Obstacles in Object-Oriented Software Development

Mehmet Aksit & Lodewijk Bergmans

TRESE project, Dept. of Computer Science, University of Twente,
P.O. Box 217, 7500 AE, Enschede, The Netherlands
{aksit, bergmans}@cs.utwente.nl

Abstract
Recently, a considerable number of object-oriented software development methods have been introduced to produce extensible, reusable, and robust software. We have been involved in the development of a large number of pilot applications to form our own view on object-oriented methods. Although our experiences confirmed the claims about the benefits of object-oriented methods, we identified a number of important obstacles that are not addressed by current methods. This paper summarizes these obstacles and evaluates them with respect to our pilot applications. The aim of this paper is to make software engineers aware of problems they may encounter during object-oriented development, and to inspire researchers to initiate new research activities.

1. Introduction

A significant number of object-oriented methods [1-10] have been introduced during the past several years. These methods claim that the object-oriented approach creates highly extensible, reusable and robust software. After studying most of the state-of-the-art methods [1-10], we developed our own method. We retained certain aspects of existing methods that we considered to be the most promising. To obtain a more realistic view, we have been involved in a large number of pilot applications varying from administrative systems to process automation [15-35]. Like other practitioners [55], we experienced that object-oriented methods substantially do improve productivity. On the other hand, we identified a number of important shortcomings within current object-oriented methods.

This paper summarizes the state-of-the-art object-oriented methods that we studied, and identifies a number of problems that may appear during software development. This work can be beneficial for at least two purposes. Firstly, since none of the methods [1-10] address these problems explicitly, software engineers can now prepare themselves to deal with these specific problems, when encountered in object-oriented development. We, in fact, spent a considerable amount of time just trying to identify what was going wrong with our analysis and design method. Secondly, the problems discussed in this paper may inspire researchers to initiate new research activities.

This paper is organized as follows. Section 2 gives background information and a short overview of the state-of-the-art methods that we investigated. The pilot applications that we have been involved with are briefly detailed in section 3. Section 4 presents problems encountered in current methods. Section 5 evaluates the pilot applications with respect to the identified problems. Section 6 briefly refers to our related research activities. Finally, section 7 gives conclusions.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1992 ACM 0-89791-539-9/92/0010/0341...$1.50

2. Background and Related Work

2.1. Object-Oriented Software Development

The target of object-oriented software development is the *object-oriented decomposition* of user's needs into executable language constructs. The object-oriented decomposition process can be sub-divided into *analysis*, *design* and *implementation* phases.

In the analysis phase, the software engineer aims at

OOPSLA'92, pp. 341-358

precise and correct identification and specification of the user's needs in an *understandable* way. This phase is mainly directed by the user's problem, or the so called *real-world domain*. In the design phase, the software engineer revises and extends the analysis model by specifying how the user requirements can be realized. The implementation phase details the design model by means of specific language constructs.

An important characteristic of object-oriented development is that the analysis, design and implementation phases adopt similar models, although each phase has a different emphasis. This enables a smooth transition between the different phases. Each phase in object-oriented software development can be divided into three sub-components: *preparatory work, structural relations* and *object interactions*.

2.1.1. Preparatory Work

The preparatory work in the analysis phase consists of mapping between the *real world* entities and the entities in the analysis model: objects[1]. This mapping process is called *domain analysis*. Another important activity is the partitioning of the problem domain into manageable sub-components called *subsystems*.

As an example of the application of real-world knowledge, most theory books introduce classification hierarchies to organize knowledge. These hierarchies can usually be directly represented as object-oriented class hierarchies. Since the basic aim of theory is to introduce sound and generic solutions, the software engineer can then create highly reusable inheritance hierarchies.

The preparatory work for the design phase consists mainly of mapping the analysis model to a design environment. In the design environment, libraries of predefined classes may be available. The objects identified in the analysis phase are mapped, as far as possible, to these predefined classes. Another preparatory activity of the design phase is collecting and formulating design requirements that may not have been relevant in the analysis phase. For example, efficiency and alternative realizations are typical design requirements.

The preparatory work for the implementation phase is concerned with the environment within which the design will be implemented. For example, the properties and restrictions of the implementation language may require a non-trivial mapping between the design model and the language model.

2.1.2. Structural Relations

Object-oriented analysis concentrates on a few specific types of relations. The two most important relations are *classification* and *part-of* relations. In addition, some methods [3, 8] introduce *associations* which describe relations other than classification and part-of relations among classes.

Classification relations indicate that one class may be considered as a generalization or specialization of another class. This is a common way of making abstractions, resulting in classification trees, with the most general cases at the root of the tree, and the most specialized and dedicated cases as the leaves.

Part-of relations may reflect part-whole relations such as wheels are part of a whole car, or for example, organizations consist of departments. Subsystem partitioning in the analysis phase often matches part-of relations.

In the design and implementation phases, classification structures manifest themselves as class-inheritance hierarchies. During design and implementation, the identified structural relations may be modified for several reasons. Design rules may result in restructuring to obtain better modularity, encapsulation, extensibility and reusability, and mapping multiple inheritance to single inheritance.

2.1.3. Object Interactions

As where the structural relations define the architecture of a system, the dynamic behavior of the system is realized by object interactions. These are represented by *message connections*[2]. A message connection simply indicates that two objects communicate. Message connections are usually identified after structural relations have been determined. In the design and implementation phases, object interactions

[1] In the analysis/preparatory phase, the term objects may correspond to both classes or instances.

[2] The terms *instance connections* and *collaborations* are used by some methods.

may be modified as a result of various design decisions, for instance to improve reusability.

In general, software engineering principles such as encapsulation and modularity tend to collide with performance requirements. As a result, the interactions between objects may be modified during design and implementation phases to fulfil performance requirements.

2.2. Overview of Object-Oriented Analysis and Design Methods

This section gives a short summary of the methods that we have referred to as the state-of-the-art object-oriented methods [1-10].

Booch's Object-Oriented Design method [1] introduces notations for representing *class utilities*, *class categories*, *classes*, *state transitions*, *objects*, *modules*, *subsystems*, *processors*, *devices*, and *processes*. Each of these elements can be represented graphically, as so-called *icons*, and textually, by so-called *templates*. Templates are less readable but more detailed than icons. Class utilities represent the so called *free-programs* which are program blocks that are not included in a class definition. Class categories are similar to Ada modules [36] and are used to organize classes. The other elements are standard terms and therefore should be self-explanatory. In addition, this method defines detailed notations for representing various kinds of visibility and synchronization constraints of object interactions. The method proposes the following steps: Identifying classes and objects, identifying semantics of classes and objects, identifying relationships between classes and objects, and implementing classes and objects.

Object-Oriented Analysis and Top-Down Software Development [2] introduces a top-down approach using *ensembles*. Ensembles are subsystems and are comparable to objects. The main difference is that they may have internal concurrency as where objects do not. This method consists of three major components called *information*, *state* and *process* models. The information model is an object model with structural relations. The state model represents the dynamic behavior within an object, which is determined by *operation triggers*. The process model specifies the object interactions, describing all the causal connections between objects. These three components were adopted by earlier methods such as Object-Oriented Systems Analysis [13].

Object-Oriented Analysis & Design by Coad & Yourdon is split in two separate parts: the analysis part [3] and the design part [4]. The analysis part defines five vertical components called *layers*. Layers are labeled as *subjects*, *classes* and *objects*, *structures*, *attributes*, and *services*. Subjects are subsystems with simple semantics. Structures are inheritance and part-of relations. Attributes characterize objects. In practice, attributes may be implemented either as local variables, or computed through operations. Services represent a set of operations provided by an object.

Coad and Yourdon's Object-Oriented Design method applies the same five layers as utilized in their Object-Oriented Analysis method. In addition, four horizontal components are introduced resulting in a matrix architecture. Horizontal components are labeled as *problem domain*, *human interaction*, *task management* and *data management*. The problem domain component is obtained from the analysis model. The human interaction component is used to design user interfaces. The task management component aims at defining object interactions. The objective of the data management component is to create persistent objects.

Johnson and Foote provide a set of design rules [5] in four categories: when to define new classes, how to improve object interfaces, how to construct abstract classes, and how to identify reusable class hierarchies called *frameworks*.

The Demeter system [6, 7] is a Computer-Aided Software Engineering (CASE) environment which introduces a set of automated tools to ease the software development process. In the Demeter method, the software engineer first identifies object instances. The system provides a tool to infer the so-called *class dictionaries* from these instances. Class dictionaries are then optimized with respect to certain criteria. The system is also able to adapt class dictionaries as new object instances are introduced. Generally, in part-of structures it may be required to invoke an operation on a number of parts to insure the consistency of the whole. Demeter provides a tool to generate the required repeated operations for each part. In addition,

the Demeter system is governed by a set of design rules. For example, there are rules for managing the software growth and to minimize the object-interaction patterns. The latter is also known as the law of Demeter [6]. When the class dictionaries are finalized, the system generates programs in the C++ language [47].

Object-Oriented Modeling and Design (OMT) [8] introduces three models and a method to apply them. In this method, first the *object model* is constructed. The object model of OMT is analogous to the object models of other methods except for a clear emphasis is placed on associations between objects. The second model is called the *dynamic model* and is based on state diagrams. The third model is called the *functional model* and consists of *data-flow* diagrams. These three models, are in fact, very similar to the models of Yourdon's Structural Analysis [14], but the design phase of OMT is centered more around the object model.

The Responsibility-Driven approach [9] defines six activities. The first activity deals with the identification of classes and class hierarchies. As a second step, operations of each class, the so-called *responsibilities* are specified. In the third step, object interactions termed *collaborations* are identified and specified. The objective of the fourth step is to improve reusability by further refining class hierarchies. The fifth step aims at grouping classes into subsystems by constructing so-called *collaboration* graphs based on object interactions. The last step is devoted to the formal specification of object interfaces, called *protocols*.

