Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

D.B.M. Otten, P.J.W. ten Hagen

On the role of delegation and inheritance in object-oriented database systems

Computer Science/Department of Interactive Systems Report CS-R9032 July

---

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

On the Role of Delegation and Inheritance in Object-Oriented Database Systems

D.B.M. Otten and P.J.W. ten Hagen
Centre for Mathematics and Computer Science (CWI),
Department of Interactive Systems, Kruislaan 413,
1098 SJ Amsterdam, The Netherlands
Email: daan@cwi.nl

Nowadays, research in the field of Computer Aided Design (CAD) is often directed towards the use of Artificial Intelligence and Logic Programming techniques. For instance, the use of backward and forward reasoning, reasoning with uncertainty and default reasoning. Our research is directed towards the Object-Oriented paradigm. It focuses on the development of CAD systems with respect to: 1) the evolutionary character of the design process, 2) the parallel exploration of possible solutions of the design, and 3) the integration of knowledge of other domains during the design process. We believe that ***inheritance*** and ***delegation***, two communication concepts between objects, are important mechanisms for representing CAD information subjected to the above mentioned issues. Although both mechanisms address the same goal, we believe that delegation better suits the requirements of design support. Both mechanisms have already been implemented in many systems. However, the objects and their relationships are maintained in main memory and are not supported by a secondary storage device. We are developing a Database Management System which is based on the use of these inheritances and delegations. We expect of these mechanisms that more complex structured information and knowledge about design processes and designed artifacts can be stored and maintained. In this paper, the conceptual differences between both mechanisms are explained as well as some of their implementational issues.

CR Categories and Subject Descriptors:
D.1 [Programming Techniques]
H.2 [Database Management]
J.6 [Computer Aided Engineering] - Computer Aided Design (CAD)

Key Words & Phrases:
computer aided design, delegation, inheritance, object oriented.

1. Introduction

During the last few years, progress has been made in the development of CAD systems. Interesting applications have proven the important potential of the use of techniques from the fields of Artificial Intelligence (AI) and Logic Programming, e.g. backward and forward reasoning, default reasoning, reasoning with uncertainty [Davis85a, Kleer85a, Reiter85a].

The current goal of our research is to study the role of databases with respect to the characteristic ways, enumerated below, information and knowledge are treated in CAD. We distinguish knowledge from information because only knowledge includes the notion of activity, i.e. recognition of whether and how knowledge may be applied in the current context. In contrast, information involves only context independent data. The characteristic ways of treating information and

Report CS-R9032
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

knowledge can be summarized as follows:

- the evolutionary character of the design process, e.g. the stepwise refinement of an artifact,
- the parallel exploration of possible alternatives of the design and merging the partial solutions into one,
- the application of knowledge of other domains of technology to the description of an artifact. This includes the conversion of one format to another format appropriate to the knowledge of another domain. For instance, in planning the building process of a house, a constructor needs its complete description. Therefore, the design has to be translated into a constructor-usable format,
- the application of knowledge of other domains of technology to an incomplete description of an artifact. This includes the idea that an application, using the knowledge of the other domain, has to make assumptions concerning unknown information. These assumptions may lead to the completion of the description of the artifact. For instance, knowledge about fire security may affect the design of a building by necessitating a certain solution to be selected.

The kinds of information and knowledge involved in the designed artifact and the design process are:

- information about the designed artifact itself, e.g. its decomposition,
- information about decisions in the design process of a designed artifact which can be used as knowledge for the design of new artifacts, e.g. why a certain form of roof is chosen for a building,
- knowledge about design processes, e.g. sketching is done before detailing,
- knowledge about the intended use of the artifact, e.g. expertise about the environment in which it will be placed and its functionality in this environment,
- knowledge about the order in which knowledge from other fields of technology can be used.

Studying the literature on the development of new generations of CAD systems, we see that many of them are *object-oriented.* Object-oriented means that information, knowledge and computation are factored in objects which themselves can be factored again. This line of decomposition is continued until objects cannot be decomposed any more or until they are considered as elementary objects. Information, knowledge and computation are stored as methods. Attributes, a form of information, are represented by two methods which store a value in and retrieve a value from private memory [Arbab89a]. These methods are called attribute-methods.

In this object-oriented approach, two mechanisms are known which supply i) a structure for representing information, knowledge and computation as methods in objects and ii) a communication protocol for interaction among the objects in order to provide sharing of methods. The mechanisms are called *inheritance* and *delegation.* Although the underlying ideas of both mechanisms address the same goal, the implemented mechanisms differ in structure and communication protocol. Here, the concept of both mechanisms is explained but a more elaborated explanation is given in Sections 5, 6 and 7.

Inheritance follows a set-oriented approach in which similar objects are considered as a higher level generic object, called *class.* The objects themselves become *instances* of the class. Repeated application of this creates an inheritance structure which is little flexible. Interaction between an instance and the outside world is performed by a public interface which is defined by the methods contained in the class related to the instance. To provide sharing of information, knowledge and computation, inheritance supplies a communication mechanism in order to pass methods down the relationships of the class-structure.

Delegation follows a prototype-oriented approach to build a network of prototypes. This network is, in comparison to inheritance, more flexible. A prototype fulfils the task of a class as well as the task of an instance. Delegation supplies a communication mechanism which delegates tasks to other prototypes by means of messages. The receiver of the message will perform the task on behalf of the message's sender and returns the result of the evaluation.

This paper will present a definition of inheritance and delegation, their differences and their roles in defining more powerful databases. We try to give evidence that on the basis of inheritance and delegation CAD systems will give i) better representation of the designer's knowledge, ii) better performance for retrieving knowledge, iii) easier implementation of CAD tools and shells.

