in: Data & Knowledge Engineering 9 (1992/93), pp. 321-352, North Holland

Grand Tour of Concepts for Object-Orientation from a Database Point of View

Nelson M. Mattos[1], Klaus Meyer-Wegener[2], and Bernhard Mitschang

University of Kaiserslautern
Department of Computer Science
P.O. Box 30 49
6750 Kaiserslautern
Federal Republic of Germany
e-mail: mitsch@informatik.uni-kl.de

Keywords: Object-Orientation, Semantic Modeling, Data Model, Database

Abstract

Over the last few years, object-orientation has gained more and more importance within several disciplines of computer science (e.g., programming languages, knowledge engineering, and database systems). Numerous papers have defined one or another of its underlying concepts (sometimes in quite different ways), and some systems have been developed following those heterogeneous definitions. Nevertheless, papers investigating the dependencies and degrees of freedom of these concepts are rarely found. For this reason, the goal of this paper is not to add yet another definition of object-oriented concepts, but to identify existing relationships among these basic concepts that allow one to cover and classify various conceivable combinations of these conceptual building blocks. Dependencies, orthogonalities, and relations among concepts like object identity, encapsulation, classification, generalization, inheritance, etc. are revealed, showing numerous ways to compose different shades of object-orientation. This leads to alternatives encountered when constructing object-oriented systems, which are illustrated by classifying some well-known systems and prototypes from different areas. However, it is not our purpose to analyze the relative importance of these concepts. Instead, we investigate the concepts from a neutral point of view, presenting (but not evaluating) several degrees of object-orientation.

1. Introduction

Since the beginning of computer science, there has been a continuing effort to cope with the ever-increasing complexity of software. A software system constitutes a model of the real world and contains

1. IBM Database Technology Institute, Santa Teresa Laboratory, 555 Bailey Ave., San Jose, CA, 95150 USA, e-mail: mattos@stlvm14.vnet.ibm.com
2. University of Erlangen-Nuremberg, IMMD VI, Martensstrasse 3, 8520 Erlangen, e-mail: Klaus.Meyer-Wegener@informatik.uni-erlangen.de

specific information needed in an application [1, 7]. Hence, the process of software development can be viewed as the process of building an accurate model of the real-world situation at hand [10]. A lot of research activities, mainly in the fields of programming languages, software engineering, artificial intelligence, and database management, have been focussing on the development of models that naturally and directly reflect the user's conception of the real world in discourse [38]. Such a model will ease the task of the modeler as well as the user-system interaction, accelerating the process of software development as a whole.

One of the many research directions that have emerged in the last years with the purpose of achieving this goal circles around the concept of *object-orientation*. In fact, an opalescent variety of systems that claim to be 'object-oriented' have been developed. Today, it is commonly agreed to justify a system as being object-oriented just by saying that it has encapsulation, inheritance, object-identity, and so on. But what does it mean? Is this 'definition' precise enough? For example, inheritance in POSTGRES [50] is not the same as in Smalltalk [24] and both differ from KEE [23,26]! The reason for these different notions is that the concept of inheritance is not self-contained; rather it depends on the notions of classification and generalization. Since Smalltalk only allows tree-like generalization hierarchies, there is no need to cope with multiple inheritance. With POSTGRES it is just the opposite: network-like generalization hierarchies raise multiple inheritance. Both systems provide only for strict inheritance of attributes instead of default inheritance as addressed in KEE. Thus, there are several dependencies among the concepts (e.g. generalization and inheritance) as well as several degrees of freedom within such concepts (e.g. strict versus default inheritance) that are responsible for various shades of a concrete concept as well as for different system's overall functionality and behavior. For this reason, when talking about the concepts required to accept a system as being object-oriented, it is not sufficient to claim for a group of system characteristics (as discussed by [5,52]). It is above all necessary to investigate the dependencies and degrees of freedom of these concepts in order to be able to provide a sound basis for a definition of not one but several degrees or notions of object-orientation.

The claim of this paper is to move one step further towards this goal by identifying existing relationships among these basic concepts. Thus, existing dependencies and orthogonalities will emerge, showing conceivable combinations of these conceptual building blocks that compose different shades of object-orientation. However, we shall not determine a precise borderline of object-oriented systems, since we believe such a borderline can be quite different from application area to application area. Instead, we show different degrees of object-orientation from a neutral point of view. Thus, we neglect a discussion about the specific importance of or the comparison between object-oriented concepts. This should be carried out by the reader with the characteristics and requirements of a specific application in mind.

Although the literature on object-oriented concepts is quite rich, the idea to identify existing relationships among object-oriented concepts has not been the primary issue in the existing publications. Some approaches (e.g. [5,52,44]) explain and list the concepts side by side, whereas other ones [53] broadly examine the origins and paradigms of object-oriented programming and explore some language design alternatives. In some sense, the focus of [53], for example, can be compared to our approach, however with the main difference that its interest is in the field of programming languages, whereas we primarily concentrate on the areas of software engineering, artificial intelligence, and database management.

In this paper, we organize a grand tour of object-oriented concepts in the form of a road map that covers several 'regions' resembling the conceptual building blocks investigated. The 'crossroads' within each region form a classification tree that visualizes alternatives and dependencies among the concepts. Moving along the roads, one finally reaches some 'stops' representing combinations of concepts that can be found in existing systems. The feasibility of this approach is demonstrated by classifying existing systems and proposals from different computer science disciplines all along the way.

In the first region, we define the basic concepts we have to deal with: *objects* and their different kinds of *properties*. In addition to the usual *descriptive properties* (i.e., attributes), we also have to deal with operational and organizational properties to cover the modeling concepts underlying object-orientation. *Operational properties* are also called methods. They are in detail discussed in chapter 3 as we move to another region. Continuing our trip, we reach the subsequent region, i.e., chapter 4, that contains the *organizational properties*. Here, we provide a fine-grained view to the well-known abstraction concepts of classification, generalization, association, and aggregation. In chapter 5, we give a conclusion and a discussion of some intermediate stops not visited on this trip.

Throughout the paper, we assume that the reader is familiar with the concepts normally addressed in object-oriented systems. He/she should have some basic understanding of e.g., issues like object-identity, encapsulation, overloading, late binding, classes, and inheritance as discussed for example in [6,44].

2. Objects and their Properties

From the user's point of view, a universe of discourse is comprehended in terms of *entities* [17]. To constitute an entity, something must be

- identifiable (have identity),
- relevant (of interest),
- describable, i.e., have characteristics.

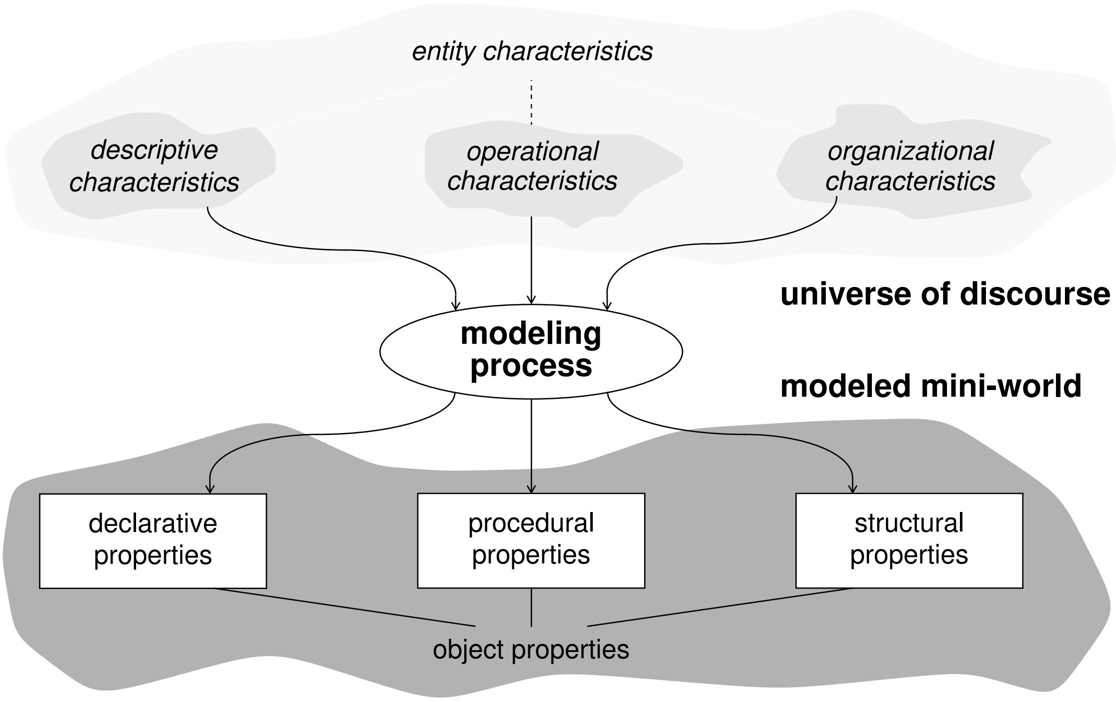


Figure 1: State-oriented view of modeling

The prime characteristics of entities refer to their state (or value). As part of their characteristics, entities may also have some meaningful *relationships* with other entities for organizational purposes. Furthermore, *activities* (i.e., operations on entities) occur over time, resulting in changes to the entities' states and to their interrelationships. Both the entity states and the activities are subject to *constraints* which more precisely define the characteristics, thereby distinguishing 'reality' from other possible worlds [15]. Assume for example the constraint 'always increasing' defined on the characteristic 'salary' of employee entities that expresses that an employee's salary does never decrease. Another example is the constraint 'employees older than 60 years or working for more than 25 years for the company will not be fired'; instead the only activity allowed is 'retiring'.

This state-oriented view of the world is depicted in Figure 1 that shows the conceptual primitives, i.e., entities, having descriptive, operational, and organizational characteristics as mentioned above. By means of the modeling process, each entity relevant to the current universe of discourse is represented by a corresponding *object* in the model. Thus, when talking in terms of the mini-world model, we use the notion of object being the symbolic representation of a real world entity. Of course, each object in the model should have *properties* expressing the characteristics of the corresponding entities in the universe of discourse. For this reason, it is necessary to distinguish three kinds of object properties [40]:

- *Declarative* properties are used for descriptive purposes. Sometimes they are called slots, variables, or simply attributes.

- *Procedural* properties are applied for operational purposes. They reflect activities of the real world and are attached to objects to manipulate or access the object's attributes. Here, we name them methods.
- *Structural* properties are employed for organizational purposes. They represent relationships among entities, e.g., abstractions like classification, generalization, association, and aggregation.

Concerning the modeling process, it is evident that a one-to-one correspondence between the entities in the universe of discourse and the objects of the model should exist. This prevents, for example, that information about a single entity (e.g., a person) will be spread over several objects in the model, resulting in a number of obstacles concerning integrated and efficient management.

A detailed discussion of all procedural and structural properties will be found in the next and its subsequent region (i.e., chapters 3 and 4), whereas the declarative properties are within the scope of the region covered in this chapter.

2.1 Object Specification

After these more general remarks, we are now ready to start our grand tour on object-oriented concepts. The first region to be visited presents the essential object specifications and is outlined by the road map shown in Figure 2. We use a tree-like classification scheme to express the building rules of a specific concept. Normally, it is possible to combine several different criteria in various ways to reach different shades (for possible destinations) of the concept at hand. Each criterion is assigned to a level of the tree listed to its left side, whereas the nodes (or crossroads) represent the different choices (called features) belonging to the criterion.

All conceivable combinations of two criteria on subsequent levels of the tree are represented by edges (corresponding to possible paths) connecting the appropriate choices. There are two combination rules: *alternative* paths (i.e., bifurcations in the road) are drawn as normal edges, whereas *dependencies* are drawn as arrows to a tinted criterion node representing the dependent choice (actually, no choice).

In the context of our road map, the criteria can be seen as intermediate crossroads on the way to a final destination, i.e., a specific system layout. Thus, the tree defines different ways to that stage, i.e., the various shades of the current concept. This is illustrated by naming some well-known systems that can be found at the destination points.

Please note that we have no intension to classify all systems like that, instead we simply want to illustrate what certain shades of concepts mean. That implies that we will name a varying number of sample systems from tree to tree. Sometimes it is not possible to classify a system uniquely because it has different versions or because there is a lack of detailed information needed to reach at a clear decision.

Furthermore, it is not always necessary, and sometimes very hard, to find a sample system for each classification path. If it is obvious that a specific path is not meaningful, we have labeled the corresponding destination with a question mark.

Referring to Figure 2, we can combine four quite important criteria to define various shades of basic object specifications. First of all, we have to decide where to go considering the criterion 'declarative properties'. For completeness, we distinguish the two alternatives shown; although the case that objects cannot have attributes seems to be somewhat strange; we are not aware of any system implementing this choice. Therefore, we have labeled this destination with a question mark.

The other choice moves us to the conceivable crossroad where objects possess some attributes. At this crossroad of our road map, the alternative is whether an object has an identifier or not, i.e., whether *object identity* is supported or not [28]; with this, we mean that the object identifier is valid as long as the object exists and that it is neither updatable nor reusable. Apart from the CODASYL approach (and its database key concept; [45]) conventional data models [18] and straightforward extensions (e.g., NF^2 [19,47]) do not adhere to the latter category.