Object-Oriented Role Analysis, Synthesis and Structuring [10], introduces five steps. In the first step, so-called *roles* and *behaviors* are identified. Roles and behaviors roughly correspond to instance objects and classes, respectively. Then roles of different subsystems are compared to identify class hierarchies. The second step aims at specifying object interactions. The third step details class implementations. The next step is to define a meta model which includes rules for structural relations and object interactions. The final step specifies the system initialization rules.

3. Our Experience

3.1. The Approach

Several years ago, when we decided to start a new research project on object-oriented software engineering, we wanted to identify research topics based on the benefits and shortcomings of the state-of-the-art object-oriented methods. The best way to determine these was to apply object-oriented techniques to realistic examples in the form of pilot studies. At that time, however, not so many methods were available. We studied Booch's earlier publications [11, 12], and defined our own method. When new methods were introduced [1-10, 13], we updated our method carefully [60]. Our intention was to combine what we considered to be the best of these methods. For example, we used Coad and Yourdon's layered approach and their hints for object identification [3, 4], adopted Booch's notation [1], employed the rules of Johnson and Foote [5], applied the Law of Demeter [6], incorporated the associations and the dynamic model of OMT [8], and included the collaboration graphs of the Responsibility Driven approach [9].

We used the same documentation means in each of our pilot studies. Whenever we encountered a problem, we examined the methods [1-10] to understand how these problems were addressed by them. In cases where we could not find a solution to our problem, we referred to other related research work. At the end of each pilot application, we tried to categorize and generalize the problems that we experienced. These problems were explicitly used to initiate our research activities.

3.2. Pilot Applications

In this section, we briefly list the pilot applications to provide a framework to compare and categorize problems with respect to the type of applications in which they may be encountered. We further classify these applications into external and internal assignments. External assignments were carried out under conditions that were not fully under our control.

3.2.1. External Assignments

Administration system for social security services: To gain experience in object-oriented analysis and design within the area of data-intensive applications, an

object-oriented model of an administrative system for disablement insurance laws was realized [25].

Network database: This assignment aimed at the construction of a simple network database in relation to the workshop "Different Paradigms in Software Development" which was organized at the University of Twente [22].

Chemical process control system: A process control system for a distillation process was developed at the Faculty of Chemical Engineering, University of Twente[26].

Mechatronic modeling system: This activity aims at the design and implementation of a mechatronic modeling environment using object-oriented principles and is being realized at the Faculty of Electrical Engineering, University of Twente [21].

Intelligent tutoring system: This work is being carried out in conjunction with the Faculty of Applied Education, University of Twente to design and implement an intelligent tutoring system in the area of software engineering [30].

3.2.1. Internal Assignments

Concurrent processing and synchronization: This study [15] contained various examples to demonstrate the capabilities of the object-oriented mechanisms to solve some well-known synchronization problems. We chose examples in five different categories: process communication and coordination, implementation of concurrent function evaluations, scheduling, resource management, and client-server communication.

Distributed office system: A distributed office system was simulated [16, 17] on a workstation to illustrate the object-oriented approach for the following applications: layered communication architectures, security protocols, asynchronous communications, and atomic transactions.

Object-oriented language and environment implementation: The object-oriented language Sina [37] and its programming environment were implemented by a number of students. This implementation included a compiler [29], object-manager, concurrent processing constructs [18], interpreter [34], atomic transactions, deadlock detection, type checking [23], graphical programming environment [27, 31] and remote invocations [20].

Parser generator for Smalltalk: This study [28] aimed at the development of a lexical analyser and parser generator for Smalltalk [49].

Distributed operating system design: Various persons were involved in distributed system design. Some activities were carried out during the construction of a distributed version of the Sina language. Other important activities were completed in the area of distributed object management [19, 24, 35]

Temperature control system: This study was selected from some example problems presented by Booch [1] for the purpose of identifying the problems associated with control system design [32].

Intelligent mail: An intelligent mail application [33] was developed to experiment with the Ontos object-oriented database system [61].

4. Problems

4.1. Problems Related to the Preparatory Work

In the analysis phase we experienced six major problems in the preparatory work. The first two problems are related to domain analysis and are presented in section 4.1.1. The remaining four problems are related to subsystems and are explained in section 4.1.2.

4.1.1. Domain Analysis

Identification of Problem-Domain Structures:

In each of the pilot studies we carefully considered the available domain knowledge. We tried to identify classifications in the problem domain to map them directly into inheritance hierarchies. For some well-structured and theoretically founded domains, indeed, we could identify highly general and reusable hierarchies. However, there were many problems which did not demonstrate any clear structure. Quite often underlying theories of large systems are not completely understood, and it is difficult to define reusable hierarchies for these types of systems. One may not expect software engineers to organize inheritance hierarchies any better than their understanding of the classifications within the theory itself.

Dealing with Excessive Domain Objects:

Many object-oriented methods [1, 3, 8, 9] expound the

benefits of using domain knowledge while preparing the user's requirement specifications. Integrating the domain knowledge with these specifications, however, can create an excessive number of objects, although only a few of these objects may be relevant to the problem being analyzed.

For example, in the *network database* assignment [22], we considered graph-theory as domain knowledge since networks can easily be modeled as graphs. We used two books for this purpose [54, 63]. With these two books, and the user's specification, we identified a number of classes representing different types of graphs. Based on the features of these classes, we have constructed the inheritance hierarchy as depicted by figure 1.

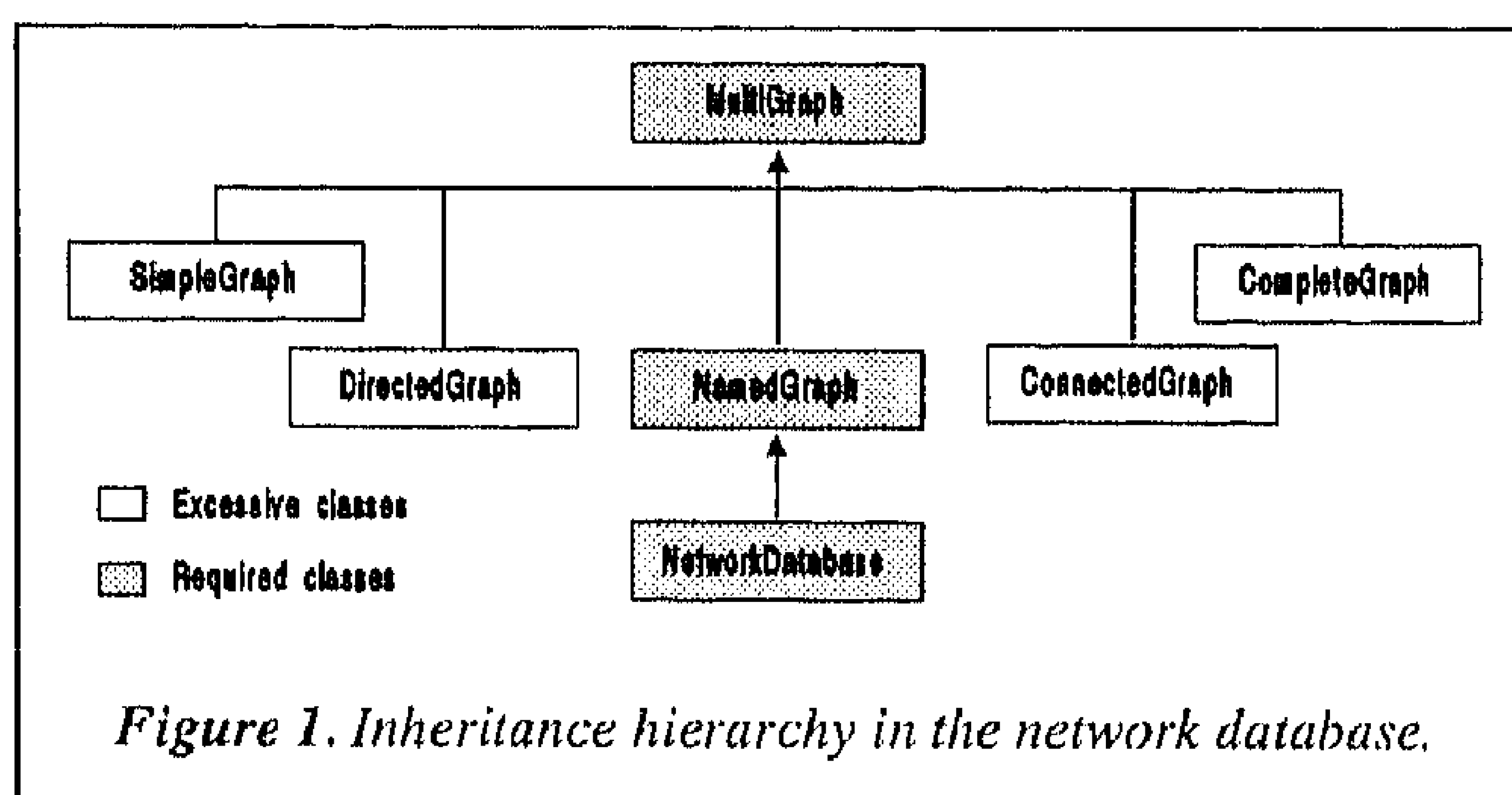

Figure 1. *Inheritance hierarchy in the network database.*

Although this graph hierarchy can be reused in many different applications, for our specific problem, only classes *MultiGraph*, *NamedGraph* and *Network-Database* were relevant. However, until the inheritance hierarchy was constructed, we had to deal with these excessive objects. Obviously, this problem can be even more difficult to handle in large and complex applications.

4.1.2. Subsystems

Early Decomposition:

All the methods [1-10] emphasize the importance of proper object identification. One important problem in the object identification process is how to deal with a large number of objects. This forces the software engineer to partition the application into subsystems prior to the object identification phase, and only then consider the objects within the context of these subsystems. The prior identification of subsystems is also proposed by some methods [2, 3].