The paper is organized as follows. In Section 2, our research background and aims are explained. We describe some shortcomings of current inheritance-based systems when they are used in a CAD environment. To adjust these systems to the characteristic ways of treating information and knowledge in CAD, three additional requirements are needed. These are also mentioned in Section 2. In Section 3, four kinds of abstraction mechanisms in the field of database technology, viz. *generalization*, *aggregation*, *association* and *classification*, are described. We explain why inheritance and delegation can only be based on generalization and aggregation and not on association and classification. Section 4 gives an intuitive description of inheritance and delegation. Sections 5, 6 and 7 explain the differences between inheritance and delegation by describing the features concerning the object-structuring, the features concerning the way method-sharing among objects is provided and some secondary features which spring from other features. Sections 8 and 9 contain the conclusions and describe future work, respectively.

2. Research Background

Our research encompasses the investigation and development of an inheritance and/or delegation-based database system better suited to represent and store information and knowledge in the field of CAD. We believe that current object-oriented databases (OODB), even if equipped with an inheritance or delegation-like mechanism, do not use these mechanisms adequately when they are applied in the field of CAD. They do not meet the requirements of storage and retrieval which are demanded by the way CAD treats information and knowledge.

Firstly, these systems are not suitable for the dynamic creation and deletion of objects. In database management systems (DBMSs), a distinction can be made between the expert-level and user-level. At the expert-level, the database schema is defined. At the user-level, data can be stored and retrieved according to the schema. Once the schema is in use, it is not intended to to be changed any more. This separation of levels is reinforced by the fact that changing the schema of an already filled database is a risky and time consuming task. In CAD systems, a similar distinction can be made. At the expert-level, information and knowledge are conceptualized as, for instance, classes. At the user-level, these concepts are used in order to store and retrieve data. However, the design process alternates between expert and user-level. A designer work from a functional description and works out the details of the artifact to be designed, step by step during the design process. The experiences obtained by using the concepts are fed back to the expert-level and are used to redefine and refine the concepts. Therefore, database systems applied to the field of CAD should integrate the expert and user-level and support the continuous change of the database schema.

Secondly, besides Is-A-relationships, Part-Whole-relationships between objects also have to be supported by the system. As has been described in the work of [Blake87a], Part-Whole-relationships are of great importance in branches of computation such as *model-based computer*

vision and *computer graphics*. It is important not to confuse the Is-A with Part-Whole-relationships:

i) Through the Is-A-relationships, information is passed from superclass to subclass, while through the Part-Whole-relationships information can be passed between any kind of class as long as they fulfil in this relationship the roles of part and whole.

ii) The initiative for *message passing* [Lieberman86a, Lieberman86b] through the Is-A and Part-Whole-relationships lies at the subclass and the whole. They have to know which are their related classes and parts, respectively. Regardless of efficiency reasons, the classes and parts themselves should not know of the existence of any relationship. Unfortunately, the implementation of classes and parts as independent objects is a difficult task because their abstracted information is continuously used by other objects (see e.g. the effects of changing the class-representation and class-hierarchy in Smalltalk-80†).

iii) One of the aims of Is-A-relationships is to make the abstracted information, hidden in the methods of a class, accessible to a subclass. This can, for example, be performed by copying the method from class to subclass. The subclass can then evaluate these methods against its own data-environment in order to calculate a result. Through the Part-Whole-relationships, not the methods themselves become accessible for the wholes but only results of evaluation. The methods are evaluated against the data-environment of the part and the result is passed to the whole. An example of the kind of information passed along the lines of the Is-A-structure is the passing of the speed-calculation method itself from a class **car** to a subclass **convertible**. With this method, an instance of the class **convertible** can calculate its maximum speed. An example of the kind of information passed through a Part-Whole-relationship is information about the length of the part **leg** to the whole **table**. A length-calculating method is evaluated by the part **leg** and the result is passed to the **table**. The **table** can use this result as data for calculating the height of the table.

Thirdly, the system has to be capable of maintaining different representation structures of the same artifact. During the design process the same artifact is often regarded from different points of view. Usually, each view has its own representation structure of the artifact. For instance, the representation structure of an artifact defined by the designer will most likely be different from the one defined by a manufacturer.

We believe that the three points mentioned above are not unique to the field of CAD. Therefore, we expect that the results of our research might be useful in fields like Computer Aided Manufacturing (CAM) and Computer Integrated Manufacturing (CIM).

3. Generalization, Aggregation, Association and Classification

In general, four kinds of relationships between objects can be distinguished in object-oriented systems [Brodie84a]. They are

1) generalization,
2) aggregation,
3) association, and
4) classification.

---

† Smalltalk-80 is a registered trademark of Xerox Corporation

Generalization describes similar objects as a higher level generic object. It describes the Is-A-relationships between objects and places them in a tree or directed acyclic graph. For each leaf and node in this structure all methods (including attribute-methods) of its ascendant(s) must be inherited. Furthermore, in comparison to its ancestor, new features have to be added or one of the domains of the inherited features has to be restricted. Objects with the same generic object(s) as ancestor(s) have equal internal structures. *Specialization* is the opposite of generalization. An example of generalization and specialization is depicted in Figure 1.

In this figure, "vehicle" and "coupé" are specializations of "transport" and "car", respectively. Namely, both entities inherit all features of their ancestors and either some new attributes are added, e.g. wheels and suspension in the case of "vehicle", or the domain of an attribute is restricted, e.g. the attribute containing the shape of the top of the car which is specialized to coupé.