Moving further on, we have to deal with the criterion 'procedural properties' that introduces the alternative of having *methods* attached to objects or not. It is important to say that by methods we mean user-

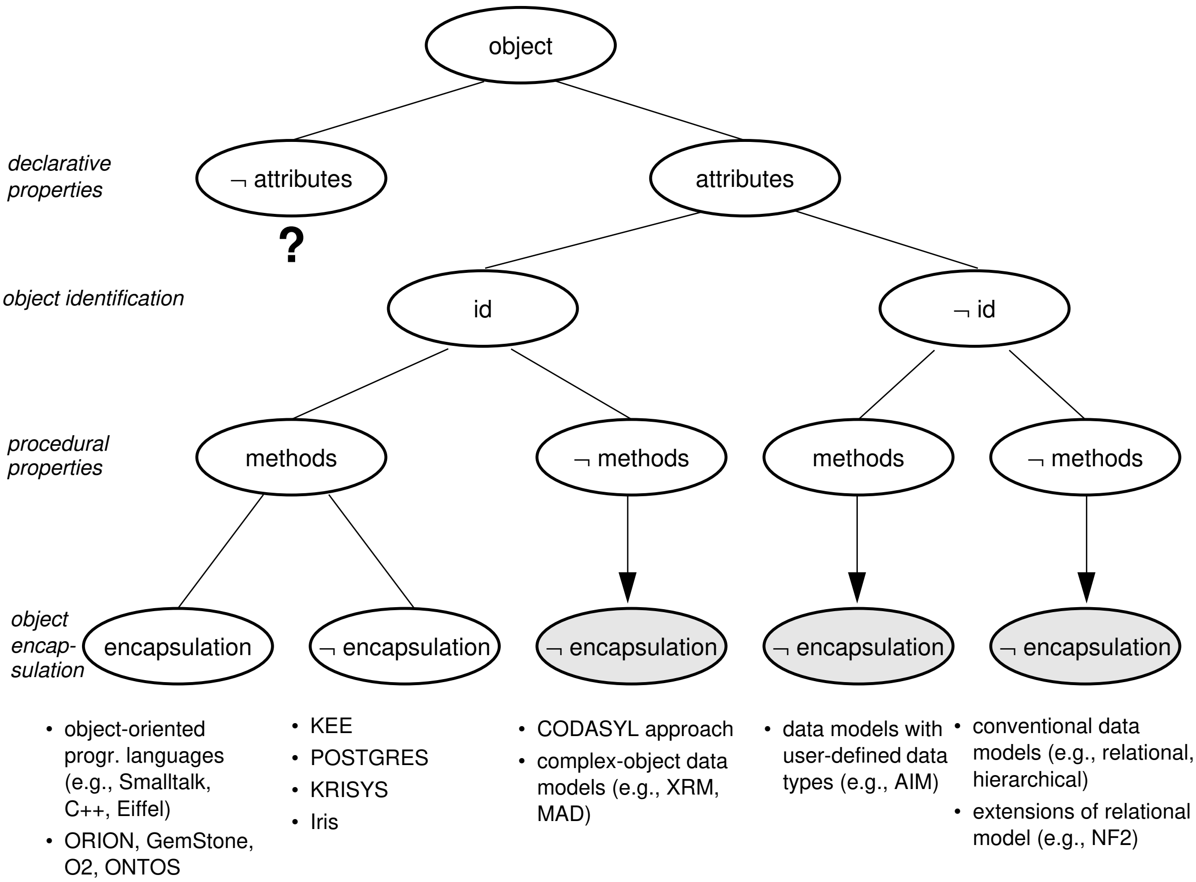


Figure 2: Concept terrain for objects

defined procedures (i.e., programs but not generic operations of a query language (cf. section 4.5)) that provide access to the attributes of an object, i.e., to all information within an object. This is the case if, for example, attribute domains can be abstract data types. If methods are not permitted, there is only one way to move further, i.e., to a destination where the criteria of object encapsulation cannot be supported: *encapsulation* is not possible because without methods, attributes must be accessed directly. (Note that encapsulation is guaranteed iff access to an object's attribute is done only via methods applicable to that object). Also, if the object concept does support methods, but not object identification, it is not possible to implement encapsulation since there would be no way to address an object. Again, we have drawn the corresponding dependency edge as a path in the road map. An example of this category is AIM [34], a data model supporting user-defined data types.

Instead of supporting methods, all (conventional) data models are equipped with powerful query and manipulation languages. This is also true for some data models with object identification, e.g., the well-known CODASYL approach and the emerging non-standard data models that support a complex-object concept (XRM [35], MAD [42, 43] etc.).

In the literature, most systems that provide attributes, object identification, and methods are basically termed object-oriented. The encapsulation criterion separates mostly the so-called object-oriented programming languages/environments (e.g., Smalltalk, C++ [51], Eiffel [41]) and some object-oriented databases (OODBS; e.g., ORION [29], GemStone [37], Iris [22,36], O2 [20], ONTOS[46]) from the so-called expert system tools (e.g., KEE) and some knowledge base management systems (KBMS; e.g., KRISYS [39]).

Nevertheless, there can be several degrees of encapsulation leading to 'hybrid' encapsulation approaches or encapsulation on demand. In the O2 system, attributes are private (encapsulated) by default, but can be made readable and/or updatable. Methods are public by default, but can be made private, which means that they can only be executed by other methods of the same object. Hence, encapsulation concerns both attributes and methods, but in a different way. Also, encapsulation can be total or partial. In the case of methods, systems either have only public methods or allow to declare some of them private; total encapsulation does not make sense. Attributes are either all hidden or can be made visible selectively; the case of all attributes being accessible is considered to be "no encapsulation" and has thus been handled in the tree of Figure 2 already. We do not distinguish whether attributes are hidden by default and can be made visible (through an "export" or "public" declaration, as in Eiffel, ONTOS, and C++, respectively), or whether they are accessible by default and can be hidden (AIM [34]); the result will be the same. If we have visible attributes, we can further distinguish the types of access allowed: read only or update. While a C++ public attribute cannot be write-protected, Eiffel exported attributes cannot be made available for change. Only O2 offers both alternatives.

2.2 Attributes: declarative properties

After having worked out the various routes along the intermediate crossings defined by the above mentioned criteria in the first region, i.e., a course-grained specification of conceivable object concepts, it is now worthwhile to draw attention to the three object properties more distinctively.

Figure 3. depicts the road map for attribute specification being a particularization of the feature 'attributes' of Figure 2. For explanatory purposes, we have decided to use separate classification trees to represent these road maps instead of attaching them as subtrees to one single tree. This would have resulted in an extremely blown up and badly arranged tree where it is almost impossible to explain anything.

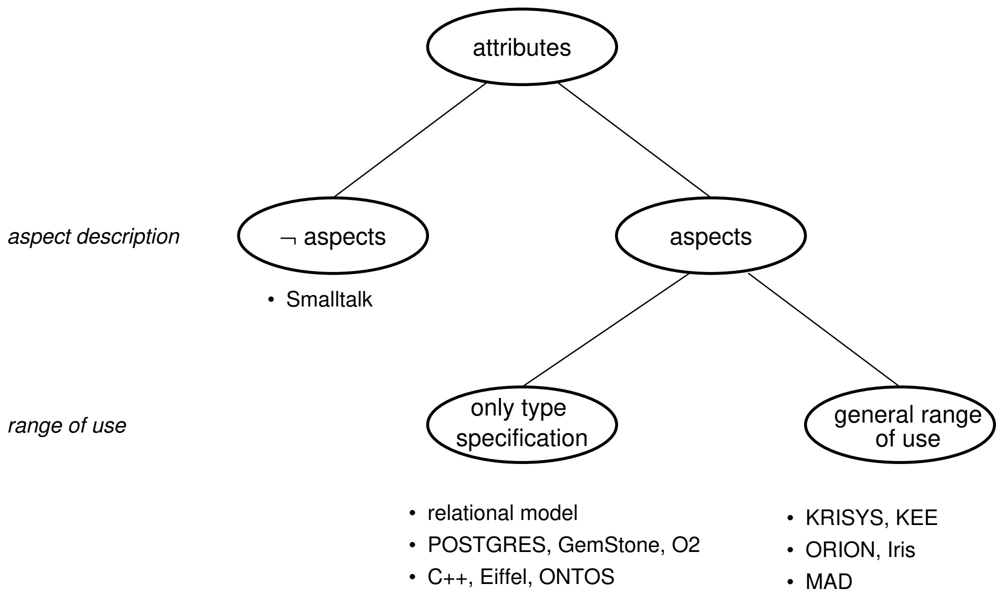


Figure 3: Attribute specification

A fine-grained view to attributes is inclined to the existence of *aspects* . In fact, aspects are assumed as facilities to specify some constraints on declarative properties such as possible values, default, cardinality, etc. As a special case, they are only used to specify an attribute's type as it is the case in systems with typed attributes (e.g., the relational data model, C++, ONTOS, POSTGRES, and GemStone). Here, we do not distinguish whether a type specification is optional, as is the case in GemStone, or whether it is mandatory as for example in the relational data model or in C++. A more general range of use additionally allows for explicitly defined integrity constraints via specific aspects, for example, for cardinality restrictions, default values, used-defined specifications, etc. as done in systems like KRISYS, KEE, ORION, and MAD for example. Astonishingly, systems like Smalltalk have neither typing nor any other kind of aspect description. Therefore, they have no ability to explicitly define any constraints that could be used for system-enforced integrity. Instead, everything has to be embodied in the code of methods. With this, Smalltalk stands alone facing approaches that support any kind of typing or attribute description

Here, we have restricted the use of aspects with respect to declarative properties. Of course, it is conceivable to enlarge the applicability of aspects also to express constraints on procedural properties (i.e., methods). For example, KRISYS allows such specifications. There, demon aspects (in the sense of procedural attachment) might be used for constraint specification on both descriptional as well as procedural properties. With this it is possible to specify a so-called before demon that is evaluated before method invocation and an after demon for evaluation after the method is performed, thus checking specific constraints before and/or after method execution. Eiffel uses pre- and post conditions for this purpose. For procedural properties, we can also perceive type specifications by means of the so-called signature concept that combines the name of the method, its parameters and their types/classes as well as the method's result type/class into the method's signature.

3. Methods: procedural properties

The initial purpose of defining methods was to protect (or to encapsulate) the declarative properties of objects from being read and manipulated in an arbitrary way. A second reason was to provide more abstract and problem-oriented ways of working with objects, e.g., having a function called 'hire_employee' instead of inserting tuples and updating attribute values in a lengthy and error-prone process.

To stress the desired degree of data independence and the gap between the user and the object, the notion of 'sending a message' has been chosen as the only means by which a user can access an object. A message has to be sent to an object to inquire about its state or to change its state. The object may or may not accept the message and return a value as a response. A message consists of a receptor, a selector, and a varying number of arguments. The receptor specifies the object to which the message is addressed. The selector tells the receptor what the user wants, i.e., it acts as a function name or operation code. The arguments are function parameters or operands as usual. If the receptor accepts the message for processing, it analyses the selector to determine the method to be executed. This is called binding, and we will return to it later on.

Modifying object states only by methods seems reasonable because it guarantees consistent state transitions supposing that the methods work correctly. Reporting object states through messages, however, seems awkward and cumbersome, but nevertheless has its advantages: The object hides the number of internal attributes as well as their storage representation. It may even compute values from a number of other attributes. Thus, the internal object structure can be changed (at least to some extent) without any user being affected. This is much like the view concept in conventional databases. If exported attributes are read-only, their access and the invocation of a parameter-less state-reporting method can be made indistinguishable for the user, supposed an appropriate syntax is used (e.g., 'person.age' not only as an attribute access, but as a method invocation, too; this is the case in Eiffel).

Hardly any system carries object autonomy that far. In fact, some systems do not even support encapsulation, so that the user can access the declarative properties (the attributes) directly. To do so, generic operations can be used, i.e., a query language or data manipulation language (DML). We discuss query and manipulation languages in a separate section (section 4.5), there taking also into account the concepts for structuring purposes to be introduced in the meantime. Still such a system can have methods. Their purpose then is to simplify access to an object (instead of making it possible at all) as well as to integrate application-oriented operations into the object description. They are sometimes called 'stored procedures'. For instance, in POSTGRES they are used to locate components of the object under investigation from various other relations.

Although the purpose of a method is to report or change the state of one object, it can usually send messages to any other object, too. This is a very powerful mechanism, but it can be misused in the sense that state changes propagate through the network of objects in an uncontrollable manner (see [32] for some issues to avoid this by means of good programming rules). From the user's point of view, this may look like an immense number of side effects.

Of prime importance are the issues of binding and overloading [6]. Their dependency on message sending and method execution is depicted in Figure 4. As mentioned before, *binding* describes the process of locating the procedure code of an object method given a receptor and a selector. Assuming that there is a user program containing a 'send message' operation to some object in some syntax, the binding can either be done at compile time ('early binding') or at run-time ('late binding').

The first criterion to be considered here is that of method modification: If methods can be changed at any time during system operation - including deletion and insertion -, the only choice is late binding. So, there is no way to restrict overloading and message specifications to achieve binding at compile time. Hence, we have used only one dependency edge to represent this in Figure 4, although various combinations of overloading and message specifications are also possible here. This is the case of KRISYS, KEE, ORION, and POSTGRES. (Please note that an 'early' binding with an invalidation mechanism is just another form of late binding).

