Навчальна дисципліна

БАЗИ ДАНИХ

Лектор - к.т.н., доцент

Баклан Ігор Всеволодович

Site: baklaniv.at.ua

E-mail: iaa@ukr.net

2016-2017

Лекція №5.

Моделювання та проектування бази даних

- Database Model and Database Design
- The E-R Model Revisited
- Database Design With the E-R Model
- The Extended Relational Model
- Database Design With the Extended Relational Model
- The UML Model
- Database Design With the UML Model
- Innovation: The Object/Entity Specification Grid
- Database Design Using Normalization Theory
- Database Model and Design Tools
- Summary and Concluding Remarks

5.1 Database Model and Database Design

Database modeling and database design are closely related; in fact, the former leads to the latter. However, it is incorrect to assume that the path from database modeling to database design is an irreversible one. To the contrary, changes in your database model will affect your database design and vice versa. This course therefore purports that you can work on your database model and your database design in parallel, and with experience, you can merge both into one phase. For the purpose of discussion, let us look at each phase.

5.1.1 Database Model

The database model is the blueprint for the database design. Database modeling is therefore the preparation of that blueprint. In database modeling, we construct a representation of the database that will be useful towards the design and construction of the database. Various approaches to database modeling have been proposed by different authors; the prominent ones are:

- The Entity-Relationship (E-R) Model
- The Object-Relationship (O-R) Model
- The Extended Relational Model

The E-R and O-R models were introduced in chapter 3 (sections 3.5.1 and 3.5.2). Chapter 3 also introduced the Relation-Attributes List (RAL) and the Relationship List (RL section 3.6) as an alternative to the E-R model in situations where E-R modeling is impractical. We will revisit the E-R model later in this chapter, and then introduce the extended relational model.

5.1.2 Database Design

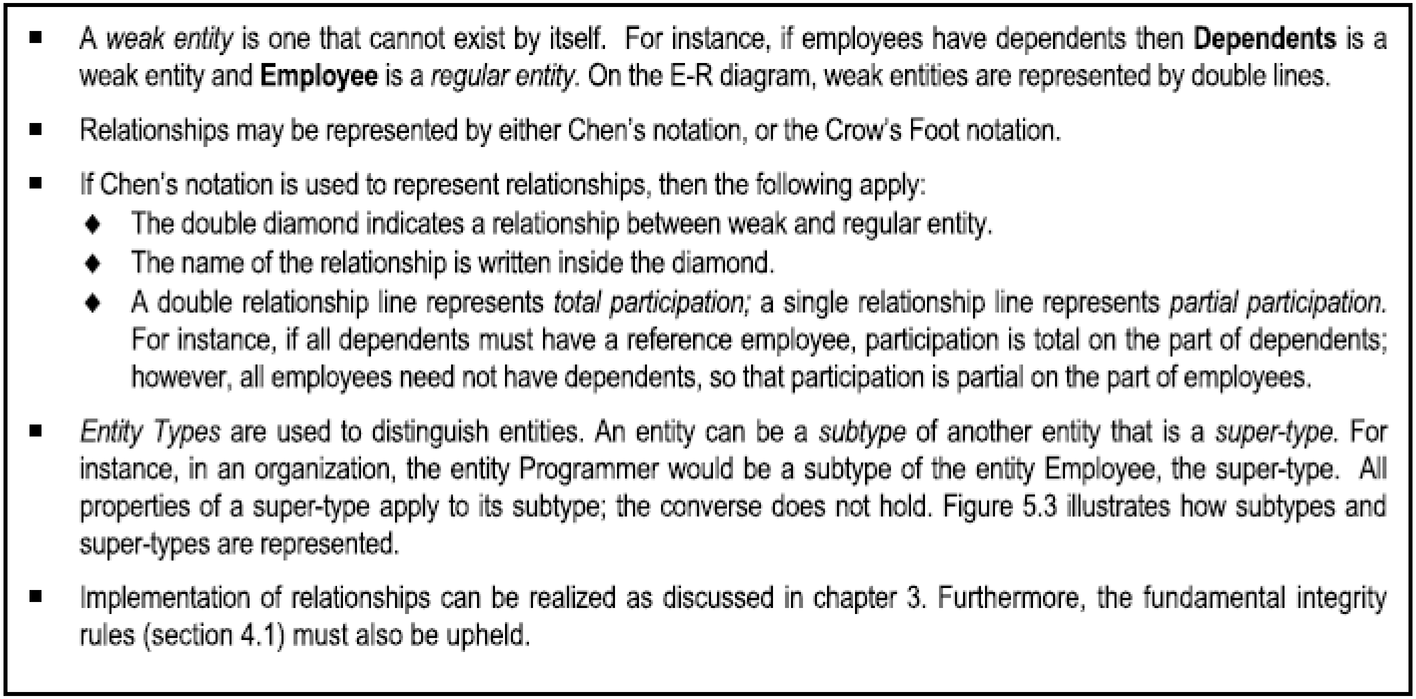
The database design is the (final) specification that will be used to construct the actual database. Database designing is therefore the preparation of this specification. In preparing the database specification, the database model is used as input. As such, the guidelines given in chapter 3 (section 3.5.4) on implementing relationships are applicable. Five approaches to database design that will be discussed later in this chapter are:

- Database Design via the E-R Model
- Database Design via the Extended Relational Model
- Database Design via the UML Model
- Database Design via the Entity/Object Specification Grid
- Database Design via Normalization Theory

5.2 The E-R Model Revisited

Recall that in chapter 3, the similarity and differences between an entity and a relation were noted. If we assume the similarity, then the E-R model can be construed as merely a specific interpretation of the relational model.

In order to unify the informal E-R model with the formal relational model, Codd introduced a number of conventions specific to the E-R model. These are summarized in Figure 5-1 and illustrated in Figure 5-2. Note that the model displayed in the figure represents only a section of the (partial) database model introduced in chapter 3 (review Figure 3-4b). Figure 5-2a employs the Chen notation, while Figure 5-2b employs the Crow's Foot notation. In both cases, the attributes of entities have been omitted, except for primary key attributes (as mentioned in chapter 3 and emphasized later in this chapter, there are more creative ways to represent attributes). Figure 5-3 illustrates the representation of super-type and subtype relationships.



- A *weak entity* is one that cannot exist by itself. For instance, if employees have dependents then **Dependents** is a weak entity and **Employee** is a *regular entity*. On the E-R diagram, weak entities are represented by double lines.
- Relationships may be represented by either Chen's notation, or the Crow's Foot notation.
- If Chen's notation is used to represent relationships, then the following apply:
  - ♦ The double diamond indicates a relationship between weak and regular entity.
  - ♦ The name of the relationship is written inside the diamond.
  - ♦ A double relationship line represents *total participation;* a single relationship line represents *partial participation.* For instance, if all dependents must have a reference employee, participation is total on the part of dependents; however, all employees need not have dependents, so that participation is partial on the part of employees.
- *Entity Types* are used to distinguish entities. An entity can be a *subtype* of another entity that is a *super-type.* For instance, in an organization, the entity Programmer would be a subtype of the entity Employee, the super-type. All properties of a super-type apply to its subtype; the converse does not hold. Figure 5.3 illustrates how subtypes and super-types are represented.
- Implementation of relationships can be realized as discussed in chapter 3. Furthermore, the fundamental integrity rules (section 4.1) must also be upheld.

