Object-oriented data modeling

Donald W Baltz
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
Baltz, Donald W, "Object-oriented data modeling" (1989). UNLV Retrospective Theses & Dissertations. 64.
https://digitalscholarship.unlv.edu/rtds/64

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Object-oriented data modeling

Baltz, Donald W., M.S.
University of Nevada, Las Vegas, 1989
OBJECT-ORIENTED DATA MODELING

by

Donald W. Baltz

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

Department of Computer Science and Electrical Engineering
University of Nevada, Las Vegas
December, 1989
The thesis of Donald W. Baltz for the degree of Master of Science in Computer Science is approved.

Chairperson, Thomas Nartker, Ph.D.

Examiner Committee Member, Kazem Taghva, Ph.D.

Examiner Committee Member, Ajoy Datta, Ph.D.

Graduate Faculty Representative, Ebrahim Salehi, Ph.D.

Graduate Dean, Ronald W. Smith, Ph.D.

University of Nevada, Las Vegas
November, 1989
Abstract

The object-oriented paradigm models the local behavior, and to a lesser extent, the structure of an application. Semantic data models describe structure and semantics. This paper unifies the behavioral aspects of the object-oriented paradigm with the structural and semantic aspects of semantic data models. The modeling approach contains expressive abstractions to model static and derived data, semantics, and behavior. The abstractions keep the data model closer to the problem domain than the entity-relationship or relational models, and in a form that can readily be translated into a relational (or other) implementation. The paper makes six principal contributions. First, a comprehensive set of data structuring abstractions originating from research into semantic data models, (AI) knowledge representation, CAD/CAM applications, and object-oriented programming languages are described. Second, the abstractions are compared to the entity-relationship and relational models. Third, semantic information inherent in the functional representation of the structuring abstractions is identified. Fourth, a set of behavioral abstractions originating from semantic data models, CAD/CAM applications, and object-oriented programming languages are described. Fifth, an algorithm that describes the dynamics between mathematically derived attributes of cooperating objects is presented. The functional dependencies between derived attributes themselves are modeled using a transform-centered approach. Sixth, some weaknesses of object-oriented programming languages with respect to supporting the structuring abstractions, semantic constructs, and behavioral abstractions are identified.
Table of Contents

1. Introduction 1
2. Definitions 6
   2.1 Entities 6
   2.2 Specialized Entities 8
   2.3 Relationships 9
   2.4 Objects 11
   2.5 Problem Definition 12
3. Data Structuring Abstractions 14
   3.1 Objects 15
   3.2 Relationships 19
   3.3 Attributes 25
   3.4 Extended Forms of Specialization 36
   3.5 Extended Forms of Constructed Objects 43
   3.6 Versions 49
   3.7 Summary 51
4. Integrity Rules 52
   4.1 Extracting Semantics From Structure 53
   4.2 Rules for Maintaining Structural Integrity 54
5. Operators: Adding Behavior to Structure 61
   5.1 Identifying Behavior 61
   5.2 Operator Inheritance 63
   5.3 Semantic Operators 65
   5.4 Propagation Of Operators 68
   5.5 Object Dynamics 71
   5.6 Classes 82
6. Object-Oriented Languages 83
7. Conclusions 92
Bibliography 97
Acknowledgements

I would like to acknowledge three factors that contributed heavily to my achieving this long sought after goal; the academic environment, my professional environment, and my personal home life. The computer science staff at UNLV is outstanding. I would particularly like to thank Dr. Thomas Nartker, who charted a program of study in line with my professional objectives, and Dr. Kazem Taghva, for the many hours guiding me on a topic of my choosing. In my professional environment, I would like to thank Dr. Jerry Blair for fighting the bureaucracy (and for looking the other way). Finally, my wife, Pamela, for her continued support.
Chapter 1

Introduction

A data model is an abstraction that provides a conceptual representation of relevant data of a system that does not contain the details of how the data is stored or manipulated. A data model uses concepts such as objects, entities, attributes, relationships, constraints, and operators to describe the problem domain. These concepts are closer to the problem domain (and consequently easier for the users to understand) than the design details of the software or the physical storage.

The relational model, introduced by Codd in 1970 provided a sound theoretical basis for database design because of it's foundation in first order predicate logic [17]. The principle motivation for the research effort that resulted in the model was that of data independence, i.e., there be a sharp and clear boundary between the logical data representation and the physical implementation. Other motivations were structural simplicity to aid communication between the software developers and the users of the system, and a high level language for set-processing operations [19]. The relational model contained:

A collection of data structure types,
A collection of operators to retrieve, derive, or modify data from the structures;
A collection of integrity rules.
It is still accepted that a data model should contain a structural component, a data manipulation component, and an integrity-specification component.

The network model was first presented in the CODASYL Data Base Task Group 1971 report [24]. The principal motivation for the network model was to achieve a consistent approach for schema design. While the network model is used in many commercial DBMS implementations, it lacks the theoretical basis of the relational model.

The third of what are considered the *classical* data models is the hierarchical model. There is no original document that describes the hierarchical model. The model evolved pragmatically from several early computer information management systems that were developed using hierarchical storage structures to manage hierarchically organized data. It, too, lacks the theoretical basis of the relational model.

Experience with the three classical data models has shown that they do not satisfy the role of being a conceptual representation of the relevant data of the systems they describe [42]. The fundamental problem with classical data models is the primitives they provide (strings, integers, real numbers) for describing data are more appropriate for modeling the physical storage of the data and for printing reports than for modeling the concepts underlying them [11]. An information system built on these low-level primitives is hard to develop and difficult for the user to conceptualize.

Semantic data models [36,41,48,60] were initially introduced primarily for conceptual schema design purposes overlaying one of the classical models, rather than as replacements for them. Some of the modeling approaches use a graphical formalism, the Entity-Relationship (ER) model being the most widely known1 [15]. Other data models use a language formalism [5,32,56,66]. Others use a mathematical formalism [3,40] augmented with a graphical formalism. Codd himself defined a new model, RM/T, that extended the relational model to capture more semantics and structure [18]. The fact that

---

1Chen actually showed how to map the ER model into the three classical models.
Database concepts have evolved in recent years is best articulated by Ullman [77] where he states that what he had thought of as a “database system” in his earlier book [76] “formed but one (important) point in a spectrum of systems that share a capability to manage large amounts of data efficiently”.

Two principal reasons are cited for layering semantic models over the relational model. First, the relational model fails to capture the semantics of a system. Second, attempts to extend the relational model from enterprise modeling into other applications (such as office environments, scientific and engineering environments, CAD/CAM and design applications, and multimedia applications) resulted in a model that was far from the users conceptual view of the system. Recommendations for using two data models; a higher level conceptual model which is closer to the problem domain which can then be translated to one of the classical (or other) models continue to appear in the literature [28].

According to Michael Stonebraker, “Semantic data models are now ‘ passé ’ ” [72]. Their fundamental contribution, however, was the definition of high level abstractions for representing the structural aspects and integrity constraints of a system. One of the abstractions, that of modeling the important things (entities) of a system and their relationships, is similar to concepts of object-oriented programming. There are other similarities too. Concepts of object-oriented programming such as objects, classes, class hierarchies, and inheritance have similar counterparts in semantic data models. In fact, the differences between semantic data models and object-oriented data models is vague. Unland has proposed that object-oriented data models are “a specialization of semantic data models in that object-oriented data models additionally provide the concept of abstract data type” [78].
Hypothesis

The object-oriented paradigm models the local behavior, and to a lesser extent, the structure of an application. Semantic data models describe structure and semantics. The purpose of this thesis is to unify the behavioral aspects of the object-oriented paradigm with the structural and semantic aspects of semantic data models. A secondary goal is to show there is a theoretical foundation for data modeling concepts. Four premisses are developed:

1. The abstractions of semantic data models and the object-oriented paradigm are compatible with the relational model.

2. The relational model is an implementation model, not a specification model.

3. Semantic information is a beneficial side effect of a stronger theoretical foundation.

4. The abstractions of an object-oriented data model is a necessary first step to the bottom-up approach of object-oriented programming languages.

The fact that the relational model is examined with respect to current software technology should not be surprising. The relational model was defined in 1970. The major programming languages of the day were Cobol and Fortran. The most significant advance in software technology in the 1970’s was the concept of abstract data types. The most significant advance in software technology in the 1980’s was the development of object-oriented programming languages that support these concepts. The relational model does not even support records, aggravating what Dittrich calls the impedance mismatch between programming languages and data base technology [26]. Some consolidation of the theoretically sound relational model with these advances in software technology is inevitable.
Organization

The organization of the thesis is:

Chapter 2 provides working definitions for basic terms, and defines the problem.

Chapter 3 describes high level abstractions for representing the structural aspects of a system, and a graphical notation for representing them. The abstractions are compared with respect to relational and other data models.

Chapter 4 shows integrity constraints to be inherent in the representation of the abstractions and are independent of the problem domain.

Chapter 5 adds operators (behavior) to the structuring abstractions. Also presented is an example of how the object-oriented approach can be used to manage mathematically derived attributes.

Chapter 6 examines how object-oriented programming languages support the abstractions, and reveals some limitations of these languages.

The conclusions are presented in Chapter 7.
Chapter 2

Definitions

The purpose of this section is: 1) to give working definitions to the terms entity, entity set, attribute, relationship, specialized entities, and inheritance; 2) to provide a precise meaning to the term object; and 3) to define the problem being addressed.

2.1 Entities

An entity is something that has real existence (either physically or conceptually) in the system being modeled, is distinguishable (it can be differentiated from other entities), and is important to that system, i.e., there is information concerning the entity that is to persist, usually in a database\(^1\).

Entities have properties, called attributes, that describe it. Attributes are measurable or identifiable characteristics such as name, address, birthdate, height, weight, typing speed. Attributes associate a value with each entity from a domain of values for that attribute. The domain of values for attributes are sets of integers, real numbers, or character strings. The domain of values is sometimes called the type of the attribute, in the same sense as variables in a programming language have a type.

---

\(^1\)A database is a repository for storing persistent information. A database could be a commercial data base management system (DBMS), a file shared by several programs, main memory variables shared by otherwise independently executing programs or processes, or some combination of these.
Only attributes that are meaningful or relevant to the system being modeled are associated with entities. Height, for example, which is certainly a measurable characteristic of an employee would not be included as an attribute in a company database unless it is a requirement of the system to include it.

Entities which have the same attributes form an entity set. Examples are:

- All companies.
- All departments in a company.
- All employees.

An entity is one instantiation of an entity set, e.g., one specific employee (John Smith).

Entity sets are said to have a type which is the aggregated type of its attributes. To use a metaphor, the type of an entity set is like a template. All entities in the entity set fit the same template. So as to distinguish when the term entity is referring to an entity set as a template, and when it is referring to a single entity, the term entity type is sometimes used.

An entity set has an attribute (or set of attributes) whose value(s) are distinct for each entity in the set. This attribute is called the key\(^1\). The value of the key attribute uniquely identifies (distinguishes) each entity in the entity set. Social Security Number might be the key attribute for the EMPLOYEE entity set. Name could not be a key attribute because persons with the same name would not be distinguishable.

Figure 2.1 shows the EMPLOYEE entity set with attributes Name, Address, Birthdate, Social Security Number and Job Type using the drawing notation of Entity-Relationship Approach (ER). Entity set names are upper case and enclosed in a box, attribute names are capitalized and enclosed in an oval. Key attribute(s) are underlined.

---

\(^1\)It should be pointed out that a key identifier is not necessary in many semantic and object-oriented implementations, where distinct object identity is maintained (even when all attribute values are identical) through the use of surrogate identifiers.
The EMPLOYEE entity set represented in Figure 2.1 is sometimes represented as a table named EMPLOYEE, and with columns labeled Name, Address, Birthday, Social Security Number, and Job Type. The notation for representing such a table is (the key attribute is underlined):

EMPLOYEE (Name, Address, Birthday, Social Security Number, Job Type)

2.2 Specialized Entities

Entity sets can be specialized. SECRETARY, for example, is a specialization of EMPLOYEE. Fundamentally, a secretary is an employee.

Attributes of the fundamental entity set are not duplicated in the specialized entity set. The attributes for a specific secretary are contained in two entity sets, one representing the secretary as an employee, and one representing the secretary as a secretary. The
specialized entity set is said to inherit the attributes of the generalized entity set. Figure 2.2 shows SECRETARY as having only the Typing Speed attribute, SECRETARY in fact inherits Name, Address, Birthdate, Social Security Number, and Job Type from EMPLOYEE, because every secretary is fundamentally an employee.

The purpose of specialization is to model the relevant information of the system, not to build an exact replica of the system. For example, employees within a company might be categorized as ENGINEER, TECHNICIAN, and SECRETARY. If there is information that is to persist for secretaries that is not to persist for employees in general, and there are no additional attributes to persist for engineers and technicians, then EMPLOYEE would be specialized to include only SECRETARY. The empty entity sets ENGINEER and TECHNICIAN need not be modeled, although sometimes it helps to represent them in the ER diagram because they are recognizable landmarks from the problem domain.

Specialized entities are said to participate in the ISA relationship. (A secretary ISA employee.) Specialized entity sets derive their type definition from their more general entity sets.

2.3 Relationships

Entity sets may be related to one another in ways other than specialization relationships. Figure 2.3 is an ER diagram (without attributes) that represents a company that is composed of several departments, where each department is assigned several employees, some of the employees manage departments, one of the employees is the president of the company, and the employees are assigned to projects. Interactions between entity sets are called relationships. Relationships represent information about the associations among

\[1\text{The goal is to avoid defining attributes that do not apply to each entity in the set. Non-key attributes that apply to each entity, but for which a value is not presently known are acceptable [18].}\]
entities, rather than information about entities in isolation. They are represented on the ER diagram by diamonds enclosing the name of the relationship, which is capitalized.

