOL supports the descriptors used in INCA-COM. Second, INCATOOL supports the *incremental* description of a domain (Smalltalk/V's ClassHierarchyBrowser does not support the addition of a new class which is a generalization of two (or more) other classes: one can only *specialize* by adding classes as subclasses to a hierarchy.)

Figure B.1 gives a first image of the user interface INCATOOL may possess. In the hypothetical session we use a model in the domain of data-flow diagrams, which we described in Chapter 7. One part of this model describes a model of a data-flow diagram, and the associated data-flow diagram editor. The presentation of objects in this part is realized by *visualization* sorts, described as INCA objects too.

Since every object in INCA-COM is an instance of a sort, and every sort occurring in a particular domain is part of a hierarchy and itself an instance of a so-called metasort, INCATOOL initially shows a survey of all metasorts present. These metasorts correspond to the different modelling domains the modeller has established (see also Chapter 7). INCATOOL shows all domains, unless the

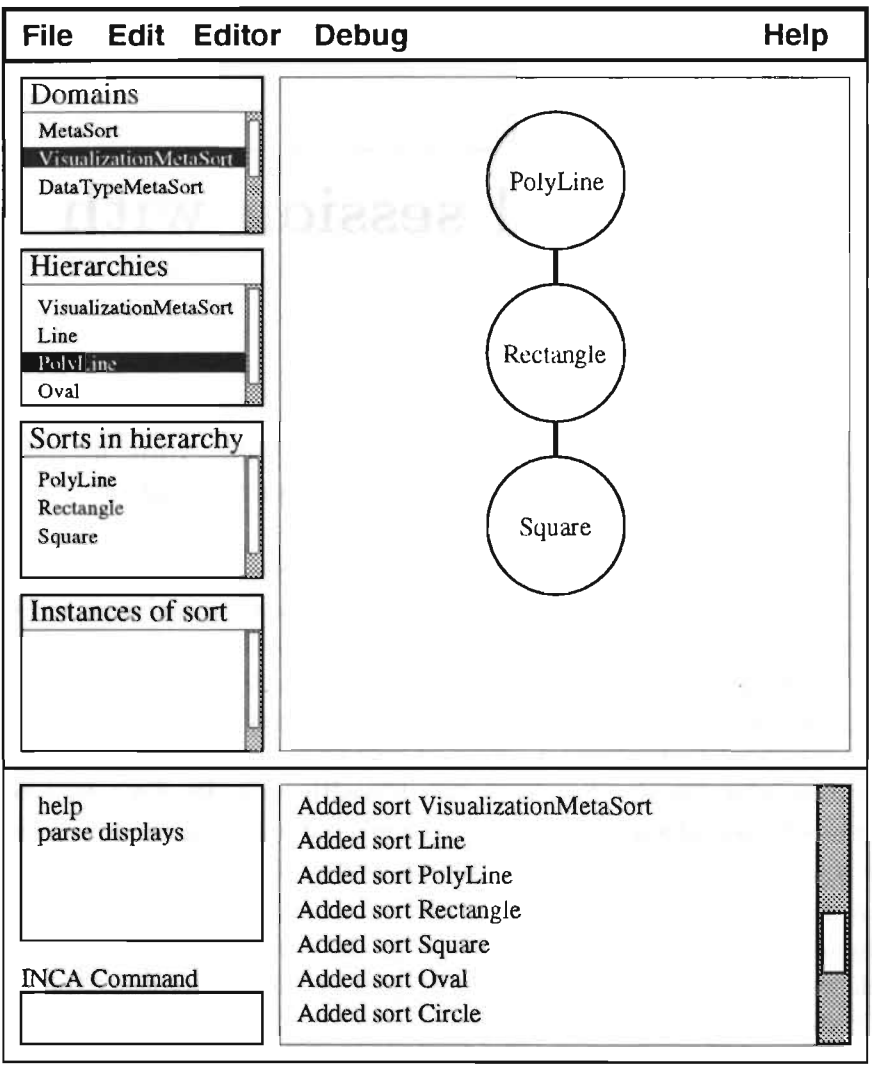


Figure B.1: INCATOOL's user interface.

user wishes to see only a subset of the available domains. INCATOOL visually indicates which domains the user can examine and modify.

The modeller now can select a domain (i.e., a *particular* metasort), which will show all the names of the sort hierarchies in the domain. The name of a sort hierarchy is determined uniquely by the name of the top sort of the hierarchy: the name of this *top sort* is characteristic for the modelling perspective from which the domain has been modelled by means of the hierarchy. Selecting a hierarchy is followed by showing it: the sorts in the hierarchy are displayed in a list, and a graphical presentation of the hierarchical structure is shown. The modeller can now select a sort. This can be done both in the sort list and in the graphical presentation. The selection of a sort is displayed in both ways and results in displaying the instances of the selected sort.

Figure B.2 shows two sort hierarchies (PolyLine and Oval) and their associated metasort VisualizationMetaSort, which instantiates into the (top) sorts PolyLine and Oval.

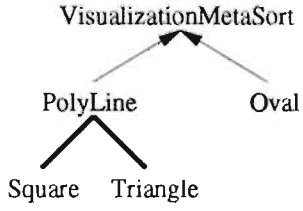


Figure B.2: Sort hierarchies.

For the sake of the example, suppose the modeller wishes to add the sort Circle to the Oval hierarchy depicted in Figure B.2. To that end he selects the sort Oval and chooses for addition of a subsort to this sort. Now INCATOOL creates a new sort, which is given automatically the initial description as presented in Figure B.3.

We recall that the description of an object in INCA-COM consists of the name of the object, followed by a set of descriptor-value pairs, and that both descriptor and value are typed. The types of descriptors represent the semantical categories within INCA-COM, which we have described in Chapter 4. We refer to Chapter 7 for examples of their use.

An object's sort descriptor SORT indicates to which sort the object belongs. For sorts, the value of the sort descriptor is necessarily a metasort. The supersort descriptor describes which sort is located above the sort in question. The

```
Circle {
  SORT: VisualizationMetaSort;
  SUPERSORT: Oval;
  A :: instance_descriptors: {
    P :: location : _ :: Coord; - - from VisualizationMetaSort
    B :: Draw : ... ;           - - from VisualizationMetaSort
    P :: radius1 : _ :: Real;   - - from supersort
    P :: radius2 : _ :: Real;   - - from supersort
  }
}
```

Figure B.3: The sort Circle.

attribute (A) named INSTANCE_DESCRIPTORS is used only for the description of sorts and indicates the descriptors which the *instances* of the sort Circle will have. In this case, *a* circle will have a property location, a behaviour Draw, and two properties, radius1 and radius2.

The modeller now has to indicate in what ways the sort Circle differs from its supersort Oval. The sort Circle differs from Oval in its constraint on the properties radius1 and radius2: a circle has two equal radiuses. The modeller can specify this by constraining the value of radius2 to be equal to the value of radius1. Such a constraint is expressed in a *law* attached to radius2, such that the values of both radiuses are always equal.

The above example shows the creation of a subsort by *constraining the value* of one of its descriptors. In this case, the values of two descriptors are linked by putting them on a par. In addition, a subsort can be specified by constraining the *value domain* of a descriptor. It is not possible to create a subsort which has a broader value domain. If such is the case, the modeller possibly should exchange the subsort and supersort.

