CHAPTER 1

Introduction to Object Databases

This monograph covers *object databases* (ODBs), which is a term that refers to databases with object features. Historically, object-oriented databases (OODBs) developed first as an approach to add persistence seamlessly into object-oriented programming languages (OOPLs). In response to the development of OODBs, the relational database community developed object-relational databases (ORDBs), which extend the relational data model with support for many of the similar object-oriented concepts. ORDBs blur the distinction between object-oriented and relational databases. Thus, the term object databases refers to OODBs and ORDBs collectively.

This chapter first covers a historical view of object databases, providing a context in which to understand the motivation for ODBs. The fundamental concepts of object features are then reviewed briefly, since it is assumed that the reader is familiar with an OOPL. The chapter concludes with coverage of the conceptual modeling of ODBs to provide the basis for introducing case studies that illustrate the use of object features within the design of object-oriented and object-relational databases. This design perspective emphasizes class hierarchies and the specialization constraints that need to be considered when modeling enterprises that are object-based. The constraints on the class hierarchies are presented with side-by-side Enhanced Entity-Relationship (EER) diagrams used by the database community and the Unified Modeling Language (UML) conceptual class diagrams used by software engineers. These conceptual models are used extensively in Chapters 2 and 3 to illustrate the mapping of the conceptual design of the database to its implementation data model, such as OODBs in Chapter 2 and ORDBs in Chapter 3. For completeness, Appendix A includes the mapping of EER and UML diagrams to the relational data model.

1.1 A HISTORICAL VIEW OF OBJECT DATABASES

For many years, relational database systems (RDBs) were well-suited for a wide range of business applications that effectively represented the enterprise of interest using a collection of tables. As interest in the use of database technology began to spread, relational database systems were used for a variety of new applications. These new applications included areas such as engineering design, geographic information systems, and telecommunication systems. A common characteristic of these new applications involved the use of large amounts of complex data, which is challenging to represent since the value of relational attributes are typically restricted to be simple, atomic values.

Although Appendix A illustrates how the enterprises represented by the EER and UML object-oriented conceptual data models can be mapped to the relational data model to represent the data and constraints of the underlying application, the discussion does not address whether the approaches to representing objects in a relational data model are efficient. The view of an object in the relational model requires the computation of joins over the various tables that may describe the object's properties, including the inherited ones. This join computation may require accessing multiple tables that may not be physically stored near each other on disk. In an OODB, the properties of an object are typically clustered together on disk so that when an object is retrieved, all of its properties are accessible in memory. Another efficiency concern for RDBs versus OODBs is based on the navigation of associations. In a RDB, associations are by value, requiring a join on the value given by the association. In an OODB, associations are inherently represented by object references, which are an integral component of the underlying database system. Therefore, the navigation through associations between objects may be more efficient in OODBs than the simulated navigation using joins in RDBs. Therefore, forcing such complex data into the atomic-valued requirement of the relational model may result in inefficient queries with numerous join conditions required to reconstruct complex objects.

Furthermore, these new applications often required the representation of non-traditional data forms, such as spatial, multimedia, or voice data. The relational data model, which restricted column types to be atomic values such as strings, integers, real numbers, and Boolean types, did not provide the type of extensibility needed to fully capture the data semantics of these new application domains. OODBs were developed in response to a need for managing data that did not fit well in the traditional table-oriented view of the relational model.

The development of OODBs in the mid-1980's was also influenced at the time by the growing interest in OOPLs. OOPLs such as Simula and C++ had emerged on the programming language scene, providing a new approach to software development that emphasized the use of objects, with encapsulation of object structure *and* behavior through the use of abstract data types. With similar interests in modeling complex, persistent data as objects within the database community, the merger of object-based persistent data with OOPLs provided the promise of a new database paradigm for the efficient representation of large, complex data sets together with the extensibility offered by the programming language capability of user-defined abstract data types. A major advantage of the object-based data/language merger of the OODB paradigm, however, was resolution of the *impedance mismatch* problem, thus providing seamless integration of the database with the programming language. The impedance mismatch problem refers to the differences that have traditionally existed between the set-oriented, declarative database approach to relational data access and the one-record-at-a-time, imperative programming language approach to data access. These different styles of data access coupled with differences in data types between the relational database system and application programming language causes data translation and conversion problems for data-intensive applications. OODBs do not suffer from the impedance mismatch problem because the computationally-complete programming language *is* the database language. Furthermore, the

database and programming language data types are aligned into a consistent view of objects as instances of abstract data types. The details of OODBs are discussed in Chapter 2, describing the Object Data Management Group standard and a case study of the LINQ (Language INtegrated Query) object query language with the db4o open-source OODB.

In response to the development of OODBs, the relational database community developed ORDBs, which extend the relational data model with support for many of the similar object-oriented concepts. Object-relational features were first introduced in the SQL:1999 version of the standard. The details of ORDBs are discussed in Chapter 3, describing the object-relational features of the SQL standard along with a case study of object-relational features in Oracle.

1.2 FUNDAMENTAL CONCEPTS

The goal of an ODB is to provide support for the persistence of objects, while supporting the myriad of features expected of a database system. Some of these expected database features include: the efficient management of persistent data; transactions, concurrency and recovery control; and an ad hoc query language. The challenge for an ODB is to provide these database features in the context of the complexities introduced by object-orientation.