![ER Diagram for a Company](image)

Figure 2.3. An ER Diagram For A COMPANY

Relationships are said to have a one-to-one, one-to-many, and many-to-many *cardinality*, which is represented by annotating the line connecting the entity sets participating in the relationship with $1$ (for one) or $N$ (for many). A *one-to-one* relationship means that there is one entity in each entity set that satisfies the relationship. The PRESIDENT is an example of a one-to-one relationship between COMPANY and EMPLOYEE, meaning that one employee is the president of the company. The set of employees assigned to one department is an example of a *one-to-many* relationship between DEPARTMENT and EMPLOYEE through the DEPARTMENT ASSIGNMENT relationship, meaning that a department is assigned many employees. PROJECT ASSIGNMENT is an example of a *many-to-many* relationship between EMPLOYEE and PROJECT, meaning that one employee is assigned to one or more projects, and one project is assigned many employees.
2.4 Objects

A common characteristic of all data models conceptually higher than the classical models is the identification of entity sets that are independent of each other. Some examples follow.

Codd classified entities as characteristic, associative, inner kernel, and non-inner kernel according to their role in the system being modeled in RM/T, a semantic extension of the relational model [18]. Without describing the details, all characteristic, associative, and non-inner kernel entities are subordinate to, and existence dependent\(^1\) on, a superior entity; whereas inner kernel entities are defined independent of all other entity types. "The (inner) kernel entities in a given database are what that database is really all about" [23].

The entity-relationship (ER) approach [15, 28] distinguish between strong entity sets and weak entity sets, the distinction being existence dependent relationships. A drawing notation is used to emphasize the distinction.

Hull [41] distinguishes between "fundamental object types" called abstract data types, and other entity types (i.e., specialized entities and constructed object types) that derive their type definition from abstract data types or from other entity types. A drawing notation is also used to emphasize the distinction.

ODE (Object Database and Environment) [4] distinguishes between base classes and derived classes, the distinction being one of type derivation.

This thesis synthesizes the identification of special entities along with a common trend to use the terms object and entity interchangeably, to provide a precise meaning to the term object. An object is an entity that is not existence dependent on any other entity. An object set is an entity set that contains objects. Just as entity sets have a type, so too, object

---
\(^1\)Existence dependent is used by Chen [15], Tsichritzis [75], Date [23], Batory [7], and Elmasri and Navathe [28] (among others). Hull [41] and Ullman [77] uses the term existence constraint. Existence dependency is something like an inclusion dependency of the relational model but at a higher level of abstraction. It states, for example, the subtype entity secretary cannot exist unless the superior entity employee also exists. The use of foreign keys in a relational implementation results in an inclusion dependency. An inclusion dependency is a term specific to a relational implementation of an existence dependency.
sets are assigned a type, called an object type. The type of an object set is not derived from the type of any other object set. The term entity, then, refers to either an object entity or a specialized entity when the distinction is not important.

This definition of object is not too different from its use in object-oriented programming languages, where objects encapsulate type and behavior. A data model describes “a collection of data structure types” (i.e., type) and a “collection of operators” (i.e., behavior). The distinction between entity sets that derive their type definition from those that do not will prove useful when identifying the integrity rules (semantics) of operators, something that is severely lacking in current object-oriented approaches.

2.5 Problem Definition

Consider the ER diagram of a company:

![ER Diagram of a Company](image)

Figure 2.4. ER diagram of a company:
Notationally, there is no distinction between entities. Beyond that, the following questions cannot be answered:

Must all employees be assigned to departments?

Does a department exist if it has no assigned employees?

Can an employee be both an engineer and a secretary?

Are there other job classifications?

What entities are affected when an employee is fired or reclassified?

Can some attribute values that may not be known be left undefined?

These issues, and others like it, are semantic issues. The answers to the questions are the “collection of integrity rules” that a data model should contain. They cannot be answered by a simple ER diagram, nor by the relational model. These models show the record structure of the application. They are void of semantic content. Furthermore, the operations defined for record oriented models are tuple creates, deletes, and updates; operations that do not coincide with the operations of the problem domain such as fire an employee, reclassify an employee, etc. Knowledge of the detailed structure of the physical implementation is a necessary prerequisite to applying sequences of creates, deletes, and updates that map the operations of the problem domain into database operations.

This thesis addresses the modeling of structure, operations, and integrity rules. The subject matter is one of software engineering, i.e., applying theory to build a model of the problem domain that can be translated into a computer program in a straight-forward manner. There are four requirements for such a model: 1) The model should accurately represent the problem domain; 2) The constructs of the model should be compatible with programming language technology; 3) The model should present an external view of the system to be implemented; and 4) The model should be void of implementation bias.
Chapter 3

Data Structuring Abstractions

An abstraction of a system is a description of that system with certain details omitted. The purpose of an abstraction is to allow the reader to pay attention to the relevant details of the system and to allow them to ignore other details. A hierarchy of abstractions allows relevant details to be introduced in a controlled manner, making the system intellectually manageable [69].

One of the results of research into software engineering, semantic data models, (AI) knowledge representation, abstraction based programming languages, and attempts to extend database technology into design environments and data-intensive programming in the large applications, is a set of high-level data structuring abstractions. The data structuring abstractions that were identified are:

1. The ability to model objects, their attributes, and the relationships between them in a direct manner as functions and (mathematical) relations.

2. The ability to model aggregate attributes.

3. The ability to model extended forms of specialization.

4. The ability to build objects out of other objects and to generate object versions.

These abstractions are described, a graphical notation is defined for representing them, and they are compared to the entity-relationship and relational models. The graphical notation is
taken principally from the works of Richard Hull [2,3,14,40,41], augmented somewhat by notation from the Extended Entity Relationship model (EER) as described by Elmasri and Navathe [28], and extended where neither supported a particular abstraction.

3.1 Objects

Semantic data models introduced the abstraction of modeling objects, their attributes, and the relationships between them in a direct manner as functions and (mathematical) relations. Additionally, semantic data models used the concept that objects in the model directly correspond to entities in the world being modeled. These notions contrasted with the less direct abstractions of the relational approach that modeled associative, characteristic, kernel, and inner kernel entities (most of which are artifacts of normalization) as tuples and relations, and modeled relationships between them using foreign keys or associative relations.

3.1.1 Representing Objects

Semantic data models distinguish between constructed objects and non-constructed. Non-constructed objects are sometimes called atomic objects, a term that has different meanings in the literature and is not used here. Constructed objects are constructed out of other objects; non-constructed objects are not.

Semantic data models provide two construction operators, the set-constructor and the aggregate-constructor, that are used to form constructed objects from non-constructed (and constructed) objects. The set-constructor forms finite sets from one type using the finite power set operator; the aggregate-constructor forms new objects out of existing objects using the Cartesian product operator.
Figure 3.1 shows the notation for representing constructed and non-constructed objects. An object is represented as a triangle enclosing its name.

![Diagram of set-constructor and aggregate-constructor](image)

Figure 3.1. Notation for set-constructor and aggregate-constructor.

The set-constructor (⊗) forms a set of EMPLOYEES (Figure 3.1(a)). The aggregate-constructor (⊗) (Figure 3.1(b)) forms DEPARTMENT ASSIGNMENT, constructed from the non-constructed objects DEPARTMENT and EMPLOYEE. Objects associated with an aggregate-constructor as a root is an ordered pair that is a subset of the Cartesian product of the domains of the underlying nodes. The ordered pair is represented by the dashed lines.

3.1.2 Representing Attributes

The assignment of attributes to an object is based on the formal mathematical concept of a function from a domain to a range, where the domain is the entity set and the range is the domain of values for each attribute associated with the entity set.

Figure 3.2 shows the functional mapping of EMPLOYEE to its attributes, where EMPLOYEE and String are sets, and the directed edge represents a function. Function names annotate the directed edges from EMPLOYEE to its attributes. The edge is terminated with a single arrowhead at the range, representing the single-valued nature of the function. Attribute domains are enclosed in an ellipse.
The Name function, for example, maps EMPLOYEE to the domain of string values for the Name attribute. Similarly, the Address, Birthdate, Job Type and Social Security Number functions map EMPLOYEE to their appropriate domains.

Figure 3.3 shows the functional notation that is used here. Function names are enclosed in an ellipse. The annotations Total and 1:1 on the directed edge describe the nature of the function. Edges without annotation have no restrictions, i.e., they can represent partial functions and attribute values can be null.

The attribute domains are not represented in the notation. They are defined in a separate document. The description for the domain of values for, say, Name, Birthdate, Social Security Number, and Job Type might read:

Name: Twenty or less ascii characters representing the names of employees. The active domain of Name(EMPLOYEE) represents the names of actual employees.
Birthdate: Eleven ascii characters expressed as \texttt{dd-mmm-yyyy}, where \texttt{dd} are ascii integers between 1 and 31 representing the day of the month, \textit{mmm} are the months of the year expressed as JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC, and \texttt{yyyy} are ascii integers representing the year.

Social Security Number: Eleven ascii values represented as XXX-XX-XXXX, where each X is an integer number. The values are restricted to represent social security numbers of actual employees.

Job Type: Fifteen or less ascii characters that may be "engineer", "technician", "receptionist", "accountant", or "secretary".

These domain descriptions are more precise than \textit{String}, e.g., the domain description states that function \textit{Name(EMPLOYEE)} maps to a separate and distinct subset of \textit{String} than, say, the function \textit{Job Type(EMPLOYEE)}. Notice the domain for \textit{Birthdate} is not restricted to dates for which actual employees were born, whereas the domain for \textit{Name} and \textit{Social Security Number} are restricted.

Key attributes are not represented in Figure 3.3. This makes the model suitable for object-oriented, relational, and network implementations by removing implementation bias. Attribute mappings that are \textit{Total, 1:1} functions are candidates for key attributes in a relational implementation. Thus, \textit{Social Security Number}, which is \textit{Total, 1:1} (representing the fact that each employee has a unique Social Security Number), would be a candidate for a key attribute. \textit{Name}, which is not \textit{1:1} (representing the possibility that several employees may have the same name) is not a candidate.

### 3.1.3 Representing Specialized Entities

Object entities can be specialized by ISA relationships. Notationally, specialized entities are represented as a circle node (so as to distinguish them from object entities) with an edge (drawn with a lightly shaded fat arrow) to the more general entity,
which may be an object or another specialized entity. The functional mapping from the more general to the specialized entity is always partial, 1:1. The mapping from specialized to more general entity is always total, 1:1. A specialized entity derives its type from the more general entities. The attributes associated with specialized entities are represented in the same manner as attributes of object entities. Without any indication to the contrary specialized entities inherit the attributes all its more general entities.

3.2 Relationships

Objects, because they are entities, can be related to one another by relationships\(^1\) other than ISA relationships. These relationships have a cardinality identified as one-to-one, one-to-many, and many-to-many.

\(^1\)Technically, the mathematical term *relation* should be used. The term *relationship* originates from the Entity-Relationship model, and is used here because of its common usage to describe relations between entity sets.
3.2.1 One-To-One Relationships

The representation of one-to-one relationships is also based on the formal mathematical concept of a function from a domain to a range, where the domain is one entity set, and the range is the related entity set. Figure 3.5 shows the notation for representing the one-to-one relationship DEPARTMENT MANAGER between EMPLOYEE and DEPARTMENT, and the inverse relationship. Mathematically, this represents the fact that if set $A$ is related to set $B$, expressed as $A \mathcal{R}_B$, then it is also true that $B \mathcal{R}_A$. The single arrowhead on each function represents the one-to-one cardinality of the relationship. The relationship is a Partial, 1-1 function, representing the fact that some employees do not manage departments and those that do only manage one department. The inverse relationship is Total, 1-1 representing the fact that every department has only one manager, who is an employee. Had the inverse relationship been Partial, 1-1, then the fact represented would be that some departments may not have a manager. Both the relationship and the inverse relationship

Figure 3.5. Representing one-to-one relationships as functions
provide semantic information. Unless specified otherwise, either the relationship, or its inverse, but not both, are part of the actual implementation.

3.2.2 One-To-Many and Many-To-Many Relationships

One-to-many relationships are represented using either the functional notation or the aggregate-constructor ($\otimes$). Figure 3.6 shows all combinations of total vs partial functions for the one-to-many relationship between EMPLOYEE and DEPARTMENT.

![Figure 3.6](image)

Four representations of the DEPARTMENT ASSIGNMENT relationship $\mathcal{R}$ and its inverse $\mathcal{R}^{-1}$

Figure 3.6. Representing one-to-many relationship using functions

Again, both the relationship and the inverse relationship are represented. The one cardinality is represented by a single arrowhead on the edge from EMPLOYEE to DEPARTMENT, which represents the fact that an employee is assigned to only one department. The many cardinality is represented by a double arrowhead on the edge from

---

1In this example, the total relationship would be implemented, rather than the partial relationship, so as to avoid reference attributes that would not apply to each object in the set.
DEPARTMENT to EMPLOYEE, which represents the fact that many employees are assigned to a department.

Figure 3.6 illustrates the semantic information functional modeling provides. Figure 3.6(a) models the fact that all employees are assigned to a department, and all departments have employees assigned to them. Figure 3.6(b) models the fact that some employees may not be assigned to a department, but all departments have employees assigned to them. Figure 3.6(c) models the fact that all employees are assigned to a department, but some departments may have no employees. Figure 3.6(d) models the fact that some employees may not be assigned to a department, and some departments may have no employees. The Create/Delete/Update semantics of each is different, something that will be addressed in more depth in Chapter 4.

One-to-many relationships can alternatively be represented using the aggregate-constructor (₫ ). Figure 3.7 shows the alternate representation of Figure 3.6(c). The aggregate-constructor elevates the functional relationship to an entity (called an associative entity). This permits attributes (e.g., Date Assigned To Department) to be assigned to the relationship that cannot be represented with the functional representation.
The similar functional representation for attributes and relationships extends beyond the graphical notation. Relationship values are accessed using the same functional notation, e.g., *Department Assignment*(EMPLOYEE) maps an employee to a department. If the relationship has attributes as in Figure 3.7, then the mapping is also to *Date Assigned To Department*.

Many-to-many relationships are represented using the aggregate-constructor (\( \otimes \)). Figure 3.8 shows the many-to-many PROJECT ASSIGNMENT relationship between EMPLOYEE and PROJECT.