If methods are not modified at run-time (that is to say, are not modified after compilation of the user program), the next crossroad is that of method invocation. If the syntax of the send-message statement in

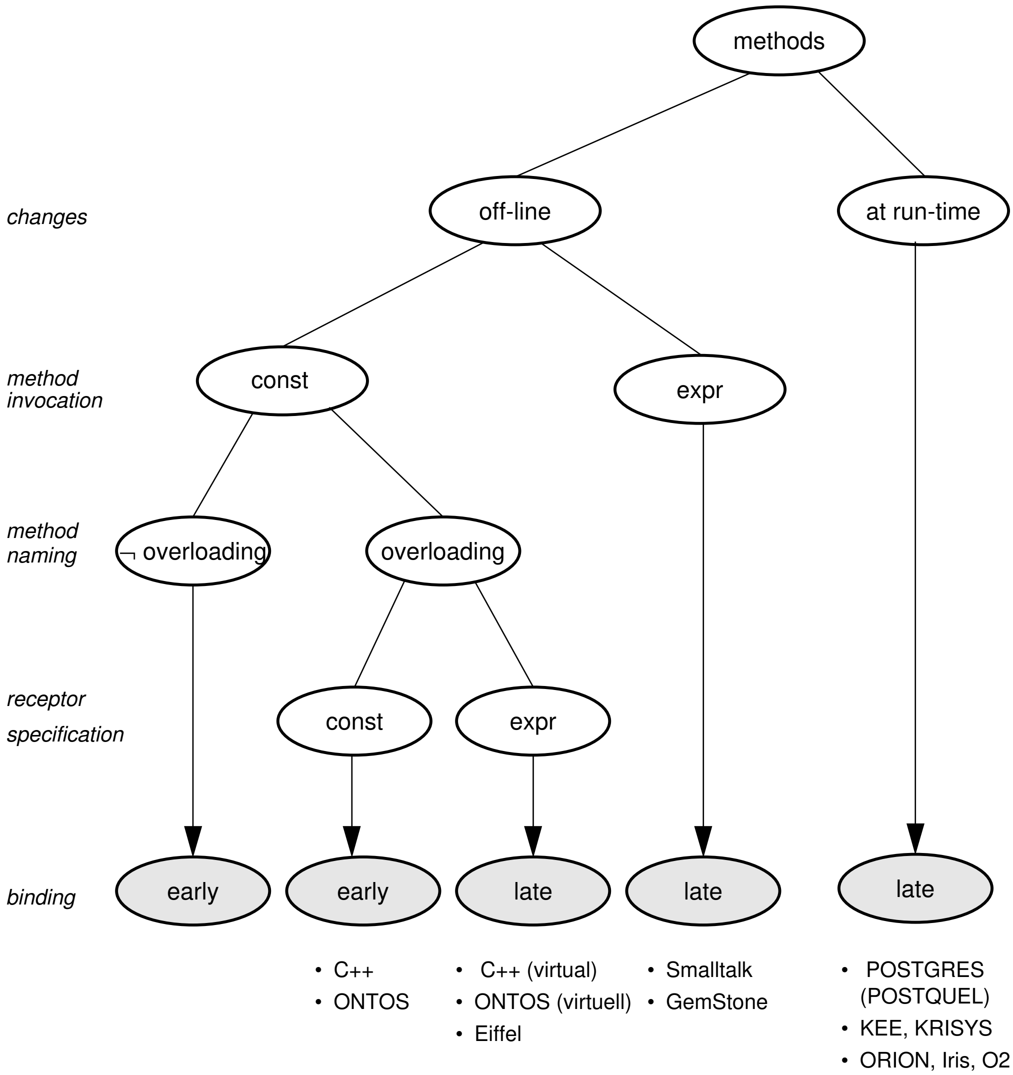


Figure 4: Concept Terrain for Methods and Binding

the user program allows for variables or even expressions used in place of the selector, again, the only choice is late binding. However, if the selector must be specified as a constant, other choices show up due to the next crossroad of *overloading* . Overloading means that methods in different objects can be invoked using the same name (the same selector). It requires the existence of different implementations of a same method. If overloading is not allowed, selectors are unique. Therefore, the dependent choice is that of early binding. Note that in this case the way the receptor is specified is irrelevant for the decision on binding, since the selector alone suffices to identify the receptor.

In case of overloading, the object identification cannot be derived from the selector, but must be taken from the receptor specification in the send-message statement. Again, it can either be a constant or an expression. Here, early binding is possible only if the receptor is known at compile time, i.e., it is specified as a constant. In the other case, where the receptor specification is done by variables or even expressions, the remaining choice is that of late binding.

We have tried to introduce the concepts of late binding and overloading without referring to classification and generalization, since they are discussed in subsequent chapters. However, one can imagine that early binding becomes even more difficult in cases where methods are not stored with the objects themselves but with their classes, or superclasses of their classes in a class hierarchy. This will be discussed in a more general setting in the next chapter.

4. Object Abstractions: structural properties

Also very important features associated with object-orientation are the so-called abstractions. They permit the suppression of specific details of particular objects, emphasizing those pertinent to the problem or view of information at hand. They enrich the semantics of the underlying model since they directly reflect the constructs people apply in the process of describing the real world [38].

Abstractions are in general expressed as relationships between objects, having as their purpose the organization of these objects in some desired form. There are several kinds of abstraction relationships: instance-of, subclass-of, element-of, subset-of, part-element-of, and subcomponent-of. Depending on the kind of relationship existing between objects of the model, the objects, like the real world entities, can play different roles (i.e., class, instance, set, element, aggregate, component) characterized by very different semantics. However, the organization of these objects by means of grouping them together according to the specific semantics of an underlying abstraction concept is the inherent property of all abstraction concepts:

- grouping of components in the aggregation,
- grouping of elements in the (set) association, and
- grouping of instances in the classification.

Consequently, an object may be found in different abstraction-specific groupings.

4.1 Classes

Classification is one of the most important and the best understood form of abstraction. It provides means for introducing generic information into the model by allowing one to refer to a class as a representative or prototype of several other objects (i.e., the instances), which have the same properties and constraints.

Actually, our notion of a class encompasses three different concepts that are often not distinguished properly. The first one is a structural definition for a group of objects, that is the names and types of their attributes and their methods. This is to avoid repeated definition and to enforce certain integrity constraints ('a human must have an age' - even if it is 'unknown'). The second one refers to the production of new objects, or instances of that class (i.e., the object factory). This concept can be provided either by the class itself as is the case in some programming languages (e.g. Smalltalk) or by the run-time system (e.g., C++, Eiffel). In either case the class definition serves as a template. The last notion covers the group of objects, that is the instances of a class. As we said before, this is inherent to each abstraction concept and therefore also part of the meaning of 'class'. Further on, the notion of such a 'homogeneous' set is very important in performing search.

Our class definition differs from some others, which view a class only as means for grouping some objects of one type defined by a type system [5].That is, they treat the structural definition of a group of objects as part of the semantics of types and only the meaning of grouping (building up collections of objects) as part of the semantics of classes. The reason for this separate meaning of types and classes comes from the need for a representation of various collections of objects having the same structure, i.e., type (for example, 'green cars', 'red cars', 'expensive cars', ' cars from California', ' cars from Europe'). However, it is not necessary to make such a distinction because there are other (better) ways to express these collections. One way is the definition of subclasses with more restrictive constraints (narrowing the value range of attributes, cf. section 4.1.2) and another way is using the association concept (cf. section 4.2). For this reason, our notion of class includes the above meaning of types and the notion of grouping the instances as already mentioned at the beginning of this section.

Missing here is a notion of 'interface' of a class (or of all its instances), which is necessary when encapsulation is considered. In our opinion, this is the right notion of the 'type' concept (thus resembling the use of this term in programming languages). A type restricts the way certain objects can be used. If access to an object is through methods only, the type comprises just those methods, their selectors, parameter types (classes), and return type (class). Hence, two different classes that offer the same set of methods with different internal (hidden) attributes and different method implementations can still be of the same type and consequently behave in the same way (from an external point of view).

Even if encapsulation is less strict in that the attributes are visible, method implementation can still be hidden. Then, the type must include the attribute types as well. However, classes can still be different internally (method implementation) and have the same type.

4.1.1 Handling Classes in a Model

There are basically two different ways to treat classes in a data model. First, classes can be viewed as a kind of meta-information defining the structures of objects of the model. In this sense, they are, in general, incorporated in the DB-schema or in the type system of the programming language so that they are not treated as regular objects of the model. This is the way conventional data models view classes. The other viewpoint eliminates the difference made by conventional DBS between classes incorporated in the DB-schema and instances incorporated in the DB. In this case, classes and instances are treated as regular objects of the model so that definitions of static structures as meta-information are not completely separated from the objects themselves (i.e. in a database catalogue or in compiler code). Consequently, classes and instances are treated in the same way although they can have very different semantics in particular contexts. For the user, it is, therefore, possible to apply operations to manipulate classes as he/she does to instances. Additionally, since classes are treated as regular objects of the model, they may have their own attributes, methods, and related integrity constraints. Thus, this shift of viewpoint covers the role of classes as the 'object factory' (supported, for example, by Smalltalk) because 'create an instance' is usually one of their methods. It is important to note, however, that if classes are treated as regular objects, they are instances of another class as well. Usually, this 'super' class is denoted *object* or *global* . Since global is again an object, it is also an instance of another object, which is, in this case, itself. Such a well-known paradox can only be 'solved' by making some exception in the model as for example 'global is not an instance of any other object'.

The road map concerning classification is presented in Figure 5. The first criterion to be considered here is the existence of classes (in either one of the above mentioned ways) in the model. If classification is not supported, no typing concept at object level is provided by the system. This is the case of LISP [54] and the ACTOR language [3,4,33]. However, even if classes can be defined in the model, a typing concept at object level is not mandatory. This depends on whether the existence of an object is necessarily bound to its role as an instance of a particular class or not. That is, whether the system allows for the definition of objects by their own without any association to a particular class. This is only possible if any object may contain the specification of attributes, methods, or integrity constraints, i.e., the system allows for a kind of '*self-describing objects*'. This is the case of some expert system modeling tools as well as of the KBMS KRISYS. If properties may be defined only in classes, an object can only exist if it has a class associated with it, otherwise it would be an 'empty' object. Systems following this approach (Smalltalk, GemStone, relational DBS, POSTGRES, ORION, ONTOS etc.) allow only for class-described objects. For this reason, instances cannot change or extend the structure imposed by their classes since there is no facility to manage definitions in the instances. Therefore, the insertion of an object is conditioned to the specification of its class. That is, the object relationship to classes is, in this case, always mandatory.

In the case of class-described objects, all attributes as well as integrity constraints defined by a class must be pertinent and applicable to all its instances. In other words, objects always have exactly the structure specified by their classes. Therefore, systems following the road that leads to 1:n relationships

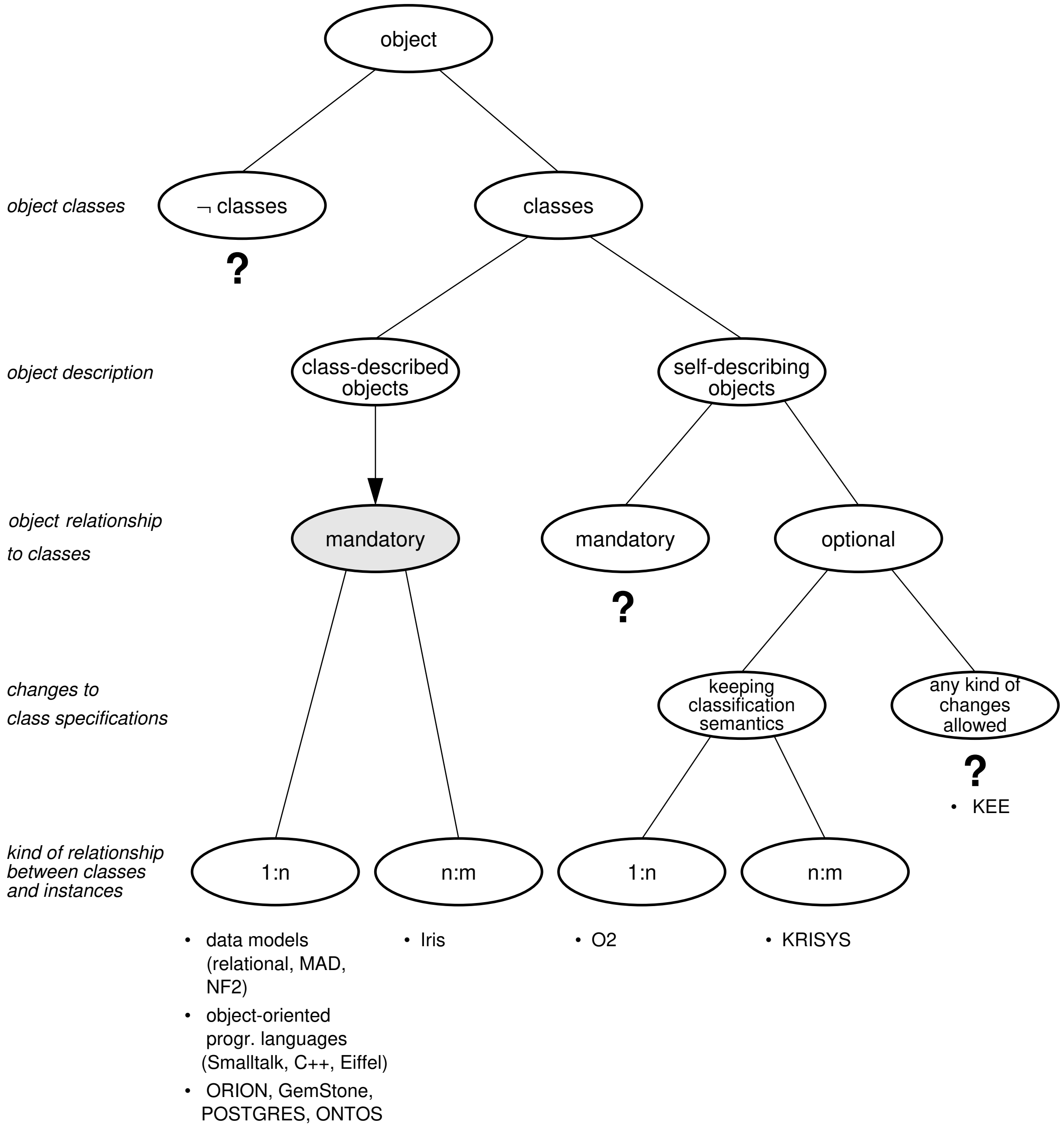


Figure 5: Object classes and their relationship to instances

between classes and instances are adequate only for supporting applications with a strong homogeneity within an object type. This final destination corresponds to the approach mostly followed by DB systems. On the other hand, if one takes the road leading to n:m relationships, heterogeneities within an object type may be handled by specifying additional classes and relating instances to several different classes. What we mean here is that an object can be made an instance of any number of classes (using e.g. the “AddInstance” database command of Iris), irrespective of relationships among the classes. In particular, if an object can only be an instance of two classes at the same time under the condition that they are connected by the subclass relationship (see next section), we would still consider this to be of the “1:n” type. For this reason, most systems attempting to support such applications provide ways to incorporate attribute and constraints definitions directly into the object description leading to the self-describing objects crossroad. These systems sometimes also address the problem of modeling instances that differ from the structure imposed by their classes. By providing self-describing objects, they can allow attributes that are passed on to instances to be modified in order to reflect a special, desired structure or behavior in a particular instance. While most DB systems do not support such a concept of exceptional instances, the O2 system as well as most expert system tools (e.g. KEE, KRISYS) allow instances to redefine the structure and the behavior of their classes.