An *object* is an abstract concept, generally representing an entity of interest in the enterprise to be modeled by a database application. An object has *state* and *behavior*. The state of an object describes the internal structure of the object where the internal structure refers to descriptive properties of the object. Viewing a person as an object, the state of the object might contain descriptive information such as an identifier, a name, and an address. The behavior of an object is the set of *methods* that are used to create, access, and manipulate the object. A person object, for example, may have methods to create the object, to modify the object state, and to delete the object. The object may also have methods to relate the object to other objects, such as enrolling a person in a course or assigning a person to the instructor of a course. A method has a *signature* that describes the name of the method and the names and types of the method parameters. Objects having the same state and behavior are described by a *class*. A class essentially defines the type of the object where each object is viewed as an *instance* of the class. A method is a specific implementation of a method signature.

Given this fundamental definition of an object, the following object-oriented concepts are typically associated with an ODB:

- complex objects
- object identity
- encapsulation
- extensibility
- class hierarchies and inheritance
- overriding, overloading and late binding

The ability to define *complex objects* from simpler ones is an important property of object-orientation. Complex objects are defined using constructors, such as a tuple constructor that combines simple objects to create a high-level form of an object. Using an engineering design example, an airplane can be viewed as a higher level object that is constructed from lower level objects, such as the airplane body, the wings, the tail, and the engine. Each of these objects can, in turn, be complex objects that are constructed from other simple or complex objects. Other examples of constructors include collection constructors that define sets, bags or lists. Sets are unordered collections of elements without duplication. Bags are unordered collections that may contain duplicate elements. Lists are a sequenced or ordered collection of elements. A plane, for example, has two wing objects, which can be represented as a set. If the seats inside of the plane are viewed as objects, then a list can be used to represent the ordered sequence of seats. ODBs provide inherent support for the construction of complex objects.

Objects also have *object identity* through an internally assigned *object identifier (oid)*. Unlike keys in the relational model, an oid is *immutable*, which means that once an object is created, the oid remains invariant during the lifetime of the object. In contrast, the state of the object is *mutable*, meaning that the values of the properties of an object can change. The database system uses the oid for references between objects to construct complex objects. Values of properties within the state of the object are not used to uniquely identify the object in the database. Therefore, changing the value of a property of an object does not change the object's identity. This is in contrast to the relational data model, which uses the value of the attributes that form a candidate key of the table to uniquely identify a tuple in the table. If the value of the attributes that form a candidate key change in the relational data model, then every value-oriented reference to the tuple must be updated.

The term *encapsulation* refers to the ability to create a class as an abstract data type, which has an interface and an implementation. The interface defines the behavior of the abstract data type at a conceptual level. The implementation defines how this abstract behavior is realized using the programming language. Using the concept of encapsulation, the implementation of a class can change without affecting the interface that the class provides to the rest of the application. Encapsulation therefore supports an important software engineering concept of separating class specification from class implementation. ODBs support encapsulation through the specification of user-defined types.

The ability of an ODB to support encapsulation also contributes to its *extensibility*. With extensibility, there is no distinction between the usage of system-defined types versus user-defined types. In other words, users can create new types that correspond to the semantics of the application and then use these new types in the same manner as system-defined types. Extensibility is an appealing feature to non-traditional database applications that require the use of types other than the base types provided by the database management system. RDBs originally provided a system-defined table as an unordered collection of tuples of simple types, where operations on tables are theoretically captured by the relational algebra operators and pragmatically realized by SQL. OODBs provide a flexible approach to extensibility through the use of user-defined types via the OOPL.

Since the development of OODBs, RDBs and ORDBs have also been extended with the capability to define new types.

Another capability of object-orientation is the power to express *class hierarchies* and the *inheritance* of state and behavior. Later in this chapter, a review of the EER and UML object-oriented conceptual data models discusses the details of how these models support inheritance. Appendix A illustrates how the object-oriented conceptual data models can be mapped to a relational schema using tables, views and constraints to creatively capture the semantics of an object-oriented enterprise. ODBs, however, inherently support class hierarchies and inheritance as an integral component of the data definition language.

Overriding, *overloading*, and *late binding* are additional characteristics of object-orientation, which are provided by ODBs. When defining the behavior of class hierarchies, a default behavior for a method name can be specified at the superclass level and redefined with a specialized behavior for the subclass. This redefinition is known as *overriding* since the redefinition at the subclass level overrides the default behavior at the superclass level. The term *overloading* refers to a concept that allows a single class or multiple classes to have methods with the same name. The methods are distinguished by the number and types of the parameters. Overloading is used together with the concept of *late binding*, which means that the translation of an operation name to its appropriate method implementation must be resolved at run-time based on the type of an object.

The next section covers two fundamental object-oriented conceptual data models that graphically describe the data to be stored in an object database. Both the established EER diagrams used by the database community and the UML conceptual class diagrams used by software engineers provide advanced semantic modeling features that address class inheritance and constraints on subclass membership.

1.3 OBJECT-ORIENTED CONCEPTUAL MODELING

Conceptual modeling is the process of developing a semantic description of an enterprise that is to be captured in the design of an application. The Entity Relationship (ER) model has been one of the most well-known techniques associated with conceptual database design. Introduced in 1976 by Peter Chen, the ER model provides a database-independent approach to describing the entities involved in a database application, together with the relationships and constraints that exist between such entities. The ER model has evolved into the EER model, which enhances the original ER model with advanced features for conceptual modeling of class hierarchies. Introduced in the 1990s, software engineering uses UML as a visual, object-oriented modeling language to document the structural and dynamic aspects of a software system, including many types of diagrams that capture static and dynamic interactions that are inherent in software. From a database perspective, UML *class* diagrams capture the static structure of the classes and the associations between classes that are similar to entities and relationships in EER diagrams. This section first presents a brief review of the notation for EER and UML class diagrams because dialects of these models exist throughout the literature. Then the object features supported by the conceptual modeling diagrams, such as class

hierarchies and specialization constraints, are described. The enterprises presented in this section are used in later chapters to illustrate how the object-based models are mapped to the implementation level.