![Figure 3.8. Representing many-to-many relationships.](image)

### 3.2.3 Comparison With Other Models

Figure 3.9 shows the functional representation with respect to five other representations. The entity-relationship diagram is from Chen [15]. It is semantically void. The relational schema representation is from Ullman [77]. The single arrowhead is placed at the *one* side of the *one-to-many* relationship, and no arrowhead is placed on the *many* side of the relationship. This is a conceptually accurate model for a relational implementation in so far as EMPLOYEE contains a foreign key attribute that is the value of the identifier of the related tuple in DEPARTMENT. The representation is also semantically void. The Network representation is from Date [23]. The arrow points (identifies a link named DeptEmp) between the owner DBTG set (DEPARTMENT) and the member DBTG set (EMPLOYEE). This is a conceptually accurate model for a network implementation but is
also semantically void. The relational table is also semantically void, only more so. There is nothing in the representation to even suggest that the two tables are related, and the fact that \textit{Dept No} is a foreign key is not represented (the worst kind of information hiding).

The EER representation has almost the same information as the functional representation. (The double line represents a total function, the single line represents a partial function, and 1 and \textit{N} represent the cardinality of the relationship.) The nature of the \textit{total vs partial} mappings adds semantic richness. Both are void of implementation bias. The functional representation, however, represents the \textit{1:1} nature of the relationship, and also identifies attributes that cannot have a null value by carrying the \textit{total vs partial} mappings through to

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.9.png}
\caption{Six modeling approaches.}
\end{figure}
the attributes. The functional representation also carries through to the notation for accessing attributes, i.e., both attribute and relationship values are accessed using one functional notation. The EER model does not have any of these capabilities.

3.3. Attributes

Informally, attributes are the measurable or identifiable characteristics of an entity. Several kinds of attributes can be identified. They are:

- Simple attributes.
- Composite attributes.
- Multivalued attributes.
- Derived attributes.
- Acquired attributes.
- Class attributes

3.3.1 Simple Attributes

Formally, a simple attribute, $a$, is a function between an entity, $E$, and a domain of values, $V$, composed of integers, real numbers, or character strings that represent a measurable or identifiable characteristic of the entity.

$$a: E \rightarrow V$$

Simple attributes are sometimes called atomic attributes or printable attributes. They are also sometimes called *atomic objects* in object-oriented programming languages.
3.3.2 Composite Attributes

The aggregation\(^1\) of attributes to form higher-level attributes is called Cartesian aggregation\(^2\). Attributes so constructed are called composite attributes. Formally, the domain of values, \(V\), for the attribute function, \(a\), from the entity, \(E\), is the cross product of several domain sets \(V_1, \ldots, V_n\).

\[ a: E \rightarrow V_1 \times V_2 \times V_3 \times \ldots \times V_n \]

A composite attribute is defined recursively as an attribute composed of simple attributes and composite attributes.

Figure 3.10 shows Cartesian aggregation. The composite attribute \(Name\) is composed of the simple attributes \(First Name, MI, and Last Name\); the composite attribute \(Address\) is composed of the composite attribute \(Street Address\), and the simple attributes

\(^1\)Chen [15] first identified the concept of attributes being able to map onto what he called multiple value sets. He did not name the concept. The terminology used here is extended from Hull [41].

\(^2\)The term, Cartesian, originates from C. A. R. Hoare's Cartesian product data structure [22]. A Cartesian product data structure \(date\), for example, is an ordered 3-tuple from the cross product of the primitive (or previously defined) data structures of \(day, month, and year\). The valid values for \(date\) is a proper subset of the cross product. A Pascal record is a language implementation of Hoare's Cartesian product data structure.
*City, State, and Zip*; and the composite attribute *Street Address* is composed of the simple attributes *Number, Street, and Apartment Number*.

There are two reasons for using Cartesian aggregation. First, the attributes form a cluster that are related by a relationship not shared by other attributes. Second, it is useful to refer to a cluster as a unit [28], e.g., “6” “feet”, “140” “miles”. Codd [18] also identified clusters of information that constitute what he called *meaningful units*.

Figure 3.11 shows EMPLOYEE with Cartesian aggregation represented by placing an \( \times \) through the ellipse enclosing the *Name* and *Address* attributes, indicating these attributes are formed by Cartesian aggregation but their constituent attributes are omitted from the drawing. This notation allows relevant details to be introduced to the drawing in a controlled manner, making the system intellectually manageable.

![Figure 3.11. Representing Cartesian aggregation](image)

Kim [46], and Elmasri and Navathe [28] augmented the relational table notation to represent Cartesian attributes using parenthesis and indentation in the following manner:

```plaintext
EMPLOYEE ( Name
  (first name,
   MI,
   Last name),
 Address
  (Street Address
   (Number,
    Street,
    Apartment Number),
 City,
 State,
 Zip),
 Birthdate,
 Social Security Number,
 Job Type).
```
3.3.3 Multivalued Attributes

Informally, multivalued attributes are attributes that have multiple values from a domain of values. Formally, the domain of values, $V$, for the attribute function, $a$, from the entity, $E$, is the powerset $P(V)$.

$$a: E \rightarrow P(V)$$

The domain of values for multivalued attributes can be simple or composite attributes.

Figure 3.12 shows two notations for representing multivalued attributes. Multivalued attributes where the sequential ordering is not significant are represented with double arrowheads on the directed edge connecting the entity with the attribute. Multivalued attributes where a sequential ordering is to be maintained are represented using the set-constructor, with a dashed directed edge connecting the set-constructor with the attribute name. (The dashed edge is Hull's notation for indicates ordering.) The multivalued attribute Degree represents the fact that an engineer may have zero (the

---

1Elmasri and Navathe [28] and others use a double ellipse enclosing the attribute name. Tsichritzis [75] uses single headed arrowheads and annotates the connecting arc with "1" and "N" to show the multivalued nature of the attribute. The double headed arrowheads on the connecting line is from Hull.
functional mapping is *partial*), one, or more (college) degrees that are not ordered. The function $Degree(EMPLOYEE)$ evaluates to a (possibly empty) list from the domain of $Degree$. The multivalued attribute $City$ represents the fact that a flight plan will have zero, one, or more $City$ attributes that will be ordered in some manner. The function $City(FLIGHT\ PLAN)$ evaluates to a (possibly empty) ordered list from the domain of $CITY$. The domain specification for $City$ will state the ordering that is imposed. Notice that $City$ is a *partial* functional mapping, representing the fact that the cities of a particular flight plan may not (yet) be known, not that it is possible for a flight plan to have no cities.

The notation for representing unordered multivalued attributes that themselves are ordered pairs appears in Figure 3.13. $Date$ and $Hours\ Worked$ are ordered pairs.

![Diagram](image)

**Figure 3.13.** Representing unordered multivalued attributes that are ordered pairs.

Kim [46], and Elmasri and Navathe [28] augmented the relational table notation to represent multivalued attributes using braces in the following manner:

\[
ENGINEER(\{Degree\})
\]

where the attributes of an engineer as an employee are in $EMPLOYEE$. 
3.3.4 Derived Attributes

A derived attribute is one whose value is mathematically determined from other attribute values [11,28,33,37,41,66]. This is a different concept than derived relations of the relational model, which are constructed from selection, projection, and join operators. The ability to model derived attributes extends the usability of a data model, particularly when derived attributes themselves enter into relationships that need to be modeled.

Figure 3.14 shows the derived attribute names Expenses, Projected Expenses, and Shortfall enclosed by an dashed ellipse [41,28]. The derived attribute Expenses is obtained by summing Hours Worked and multiplying by Hourly Rate for each employee that participates in the PROJECT ASSIGNMENT relationship for a specific Project No. The derived attribute Projected Expenses is obtained by applying a trend analysis algorithm to Expenses on a periodic basis. The derived attribute Shortfall is obtained by subtracting the derived attribute Projected Expenses from the non-derived attribute Budget.

Figure 3.14. Derived attributes.
Hudson and King [37] represented the functional dependencies of derived attributes on one another using a dependency graph. Figure 3.15 shows the dependency graph for the derived attributes of Figure 3.14, where the edges represent the fact that a change in one derived attribute invokes a change in another derived attribute.

The domain description of a derived attribute consists of two parts; a structural description common to all attributes and a derivation rule unique to derived attributes [41]. The derivation rules defining derived attributes can be quite complex, and can use previously defined derived data.

Derived attributes cannot be directly changed; they are only changed indirectly from changes in non-derived attributes according to the derivation rules. However, derived attributes should appear (to the database user) as if their values were dynamically recomputed each time they are read. This permits derived attributes to be active in responding to changes in the environment rather than simply passively storing data. The requirement for an active response does not imply that derived attributes actually need to be dynamically recomputed. An implementation which stores the derived values is acceptable as long as it produces the same values as the dynamically recomputed values [66]. An algorithm for doing this is presented in 5.4.
3.3.5 Basic And Acquired Attributes

Entities have attributes which describe it. A basic attribute is a measurable or identifiable characteristic that is fundamental or characteristic to the entity, independent of any relationships. Height and Weight are characteristic of EMPLOYEE.

An acquired attribute [27] is not characteristic to the entity, but, rather is associated with the entity as the result of an abstraction process in which a relationship involving the entity has been omitted. The abstraction process is illustrated in Figure 3.16. Neither Job Type nor Date Of Hire are characteristic of EMPLOYEE, but rather, are attributes of the employment relationship between EMPLOYEE and COMPANY. In the case of a database for a single company the employment relationship is usually omitted and Job Type and Date Of Hire are moved to (acquired by) EMPLOYEE, whereas the similar attribute Date Assigned To Project (Figs 3.14 & 3.14) were directly representable.

Figure 3.16. Acquired attributes.
3.3.6 Class Attributes

Semantic data models permit an object type to be treated as an object itself, thereby permitting an object type to have attributes. A type is an object. Historically, such attributes have been called class attributes. A class attribute, then, is a single attribute that applies to each object in an object set and, with inheritance, to each specialized entity.

Two kinds of class attributes have been identified; a *shared-value* attribute and a *default-value* attribute [6]. A shared-value attribute is shared by each object (and specialized entity). A default-value variable is shared by each object (and specialized entity) when the object (or specialized entity) has a null value for its similarly named attribute. These concepts are illustrated in Figure 3.17. The object type AUTO is represented as a class CLASS OF AUTO using a function from AUTO to one box with rounded corners that lists the class attributes with *(shared)* and *(default)* appended to the attribute names\(^1\).

![Figure 3.17. Representing class attributes.](image)

The shared-value class attribute *Number Of Wheels* applies to all AUTO objects. The default-value class attribute *Fuel Capacity* applies to those autos that have a null value for their similarly named attribute (because a standard fuel tank is installed in most autos). The definition of class attributes requires that classes, shared-value class attributes, and default-

---

\(^1\)Derived class attributes also need *(derived)* appended because the notation eliminates the dashed ellipse.
value class attributes are always total functions. The function \( \text{NumberOfWheels}(a) \) where \( a \) is the identifier for a specific auto evaluates to the same (shared) value for each auto. The function \( \text{FuelCapacity}(a) \) evaluates in the obvious way. Object sets that do not show a class have an object type that is simply the aggregation of all attributes. Object sets that show a class have an object type that extends to also include the class attributes.

3.3.7 Comparison With Other Models

The most noticeable difference of the modeling abstractions with respect to the relational approach is to build a data model from a small number of disjoint object types. This has been found to be faster and less tedious than collecting attributes into a dictionary, identifying functional dependencies, and synthesizing tables. Several researchers report that the resulting data model is mostly in third normal form [8, 67]. The abstractions also differ from more traditional approaches in that there is a stronger reference to the problem domain when assigning attributes to objects. The formal underpinnings, however, are provided by the relational model. Functional dependencies are the final authority in the correct assignment of attributes, particularly in problem situations, and also when the higher level model is transcribed to a relational implementation.

Composite attributes are supported in the relational model through the concept of views. Views provide a level of abstraction that is separate from the implementation decisions. Views, however, do not enter into relationships. Views are externalized rather than internalized (they cannot be used to form entities within the database itself). Views are an end product, not a construct that can be used in the schema.

Multivalued attributes are also supported in the relational model through the concept of views. A view can present data that is not in 2nd or 3rd normal form. It is a limitation of first order predicate logic (upon which the relational model is based) that multivalued
attributes cannot be expressed in the schema. While this does not limit the power of the relational approach, it does limit its naturalness by widening the conceptual gap between the problem domain and its representation. This forced database researchers to define higher order normal forms (fourth and fifth order normal forms [77]) or artificially constructed characteristic entities [18,23] that guaranteed protection from update anomalies. The result of maintaining higher order normal forms and characteristic entities is a collection of highly fragmented relations less closely related to the users conceptualization of the system [60]. Normalized tables are implementation decisions, not modeling primitives. They have no place in a high level model. Translating multivalued attributes into normalized relations for a relational implementation is a straightforward mechanical procedure. The relational model is an implementation model, not a high level data model.

Derived attributes would seem to violate the principle of normalization by storing redundant information. Codd himself, however, stated that it was acceptable to store derived relations in the database [18]. Not representing derived attributes in a data model again widens the conceptual gap between the problem domain and its representation.

There is nothing in the relational model to inhibit acquired attributes, since they are not distinguishable from basic attributes.

The concept of class attributes does not exist in the relational model. It can be built into a relational schema by defining a single tuple relation that again that widens the conceptual gap between the problem domain and its representation.

Nothing that has been proposed, then, conflicts with the theoretically sound relational model. The abstractions serve to elevate a data model from an implementation specific model to a model that is closer to the problem domain.

With regard to other data models, only the EER model has similar modeling power. The EER model does not support the concept of an ordered list, but it could easily be extended to do so. However, the EER model does not support the concept of an aggregate-constructor, so an ordered pair cannot be represented.
3.4 Extended Forms of Specialization

3.4.1 Specialization Hierarchies

Specialization is an abstraction mechanism whereby a group of similar entities are considered as a sub-classification of an entity set. When specialization is applied the higher-level entity set $E$ is defined in such a way that it contains the attributes common to its constituent entity sets $E_1, E_2, \ldots, E_n$. The attributes of entity sets $E_1, \ldots, E_n$ that are not common form specialized entity sets. (This concept is modified in the next section.)

There are two independent reasons for specialization. First, there are attributes relevant to the specialized entity that do not apply to the more general entity. Second, specialized entity sets can participate in relationships that the more general entity set cannot.

Specialized entity sets may themselves be specialized, forming a hierarchy or lattice. Figure 3.18 shows ENGINEER as a specialization of EMPLOYEE and a generalization of ENGINEERING MANAGER, representing the fact that every engineering manager is

![Diagram showing specialization hierarchy with nodes for EMPLOYEE, ENGINEER, MANAGER, and ENGINEERING MANAGER connected by arrows.]