Besides specialization by *constraining the value domain* of descriptors, there is a second way of specializing sorts. It consists of the *addition* of descriptors to the description of a sort. For example, the modeller can specify a subsort of Circle, called LabelledCircle, which adds a text string to supply a circle with a label. Such a labelled circle can be used for example for the visualization of processes in a data-flow diagram. The specification of such a subsort is done in an analogous way as described above, taking into account that a new descriptor is being added. Of course, it is also possible to create subsorts which

combine both constraints on descriptor values and extension of descriptors.

A second way of creating new sorts is *generalization*. It consists of leaving out irrelevant details of a number of sorts, thus creating a *generic* sort with the relevant descriptions of the more specialized sorts. The modeller can for example specify a generalization of the sorts Square and Triangle, to be located in between PolyLine on the one hand, and Square and Triangle on the other hand. For that purpose, he selects the sorts to be generalized. The specifications of these sorts are given in Figure B.4.

The modeller now indicates that the generalization of Square and Triangle will be called PolyGon. INCATOOL assumes that the new sort will be located between Square and PolyLine (i.e., as a subsort of PolyLine and as supersort of Square and Triangle). Subsequently, the modeller is asked to specify which descriptors of Square and Triangle are to be specified for PolyGon, and which remain to be specified by Square and Triangle themselves. The modeller indicates that the coinciding constraint on starting and ending point is to be specified for PolyGon. The remaining ones are specified for Square and Triangle. Figure B.5 describes the sorts PolyGon, Square and Triangle after this generalization operation.

Note that the number of constraints of both Square and Triangle has diminished by one as a result of the constraint specified for PolyGon. In fact, Triangle's only difference with respect to PolyGon is its number of lines, which is restricted to three. By performing the above generalization, one could use the more generic sort PolyGon to specify new subsorts, such as, e.g., a parallelogram.

```
PolyLine
SORT: VisualizationMetaSort;
A :: instance_descriptors:
  {
    PARTS : { Line * };
  }

Square {
  SORT: VisualizationMetaSort;
  SUPERSORT: PolyLine;
  A :: instance_descriptors: {
    PARTS : { Line 4 };
    - - with 3 constraints:
    - - the four lines have equal lengths,
    - - are perpendicular,
    - - and the starting point of the first
    - - line is the ending point of the
    - - fourth line.
  }
}

Triangle {
  SORT: VisualizationMetaSort;
  SUPERSORT: PolyLine;
  A :: instance_descriptors: {
    PARTS : { Line 3 };
    - - with the constraint:
    - - the starting point of the first
    - - line is the ending point of the
    - - third line.
  }
}
```

Figure B.4: The sorts PolyLine, Square and Triangle.

```
PolyLine
SORT: VisualizationMetaSort;
A :: instance_descriptors:
  {
   PARTS : { Line * };
  }

PolyGon {
  SORT: VisualizationMetaSort;
  SUPERSORT: PolyLine;
  A :: instance_descriptors: {
    - - constraint: the starting point of
    - - line 1 is the ending point of
    - - line n.
  }
}

Square {
  SORT: VisualizationMetaSort;
  SUPERSORT: PolyGon;
  A :: instance_descriptors: {
    PARTS : { Line 4 };
    - - with 2 constraints:
    - - the four lines have equal lengths and
    - - are perpendicular
  }
}

Triangle {
  SORT: VisualizationMetaSort;
  SUPERSORT: PolyGon;
  A :: instance_descriptors: {
    PARTS : { Line 3 };
  }
}
```

Figure B.5: The sort hierarchy after creating the sort PolyGon.

Glossary

active object an object which has an active role within an information system. Active objects perform a discourse on an object world among each other by means of speech acts.

actor synonymous with *active object*.

application domain part of a UOD that is specific for an application.

attribute (a descriptor type). A contingent feature of an object when it plays a certain role in an application domain.

composite object an object composed of objects other than itself.

composition the set of component objects of a composite object.

conceptual model resulting model from conceptual modelling. In the model triangle, a conceptual model is a role of a conceptual system with respect to the three types of possible systems.

conceptual modelling the activity of formally describing some aspects of the physical and social world around us for purposes of understanding and communication.

conceptual object model conceptual model having objects as its primary concept.

conceptual schema Formal description which results from conceptual modelling (Mylopoulos, 1992).

conversation combination of different types of speech acts, by which means active objects perform a discourse, and by which active object may influence each other.

description Annotation of an object, resulting from modelling. Consists of a combination of a (typed) descriptor, and a (typed) value.

descriptor element of a description, used to designate the description.

descriptor type Type of a descriptor. INCA-COM contains six descriptor types: property, attribute, relation, link, display, and version. The set of available descriptor types may be extended by a modeller.

display (a descriptor type). A display describes how an object manifests itself, e.g., by means of a visual presentation, a textual presentation, or audible presentation.

empirical model resulting model from empirical modelling. In the model triangle, an empirical model is a role of a concrete system with respect to the three types of possible systems. An example of an empirical model is a simulation of a logistics system.

entity substantial individual, something which exists.

extension the set of objects denoted by a token (or belonging to a sort in INCA-COM).

formal model resulting model from formal modelling. In the model triangle, a formal model is a role of a formal system with respect to the three types of possible systems.

illocutionary act Type of speech act, in which a speaker expresses her/his intention, by means of an illocutionary force, which includes an illocutionary point.

illocutionary point intention of an illocutionary act. There are five illocutionary points: (1) assertive or descriptive; (2) commissive; (3) directive; (4) declarative; and (5) expressive.

INCA Abbreviation of INtelligent CASE. Refers to a next generation of CASE tools, based on a formally defined conceptual model of an information system. The two main goals of an INCA environment are (1) to furnish a representation system for capturing the real world in which a (future) information system is to be employed, and (2) to offer concepts for modelling and constructing information systems, in terms of the representation of (1).

INCA-COM Abbreviation of INtelligent CASE Conceptual Object Model. INCA-COM has been designed with the purpose of providing a conceptually rich basis for the modelling of information systems, in order to enable a modeller to express a model in a natural way (i.e., with concepts being used by humans, not by computers).

information system An ensemble of information and information processing agents, humans or machines.

intension the concept being designated by a token, i.e., the meaning of a token.

interface part of an object which defines the operations other objects may invoke, and which parameters must be provided.

link (a descriptor type). A contingent association of an object with another object, when the object is playing a role.

meaning triangle A way of visualizing the relations between tokens (symbols), concepts, and things (objects). A token designates a concept and denotes a thing, and a concept refers to a thing. The relations in the meaning triangle are the basis for defining the concepts *intension*, *extension*, *referent*, and *population*.

method a way of working in order to solve a problem.

methodology set of methods for solving a particular kind of problems (such as system development). In Dutch there is a distinction between a 'methodiek', standing for a set of methods for solving practical kinds of problems, and a 'methodologie', standing for the scientific knowledge on which method to use.

model the notion of model is based on a relation between two different systems: "someone, who uses a system M, which is independent of a system S, in order to understand system S, uses system M as a model of system S." There are three types of models: empirical models, consisting of concrete things, conceptual models, consisting of concepts, and formal models, consisting of uninterpreted syntactic symbols (forms).