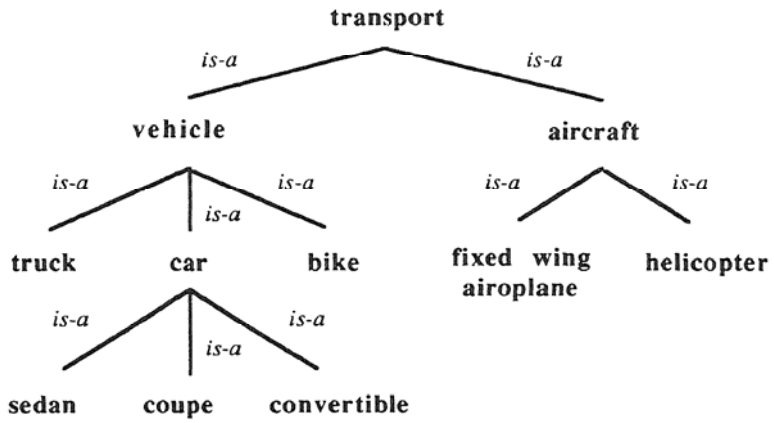

Figure 1 Generalization and specialization

Aggregation describes distinct component objects as a higher level aggregate object. The objects in such an aggregation structure are organized according to a directed acyclic graph. In DBMSs, aggregation often describes artificial relationships. For instance, the component objects **employee-number**, **name** and **address** can be considered as the aggregate object **employee**. Instances of aggregate objects have similar internal structures. The opposite of aggregation is *decomposition*. In design, objects are often considered as an assembly of other objects which themselves may consist of smaller, simpler objects. This assembly is similar to the aggregation abstraction in DBMSs. It describes the Part-Whole-relationships between objects. An example of this is depicted in Figure 2.

Association is an abstraction which describes member objects as a higher level set object by considering equal valued attributes. It describes the *is-member-of-relationship* and is based on the restriction of the domain of some attribute to one element, e.g. the set of round-tables has the value **round** assigned to the attribute **form**. In contrast to generalization and aggregation, association does not require similar internal structures of each of the member objects of the set. Due to this fact and the free choice of which attribute (attributes) is (are) restricted, various kinds of combinations of objects can be made to form a set. Therefore, the association-structure forms a network.

A table is decomposed into legs and a top. The top is decomposed into a layer and a cover.

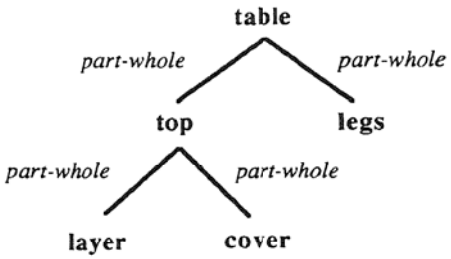

Figure 2 Aggregation

Classification is the means by which a collection of entities can be considered as a higher level abstracted entity. An object class defines precisely which attributes are shared by each object in the collection. Classification represents the *instance-of-relationship*. For instance, an object class **table** with attributes **length**, **width** and **height** has as an instance the object with values 100, 80 and 70. The difference between classification and the other three kinds of relationships is that classification defines an object class for a collection of already existing individual objects. In conceptual modeling, classification is used to identify an object in terms of an object class. The other three kinds of relationships involve the derivation of a new type of object from an existing object by introducing an extra, more specified level of abstraction. They model objects and their related information and knowledge of the real world. These models are used to create instances which are as close as possible to what the user intends to create.

Due to the fact that classification is not a modeling technique, it does not describe how information, knowledge and computation of higher level objects can be inherited by descendants. Therefore, it cannot be used for the definition of inheritance and delegation. Also association cannot be used. Although association creates new levels of abstraction and values can be inherited from ancestors, it cannot be used for our definition of inheritance and delegation. In this paper, we consider only the inheritance of the methods themselves. In Section 7.1.1 "Consistent", we only consider the inheritance of values assigned to attributes. For this kind of inheritance, we treat these values as either global values or as default values. In the latter case, a default reasoning mechanism is needed to maintain the values. This kind of inheritance lies outside the scope of this paper.

4. Inheritance and Delegation

Intuitively, inheritance and delegation can be described as follows. Inheritance is based on the notion of sets of objects with common behaviour. Given a group of objects which are believed to be in some way structured similarly or have similar behaviour, these similarities can be abstracted in an entity called a *class*. Each individual object becomes a member of this class and is called an *instance*. Given this, queries about an instance can be answered by its class. Note

that a class has to be created first before instances can be assigned and used. Let us consider a second group of objects which exhibits behaviour similar to the first group but has some additional specific behaviour. A new class can be created for this second group which defines the specific behaviour of that group and inherits the behaviour of the first group. This second class is called a *subclass* of the first class. Repeated application of this creates an inheritance graph in which the links between the classes represent the generalization relationship. In order to answer the queries which cannot be answered by the class of the instance due to the generalization process, a mechanism is provided to locate and bring back the desired information and knowledge.

Delegation is based on the notion of individual objects, called *prototypes*, which are used as templates for other objects. In such a system, no distinction is made between a class and an instance. In this approach, the understanding of an object is based on the recognition of its structure and behaviour as that of one specific representation of an already known object. At any time, an object may deviate from its prototype and may be changed when new knowledge has to be added or deleted. New objects may become prototypes for other objects. Repeated application creates a structure of prototypes. Queries to a prototype, which cannot be answered by the prototype itself, are delegated to other prototypes. These prototypes try to answer the query on behalf of the originally called prototype.

Recapitulating: both inheritance and delegation provide an object-oriented structure in which information and knowledge can be organized and an interaction protocol for sharing this information and knowledge among the entities in the structure can be provided. However, structure and protocol are implemented differently. The objects in the inheritance and delegation structures are called *classes* and *prototypes*, respectively. In Section 5, 6 and 7, the main differences between inheritance and delegation are explained. Section 8 "Conclusions" contains a table in which these differences are enumerated.