Figure 3.18. Specialization lattice.

---

1The goal is to avoid defining attributes that do not apply uniformly to all entities of an entity set.
required to be an engineer. Every entity in a *specialization hierarchy* participates as a specialized entity in one ISA relationship; every entity in a *specialization lattice* can participate as a specialized entity in more than one ISA relationship. The term *hierarchy* is customarily used when, in fact, either hierarchy or lattice may apply.

Specialization defines roles for members of an entity set (e.g., an employee might be an engineer). An entity may change roles without changing its fundamental identity (e.g., an engineer may become a manager without losing his or her fundamental employee identity).

Identifying the fundamental entity sets depends on the problem domain. Consider a university application illustrated in Figure 3.19 where entity sets for EMPLOYEE, ALUMNUS, STUDENT, and TEACHING ASSISTANT have been identified. The fundamental object type PERSON is obtained by generalizing common attributes. In this application, EMPLOYEE is a specialized entity set.

![Diagram](image)

**Figure 3.19.** EMPLOYEE as a specialized entity set.

Figure 3.19 also illustrates another important aspect of representing the problem domain. TEACHING ASSISTANT certainly could be an ALUMNUS, but that particular form of specialization may not be important to the application so it is not represented.
3.4.2 Attribute Inheritance

The concept of attribute inheritance is closely coupled with the concept of type. There are two prevailing models; type as a prototype and type as a template [13]. The type as a prototype model supports default inheritance, i.e., inheriting attributes from the more general entity is done by default unless specifically excluded by the specialized entity so as to be able to deal with exceptions. (Smalltalk supports this model.) The classic example is an attribute fly applied to birds, but neither penguins nor ostriches fly, so the attribute is excluded from penguin and ostrich specializations. The type as a template model supports strict inheritance, i.e., all attributes from the more general entity are inherited by the specialized entity, which then augments the inherited attributes by adding additional ones that do not apply to the more general entity. (Simula supports this model.) Wenger calls this a horizontal extension of the attributes [79]. The general entity, then, is a subset of the specialized entity. Another variation that applies to either default or strict inheritance, called attribute override, permits the domain of the attribute to be restricted in a manner that does not contradict the domain of the more general entity [13,79]. An employee entity set may, for example, have a derived attribute age with a domain of 18-120, and a specialized entity set for retired employees that restricts the domain of age to 65-120. Wenger calls this a vertical modification of the attributes. The specialized entity in this case is a subtype of the general entity. The terms supertype/subtype and superset/subset, which now have precise meanings in with respect to forms of inheritance are often inaccurately used to refer to generalized/specialized entities. The terms superclass/subclass are preferred as synonyms for generalization/specialization because they do not convey implications with respect to attribute inheritance.

One additional concept that will modify inheritance is the concept of public vs private. Private attributes are never inherited by subclasses, public attributes are always inherited (unless specifically excluded or overridden). The concept of public vs private is
actually more meaningful for operator inheritance (Section 5.1) and for implementation decisions, but is mentioned here for completeness of the subject matter.

An accurate model of a system requires that all forms of inheritance be supported in the description of the problem domain. The details of the actual implementation are left to a later design and coding stage, with the design reflecting the limitations and capabilities of the implementation language. The complete list of modifiers, then, for attribute inheritance is *override, exclude, and private*. (Public need not be a modifier because anything that is not private is public. Augment is not included because it would be redundant). The notation for representing modified attribute inheritance is to append *(exclude), (override), or (private)*, to the attribute name. The concepts are illustrated in Figure 3.20.

![Figure 3.20(a)](image1)

![Figure 3.20(b)](image2)

**Figure 3.20.** Four forms of attribute inheritance.

Figure 3.20(a) shows the subclass RETIRED EMPLOYEES having an attribute *age* which overrides the same named attribute of the superclass in the manner previously described. The attribute *Typing Speed* of SECRETARY augments the attributes inherited by default from EMPLOYEE. Figure 3.20(b) shows the attribute *Fly* being excluded from the subclass PENGUIN, which inherits all other attributes of BIRD. Without any notation to the contrary, subclass entities inherit the attributes of all superclass entities.
One last concept having to do with inheritance is name conflicts from multiple inheritance. Simply stated, a subclass cannot inherit the same named attribute from more than one superclass [55].

3.4.3 Attribute, Predicate, and User-Defined Specialization

There are three forms of specialization; attribute, predicate, and user defined. These are illustrated in Figure 3.21. Attribute defined specialization is characterized by a defining attribute in the supertype entity. The attribute *Job Type* in EMPLOYEE distinguishes which employees are secretaries and which are not. Notationally, the name of the defining attribute is placed on the inheritance arrow. Predicate defined specialization is characterized by a defining logic condition. The predicate is also placed on the inheritance arrow, e.g., the clause EMPLOYEE - MANAGER defines NON-MANAGER as the subset of employees who are not managers. Non-managers have no additional attributes. The
reason the entity set is represented is because non-manager employees enter into the LABOR COMMITTEE relationship that manager employees do not. (The construct used to represent committee membership is explained in Section 3.5.1.) Attribute defined and predicate defined subclasses are called derived schema components.

User defined specialization does not have a defining attribute in the supertype entity (there is no attribute in EMPLOYEE that distinguishes managers from other employees). Entities included in a user defined subclass are identified by the user at the time the entity is created rather than by any condition that may be evaluated. Some explicit operation by the user is necessary to add or remove MANAGERS. Notationally, inheritance arrows for user defined subtypes are not annotated.

The dynamic nature of subclass entities (as when an employee is promoted from a non-manager to a manager, or changes job classification) is one of the reasons for research into dynamic database schema evolution.

### 3.4.4 Set Constraints

Specialized entity sets $E_1, \ldots, E_n$ are said to be **disjoint** if:

$$E_i \cap E_j = \emptyset$$

Disjoint entity sets are represented notationally by connecting all inheritance arrows of the specialized entity sets to a small circular node circumscribing the letter $d$ \(^1\), followed by one inheritance arrow from the circular node to the more general entity set. Entity sets that are not disjoint are said to be **overlapping**.

---

\(^1\)The notation for representing disjoint and covering constraints is from Elmasri and Navathe [28].
Entity set $E$ is said to be the union of its specialized entity sets $E_1,...,E_n$ if:

$$E = \bigcup_{i=1}^{N} E_i$$

The entity sets $E_1,...,E_n$ are said to cover the superclass entity set $E$. The covering constraint is represented notationally by connecting all inheritance arrows of the specialized entities to a small circular node circumscribing the letter $U$, followed by one inheritance arrow from the circular node to the more general entity set. Entity sets that are both disjoint and covering are represented notationally by connecting all inheritance arrows of the specialized entity sets to a small circular node circumscribing the letters $d/U$, followed by one inheritance arrow from the circular node to the more general entity set. Without any notation to the contrary, specialized entity sets are neither disjoint nor covering.

Figure 3.22 shows two equally reasonable specialization hierarchies.

![Diagram](image)

Figure 3.22. Two equally reasonable specialization hierarchies.

Figure 3.22(a) models the fact that EMPLOYEE is covered by the disjoint specialized entity sets MALE and FEMALE. Figure 3.22(b) models an application where the sex of an employee may not be known because it is not a field on the job application.
3.4.5 Comparison With Other Models

The relational model does not support attribute inheritance. The complete set of attributes for a secretary, for example, are modeled in two separate relations, one relation containing the information of the secretary as an employee, and the other relation containing the secretary specific information. A natural join of the two relations is required to access all the attributes of the one real-world entity secretary.

None of the concepts described here conflict with the relational model. Every specialized entity set that has been described can be implemented in a static relational schema. The relational concept of non-first-normal-form (NF\textsuperscript{2}) relations [1] captures many of the static concepts described here, although no commercial data base system has implemented them. The concepts described here, however, are substantially more powerful than the static schema components of the relational model. The concepts described here are dynamic. Derived schema components add semantic information which is not expressible in the relational data model.

With regard to other data models, only the EER model has similar modeling power. Many of the concepts described were adapted from the EER model [28]. The EER model does not support inheritance modification, but could easily be extended to do so.

3.5 Extended Forms of Constructed Objects

One of the fundamental abstractions from semantic data models is the concept of constructing object types (i.e., independent entity sets) out of other object types. The abstraction has two forms, composite objects and part hierarchies\textsuperscript{1}. The two forms are

\textsuperscript{1}The term complex object is frequently used to mean either composite object, or part hierarchy, or both.
closely related; composite objects being a more general form that embodies the concept of a
collection or an additive whole; and part hierarchies being a more specialized form that
embodies the concept of a more structured, tightly coupled whole. Each adds a different
dimension to the concept of an object than that addressed by a specialization hierarchy.

3.5.1 Composite Objects

The seminal definition of composite objects was presented in SDM by Hammer and
McCleod [33] and further defined in a relational context in RM/T by Codd [18]. The
example was a CONVOY composed of SHIPS, where both entity sets are independent.
Following Hulls notation, Figure 3.23 shows the independent object sets CONVOY and
SHIP represented as triangles, and their relationship to each other represented as an
attribute function to a set constructor. Following the original example, the set constructor
has a derived class attribute # Ships, which is the number of ships in the convoy. The
attributes Row Position and Column Position add to the original example. They represent
attributes of ships by virtue of their membership in convoy.

Figure 3.23. CONVOY is a composite object.
A convoy is an independent object with attributes separate and distinct from the attributes of its memberships. A convoy can exist even if it has no member ships. Inheritance does not apply, i.e., a convoy does not inherit attributes of its member ships, nor do the member ships inherit attributes from the convoy. The function `Member(CONVOY)` evaluates to a (possibly empty) list of ships.

Composite objects can be nested. Expanding on the example in the manner similar to Codd, a TASK FORCE can be a composite object formed from CONVOY, FIGHTER WING, and BATTLE GROUP, which themselves can be composite objects. Because of nesting and two type constructors, The notation is extensible in its ability to represent composite objects of arbitrary complexity.

The membership of separate and distinct object types in a composite object may be attribute defined, predicate defined, or user defined [33]. Notationally, the defining attribute or predicate annotates the directed edge between the set constructor symbol and the participating object. Unlabeled edges are user defined. Figure 3.24 shows a composite object with attribute defined member objects.

![Figure 3.24. A generalized object.](image-url)
Figure 3.24 also illustrates another form of composite object, i.e., a generalized object formed by the disjoint union of otherwise independent objects [3,18]. Registered vehicles, in general, are either cars or trucks, but any specific registered vehicle is only one or the other. (The domain of the reference attribute that represents this relationship is either a Car ID or a Truck ID.) The figure also shows that all trucks must be registered because that attribute function is total, but cars need not be registered because that attribute function is partial. A generalized object is different than an ISA relationship (cars and trucks cannot switch roles, and neither is fundamentally a registered vehicle).

REGISTERED VEHICLE is sometimes called a union type [72], or a category [27,28]. Codd called the concept alternative generalization [18]. The concept is similar to a Pascal variant record. Hull calls the relationship a generalization ISA relationship [3], and used a different symbol for representing it than that used here. The notation for disjoint sets is used here for consistency.

A composite object, then, is an object that is defined in terms of other objects. A composite object is formed by aggregating objects to form an object that is an additive whole. The underlying notion of a composite object is the attribute, predicate, or user defined collection of objects from an object set into another set.

### 3.5.2 Part Hierarchies

The object sets modeled in CAD/CAM, CASE, geographic mapping, and similar applications are assemblies that are themselves aggregates of smaller object sets. At one level of abstraction an assembly is defined and manipulated as a single object (with attribute values and other non-decomposable objects), while at another level of abstraction the
complete and detailed aggregated structure of an assembly may be viewed. An assembly of such object types is called a part hierarchy\(^1\) [6,43,44,47,54,63].

A part hierarchy is a special case of a composite object that collects object sets into a more structured, tightly coupled whole rather than into an additive whole. A part hierarchy enforces referential integrity that is not enforced with the more general composite object [64]. Part hierarchies impose the IS-PART-OF relationship [44].

Consider an assembly, CAR, (Figure 3.25) which is composed of component assemblies BODY, DRIVE TRAIN, and ELECTRICAL SYSTEM, which themselves are assemblies, down to some primitive level. Each component object in the parts hierarchy has its own attributes independent of its participation in the hierarchy. A car, for example, has a Vehicle ID, which is an attribute of CAR, rather than a component part.

An object set may participate in more than one parts hierarchy (e.g., DRIVE TRAIN may also be a member of a parts hierarchy with TRUCK). In a physical part hierarchy each individual drive train may be a member of (at most) one parts hierarchy

\(^1\)The term complex object is sometimes used.
(e.g., with either CAR or TRUCK). In a *logical part hierarchy* an object may be part of two different assemblies [46]. Each object in a parts hierarchy may also participate in other relationships, e.g., MANUFACTURED BY.

The notation for representing part hierarchies is slightly different than the notation for modeling composite objects because the relationship is stronger than the relationship of attributes or composite objects. Kim [46] has reported that both the relationship, and the inverse relationship needs to be represented because a part mediates the behavior of its components, and the components affect the behavior of the assembly. The functional representation of relationships and inverse relationships between objects serves quite nicely. For simplicity, consider only the part hierarchy between CAR and BODY (Figure 3.26). Neither Hull’s notation, nor the EER model, represent part hierarchies, so Hull’s notation is extended. The fat directed edge is used to represent the part hierarchy so as to distinguish it from other kinds of relationships. The assembly to component relationship is

![Figure 3.26. Representing a bottom-up, physical parts hierarchy.](image)

*partial* when the assembly is constructed from the top down and *total* when the assembly is constructed from the bottom-up, and is always *1:1*. The inverse relationship from component to assembly is always *1:1* for a physical parts hierarchy and cannot be *1:1* for a logical parts hierarchy, and can be either *total* or *partial* depending on whether components can exist independent of assemblies. Figure 3.26 shows a physical parts hierarchy for a
car assembly that cannot exist without a body component, and a body that may exist without being assigned to a car. It also shows the default-value class attribute *Color*. This represents an attribute that applies to the entire assembly as a whole [6,54]. The search for a component object's attribute value proceeds through the inverse relationship, stopping when the similarly named attribute is found with a non-null value.