model triangle way of visualizing the relations between the three different kinds of models, empirical model, conceptual model, and symbolic model.

modeller anyone who is modelling. Programmers, designer, analysts are considered to be modellers.

modelling the process of (1) determining a system in the problem domain; (2) determining an aspect system of the system; and (3) finding an isomorphic transformation (translation) of the aspect system into a model system.

object In object-oriented programming: an encapsulated combination of data and procedures that act upon the data. The data form the state of the object, the procedures are the operations other objects may invoke through the object's interface.
In INCA-COM: an individual concept with an own identity. An object has a set of descriptions. Every object has a unique name. An object o is formalized as a tuple $\langle n, D(n)\rangle$, in which n is the object name, and $D(n)$ the set of descriptions of n.

object model An *object model* (OM) defines the conceptual entities that can be used for the object-oriented modelling of an application domain or UOD.

object-oriented model An object-oriented model is the result of expressing a particular domain using the concepts offered by an object model.

object world Conceptualization of a UOD in terms of concepts offered by object orientation (objects).

ontology the set of concepts to be used when modelling with INCA-COM. More generally, an ontology defines the basic concepts for modelling a domain.

passive object an object which has a passive role within an information system. A passive object is a representation of a thing in the UOD of an information system.

property (a descriptor type). An essential feature of an object, meaning that the object has the property in every possible application.

protocol A typical interaction sequence between two or more objects. A protocol has a dynamic nature, as opposed to an interface, which is static.

relation (a descriptor type). An essential association between objects, without which the description of the objects is not complete.

role when an object plays another object in an application domain, it has a role in the application domain.

sort A sort represents a concept which can be instantiated. A sort acts as a semantical primitive, and allows its instances to receive an initial set of descriptions.

speech act minimal unit of human communication. One type of speech act is the *illocutionary act*. In speech act theory any utterance of a sentence has a *propositional content* and an *illocutionary force*.

speech act theory theory of communication between people, stating that communication is not only a means for spreading information, but also a way of acting (Searle, 1969; Searle and Vanderveken, 1985). Communication consists of sequences of speech acts.

symbolic model See formal model.

system object with a composition (the set of component objects), a structure (the relations between the component objects, and the relations between the component objects and objects in the environment), and an environment (the objects not contained in the system, but having relations with components in the system).

thing ontological term for a substantial individual together with its properties.

Universe of Discourse see UOD.

UOD The UOD of an information system consists of the things in a particular domain, about which people in the information system perform a meaningful discourse.

version (a descriptor type). A version is a variation of an object.

world core of the representation of a UOD containing natural sorts, being representations of concepts which can be reused in different application domains.

References

The numbers between parentheses after each entry refer to the pages on which a reference to the entry in question occurred.

Abbott R. J. (1987). Knowledge Abstraction. *Communications of the ACM*, Vol. 30, No. 8, pp. 664–671. (65)

ACM (1958). Table of automatic programming systems. *Communications of the ACM*, Vol. 1, No. 4, p. 8. (28)

ACM (1991). Special issue on next-generation database systems. *Communications of the ACM*, Vol. 34, No. 10, pp. 31–120. (61)

ACM (1993). Discussion on concept-oriented view and program-oriented view. *Communications of the ACM*, Vol. 36, No. 1, p. 112. (44)

Aho A. V., Sethi R., and Ullman J. D. (1986). *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts. (20)

Alderson A. (1989). Current Approaches to Configuration Management. Esprit MACS project, document number MCSS0006. (155)

America P. (1987). Inheritance and Subtyping in a Parallel Object-Oriented Language. *Proceedings of the European Conference on Object-Oriented Programming (Bézivin* et al., *1987)*, pp. 234–242. (44)

Andrews T. and Harris C. (1987). Combining Language and Database Advances in an Object-Oriented Development Environment. *Proceedings of the Second ACM Conference on Object-Oriented Programming Systems, Languages and Applications (Meyrowitz, 1987)*, pp. 430–440. Reprinted in Zdonik and Maier (1990). (61)

Apostel L. (1960). Towards a formal study of models in the non-formal sciences. *Synthese*, Vol. 1960, No. 12, pp. 125–161. (55)

Bailin S. C. (1989). An Object-Oriented Requirements Specification Method. *Communications of the ACM*, Vol. 32, No. 5. (66)

Bakker H. and Braspenning P. J. (1991a). Geautomatiseerde ondersteuning van object-georiënteerd modelleren. *Proceedings AIToepassingen '91* (eds. B. R. van der Spek and J. Treur). Nederlandse Vereniging voor Kunstmatige Intelligentie. In Dutch. (195)

Bakker H. and Braspenning P. J. (1991b). Modelling Objects using the INCA Conceptual Object Model. *Proceedings of Technology of Object-Oriented Languages and Systems (Meyer and Bézivin, 1991)*, pp. 175–186. (169)

Bakker H. and Braspenning P. J. (1992). An Object-Oriented Model of a Spreadsheet. *Proceedings of the European Simulation Symposium* (eds. W. Krug and A. Lehmann), pp. 193–197, San Diego, California. Society for Computer Simulation International. (183)

Bakker H., Leeuwen L.C.J. van, Braspenning P. J., and Uiterwijk J. W. H. M. (1990). Version Management of Composite Instantiated Objects. *Proceedings of the European Simulation Multiconference* (ed. B. Schmidt), pp. 93–98, San Diego, California. The Society for Computer Simulation International. (151)

Bakker H., Braspenning P. J., and Vassilev V. (1992). De ontologie en generieke kenmerken van een object-georiënteerde taal voor kennisgebaseerde analyse en ontwerp. *Proceedings NAIC '92* (eds. H. de Swaan Arons, H. Koppelaar, and E. J. H. Kerckhoffs), pp. 117–128. Nederlandse Vereniging voor Kunstmatige Intelligentie. In Dutch. (195)

Bancilhon F., Barbedette G., Benzaken V., Delobel C., Gamerman S., Lécluse C., Pfeffer P., Richard P., and Velez F. (1987). The Design and Implementation of O_2, an Object-Oriented Database System. *Advances in Object-Oriented Database Systems (Dittrich, 1987)*, pp. 1–22. (61)

Banerjee J. and Kim W. (1987). Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *Proceedings of the ACM SIGMOD Conference on the Management of Data*, pp. 311–322. (61)

Barstow D., Shrobe H., and Sandewall E. (eds.) (1984). *Interactive Programming Environments*. McGraw-Hill, New York, New York. (253)

Bézivin J., Hullot J.-M., Cointe P., and Lieberman H. (eds.) (1987). *Proceedings of the European Conference on Object-Oriented Programming*. Springer-Verlag, Berlin, Germany. (241, 251)

Birtwistle G. M., Dahl O-J., Myhrhaug B., and Nygaard K. (1977). *Simula Begin*. Studentlitteratur, Lund, Sweden. (36, 62)

Björnerstedt A. and Hultén C. (1989). Version Control in an Object-Oriented Architecture. *Object-Oriented Concepts, Databases, and Applications (Kim and Lochovsky, 1989)*, pp. 451–485. (151)

Bobrow D. G. and Collins A. (eds.) (1975). *Representation and Understanding (Studies in Cognitive Science)*. Academic Press, New York, New York. (253)

Bobrow D. G. and Stefik M. (1983). *The* LOOPS *Manual*. Xerox Palo Alto Research Centre. (132)