In the case of being self-describing, objects may exist on their own. Here, it is not logically impossible to combine the above mentioned semantics of mandatory classes with the authorization of instances to change or extend the specification received from their classes. However, this does not seem to make sense since all the facilities for self-describing objects are there anyway, and nothing can be gained from forcing objects to be an instance of some class. Optional relation with classes can be allowed at no extra cost.

When the relationship between objects and classes is optional, classes are only defined when several objects have the same structure. Here, classes are employed to abbreviate the descriptions introduced into the model in order to avoid error-prone redundancy. Starting from this crossroad, there are two basic ways to view the description associated with a class [38]. The first one, which is the way most expert system tools view classes, is that classes simply characterize prototypical instances. As a consequence, attributes and constraints specified by a class may be contradicted by instances of it. At a first glance, this prototype role of classes seems to be very appropriate since it allows for a direct representation of exceptions. However, it contradicts the semantics of the abstraction concept of classification, hindering the definition of operations valid for all instances. Systems following this approach (e.g., KEE) do not provide means for introducing generic information through classes impeding a more efficient implementation. As a consequence, the abstraction usually gained by classification cannot be appropriately employed for query processing and optimization.

The second way, which represents the approach followed by O2 and KRISYS, views the description associated with a class as necessary conditions (i.e., attributes and constraints) that each instance must satisfy. Thus, instances of a class may have some additional attributes and constraints or even restrict the constraints received from their classes but are not allowed to contradict those defined by the class.

Here, changes of methods have to be considered apart:

- If one views the behavior of instances as part of the object specification determined by classes, all instances must act in the same way. Consequently, modifications of methods must be prohibited (for some exceptions, see section 4.1.2.1).

- However, if one considers classes as only responsible for the specification of the interface of methods (i.e., one explicitly distinguishes between interface and behavior), changes on methods do not conflict with this semantics of classification, and may, for this reason, be allowed.

4.1.2 Class hierarchies and Inheritance

Sometimes, it is important to have a form of referring to several different classes, or even to all their instances, abstracting the details of each one. *Generalization* provides this way by allowing more general classes, or *superclasses*, to be defined in order to capture the commonalities existing between these different classes, or subclasses. Since generalization may be applied recursively to specify other more general superclasses, it provides the means for organizing classes in a class hierarchy.

The most important feature associated with such hierarchies is the concept of *inheritance*. Since properties kept by superclasses are generally also valid for their subclasses, there is no need to repeat the description stored in the superclass for each of its subclasses. These properties are just inherited by each subclass. There are several different aspects involved with inheritance, as illustrated in the road map shown in Figure 6.

4.1.2.1 Default vs. Strict Inheritance

The starting point of this part of our tour is, consequently, the existence of generalization hierarchies, otherwise it makes no sense to talk about inheritance. From this starting point, the first criterion to be considered is the kind of inheritance supported by the model, which in turn, determines the semantics of class descriptions. If inheritance is done by default (*default inheritance*), inherited properties and associated constraints can be modified. That is, they are passed on only if they are not *overridden* or redefined in a subclass. In this case, superclasses are treated as 'prototypes' or 'blueprints' of their subclasses so that exceptions as well as refinements of specified descriptions can be directly handled in the model. For example, even if there is a property in the description of the class person which says that all persons have an age in the interval {0..100} and teenagers is a subclass of persons, teenagers are defined to have an age in a more restricted interval, and thus the inherited property should be redefined in the subclass. If inheritance is strict (*strict inheritance*), i.e., all inherited properties and constraints must exist and must not be overridden or redefined in a subclass, superclasses are viewed as templates of or regulations for their subclasses. In this case, the representation of exceptions as well as refinements cannot be supported, since subclasses must have all the unchanged properties and constraints of their superclasses and may have new properties and/or new constraints in addition to them. For this

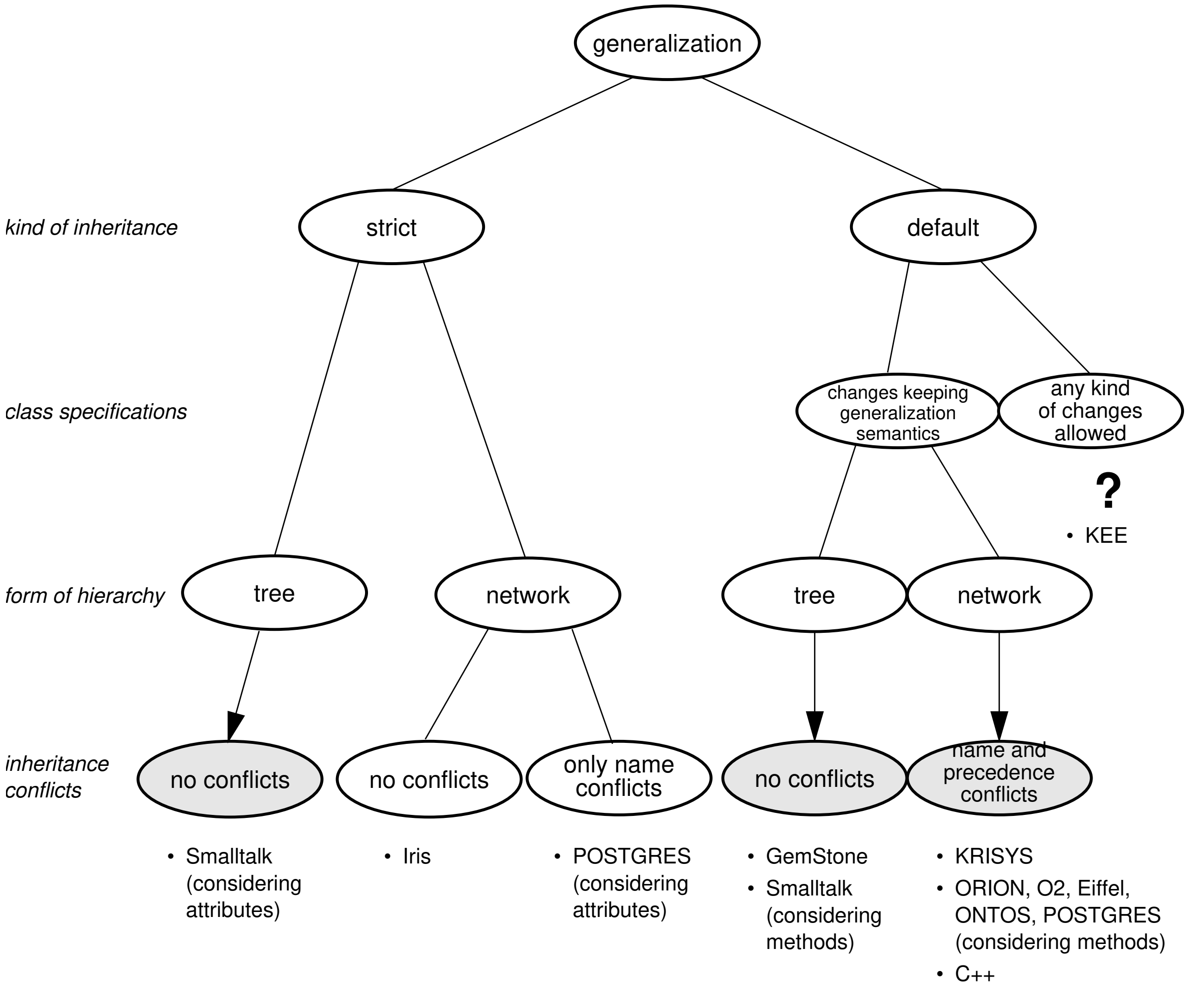


Figure 6: Class hierarchies

reason, it would not be possible to assert that teenagers are persons having their age in the interval {10..19}. On the other hand, the support of strict inheritance simplifies the implementation of an object-oriented system. Strict inheritance guarantees that properties specified in a superclass are found in every instance of its subclasses and with exactly the same semantics (i.e., integrity constraints). Therefore, there is no difficulty in defining operations specific for an object type.

It might be important to point out here that different types of information can be passed on by means of a generalization hierarchy defining conceivable 'heritages' that an object may receive. The simplest of such heritages comprises just the attributes and is also known as *type inheritance*. If, in addition to an attribute, also its value is inherited, then it is the case of *value inheritance*. Further on, it is also conceivable that the attribute and only the associated constraints are inherited or that the attribute, its constraints, and its value are passed on via the hierarchy. Of course the kind of inheritance (strict or default) and the heritage (type, value, constraints) are independent from each other allowing for different combinations of the two categories. For example, the combination of type inheritance with strict inheritance or the combination of value inheritance with default inheritance are somehow obvious. Others are conceivable, too: the combination of type with default inheritance allows for homogeneous generalization hierarchies where specialization is still permitted.

Also important to mention is that systems do not necessarily have to support a single kind of inheritance for all different types of information passed on. Actually, they impose distinct inheritance semantics to each type of information. So, Smalltalk, for example, supports a strict inheritance of attributes but a default inheritance of methods. In KRISYS and KEE, on the other hand, the inheritance of attribute names is strict, whereas their aspects are inherited by default.

In many systems (e.g., most DBS, Smalltalk, and KRISYS), methods are inherited by default, although these systems do not provide mechanisms to 'specialize' the implementation of such inherited methods in the subclasses. In this case, to support some degree of specialization, such systems allow inherited methods to be overridden in the subclasses, causing overloading. However, in other systems (e.g., ONTOS and in many expert system tools as for instance KEE), where mechanisms to 'specialize' methods are supported, methods are not necessarily overridden, but have their code extended or modified. In general, such mechanisms allow one to take the inherited code (usually called 'wrapperbody') by a subclass and extend it with before and after code pieces (called before and after wrappers), so that the method maintains its behavior from the point of view of the superclass, although it was 'specialized'. Let us assume, for example, a class persons and a method 'have-birthday' which increases the age attribute. Suppose that employee is a subclass of person with an extra attribute salary. One could then specialize the 'have-birthday' method to raise the salary of an employee every two years, that is, whenever his/her age turns out to be an odd number, and the method would still behave the same on all persons' attributes. Thus, such a kind of specialization still maintains the semantics of classification (see section 4.1.1) and generalization.

In the case of default inheritance, it is important to differentiate between changes that are in agreement with the semantics of the generalization concept from those that may contradict such semantics (as done for the classification concept). Generalization requires that all properties (i.e., attributes and methods) as well as integrity constraints specified by superclasses must be valid for their subclasses and (as a complementary concept to classification) to all instances of these subclasses [38]. Consequently, the support of a generalization concept restricts allowed modifications to the addition of new attributes, methods, or aspects or to the specialization of methods or the restriction of constraints, requiring that everything else must be kept constant. (A popular way of restricting constraints is to modify “possible values” to a subclass of the inherited entry.) This means that KEE does not support the concept of generalization (see Figure 6). Since it allows for changes that contradict the semantics of this concept, inheritance can be only used to save typing work. (Note that the above introduced comments on the different views of methods as well as on the problems resulting from the support of any kind of changes in the instances of a class are also valid and relevant here.)

When inheritance is strict or when the applied default inheritance guarantees the semantic of the generalization concept, then every instance of a class is also an instance of all superclasses of that class. In these cases, several implicit (or virtual) relationships between instances and classes are implied by the system. They are called this way because they are not explicitly defined by a user. For example, if John is defined to be an instance of teenagers, the system implicitly generates a relationship between John and persons, since teenagers is a subclass of persons. No explicit relationship between John and persons has to be explicitly specified by the user. Thus, when a relationship between an instance and a class is defined by a user, not only one, but possibly several relationships are set by the system. Although this may imply n:m relationships between classes and instances, we did not consider implicit relationships in our discussion on section 4.1.1.