5. Structuring the Objects

5.1. Set-oriented versus prototype-oriented approach

In object-oriented systems, two approaches are used to organize individual objects [Lieberman86a, Lieberman86b]. The first approach is *set-oriented* and is used by inheritance. In this approach, individual objects are considered as sets. The second approach is based on prototyping and is used by delegation. Here, objects are considered as prototypes which act as templates for new objects. Both approaches are elaborated below.

5.1.1. Set-oriented approach

In the set-oriented approach, a class represents a set of objects, a subclass represents a subset of a set and an instance represents one member of a set or subset. From now on, a class is considered as a set of objects and is always used in the context of inheritance. For the definition of a class, the analogy can be made with Abstract Data Types of programming languages. Classes, analogous to types, represent abstracted information and knowledge of real world objects as methods. These classes form the predefined components of a working environment and, analogous to declaring variables in programming languages, they have to be instantiated before they can be used. Instances have private memory for the storage of data, have an object identity and obtain type-information by inheriting the methods of the class. The inherited methods define the public interface between the instance and the outside world.

There are two disadvantages using the set-oriented approach for representing CAD information and knowledge. Firstly, set-oriented approach is characterized by the distinction between

expert-level and user-level. The abstraction of information is performed mostly by an expert. Users, who are generally not allowed to extend or to change classes, create a working environment by instantiating the classes. The disadvantages of this distinction for CAD has been explained in Section 2 "Research Background". Secondly, environments with a heterogeneous set of objects, e.g. CAD, have to face the *one-instance-class problem* [Lieberman86b]. This problem is caused by the necessity of having a class before an instance can be created and results in the creation of a class for only one or a few instances. This violates the class philosophy in which one class represents many instances. Due to both reasons, we believe that this approach is not appropriate for representing CAD knowledge.

5.1.2. Prototype-oriented approach

In the prototype-oriented approach, prototypes are considered as templates for new objects to be created. One of the main differences in comparison to the set-oriented approach is that no distinction is made between a class and an instance. A prototype represents both. Therefore, the distinction between expert and user-level is also removed. Creation of a new object is performed by copying the current state of a prototype and changing it until the desired object is obtained. Due to the fact that the relation between object and prototype is not maintained, changes made to a prototype will not affect an already created object. Note, that "changing a prototype" again implies copying the prototype and then changing the copy.

This approach seems to solve the two problems mentioned in the previous section. There is no distinction between expert and user-level and the one-instance-class problem simply does not exist. However, due to the fact that prototypes are only templates, objects which are created according to the same prototype, but at different points of time, might differ in behaviour. This, and the fact that changes made to a prototype will not affect any of the already created objects, violates the philosophy of sharing abstracted information and knowledge. A solution to this might be the retention of the relation between object and prototype. New knowledge and information added to the prototype will still never be inherited directly by an existing object. However, when an object cannot perform a certain task, it can try whether any of the updates of its prototype can solve the task. By delegating the task to them, the new information becomes accessible to an already existing object.

Such a solution raises a lot of questions. For instance, to what extent can an object be called an update of a prototype or how can the propagation of delegation tasks be controlled? Although solutions are not fully explored yet, we believe that this approach better suits the behaviour of design than the set-oriented approach does.

5.2. Static versus dynamic relationships

In object-oriented systems, the function of a relationship is twofold. Firstly, it is obviously needed to build the structure itself and also to explain the relationships among the objects. Secondly, it is used as a communication-line between objects to provide sharing of methods. The protocol which controls the communication is elaborated in Section 6 "Communication Among Objects". Here, the relationships themselves are explained.

5.2.1. Static relationships

In inheritance, the object-structure is built upon the generalization principle, i.e. relationships between the classes are Is-A-links. The structure of an inheritance system is static, i.e. it is fixed at the time classes become instantiated. Firstly, during run-time there are conceptually no primitives provided to create or modify the relationships between the classes or classes and instances. Furthermore, relationships are static as a result of i) the two application levels, expert

and user-level, in inheritance mechanisms, and ii) classes are implemented as *glass boxes* (see Section 6.2.1 "Glass box") with respect to other objects and classes.

Concerning the first point, relationships between classes are built at the expert-level in which the classes are created and modified. At the user-level, the classes and their relationships are fixed and cannot be modified by the user. In this sense, the relationships are static. However, even at the expert-level these relationships are, to some extent, static. It is not the case that an expert is not allowed to modify the class-structure, but that it is a risky and laborious task. An example of this is Smalltalk-80. In Smalltalk-80, the two application levels are mixed which causes two problems. The first problem is that changes made to classes, e.g. adding attributes or changing global variables, have to be propagated to their descendent classes and instances immediately (see Section 7.1.1 "Consistent"). In large systems, this is a time consuming task. The second problem is that subclasses have direct access to the internal structure of their ancestor classes (see Section 6.2.1 "Glass box"). Therefore, changing the internal structure and behaviour of a class may affect the descendent classes with the possible result that they do not function properly any more. The propagation of this kind of changes cannot be done by the system itself and has to be done by the expert. As has been pointed out, a risky and laborious task.

Such behaviour can hardly be used in an environment in which non-computer-scientists, i.e. people who do not know or do not want to know how the system works, use an intelligent system to describe their knowledge.

5.2.2. Dynamic relationships

In contrast to inheritance, relationships in delegation are dynamic. Firstly, primitives are provided for the creation and modification of relationships during run-time. Secondly, delegation does not require a distinction to be made between expert-level and user-level. Thirdly, the objects are implemented as black boxes (see Section 6.2.2 "Black box"), i.e. *how* a certain task is performed by one object is kept hidden from the other objects. In general, the relationships in delegation mechanisms are Part-Whole-links or Is-Kind-Of-links. An Is-Kind-Of-link is an Is-A-link with the difference that an object is allowed to delete inherited methods. Therefore, the generalization principle cannot be applied. In the context of delegation, these relationships are called Is-Kind-Of. As already mentioned in Section 2 "Research Background", Is-A and Part-Whole-relationships should not be confused.