1.3.1 REVIEW OF ER AND UML FUNDAMENTALS

An Abstract Enterprise that illustrates various features of the conceptual models is introduced to compare the terminology between ER and UML diagrams. Figure 1.1 provides an ER diagram of the Abstract Enterprise, while Figure 1.2 gives a corresponding UML diagram. The features of the conceptual models with respect to classes and associations will be discussed in terms of side-by-side diagrams that illustrate the similarity between the EER and UML features.

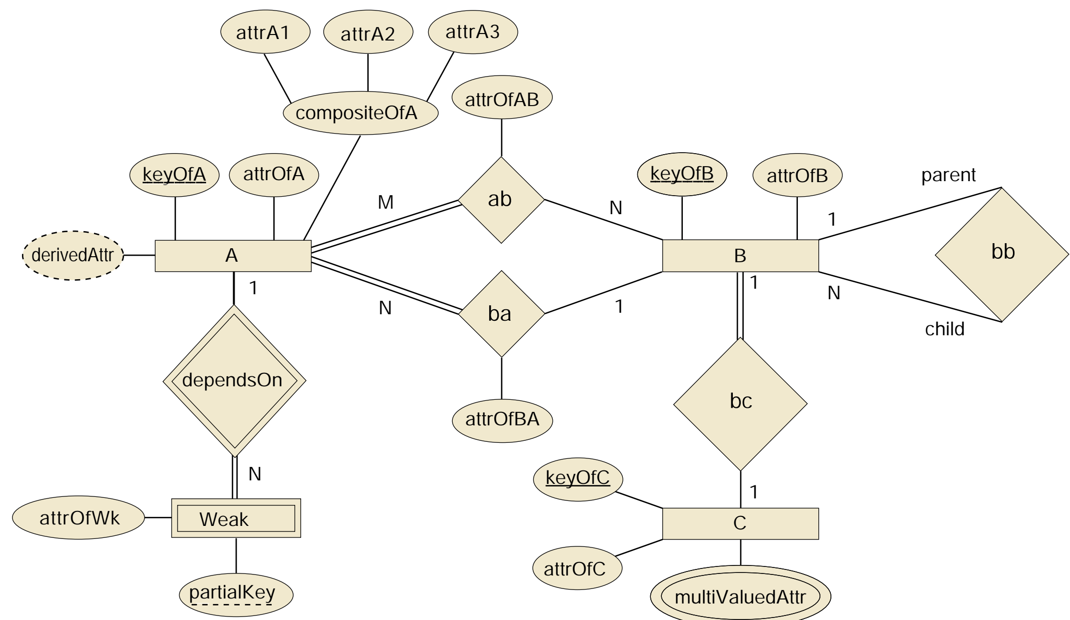

Figure 1.1: Abstract Enterprise in ER Notation with Cardinality Ratios

Figure 1.3 illustrates the class A that has simple attributes (keyOfA, attrOfA), a composite attribute (compositeOfA) and a derived attribute (derivedAttr). In the ER model, each entity is denoted as a rectangle, with the name of the entity, such as A, inside of the rectangle. The properties describing the class are enclosed in ovals and attached by a line to the entity that it describes. Attributes denoted by single ovals are single-valued attributes, indicating that only one value will be stored for the attribute. An underlined attribute name indicates the single-value or composite attribute that forms the *candidate key* of an entity, which uniquely identifies an entity instance in