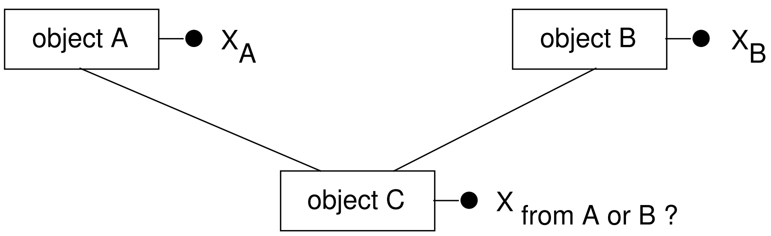
When talking about the kind of relationship between classes and instances in section 4.1.1, only relationships explicitly defined by the user were considered. Transporting the semantics discussed in that section to the context of generalization would mean that the kind of relationship between instances and classes is n:m only if instances can have relationships to more than one class, which are not in a (possibly indirect) subclass relationship to each other. Since a system will never imply such relationships, n:m relationships between classes and instances can only occur if the user is allowed to explicitly define more than one class for a (or several) particular instances. (Remember, however, that the classes of such instances should not be in a subclass relationship to each other.) While most DBS do not allow instances to have more than one class, most expert system tools provide mechanisms to attach and detach instances to and from any class at any time.

Another important observation is the relation existing between *overloading* and the kind of inheritance supported by the system. If information is passed on by default (and only in this case), overloading occurs every time changes caused by overriding or specializing inherited methods are introduced in the class hierarchy. One could then conclude that default inheritance is the indispensable requisite for the occurrence of overloading since strict inheritance does not allow any kind of changes at all. However, this is not true! Actually, overriding and specializations are the most common ways of producing overloading within a class hierarchy, but not the only ones. Even when inheritance is strict and, consequently, redefinitions are prohibited, overloading may occur if the user specifies two or more methods with the same name in distinct objects. In this case, different methods would be invoked by means of the same selector but different receptors, i.e., sending the message to one or the other object (see chapter 3).

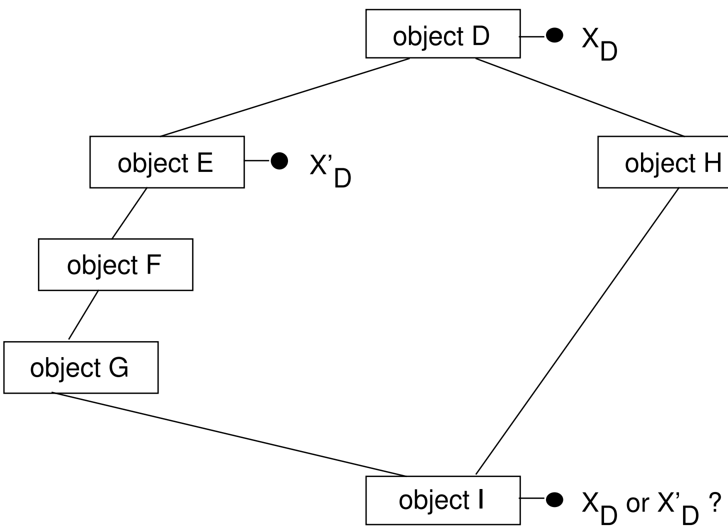
4.1.2.2 Single vs. Multiple Inheritance

Proceeding our tour on generalization, the next crossroad corresponds to the two forms of class hierarchies. Actually, the form of a hierarchy determines the amount of heritage received by an object. For example, systems providing tree-like hierarchies allow only for *single-inheritance*, i.e., objects receiving information from one single superclass (e.g., Smalltalk and Gemstone). On the other hand, network-like hierarchies may also support *multiple-inheritance*, i.e., objects receiving information from several classes (e.g., POSTGRES, ORION, O2, ONTOS, KRISYS, KEE).

Multiple-inheritance reflects the conceptualization of the universe of discourse more directly. In general, real world objects present different characteristics corresponding to the several distinct roles they play in the application. For example, trams have properties that describe their characteristics as a vehicle running on railways and properties defining their role as a vehicle for public transportation. However, the price one pays for this flexibility is reflected on the more complicated implementation of multiple-inheritance as well as on the mechanisms necessary for handling conflicts.



a) example of name conflict



b) example of precedence conflict

Figure 7: Multiple-inheritance conflicts

There are basically two kinds of conflicts that may occur during the inheritance of information: name conflicts and precedence conflicts. Name conflicts, as illustrated in Figure 7a, occur when two attributes, methods, or aspects with the same name have been defined for distinct classes of a hierarchy and are inherited by a common class. Such conflict occurs because systems, in general, do not allow for the existence of duplicates of attribute or method names of a same object or of duplicates of aspect names of a same attribute or method. Precedence conflicts can only occur in environments inheriting information by default. They occur when an attribute, method, or aspect has been once defined, however modified in some place down in the class hierarchy (see Figure 7b). Note that such conflict would not take place if the property had not been modified. In this case, any well implemented system would recognize that it deals with the same property (i.e., originating from one single same class) however being inherited on two different ways. The problem with precedence conflicts is to identify which of both inherited information is 'more specialized' (i.e., has precedence over the other) since it is the one that has to be taken. Such identification is not always trivial because the specification that is closer to the object in which the conflict is occurring is not necessarily the preceding one (observe, for example, the case illustrated in Figure 7b).

Summarizing, we can conclude that inheritance conflicts may occur only in network-like class hierarchies. Name conflicts require, additionally, the occurrence of overloading of any kind, whereas precedence conflict may happen only when overloading has been generated by means of modifications on information inherited by default.

4.2 Sets

Frequently, it is necessary to group objects together not because they have the same properties (classification) but because it is important to describe properties of the group of objects as a whole. To model such situations, one has to use the abstraction concept of *association* . According to it, another kind of object, called set, is introduced to represent these kinds of properties. Therefore, in the association concept, the details of the element objects are suppressed and the properties of the group are emphasized in the set object [38]. This concept has its origin in set theory, so that it does not allow for the existence of duplicate elements. Consequently, it is different from the concept of bags [5].

The starting point of our tour on association is, therefore, the existence of sets in the model (Figure 8). If they are not supported, no set-orientation at object level can be provided by the system. This seems to be the case of most programming languages. However, sets are not the only prerequisite for set-orientation. It is also necessary that the system exploits their existence in the query, manipulation, or definition languages provided. This is, for example, lacking in some expert system tools [16,CI85,13] which

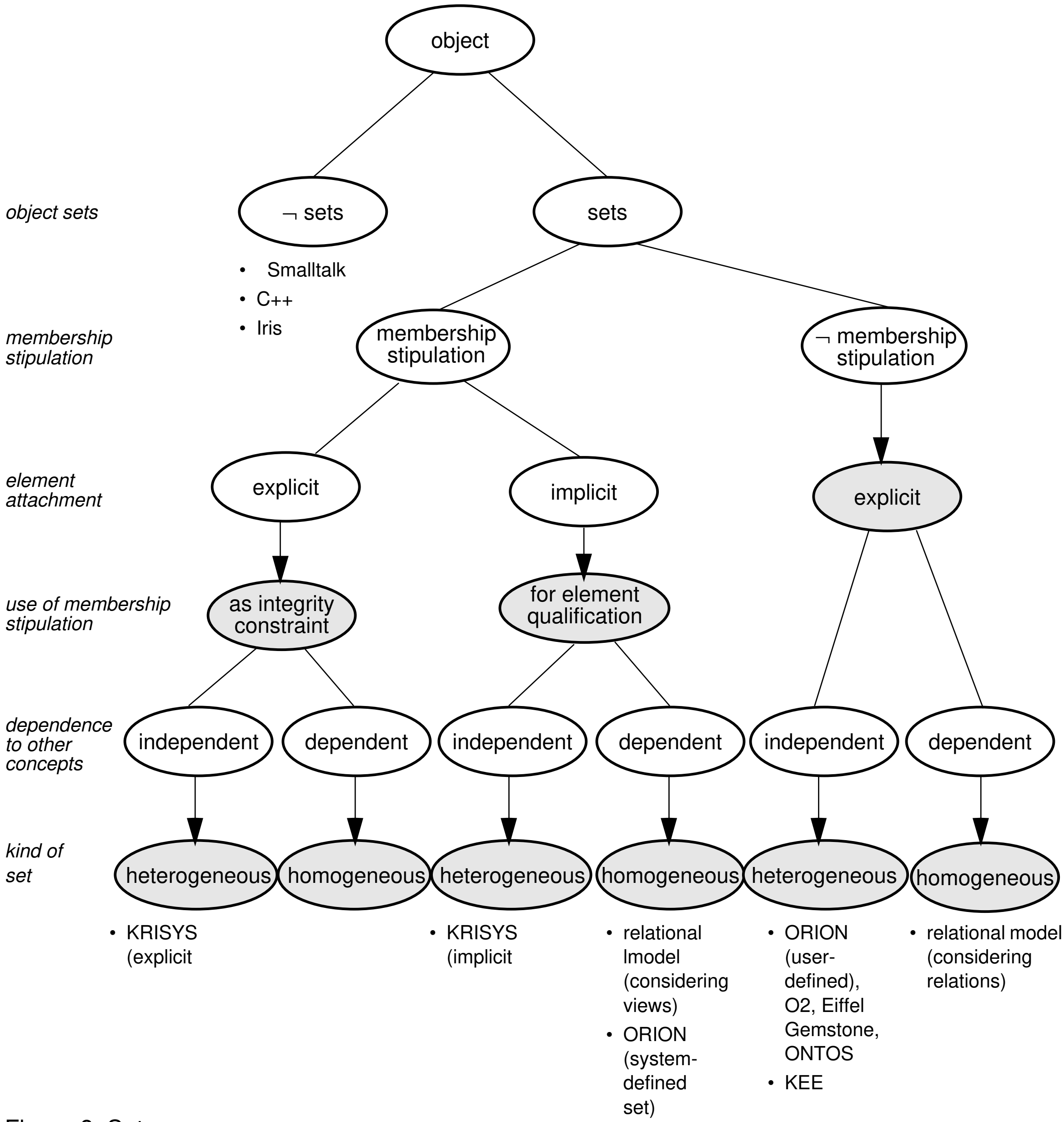


Figure 8: Sets

incorporate some of the semantics of sets in classes but do not provide the user with set-oriented operations that manipulate collections of objects.

4.2.1 Handling Sets in a Model

If sets are supported, one has to decide whether or not set descriptions may contain the specification of a kind of condition which has to be valid for each element of the set. This specification, denoted *membership stipulation* [38], is a kind of attachment criterion for the elements, i.e., an object is an element of a set only if it satisfies the set's membership stipulation.

Membership stipulation, just as inheritance, is another kind of built-in reasoning facility provided by the abstraction concepts [38] and therefore treated as another important crossroad of our road map. It may be used by the system either to make deductions (i.e., element qualifications) or as the basis for controlling system-inherent integrity constraints. For example, a model supporting membership stipulations could determine whether a change in an element object may be allowed in order to not violate the semantics of its relationship to a set. In case of violation, the system could either reject such a change (membership stipulation treated as integrity constraint) or accept it. In the latter case, the violated relationship would be dissolved and new ones, now satisfied by the 'new' object, would be created by the system (membership stipulation treated as means for qualifying elements).

Actually, the kind of reaction performed by the system is not governed by the membership stipulation, but by the way elements are attached to particular sets. When elements are explicitly connected to sets by the user, for example by means of an insert operation, the system has to utilize the membership stipulation as an integrity constraint that rules the semantics of the sets. So, the insertion of objects not satisfying the stipulation has to be rejected in order to keep the integrity of the model. If the attachment of elements is not defined by the user, the only way to find out which objects are valid elements of a set is to use the membership stipulation as qualification criterion of the set. In this case, objects not satisfying the stipulation criterion are simply not qualified.

It is important to observe that the above definition of sets and of the two distinct ways of attaching elements build a sound basis for the definition of different kinds of collections to be supported in a model. They represent therefore important crossroads in the map of Figure 8. For example, views in the relational model can be treated as sets with a membership stipulation used for element qualification (i.e., representing the view definition). Relations, on the other side, can be treated as sets without membership stipulations, whose elements have to be explicitly attached by the user (by an insert operation).

Proceeding our tour on association, the next crossroad corresponds to the decision whether association exists as an independent concept or only bound to another abstraction concept. Systems allowing only for sets bound to other abstraction concepts (mostly classes) cannot support the definition of sets of heterogeneous objects. If sets can be heterogeneous and are attached to classes (i.e., are typed), then the problem arises that for a same set two different (minimal) classes can be deduced depending on the actual construction of this set and on the existence of class hierarchies. Thus, by binding association to classification, object qualifications are invariably restricted to selections over an object type (i.e., class). This is, for example, the case of the relational model considering either views or relations. However, such dependencies are not mandatory. Imagine, for example, a query to qualify all objects (independent of their type) having a particular value in a specific attribute or in any attribute, corresponding to something like

SELECT *
FROM *
WHERE attribute = value

SELECT *
FROM *
WHERE * = value

In this case, the resulting set would not be defined by any class so that association would be an independent concept. The specification of set/element relationship would not be based on the structure (or type) of the elements so that sets of heterogeneous objects would result.

4.2.2 Set Properties

In the next step of our tour, one has to define the way sets should be represented in the data model (Figure 9). Basically, there are two different ways to do this. First, sets can be treated as any other regular object of the model in an explicit manner. In this case, they are treated just like instances, classes, and elements although they are additionally characterized by a very distinct semantic. The other way, which corresponds to the view of conventional data model, considers sets as the extension of a collection. In this sense, they exist only as some definition in the DB-schema and are not incorporated in the DB as real objects. They are only implicitly represented as a group of objects which either are instances of a base relation or have been qualified by a view or query.