Identification of subsystems can be done more accurately after structural relations and object interactions have been determined. This is because subsystem boundaries are largely determined by inter-object relations and interactions. Some methods, therefore, defer the subsystem identification step to a later phase [1, 8, 9]. Coad and Yourdon's analysis method [3] considers both alternatives.

The dilemma here is that if the software engineer does not identify subsystems before starting with object identification, then the project probably becomes unmanageable. On the other hand, if the software engineer identifies subsystems prior to object-identification, then the defined subsystem boundaries may not be optimal.

Subsystem-Object Distinction:

Almost all methods consider subsystems as being different from objects. Some methods consider subsystems as a collection of objects and assign simple semantics to them [3, 4, 8]. Booch [1] introduces two separate constructs, *class categories*, which are similar to Ada modules [36], and *subsystems*, which are basically collections of objects. In the Responsibility-Driven approach [9], subsystems are identified using the so-called *collaboration graphs*; objects that have frequent interaction are placed into the same subsystem. De Champeaux introduces *ensembles* as subsystems [2] and emphasizes their use in a top-down manner. Although ensembles appear to have similar semantics to objects, they are different in that ensembles may contain concurrently active objects while objects cannot.

We have experienced difficulties due to the distinction of subsystems from objects. During the analysis phase, objects may eventually act as subsystems if their internal structures get too complicated. Similarly, subsystems may be defined as objects if their functionality can be structured in a class hierarchy and reused in different applications. Since object-oriented methods are largely iterative, one may need to convert subsystems into objects (or vice versa). This requires modifications to the semantics of these constructs, which is obviously very error prone.

Commonality versus Partitioning:

In general, subsystems are assigned to different soft-

ware engineers, and/or are handled sequentially one at a time. In order to identify class hierarchies, however, the software engineer must compare features of objects. Since subsystems partition the system, classes belonging to the same hierarchy can be scattered over different subsystems. This can make the task of finding the proper inheritance hierarchy very difficult. This problem is also considered in Role Analysis, Synthesis and Structuring [10].

As an example, consider the sensor hierarchy of the temperature control system as shown by figure 2.

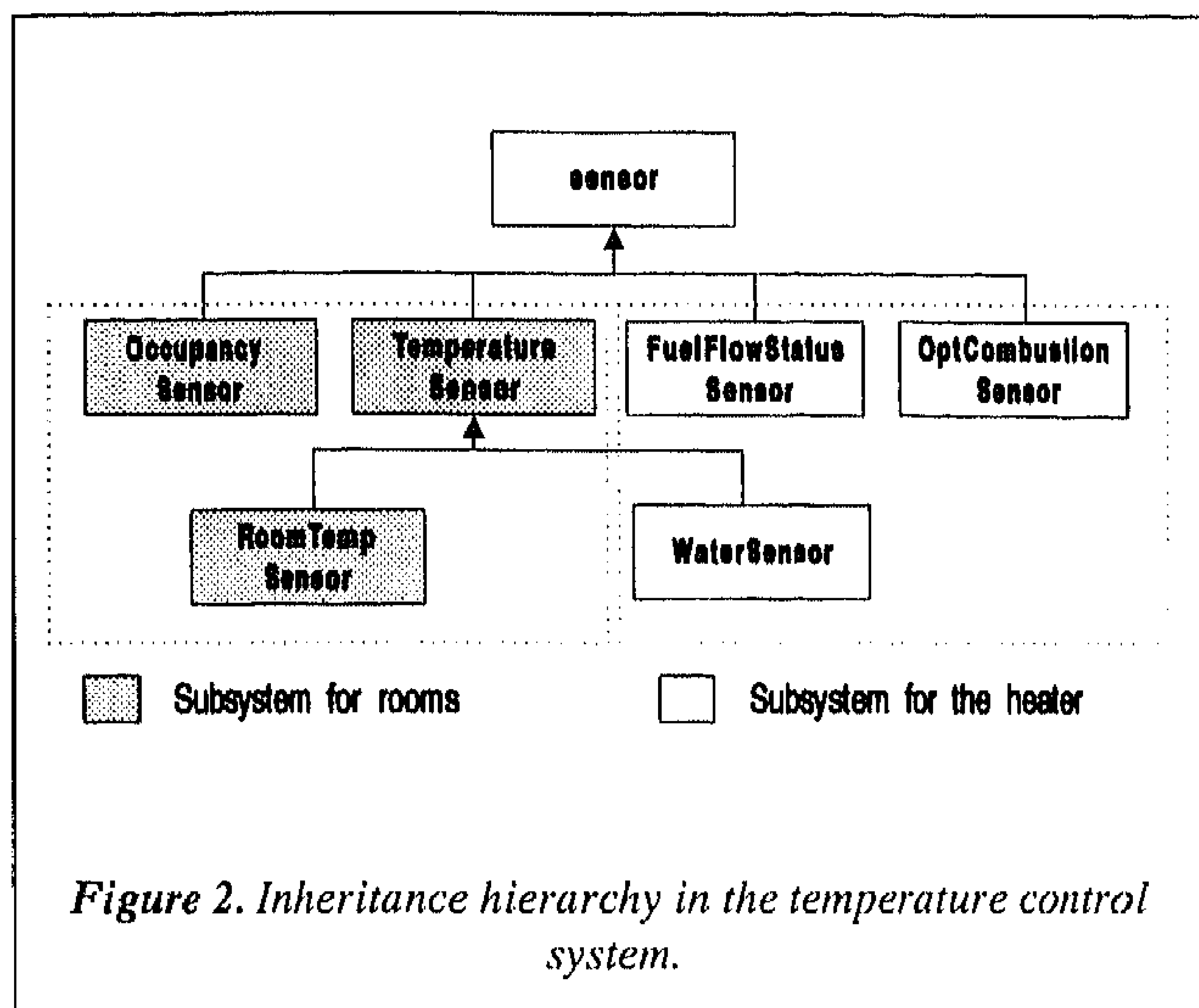

Figure 2. Inheritance hierarchy in the temperature control system.

The sensor hierarchy classifies the various sensors in the system, like *temperature sensors*, *room occupancy sensors*, etc., but, these sensors lie in separate subsystems, such as *rooms* and *heater*. However, to build the proper inheritance hierarchy, objects within all subsystems must be considered.

Subsystems Identification Using Object Interactions:
Some object-oriented methods [2, 9] introduce subsystems mainly for structuring interactions among objects. Software engineers may also aim at creating reusable subsystem modules and try to incorporate functionality into partitioning. However, most existing methods only provide intuitive techniques for subsystem identification. This is not sufficient for large systems, since interactions can be too complex and subject to changes. We therefore believe that proper object-oriented subsystem identification is only feasible if the software engineer is equipped with tools that identify and configure subsystems automatically. This requires algorithmic techniques.

One of the major goals in distributed system design is to partition applications in such a way that the cost of distribution is minimal [46]. It turns out that these problems are analogous not only to the subsystem partitioning problem, but also to software modularization problems, in general.

4.2. Problems Related to Structural Relations

4.2.1. Sharing Behavior with State

In general, instances store states as where classes behave as templates, defining the common features of their instances. For certain applications, it may be desirable that the state shared by instance objects affects their operations defined at the class level. In current object-oriented models, however, classes can not conveniently express features that are affected by shared state information stored in their instances.

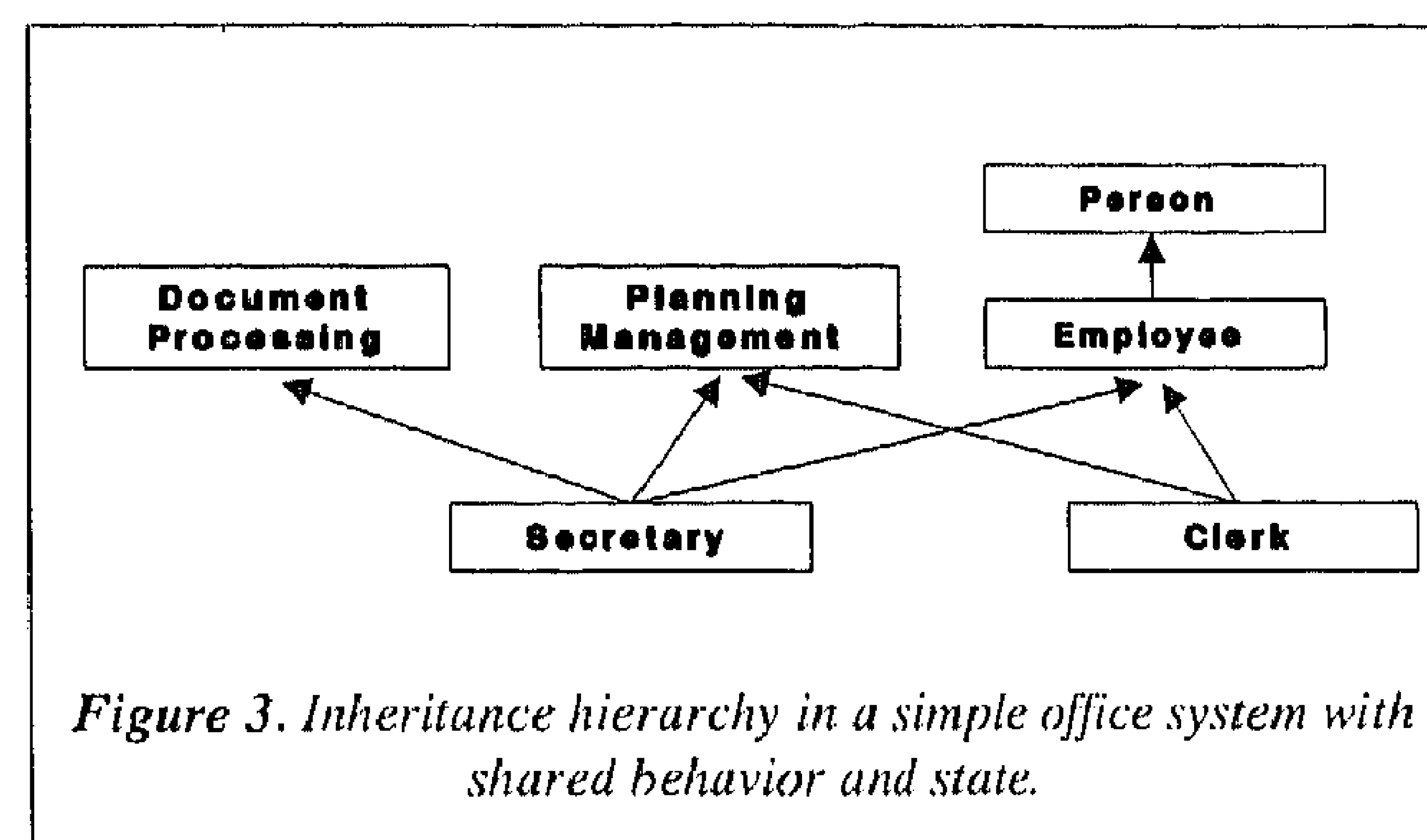

Figure 3. Inheritance hierarchy in a simple office system with shared behavior and state.