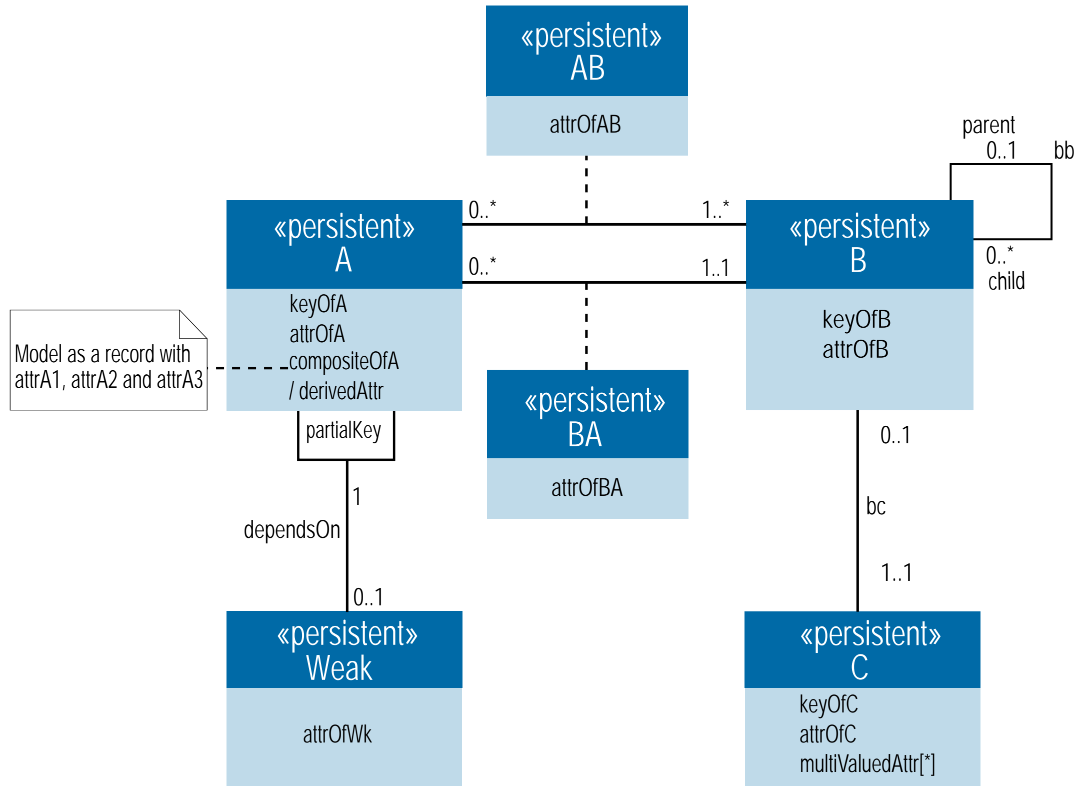

Figure 1.2: UML Class Diagram of the Abstract Enterprise

an entity set. A *composite* attribute has two or more single-valued attributes as subcomponents. A *derived* attribute, which is indicated by a dashed oval, represents a value that is not stored in the database but can be calculated by other stored values. In UML, a rectangle encloses the entire class. The first compartment gives the name of the class. The second component lists the names of the properties of the class. The behavior of the class is typically provided in a third component, which may be omitted for abstraction purposes as in the diagrams provided in this chapter. The `persistent` stereotype in guillemets (⟨⟨ ⟩⟩) above the class name identifies the class as a persistent class. A derived attribute in UML is annotated by a / preceding the attribute name. The dog-eared notes in UML are used to indicate composite attributes. There is no explicit annotation in UML for identifying candidate keys. However, the dog-eared notes can be used to indicate keys, too.

The class `C` shown in Figure 1.4 has simple attributes (`keyOfC`, `attrOfC`) and a multivalued attribute (`multiValuedAttr`). A multivalued attribute indicates that the entity can have more than one value for the attribute, where each value is of the same type. In an ER diagram, a multivalued

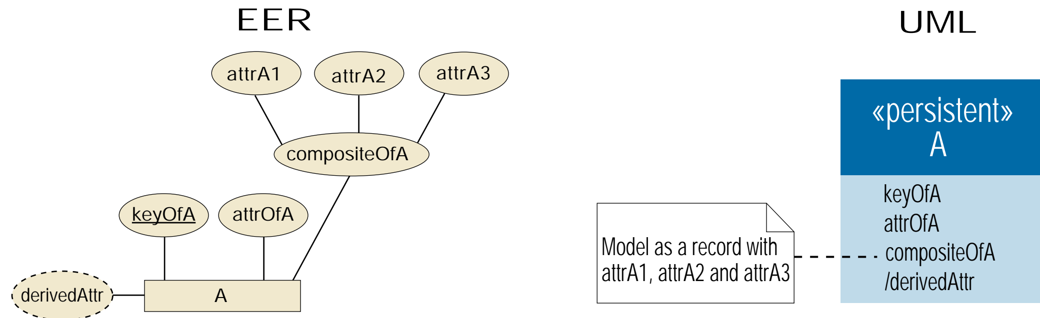

Figure 1.3: EER and UML Diagrams for Class A of the ABSTRACT ENTERPRISE

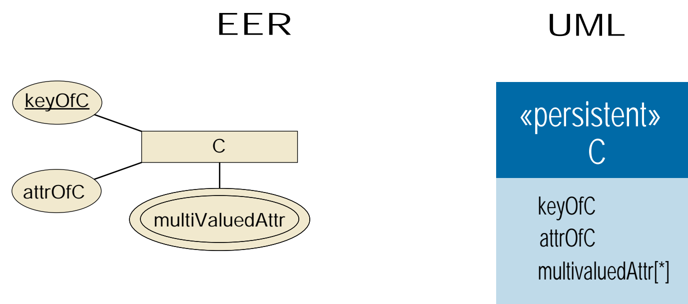

Figure 1.4: EER and UML Diagrams for Class C of the ABSTRACT ENTERPRISE

attribute is indicated by a double oval. In a UML diagram, a multivalued attribute is indicated by an array type property.

Interactions between classes are called *relationships* in an ER diagram and *associations* in a UML diagram. A relationship is denoted as a diamond, with lines connecting the diamond to the entities involved in the relationship. The name of the relationship appears inside of the diamond. In Figure 1.1, entities B and C are related through the bc relationship, while A and B are related through two separate relationships: the ab relationship and the ba relationship. The numbers indicated on the edges linking the relationship to its associated entities indicate the number of times that each entity potentially participates in the relationship. These numbers, typically indicated by 1 or N, are called *cardinality ratios*. The cardinality ratios define cardinality constraints on the number of entities that can participate in a relationship. Cardinality constraints are typically used in conjunction with *participation* constraints, which are either *partial* or *total*. Partial participation, which is denoted as a single line between a rectangle and a diamond, defines optional participation in a relationship.

Total participation, which is denoted as a double line between a rectangle and a diamond, indicates required participation in a relationship. In a UML diagram, an association is represented by a single edge linking the classes. The structural constraints on the associations are indicated by *multiplicities*, which are indicated by the *min..max* number of times that the class participates in the association.

A similar notation for providing more specific semantics on the participation constraints is available on ER diagrams using *(min, max)* pairs. Figure 1.5 shows the ABSTRACT ENTERPRISE in ER notation with (min, max) pairs. The pair labels the edge linking the entity to the relationship, indicating the *min* and *max* number of times that an entity instance participates in the relationship. The (min, max) pairs replace the use of cardinality ratios and participation constraints. A minimum value of at least one implies total or required participation in the relationship, whereas a minimum value of zero implies partial participation. This use is consistent with multiplicities in UML diagrams. One observation is that the placement of multiplicities in UML appears opposite from the placement of (min, max) pairs in the ER diagram because UML only uses an edge to represent an association. UML diagrams also provide for shorthand notations for some commonly occurring multiplicities. For example, 1 is the same as 1..1 and * is the same as 0..*.