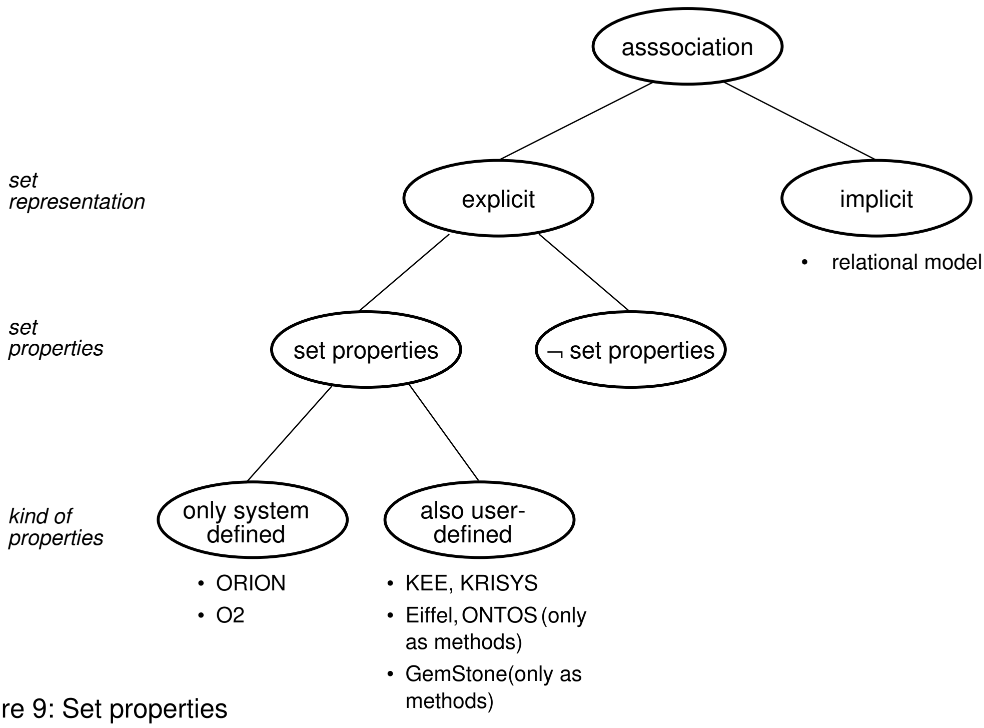


Figure 9: Set properties

When sets are explicitly represented, they may have their own attributes, methods, and aspects. Such set specific characteristics are denoted set properties and can be used to describe properties of the group of elements of a whole. At this point, another important observation, which has been considered at the next crossroad of Figure 9, refers to the kind of set properties allowed by the model: only system-defined (like number of elements, most recently inserted element) or also user-defined. These properties express characteristics of the set as a whole, but they are determined based on properties of the elements, for example: average, maximum, and minimum of values. Thus, while changes in the classes imply downward changes in the model (provoked by inheritance), changes in the elements causes upward modifications in the set properties. For this reason, systems allowing for user-defined set properties must supply the user with means to specify how changes in the elements should be propagated to the sets.

4.2.3 Set Hierarchies

Analogous to classification/generalization, one can imagine to have ways to refer to several different sets, or even to all their elements, abstracting the details of each one. Set association provides these ways by permitting the specification of more general sets, or *supersets* , in order to capture set properties defined over a wider spectrum of elements. Thereby, subset-of relationships are generated, organizing the objects into an association hierarchy.

Similar to the assertion made for a generalization hierarchy that every instance of a class is also an instance of the class' superclasses, elements of a set are here also elements of the set's supersets. In the same way, when supporting association, one has to decide which form of set hierarchies should be supported. Actually, the form of a hierarchy determines the degree of disjunction existing between sets and, consequently, the homogeneity of membership stipulations and set properties. Tree-like hierarchies allow only for completely disjoint sets whereas network-like hierarchies permit the definition of several sets over the same elements for the representation of, for example, different groups of set properties.

However, independent of the form of set hierarchies, a set object can be a subset of another only if it presents a more restricted membership stipulation, otherwise it cannot be guaranteed that its elements are also elements of its supersets. This provides means for embodying a refined notion of integrity constraints into the model. This also enables the system to derive subset-relations from the membership stipulation defined for two given sets. Thereby, the calculation of the set properties for the subset will be consequently restricted to the corresponding subgroup of elements of the superset, determining different (probably, more specific) values for these properties. Finally, this allows the construction of hierarchies of collections of a particular type by defining various sets with different (more general or more specialized) membership stipulations, but all referring only to the instances of this type.

There are only very few systems allowing for an object organization on association hierarchies. KRISYS and KEE are examples of them.

4.3 Aggregates

Frequently, it is necessary to treat objects not as atomic entities, as classification, generalization, and association do, but as a composition of other objects. This conforms the abstraction concept of aggregation, according to which a collection of objects is viewed as a single higher-level object.

The above definition should make clear that aggregation, like the other abstraction concepts, involves two types of objects. The simple objects are here called *parts* or *components* . They express atomic ones, i.e., those objects which cannot be further decomposed. Together, they build the lowest-level composite objects or simplest aggregates. These can, in turn, be used to build more complex higher-level composites so that an aggregation or part-of hierarchy is formed.

In general, data models (above all those supporting the semantics of complex objects [43, 47, 50, 35]) allow for the specification of references between objects. Such references, however, do not capture any special semantics, serving only as flexible means for the representation of any kind of relationships between the objects. Aggregation relationships (or part-of relations), on the other hand, have a very specific meaning. They represent the idea that an object 'consist-of' other objects, thereby expressing that an object cannot 'consistently' exist without its parts. Note that this requirement makes the aggregation concept quite different from the others, since classes may exist without their instances, and empty sets may occur in the association. Therefore, it is necessary to observe the importance of modeling aggregates by means of special-purpose relationships. This permits the incorporation of the semantics underlying the notion of aggregates in the system, allowing for operations that directly make use of such semantics, for example, to

- determine the components of an object,
- find aggregates of which an object is a part,
- create an aggregate together with all its components in one single operation, or
- remove all parts of an aggregate when it is deleted.

So, the first step of our tour on aggregation is the support of such a notion of aggregates, which is not provided by Smalltalk, GemStone, ONTOS, and C^{++} (Figure 10).

4.3.1 Dependent vs. Independent Components

When supporting aggregation, two types of part-of relationships must be distinguished: dependent or independent [27]. On the basis of this refined semantics, the existence of parts will directly depend on the existence of its aggregates or not. In the case of a dependent relation, for example, the deletion of an aggregate will recursively remove all its parts, freeing the application from the search and subsequent deletion of all parts of the removed object. However, dependent relations do not allow for later reuse of objects in another aggregate, what is certainly desirable in some application environments. The concept of dependent vs. independent components is orthogonal to all other aspects discussed w.r.t. aggregation and is, for this reason, not presented in Figure 10.

Like association, aggregation is not dependent on the structure of the objects. Just on the contrary, it provides means for the definition of highly-structured objects, allowing a user at a particular moment to view such structured object as a single unit (i.e., together with its parts) and at another time to view only parts of it as independent or dependent objects. By the aggregation, objects are, therefore, treated in one context as objects of their own and in another as components of other objects.

4.3.2 Aggregation Hierarchies

If the concept of aggregation is available, then it is obvious to exploit it more than once, i.e., resulting in aggregation hierarchies. Therefore, a restriction to only one level of aggregation is not overly meaningful. Hence, we have marked this crossroad with a question mark and we do not know any system that belongs to this category. When observing components within an aggregation hierarchy, one can differentiate between *exclusive components* (i.e., those being part of only one composite object) or *shared components* (i.e., those being part of several aggregates) [27]. Actually, the kind of components supported by the model determines the form of allowed aggregation hierarchies (see Figure 10). Tree-like hierarchies provide only for exclusive components, whereas network-like ones also for aggregates with common/shared parts. This is reflected in Figure 10 by dependency paths between the respective crossroads.

In the case of shared components, the support of object identity turns out to be quite important. Note that if object identity is not supported, aggregation has to be represented by means of attributes (of type set, list, table, etc.) which contain the components of the aggregate. Hence, shared components can only be expressed by repeating the same components in different objects, resulting in redundancy in the representation that must be controlled by the system.

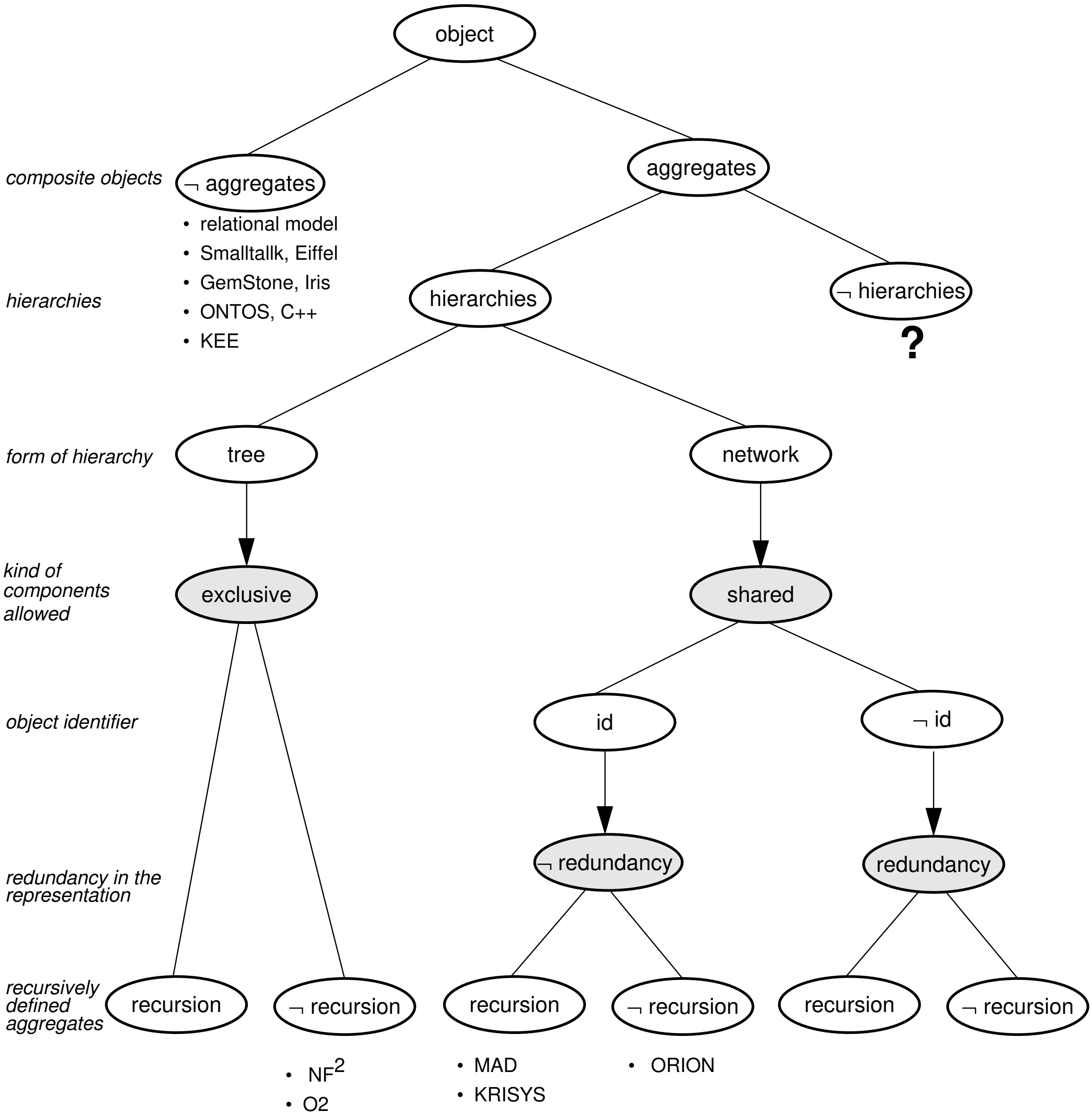


Figure 10: Aggregation

Another important criteria to be considered here and therefore used to define crossroads is the support of *recursion* . This is certainly the necessary concept for representing aggregates that at the type level are recursively defined, e.g., the parts of automobiles consist of other automobile parts.

Finally, the support of reasoning facilities on aggregation hierarchies should be considered. In other words, whether the system allows for the specification of *predicates* based on common properties of components and aggregates by means of which it may either control the semantic integrity or make conclusions about aggregates and/or components [38, 48]. For example, by expressing that the value of a particular property is upward implied (e.g., weight, completion time), the system may control whether it increases going upward over the aggregation hierarchy, thereby either rejecting changes that violate such predicate or triggering further changes to keep the truth of the predicate.

4.4 Integrated View of Object Abstractions

4.4.1 Handling Different Object Roles

Until now, we have isolated the discussion about each abstraction concept from the others. Naturally, in the real world, such isolation occurs seldom. Objects may obviously be instances of classes, elements of sets, and components of aggregations at the same time. Hence, objects may have up to six types of relationships to other objects (instance-of, subclass-of, element-of, subset-of, part-element-of, subcomponent-of) corresponding to the different roles they can play in each abstraction hierarchy. Consequently, the semantic meaning of the objects can be very different from context to context in which they are found. There are objects expressing instances in one context and elements in another, and even objects representing classes, sets, and components in different contexts. Thus, the semantic meaning (i.e., the role) of an object can only be determined in regard to a particular context (i.e., together with the corresponding abstraction relationship). As a consequence, object descriptions may embody both classification/generalization and association as well as aggregation properties.