This problem is exemplified by the simple office system as shown in figure 3. In this example, classes *Secretary* and *Clerk* represent office employees. According to the company strategy, every clerk and secretary is responsible to carry out a number of tasks in a certain order (e.g. based on company strategies). In addition, secretaries are responsible for processing documents. Both secretaries and clerks keep their documents by themselves, and in certain cases it may be desirable that they take their own initiative to overrule global company strategies.

In figure 3, classes *Secretary* and *Clerk* inherit from classes *Employee* and *PlanningManagement*, and class *Employee* inherits from class *Person*. In addition, *Secretary* inherits from *DocumentProcessing*. This hierarchy allows classes *Secretary* and *Employee* to inherit management operations that reflect the company

policy, for example, to order their tasks. To provide the ordering operation, class *PlanningManagement* implements the operation *sort* as shown in figure 4.

In line 4, the operation *sort* obtains the size of the list by invoking the operation *size* on *self*. Here the expression *self.size* represents a message expression where *self* is the receiver object, either an instance of class *Secretary* or *Clerk*, and *size* is the operation to be invoked on this instance. This is because the operation search always begins from the receiver object and continues towards the higher classes in the inheritance hierarchy. After determining the size of the list in the inner for-loop (lines 5-11), two adjacent elements of the list are retrieved and compared (line 7). The comparison operation is carried out on *self* by invoking the operation *compare*. Normally, the operation *compare* is inherited from class *PlanningManagement*, but can be overridden by the subclasses *Secretary* and *Clerk*. If the operation *compare* evaluates to *true*, then the two adjacent elements are interchanged (line 10).

```
sort
  "sorting the office tasks"

(1)  i, j: Integer; "Indices"
(2)  shift: Boolean;
(3)
(4)  for i:= (self.size) downto 2 do
(5)     for j:= 1 to i-1 do
(6)     begin
(7)        shift:=self.compare(self.activity(j),
(8)                            self.activity(j+1) );
(9)        if shift
(10)       then self.swapActivity(j, j+1);
(11)    end;
```

Figure 4. *Implementation of the operation sort of class PlanningManagement.*

The operation *compare* uses the current strategical information of the office which is adapted from time to time. Ideally class *PlanningManagement* would store this information locally. Retrieving this information from an external object would not be desirable for two reasons. First of all, strategical information should be private and encapsulated in class *Planning-Management*. Secondly, message invocation on an external object would transfer the identity *self* to the identity of the external object since *self* always refers to the object that receives the message. Therefore, it would be impossible for the external object to request additional information from the instances of classes *Secretary* or *Clerk* by sending a message to *self*. In addition, overriding the methods of the external object by classes *Secretary* and *Clerk* would no longer be possible. This, the so called *self-problem*, is defined by Lieberman in [56].

Our implementation in figure 4 can not achieve the requirements of the office system. Since class *PlanningManagement* is a class, it is not suitable to store the strategical information locally. Using *class variables*, as provided by Smalltalk [49], is not appropriate because we may want to create several instances of class *PlanningManagement*, each with its own strategical information.

One proposed solution to this problem is to use a delegation hierarchy instead of inheritance [56]. Delegation is mechanism that allows objects to redirect messages to one or more designated objects. The delegated object becomes part of the *extended identity* of the delegating object, thereby solving the self-problem. We are convinced that both inheritance and delegation must be employed to model shared behavior. Current object-oriented methods [1-10], however, do not support delegation hierarchies[3] and therefore do not provide convenient mechanisms to model shared behavior with state.

4.2.2. Atomicity versus Inheritance

Atomic actions have proven to be a useful mechanism to preserve consistency [58]. *Serializability* and *indivisibility* [50] are the two important properties of atomic actions. Serializability means that if several actions are executed concurrently, they manipulate the affected data as if they were executed serially in some order. Indivisibility means that either all or none of the atomic actions are performed. Atomic actions lessen the burden on software engineers by providing them with a high-level mechanism to deal with the effects of concurrency and failures. Serializability makes cer-

3 Some methods advise the use of a delegation hierarchy in case inheritance is not suitable [4, 9]. Delegation, as presented in these methods is not the true delegation, because it is based on operation invocation on a delegated object, and therefore does not support extensibility through the pseudo variable *self*.

tain that concurrent actions do not interfere with each other. Indivisibility guarantees that when an atomic action is affected by a failure, its partial results are undone.

Atomic actions were first adopted in databases as *transactions* [48, 50]. Database applications are typically characterized by multiple accesses and permanent updates to shared data, and they require mechanisms that guarantee data integrity. Object-oriented methods [1-10] do not support the use of transactions.

Database transactions are somewhat limited since they are provided to users through the database but they are not applicable to arbitrary objects. There has been a considerable effort to provide transaction mechanisms as a general tool to construct distributed systems. Most of these systems provide transactions as an operating system support in the form of system calls or run-time libraries [64, 70]. Only a few languages, like Argus, support transactions within a language [57].

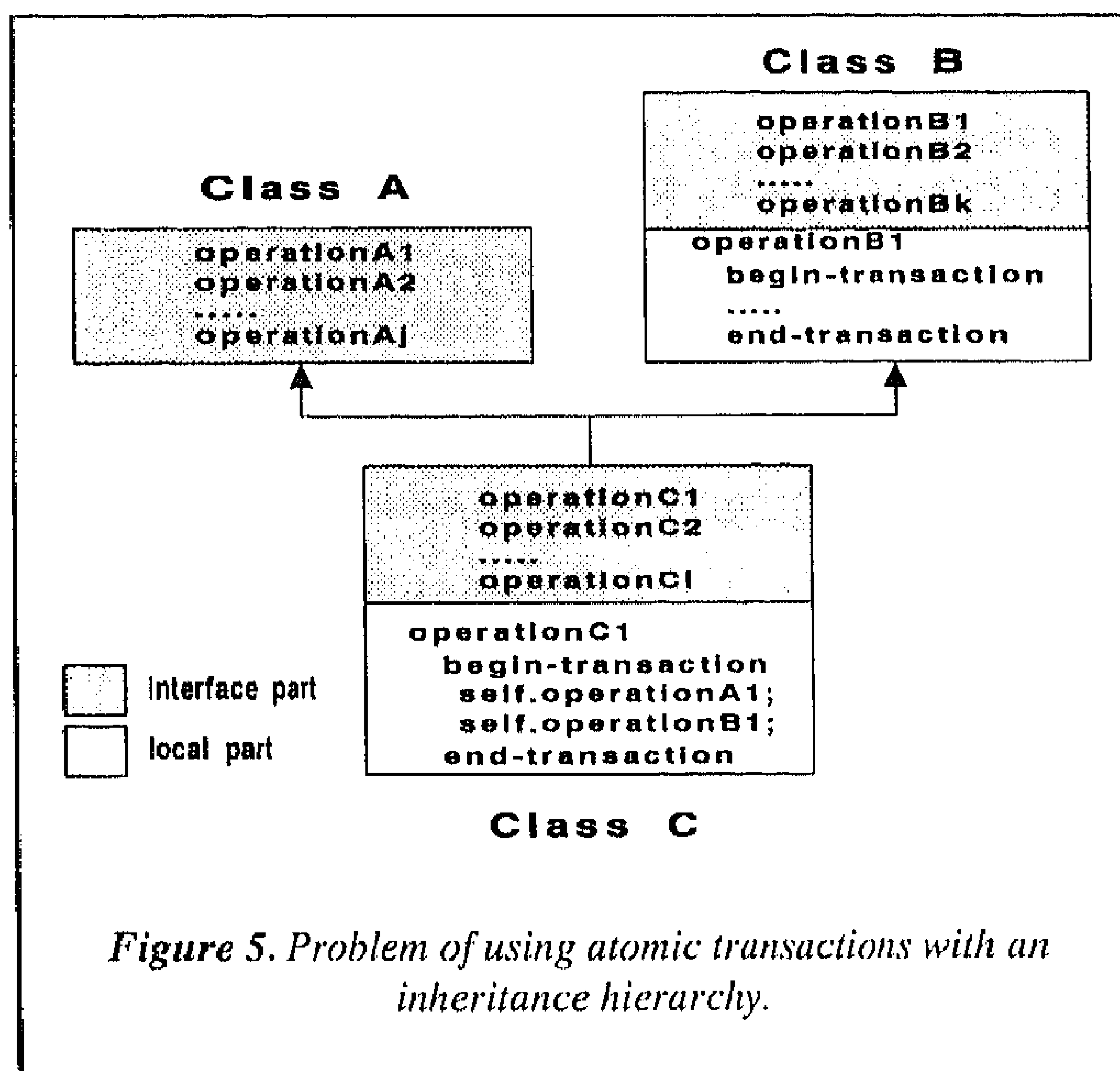

Figure 5. Problem of using atomic transactions with an inheritance hierarchy.