Figure 5-1. *E-R Model Conventions*

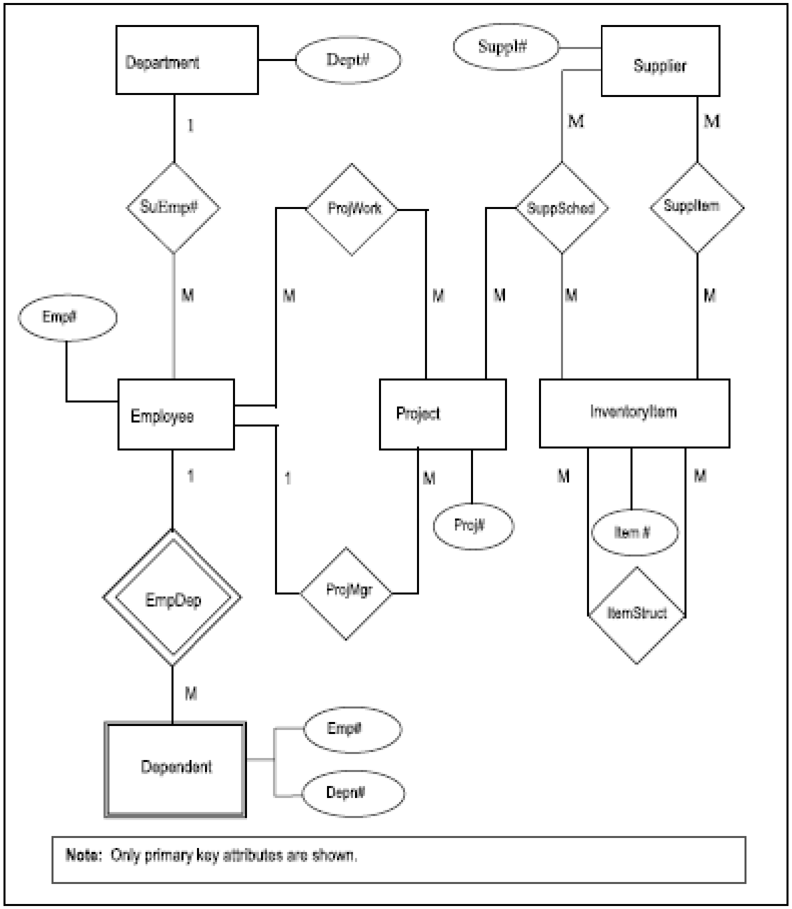
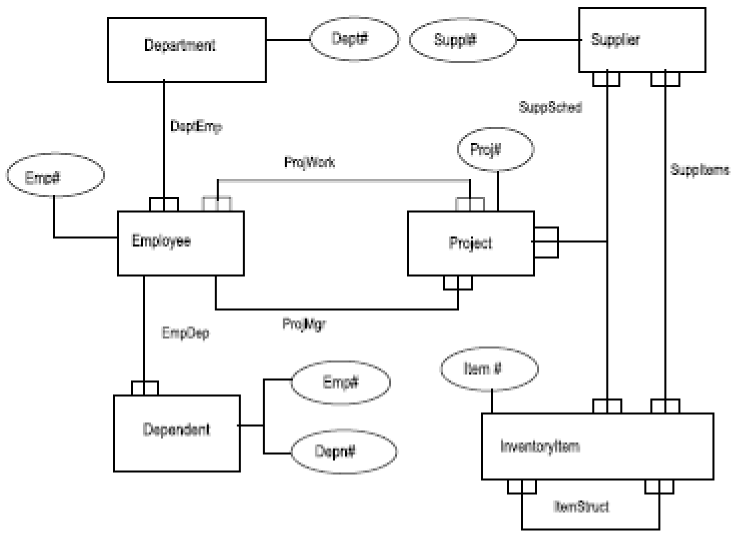


Figure 5-2a. Partial E-R Diagram for Manufacturing Firm (Chen's Notation)



Note: Only attributes that constitute the primary keys are shown.

Relationship	**Classification**
DeptEmp	1:M; total participation on the **Employee** side.
EmpDep	1:M; total participation on **Dependent** side
ProjWork	M:M
ProjMgr	1:M
ItemStruct	M:M
SuppItems	M:M
SuppSched	MM; can be replaced as shown in chapter 4

Figure 5-2b. *Alternate Partial E-R Diagram for Manufacturing Firm (Crow's Feet Notation)*