For our environment in which expert and user-levels alternate and in which we want to abstract from how tasks sent to objects are performed, we prefer dynamic relationships over static relationships. However, the success of using the dynamic relationships depends on to what extent objects can be implemented as black boxes.

6. Communication Among Objects

In Section 5, the differences between the structuring of objects in inheritance and delegation mechanisms were explained. In this section, we elaborate the differences in the interaction among objects in inheritance and delegation. For explanatory purposes it is assumed that these relationships form a network through which information (e.g. messages and results) are exchanged.

The major task of this network is to pass information and knowledge to be shared to other objects. For this, both mechanisms use different strategies. The abstracted information contained in the objects can be distinguished into attributes, i.e. the attribute-methods, and methods. In the next four sections, distinctions are made in:

i) how shared information are passed to other objects,

ii) the degree of information and knowledge-hiding with respect to other objects,

iii) whether separate interfaces are needed to pass messages and results, and

iv) the number of potential objects which respond on receiving a message.

6.1. Method returning versus value returning

When an object cannot perform a specific task, inheritance as well as delegation use *message passing* [Lieberman86a, Lieberman86b] to pass a message to another object. The message is a request to the object enquiring whether it has a method capable of performing the task or not. When it does not have such a method then it propagates the message. This process is continued until no object can propagate the message (in Smalltalk this will lead to an error message) or until the required method is found. In this section, we explain for both inheritance and delegation the actions performed by the object which contains the method. These actions are called *method returning* and *value returning*, respectively. In the next sections, the following different names for objects are used for explanation;

- *sender*: an object which wants to perform a task but does not have the desired method,
- *propagator*: an object which receives the message but also does not have the required method, and
- *receiver*: an object which receives the message and has the required method.

6.1.1. Method returning

In inheritance, method returning is used to pass a method from a class to its subclasses in order to provide inheritance. The objective is that when a certain task has to be performed, a corresponding method is evaluated against the data-environment of the sender. Furthermore, the sender remains responsible for the evaluation of the method. To clarify method returning we define:

- a class Table with characteristics *length*, *width*, *height*, *material* and *four legs and a method to calculate the space (volume) occupied by the table. The method is using length, width and height.*
- a class Desk which Is-A Table, i.e. inherits all characteristics and methods of the class Table, and in addition has a block of drawers hanging from the table top.

To calculate the volume of a certain Desk desk-1, the volume-calculation-method is looked up in the class Table, passed from Table to desk-1 and evaluated against the data-environment of desk-1. To calculate the volume, it will take the locally defined length, width and height of desk-1.

In principle, there are two points in time when methods can be passed. Either they are passed once at the time classes are created and related to each other, or every time when a specific task has to be performed by an object. Conceptually, it is unimportant at which time the methods are passed due to the fact that inheritance provides static relationships. Therefore, at both points in time the relationships between objects are exactly the same and an instance will always end up with the same inherited methods. This is the case even when multiple inheritance is provided and methods are similarly named, because which method will be passed is always defined by the expert and this always happens before both points in time. For efficiency or implementational reasons, one might prefer one of the two. In most current systems, the methods are inherited when they are needed, e.g. Smalltalk-80 [Goldberg83a].

6.1.2. Value returning

In delegation, the strategy for providing sharing of information is different from inheritance. *Value returning* is used to pass the result of a method evaluation from receiver to sender. After the receiver receives the message and decides to respond, it collects all necessary information, e.g. facts from the sender's environment, evaluates the method and returns the result directly to the sender. Thus in delegation, with the message itself and some data, the control for evaluating a task is also passed to another object. To explain *value returning* we use the example of Section 6.1.1. However, the volume-calculation-method is not defined any more as a method of the class Table but as a method of the geometric class Block. The class Block is a specialization of the class Geometric Entities. The class Geometric Entities has a method Volume which identifies the shape of an object and propagates the task to one of its specializations, e.g. the class Block, Cylinder or Sphere. To calculate the volume of Table table-1, table-1 delegates the task to the class Geometric Entity. Geometric Entity tries to identify the shape of table-1 (e.g. by examining the used terminology), gather the necessary information and delegates the task to one of its specialized classes. This class will calculate the volume and send it to table-1. Whether the top of table-1 has a circular or rectangular shape, or whether its shape has been changed over the time, it is up to the intelligence of the class Geometric Entity to identify its current shape and to activate the most appropriate calculation method. Even a refinement of the identification method of the class Geometric Entity will not affect table-1.

6.1.3. Conclusions concerning method returning vs. value returning

From the Sections 6.1.1 and 6.1.2 we can conclude the following. Method returning can be used when the user:

- is able to define the class completely at once and is able to place it in the class-subclass-structure,
- is able to locate the methods which provide the user the required behaviour for the to be designed object, and
- requires that at a later point in time a message will be evaluated in the same way a similar message was evaluated earlier.

Value returning can be used when:

- a complete description cannot be given at once and therefore it is difficult to place the new object in the class-subclass-structure,
- the user does not want to be bothered with the various methods to provide the required behaviour. In our example in Section 6.1.2, the user might not be familiar with the several methods to calculate the volume. When the shape of the table-top becomes circular, the user does not want to reorganize the class-subclass-structure or to locate another volume-calculation-method to finally get the volume,
- it does not bother the user when the same message for the same environment might exhibit behaviour different from the behaviour exhibited before, as long as it is a refinement or improvement.