Most object-oriented database systems provide transactions for a program block by delimiting it with `begin-transaction` and `end-transaction` like constructs, or by making the complete operation body atomic [43, 53, 61]. Although transactions are useful abstractions to preserve consistency, they are not uniformly integrated with the object-oriented concepts. This is due to the conventional procedure-call like semantics of transaction executions. Consider, for example, the inheritance hierarchy as depicted by figure 5.

In this example, Class *C* inherits from classes *A* and *B*. The operation *operationB1* of class *B* declares an atomic transaction block in its implementation. Assume that class C requires to execute two inherited operations *operationA1* and *operationB1* atomically. The *operationC1* of class *C*, therefore, declares an atomic transaction block, and reuse the operations *operationA1* and *operationB1* by invoking these operations on the pseudo variable *self* within the atomic block.

In an extreme case, assume that class *C* requires to execute all the combinations of operations of classes *A* and *B* atomically. If these combinations are restricted only to a pair of operations, then class *C* has to declare $J \times K$ operations atomically. If the operations declared by class *C* are also to be included in atomic declarations, and combinations are not restricted to a pair of operations, then the required number of atomic operation declarations grows exponentially.

4.2.3. Arbitrary Inheritance Mechanisms

Class inheritance can be seen as excluding, overriding and/or extending the operations and local variables of the superclasses. This kind of inheritance mechanism, however, fails in modeling inheritance hierarchies which require semantics other than overriding or extending operations. To make this more clear, consider a class *Calculator* which defines the operation *compute* that processes the input text stream according to its specifications.

This calculator implements four arithmetic operations (*operators*) and the input text stream can be provided in the usual form, such as "2 + 2 =". The operation *compute* of class *Calculator* checks the syntax of the input stream and computes it if it understands the syntax. Ideally, class *Calculator* can be extended, for example, by defining a subclass *ScientificCalculator* extending the grammar rules for some scientific computations, such as trigonometric functions. The object-oriented model has no adequate means to implement such hierarchies.

One might claim that implementation could be realized by defining a class with basic arithmetic operations and its subclass providing a set of scientific operations. This solution, however, still requires a parser to be further extended in order to parse the input text stream containing additional trigonometric operators, and to invoke the corresponding operations.

A possibility could be to define operations for realizing a parser, and reuse them through the class hierarchy. However, this would force the software engineer to choose implementations early in the analysis phase which is obviously not desirable. This implies that using class inheritance only is not sufficient for dealing with the evolution of software systems incorporating parser modules.

The need for an inheritance mechanism other than class inheritance becomes very apparent when building *application generators*. An application generator accepts a certain *specification*, in our example a grammar specification, and generates executable code in its application domain. When developing such systems, especially in the analysis phase, the software engineer needs to define hierarchies that organize the specifications of the application domain.

4.2.4. Inheritance versus States

Most methods [1-4, 8] consider states as an important aspect of object-oriented software development. States are used to capture the dynamic behavior of systems, and are also used as a means for identifying operations of objects. In general, a state represents the condition of an object at a certain moment. This condition is expressed in terms of the values of local variables. *Events* indicate the transitions from one state to another. In object-oriented systems, events are usually initiated by the reception of messages.

Although these methods consider states as an essential issue in object-oriented modeling, they do not address the integration of states with inheritance. First of all, it is not clear whether the state specification of a class should be considered to be inherited by its subclasses. If this is not the case, it means that state specifications have to be provided again for every new subclass (i.e. lack of reuse). If state specifications are inherited by subclasses, it must be clearly defined how extensions are to be made to the original specifications. Things become more complex when multiple inheritance is considered; inheriting multiple state specifications may cause inconsistencies or conflicts.

Only OMT [8] considers the issues of *generalization and/or specialization* of state specifications as significant. State diagrams are inherited via the class inheritance mechanism. Specialization of state diagrams is partially possible, by replacing a single state of the superclass by a state diagram in the subclass. The method also proposes a generalization hierarchy of events, which should be independent of the class hierarchy. Reuse and extensibility of state diagrams are restricted due to the limited possibilities for extension. Although multiple inheritance is supported by the object-model, the consequences of this are not worked out in cases where states are inherited.

We claim that a notation for the specification of state diagrams should be suitable for extension by subclasses. Although such a mechanism is not provided by most conventional methods, several object-oriented languages provide a mechanism for specifying states. This is primarily useful for purposes of synchronization and concurrency control. Examples are ACT++ [52] and Rosette [67]. However, there may be some situations where a certain extension of a class requires extensive redefinitions, whereas this seems intuitively unnecessary. We demonstrate an example of the so-called *State Partitioning Anomaly* [59] which is exemplified by a class *BoundedBuffer*.

The bounded buffer is a FIFO buffer with a limited capacity for storing data elements. The buffer provides two operations: *put* and *get*. The operation *put* adds an element to the end of the buffer, provided there is space available. The operation *get* retrieves an element from the head of the buffer, if there are any elements available. If a request cannot be executed at the moment of invocation, it is queued. An instance of *BoundedBuffer* can be in one of the following states: *empty*, *full*, and *partial*. The state *partial* means that the buffer contains at least one element but is not entirely filled. Synchronization of the bounded buffer is then described as follows:

empty --> only the *put* operation can be executed; the *get* operation is queued.

partial --> both the *put* and the *get* operations can be

executed.

full --> only the *get* operation can be executed; the *put* message must be queued.

The state partitioning anomaly occurs when the bounded buffer is extended, for example, to a subclass called *BoundedBuffer2* with an operation that returns two elements at a time. Then the state *partial* of the buffer is partitioned into two *sub-states*; one being the state where a single *get* is allowed, but a *double get* is not. In the other sub-state both a *single* and a *double get* are allowed. Clearly, this gets more complicated when an arbitrary number of elements is to be added and/or extracted.

When a state partitioning is required in a subclass, this creates two problems. Firstly, all operations in the superclass that explicitly identify the partitioned state have to be redefined in order to make the distinction between the two *sub-states* that are required in the subclass. Secondly, the introduction of the two *sub-states* eliminates the *super-state*, requiring redefinition of all references to this state.

4.3. Problems Related to Object Interactions

4.3.1. Multiple Views

Not all operations provided by an object are necessarily of interest to other objects that use its services. Therefore, it is desirable to define *views* on an object, differentiating between clients. Consider the following example.

In our example school, as shown in figure 6(a), teachers and students are represented by classes *Teacher* and *Student*, respectively. Classes *TeacherRegistration* and *StudentRegistration* are used for administrative purposes. Now assume that part of the teachers are willing to take courses, and are registered as students. In such a case, a teacher may be viewed either as a student or as a teacher depending on the context that he or she is functioning in. This situation is illustrated in figure 6(b).

The object-oriented methods that are studied in this paper [1-10] cannot express multiple views of objects. In languages such as C++ [47], Trellis/Owl [62] and PAL [42], different views can be defined by the programmer with respect to the different categories of clients of an object. These mechanisms in general only distinguish between the following categories: the object itself, the subclasses of an object, and other client objects. However, they do not allow any distinction between different kinds of external client objects. In the Smalltalk programming environment [49] the concept of *private operations* is introduced, but it is not enforced by the language. Note that multiple views cannot be simulated by introducing a different object for every view. This is because there must be one object, with a single identity, and a single state, which behaves differently according to the way it is being viewed.

4.3.2. Queries and Language-Database Integration

Most object-oriented methods do not address the database issues, such as *persistent data structures*, *transactions*, and *queries* in software development. A few methods [4, 8] address database issues, but only in respect to the definition of persistent objects.

Traditionally, data-intensive applications have been

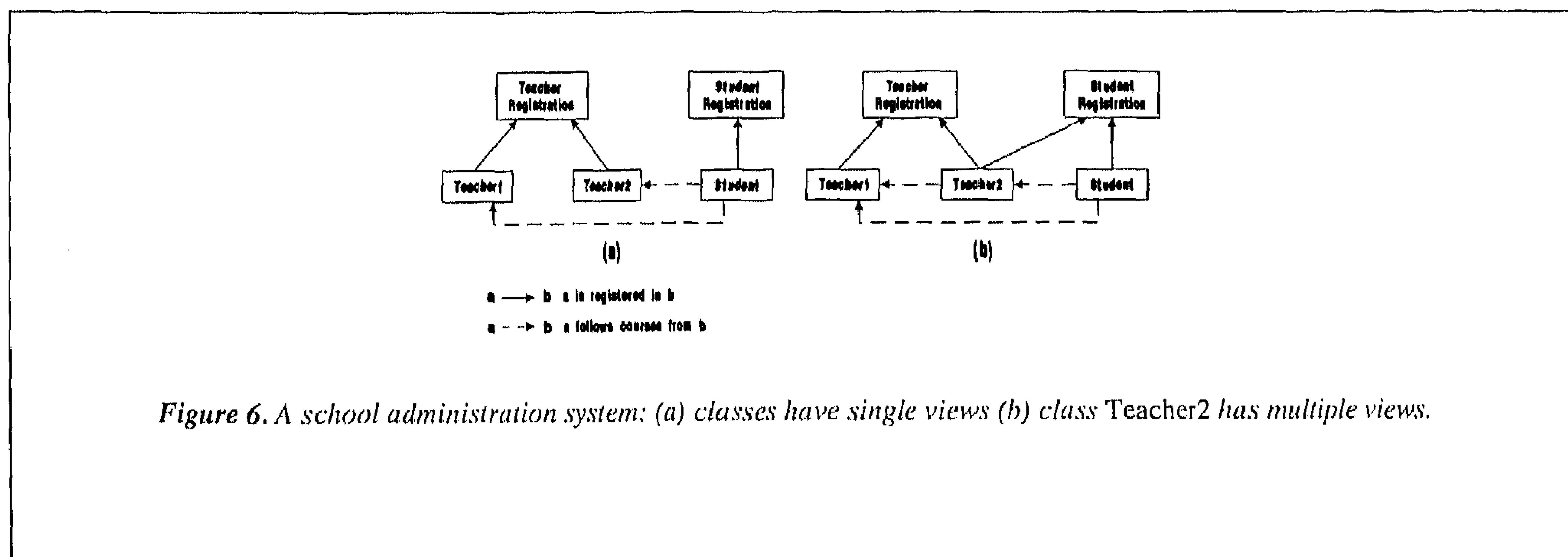

Figure 6. A school administration system: (a) classes have single views (b) class Teacher2 *has multiple views.*

developed as application programs executing on top of a database management system by using database services [44]. This approach suffers from the need to manage two different languages and data structures.