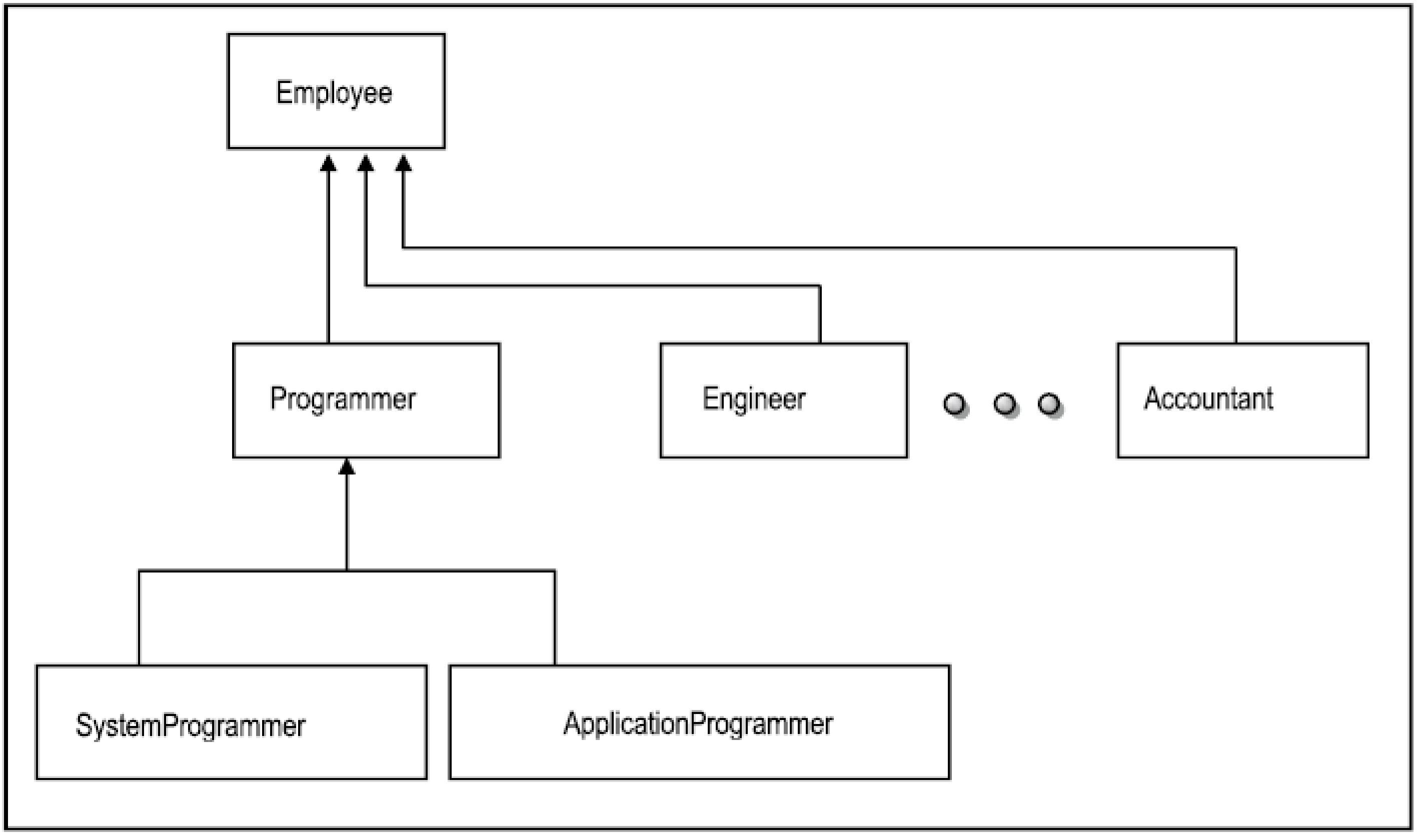


Figure 5-3. *Example of Type Hierarchy*

5.3 Database Design via the E-R Model

Database design with the E-R model simply involves following the rules established in chapter 3 on implementing relationships (section 3.5.4). These rules tell you exactly how to treat the various kinds of relationships; take some time to review them. We may therefore construct a procedure for database design via the E-R model as follows:

1. Identify all entities and their related attributes.
2. Classify the entities (weak versus strong).
3. Identify all relationships among the entities.
4. Classify the relationships (mandatory versus optional); decide on which optional relationships will be retained and which ones will be eliminated.
5. Construct an ERD or the equivalent (review chapter 3).
6. Refine the model.
7. Using the guidelines for implementing relationships (section 3.5.4), construct a final set of relations, clearly indicating for each relation, its attributes, candidate key(s), and primary key. The RAL and RL of chapter 3 (section 3.6) may be employed.
8. By consistently following this procedure, you will obtain a set of relations that will be normalized to at least the Boyce-Codd normal form (BCNF). You can then apply your normalization theory until you achieve the desired level of normalization.

Note: The illustrations given in chapter 3 (figures 3.4b, 3.13 and 3.14) are applicable here.

Figure 5-4. *Database Design Procedure Using the E-R Model*

5.4 The Extended Relational Model

Even with the conventions above, the E-R model was found lacking in its treatment of certain scenarios. Recognizing this, Codd and Date introduced an alternate *extended relational model* (which for convenience will be abbreviated as the XR model), the essence of which is described here (for more details, see [Date, 1990] and [Date, 2004]).

The XR model makes no distinction between entities and relations; an entity is a special kind of relation. Structural and integrity aspects are more extensive and precisely defined than those of the E-R model. The XR model introduces its own special operators apart from those of the basic relational model. Additionally, entities (and relationships) are represented as a set of E-relations and P-relations.