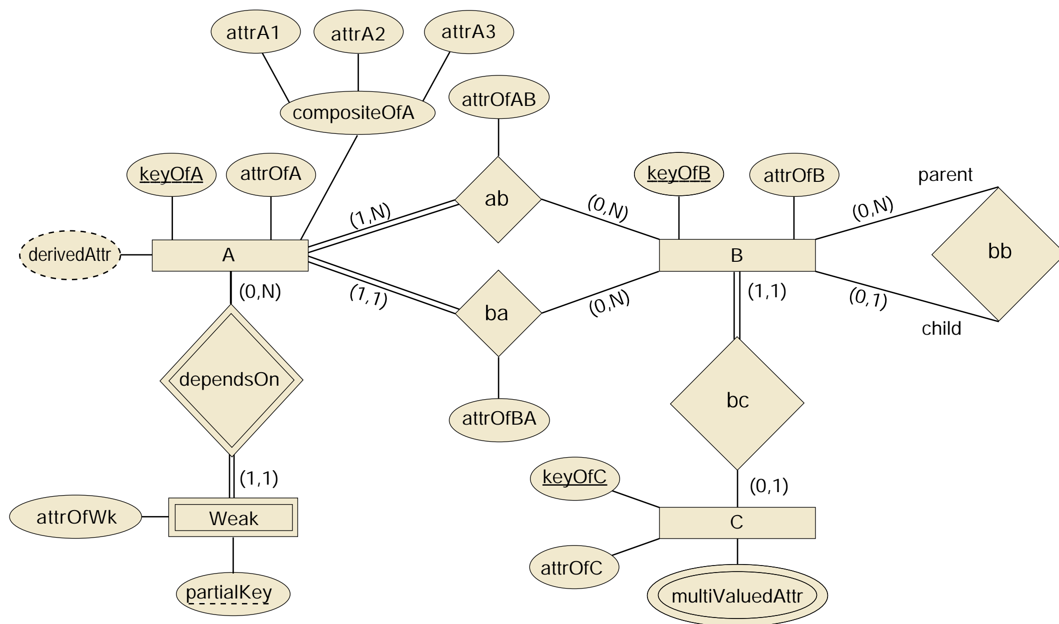

Figure 1.5: ABSTRACT ENTERPRISE in ER Notation with (Min, Max) Pairs

The ABSTRACT ENTERPRISE illustrates relationships having many-to-many, one-to-many, and one-to-one cardinality ratios. The ab association shown in Figure 1.6 represents a many-to-many relationship between classes A and B, where A has total participation in the ab relationship. An

A is related to at least one but potentially many B's, as indicated by the 1..* multiplicity on the UML diagram, and a B is related to potentially many A's, with a multiplicity of 0..*. The ba association of Figure 1.7 is an example of a one-to-many relationship between classes B and A, where A has total participation in the ba association. An A is related to exactly one B, with a 1..1 multiplicity, and a B is related to potentially many A's. The bc association shown in Figure 1.8 illustrates a one-to-one relationship between classes B and C, where B has total participation in the bc association. A B is related to exactly one C, as indicated by the 1..1 multiplicity, and a C is related to (at most) one B, as shown by the 0..1 multiplicity.

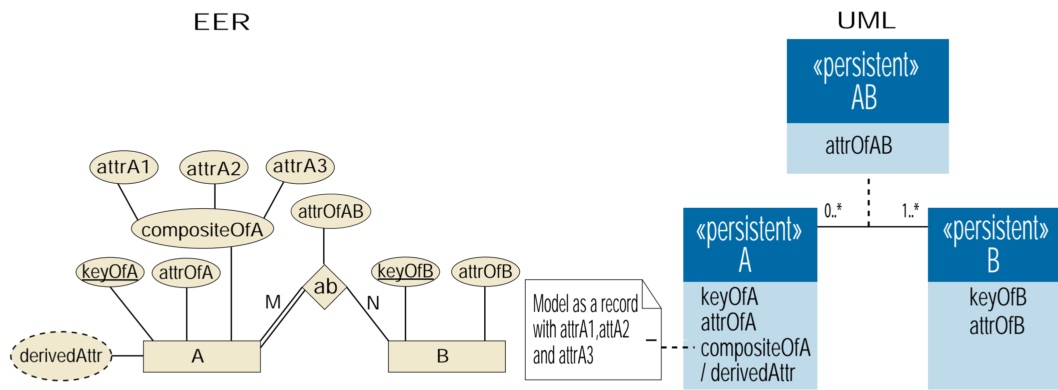

Figure 1.6: EER and UML Diagrams for M:N Association ab of the ABSTRACT ENTERPRISE

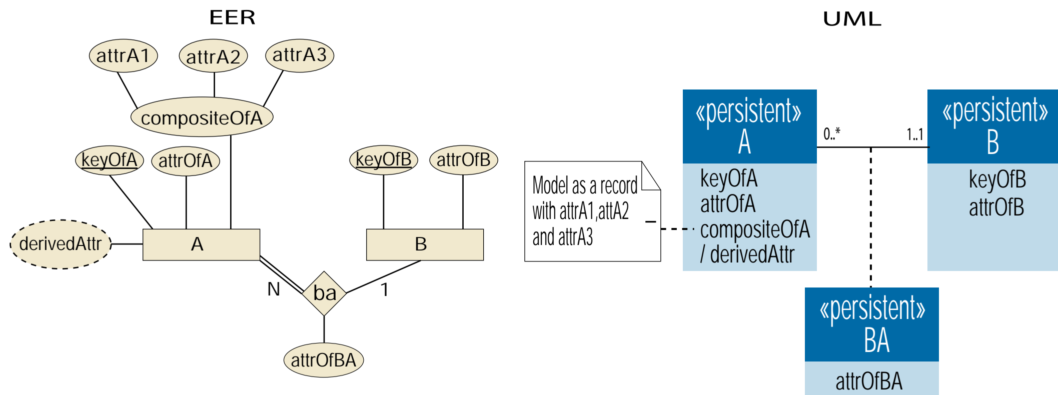

Figure 1.7: EER and UML Diagrams for 1:N Association ba of the ABSTRACT ENTERPRISE