There have been a number of attempts to integrate the two systems within the framework of the object-oriented paradigm [43, 53, 61]. It is claimed that there would be a higher level of integration since the object-oriented model is suitable as a common computation model for both application programming and data management operations.

Current object-oriented database systems support the basic elements of the object-oriented model, and provide efficient data management, transaction support, and querying facilities. However, the complete integration of language and database systems cannot be considered to be accomplished satisfactorily. The problem is many-fold.

Firstly, since these systems extend an object-oriented computation model with conventional database mechanisms, such as (non-object-oriented) query languages, the programmer still has to deal with two different systems. For instance, the usage of a *separate block constructor* in GemStone's OPAL [43], or the necessity of explicit *object lookups* and *puts*, *object-type links*, and the *SQL interface* in Ontos [61], force the programmer to deal with two distinct systems.

Secondly, introducing database-like features into the object-oriented language model generally weakens encapsulation. In Gemstone [43], Orion [53] and Ontos [61], attempts to formulate object queries have resulted in *path expressions* which make object structures visible, contrary to the encapsulation principle of the object-oriented model: encapsulated data should be accessible via message sends only.

Thirdly, for almost all systems, queries are restricted to a fixed number of classes, and thus objects to be accessed associatively have to be inserted into an instance of these classes explicitly. For example, the query capabilities in Smalltalk [49] and Gemstone [43] are restricted to instances of *collection* classes. The problems with Orion's [53] approach are that queries are defined on all instances of a class, thereby produces sets, and the resulting sets cannot be further restricted. In Ontos [61], queries can only be directed to classes and *aggregates*. Similar to Orion, return values are restricted to a few types of classes. In addition, a query may return *rows* that are not objects.

Other problems with respect to data management are related to atomic transactions and multiple views, and have been handled in sections 4.2.2 and 4.3.1, respectively.

4.3.3. Coordinated Behavior

The object interaction model of object-oriented methods [1-10] is mainly based on *message send* semantics, where the sender object transmits a message to a receiver object, and depending on the synchronization semantics, either it waits until the receiver object explicitly returns from performing its task or continues with its processing[4]. We consider the message send model as being too low-level, because it can only specify communications that involve two partner objects at a time and its semantics cannot be easily extended.

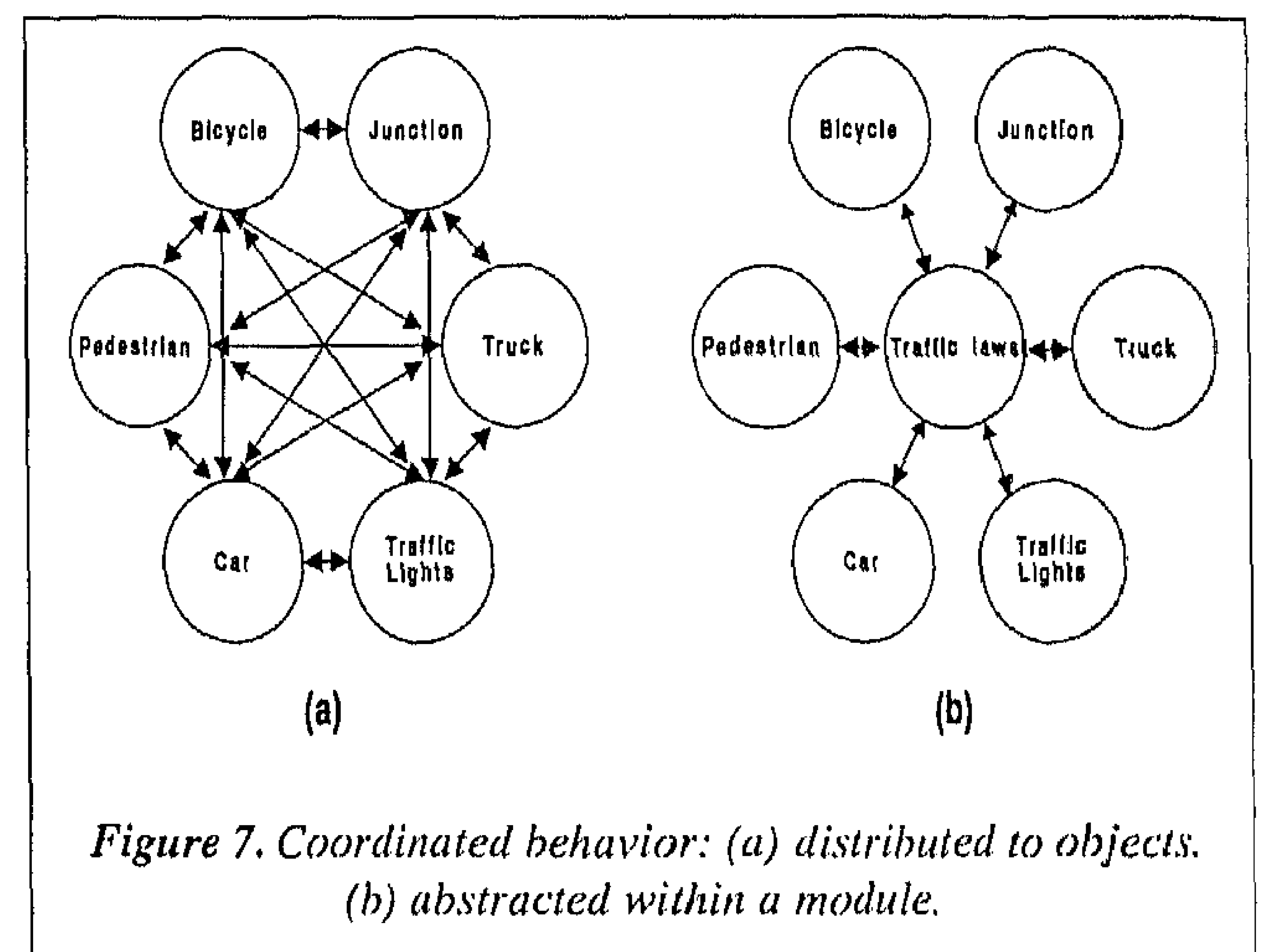

Figure 7. Coordinated behavior: (a) distributed to objects. (b) abstracted within a module.

Mechanisms like inheritance and delegation only support the construction and behavior of objects but not the abstraction of communication among objects. These mechanisms therefore fail in abstracting patterns of messages and larger scale synchronization involving more than just a pair of objects. Consider the interaction pattern of objects trying to cross a

[4] Booch's design notation [1] introduces different kinds of synchronization, such as *simple*, *synchronous*, *balking*, *time-out* and *asynchronous* communication.

junction as illustrated by figure 7(a).

Application of current object-oriented methods would probably result into a specification similar to figure 7(a) which shows a pattern of interactions among objects *Bicycle, Truck, Car, Pedestrian, Junction* and *TrafficLights.*

In the real-world, however, we define traffic laws to specify the rules to participate in traffic flow. Traffic laws simply define the coordinated behavior of objects in traffic. The modularization of traffic rules has two obvious advantages. Firstly, it is easier to enforce traffic rules if there is a module explicitly representing them. Secondly, similar to the object-oriented subclassing principle, modularly specified rules can be extended easily by introducing new rules.

The idea of specifying object interactions is only applied[5] by *contracts* [51]. Contracts define the *contractual obligations* that a set of *participants* must satisfy. It is possible to *refine* a contract in order to make it more specific, and it is possible to *include* existing contracts into a new contract. Although contracts are very useful in verifying the communication between its participating objects, they cannot be fully exploited for abstracting actual communications. Firstly, contracts have their own inheritance hierarchy which is not integrated with the class inheritance hierarchy. Secondly, contracts are only concerned with making sure that the communication between its participants satisfies the *contractual obligations*, but the actual communication activities are distributed over the participants. This makes it impossible to reuse the implementation of the object interactions.

Coordinated behavior can not be modeled by class inheritance. For instance, classes of *Bicycle*, *Car* and *Truck* do not inherit from class *TrafficLaws.* These classes use traffic laws when they need them; they are not laws by themselves. What is required here is the abstraction and realization of the coordinated behavior by a dedicated class *TrafficLaws*, as illustrated by figure 7(b).

5. Evaluation

An overview of which specific problems were encountered in each of our pilot-studies is given in figure 8. We will briefly explain the connections between the identified problems and the pilot studies in which they were encountered.

The problem of identifying classification structures in the problem domain was very obvious while we were analyzing large systems, such as the administrative system, the intelligent tutoring system, the distributed office system, and to a lesser degree the distributed operating system. The classification structures that we could identify were too simple to be usable. Finding hierarchies within the identified subsystem, however, was not as difficult since these sub-components were highly specified and well-understood. Generally, subsystems are responsible for a well-defined function, and therefore can be related to a specific and well-organized knowledge domain.

We observed the problem of creating excessive objects while we were analysing the network database, the chemical process control system and the mechatronic modeling system. These are relatively well-documented and understood problem areas, with a large number of related publications. The books that we referred to were quite voluminous. Referring to theories is useful in defining reusable class hierarchies but may result in an excessive number of objects.

The problem of early decomposition was evident in the administrative system design. When we started with the analysis phase, we could not partition the system into subsystems due to our restricted knowledge of the problem domain. We underestimated the early decomposition problem and started to identify objects within the whole system immediately. The analysis process soon became unmanageable because we had too many objects to deal with.

The problem of subsystem-object distinction was relevant in the distributed operating system design assignment. In operating systems, resources are typically encapsulated within each other like *onion-rings* [45]. In case concurrent resources are encapsulated within each other, it is likely that subsystems will be turned into objects in later steps of the analysis process.