The model includes a formal catalog structure by which relationships can be made known to the system, thus facilitating the enforcement of integrity constraints implied by such relationships.

As you will see later in the course, it turns out that the XR model forms the basis of how the system catalog is handled in most contemporary DBMS suites. We shall therefore spend a few moments to look at the main features of the model.

5.4.1 Entity Classifications

Under the XR model, the following entity classifications hold:

- Kernel entities
- Characteristic entities
- Designative entities
- Associative entities
- Sub-type/super-type entities

Kernel Entities: *Kernel entities* are those entities that have independent existence. They are what the database is really about. For example, in an inventory system, kernel entities might be **Purchase Order**, **Receipts of Goods** (Invoice), **Inventory Item**, **Department** and **Issuance of Goods** (to various departments). Referring to the example used in the previous section, kernel entities would be **Suppliers**, **InventoryItems**, **Projects**, **Employees**, and **Departments**.

Characteristic Entities: *Characteristic entities* describe other entities. For instance (referring to Figure 5-2b), **Dependent** is a characteristic of **Employee**. Characteristics are existence-dependent on the entity they describe.

Designative Entities: An entity, regardless of its classification, can have a property (attribute) whosefunction is to *designate* (reference) some other entity, thus implementing a 1:M relationship. For instance (referring to Figure 5-2b), **Employee** is *designative* of **Department**, and **Project** is designative of **Employee** (due to relationship **ProjMgr**). Put another way, a designation is the implementation of an M:1 relationship. The designating entity is the entity on the "many-side" of a 1:M relationship. **Note** that a characteristic entity is necessarily designative since it designates the entity on which it is existence-dependent. Note however, that a designative entity is not necessarily characteristic. Entities **Project**, **Employee**, and **Dependent** amplify these points (see Figure 5-2).

Associative Entities: *Associative entities* represent M:M relationships among two or more entities. For example, from Figure 5-2b, the relationships **ProjWork**, **SuppSched**, **SuppItems**, and **ItemStruct** would be implemented as associative entities. In a college database with kernel entities **Program** and **Course** (among others of course), and a M:M relationship between them, the associative entity representing the relationship could be **ProgramStructure**, which would include foreign keys referencing **Program** and **Course** respectively. Note that the associative entity is the intersecting relation in the implementation of a M:M relationship (review chapter 3, section 3.5.4).

Subtype/Super-type Entities: If we have two entities E1 and E2, such that a record of E1 is always a record of E2, and a record of E2 is sometimes a record of E1, then E1 is said to be a subtype of E2. The existence of a subtype implies the existence of a super-type: To say that E1 is a subtype of E2, is equivalent to saying that E2 is the super-type of E1. For illustration, review Figure 5-3.

5.4.2 Surrogates

Recall that the concept of a surrogate was first introduced in chapter 3 (section 3.5.4). In understanding the X-R model, the role of surrogates is very important; we therefore revisit the concept here. Surrogates are system controlled primary keys, defined to avoid identification of records (tuples) by user-controlled primary keys. They are also often used to replace composite primary keys. Two consequences of surrogates arise (both of which can be relaxed with a slight deviation from the XR model which does not enforce surrogates, *E-relations* and *P-relations* as mandatory):

- Primary and foreign keys can be made to always be non-composite.
- Foreign keys always reference corresponding E-relations (more on this shortly).

Surrogates provide two significant benefits:

- In some traditional DBMS suites, composite primary keys are not allowed; surrogates are therefore imperative.
- Even if allowed by the DBMS, composite primary keys are sometimes cumbersome; surrogates are useful replacements in these circumstances.

To demonstrate the usefulness of surrogates in simplifying database model and ultimate design (with respect to avoiding cumbersome composite primary keys), let us suppose that we want to track purchase orders and their related invoices. The related entities that we would need to track are **Supplier**, **InventoryItem**, **PurchaseOrder** and **PurchaseInvoice**. These are not all included in Figure 5-2; however they are represented in figure 3-4b of chapter 3 (please review). By following through on the E-R model, or by applying normalization principles of chapter 4, we may construct a tentative set of normalized relations as illustrated in Figure 5-5a. Notice how potentially cumbersome the composite keys would be, particularly on relations **PurchaseOrdDetail** and **PurchaseInvDetail**. However, by introducing surrogates as illustrated in Figure 5-5b, we minimize the need to use complex composite keys.