The criterion to be analyzed here is therefore related with the number of different descriptions which may be integrated into one single object, i.e., how may *different kinds of roles* an object is allowed to play (Figure 11). Systems allowing for objects with just one role necessarily make a strict distinction between 'regular' objects representing data (e.g., instances) and 'meta' objects describing other objects (e.g., classes).

Relational DBS, for example, define the role of an object regarding whether it is part of the DB or of the DB schema. (Here, we are considering the contents of the DB schema as objects, too). In other words, relational DBS do not allow for an integration of several abstraction roles into one single object of the model: objects are either instances (when part of the DB) or classes (when part of the DB schema). Consequently, the representation of real world entities having several abstraction roles have to be spread out among several objects of the model, and the system has no way to recognize the tight connection existing among them. (For example, between the tuple with primary key 'refrigerator' in the relation 'products of company X' and the relation 'refrigerator' in which every refrigerator produced by company X is stored.) Additionally, since objects may play one single role and there is a clear distinction between classes and instances, no ways to allow for changes on the kinds of roles of an object during its life time can be provided.

In contrast to relational DBS, systems allowing for objects with different roles provide a more integrated view of abstraction concepts. The more integrated such view is, the more information about a real world entity may be concentrated into one object of the model. This has an additional advantage. By considering all abstraction concepts together, descriptions of one concept can use the specification of the others, enabling much richer and more precise descriptions of world entities. In this way, objects can be concisely described without losing the necessary flexibility, and some classical errors provoked by the introduction of redundancy can be avoided by reducing the amount of repetition in the descriptions.

Nevertheless, not all systems providing objects with several kinds of roles allow for a free combination of such roles. In order to reflect this feature, we have introduced a criteria to measure the degree of role combinations. In general, systems prohibit the combination of class and instance roles at the same time. In ORION, for example, instances may be either components of other objects or aggregates but not classes. On the other hand, Smalltalk, GemStone, and ONTOS for example, present an exception by permitting classes to be instances of so-called meta-classes. That is, it still makes a distinction between classes and instances but enables a combination of class and instance roles (at least) on the class level.

Another important observation introduced as a criteria in Figure 11 considers the kind of changes allowed on the roles of the objects. The feature *static* means that an object must keep the role that is given at its creation unchangeable. Since ORION allows for the definition of aggregates dynamically, it supports dynamic changes of object roles. (Note however that in ORION this is restricted to the role of aggregates and parts and not to instances and classes.)

The final and most flexible approach is to enable a free combination of abstraction roles. Systems supporting this do not differentiate between data and meta-data, thereby treating all objects in the same way although it exactly knows the semantic meaning of each one in every particular context of an abstraction concept. For this reason, it is therefore possible to apply operations to classes just as they are applied to instances, so that any kind of dynamic changes are allowed. (Note that systems providing such an issue can be used as a kind of modeling tool as is the case of KEE and KRISYS).

4.4.2 Handling the Object Structure

Concentrating several roles into just one object has, however, some further implications. For example, objects playing the role of a set and a class at the same time will also pass on the set properties to its instances. This is, clearly, not desirable. To avoid this problem, it is important to differentiate set and class properties (those defined for the instances) in order to restrict inheritance to class properties. This is achieved in some systems by including a property type (instance or own) associated with each property. In this way, instance properties are those concerned with classification/generalization and own properties those with the association. Instance properties are therefore inherited since they describe characteristics of the instances of a class, while own properties are not inherited since they describe characteristics of the set objects themselves.

In the case that an object plays the role of a class in one context and of an instance in another, it is also important to avoid inheritance of instance properties. Clearly, properties which are inherited by the object in the second context because it is an instance should not be further inherited in the first context where it is a class since they describe characteristics of the object related only to its existence as an instance. Because of this, such systems generalize the meaning of the own-properties by not restricting them to the association concept. Therefore, own properties express properties of any object itself, e.g., an instance or a set. (In Smalltalk, such own and instance properties are denoted respectively class and instance variables.)

By combining aggregation with the other concepts, it is also important to distinguish the aggregation properties from the other properties of the objects since values of aggregation properties are to be interpreted as other objects of the model. As a consequence, some systems introduce in the description of each object property (in addition to the type own or instance) an indication as to whether a property describes an aggregation relationship or not. KRISYS, for example, indicates part-of properties as non-terminal ones since their values correspond to other objects of the model, whereas the other ones (generalization/classification and association properties) are denoted as terminal properties. (Similar differentiation is also made by ORION).

4.4.3 Handling Specifications

In addition to explicit specification concepts it is conceivable to apply also implicit specification that is based on e.g. rules or triggers. In a first stage we can think of those implicit specification concepts being applied at the attribute level. Instead of explicitly setting the attribute value, we might give a 'derivation' (or 'derivation rule') that, if performed, produces the actual attribute value. For example, support for derived attributes is a main feature of the CACTIS system [25], where any attribute can have attribute evaluation rules associated that describe how the attribute value can be derived, and that are (re-)evaluated when the object is referenced. POSTGRES embeds either a QUEL query, a single procedure or even a trigger/rule, and KRISYS offers a generic demon concept for that purpose. Further implicit specification concepts have already been mentioned before. One example is the transitivity of class instances that is implicitly expressed by the transitivity rule telling that the instances of a class are also instances of the class' superclass as well as instances of its super-superclasses and so on; the same implicit transitivity applies for association and aggregation abstractions. In section 4.1.2.1 we have termed the relationships resulting from that transitivity implicit or virtual relationships. Another example also men-

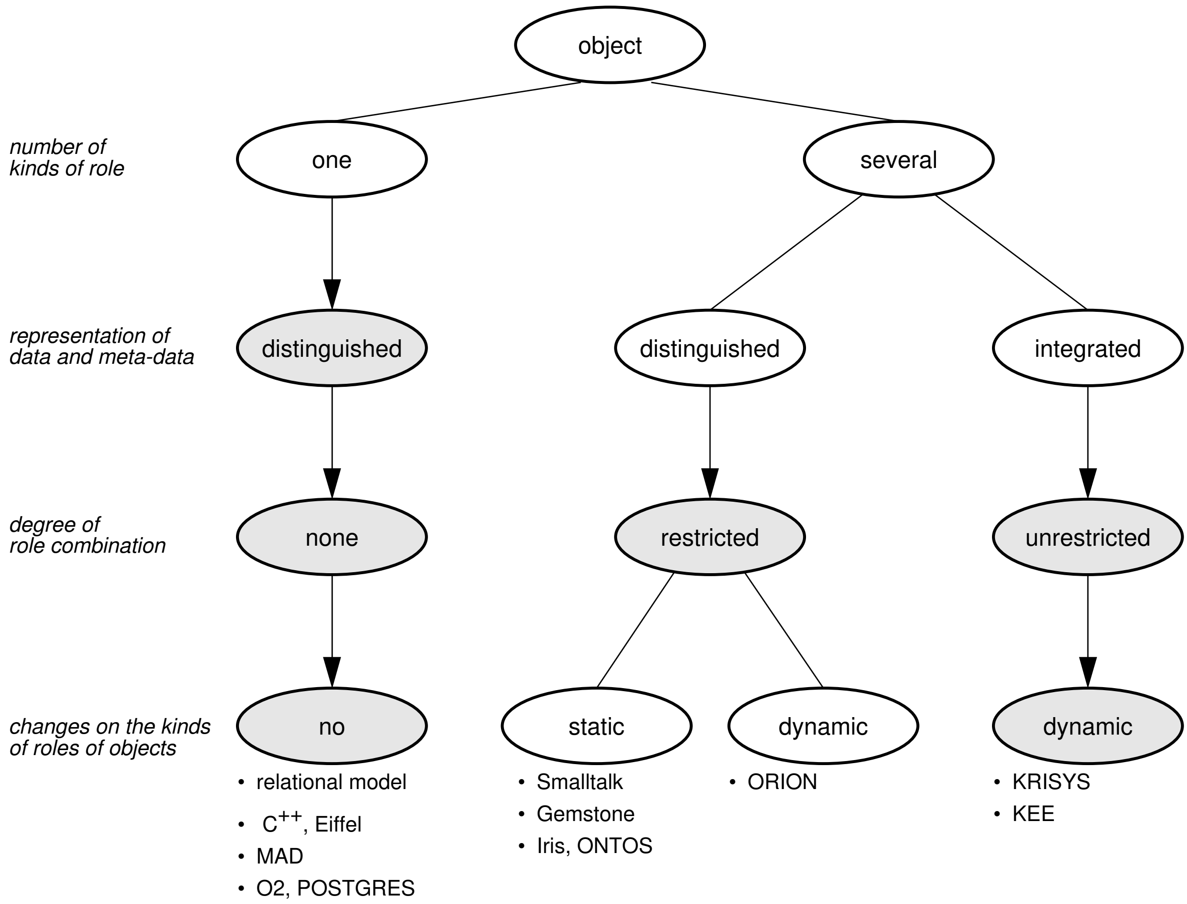


Figure 11: Integration of abstraction roles

tioned before are membership stipulations that can be used to automatically insert elements (or sets) into their sets implicitly setting the corresponding association relationships (cf. section 4.2.1).

Automatic classification of instances into their corresponding classes as well as automatic subsumption of classes into the class hierarchy are examples of implicit specification, where the system knows the 'insertion rules' to be applied. These concepts mainly stem from the area of modeling tools and are mostly applied by systems based on KL-ONE [14] as e.g. KRYPTON [9]. In addition to these implicit specification concepts we can also perceive explicit ones. For example an explicit classification rule telling that a person with age less than 20 has to be classified as a teenager, i.e., inserted as an instance of the teenagers class. Another kind of explicit classification rule might be exemplified by the rule telling that all (direct as well as indirect) instances of class trucks that have a loading capacity of more than 10 tons are as well elements of the set called 'potentially military vehicles'.

This last example is very close to what views are used to, i.e. intentional classes based on view definitions. In fact, there are OODBMSs, like the COCOON system [49], where a view concept is used to implicitly define a class that is then automatically inserted into the class hierarchy. As other declarative approaches we can view the Frame-logic [31] that integrates objects, predicates, and rules as well as the rule-based approach towards a deductive object-oriented database language reported in [2].

4.5 Query Languages

Originally, object-oriented systems have been designed without any ad hoc query facility. Designers considered it to be irrelevant, because a rich set of methods should be available to retrieve and access objects. This is particularly justified, if classes, sets and aggregates are themselves managed as objects with their own set of methods, e.g. a select method available on each class (ORION [Kim89], OPAL/GemStone [37]). However, not everything can be easily assigned to a class; consider the equivalent of the famous relational join operation. In [21] it is shown why associate retrieval is difficult to support through methods only and advantages of an embedded query and manipulation language are demonstrated. Additionally, as pointed out in [8], a query language should be easy to use and need not be computationally complete; it will be used to retrieve data without having to write a program. For example, ONTOS provides its Object-SQL as a report generation tool as well as an ad hoc query language. If classes, sets, and aggregates are represented as objects themselves, their methods can provide the power of a query language, otherwise the facilities of a query language must be provided by the run-time system.

An interesting question is whether query formulation should be allowed to break encapsulation. While this is the case in O2, it is not in ORION. The argument given in [8] does not give proper attention to state-reporting methods, so we would consider the decision still a matter of taste.

5. Conclusion

This paper should be viewed as a first attempt to sort the large variety of concepts related to object-orientation in the database field. The goal was not to add yet other definitions of object-oriented concepts, but to identify existing relationships among these basic concepts that allow one to cover and classify various conceivable combinations of these conceptual building blocks. Thus, the main contribution is the revealing of *dependencies* and *degrees of freedom* among the concepts, in contrast to many other papers that only list them side by side. This makes visible numerous ways to compose different shades of object-orientation leading to alternatives encountered when constructing object-oriented systems. We are certainly aware of the fact that some classification trees could have been refined and discussed in more detail. But we hope to have reached a state of expertise that makes the reader sensitive for the existence of these alternatives and encourage him or her to question the meaning of ubiquitous phrases.

Throughout the paper, we have revealed a lot of concepts that are related to each other. Sometimes even concepts from different areas show some dependencies. For example, the concepts of late binding, overloading, overriding, kind of inheritance, and message concept define one of those 'dependency' groups: one important usage of generalization hierarchies is modeling of subclasses through specialization of the superclasses' attributes and methods. Of course, this is possible only if default inheritance and overriding are valid. Furthermore, overriding provokes overloading, which in turn implies late binding assuming that changes occur before runtime and also assuming that messages have variable selectors or receptors. Note that in this case the language to specify messages might influence these dependencies: if there are only constants allowed to specify selector and receptor, then it is the case for early binding. On the other side, strict inheritance implies no overriding, and overloading is reduced to only name conflicts, leaving early or late binding up to the language used to specify messages. Another group of concepts worth mentioning consists of the concepts set, class, or type. This group is different because it comprises concepts that should be treated independent, i.e., orthogonal, from an abstraction point of view, thus defining degrees of freedom. This is true despite the apparent dependency of set elements normally being instances of (not necessarily) the same class. For example, when the elements of a set are defined to be the instances of the root of a generalization hierarchy, then the instances of the corresponding subclasses are automatically elements, too. Since these instances belong to different classes normally showing different class structures, the set defined is truly heterogeneous. Automatic projection of those instances to the class structure of the root is not a set property, instead it belongs to the semantics of classification and generalization. In general, we have argued for an orthogonal integration of the abstraction concepts allowing for a joint usage and mutual exploitation of the abstraction concept properties. In Figure 11, we have classified some existing systems and prototypes according to their ability to integrate the abstraction concepts. There, it is evident that the systems that do fully integrate the concepts in an orthogonal way are mostly modeling tools, e.g., KEE and KRISYS. The other side of the spectrum where the concepts are kept separate, i.e., not integrated, is taken by mostly database systems, which historically emphasize data management and not modeling.