6.2. Glass box versus black box

Another important difference between inheritance and delegation is the degree of freedom in accessing the internal structure of another object. For instance, in Smalltalk-80 it is possible for a class to take a shortcut to its inherited attributes and to change the values without using the predefined attribute-methods which normally form the interface. This means that classes are implemented as *glass boxes* [Akman87a], visible to other objects and classes. In delegation

mechanisms, access to the internal structure of a prototype must always be done through the predefined interface. Prototypes hide how tasks are performed and act as *black boxes* [Akman87a] to each other. The result of this is that the internal structure of prototypes is better protected.

6.2.1. Glass box

A user normally creates an object with the desired properties by defining the appropriate class for it, possibly by using inheritance from existing superclasses. However, the user can modify a class by adding his own defined methods or overruling the inherited ones. The latter possibility allows the user to alter the existing internal structure. Therefore, superclasses of a class are implemented as *glass boxes*, i.e. at any time one can look through them to have access to their internal structure.

In the following example, we will explain the drawbacks of a glass box representation. Lets define a class FIFO-queue (First-In-First-Out queue) based on the representation of an 1-dimensional array of 10 long. Over this array, we define two methods, *push* and *pop*, and an attribute *Next-Free* which contains the number of the next free array-element. Push stores an item at the Next-Free array-element. Pop retrieves the item from the first array-element and shifts the rest of the elements one place forward.
Lets define a second class Conveyor-Belt which is a subclass of the class FIFO-queue. Conveyor-Belt is a specialization in the sense that there is a time-limit between the push and pop otherwise the items on the belt will fall off at the end. The user can define this new behaviour in a method *Transport* by 1) using the inherited push-method, pop-methods and the method to access the Next-Free attribute, or 2) bypass the inherited methods and access the array directly. The latter case will cause problems when the representation of the queue in the class FIFO-queue is changed from array to pointerlist. The code written for the method Transport will not match the underlying structure any more.

The example shows that changes made to a intermediate class will cause that methods defined in subclasses have to be checked and, possible, have to be adapted to the new representation. This cannot be done by the system itself but has to be done by the expert who defined the specific behaviour of the shortcut. This is a dangerous and tedious task. Furthermore, it contributes to the static definition of the class-structure of inheritance systems.

6.2.2. Black box

In delegation systems, prototypes are implemented as black boxes, i.e. direct access to the internal structure of an object is not allowed. The objective of using black boxes is to abstract from the performance of the task, i.e. implementational issues are hidden from other objects. The outside world is provided an interface through which input to and output from the black box can be passed. The result is that changes made to the representation or implementation of a class will not affect any other object. Of course, this only holds when the functional description of the class remains the same. Furthermore, changes do not have to be propagated.

Consider again the example about the volume calculation of the table in Section 6.1.2. In this example, we claimed that the designer is not familiar with the several volume calculation methods and is not interested in them as long as the returned value is a precise approximation of the volume. To the designer, the several calculation methods and the selection of one of them are hidden in the black box. He/she has only access to the input and output interfaces of the black box (see Figure 3).

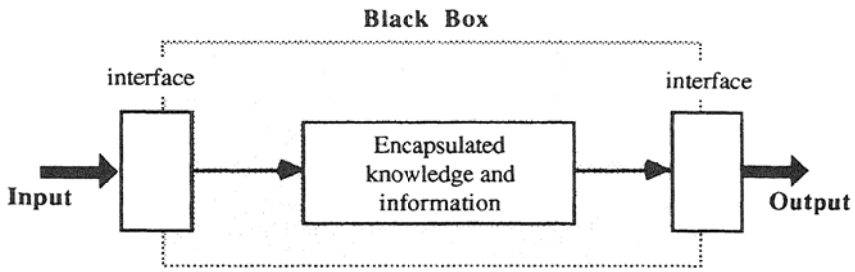

Figure 3 Black Box

6.2.3. Conclusions concerning glass box vs. black box

Again the field of application is determining for the choice between the glass or black box approach. However, it is clear that the black box approach is preferred over the glass box approach in environments as CAD, in which

- several persons contribute to the design,
- it is difficult to enumerate all essential features of a class and to define a class-structure beforehand, and
- knowledge of other fields of technology are involved during the design process.

6.3. One-way interface versus two-way interface

In the Sections 6.1 and 6.2, it has been shown how the relations among entities are used to pass a request for a method. However, it has not been mentioned how answers are returned and whether they are understood by the calling entity. In this section, we show that a simple one-way interface is sufficient for method returning while value returning requires a more complex two-way interface.

6.3.1. One-way interface

As was mentioned in Section 6.1.1 "Method returning", methods are passed from receiver to sender where they are evaluated against the sender's data-environment. In other words, conceptually the method should be at the sender's site. It is only implemeted as lazy message passing As, the methods are only procedural descriptions, i.e. in this case a bunch of characters, they do not have to be interpreted by the intermediate classes and only have to be passed. The sender of the message itself remains control for evaluation of the method. Due to the use of the generalization/specialization structure, the sender of the message is always in the subclass-chain of the receiver of the message. It is guaranteed that the sender's environment has the competence to evaluate the returned method. Therefore, the same path can be taken to send a message up the structure and to return the method down to the sender. A simple one-way interface is sufficient to perform the task.

6.3.2. Two-way interface

In delegation mechanisms, the objective is to pass messages, facts and control to other objects in order to delegate tasks. As, the task is performed inside a black box (see Section 6.2.2), only input and output are accessible for the user. The interface to pass a message to the black box is similar to the one-way-interface in Section 6.3.1. The interface to pass the result is far more complex. Firstly, to interprete the semantics of the returned value, semantic information need to be passed in addition to the result. Secondly, to avoid accumulation of transformation errors by interpreting the result by each object passed when the message was passed, it is necessary to return the result directly to the destination (e.g. the sender of the message). Thirdly, as a sender does not know which object will respond (see Section 6.4.2), it is a laborious task to set up all possible interfaces between objects. Therefore, this interface has to be set up at the time the result is passed to the destination.