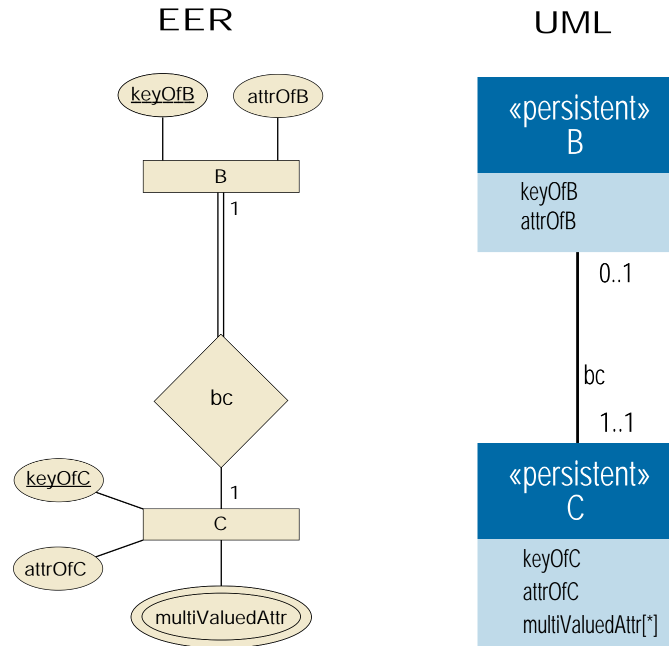

Figure 1.8: EER and UML Diagrams for 1:1 Association bc of the Abstract Enterprise

The ab and ba associations of Figures 1.6 and 1.7, respectively, illustrate descriptive attributes on the association itself. In an ER diagram, the attribute's oval is linked to the relationship diamond. In a UML diagram, however, descriptive attributes are represented by an *association class*. A class representing the association is created with the attributes describing the association and this association class is then linked to the association with a dashed line.

The Abstract Enterprise also represents a *recursive* association, in which a class has an association with itself. As an example, Figure 1.9 illustrates the bb association that is a recursive relationship on the class B. Each instance of B that participates in the relationship plays a specific *role* in the relationship. *Role names*, such as parent and child can be added as notations to associations to clarify the semantics of the relationship in both the ER and UML conceptual models. The instance of B as a parent is related to 0 or many (0..*) children, and an instance of B as a child is related to at most one (0..1) parent.

The associations that have been examined up to this point have been *binary* associations, defining a relationship between two entities. A relationship can define an *n*-ary association between several entities. Figure 1.10 illustrates a *ternary* finance relationship between three entities: Car, Person, and Bank. This association may exist in a car dealership application where a person buys a car that is financed by a particular bank. In an ER diagram, the relationship is linked to the three

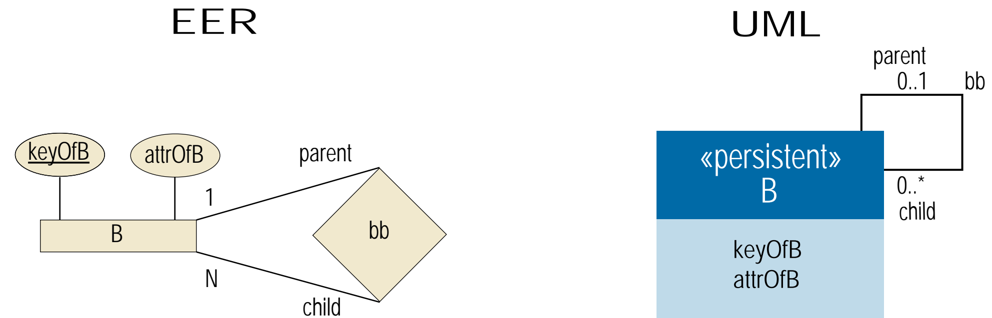

Figure 1.9: EER and UML Diagrams for the Recursive Association bb of the Abstract Enterprise

entities that it relates. In a UML class diagram, an n-ary association is modeled as a diamond with lines branching out to the classes involved in the association. If the association has attributes, as in the loanAmount in this example, then an association class can be attached to the diamond with a dashed line.

Determining cardinalities for a relationship that is not binary is more difficult than for a binary relationship. To determine the cardinality on the Car end of the relationship, consider how many times a specific (p, b) pair can be related to c, where c is an instance of Car, p is an instance of Person, and b is an instance of Bank. One (p, b) pair can be related to many c entities since a person can work with a specific bank to finance the purchase of many cars. A (c, b) pair, however, can be related only to one p entity, assuming that a car can only be sold and financed to one person. As a result, a 1 is placed next to the Person entity. Likewise, the 1 on the Bank end of the relationship states that a (c, p) pair can only be related to one b entity, indicating that the sale of a car to a person can be financed only by one bank. In general, the lines between an entity and a relationship in a nonbinary relationship can be labeled with a 1 to represent participation in one relationship instance or with a letter, such as M, N, or P, to represent participation in many relationship instances, creating many different relationship combinations.