Boehm B. W. (1981). *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, New Jersey. (68)

Boehm B. W. (1988). A Spiral Model of Software Development and Enhancement. *Software Engineering Project Management*. In Thayer (1988), pp. 128–142. (8, 15)

Booch G. (1991). *Object-oriented design with applications*. Benjamin Cummings, Englewood Cliffs, New Jersey. (22, 31, 37, 64, 66)

Borgida A., Mylopoulos J., and Wong H. K. T. (1984). Generalization / specialization as a basis for software specification. *On Conceptual Modelling (Brodie* et al., *1984)*, pp. 87–114. (61)

Borning A. H. (1981). The Programming Aspects of ThingLab, a Constraint-oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, Vol. 3, No. 4, pp. 353–387. (133)

Brachman R. J. and Levesque H. J. (eds.) (1985). *Readings in Knowledge Representation*. Morgan Kaufmann, Los Altos, California. (253)

Brachman R. J. and Schmolze J. G. (1985). An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science*, Vol. 9, pp. 171–216. (61, 207)

Braspenning P. J. (1989a). A Framework for Modelling Complex Objects. *Proceedings of Hierarchical Object-Oriented Design*, Brunel Science Park, Uxbridge, England. UNICOM Seminars Ltd. (196)

Braspenning P. J., Uiterwijk J. W. H. M., Bakker H., and Leeuwen L. C. J. van (1989b). Conceptual Tools for Modelling Complex Objects. *Proceedings of the European Simulation Multiconference* (eds. G. Iazeolla, A. Lehmann, and H. J. van den Herik), pp. 256–262, San Diego, California. The Society for Computer Simulation International. (37, 68)

Braspenning P. J. (1990). A Reflexive Representation System for Objects. *Proceedings of the 1990 Summer Computer Simulation Conference* (eds. B. Svrcek and J. McRae), pp. 854–859, San Diego, California. The Society for Computer Simulation International. (201, 207)

Briot J.-P. and Cointe P. (1987). A Uniform Model for Object-Oriented Languages using the Class Abstraction. *Proceedings IJCAI '87*, pp. 40–43. (117)

Brodie M. L., Mylopoulos J., and Schmidt J. W. (eds.) (1984). *On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases, and Programming Languages*. Springer Verlag, New York, New York. (61, 243)

Brooks F. P. (1975). *The Mythical Man-Month*. Addison-Wesley, Reading, Massachusetts. (9)

Brooks F. P. (1987). No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, Vol. 20, No. 4, pp. 10–19. (9, 11)

Brussaard B. K. and Tas P. A. (1980). Information and Organization Policies in Public Organization. *Information Processing 80* (ed. S. H. Lavington), Amsterdam, The Netherlands. North-Holland. (5, 6)

Bunge M. (1974). *Treatise on Basic Philosophy, Semantics I: Sense and Reference*. Reidel, Boston, Massachusetts. (52)

Bunge M. (1977). *Treatise on Basic Philosophy, Ontology I: The Furniture of the World*. Reidel, Boston, Massachusetts. (71, 72, 81, 94, 112, 121, 133, 134)

Bunge M. (1979). *Treatise on Basic Philosophy, Ontology II: A World of Systems*. Reidel, Boston, Massachusetts. (112, 113)

Chen P. P. (1976). Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, Vol. 1, No. 1, pp. 9–36. (18, 61)

Coad P. and Yourdon E. (1991). *Object-oriented analysis (second edition)*. Prentice Hall, Englewood Cliffs, New Jersey. (66)

Codd E. F. (1979). Extending the database relational model to capture more meaning. *ACM Transaction on Database Systems*, Vol. 4, No. 4, pp. 397–434. (61)

Cointe P. (1987). Metaclasses are First Class: the ObjVlisp Model. *Proceedings of the Second ACM Conference on Object-Oriented Programming Systems, Languages and Applications (Meyrowitz, 1987)*, pp. 156–167. (117)

Cox B. J. (1987). *Object-Oriented Programming – An Evolutionary Approach*. Addison-Wesley, Reading, Massachusetts. (35, 36, 47, 65, 128)

Dart S. A., Ellison R. J., Feiler P. H., and Habermann A. N. (1987). Software Development Environments. *IEEE Computer*, Vol. 20, No. 11, pp. 18–28. (26)

DeMarco T. (1979). *Structured Analysis and System Specification*. Prentice-Hall, Englewood Cliffs, New Jersey. (18, 66, 169, 170)

DeRemer F. and Kron H. (1975). Programming in the large versus programming in the small. *IEEE Transactions on Software Engineering*, Vol. 1, No. 1. (9)

Dietz J. L. G. (1987). *Modelleren en specificeren van Informatiesystemen*. PhD thesis, Technische Universiteit, Eindhoven, The Netherlands. In Dutch. (55, 56)

Dietz J. L. G. (1992a). *Leerboek Informatiekundige Analyse*. Kluwer, Deventer, The Netherlands. In Dutch. (19, 112)

Dietz J. L. G. (1992b). Subject-oriented modelling of open active systems. Technical report, Limburg University, Faculty of Economics, Maastricht, The Netherlands. ISSN 0926-9878. (84, 86, 104, 107, 109, 111, 113)

Digitalk Inc. (1988). *Smalltalk/V 286: Tutorial and Programming Handbook*. Digitalk Inc., Los Angeles, California. (227)

Dijkstra E. W. (1968). Go To Statement Considered Harmful. *Communications of the ACM*, Vol. 11, No. 3, pp. 147–148. (67)

Dittrich K. R. (ed.) (1987). *Advances in Object-Oriented Database Systems*. Springer-Verlag, Berlin. Lecture Notes in Computer Science, number 334. (242)

Etherington D. W. and Reiter R. (1985). On Inheritance Hierarchies with Exceptions. *Readings in Knowledge Representation*, pp. 329–334, Los Altos, California. Morgan Kaufmann. Reprint of article in Proc. AAAI-83, Washington D.C., pp. 104–108. (129)

Evans M. W. (1989). *The Software Factory*. John Wiley and Sons, New York, New York. (7)

Ferber J. (1989). Computational Reflection in Class-based Object-oriented Languages. *Proceedings of the Second ACM Conference on Object-Oriented Programming Systems, Languages and Applications (Meyrowitz, 1989)*, pp. 317–326. (202)

Fiadeiro J., Sernadas C., Maibaum T., and Sernadas A. (1992). Describing and Structuring Objects for Conceptual Schema Development. *Conceptual Modelling, Databases, and CASE (Loucopoulos and Zicari, 1992)*, pp. 117–138. (69)

Fichman R. G. and Kemerer C. F. (1992). Object-Oriented and Conventional Analysis and Design Methodologies. *IEEE Computer*, Vol. 25, No. 10, pp. 22–39. (66)

Fishman D. H., Beech D., Cate H. P., Chow E. C., Connors T., Davis J. W., Derrett N., Hoch C. G., Kent W., Lyngbaek P., Mahbod B., Neimat M. A., Ryan T. A., and Shan M. C. (1987). Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, pp. 48–69. Reprinted in Zdonik and Maier (1990). (61)

Gane C. and Sarson T. (1979). *Structured Systems Analysis*. Prentice-Hall, Englewood Cliffs, New Jersey. (18)