### 3.5.3 Comparison With Other Models

No conventional data model supports the abstractions of composite objects or part hierarchies. The problem with the relational model is that a composite object (or a part hierarchy) must be decomposed into tuples over several relations. One researcher shows that "object fetches" can be expressed as a sequence of SQL statements [64]. Another researcher has built a prototype relational data base for non-first-normal-form (NF2) relations that supports relations as attribute values [21]. Stonebraker [72] states that relational databases should support union types, but points out implementation problems. Clearly the concepts are relational compatible.

The EER model supports generalized objects with categories [27,28]. The functional representation is more straightforward. The EER model supports the notion of part hierarchies in a limited manner [73]. The fact that the EER model does not support the concept of inverse relationships severely limits its usefulness for modeling part hierarchies. Inverse relationships are necessary for modeling transitive attributes.

### 3.6 Versions

Many design applications (e.g., CAD/CAM and software development applications) require the ability to manage multiple versions of an object [6,16,44,47] before selecting
the one that satisfies the requirements. Object versions are also important for publishing applications, and for historical databases, such as those used in accounting, legal, and financial applications that require access to the past information of the database. The concept of object versions has less to do with an object-oriented data model and more to do with the kinds of applications for which the model is developed.

An object can be versioned in one of two manners, i.e., a transient version or the working version. (An object need not be versionable.) A transient version may be created from scratch or derived from an existing transient or working version. A transient version may be deleted or updated at will, or explicitly upgraded to the working version. The working version may be updated at will, but it may not be deleted (it must first be replaced by a transient version that is upgraded to the working version).

Versioning is an object property, not a class attribute. Objects belonging to the same object can have different versions. A version number is automatically generated by either the underlying database or the application, and is associated with each version of an object. Also associated with each object set are a default version number and a set of version descriptors. The default version number is the version number of the working version. The version descriptors, one for each version of the object, includes implementation specific information. Banerjee [6] provides an example of the version descriptors used in the ORION object-oriented database system.

References to versioned objects, such as in a one-to-many relationship or a parts hierarchy, may be either specific or generic. A reference to a specific object version is statically bound. A reference to a generic version is dynamically bound to a default version of the object. In the absence of a user-specified default, the most recent transient version is the default version.

Notationally, versioned objects are represented by appending (versioned) to the version name so as to distinguish them from non-versioned objects. The presence of
version number, default version number, and version descriptors also helps to identify versionable objects.

### 3.7 Summary

The concepts embodied by composite attributes, multivalued attributes, class attributes, specialization hierarchies, and attribute inheritance is an attempt to extend the type system of a data model. The concepts embodied by derived attributes, composite objects, and part hierarchies is an attempt to support extended forms of relationships. The concepts embodied by the functional representation of attributes, by the functional and inverse functional representation of relationships is an attempt to add semantic information to a data model.

Compare how the data structuring abstractions extend a data model with respect to the “drawbacks” of the current relational model identified by Michael Stonebraker [72]:

<table>
<thead>
<tr>
<th>Data Structuring Abstractions</th>
<th>“Drawbacks” of The Relational Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extended type system</td>
<td>Anemic type system</td>
</tr>
<tr>
<td>Extended forms of relationships</td>
<td>No support for complex objects</td>
</tr>
<tr>
<td>Extended semantic information</td>
<td>No rules</td>
</tr>
</tbody>
</table>

Stonebraker’s rules not only add the kind of semantic information identified here as integrity rules, but also add user defined semantic knowledge. Clearly, the data structuring concepts of an object-oriented data model are relational compatible.
Chapter 4

Integrity Rules

Integrity constraints are restrictions on entity Create/Delete operations, and on attribute value updates, that guarantees data always reflects the underlying data model. Constraints are (historically) identified as:

1. Domain integrity.
2. Entity integrity.
3. Referential integrity.
4. User-defined.

Domain integrity guarantees that each attribute value conforms to the domain descriptions (Section 3.1.2). This constraint is assumed, and is not elaborated. Entity integrity guarantees that no component of a primary key has a null value \([17,18,19]\). This constraint is subsumed by the broader constraint that says attributes that are \textit{total} cannot have a null value. (Primary keys are \textit{total}, \(1:1\)). Referential integrity guarantees that each distinct nonnull entity reference (foreign key in a relational system) is a reference to an entity that actually exists \([17,18,19]\). (There can be no \textit{dangling references}.) User defined integrity constraints are, according to Codd, "additional integrity constraints reflecting either business policies or government regulations" \([20]\). User defined integrity constraints are beyond the scope of this thesis.\(^1\)

---

\(^1\)An example of a user defined integrity constraint is the restriction that the salary of an employee cannot exceed the managers salary.
4.1 Extracting Semantics from Structure

Integrity rules are inherent in the functional representation of attributes, objects, subclasses, and relationships. They require no knowledge of the application. Consider Figure 4.1.

![Diagram](image)

Figure 4.1. Integrity rules are inherent in the functional representation.

The functional mapping from DEPARTMENT to Dept No is total, 1:1, indicating that each department must have a unique value in the Dept No domain. The functional mapping from DEPARTMENT to Name and Staffing Level is partial, indicating that these attribute values may have null values. The total and partial functional mappings between EMPLOYEE and its attributes is identical. The attribute Job Type is a defining attribute between
EMPLOYEE and its subclasses, which are disjoint and cover EMPLOYEE. Job Type, then, must be total (cannot have a null value), or the disjoint, cover set constraints would be violated. Furthermore, inserting a new employee will cause an insertion in either ENGINEER, TECHNICIAN, or SECRETARY because they cover EMPLOYEE. Similarly, deleting one of the subclasses will cause a deletion in the superclass. The relationship between EMPLOYEE and DEPARTMENT is total meaning departments that are the domain of the relationship cannot be deleted. If the relationship were partial, deleting a department would cause interobject references from EMPLOYEE to the deleted department to receive a null value.

The example illustrates that the rigorous application of functions and set constraints between objects, attributes, and relationships identify integrity rules for Create/Delete/Update operators. The integrity rules are inherent in the conceptual representation. They require no special knowledge of the problem domain. In fact, the procedure for identifying integrity rules can be automated if a data description language is formalized that represents the data structure and relationships.

4.2 Rules for Maintaining Structural Integrity

Update Rules

Rule U1. Values for total attributes cannot have a null value.

EXAMPLE (Ref: Figure 4.1) Name, Job Type and Social Security Number of EMPLOYEE, and Dept No of DEPARTMENT cannot have null values.

Rule U2. Values for 1:1 attributes can be used as the domain of only one entity.
EXAMPLE (Ref: Figure 4.1) *Social Security Number* of EMPLOYEE, and *Dept No* of DEPARTMENT cannot have null values (because they are *total*). Each employee must have a unique *Social Security Number* and each department must have a unique *Dept No* (because they are 1:1).

Rule U3. The defining attribute for an attribute-defined or predicate-defined group of subclasses that *cover* a superclass must be *total*.

EXAMPLE (Ref: Figure 4.1) *Job Type* of EMPLOYEE must be *total*.

Rule U4. The defining attribute for an attribute-defined or predicate-defined generalized object covered by its member objects must be *total*.

EXAMPLE (Ref: Figure 3.24) *Vehicle Type* of REGISTERED VEHICLE must be *total*.

Rule U5. Modifying the value of an attribute that is used in the derivation rule of a derived attribute makes the derived attribute appear (to the database user) as if its value was dynamically recomputed the next time it is read. This rule applies recursively to all derived attribute values dependent on the newly derived value.

EXAMPLE (Ref: Figure 5.9) Modifying the value of the non-derived attribute $d_1$ makes the derived attributes $d_d, e_{d1}, e_{d2},$ and $e_{d3}$ appear (to the database user) as if their value were dynamically recomputed the next time they are read.

Rule U6. Modifying the *value* of an attribute that is used as the defining attribute in an attribute-defined or predicate-defined subclass (or composite object) will cause the entity to be added to (or deleted from) the related subclass entity (or object)
set as required. This rule applies recursively to all subclass entity sets that are dependent on the (possibly) new subclass entity set.

EXAMPLE (Ref: Figure 4.1) Modifying, for example, JobType(John) in EMPLOYEE from Technician to Engineer will cause the John as TECHNICIAN subclass to be deleted, and a new subclass for John as ENGINEER to be created. Total attributes of ENGINEER (if there were any) would also be required to complete the operation.

Rule U7. In a system that references objects by value instead of using distinct object identifiers (such as a relational system), changing the value of an attribute that is part of the object identifier (i.e., primary key) will cause the value to change in all related relations.

EXAMPLE (Ref: Figure 4.1) Giving an existing department a new Dept No will cause all foreign keys in EMPLOYEE to have the reference similarly changed.

Delete Rules

Rule D1. An entity that is the domain of a total relationship cannot be deleted. This rule takes priority over all other deletion rules.

EXAMPLE (Ref: Figure 4.1) If a department \(d \in \text{DEPARTMENT}\) is the domain of any employee \(e \in \text{EMPLOYEE}\), then \(d\) cannot be deleted.

Rule D2. Deleting an entity that is the domain of a partial relationship implies either: 1) the reference from the related entity will set to a null value; or, 2) or the aggregate-constructed entity that represents the relationship is also deleted.
EXAMPLE (Ref: Figure 3.13) If an employee $e \in$ EMPLOYEE that is the domain of $p \in$ PROJECT ASSIGNMENT is deleted, then the aggregate-constructed entity $pa \in$ PROJECT ASSIGNMENT that represents the relationship must be deleted.

Rule D3. Deleting an entity implies that it is also deleted from all subclass entities (if any) and all composite entities (if any).

EXAMPLE (Ref: Figure 4.1) If an employee $e \in$ EMPLOYEE is deleted and if $e \in$ NON-MANAGER, then $e$ must also be deleted from NON-MANAGER. If $e \in$ LABOR COMMITTEE, then $e$ must also be deleted from LABOR COMMITTEE.

Rule D4. Deleting an entity from a group of subclasses that cover a superclass will also delete the superclass entity.

EXAMPLE (Ref: Figure 4.1) If an engineer $e \in$ ENGINEER is deleted then $e \in$ EMPLOYEE must also be deleted.

Rule D5. A versioned object cannot be deleted if its version number is the same as the default version number for the object, or if it is the working version.

EXAMPLE. Refer to references [16,45,46,47].

Rule D6. Deleting an entity that is used in the derivation rule of a derived attribute makes the derived attribute appear (to the database user) as if its value was dynamically recomputed the next time it is read. This rule applies recursively to all derived attribute values dependent on the newly derived value.
EXAMPLE (Ref: Figure 4.1) If an employee $e \in \text{EMPLOYEE}$ is deleted, then the derived shared class attribute $\# \text{Employees}$ should appear (to the database user) as if its value was dynamically recomputed the next time it is read.

**Insert Rules**

**Rule I1.** Inserting an entity that is the range of a *total* relationship implies the relationship with a related entity will be made.

EXAMPLE (Ref: Figure 4.1) Inserting a new employee $e$ into EMPLOYEE will make a relationship from $e$ to a department $d \in \text{DEPARTMENT}$.

EXAMPLE (Ref: Figure 3.24) Inserting a new truck $t$ into TRUCK will make a relationship with $r \in \text{REGISTERED VEHICLE}$.

**Rule I2.** Inserting an entity into a superclass causes the entity to be inserted into all attribute-defined or predicate-defined subclasses or composite objects as determined by the defining attribute or the defining predicate.

EXAMPLE (Ref: Figure 4.1) Inserting a new employee $e$ into EMPLOYEE will cause a new $e$ to be inserted into either ENGINEER, TECHNICIAN, or SECRETARY. *Total* attributes of the new subclass (if there were any) would also be required to complete the operation.

**Rule I3.** Inserting an entity into a superclass that is covered by a specialization hierarchy causes the entity to be inserted in at least one of the subclass.

EXAMPLE The same example as Rule I2 applies.
Rule I4. A subclass entity cannot be inserted into a group of disjoint subclasses if it is already an entity in one of the other subclasses.

EXAMPLE (Ref: Figure 4.1) If a secretary $e \in \text{SECRETARY}$, then $e$ cannot be inserted into ENGINEER without first deleting $e$ from SECRETARY.

Rule I5. Adding an entity into a group of subclasses that cover a superclass will also add the entity to the superclass. If the covering subclass is user-defined, then additional user information may be required for the insert.

EXAMPLE (Ref: Figure 4.1) Adding a new engineer $e$ into ENGINEER will cause a new $e$ to also be inserted into EMPLOYEE. Total attributes of the new employee are also required to complete the operation.

Rule I6. Inserting a versioned object will cause a system assigned version number to be assigned, which (optionally) will become the default version number.

EXAMPLE Refer to references [16,45,46,47].

Rule I7. Inserting an entity that is used in the derivation rule of a derived attribute makes the derived shared class attribute appear (to the database user) as if its value was dynamically recomputed the next time it is read. This rule applies recursively to all derived attribute values dependent on the newly derived value.

EXAMPLE (Ref: Figure 4.1) If a new employee $e$ is inserted into EMPLOYEE, then the derived attribute $\# Employees$ should appear (to the database user) as if its value was dynamically recomputed the next time it is read.
Comparison With Other Models

The EER model has similar capability. It is somewhat limited in this respect, however, because inverse relationships are not modeled, and because it does not identify total and 1:1 attributes (although it could be extended to do so).

There is no comparable aid inherent in the relational model. The consequences of Create/Delete/Update operations must be modeled using the insight of the data modeler or knowledge of the problem domain. The procedure cannot be automated.
Chapter 5

Operators: Adding Behavior to Structure

The three components of a data model, as defined by Codd and still accepted today, are data structure, operators, and integrity rules. Data structure and integrity rules have been addressed. It remains to describe operators to insert, delete, retrieve, and modify attributes that will guarantee structural integrity by adhering to the integrity rules.

The relational model defined tuple selection, projection, join (and other) operators using relational algebra. While relational algebra provided a sound theoretical basis for what had previously been an ad-hoc technology, it is cumbersome with extensive data structures, for several reasons. First, complex structure requires the scope of operators apply over many relations; Second, the application program (or user) must consciously navigate through many relations using “knowledge” of their name and their relationships. Third, more complex operators are required, particularly for derived attributes.