A nonbinary relationship can always be modeled by introducing an entity to denote the relationship. Then a binary relationship is created for each entity involved in the n-ary relationship, linking it to the new entity. In the finance n-ary association, Finance is modeled as an entity, while Car, Person, and Bank have binary relationships with Finance. Figure 1.11 provides a view of this representation. This approach is also known as a *reified* association in UML and represents an alternative notation to the use of association classes. In a reified association, the association is modeled as a class. The process of transforming an association into a class is called *reification*. When reification is applied, something that is not usually viewed as an object in the application, such as an association between two classes, is modeled as a class.

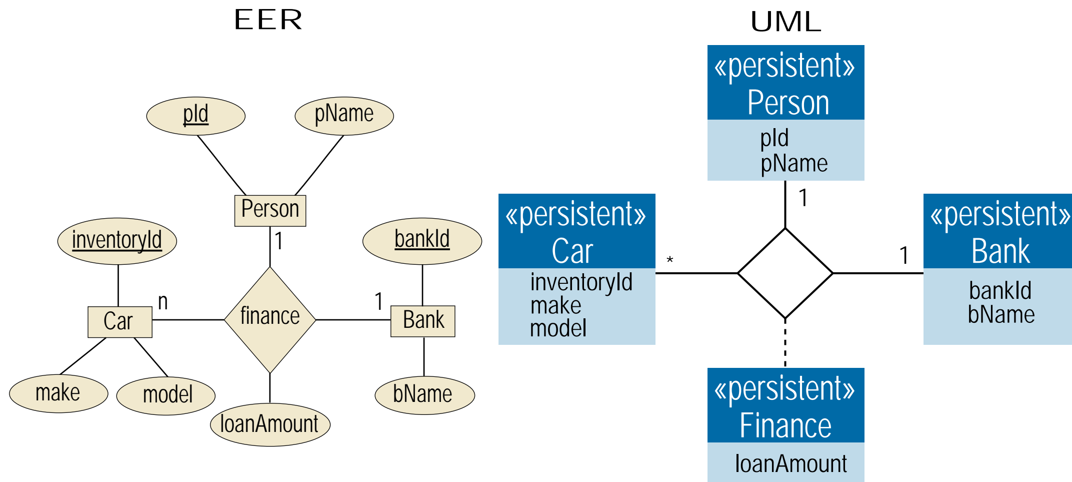

Figure 1.10: EER and UML Diagrams for the Ternary Association finance

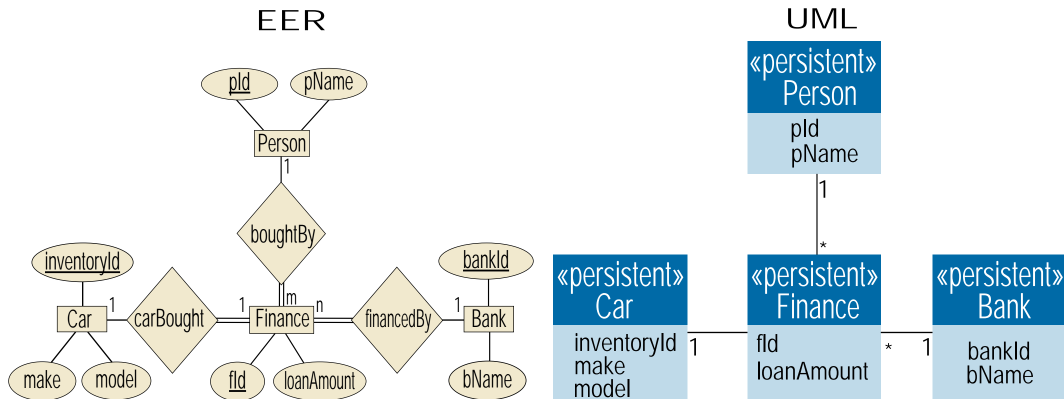

Figure 1.11: Binary Approach to Modeling the finance Relationship

The associations that have been presented so far are *bidirectional* associations. In a bidirectional association, it is assumed that the corresponding implementation of the schema will allow the user to traverse the association in either direction. ER diagrams only support bidirectional relationships. However, there is a concept of navigability of associations in UML class diagrams, allowing the designer to restrict the association to be *unidirectional*. This means that the implementation of the association is stored only in one direction. Figure 1.12 illustrates navigability in the context of the ABSTRACT ENTERPRISE where the 1:N ba association and the 1:1 bc association have

been changed to be unidirectional. The `ba` association is unidirectional from `A` to `B`, and the `bc` association is unidirectional from `B` to `C`. Since the navigability of associations provides semantics for the implementation of the model, their use will be more evident in later discussions involving mapping of the object-oriented conceptual models to the various implementation data models.