Gibson M. L. (1989). The CASE Philosophy. *Byte*, Vol. 14, No. 4, pp. 209–218. (27)

Goldberg A. and Robson D. (1983). *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Massachusetts. (36)

Gorlen K. E., Orlow S. M., and Plexico P. S. (1990). *Data Abstraction and Object-Oriented Programming in C++*. John Wiley & Sons, Chichester, England. (24, 35)

Hammer M. and McLeod D. (1981). Database description with SDM: A semantic database model. *ACM Transaction on Database Systems*, Vol. 6, No. 3, pp. 351–386. Reprinted in Zdonik and Maier (1990). (61)

Hofstadter D. R. (1979). *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, New York. (40)

In 't Veld J. (1981). *Analyse van Organisatieproblemen: Een toepassing van denken in systemen en processen*. Elsevier, Amsterdam, The Netherlands. In Dutch. (55)

Jackson M. A. (1983). *System Development*. Prentice-Hall, Englewood Cliffs, New Jersey. (18, 19, 66)

Joseph H. W. B. (1916). *An Introduction to Logic*. Clarendon Press. (121)

Kaiser G. E. and Habermann A. N. (1983). An Environment for System Version Control. *Digest of Papers Spring CompCon*, pp. 415–420. Also appeared as Tech. Rep., Dept. of Comp. Science, Carnegie-Mellon University, Nov. 1982. (152)

Kay A. C. (1977). Microelectronics and the Personal Computer. *Scientific American*, Vol. September, pp. 231–244. (45)

Kernighan B. W. and Pike R. (1984). *The Unix Programming Environment*. Prentice-Hall, Englewood Cliffs, New Jersey. (26)

Kim W. and Lochovsky F. H. (eds.) (1989). *Object-Oriented Concepts, Databases, and Applications*. ACM Press, New York, New York. (243, 249, 252)

Kim W., Banerjee J., Chou H.-T., Garza J. F., and Woelk D. (1987). Composite Object Support in an Object-Oriented Database System. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (Meyrowitz, 1987)*, pp. 118–125. (61, 133, 155, 161)

Kim W., Bertino E., and Garza J.F. (1989). Composite Objects Revisited. *Proceedings of the ACM SIGMOD Conference on the Management of Data*, pp. 337–347. (61)

Kramer N. J. T. A. and De Smit J. (1987). *Systeemdenken.* Stenfert Kroese, Leiden, The Netherlands. In Dutch. (56)

Kristensen B. B. and Østerbye K. (1991). Conceptual Modeling and Programming. Technical report, Institute for Electronic Systems, Aalborg, Denmark. ISSN 0106-0791. (71)

Lenat D. B., Guha R. V., Pittman K., Pratt D., and Shepherd M. (1990). Cyc: Toward Programs With Common Sense. *Communications of the ACM*, Vol. 33, No. 8, pp. 30–49. (68, 121, 199)

Lieberman H. (1986). Using Prototypical Objects to Implement Shared Behaviour in Object-Oriented Systems. *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages and Applications.* (41, 42)

Loucopoulos P. and Zicari R. (eds.) (1992). *Conceptual Modelling, Databases, and CASE.* John Wiley and Sons, New York, New York. (246, 248, 249, 250)

Loucopoulos P. (1992). Conceptual Modelling. *Conceptual Modelling, Databases, and CASE (Loucopoulos and Zicari, 1992)*, pp. 1–26. (61)

Macdonald I. (1986). Information Engineering – An improved, automatable methodology for designing data sharing systems. *Information System Methodologies: Improving the practice (Olle* et al., *1986).* (18)

Maes P. (1987). Concepts and Experiments in Computational Reflection. *Proceedings of the Second ACM Conference on Object-Oriented Programming Systems, Languages and Applications (Meyrowitz, 1987)*, pp. 147–155. (202)

Maier D. and Stein J. (1987). Development and Implementation of an Object-Oriented DBMS. *Research Directions in Object-Oriented Programming (Shriver and Wegner, 1987)*, pp. 355–392. Reprinted in Zdonik and Maier (1990). (61)

March S. (ed.) (1988). *Entity-Relationship Approach.* North-Holland, New York. (18)

Martin J. (1990). *Information Engineering, Books I, II, and III.* Prentice Hall, Englewood Cliffs, New Jersey. (66)

McIlroy D. (1968). Mass-produced software components. *Proceedings NATO Conference on Software Engineering.* In Naur and Randell (1968). (47)

Meyer B. and Bézivin J. (eds.) (1991). *Proceedings of Technology of Object-Oriented Languages and Systems.* Prentice-Hall, Hemel Hempstead, United Kingdom. (242)

Meyer B. (1988). *Object-Oriented Software Construction.* Prentice-Hall, New York, New York. (24, 36, 47, 86, 105, 128)

Meyrowitz N. (ed.) (1987). *Proceedings of the Second ACM Conference on Object-Oriented Programming Systems, Languages and Applications.* Association for Computing Machinery, New York, New York. Appeared as Special Issue of SIGPLAN Notices, Vol. 22, No. 10. (241, 245, 247, 248, 251, 253)

Meyrowitz N. (ed.) (1989). *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications.* Association for Computing Machinery, New York, New York. Appeared as Special Issue of SIGPLAN Notices, Vol. 24, No. 10. (246)

Minsky M. (1975). A Framework for Representing Knowledge. *The Psychology of Computer Vision* (ed. P. H. Winston), New York, New York. McGraw Hill. (2)

Minsky M. (1981). A Framework for Representing Knowledge. *Mind Design: Philosophy, Psychology, Artificial Intelligence* (ed. J. Haugeland), pp. 95–128, Montgomery. Bradford Books. (2)

Moon D. A. (1989). The COMMON LISP Object-Oriented Programming Language Standard. *Object-Oriented Concepts, Databases, and Applications (Kim and Lochovsky, 1989)*, pp. 49–78. (36, 130)

Mylopoulos J. (1992). Conceptual Modelling and Telos. *Conceptual Modelling, Databases, and CASE (Loucopoulos and Zicari, 1992)*, pp. 49–68. (60, 61, 207, 235)

Naur P. and Randell B. (1968). *Proceedings NATO Conference on Software Engineering.* NATO, Scientific Affairs Division, Brussels, Belgium. (7, 248)

Nijssen G. M. (1989). *Grondslagen van Bestuurlijke Informatiesystemen.* Nijssen Adviesbureau voor Informatica, Slenaken, The Netherlands. In Dutch. (10, 18, 19, 85, 215)

Nygaard K. (1986). Basic Concepts in Object-Oriented Programming. *SIGPLAN Notices*, Vol. 21, No. 10, pp. 128–132. (36)

Olle T. W., Sol H. G., and Verrijn-Stuart A. A. (eds.) (1986). *Information System Methodologies: Improving the practice*. IFIP-North Holland. (248)

Orr K., Gane C., Yourdon E., Chen P. P., and Constantine L. L. (1989). Methodology: The Experts Speak. *Byte*, Vol. 14, No. 4, pp. 221–233. (18)

Page-Jones M. (1980). *The practical guide to Structured Systems Design*. Yourdon Press, New York, New York. (67)

Peckham J. and Maryanski F. (1988). Semantic Data Models. *ACM Computing Surveys*, Vol. 20, No. 3, pp. 153–189. (61)