5.1 Identifying Behavior

Operators are similar to class attributes. Just as one class attribute applies to each object in an object set, so too, one operator specification applies to each object in an object set. The similarity suggests that the same graphical notation and functional representation be used so as to combine attribute modeling and operator identification into a common object metaphor. Loomis [54] took a similar approach.
An entity has operators to insert, delete, retrieve, and modify attributes, each with a unique name and a specification. Notationally, operators for an object type (or specialized entity) are represented by connecting the object (or specialized entity) symbol to one CLASS OF box with rounded corners using a thin directed edge. The box lists the names of all operators and all class attributes (if there are any). Operators to set, retrieve, or modify simple attribute values are assumed and are not usually shown at the conceptual level [54]. The specification of each operator is modeled separately using any of the commonly accepted approaches. This conceptual approach is similar to the approach of graphically representing attributes as functions, and relegating their precise domain specification to a separate document. These concepts are illustrated in Figure 5.1, which shows EMPLOYEE with the derived shared-value class attribute # Employees and having Age and LengthOfService operators.

![Figure 5.1. Representing operators](image)

The specification for these operators might read:

- Age: returns the current date - Birthdate(EMPLOYEE)
- LengthOfService: returns the current date - DateOfHire(EMPLOYEE)

where Age(EMPLOYEE) and LengthOfService(EMPLOYEE) each return values from domains determined by their specifications. Notice there is no distinction between attributes and operators. The graphical notation and the functional syntax are identical.
5.2 Operator Inheritance

Just as a subclass inherits attributes from a superclass, so too, they inherit operators. Inherited operators can be modified in much the same manner as inherited attributes, i.e., augment, override, public, private, and exclude. These are clarified below.

Operator override causes the subclass operator to be executed instead of the superclass. Operator augmentation causes the subclass operator to be applied first, followed by the same-named operator in the superclass, or vice versa. The distinction between override and augmentation is described in the specification. Override is illustrated in Figure 5.2, where POLYGON has one operator, Perimeter, which is inherited by TRIANGLE. RECTANGLE modifies the specification of Perimeter, and SQUARE, modifies the specification that it would have otherwise been inherited from RECTANGLE.

![Figure 5.2. Operator override.](image)

The three definitions for Perimeter are:

- Perimeter(POLYGON): Sum the length of the edges.
- Perimeter(RECTANGLE): $2 \times$ the length of two consecutive edges.
- Perimeter(SQUARE): $4 \times$ the length of any edge.
Meyer [55] identifies one exception to the rule that all operators are inherited by subclasses for the Eiffel language, and that has to do with a Create operator, which is private. He points out the reason for this exception is that subclasses have more attributes than their superclass and must, therefore, augment the operator. Halbert and O’Brien [31] found other exceptions and made all operators public or private in Trellis/Owl. Private operators are not inherited by their subclasses. Using material presented in this thesis, rather than from Halbert and O’Bien’s example, the reason for permitting private operators is that operator semantics often differ in the substructure of a specialization hierarchy. Consider, for example, the operator Fire applied to employee John, who is an engineer and also a manager and an engineering manager as represented in Figure 5.3. Integrity rule D3 requires an operator that results in John being removed from EMPLOYEE implies that John will also be removed from all subclasses. The operation is not clear if the operator is inherited by MANAGER or ENGINEER. It could be that the intent is to remove John from one subclass but not the other. The modeler, then, may choose to inhibit operator inheritance because of semantic uncertainties such as these, or in lieu of augmenting the operator in the subclass to account for slightly different semantics.

![Figure 5.3. Representing private operators](image-url)
Alan Snyder [70] argues for permitting a subclass to \textit{exclude} an inherited operator. This thesis suggests operators be designated as \textit{private} or \textit{exclude}. (Public is the norm because anything that is not private is public.) The notation for operator modification is to append \textit{(private)}, \textit{(exclude)}, \textit{(augment)} or \textit{(override)} to the operator name.

One last point is important regarding operator inheritance. Whereas attribute inheritance occurs only within a specialization hierarchy, operator inheritance is not similarly restricted. Operators can be inherited across differing object types. This is particularly significant at the detailed design stage of the software life-cycle, and less important when modeling the problem domain.

5.3 Semantic Operators

Klaus Dittrich [26] identifies three levels of object-orientation:\footnote{Dittrich also identifies two other directions where the notion of object-orientation is used, but states that they do not contribute to the definition as applied to database systems. These are \textit{object-orientation implementation} (i.e., a specific kind of modularization) and \textit{object-oriented user/programming interface}.}

1. \textit{structural object-orientation}: the data model defines structures to represent entities of any complexity.

2. \textit{operational object-orientation}: the data model is structurally object-oriented, and also includes operators to deal with objects in their entirety.

3. \textit{behavioral object-orientation}: the data model includes features to define new object types of any complexity, together with their operators.

The data model described thus far is structurally object-oriented. \textit{Semantic operators} to deal with objects in their entirety have not been identified. The concept is called \textit{view operations} in a relational system, and \textit{perspectives} in object-oriented programming languages [71]. The goal is to reduce the semantic gap between the database
representation and the representation captured by the applications code. An operationally object-oriented data model identifies operators that reduce the semantic gap.

Consider the composite attributes represented in Figure 5.4. Twelve trivial tuple read operators, which are not usually represented, are required to read the simple attributes.

The data model becomes operationally object-oriented when semantic operators are added. The following semantic operators can be identified by inspecting Figure 5.4:

ReadEmployee ← First Name, MI, Last Name, Number, Street, Apartment Number, City, State, Zip, Birthdate, Social Security Number, Job Type

ReadName ← First Name, MI, Last Name

ReadAddress ← Number, Street, Apartment Number, City, State, Zip

ReadStreetAddress ← Number, Street, Apartment Number

where the attributes to the right of the "←" symbol represent the simple attributes whose values are read with the semantic operator to the left of the symbol.

Consider the more complex structure represented in Figure 5.5. Operators to deal with EMPLOYEE are more complex because of the attribute-defined SECRETARY and the user-defined MANAGER subclasses. An example of one such semantic operator was already provided, although it was not identified as such, i.e., Name(SECRETARY) returns the inherited name of the currently active secretary. Inheritance is conceptually satisfying.
but implementation dependent. An implementation that overlays a relational DBMS will require semantic operators such as \textit{Name(SECRETARY)} be implemented in code, and therefore, a proper component of an operationally object-oriented data model.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure5_5.png}
\caption{Subclasses make semantic operators complex.}
\end{figure}

\textit{ReadEmployee}, then, will not deal with EMPLOYEE in its entirety until it has provisions for the single-valued attribute \textit{Typing Speed} and the multivalued attribute \textit{Training Classes}. The more complete description of \textit{ReadEmployee} is:

\begin{verbatim}
ReadEmployee ← First Name, MI, Last Name, Number, Street, Apartment Number, City, State, Zip, Birthdate, Social Security Number, Job Type, [Typing Speed], {[Training Classes]}
\end{verbatim}

where the brackets represent attributes that may not apply to all employees, and the braces represent a (possibly empty) set of values.

The concepts of dealing with objects in their entirety becomes more complicated when relationships are considered. Consider Figure 5.6 with the operator \textit{Fire} applied to employee \textit{John}, when John is assigned to a project. Deletion rule D2 requires that a side
effect of such an operator should also update the PROJECT ASSIGNMENT relationship appropriately. Korth [51] makes the same point with respect to relational languages.

5.4 Propagation of Operators

Just as selected attribute values of an object that is an assembly in a parts hierarchy can be propagated to the component objects (rather than being inherited by them), so too, operators in a parts hierarchy can trigger same-named operators of component objects [54]. Triggering iterates over the component objects according to a sequence identified in the operator specification. The component objects, themselves, may be assemblies, which may again trigger same-named operators of their component objects.

Consider an assembly, CAR, which is composed of component assemblies BODY, DRIVE TRAIN, and ELECTRICAL SYSTEM, which themselves are assemblies, down to some primitive level. Each component also participates in other relationships, e.g., MANUFACTURED-BY. How far should a CAD/CAM operator Display propagate? Certainly not outside the parts hierarchy. Furthermore, the application may want to control the depth of propagation such that the detailed structure of a component can dynamically be

![Diagram](attachment:diagram.png)

Figure 5.6. Attribute updates should include side effects on relationships.
made visible or invisible. And what of the inverse relationship? Displaying the detailed structure of a component assembly should not necessarily cause the higher level assembly to be displayed; whereas rotating the component should cause a rotation of the entire assembly, an operation the application should not be able to inhibit. The problem is similar to shallow vs deep operators that occurs in object-oriented programming.

Loomis [54] identified operators in an assembly that can possess the triggering ability by appending \((T)\) to their name. The notation fails, however, to allow the \(T\) attribute for the inverse relationship, and does not to permit dynamic control over the trigger. Rumbaugh [63] suggests that propagation attributes be associated with \(<\text{operator}, \text{object type}, \text{relationship}>\) triplets, and that the domain of attribute values be \(\text{none, propagate, shallow, and inhibit}\). He did not address the concept of permitting dynamic control over attribute values. The concepts of both Rumbaugh and Loomis, then, are expanded in the following manner: Propagation attributes be permitted for both the relationship and the inverse relationship; attributes have dynamic and static constraints; and that the domain of attribute values be \(\text{none, shallow, and deep}\). The following definitions apply:

*Propagation Attribute Constraints*

Dynamic  The propagation attribute can be dynamically controlled by the application.

Static  The propagation attribute cannot be altered by the application.

*Propagation Attribute Values*

None  Do not apply the operator. Do not trigger the same-named operator in the components.

Shallow  Apply the operator. Do not trigger the same-named operator in the components.

Deep  Apply the operator. Trigger the same-named operator in the components.
None and deep are identical to Rumbaugh's definitions, and shallow is equivalent to his propagate. Rumbaugh's inhibit has special meaning with respect to shared resources, an implementation (rather than a conceptual modeling) feature beyond the scope of this thesis. Without any notation to the contrary, propagation attributes are static and deep.

These concepts are factored into the extended notation illustrated in Figure 5.7, where each operator name has three attributes appended; the first attribute is the constraint (dynamic or static); the second attribute is the propagation value (none, shallow, or deep); and the third attribute indicates whether the operator applies to lower level components or to higher level assemblies (TO component or TO assembly). Operators not relevant to the parts hierarchy are purposely excluded.

Figure 5.7. Representing operator propagation in a physical parts hierarchy

Figure 5.7 shows the application may dynamically control the display operator (e.g., the detailed structure of a car body may be made visible or invisible when the detailed structure of a car is visible), and the rotate operator propagates to all components and assemblies that participate in the parts hierarchy.
5.5 Object Dynamics

Objects are dynamic, rather than static. The state of an object is the value of its attributes. An object's state changes whenever an attribute is updated. Conceptually, an attribute update will also cause an update to all derived attributes that use the non-derived attribute in a derivation rule. The effect is transitive to all derived attributes\(^1\) that use the newly derived attribute in a derivation rule. Derived attributes, then, are active in responding to changes in their environment, rather than passive data stores.

Most examples in the literature that demonstrate object dynamics use examples that model quite naturally as state machines. Booch \cite{10}, for example, uses a cruise control system. Windowing environments are another common example, although one author \cite{74} recommends that they be avoided when first learning object-oriented concepts. This thesis examines the object-oriented data modeling approach with respect to an application that can be modeled using a transform-centered approach, and, in fact, shows how the transform-centered model complements the object-oriented data model\(^2\).

A Transform-Centered Example

The example is generic. It has a structure, common to scientific and engineering applications, consisting of object types that represent experimental measurements or population samples; object types that represent calibration or background or reference measurements; and an observed phenomena obtained by mathematically or statistically combining information from the experimental measurements and calibration / background

\(^1\) Again, (ref 3.3.4) the requirement for an active response does not imply that derived attributes actually need to be dynamically recomputed each time a dependent attribute value changes. They must, however, appear to the database user as if their values were recomputed each time they are read.

\(^2\) This is an original contribution that has not appeared in the literature.
measurements into a representation of a real world occurrence such as the water content of snowpack, power dissipation, earthquake intensity measurements, etc. The observed phenomena, itself, can be further transformed into other domains of interest that represent growth rates, statistical trends, frequency component analysis, etc. The object types are represented in Figure 5.8 where A - D represent the experimental and calibration / background measurements, and E represents the observed phenomena. Each object type has an arbitrary number of attributes identified by subscripts 1 to n, one or more derived attributes identified by the subscript d, and an attribute identified by the subscript s that will be described later.

Figure 5.8. Five object types with non-derived and derived attributes.

Figure 5.9, a dependency graph, shows the functional dependencies of derived attributes. A change in the derived attribute at the tail of the arrow will cause a change in the derived attribute at the head of the arrow. (Recall that derived attributes cannot be directly changed; they are only changed indirectly from changes in non-derived attributes.

---

1Dependency graphs in the literature do not show dependencies from non-derived attributes. Figure 5.8 shows them, using an arbitrary subset from the state of the derived attribute's object.
according to derivation rules). Each derived attribute, then is functionally dependent on other derived attributes and/or non-derived attributes.

Figure 5.9. Dependency graph showing functional dependencies from all attributes.

A data flow diagram (DFD) that models the same mathematical transformations as those represented by the dependency graph appears as Figure 5.10. The circles represent rules that transform inputs from data stores (represented as a pair of horizontal lines) and data flows (represented as directed arcs) into output data flows. DFDs are used in transform-centered modeling popularized by DeMarco, Page-Jones, and others [25,35,59].

Figure 5.10. Data flow diagram from transform-centered modeling.
The central idea of a DFD is that all inputs flow into the system and that all inputs are simultaneously available to a process. Start up procedures are not modeled very well. Transform-centered modeling does not map well into designs where state is important, as it would be if the application is implemented on a graphics workstation, because state, where output is dependent on current and past inputs, is a foreign concept.

The similarity (on the surface) between the dependency graph (Figure 5.9) and the DFD (Figure 5.10) is readily apparent. Below the surface, however, there is a large difference. The dependency graph is just a graph, i.e., a picture. The DFD has written specifications that describe the precise rules to be followed for synthesizing outputs from inputs. DFD modeling is useful for precisely describing the evaluation rules of the dependent attributes.