One clear illustration of the problem of commonality versus partitioning was the analysis of the temperature control system. In this example, subsystems included

5 Apart from the object-oriented language Sina [16].

different kinds of sensors which could be organized in a common hierarchy. The distribution of sensors to subsystems, however, made the identification of class hierarchies more difficult, even though the system was rather small.

The problem of subsystem identification using object interactions was encountered in the development of the distributed language implementation and the distributed system design. In these applications, we had to deal with bulky data transfers through the network. Without using automatic mechanisms [65, 66], however, it was almost impossible to identify subsystems using object interaction patterns.

The problem of sharing behavior with state manifested itself in at least three applications: the administrative system, the intelligent tutoring system and the distributed office system. In the administrative system, some information about persons had to be reused in different calculations. The current state of the persons affected the execution of these calculations. In the intelligent tutoring system, *knowledge sources* shared common behavior affected by the current state of the *decision making* process. In the distributed office assignment, office activities shared by different employees were affected by the current office strategies.

The problem of atomicity and inheritance demonstrated itself in the intelligent tutoring system and in the distributed office system. In the first pilot study, distributed information had to be processed atomically by different *knowledge sources*. In the latter assignment, the distributed calendar management system and the financial control system required atomic transactions.

During analysis of the chemical process control system, the mechatronic modeling system, the concurrent processing and synchronization examples, and the parser generator for Smalltalk, we needed to define a different inheritance mechanism other than conven-

Pilot studies \ Problems	Problem domain structures	Excessive domain objects	Early decomposition	Subsystem-object distinction	Commonality vs partitioning	Subsystem identification	Sharing behavior with state	Atomicity and Inheritance	Arbitrary inheritance	Inheritance vs states	Multiple views	Language-database	Coordinated behavior
Administrative system	■		■				■				■	■	■
Network database		■										■	
Chemical process control system		■							■				■
Mechatronic modeling system		■							■		■		■
Intelligent tutoring system	■						■	■			■	■	■
Concurrent processing/synchron.									■	■			
Distributed office system	■						■	■			■	■	■
Distributed language implemen.						■							
Parser generator for Smalltalk									■				
Distributed op. system design	■			■		■							■
Temperature control system					■								
Intelligent mail system											■		

Figure 8. Pilot studies versus problems.

tional class inheritance. Typically all these applications were generated by using abstract specifications, and for each specification, a dedicated inheritance mechanism had to be defined.

The problem of inheritance versus states was very obvious in the concurrent processing and synchronization examples.

The multiple views problem was clearly observed in the administrative system, the mechatronic modeling system, the intelligent tutoring system, the distributed office system, and in the intelligent mail system. Multiple views were needed in these applications to properly structure object interactions.

We were confronted with the problem of language-database integration in the administrative system, the intelligent tutoring system, the distributed office system, and in the intelligent mail system. All these applications required some sort of query facilities and persistent objects. We did not face this problem in the network database assignment, because queries were very simple and objects were not persistent.

For many applications, we could benefit from mechanisms that could abstract communication details. In the administrative system, different components had to coordinate for calculations. In the chemical process control system and the mechatronic modeling system, algorithms were distributed to different components, but could be abstracted into modules. In the intelligent tutoring system, the decision making process was based on the coordination of different *knowledge sources*. Similarly, the distributed office system and the distributed operating system designs required communication abstractions, for example in building layered architectures, dedicated distributed concurrency-control mechanisms and implementing security protocols.

6. Our Research Activities

We initiated a number of research activities founded on our experiences in object-oriented design. This section briefly introduce our research activities.

We believe that an object-oriented model that provides abstract operations for its users and encapsulates its implementation details is a good starting point for building large systems. Polymorphic message passing between objects and sharing mechanisms, such as inheritance, are important techniques in building reusable and extensible software. However, we feel that the conventional object-oriented model is not powerful enough to deal with the problems that are presented in this paper.

In order to address the identified problems, we have introduced a new concept, called *composition filters*. The basic object model is extended *modularly* by introducing *input* and *output composition filters* that affect the received and sent messages, respectively. Composition filters support the following features:

Both inheritance and delegation are supported without introducing special language constructs such as class inheritance and/or delegation[6] [37]. As a result, one can implement shared object states embedded in the behavior of objects through delegations. This technique solves the problem of sharing behavior with state as presented in section 4.2.1.

Composition filters are able to express a mechanism called *atomic delegation* [39]. Atomic delegation allows an object to delegate the requests of its users to one or more objects atomically; atomic delegations are *serializable* and *indivisible*. Atomic delegation resolves the problem of atomicity and inheritance which was defined in section 4.2.2.

The inheritance versus states problem, as presented in section 4.2.4, can be solved by controlling object's interfaces through the application of composition filters [41].

The composition-filter mechanism can define multiple views on an object by differentiating between clients [40]. The multiple views problem was identified in section 4.3.1.

Composition filters can express associativity on inheritance and delegation hierarchies; this is called *associative inheritance/delegation* [40]. This feature is useful in managing complex inheritance/delegation hierarchies. In addition, data management operations such as *select* and *union* can be defined through input composition filters for any object [40]. This feature eliminates the restriction of the current object-oriented databases which only allow database operations on

[6] In [37], composition filters were called interface predicates.

class hierarchies and/or a special set of classes such as *collections*. The problem of language-database integration was described in section 4.3.2.

Output composition filters are used to support the so-called *abstract communication types* that abstract patterns of communication and large scale synchronization among objects. We are currently carrying out a research activity based on our earlier publication [16] to address the problem of coordinated behavior, as presented in section 4.3.3.

We are defining suitable software development methods to fully utilize and support the composition-filters based object model.

The concept of composition-filters is adopted by the Sina language. Various versions of the Sina language have been implemented. The early version of the language allowed only a single input filter with a fixed structure [37, 39]. In addition, some publications [68, 69] have illustrated the flexible concurrency control and synchronization mechanisms of the early version of Sina. The extended version of the language is published in our recent papers [40, 41].

To minimize expensive network traffic, we have introduced the concept of *inverse remote procedure calls* (IRPCS) [65, 66]. The IRPC system partitions a given program by using a new heuristic that derives a (sub-)optimal object allocation to the network sites. Originally, the IRPC system was intended to be a part of an operating system, thus providing transparent code partitioning. Currently, we are extending this work to support software development activities. We define algorithms to partition systems hierarchically using both the functional and object-interaction semantics of applications. The extended IRPC algorithm attempts to resolve the problem of subsystem identification using object interactions, as presented in section 4.1.2.

The problem of arbitrary inheritance hierarchies, as presented in section 4.2.3, is addressed partially by the *grammar inheritance* tool [38]. Grammar inheritance is a structural organization of grammar rules by which a grammar inherits rules from *super-grammars* or may have its own rules be inherited by *sub-grammars*.

7. Conclusions

We briefly introduced the basics of the object-oriented paradigm and the state-of-the-art software development methods. Then we presented the pilot studies that we were involved with. Based on the experiences from these pilot studies, we identified a number of problems, in three categories: preparation, structural relations and object interactions, which are explained in section 4. We then evaluated our pilot studies with respect to these problems.

The following origins of these problems can be recognized. Firstly, some problems, such as of problem domain structures, excessive domain objects, early decomposition, commonality versus partitioning and subsystem identification are inherent to the object-oriented software development methods. They arise due to the size and complexity of the problem domain, and the way in which it is modeled by object-oriented methods.

Secondly, some problems are due to the process of object-oriented development. For example in the commonality versus partitioning problem, one of the two object-organization hierarchies (i.e. classification and part-of hierarchies) prevails over the other. Whichever choice is made, the primary organization hierarchy will hinder proper identification of the second organization hierarchy. Similarly, early decomposition impedes proper structure identification, and late decomposition results in too many objects to deal with.

Thirdly, the expressive power of the object-oriented computation model is too restricted to deal with the problems that involve structural relations and object interactions.

In spite of the encountered problems we are optimistic about the application of object-oriented development methods. This is both due to our current experiences with the pilot applications, and to our expectation that most of the identified problems can be solved, at least partially.

Acknowledgements

We would like to thank Charles Grossman and Enis Yücesoy for their efforts in improving the earlier version of this paper.

References

State-of-the-art Object-Oriented Methods:

[1] G. Booch, *Object-Oriented Design with Applications*, The Benjamin/Cummings Publishing Company, Inc, 1991.

[2] D. de Champeaux, *Object-Oriented Analysis and Top-Down Software Development*, European Conference on Object-Oriented Programming, pp. 360-375, July 1991.

[3] P. Coad & E. Yourdon, *Object-Oriented Analysis*, 2nd edition, Yourdon Press Computing Series, Prentice-Hall, 1991.

[4] P. Coad & E. Yourdon, *Object-Oriented Design*, Yourdon Press Computing Series, Prentice-Hall, 1991.

[5] R. Johnson & B. Foote, *Designing Reusable Classes*, Journal of Object-Oriented Programming, pp. 23-35, June/July 1988.

[6] K. Lieberherr & I. Holland, *Assuring Good Style for Object-Oriented Programs*, IEEE Software, pp. 38-48, September 1989.

[7] K. Lieberherr *et al.*, *Graph-Based Software Engineering: Concise Specifications of Cooperative Behavior*, Northeastern University, Tech. Report: NU-CCS-91-14, September 1991.

[8] J. Rumbaugh *et al.*, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[9] R. Wirfs-Brock *et al.*, *Responsibility-Driven Design*, Prentice-Hall, 1990.

[10] R. Wirfs-Brock and R. Johnson, *Surveying Current Research in Object-Oriented Design*, Communications of the ACM, Vol. 33, No. 9, pp.104-124, September 1990.

Other methods:

[11] G. Booch, *Software Engineering with Ada*, Benjamin-Cummings, 1983.

[12] G. Booch, *Object-Oriented Development*, IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, pp. 211-221, February 1986.