Perry D. E. (1987). Version Control in the Inscape Environment. *Proceedings of the IEEE 9th International Conference on Software Engineering*, pp. 142–149, Washington, D.C. IEEE Computer Society Press. (151, 152)

Piaget J. (1972). *The Psychology of Intelligence*. Adams, Littlefield, USA. (188, 196)

Rich C. and Waters R. C. (eds.) (1986). *Readings in Artificial Intelligence and Software Engineering*. Morgan Kaufmann Publishers, Los Altos, California. (12, 28, 29)

Rich C. and Waters R. C. (eds.) (1990). *The Programmer's Apprentice*. ACM Press, Reading, Massachusetts. (29, 30)

Rolland C. and Cauvet C. (1992). Trends and Perspectives in Conceptual Modelling. *Conceptual Modelling, Databases, and CASE (Loucopoulos and Zicari, 1992)*, pp. 27–48. (61, 85)

Royce W. W. (1970). Managing the Development of Large Software Systems. *Proceedings of IEEE WESCON*. IEEE. Reprinted in Thayer (1988). (15)

Rumbaugh J., Blaha M., Premerlani W., Eddy F., and Lorensen W. (1991). *Object-oriented modeling and design*. Prentice Hall, London, United Kingdom. (31, 37, 67, 85, 124, 128)

Saunders J. H. (1989). A Survey of Object-Oriented Programming Languages. *Journal of Object-Oriented Programming*, Vol. 1, No. 6, pp. 5–11. (62)

Searle J. R. and Vanderveken D. (1985). *Foundations of Illocutionary Logic*. Cambridge University Press, Cambridge, United Kingdom. (62, 84, 107, 108, 239)

Searle J. R. (1969). *Speech Acts. An Essay in the Philosophy of Language.* Cambridge University Press, Cambridge, United Kingdom. (19, 84, 107, 108, 239)

Shipman D. W. (1981). The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, Vol. 6, No. 1, pp. 140–173. Reprinted in Zdonik and Maier (1990). (61)

Shlaer S. J. and Mellor S. J. (1988). *Object-oriented systems analysis: modeling the world in data.* Yourdon Press, Englewood Cliffs, New Jersey. (66)

Shriver B. and Wegner P. (1987). *Research Directions in Object-Oriented Programming.* MIT Press, Cambridge, Massachusetts. (248, 251, 253)

Simon H. A. (1986). Whether Software Engineering Needs to Be Artificially Intelligent. *IEEE Transactions on Software Engineering*, Vol. 12, No. 7, pp. 726–732. (28)

Snyder A. (1987). Inheritance and the Development of Encapsulated Software Components. *Research Directions in Object-Oriented Programming (Shriver and Wegner, 1987)*, pp. 165–188. (41, 128)

Sommerville I. (1992). *Software Engineering.* Addison-Wesley, London, United Kingdom. 4th ed. (5, 8, 14, 19)

Sowa J. F. (1983). *Conceptual structures: information processing in mind and machine.* Addison-Wesley, Reading, Massachusetts. (53)

Stefik M. and Bobrow D. G. (1985). Object-Oriented Programming: Themes and Variations. *The AI Magazine*, Vol. 6, No. 4, pp. 40–62. (35, 132)

Stein L. A. (1987). Delegation is Inheritance. *Proceedings of the Second ACM Conference on Object-Oriented Programming Systems, Languages and Applications (Meyrowitz, 1987)*, pp. 138–146. (42)

Stroustrup B. J. (1987). What is "Object-Oriented Programming"? *Proceedings of the European Conference on Object-Oriented Programming (Bézivin et al., 1987)*, pp. 51–70, Berlin. (22)

Stroustrup B. (1991). *The C++ Programming Language, second edition.* AT&T Bell Telephone Laboratories. (36, 119, 130)

Tello E. R. (1989). *Object-Oriented Programming for Artificial Intelligence.* Addison-Wesley. (127)

Thayer R. H. (ed.) (1988). *Software Engineering Project Management.* IEEE Computer Society Press, Washington. (243, 250)

Tomlinson C. and Scheevel M. (1989). Concurrent Object-Oriented Programming Languages. *Object-Oriented Concepts, Databases, and Applications (Kim and Lochovsky, 1989)*, pp. 79–124. (45)

Touretzky D. S. (1986). *The mathematics of inheritance systems.* Morgan Kaufmann, Los Altos. (129)

Uiterwijk J. W. H. M. and Braspenning P. J. (1990). Specification and Inheritance in the INCA-COM. *Proceedings of Software Engineering and its Applications* (ed. M. Galinier), pp. 889–901. (116)

Van de Weg R. L. W. and Engmann R. (1991). A Framework for Object-Oriented Information Systems Design. *Proceedings of Computing Science in the Netherlands 1991*, pp. 600–616. (103, 113)

Van Griethuysen J. J. (ed.) (1982). *Concepts and terminology for the Conceptual Schema and the Information Base.* ISO. Report ISO TC97/SCS/WG3. (6)

Velho A. V. and Carapuça R. (1992). SOM: A Semantic Object Model. Towards an Abstract, Complete and Unifying Way to Model the Real World. *Proceedings of the Third International Working Conference on Dynamic Modelling of Information Systems* (eds. H. G. Sol and A. Verbraeck), pp. 65–93. Delft University of Technology. (113)

Wand Y. and Weber R. (1990). An Ontological Model of an Information System. *IEEE Transactions on Software Engineering*, Vol. 16, No. 11, pp. 1282–1292. (47)

Wand Y. (1989). A Proposal for a Formal Model of Objects. *Object-Oriented Concepts, Databases, and Applications (Kim and Lochovsky, 1989)*, pp. 537–559. (65, 112, 113, 134)

Warnier J.-D. (1976). *Logical Construction of Programs.* Van Nostrand Reinhold, New York, New York. (18)

Wegner P. (1987a). Dimensions of Object-Based Language Design. *Proceedings of the Second ACM Conference on Object-Oriented Programming Systems, Languages and Applications (Meyrowitz, 1987)*, pp. 168–181. (24, 37)

Wegner P. (1987b). The Object-Oriented Classification Paradigm. *Research Directions in Object-Oriented Programming (Shriver and Wegner, 1987)*, pp. 479–560. (43)

Wegner P. (1989). Learning the Language. *Byte*, Vol. 14, No. 3, pp. 245–253. (38)

Wieringa R. J. (1990). *Algebraic Foundations for Dynamic Conceptual Models*. PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands. (62, 72, 73, 87, 88, 98, 99, 112)

Wieringa R. J. (1991). Steps towards a method for the formal modeling of dynamic objects. *Data & Knowledge Engineering*, Vol. 6, No. 6, pp. 509–540.