Supplier {Suppl#, SuppName, Address, E-mail, ContactPerson, Telephone, ...}
PK [Supplier#]

InventoryItem {Item#, ItemName, QuantityOnHand, LastPrice, AveragePrice, ...}
PK [Item#]

PurchaseOrdSummary {Order#, OrderDate, *OrderSuppl#*, OrderStatus, OrderEstimate, …}
PK [Order#, OrderDate, OrderSuppl#] /* Assuming order numbers may be repeated after a cycle of several years */

PurchaseOrdDetail {POD*Order#*, *PODOrderDate*, POD*OrderSuppl#*, *PODItem#*, OrderQuantity, …}
PK [PODOrder#, PODOrderDate, PODOrderSuppl#, PODItem#]

PurchaseInvSummary {Invoice#, *InvOrder#*, *InvOrderDate*, *InvSuppl#*, InvDate, InvAmount, InvStatus, InvDiscount, InvTax, InvAmountDue, …}
PK [Invoice#, InvoiceDate, InvOrder#, InvOrderDate, InvSuppl#]

PurchaseInvDetail {PIDInvoice#, PIDInvoiceDate, PIDInvOrder#, PIDInvOrderDate, PIDInvSuppl#, PIDItem#, PIDItemQuantity, PIDItemUnitPrice}
PK [PIDInvoice#, PIDInvoiceDate, PIDInvOrder#, PIDInvOrderDate, PIDInvSuppl#, PIDItem#]

Note: Foreign keys are *italicized*.

Figure 5-5a. *Model to Track Purchase Orders and Invoices (without Surrogates)*

Supplier {Suppl#, Supp-Name, Address, E-mail, Contact-Person, Telephone, etc.)
PK [Supplier#]

InventoryItem {Item#, Item-Name, Quantity-On-Hand, Last-Price, Average-Price, etc.}
PK [Item#]

PurchaseOrdSummary {OrderRef, Order#, OrderDate, OrderSuppl#, OrderStatus, OrderEstimate, …}
PK [OrderRef] /* OrderRef is a surrogate. Alternately, we may define Order# to be non-repeatable */

PurchaseOrdDetail {*PODOrderRef, PODItem#*,, OrderQuantity}
PK [PODOrderRef, PODItem#] / * or introduce a surrogate, **PODCode,** and make it the PK */

PurchaseInvSummary {PurchaseRef, Invoice#, InvOrderRef, InvDate, InvAmount, InvStatus, InvDiscount, InvTax, InvAmountDue, …}
PK [PurchaseRef] /* PurchaseRef is a surrogate */

PurchaseInvDetail {PIDPurchaseRef, PIDItem#, PIDItemQuantity, PIDItemPrice}
PK [PIDPurchaseRef, PIDItem#] /* or introduce a surrogate, **PIDCode**, and make it the PK */

Note: Foreign keys are *italicized*.

Figure 5-5b. *Alternate Model to Track Purchase Orders and Invoices (with Surrogates)*

5.4.3 E-Relations and P-Relations

The original XR model specification prescribes the use of *E-relations* and *P-relations*. The database would contain one *E-relation* for each entity type — a unary relation that lists surrogates for all tuples of that entity (type). To illustrate, let us revisit the manufacturing firm's partial database (Figure 5-2) of previous discussions: Suppose for a moment that the **Supplier** relation contains two tuples, the **InventoryItem** relation contains three tuples, and the **Department** relation contains three tuples. A possible internal representation of the E-relations for this scenario is illustrated in Figure 5-6a. An "E" is inserted in front of the original relation name (for example **E-Supplier**) to denote the fact that this is an E-relation being represented. The percent sign (%) next to the attributes (for example **Suppl%**) are used to denote the fact that these attributes are really surrogates.

In addition to the E-relations concerned with tuples, a special binary E-relation would be required to link so-called E-relations to the original relations. We will call this the *host E-relation*. It is illustrated in Figure 5-6b. This would allow users to relate to the database using relation names that they are familiar with; translation would be transparent to them.

Properties (attributes) for a given entity type are represented by a set of *P-relations*. The P-relation stores all property characteristics and values of all tuples listed in the corresponding E-relation. Properties can be grouped together in a single n-ary relation, or each property can be represented by P-relation, or there can be a convenient number of P-relations used; the choice depends on the designer. In the interest of simplicity, let us assume the third approach; let us assume further, that there is a P-relation for each E-relation. A convenient possible representation for the E-relations of Figure 5-6a is illustrated in Figure 5-6c. A "P" is inserted in front of the original relation name (for example **P-Supplier**) to denote the fact that this is a P-relation being represented. Notice also that each P-relation contains a foreign key that ensures that each tuple is referenced back to its correspondent in the associated E-relation.