Augmenting an Object-Oriented Model With a Transform-Centered Model

The strategy for augmenting an object-oriented data model with a transform-centered model is to assign three responsibilities to each object: 1) Responsibility for synthesizing derived attribute(s) according to the evaluation rules; 2) Responsibility for recognizing when derived attribute(s) conform to the evaluation rules and when they violate the rules; and, 3) Responsibility for notifying directly dependent derived attributes when the rules are first violated.

The first responsibility is achieved by using the evaluation rules of the transform-centered model as the evaluation rules for derived attributes. The second responsibility is achieved by introducing additional derived attributes, called state variables, one state variable for each derived attribute. State variables, which are the attributes with the $s$ subscript in Figure 5.7, take the value GOOD whenever their associated derived attribute has been successfully synthesized, and the value BAD whenever their derived attribute
cannot conform to the evaluation rules. The latter occurs whenever a non-derived attribute used in the evaluation rules is updated, or whenever a derived attribute used in the evaluation rules becomes BAD. The third responsibility is achieved through references to all directly dependent derived attributes. These concepts are illustrated (Figure 5.11) using the derived attribute $d_d$ from Figs 5.7 - 5.9. Recall, $d_d$ is dependent on the derived attributes $b_d$ and $c_d$, and that derived attribute $e_{dl}$ is directly dependent on $d_d$.

Synthesis of derived attributes starts with the $Eval d_d$ operator.

```
procedure Eval d_d (v_s, v_d)
  if d_s = GOOD
    v_s <- d_s
    v_d <- d_d
  else
    Compute d_d(d_s, d_d)
    if d_s = GOOD
      v_s <- d_s
      v_d <- d_d
    else
      v_s <- BAD
      v_d <- undefined
  end if
end procedure
```

When $Eval d_d$ receives a REQUEST (indicated with a dashed directed line) it examines the value of the state variable $d_s$. If the value of $d_s$ is GOOD, the value of $d_d$ conforms to the evaluation rules and the $Eval d_d$ operator evaluates to the ordered pair $d_s$,GOOD. If the value of $d_s$ is BAD, a REQUEST to $Compute d_d$ starts synthesis. If synthesis is successful, $Eval d_d$ again evaluates to the ordered pair $d_s$,GOOD. If synthesis is not successful, $Eval d_d$ evaluates to BAD and $d_d$ is undefined. The $Eval d_d$ REQUEST can originate from either $e_{dl}$, or it can originate from some other external source, i.e., $Eval d_d$ can be invoked independent of $e_{dl}$, which is why Figure 5.10 shows two REQUEST into $Eval d_d$. 

Figure 5.11. Synthesizing derived attributes.
Compute $d_d$ synthesizes $d_d$ according to the evaluation rules from the transform-centered model, initiating REQUESTs to $Eval_b_d$ and $Eval_c_d$, and inheriting $d_1 .. d_n$ from the object environment.

```plaintext
procedure Compute d_d ( d_s , d_d )
    status ← BAD
    Eval b_d ( b_s , b_d )
    if b_s = GOOD
        Eval c_d ( c_s , c_d )
        if c_s = GOOD
            do the evaluation rules using b_d, c_d, and d_1 .. d_n
            if successful
                d_d ← the result of the evaluation rules
                status ← GOOD
            end if
        end if
    end if
    Mark State Variable d_s ( status )
end procedure
```

If synthesis is successful, the value of $d_d$ is updated and the associated state variable $d_s$ is marked GOOD, indicating $d_d$ conforms to the evaluation rules. If the synthesis is not successful the value of $d_d$ is undefined and the state variable is marked BAD.

The Put operators serve a dual role. First, they update the value of their respective attributes (recall, attributes are only accessible from outside the object environment through operators). Second, they cause the state variable associated with the derived attribute that uses the updated attribute in its evaluation rules to be marked BAD, indicating the derived attribute cannot conform to the rules. This marking requires no judgement as to whether the value of $d_d$ previously conformed to the rules or not, i.e., whatever its previous value, it cannot now conform to the rules.

```plaintext
procedure Put d_1 ( v )
    d_1 ← v
    Mark State Variable d_s ( BAD )
end procedure
```
The Mark State Variable $d$ operator updates the value of the state variable $d$ according to the values it receives from other operators.

```plaintext
procedure Mark State Variable $d$ (v)
    if $d$ = GOOD and v = BAD
        $d$ ← BAD
        Mark State Variable $e_{d1}$ (BAD)
    else
        $d$ ← BAD
    end if
end procedure
```

The first time the value of $d$ changes from GOOD to BAD, Mark State Variable $d$ causes the state variable for $e_{d1}$, $e_{s1}$ (which is the only derived attribute that is directly dependent on $d$) to be marked BAD. Thereafter, $e_{s1}$ need never be notified if other operators would cause $d$ to receive the value BAD, because $e_{s1}$ cannot have the value GOOD until $d$ itself has the value GOOD. In a similar manner, Mark State Variable $d$ updates the value of its state variable according to the values it receives from Mark State Variable $b$ and Mark State Variable $c$. The Mark State Variable $e_{d1}$ operator will, in turn, cause the state variables of $e_{d2}$ and $e_{d3}$ to be marked BAD the first time $e_{s1}$ changes from GOOD to BAD, thereby propagating the effect of $d$ violating the evaluation rules. This is essentially a tree walk algorithm that terminates at the first node already marked BAD, because no other dependent node can be GOOD.

Discussion

The Eval algorithm illustrates four characteristics of object-oriented operators. 1) The algorithm is non-procedural. The prerequisite derived attributes $b$ and $c$ need not be evaluated prior to invoking $eval d$ (although they may be). This is in contrast to procedural approaches that require prerequisites first be satisfied. 2) The external interface
is simple, i.e., Eval \( d_d (v_s, v_s) \). 3) Details are suppressed, i.e., the algorithm has no external visibility. 4) Objects cooperate with related objects.

The theoretical underpinnings of the eval algorithm is the theory of attribute grammars defined by Knuth [49,50]. An attribute grammar is a context free grammar \( G=(V_N, V_T, P, S) \) that is augmented with attributes, evaluation rules, and conditions. An augmented grammar permits meaning to be associated with syntax. Following the approach of Pagan [58], the augmented syntax tree for the derived attributes appears as Figure 5.12. Only attributes of one nonterminal node, \( < d_d > \), and one terminal node, \( d_1 \), are shown. All other terminal and nonterminal nodes have identical attributes. Associated

![Figure 5.12. Syntax tree for derived attributes.](image)

with each nonterminal node of the attribute grammar are the two attributes representing the state (GOOD or BAD) of the node and the value of the node; and two attribute evaluation rules, i.e., one rule for the marking phase of the tree walk and one rule for the synthesis of the derived attribute phase. The two auxiliary evaluation rules \( field_1 \) and \( field_2 \) extract components of the 2-tuple that is returned from the evaluation rule. No conditions are associated with the nodes. The terminal nodes have only one attribute that represents the
value of the node, and one evaluation rule for the marking phase. The attribute grammar
represented by the syntax tree can be evaluated using the method of recursive descent.

The tree structure of the example is not an accident, it is a design goal of transform-
centered modeling. One of the steps in the methodology is to identify the "central
transform" (figuratively) pick it up, and let the other "bubbles" hang down [59]. The
model that results is called a top-down model. The tree structure is neither mentioned, nor
is it exploited, in the methodology. The algorithm is not restricted to tree structures,
however. General graph structures are permitted because the marking algorithm does not
follow paths that have already been traversed.

Algorithm Analysis

The algorithm is demand driven. There is a marking phase and a synthesis phase. The
marking phase, which occurs whenever a non-derived attribute value is updated, causes all
directly or indirectly dependent derived attributes to be marked as not conforming to the
evaluation rules. Actually, the marking algorithm is better than that in so far as marking
stops when dependent attributes are encountered that are already BAD, because nothing that
depends on them can be GOOD. The synthesis phase, which occurs whenever a specific
derived attribute value is next read, results in only the specific attribute and possibly those
used in its evaluation rules to be synthesized. No other derived attribute values are
synthesized until there is a demand for them. The worst case cost of the marking phase is
$O(|\text{affected attributes}|)$, where $|\text{affected attributes}|$ represents the cardinality of the set that
is the transitive closure of the attribute dependencies starting at the node in the dependency
graph whose non-derived attribute value was updated. The worst case cost of the synthesis
phase is also $O(|\text{affected attributes}|)$. 
In actual practice, the algorithm performs much better than the worst case cost analysis, particularly when small portions of the dependency graph are synthesized. This is the case in many graphic workstation applications, where affected attributes can be substantial because a large menu of transformations are possible, but only a small number are in effect at any one time. Under these conditions both the marking phase and the synthesis phase have a cost of $O(|\text{demanded attributes}|)$, where $|\text{demanded attributes}|$ is the sum of the synthesized attributes starting at the node in the dependency graph whose non-derived attribute value was updated up to the node whose derived attribute value is read, and $|\text{demanded attributes}|$ is always $\leq |\text{affected attributes}|$.

Hudson and King [37] describe an algorithm with a worst case cost analysis of $O(|\text{important attributes}|)$ for the Cactis database management system. The Cactis algorithm has two marking phases, the first similar to that described here, the second identifies derived attributes that are important, where important is interpreted as having been recently read. All important derived attributes are synthesized when the non-derived attribute is updated. The algorithm described here does not require the second marking phase. The recursive nature of the synthesis is more straightforward than maintaining a work-list of attributes to be derived as does Cactis. It is particularly suitable for a graphics workstation because only the derived attribute of current interest (and those required for its derivation) is/are evaluated, rather than evaluating attributes that may no longer be of interest, but may have been in the recent past. In the situation where other attributes dependent on the newly derived attribute are later read the effect is to prorate evaluation in the interest of better response. When $|\text{demanded attribute}| = |\text{important attribute}|$, the cost analysis of the two algorithms are identical, although the algorithm described here is more straightforward.

A worst case algorithm would compute all derived attributes every time a change is made to any non-derived attribute using triggers attached to data. Except for very simple
systems, this is too expensive, having a worst case cost that is a function of the transitive closure of all dependencies starting at each node in the dependency graph.

5.6 Classes

Adding operator specifications to an object type defines a class. Classes have attributes and operators, collectively called properties [13,55]. A class, then is a special kind of type that has been augmented with operators. A class is fundamentally more powerful a concept than a type [79]. A class encapsulates both state and operators. A class is an abstract data type. An object is an instance of a class.

Objects, being an instance of a class, have state. The state of an object is only accessible through the operators of the class. A class assumes full responsibility for managing the state of its individual objects. Two kinds of operators have been identified, procedures and functions [55]. A procedure performs actions, which may cause a change of state. Because an object has state, procedures are not simple input/output mappings; rather, the actions of procedures are dependent on the past states of the object (and possibly other related objects). A function performs no actions. It returns a (possibly derived) value from the state of the object.
Chapter 6

Object-Oriented Languages

“If I hear the phrase ‘everything is an object’ once more, I think I will scream.”

Michael Stonebraker

Future Trends in Database Systems, IEEE
Transactions on Knowledge And Data Engineering, 1(1), March 1989

Just how bad is it? A comparison with other areas of computer science can be helpful. A (binary) tree is defined as having a root, a left subtree and a right subtree. Each subtree is again defined as having a root, a left subtree and a right subtree. A Pascal record can be defined as containing other (embedded) records. Neither of these seemingly circular definitions cause confusion. Recursive definitions are both concise and precise.

An object is the fundamental construct of object-oriented programming languages. An object encapsulates type and operators. This is in contrast to more traditional languages that separate type from operators. The notion of an object from the language perspective is as a bottom-up component used to construct more complex objects. Everything is an object. The notion of an object from the data modeling perspective is as a top-down component which has stepwise refinement by decomposition applied.

More fundamentally, however, the underlying goals of object-oriented data modeling and object-oriented programming languages are substantially different. The goal of object-oriented data modeling is to model the problem domain. An object is viewed as an abstract representation of an entity that exists in the mini-world being modeled.
Common type (i.e., data structure) is the conscious design decision used to model entities. The goal of object-oriented programming languages is modular software construction, code sharing, and code reusability. Common behavior of data is the conscious design decision used to form classes. Object-oriented data modeling and object-oriented programming languages approach the problem from different directions and with different goals.

Language Support For Object-Oriented Data Modeling Abstractions

Assumptions:

"A language supports a programming style if it provides facilities that make it convenient (reasonably easy, safe, and efficient) to use that style. A language does not support a technique if it takes exceptional effort or skill to write such programs"

Bjarne Stroustrup


It is unlikely that all abstractions will be supported by one language. Some of the abstractions will be manually translated into the programming language of choice.

Only object-oriented languages support the object-oriented paradigm.

The object metaphor, and the functional representation of attributes and operators, provide a common link between the data modeling abstractions and object-oriented programming languages.

Only the last three assumptions require clarification.

The fact that no DBMS or programming language will support all abstractions does not deter from their usefulness when describing the problem domain. Rather, it reinforces their need. To describe a problem domain in terms of the features of the implementation
language is as foolish today as it was in the 1960's, when Fortan arrays and Cobol records were the features of the day.

Peter Wenger, in an excellent article [80] classified languages that support object-oriented concepts as object-based (e.g., Ada, Modula-2), class-based (e.g., CLU), and object-oriented. Object-oriented languages were further classified as those that were strongly typed (e.g., Eiffel, Trellis/Owl), and those that were not (e.g., Simula, Smalltalk, C++) The general consensus from Wenger, and others [74] seems to be that it is extremely difficult to build object-oriented systems using languages that are not class-based or object-oriented; that the concepts of object-orientedness go much further than the encapsulation that is provided by languages such as ADA and Modula-2.

The object metaphor was used to describe data structure and operators on that structure. The object metaphor is used in object-oriented programming languages to encapsulate type and operators. The object metaphor is identical between the two. The functional representation of attributes and operators used in the data modeling abstractions translates directly to attribute and operator syntax of most programming languages. The data model representation of the problem domain can be translated into a computer program in a straight-forward manner.