Winograd T. (1973). Breaking the Complexity Barrier (again). *Proceedings of the ACM SIGPLAN/SIGIR Interface Meeting on Programming Languages – Information Retrieval*, pp. 13–30. Reprinted in Barstow *et al.* (1984), pp.3–18. (26, 29)

Wintraecken J. J. V. R. (1985). *Informatie-analyse volgens NIAM*. Academic Service, Den Haag. In Dutch. (18)

Woods W. A. (1975). What's in a Link. *Representation and Understanding Bobrow and Collins (1975)*, pp. 35–82. Reprinted in Brachman and Levesque (1985), pp. 217–241. (89)

Yourdon E. and Constantine L. L. (1978). *Structured Design*. Yourdon Press, New York. (18, 67)

Zdonik S. B. and Maier D. (eds.) (1990). *Readings in Object-Oriented Database Systems*. Morgan Kaufmann, San Mateo, California. (241, 246, 247, 248, 251)

Index

Summary

This thesis describes research on object-oriented modelling of information systems. Led by the wish to use object orientation during analysis and design, and to create the foundations for future intelligent CASE environments, Chapter 1 announces the development of the INCA Conceptual Object Model (INCA-COM) as the goal of our research (CASE stands for Computer-Aided Systems Engineering; INCA for INtelligent CASE). Chapter 2 introduces the software crisis, and describes the developments in software engineering for managing the complexity of systems development. Nowadays, object-oriented programming languages combine several important abstraction principles in programming, and CASE tools are currently widely available for aiding software engineers during systems development.

To reach our goal we took the decision to base INCA-COM on object-oriented concepts. The reason is twofold: object orientation has emerged in artificial intelligence as a knowledge-representation tool, and in programming as an evolution of different programming paradigms. The requirements for INCA-COM are that (1) the Universe of Discourse (UOD) of an information system can be represented; (2) application domains can be represented; and (3) systems can be modelled.

Chapter 3 assesses object-oriented modelling from both the programming and the modelling perspective. The programming perspective shows that object orientation combines different forms of abstraction, such as identity, state, behaviour, class, class hierarchies, and inheritance. The modelling perspective sheds light on the use and nature of different types of model, i.e., concrete, conceptual, and formal models. Modelling a system consists of finding or constructing an appropriate system that can act as a model for the system being modelled. As a case in point we mention that programming is a way of conceptual modelling resulting in computer programs, i.e., models to be executed on a computer.

In Chapter 4 the concepts of object-oriented modelling are identified. For modelling of information systems a conceptual framework consisting of three distinct parts is described. First, things occurring in the UOD of an information system are modelled by a core of objects belonging to so-called *natural sorts*. Second, objects in an application domain are modelled by describing *role sorts*, i.e., sorts to be instantiated by applications. Third, the active part of an information system consists of *active objects* communicating with each other about things in the UOD and the roles these things are playing. Communication between active objects is based on speech-act theory.

The models in the proposed framework are (1) the *information model* describing the structure of information, including roles; (2) the *event model* describing which events can take place in the object world and which order is imposed on these events; (3) the *behavioural model* describing which objects are active by the performance of particular actions, which give rise to events in the object world; and (4) the *communication model* describing what interactions take place between active objects. The information model and the event model are used for modelling a passive object world, whereas the behavioural model and the communication model are used for modelling a system of active objects. Such a division between a passive-object world and an active-object system may seem counterintuitive to the general idea of object orientation as a unifying and uniform concept. However, such a division is essential in modelling information systems: the nature of an information system entails that it contains a representation of a UOD. Although such represented objects can be active in the real world, their *representations* are under the control of active objects within the information system. Modelling an information system with only one type of object does not aid in developing a clear conceptual model.

Chapter 5 describes four structuring principles, in INCA-COM, for the domain of objects: sort hierarchies, specification, composition, and grouping. These structuring principles are based on meaningful relations between objects and sorts. The inheritance mechanisms in INCA-COM are based on sort hierarchies and specification. We have described three types of inheritance, namely common inheritance, normal inheritance, and structural inheritance.
A system is a composite object with a structure. In INCA-COM, active objects communicating among each other on an object world form an active-object system. We have illustrated the principles by modelling a conference-registration system with associated application domain.

In Chapter 6 we introduce a version concept for INCA-COM, in order to allow objects to evolve over time. We assess the problems in managing versions of

composite objects, and present algorithms for addition and deletion of versions and parts of composite objects.

Chapter 7 presents the modelling of a domain of data-flow diagrams and Chapter 8 the modelling of a domain of spreadsheets.

Chapter 9 describes the architecture of INCATOOL to be used as a supporting tool during the modelling process with INCA-COM. The architecture is based on the model for information systems presented in Chapter 4. The idea behind the architecture is that INCATOOL is an information system having the concepts in INCA-COM as its application domain. The representation of INCA concepts, in terms of the concepts themselves, is called an ontology. INCA-COM has been extended with reflective capabilities. Two types of reflection are provided: structural reflection and descriptor reflection. The former consists of treating sorts are objects, allowing them to be described in the same manner as normal objects. The latter is a form of language reflection, treating descriptors as objects. Thus, the modelling language can be extended with new descriptors and new types of descriptors.

Chapter 10 concludes the thesis by evaluating how the requirements from Chapter 2 are met by INCA-COM. It is stressed that modelling of information systems requires a division between passive objects as representations of things in the UOD of an information system, and active objects as actors, controlling passive objects in a purposeful manner. Both domains benefit from object orientation, but they should not be intermingled.

Samenvatting

Dit proefschrift beschrijft onderzoek naar object-georiënteerd modelleren van informatiesystemen. Geleid door de wens om object-oriëntatie tijdens de fasen analyse en ontwerp in te zetten en om de basis te leggen voor toekomstige intelligente CASE-omgevingen, kondigt hoofdstuk 1 de ontwikkeling van het INCA Conceptueel Object Model (INCA-COM) als het doel van het onderzoek aan (CASE staat voor Computer-Aided Systems Engineering; INCA voor INtelligente CASE). Hoofdstuk 2 introduceert de software-crisis, en beschrijft de ontwikkelingen in de software engineering voor het hanteren van de complexiteit van systeemontwikkeling. Tegenwoordig combineren object-georiënteerde programmeertalen diverse belangrijke abstractieprincipes van programmeren, en CASE-tools zijn tegenwoordig beschikbaar ter ondersteuning van systeemontwikkeling.

Om ons doel te bereiken hebben we de beslissing genomen INCA-COM te baseren op object-georiënteerde concepten. De reden is tweevoudig: object-oriëntatie is ontwikkeld in de kunstmatige intelligentie als middel voor kennisrepresentatie, en in het programmeren als een combinatie van verschillende abstractievormen. INCA-COM moet concepten bieden voor het modelleren van (1) het Universe of Discourse (UOD) van een informatiesysteem; (2) applicatiedomeinen; en (3) een systeem.

In hoofdstuk 3 wordt object-georiënteerd modelleren beschouwd vanuit het programmeerperspectief en het modelleerprespectief. Het programmeerperspectief toont dat object-oriëntatie verschillende abstractievormen combineert, zoals identiteit, toestand, gedrag, klassen, klassehiërarchieën, en overerving. Het modelleerperspectief geeft inzicht in het gebruik en de aard van verschillende typen van modellen, te weten concrete, conceptuele en formele modellen. Het modelleren van een systeem komt er op neer een geschikt systeem te vinden of te construeren dat als model van het te modelleren systeem kan dienen. Als voorbeeld noemen we dat programmeren een manier van conceptueel mo-

delleren is die resulteert in computerprogramma's, ofwel modellen die door een computer kunnen worden uitgevoerd.