6.4. One receiver versus many receivers

Another difference between inheritance and delegation mechanisms is the number of potential receivers which can respond to a request. This difference is the result of the fact that relationships in inheritance mechanisms are static and structured as a tree or lattice while in delegation mechanisms they are dynamic and structured as a network.

6.4.1. One receiver

In inheritance mechanisms, classes have only one ancestor from which they can inherit specific methods. Even if multiple inheritance is allowed and a class inherits several identically named methods, still only one class is allowed to respond is uniquely determined to the message (see Section 6.1.1 "Method returning"). Therefore, there is only one receiver.

6.4.2. One or more receivers

In delegation, an environment is created in which relations among prototypes can be changed easily. This is possible due to the prototype-oriented approach, the dynamic relationships and especially by the black box concept. However, due to the prototype-oriented approach the relationship between prototype and object is not maintained and thus, knowledge added to the prototype cannot be accessed by an existing object (see Section 5.1.2 "Prototype-oriented approach"). When an object cannot perform a certain task then in delegation the object can select one or more other objects who might do the job. Note, the word *might* because an object does not know how the delegated task is performed and whether the necessary information can be supplied. In this way, more than one candidate can be selected to send the message to, and more than one can respond. To manage all responses, a *multiple world mechanism* [Veerkamp89a] can be used to create and manage separate worlds for each response. In this paper, we will not further elaborate such mechanisms.

7. Other Features

7.1. Consistent versus temporarily inconsistent

Basis for the definition of consistent and temporarily inconsistent is that changes made to objects (e.g. adding attributes, methods, changing attribute values, etc.) are performed by atomic transactions. In other words, the parts of an object involved in an atomic transaction, cannot be accessed as long as the transaction is not finished successfully. We define consistent as that at any point in time between atomic transactions, the object is representing its actual state. We

define inconsistent as not consistent.

7.1.1. Consistent

Due to the fact that classes are implemented as glass boxes, subclasses can access inherited attributes via shortcuts without using the defined interface. The result of this is that attributes have to be consistent at all times. Therefore, changes to attributes have to be propagated immediately.

7.1.2. Temporarily inconsistent

In delegation mechanisms, temporary inconsistency is allowed because all access to attributes has to be performed through the interface, the black box concept. This interface can control and check all access to the attributes. Therefore, changes to attributes do not have to be propagated. A consequence of allowing temporary inconsistency is that *lazy evaluation* can be provided. Lazy evaluation is a delay of computation until the results of the computation are really needed. The advantage of this is shown in the next example. In a system, two possibilities are used to represent a line segment. These two are:

1) $L = [A,B]$, in which A and B are both endpoints in a two or three dimensional space.
2) $L = A + \lambda(\overline{B}-\overline{A}) \wedge \lambda = l$, in which A and B are again points, $(\overline{B}-\overline{A})$ is the direction of the line and l is the length of the line segment.

Let us say that a user draws a line on the screen and it is translated into both representations. Then, the user changes the length of the line by typing in a new value. In the first representation, the points A and B have to be recalculated and the system will ask what the new starting point of the line should be. In the second representation, only the variable l is modified. However, since it is not known whether A is still the starting point of the line, the line cannot be redrawn. Now, there are two possibilities. Either the user is asked immediately where A is placed on the line and the representation is updated or this question is postponed until someone needs to know what the real position of the line is.

In inheritance, it is not possible to postpone this evaluation because inconsistency of attributes is not allowed. The immediate evaluation is called *synchronous evaluation*. In delegation, inconsistency of attributes is allowed. When another object needs to know the exact coordinates of the line then it should pass its message via the interface. The interface will recognize that the line equation is not up to date, e.g. the equation was marked when l changed. Then, to make the equation valid again, it can ask the user where A is placed on the line.

7.1.3. Conclusions concerning consistent vs. temporarily inconsistent

In CAD, the design of an artifact is a process of stepwise refinement. In this process, the artifact is considered from different points of view. Furthermore, knowledge from other areas is often integrated in the decisions taken during the design process. Therefore, the user should be able to study the local effects of creation and modification of parts of the design without being faced with inconsistencies at a global level. Only at the time information is integrated in a more global level of the design, possible inconsistencies have to be solved. It is clear that in this situation delegation is preferred over inheritance.

8. Conclusions

In Table 1, we have summarized the main differences between inheritance and delegation as they were described in this paper.

	Features	
	Inheritance	**Delegation**
object-structuring features	• set-oriented approach • static relationships • based on generalization	• prototype-oriented approach • dynamic relationships • based on generalization and aggregation
method-sharing features	• control to perform a task remains with the original called object • method returning • simple one-way interface between objects • at most one receiver who reacts	• control and task is delegated to another prototype • value returning • complex two-way interface between objects • zero or more receivers who react
secondary features	• objects are implemented as glass boxes • consistent • synchronous evaluation	• prototypes are implemented as black boxes • temporarily inconsistent • lazy evaluation

Table 1 Features of inheritance and delegation mechanisms

Theoretically, the inheritance mechanism can be considered equal to the delegation mechanism when we look at both mechanisms at time *t* and neglect past and future. This means that for every inheritance structure an equivalent delegation structure can be made and vice versa [Stein87a]. However, for practical reasons one of the systems may be preferred when development and the use of the mechanisms are considered.

In CAD, the preference of delegation over inheritance is based upon two features of the design process. Firstly, a precise description of the artifact to be designed cannot be given beforehand. What is known, is a functional description with requirements for the designed object and perhaps a rough idea. The complete description of an artifact is created during the stepwise refinement process, in which subparts of the artifact are detailed and problems which come into existence during the process are solved. This implies that during the evolution of the artifact, its description is constantly changed. Secondly, in CAD the artifacts are mostly described by their decomposition. Although, such a decomposition can be simulated in an inheritance mechanism by adding special attributes and methods to a class, we believe that Part-Whole-relations have to be part of the structure itself. When the differences between inheritance and delegation mechanisms are considered then delegation has the following advantages over inheritance:

- no distinction is made between classes and instances, so, there is no distinction between expert and user-level,
- structuring the entities is based on a network in which the relationships between the entities are dynamic,
- the Part-Whole-relationship is supported by the delegation structure,

- prototypes are implemented as black boxes, and
- delegation allows temporary inconsistency.