In classifying some well-known systems and prototypes from different areas (mainly from programming languages, knowledge engineering, and database systems), we have illustrated the effectiveness of this approach. From a high-level view, our expectations were fulfilled because most systems presented expected properties. For example, those with programming language background are traditionally strong in procedural support (i.e., distinctive method properties, cf. Figure 4) and encapsulation (Figure 2). They are not as good in application modeling as the knowledge engineering systems. For that end, knowledge engineering systems provide, beyond procedural and declarative properties, many concepts and features (i.e., structural properties) that allow for precise and flexible modeling (cf. chapter 4, Figure 11). On the other side, systems with database background were targeted towards plain but sufficient modeling in order to provide efficient data management. This becomes especially obvious in Figure 11 where these systems do not allow for integration of their few (and sometimes restricted, cf. chapter 4) abstraction concepts which prohibits the mutual exploitation of these abstraction properties resulting in weak modeling capabilities.

However, our classification has also revealed some surprises during our investigation of specific systems. For example, KEE, being a well-known and heavily used modeling system, allows for quite some modeling flexibility because it supports integration of its abstraction concepts combined with many ways of changing the structures defined (Figure 5 and Figure 6). But, a detailed investigation of KEE clearly reveals that there are still important things missing in order to have full modeling flexibility. For example, in KEE there is no support for aggregation and only a restricted kind of association (cf. chapter 4). This is even more surprising because some systems coming from the database area (e.g., the complex object data models) do already support aggregation directly and association through their view concept. Another surprise was Smalltalk. As already mentioned above, Smalltalk is quite strong in procedural properties due to its origin from the programming language area. But on the other side, since these kinds of systems are known for their support of (software) reusability, it is very surprising that Smalltalk does not support default inheritance (Figure 6) and overriding. In addition to this, Smalltalk has no type support since there is no aspect description (Figure 3). The absence of some important abstraction concepts (aggregation and association) and some restrictions in classification (Figure 5) and generalization (Figure 6) causes only a limited modeling capability.

All in all, we do not want to judge any system mentioned - this is up to the reader, who may use our classification trees to make this judgement. It is neither our purpose to analyze the relative importance of these concepts, because this is application dependent. We believe that what is required by one application area not necessarily has to be supported for another one. In addition, not necessarily the most 'powerful' system (in terms of concepts) is the best. In contrast, more concepts result, in general, in more complex implementations, worse performance, difficulties in understanding, inappropriate usage, etc.

Therefore, we have investigated the concepts from a neutral point of view, presenting (but not evaluating) several degrees of object-orientation. Classifying an application into our classification trees reveals the characteristics of a desired system, which in turn defines the prerequisites for its implementation, representation, and optimization issues. Implications and consequences of the chosen issues are beyond the scope of the paper and therefore left for future research.

Acknowledgment

The helpful comments of S. Deßloch, L. Golendziner, E. Rahm, and U. Schreier are gratefully acknowledged. Further we would like to thank the referees for their suggestions that considerably helped improving the paper.

References

[1] Abrial, J.R.: Data Semantics, in: Data Management Systems, (eds.: Klimbie, J.W., Koffeman, K.L.), North-Holland Publ. Comp., Amsterdam, Netherlands, 1974.

[2] Abiteboul, S.: Towards a Deductive Object-Oriented Database Language, in: Proc. of the 1st Int. Conf. on Deductive and Object-oriented Databases, Kyoto-Japan, Dec. 1989, pp. 419-438.

[3] Agha, G.: Semantic considerations in the Actor paradigm of concurrent computation, in: Seminar on Concurrency (eds.: Brookes, S.D., Roscoe, A.W., Winskel, G.), Lecture Notes in Computer Science No. 197, Springer-Verlag, 1985, pp. 151-179.

[4] Agha, G., Hewitt, C.: Concurrent Programming using Actors, in: Object-oriented Concurrent Programming (eds.: Yonezawa, A., Tokoro, M.), MIT Press, 1987, p. 37.

[5] Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., Zdonik, S.: The Object-Oriented Database System Manifesto, in: Proc. of the 1st Int. Conf. on Deductive and Object-oriented Databases, Kyoto-Japan, Dec. 1989, pp. 40-57.

[6] Bancilhon, F., Object-Oriented Database Systems:, in: Proc. of 7th ACM PODS Conf., Austin, TX, 1988, pp. 152-162.

[7] Bobrow, D., Collins, A. (eds.): Representation and Understanding, Academic Press, New York, 1975.

[8] Bancilhon, F., Cluet, S., Delobel, C.: A Query Language for the O2 Object-Oriented Database System, in: Proc. 2nd Int. Conf. on Database Programming Languages (Shalishan, 1989), pp. 122-138.

[9] Brachman, R.J., Fikes, R.E, Levesque, H.J.: KRYPTON: A Functional Approach to Knowledge Representation, in: Computer, Vol. 16, No.10, Oct. 1983, pp. 67-73.

[10] Borgida, A., Mylopoulos, J., Wong, H.K.T.: Generalization/Specialization as a Basis for Software Specification, in: [11], pp. 87-114.

[11] Brodie, M.L., Mylopoulos, J., Schmidt, J.W. (eds.): On Conceptual Modelling (Perspectives from Artificial Intelligence, Databases, and Programming Languages), Topics in Information Systems, Springer-Verlag, New York, 1984.

[12] Brodie, M.L., Mylopoulos, J. (eds.): On Knowledge Base Management Systems (Integrating Artificial Intelligence and Database Technologies), Topics in Information Systems, Springer-Verlag, New York, 1986.

[13] Bobrow, D.G., Stefik, M.: The LOOPS Manual, Xerox PARC, Palo Alto, CA, 1983.

[14] Brachman, R.J., Schmolze, J.G.: An Overview of the KL-ONE Knowledge Representation System, in: Cognitive Science, Vol.9, No. 2, 1985, pp. 171-216.

[15] Borgida, A.: Survey of Conceptual Modeling of Information Systems, in: [12], pp. 461-470.

[]16 Carnegie Group Inc.: Knowledge Craft CRL Technical Manual, Version 3.1, Carnegie Group, 1987.

[17] Chen, P.P.: The Entity-Relationship Model - Toward a Unified View of Data, in: ACM Transactions on Data Base Systems, Vol.1, 1976, pp. 9-36.

[18] Date, C.J.: An Introduction to Database Systems, Vol. 1 and Vol. 2, 3rd edition, Addison-Wesley Publishing Company, Reading, Mass., 1983.

[19] Dadam, P., et al.: A DBMS Prototype to Support Extended NF^2-Relations: An Integrated View on Flat Tables and Hierarchies, in: Proc. ACM SIGMOD Conf., Washington, D.C., 1986, pp. 356-367.

[20] Deux, O., et al.: The O2 System, in: Communications of the ACM, Vol. 34, No. 10, Oct. 1991, pp. 34-48.

[21] Orenstein, J., Bonte, E.: The Need for a DML: Why a library interface isn't enough, in: handout to OODBTG Workshop, Ottawa, Canada, Oct. 23, 1990, pp. 83-94.

[22] Fishman, D.H., Annevelink, J., Chow, E., Connors, T., Davis, J.W., Hasan, W., Hoch, C.G., Kent, W., Leichner, S., Lyngbaek, P.Mahbod, B., Neimat, M.A., Risch, T., Shan, M.C., Wilkinson, W.K.: Overview of the Iris DBMS, in: [30], pp. 219-250.

[23] Fikes, R., Kehler, T.: The Role of Frame-based Representation in Reasoning, in: Communications of the ACM, Vol. 28, No. 9, Sept. 1985, pp. 904-920.

[24] Goldberg, A., Robson, D.: Smalltalk-80 - The language and Its Implementation, Addison-Wesley Publ. Comp., Reading, Massachusetts, 1983.

[25] Hudson, SE, King, R.: Cactis: Efficient Support of Functionally-defined Data in CACTIS, in: Dittrich, K., Dayal, U., Buchmann, A. (eds.) On Object-Oriented Database Systems, Springer, Topics in Information Systems, 1991, pp. 341-356.

[26] IntelliCorp Inc.: The Knowledge Engineering Environment, IntelliCorp, Menlo Park, California, 1984.

[27] Kim, W., Bertino, E., Graza, J.F.: Composite Objects Revisted, in: Proc. of ACM SIGMOD Conf., Portland - Oregon, 1989, pp.337-347.

[28] Khoshafian, S.,Copeland, G.: Object Identity, in: Proc. OOPSLA'86.

[29] Kim, W., et al.: Features of the ORION Object-Oriented Database System, in: [30].

[]30 Kim, W., Lochovsky, F. (eds.): Object-Oriented Concepts, Applications, and Databases, Addison-Wesley, 1989.

[31] Kifer, M., Lausen, G.: F-Logic: A Higher-Order Language for Reasoning About Objects, Inheritance, and Scheme, in: Proc. of ACM SIGMOD Conf., Portland - Oregon, 1989, pp. 134-146.

[32] Lieberherr, K.J., Holland, I.M.: Assuring Good Style for Object-Oriented Programs, in: IEEE Software, Sept. 1989, pp. 38-49.

[33] Lieberman, H.: Object-oriented programming in Act I., in: Object-oriented Concurrent Programming (eds.: Yonezawa, A., Tokoro, M.), MIT Press, 1987, p. 41.

[34] Linnemann, V., et al: Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions, in: Proc. of the VLDB 88, Los Angeles, 1988, pp. 294-305.

[35] Lorie, R., Kim, W., et al.: Supporting Complex Objects in a Relational System for Engineering Databases, IBM Research Laboratory, San Jose, CA, 1984.

[36] Lyngbaek, P., Kent, W.: A Data Modelling Methodology for the Design and Implementation of Information Systems, in: On Object-Oriented Database Systems (eds.: K.R. DittrichU. Dayal, A.P. Buchmann), Springer-Verlag, Topics in InformationSystems, Berlin a.o. 1991, pp. 207-227

[37] Maier, D., et al.: Development of an object-oriented DBMS, in: Proc. OOPSLA '86, pp. 472-482.

[38] Mattos, N.M.: Abstraction Concepts: the Basis for Data and Knowledge Modeling, ZRI Research Report No. 3/88, University of Kaiserslautern, in: 7th Int. Conf. on Entity-Relationship Approach, Rom, Italy, Nov. 1988, pp. 331-350.

[39] Mattos, N.M.: KRISYS - A Multi-Layered Prototype KBMS Supporting Knowledge Independence, ZRI Research Report No. 2/88, University of Kaiserslautern, in: Proc. Int. Computer Science Conference - Artificial Intelligence: Theory and Application, Hong Kong, Dec. 1988, pp. 31-38.

[40] Mattos, N.M.: An Approach to Knowledge Base Management - Requirements, Knowledge Representation, and Design Issues, Ph.D. Thesis, Computer Science Department, University of Kaiserslautern, April 1989.

[41] Meyer, B.: Object-oriented Software Construction, International Series in Computer Science, Prentice Hall, 1988.

[42] Mitschang, B.: A Molecule-Atom Data Model for Non-Standard Applications - Requirements, Data model Design, and Implemlentation Concepts (in German), Doctoral Thesis, University of Kaiserslautern, Computer Science Department, Kaiserslautern, 1988.

[43] Mitschang, B.: Extending the Relational Algebra to Capture Complex Objects, in Proc. of 15th VLDB Conf., Amsterdam, The Netherlands, 1989, pp. 297-305.

[44] Nierstrasz, O.M.: A Survey of Object-Oriented Concepts, in: [30], pp. 3-21.

[45] Olle, T.W.: The Codasyl Approach to Data Base Management, Wiley & Sons, 1978.

[46] Ontologic Inc., ONTOS Developer's Guide, Version 2.01, Billerica, Mass., May 1991.

[47] Paul, H.-B., Schek, H.-J., Scholl, M.H., Weikum, G., Deppisch, U.: Architecture and Implementation of the Darmstadt Database Kernel System, in: ACM SIGMOD Conf., San Francisco, 1987, pp. 196-207.

[48] Rosenthal, A., Heiler, S., Manola, F., Dayal, U.: Query Facilities for Part Hierarchies: Graph Traversal, Spatial Data, and Knowledge-Based Detail Supression, Research Report, CCA, Cambridge, MA, 1987.

[49] Scholl, M., Laasch, C., Tresch, M.: Updatable Views in Object-Oriented Databases, in: 2nd Int. Conf. on Deductive and Object-Oriented Databases, Springer-Verlag, LNCS 566, 1991, pp. 189-207.

[50] Stonebraker, M., Rowe, L.A.: The Design of POSTGRES, in: Proc. ACM SIGMOD Conf., Washington, D.C., 1986, pp. 340-355.

[51] Stroustrup, B.: An Overview of C++, in: SIGPLAN Notices, Vol. 21, No. 10, Oct. 1986.

[52] Stonebraker, M. et al.: Third-Generation Data Base System Manifesto, Memorandum No. UCB/ERL M90/28, April 1990.

[53] Wegner, P.: Concepts and Paradigms of Object-Oriented Programming, in: OOPS Messenger, Vol. 1, No.1, ACM Press, 1990.

[54] Winston, P., Horn, B.: LISP, in: 2nd edition, Addison-Wesley Publ. Company, Reading, Mass., 1984.