**Objects**

The fundamental differences between the data modeling and programming language perspectives, then, is top-down refinement by decomposition with: *type* as the conscious design decision vs bottom-up synthesis with *behavior* as the conscious design decision. Additionally, objects (the software construct) in most object-oriented languages are abstract data types. The external interface to an object is the set of operations defined for it. Most languages enforce encapsulation by forcing external access through the operators.
The three levels of object-orientation identified by Dittrich (ref: 5.2) were (a) structural object-orientation, (b) operational object-orientation, and (c) behavioral object-orientation. The data modeling concepts that have been presented are within the range of (a) and (b). Dittrich claims that object-oriented programming languages concentrate on (c) and to a lesser extent (b). Specifically, object-oriented programming languages include features to define new object types of any complexity, together with their operators. Furthermore, new objects can be created (and disposed) dynamically during program execution, and, with polymorphic typing, can change (sub)class binding during program execution. These concepts enhance the concepts of (a) and (b) in a manner far superior to the static concepts available through the object-based encapsulation languages.

Relationships

Relationships between objects, one of the fundamental abstractions of data modeling, are not directly supported by object-oriented languages\(^1\). This is an inherent weakness resulting from the fact that each object is treated as a self-contained unit of information. Relationships, instead, must be embedded into applications code. This is normally done in one of two ways, interobject references or relationship classes.

In general, embedding interobject references (i.e., attributes that refer to other objects) into applications code contaminates objects with knowledge of other objects, corrupting the principles of encapsulation and abstraction. Embedding relationships that have attributes into code also violates the functional dependencies of third normal form. Even then, interobject references can only be used for one-to-one and one-to-many relationships. Many-to-many relationships must be externalized as relationship classes.

---

\(^1\)Instantiation and subclasses, the only relationships directly supported, have syntax and semantic language support.
A relationship class elevates the relationship between objects to a separate and distinct object in the same manner as the aggregate-constructor elevates the functional relationship between entities to an entity (3.3.2). The same approach is used in relational systems. Rumbaugh [62] makes a strong case that relationship classes are an implementation tool; that they do not raise the abstraction to the same semantic level as class hierarchies, which have language support with built in syntax and semantics. The abstraction gets lost in the implementation details because there is no language support, e.g., the semantic guards necessary to avoid interobject references to nonexistent objects must be embedded into code. This again corrupts the principles of encapsulation and abstraction. (The same problem exists when more traditional languages are used.)

Given the inherent weakness of object-oriented languages in the support of relationships, they are not particularly well suited for applications with many user defined relationships. For example, the relationships used in enterprise modeling are user defined, e.g., employees being assigned to departments or projects is a policy decision of the enterprise that could change. Object-oriented languages are better suited for applications with tightly coupled relationships between objects where the concept of state (output is dependent on current and past inputs) is important. Tightly coupled relationships are more likely to remain stable for a longer period of time, more so than user defined relationships.

Attributes

Object-oriented languages support most data structuring abstractions for attributes. Simple attributes, composite attributes, and multivalued attributes can be represented in code through the definition of operators and type. Virtually every language provides language support for shared-value class attributes and default-value class attributes.
Specialization Hierarchies

The premier example of economies of expression and conceptualization that are achieved with object-oriented programming languages is specialization hierarchies. Even the premier example, though, is not without problems and design tradeoffs [70]. The simplest counter example that demonstrates a conceptual gap between the problem domain and its object-oriented language representation is one of overlapping subclasses. An object in all object-oriented languages can be a member of only one class. This forces artificial classes to be constructed to represent relatively simple abstractions. Consider, for example, a university where a person can be an employee, a student, or an alumnus. Object-oriented languages require four classes to represent the disjoint subclasses of Figure 6.1(a), i.e., Person, Employee, Student, and Alumnus. They require eight classes to represent the real-world situation where the subclasses overlap (Figure 6.1(b)), i.e., the four already identified plus EmployeeStudent, EmployeeAlumnus, StudentAlumnus, and EmployeeStudentAlumnus.

![Figure 6.1. Possible specialization hierarchies for a university.](image-url)

The additional classes necessary to represent Figure 6.1(b) widen the conceptual gap between the problem domain and its language representation in the same manner higher
forms of (relational) normalization results in a collection of highly fragmented relations less closely related to the users conceptualization of a system.

A concept known as a partial class is another example where the conceptual gap between the problem domain and its language representation can widen. A frequently occurring software construct in object-oriented languages is to create a partial (super)class by factoring out common attributes and behavior from a group of subclasses. The (super)class by itself is (conceptually) a partial class in so far as it is an incomplete description of any useful or recognizable object in the problem domain. The subclasses add attributes and (possibly) additional behavior that turn the partial (super)class into something useful.

The concept of default and selective inheritance again widens the conceptual gap between the problem domain and its language representation. While different languages allow varying degrees of control over inheritance, most provide features for constructing artificial classes (they do not represent any real object) that can achieve the desired result. This, by the way, reinforces the need for identifying modifiers for inherited attributes as public, private or exclude.

Strict encapsulation is a design tradeoff that is common to all object-oriented languages. Many languages (e.g., Smalltalk, Flavors, Objective C, Loops) allow free access to inherited attributes by its subclasses, i.e., the internal interface presented to the subclass hierarchy is much less restricted than the external interface presented to other objects. This permits a language to support a knowledge representation style of programming, but violates the principles of encapsulation. Other languages (e.g., Common Objects, C++, Trellis/Owl) force access to inherited attributes through the operators defined for the superclass. This isolates responsibility for the state of the object, and is more in line with the principles of encapsulation and abstraction, but may not be appropriate for some problem domains.
Composite Objects

Object-oriented programming languages support composite objects through interobject references. This support does not conflict with the principles of encapsulation, because these interobject references are merely complex attributes; the composite object performs no operations on them nor mediates their behavior in any manner. This is consistent with the drawing notation introduced for composite objects in Section 3.5.1.

Part Hierarchies

No language, object-oriented or otherwise, (except Loops) supports part hierarchies. Part hierarchies are implemented using forward interobject references from assemblies to parts and backward interobject references from part to assembly (i.e., the relationship and the inverse relationship). Parts "know" what changes are significant to the assembly, and the assembly has "intimate knowledge" of the details of the parts and mediates their behavior. When an object is assembled in this manner the parts are no longer independent and the principles of encapsulation are severely compromised [9]. Even with these limitations, however, an object-oriented language should be the language of choice because of the tightly coupled nature of the relationships between assemblies and parts and the importance of the concept of state that is inherent in these applications.

Versions

No object oriented programming language supports object versions. Versioning has less to do with an object-oriented (or any other kind of) data model and more to do with the kinds
of applications being modeled. Versioning would seem to very application specific, and not the kind of feature supported by a language.

**Integrity Rules**

There is no support whatsoever for the kind of integrity rules described in Section 4. These must all be implemented in code. As pointed out, however, the integrity rules to be implemented are inherent in the structure of the data model, so the data model provides guidance. Without this data model intimate knowledge of the problem domain is required so as to identify the integrity rules that should be embedded into code.

**Operators**

All object-oriented languages permit a subclass to override an inherited operator. Most provide for augmentation of an inherited operator with built-in language syntax that causes the superclass operator to be invoked from the subclass. Virtually no object-oriented language permits a class to exclude an inherited operator from its own external interface. Some languages permit operators to be public or private.

All object oriented languages provide limited language support for operators to deal with objects in their entirety, e.g., all superclass attributes are accessible. This goes beyond the what is available with more traditional languages. The concept of Create/Delete/Update semantics, however, is foreign to object-oriented (and more traditional) languages and must be embedded in code.

No object-oriented language (except Loops) supports part hierarchies, the abstractions of dynamic / static propagation attribute constraints or none / shallow / deep propagation attribute values must be embedded in code.
Chapter 7

Conclusions

A comprehensive set of data structuring abstractions were presented. The first abstraction
was that of an object being an entity with a type that is not derived from the type of any
other entity. Attribute assignments to a small number of objects proved to be superior than
the traditional approach of collecting attributes into a data dictionary, identifying functional
dependencies, and synthesizing (a large number of) tables. The data model remained closer
to the problem domain than the highly fragmented entity relationship or relational models.

The assignment of attributes to objects was represented as functions. This added
semantic information that is missing from the ER and relational models. Attribute functions
that are total identify attributes that cannot have a null value; something that will be enforced
in either the data description language of an underlying data base management system, or in
the applications code. Attribute functions that are total, 1:1 identify candidate keys for a
relational implementation. The abstraction of aggregating simple attributes into composite,
multi-valued, and derived attributes kept the data model close to the problem domain rather
than the implementation (i.e., data base) domain. The abstraction of class attributes
modeled real-world situations that are difficult to represent in the relational model. None of
these abstractions conflicted with the underlying theory of the relational model, nor did they
add any additional power. They served to keep the data model close to the problem domain
and in a form that can readily be translated into a relational (or other) implementation.

Specialization hierarchies (or lattices) modeled roles for an object. Two independent
reasons for specializations were presented; i.e., there are attributes relevant to the
specialized entity that do not apply to the more general entity, and/or, the specialized entity can participate in relationships that the more general entity cannot. Three forms of specialization were identified, i.e., attribute, predicate, and user defined. The need for public, private, and exclude constraints to restrict attribute visibility in a specialization hierarchy was identified. Composite objects extended the object abstraction to include objects that represented collections of other objects. Part hierarchies added another dimension to composite objects by addressing hierarchically organized collections of interrelated objects. Multiple object versions completed the structural abstractions.

The use of functions and set constraints identified integrity rules for Create/Delete/Update operators. The integrity rules are inherent in the functional representation of the data model. They require no special knowledge of the problem domain. The procedure for identifying integrity rules can be automated if a data description language is defined that represents the data structure and relationships.

A graphical notation was defined for representing these abstractions using fourteen symbols. There are five symbols to represent objects, subclasses, derived and non-derived attributes, and classes; five function symbols to represent relationships of inheritance, attribute functions, ordered functions, functional relationships, and parts relationships; two constructor symbols (set-constructor and aggregate constructor); and two set constraint symbols (disjoint and union). The graphical notation permits the definition of structures of any complexity, while at the same time keeps the data model conceptually close to the problem domain. The data model was structural object-oriented at this point.

An approach for identifying behavior was defined. The need for public, private, and exclude inheritance modifiers to restrict operator visibility was identified, as well as propagation constraints and propagation attribute values in a parts hierarchy. Operators dealt with objects in their entirety; were close to the problem domain; had a broader meaning than the tuple selection, projection, and join operators of the relational model; and were constrained to not violate the Create/Delete/Update constraints of the integrity rules.
Encapsulating operators with object type defined classes. Classes elevated the data model to *operational object-orientation*.

Objects were seen to be active in responding to changes in their environment, rather than passive data stores. A form of functional relationship between derived attributes that can be modeled using a transform-centered approach was used to illustrate the dynamic nature of objects. This form is common in scientific, engineering, CAD/CAM, and statistical applications. The data flow diagram of the transform-centered model added more substance to the data model than dependency diagrams. The relationships between derived attributes was formalized using an attribute grammar. Attribute grammars can be translated into subroutine or function calls in most programming languages in a straight-forward manner. Being able to represent derived attribute dependencies using an attribute grammar is a beneficial side effect to the tree structure that results from transform-centered models.

An algorithm was presented for implementing the attribute grammar. The algorithm was an improvement over previously published results. It is particularly well suited for a graphics workstation where a large menu of derived attributes may be possible, but only a small number actually in effect.

Object-oriented programming languages were examined. They were found to approach the problem domain from a different direction and with different goals. While these languages provided the best support for object-oriented abstractions, the language constructs necessary to implement some of the abstractions were found to widen the conceptual gap between the problem domain and its language representation. Table 7.1 provides a side by side comparison of how the structural abstractions presented in this thesis are supported by object-oriented programming languages.
Table 7.1
Comparison of Object-Oriented Data Models and Languages

<table>
<thead>
<tr>
<th>Object-Oriented Data Models</th>
<th>Object-Oriented Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uses the object metaphor</td>
<td>Uses the object metaphor</td>
</tr>
<tr>
<td>Functional representation for attributes and relationships</td>
<td>Functional syntax for attributes and relationships</td>
</tr>
<tr>
<td>Top-down refinement by decomposition</td>
<td>Bottom-up construction</td>
</tr>
<tr>
<td>Data type is the conscious design decision</td>
<td>Behavior is the conscious design decision</td>
</tr>
<tr>
<td>Structural object-orientation</td>
<td>No structural object-orientation</td>
</tr>
<tr>
<td>Operational object-orientation</td>
<td>Partial operational object-orientation</td>
</tr>
<tr>
<td>No behavioral object-orientation</td>
<td>Behavioral object-orientation</td>
</tr>
<tr>
<td>Models relationships</td>
<td>Embeds relationships into code</td>
</tr>
<tr>
<td>Identifies composite, multivalued, and derived attributes</td>
<td>Embeds composite, multivalued, and derived attributes in code</td>
</tr>
<tr>
<td>Identifies non-null attributes</td>
<td>Code guarantees non-null attribute values</td>
</tr>
<tr>
<td>Identifies shared-value and default-valued class attributes</td>
<td>Embeds shared-value and default-valued class attributes in code</td>
</tr>
<tr>
<td>Identifies subtype hierarchies in a manner close to the problem domain</td>
<td>Embeds subtype hierarchies in code in a manner removed from the problem domain</td>
</tr>
<tr>
<td>Identifies public, private, and excluded attributes and operators</td>
<td>Embeds public, private, and excluded attributes and operators in code</td>
</tr>
<tr>
<td>Models composite objects</td>
<td>Embeds composite objects in code</td>
</tr>
<tr>
<td>Models part hierarchies</td>
<td>Embeds part hierarchies in code</td>
</tr>
<tr>
<td>Models object versions</td>
<td>Embeds object versions in code</td>
</tr>
<tr>
<td>Identifies operators</td>
<td>Embeds operators in code</td>
</tr>
</tbody>
</table>
Table 7.1 shows that there is virtually no overlap, that data modeling and object oriented languages are complementary. Merging the operational object-oriented concepts of the data model with the behavioral object-oriented concepts of object-oriented programming languages elevates the system to fully object-oriented.

"Object-oriented development is only a partial-lifecycle method and so must be coupled with compatible requirements and specification methods."


Object-oriented data modeling is a necessary specification method that must precede the bottom-up approach of object-oriented languages.
Bibliography