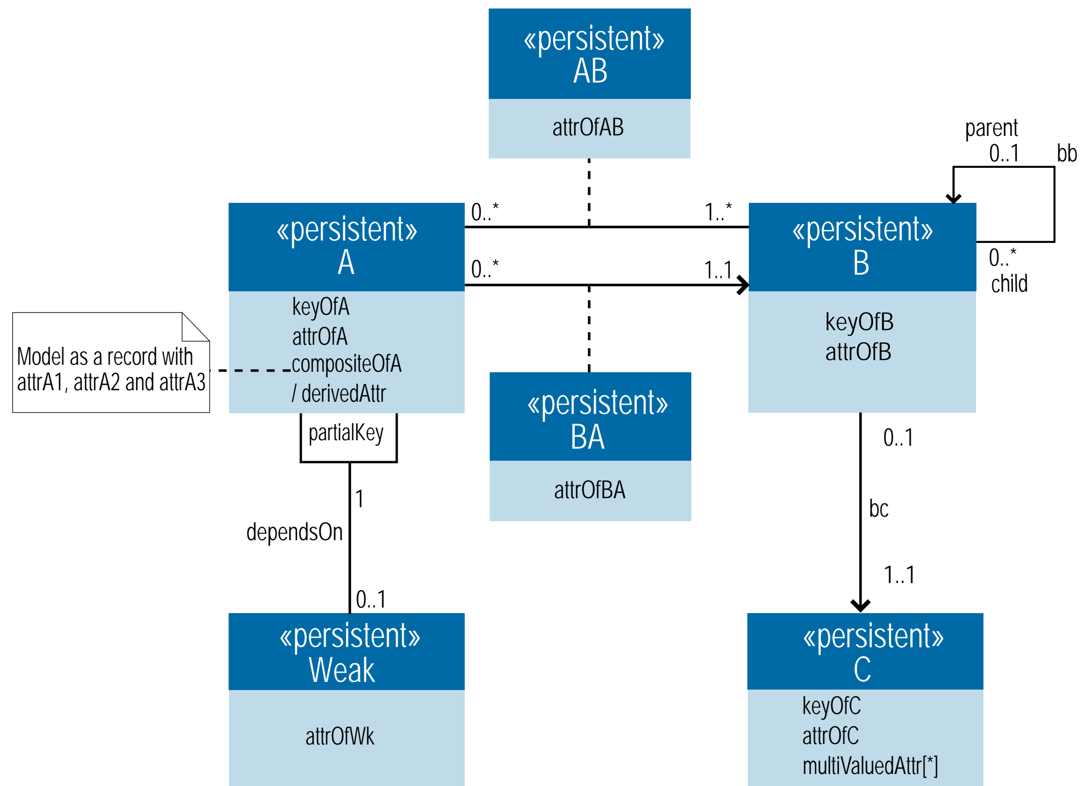

Figure 1.12: A Revised UML Diagram for the Abstract Enterprise Using Unidirectional Navigability

Figure 1.13 shows the weak entity, named `Weak`, that is denoted by a double-line rectangle in an ER diagram. The attributes of a weak entity do not uniquely identify an instance of a weak entity in the database. As a result, a weak entity participates in an *identifying* relationship with another entity, referred to as the *identifying owner* of the weak entity. The identifying relationship is denoted by a double diamond in the ER diagram, and the weak entity has total participation in the identifying relationship. A weak entity has a *partial* key, indicated by an attribute with a dashed underline. Semantically, a partial key uniquely identifies the weak entity in the context of its identifying owner. To create a candidate key for the weak entity, the partial key is combined with the

primary key of its identifying owner. In Figure 1.13, Weak has total participation in the dependsOn identifying relationship, indicating that A is its identifying owner. The candidate key of Weak is the combination of keyOfA and partialKey.

In a UML diagram, there is no explicit notation for representing weak entities. However, Figure 1.13 uses a *qualified* association. The partial key is used as a *qualifier* or index indicating the one A to which the Weak class dependsOn. Given a class A and the partialKey value, there is at most one related Weak instance.

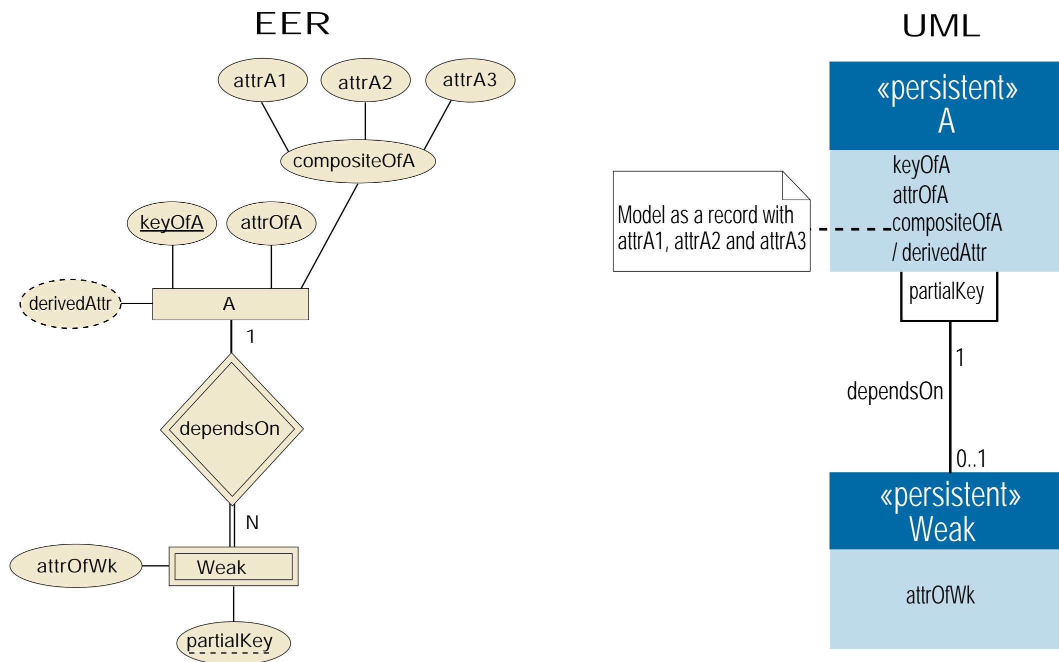

Figure 1.13: An ER Diagram for a Weak Entity and its Representation as a Qualified Association in UML

1.3.2 CLASS HIERARCHIES

EER and UML diagrams provide inherent support for modeling of classes into a hierarchical form known as a *class hierarchy* or *ISA hierarchy*. The classes can be formed into *superclasses* and *subclasses*, related through an ISA relationship. This ISA relationship is also known as *specialization* or *generalization*. Specialization emphasizes attributes and relationships of the subclasses that do not exist at the superclass level. Generalization emphasizes the common attributes and relationships of the subclasses to form a superclass.