[13] S. Shlaer and S. Mellor, *Object-Oriented Systems Analysis, Modeling the World in Data*, Yourdon Press, 1988.

[14] E. Yourdon, *Modern Structural Analysis*, Prentice-Hall, 1989.

Pilot Studies:

[15] M. Aksit, *Concurrent Processing and Synchronization*, in *On the Design of the Object-Oriented Language Sina*, Ph.D. Dissertation, Chapter 3, Department of Computer Science, University of Twente, The Netherlands, 1989.

[16] M. Aksit, *Abstract Communication Types*, in *On the Design of the Object-Oriented Language Sina*, Ph.D. Dissertation, Chapter 4, Department of Computer Science, University of Twente, The Netherlands, 1989

[17] M. Aksit, *Atomic Delegations*, in *On the Design of the Object-Oriented Language Sina*, Ph.D. Dissertation, Chapter 5, Department of Computer Science, University of Twente, The Netherlands, 1989

[18] J. Bank, *Concurrent Programming and Synchronization in Sina/st*, University of Twente, Department of Computer Science, M.Sc. Thesis, The Netherlands, June 1988

[19] M. v.d. Bempt, *Construction of Hierarchies in Distributed Computer Systems*, M.Sc. Thesis, Department of Computer Science, University of Twente, The Netherlands, November 1991

[20] L.M.J. Bergmans, *The Sina Distribution Model*, M.Sc. Thesis, Department of Computer Science, University of Twente, The Netherlands, March 1990.

[21] A. Breunese, *Design and Implementation of a Mechatronic Modeling Environment Using Object-Oriented Principles*, M.Sc. Thesis Description, Department of Electrical Engineering, University of Twente, The Netherlands, 1992.

[22] N.M. van Diepen, H. Grünefeld & W.A. Vervoort (eds.), *Ontwerpen van een Netwerkdatabase in vier talen*, Memoranda Informatica 91-22, University of Twente, March 1991.

[23] J.W. Dijkstra, *Atomic Delegations*, M.Sc. Thesis, Department of Computer Science, University of Twente, The Netherlands, December 1988.

[24] H. Dolfing, *An Object Allocation Strategy for Sina*, M.Sc. Thesis, Department of Computer Science, University of Twente, The Netherlands, November 1990.

[25] N. de Greef, *Object-Oriented System Development*, M.Sc. Thesis, Department of Computer Science, University of Twente, The Netherlands, 1991.

[26] E. Jonge, *Object-georienteerde Analyse, Ontwerp en Implementatie van een Batchdestillatiebesturing*, M.Sc. Thesis, Department of Chemical Engineering, University of Twente, The Netherlands, January 1992.

[27] K.A. Lesterhuis, *An Object-Oriented User-Interface Model*, M.Sc. Thesis, Department of Computer Science, University of Twente, The Netherlands, August 1991.

[28] R. Mostert, *SmallYacc, an Object-Oriented Compiler Generator Introducing Grammar Inheritance*, M.Sc. Thesis, Department of Computer Science, University of Twente, The Netherlands, December 1989.

[29] R. Nijhuis, *Sina/st, The Language and its Compiler*, M.Sc. Thesis, Department of Computer Science, University of Twente, The Netherlands, August 1988.

[30] M. Offreins, *Requirements for Building Intelligent Tutoring Systems*, Memo, Department of Computer Science, University of Twente, The Netherlands, 1992.

[31] W. Veldkamp, *The Sina/st User Interface*, M.Sc. Thesis, Department of Computer Science, University of Twente, The Netherlands, December 1988.

[32] J. de Visser, *Object-Oriented Analysis of a Temperature Control System*, Memo, Department of Computer Science, University of Twente, The Netherlands, 1991.

[33] S. Vural, *Object-Oriented Development of an Intelligent Mail System*, Memo, Department of Computer Science, University of Twente, The Netherlands, 1991.

[34] G. Wageningen, *Abstract Communication Types*, M.Sc. Thesis, Department of Computer Science, University of Twente, The Netherlands, 1989.

[35] E. G. Zondag, *Hierarchical Management of Distributed Objects*, Memoranda Informatica 90-73, 1990.

Other references:

[36] Ada Joint Program Office, *Reference Manual for the Ada Programming Language*, February 1983.

[37] M. Aksit & A. Tripathi, *Data Abstraction Mechanisms in Sina/ST*, OOPSLA '88, pp. 265-275, 1988.

[38] M. Aksit, R. Mostert & B. Haverkort, *Compiler Generation Based on Grammar Inheritance*, Memoranda Informatica 90-07, February 1990.

[39] M. Aksit, J.W. Dijkstra & A. Tripathi, *Atomic Delegation: Object-Oriented Transactions*, IEEE Software, Vol. 8, No. 2, March 1991.

[40] M. Aksit, L. Bergmans & S. Vural, *An Object-oriented Language-Database Integration Model: - The Composition Filters Approach*, ECOOP '92.

[41] L. Bergmans, M. Aksit, K. Wakita & A. Yonezawa, *An Object-Oriented Model for Extensible Synchronization and Concurrency Control*, working paper, University of Twente, June 1992.

[42] A. Bjornerstedt & S. Britts, *AVANCE: An Object Management System*, OOPSLA '88, pp. 206-221, 1988.

[43] R. Bretl *et al.*, *The GemStone Data Management System, Object-Oriented Concepts, Databases, and Applications*, Ch. 11, eds. W. Kim and F. H. Lochovsky, pp. 283-309, Addison-Wesley, 1989.

[44] C. J. Date, *An Introduction to Database Systems*, Vol 1., Addison-Wesley Publishing Company, 1990.

[45] E.W. Dijkstra, *The Structure of the T.H.E. Multiprogramming System*, Communications of the ACM, No. 11, pp. 341-346, 1968.

[46] J.G. Donnett, M. Starkey & D.B. Skillicorn, *Effective Algorithms for Partitioning Distributed Programs*, Proceedings of the 7th Annual International Phoenix Conference on Computers and Communications, pp. 363-368, March 1988.

[47] M. Ellis & B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[48] K.P Eswaran, J.N. Gray, R.A. Lorie & I.L. Traiger, *The Notions of Consistency and Predicate Locks in a Database System*, Communications of the ACM, Vol. 19, No. 11, November 1976.

[49] A. Goldberg & D. Robson, *Smalltalk-80 - The Language*, Addison-Wesley, 1989.

[50] Haerder & A. Reuter, *Principles of Transaction-Oriented Database Recovery*, ACM Computing Surveys, Vol. 15, No. 4, pp. 287-317, December 1983.

[51] R. Helm, I. Holland & D. Ganghopadhyay, *Contracts: Specifying Behavioral Compositions in Object-Oriented Systems*, OOPSLA '90, pp. 169-180, 1990.

[52] D.G. Kafura & K.H. Lee, *Inheritance in Actor Based Concurrent Object-Oriented Languages*, ECOOP '89, pp. 131-145, 1989.

[53] W. Kim et al., *Features of the ORION Object-Oriented Database System*, in *Object-Oriented Concepts, Databases, and Applications*, Ch. 11, eds. W. Kim and F. H. Lochovsky, pp. 251-282, Addison-Wesley, 1989.

[54] D.E. Knuth, *The Art of Computer Programming*, Vol. 1/Fundamental Algorithms, Addison-Wesley, 1973.

[55] J. A. Lewis *et al.*, *An Empirical Study of the Object-Oriented Paradigm and Software Reuse*, OOPSLA '91, pp. 184-196, October 1991.

[56] H. Lieberman, *Using Prototypical Objects to Implement Shared Behavior*, OOPSLA '86, pp. 214-223, 1986.

[57] B. Liskov et.al., *Argus Reference Manual*, MIT Lab. for Computer Science, No. MIT-TR-400, November 1987.

[58] D.B. Lomet, *Process Structuring, Synchronization and Recovery using Atomic Actions*, ACM SIGPLAN, Vol 12, No. 3, pp. 128-137, March 1977.

[59] S. Matsuoka, K. Wakita & A. Yonezawa, *Inheritance Anomaly in Object-Oriented Concurrent Languages*, Un. of Tokyo, April 1991.

[60] *Object Oriented Design*, Course Notes, University of Twente, June 1992.

[61] *Ontos Object Database version 2.0 Developer's Guide*, Ontologic Inc., Burlington (Mass.), February 1991.

[62] C. Schaffert, T. Cooper, B. Bullis, M. Kilian & C. Wilpolt, *An Introduction to Trellis/Owl*, OOPSLA '86, pp. 9-16, 1986.

[63] R. Skvarcius & W.B. Robinson, *Discrete Mathematics with Computer Science Applications*, Benjamin/Cummings Publishing Company Inc., 1986.

[64] Z. Spector, D. Daniels, D. Duchamp, J.L. Eppinger & R. Pausch, *Distributed Transactions for Reliable Systems*, ACM SOSP Conference, 1985.

[65] A.D. Stoyenko, M. Aksit & J. Bosch, *Inverse Remote Procedure Calls*, Memoranda Informatica 91-89, 1991.

[66] A.D. Stoyenko, M. Aksit & J. Bosch, *A new Heuristic for Load-Balanced Assignment of Objects and Minimized Netweork Communication in Distributed Programs Implemented through Inverse Remote Procedure Calls*, Memoranda Informatica 91-91, 1991.

[67] C. Tomlinson & V. Singh, *Inheritance and Synchronization with Enabled-Sets*, OOPSLA '89, pp. 103-112, 1989.

[68] A. Tripathi & M. Aksit, *Communication, Scheduling and Resource Management in Sina*, Journal of Object-Oriented Programming, pp 24-41, November/December 1988.

[69] A. Tripathi, E. Berge & M. Aksit, *An Implementation of the Object-Oriented Concurrent Programming Language Sina*, Software Practice and Experience, pp 235-256, March 1989.

[70] B. Walker et al., The Locus Distributed Operating System, 9th ACM Symposium on Operating System Principles, pp. 49-70, October 1983.