When the literature on DBMSs based on the object-oriented approach is studied, then it can be seen that most of them support inheritance. The advantage of implementing an inheritance-based DBMS, is that DBMSs and inheritance make a distinction between an expert and user-level. After an expert in some field of technology has represented his knowledge in a hierarchical structure, this structure can be transformed into database schemata. The activities of the user of the inheritance mechanism correspond to the actions performed on the database schemata, e.g. storage and retrieval.

When we want to use database technology for supporting CAD systems then a data model which supports the requirements of designing (or at least some of them) has to be found. The requirements for the data model are:

- a non-fixed form of schemata or at least one which can be changed easily,
- support of the Is-Kind-Of and Part-Whole-links in the data model as two distinct kinds of relationships, and
- a data model which considers the heterogeneous character of data in CAD.

9. Future Work

In the current phase of the project, we are defining a theory for the delegation mechanism in CAD. The next step is to define a data model which is based on this theory and which can be implemented in an experimental Intelligent CAD system currently being implemented at the CWI [Veerkamp89b, Veth87a].

Acknowledgements

I would like to thank all the members of the IIICAD group at the CWI for their contribution to this paper.

References

[Akman87a]
Akman, V., Ten Hagen, P.J.W., and Tomiyama, T., "Design as a Formal, Knowledge Engineered Activity," Technical report No. CS-R8744, p. 7, Centre for Mathematics and Computer Science, September 1987.

[Arbab89a]
Arbab, F., "Examples of Geometric Reasoning in Oar," in *Intelligent CAD Systems II*, ed. Akman, V., Ten Hagen, P.J.W. and Veerkamp, P.J., pp. 32-57, Springer-Verlag, Berlin, 1989.

[Blake87a]
Blake, E. and Cook, A., "On Including Part Hierarchies in Object-Oriented Languages, with an implementation in Smalltalk," in *ECOOP '87*, ed. Goos, G. and Hartmans, J., pp. 41-50, Springer-Verlag; Lecture Notes in Computer Science, Berlin, 1987.

[Brodie84a]
Brodie, M.L. and Ridjanovic, D., "On the design and specification of database transactions," in *On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases, and Programming Languages*, ed. Brodie, M.L., Mylopolous, J., and Schmidt, J.W., pp. 277-327, Springer-Verlag, New York, 1984.

[Davis85a]
Davis, R., Buchanan, B., and Shortliffe, E., "Production Rules as a Representation for a Knowledge-Based Consultation Program," in *Readings In Knowledge Representation*, ed. Brachman, R.J. and Levesque, H.J., pp. 371-388, Morgan Kaufmann Publishers, Inc., Los Altos, California, 1985.

[Goldberg83a]
Goldberg, A. and Robson, D., *Smalltalk-80; The Language and its Implementation,* Addison-Wesley Publishing Company, Massachusetts, May 1983.

[Kleer85a]
De Kleer, J., Doyle, J., Steele, G.L. Jr., and Sussman, G.J., "AMORD: Explicit Control of Reasoning," in *Readings In Knowledge Representation*, ed. Brachman, R.J. and Levesque, H.J., pp. 345-355, Morgan Kaufmann Publishers, Inc., Los Altos, California, 1985.

[Lieberman86a]
Lieberman, H., "Using Prototypical Objects to Implement Shared Behaviour in Object Oriented Systems," in *Proceedings ACM Conference On Object-Oriented Programming Systems, Languages and Applications*, ed. Meyrowitz, N., pp. 214-223, ACM Press, New York, September 1986.

[Lieberman86b]
Lieberman, H., "Delegation and Inheritance: Two Mechanisms for Sharing Knowledge in Object-Oriented Systems," *3eme Journées d'Etudes Languages Orientés Objet*, no. 48, pp. 79-89, AFCET, Paris, France, September 1986.

[Reiter85a]
Reiter, R., "On reasoning by default," in *Readings In Knowledge Representation*, ed. Brachman, R.J. and Levesque, H.J., pp. 401-410, Morgan Kaufmann Publishers, Inc., Los Altos, California, 1985.

[Stein87a]
Stein, L.A., "Delegation is Inheritance," in *Proceedings ACM Conference On Object-Oriented Programming Systems, Languages and Applications*, ed. Meyrowitz, N., pp. 138-146, ACM Press, New York, October 4-8, 1987.

[Veerkamp89a]
Veerkamp, P.J., "Multiple Worlds in an Intelligent CAD system," in *Intelligent CAD, I*, ed. Yoshikawa, H. and Gossard, D., North Holland, Amsterdam, The Netherlands, 1989.

[Veerkamp89b]
Veerkamp, P.J., Akman, V., Bernus, P., and Ten Hagen, P.J.W., "IDDL: A Language for Intelligent Interactive Integrated CAD Systems," in *Intelligent CAD Systems II*, ed. Akman, V., Ten Hagen, P.J.W. and Veerkamp, P.J., pp. 58-74, Springer-Verlag, Berlin, 1989.

[Veth87a]
Bart Veth, "An Integrated Data Description Language for Coding Design Knowledge," in *Intelligent CAD Systems I - Theoretical and Methodological Aspects*, ed. Ten Hagen, P.J.W. and Tomiyama, T., pp. 295-313, Springer-Verlag, Berlin, 1987.