In hoofdstuk 4 zijn de concepten van object-georiënteerd modelleren bepaald. Voor het modelleren van informatiesystemen beschrijft het hoofdstuk een conceptueel raamwerk bestaande uit drie onderdelen. Ten eerste worden entiteiten in het UOD van een informatiesysteem gemodelleerd door een kern van objecten die tot zogenoemde natuurlijke soorten behoren. Ten tweede worden objecten in een applicatiedomein gemodelleerd door rolsoorten te beschrijven. Dit zijn soorten die door applicaties worden geïnstantieerd. De kern en een applicatiedomein vormen te zamen een objectenwereld die de representatie van een UOD vormt. Ten derde bestaat het actieve deel van een informatiesysteem uit actieve objecten die onderling communiceren over de entiteiten in het UOD en de rollen die deze entiteiten spelen. De communicatie is gebaseerd op taalhandelingstheorie (speech-acttheorie).

De modellen in het voorgestelde raamwerk zijn (1) het *informatiemodel* waarmee de structuur van de informatie in een systeem wordt beschreven; (2) het *gebeurtenismodel* waarmee de gebeurtenissen die in een objectenwereld kunnen plaatsvinden worden beschreven; (3) het *gedragsmodel* waarmee wordt beschreven welke objecten actief zijn en bepaalde acties uitvoeren, die resulteren in gebeurtenissen in de objectenwereld; en (4) het *communicatiemodel* waarmee wordt beschreven welke interacties er zijn tussen de actieve objecten. Het informatiemodel en het gebeurtenismodel worden gebruikt voor het modelleren van passieve-objectenwereld, en het gedragsmodel en het communicatiemodel worden gebruikt voor het modelleren van een systeem van actieve objecten. De splitsing tussen een passieve-objectenwereld en een actieve-objectensysteem lijkt in strijd met het algemene idee dat object-oriëntatie een uniforme manier is om zowel gegevens als processen te modelleren. Toch is bij het modelleren van een informatiesysteem een dergelijke splitsing essentieel. De aard van een informatiesysteem bepaalt dat zich in het informatiesysteem een representatie bevindt van een UOD. Alhoewel dergelijke representatie-objecten actief kunnen zijn in de echte wereld, staat hun representatie onder besturing van de actieve objecten in het informatiesysteem. Modelleren van een informatiesysteem met slechts één type van objecten is niet bevorderlijk voor het ontwikkelen van een helder conceptueel model.

Hoofdstuk 5 beschrijft de vier structuringsprincipes in het domein van objecten: de soorthiërarchie, specificatie, compositie, en groepering. De structureringsprincipes zijn gebaseerd op betekenisvolle relaties tussen objecten en

soorten. De mechanismen voor overerving in INCA-COM zijn gebaseerd op soorthiërarchieën en specificatie. Er zijn drie typen van overerving onderscheiden, namelijk gemeenschappelijke, normale en structurele overerving.
Een systeem is een composiet object met een bepaalde structuur. In INCA-COM vormen actieve objecten die met elkaar communiceren over een objectenwereld een systeem van actieve objecten. De principes worden geïllustreerd door het modelleren van een conferentieregistratiesysteem met bijbehorend applicatiedomein.

In hoofdstuk 6 introduceren we een versieconcept voor INCA-COM, om objecten de mogelijkheid te geven in de tijd te veranderen. We behandelen de problemen die optreden bij het beheren van versies van composiete objecten, en beschrijven algoritmen voor het toevoegen en verwijderen van versies en deelobjecten van composiete objecten.

Om de modelleerconcepten in INCA-COM te illustreren worden in het proefschrift diverse voorbeelden gegeven. Hoofdstuk 7 presenteert het modelleren van een domein van data-flow-diagrammen en hoofdstuk 8 het modelleren van een spreadsheet.

Hoofdstuk 9 beschrijft de architectuur van INCATOOL, een hulpmiddel bij het modelleren met INCA-COM. De architectuur is gebaseerd op het INCA-model van een informatiesysteem. Het idee achter de architectuur is dat INCATOOL een informatiesysteem is met de concepten uit INCA-COM als applicatiedomein. De representatie van INCA-concepten, in termen van de concepten zelf wordt een ontologie genoemd. INCA-COM is uitgerust met twee mogelijkheden voor reflectie. Ten eerste worden soorten behandeld als objecten, zodat ze op dezelfde manier als normale objecten kunnen worden beschreven. Ten tweede worden de descriptoren als objecten behandeld. Hierdoor ontstaat de mogelijkheid de modelleertaal uit te breiden met nieuwe descriptoren en nieuwe typen van descriptoren.

Hoofdstuk 10 besluit het proefschrift door te evalueren op welke wijze INCA-COM tegemoet komt aan de eisen die in hoofdstuk 2 zijn gesteld. Tenslotte wordt benadrukt dat het modelleren van een informatiesysteem een scheiding noodzakelijk maakt tussen enerzijds passieve objecten als representatie van entiteiten in het UOD van het informatiesysteem, en anderzijds actieve objecten als actoren, die de passieve objecten met een bepaald doel besturen. Beide domeinen hebben profijt van object-georiënteerd modelleren, maar moeten niet door elkaar worden gehaald.

Curriculum Vitae

Harm Bakker werd op 9 februari 1964 geboren in Uitgeest. Van 1976 tot 1982 volgde hij een gymnasium-β opleiding aan het Murmellius Gymnasium in Alkmaar. Van 1982 tot 1987 studeerde hij Informatica aan de Technische Universiteit Delft. In 1985 werd hij voor een jaar studentassistent bij de vakgroep Toegepaste Taalkunde en werkte hij aan programmatuur voor het automatisch vertalen van natuurlijke taal. In december 1986 deed hij mee aan een programmeerwedstrijd voor studenten met een demonstratieprogramma voor het grafisch ontwerpen en manipuleren van programma-structuurdiagrammen. Het idee achter dit programma leidde tot zijn afstudeeropdracht bij het TNO-Instituut voor Toegepaste Informatica. Hier ontwierp en implementeerde hij een editor voor het omzetten van pseudocode in een grafisch programma-structuurdiagram. Na zijn afstuderen maakte hij van dat systeem een hulpmiddel voor onderwijsdoeleinden aan een MTS.

In 1988 trad hij in dienst van de Vakgroep Informatica van de Rijksuniversiteit Limburg in Maastricht, eerst als onderzoeksassistent later als toegevoegd onderzoeker. Hier heeft hij tot 1993 onderzoek verricht op het gebied van object-georiënteerd modelleren onder supervisie van Prof. dr. H. J. van den Herik en Dr. P. J. Braspenning. De resultaten van het onderzoek zijn in dit proefschrift beschreven. Hij heeft diverse artikelen gepubliceerd en voordrachten gehouden op congressen in binnen- en buitenland. Tevens voerde hij in 1992 een project uit voor de politieregio Limburg Zuid, waarin de mogelijkheden van regionale multimediale informatievoorziening werden onderzocht.

Sinds 1993 is hij wetenschappelijk onderzoeker bij het Telematica Research Centrum in Enschede. Hij doet daar onderzoek naar telematicasystemen en de maatschappelijke gevolgen van telematica. Zijn aandachtsgebieden zijn telematicatoepassingen in de juridische wereld, herontwerp van bedrijfsprocessen, multimediasystemen, telematica in verkeer en vervoer, en Internet-toepassingen.