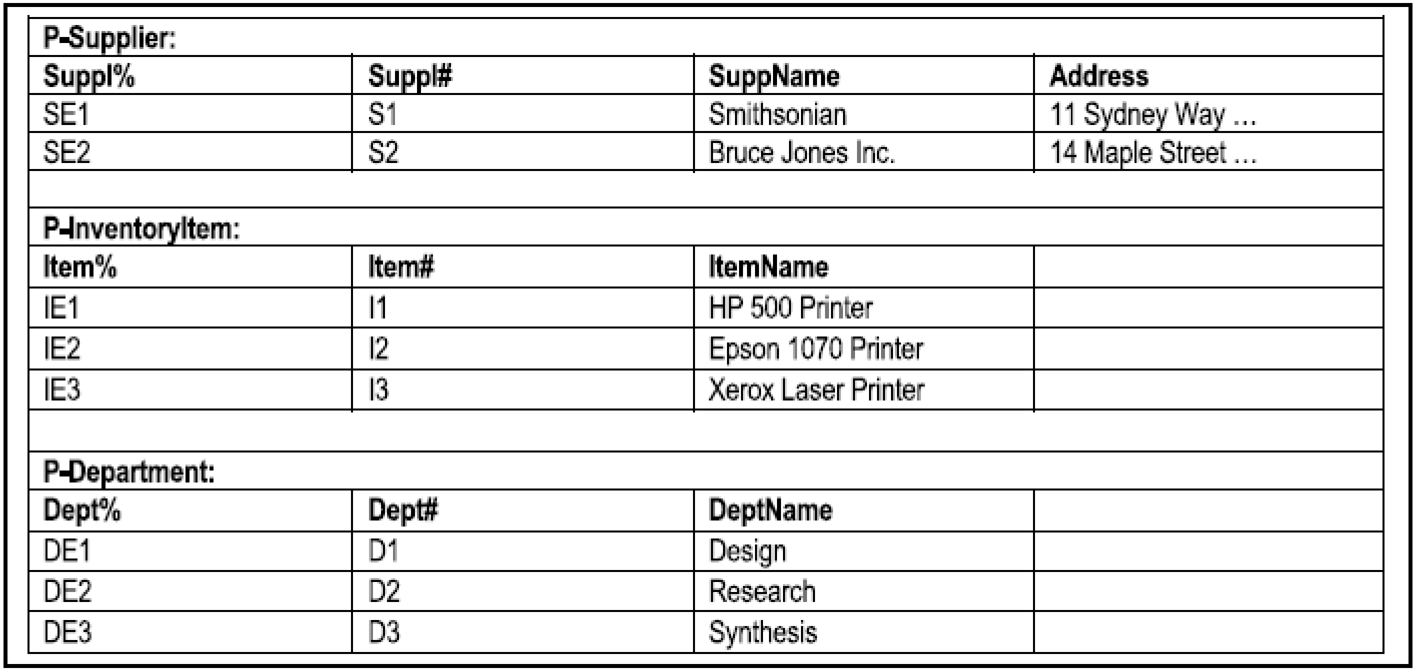
Carrying on with the assumption that there is a P-relation for each E-relation: In addition to the basic P-relations, a special P-relation would be required to store the characteristics of each property (to be) defined in the database. Let us call this the *host P-relation*. It is represented in Figure 5-6d. By including this relation, we allow users the flexibility of adding new properties to a relation, modifying existing properties in a relation, or deleting pre-existent properties from a relation. These changes are referred to as *structural changes* to a relation; they will be amplified later in the course (chapter 11).

E-Supplier: Suppl%	E-InventoryItem: Item%	E-Department: Dept%
SE1	IE1	DE1
SE2	IE2	DE2
	IE3	DE3

Figure 5-6a. Illustrating E-relations

E-Host:	
Relation	**E-relation**
Department	E-Department
InventoryItem	E-InventoryItem
Supplier	E-Supplier

Figure 5-6b. Illustrating the Host E-relation



P-Supplier:

Suppl%	**Suppl#**	**SuppName**	**Address**
SE1	S1	Smithsonian	11 Sydney Way …
SE2	S2	Bruce Jones Inc.	14 Maple Street …

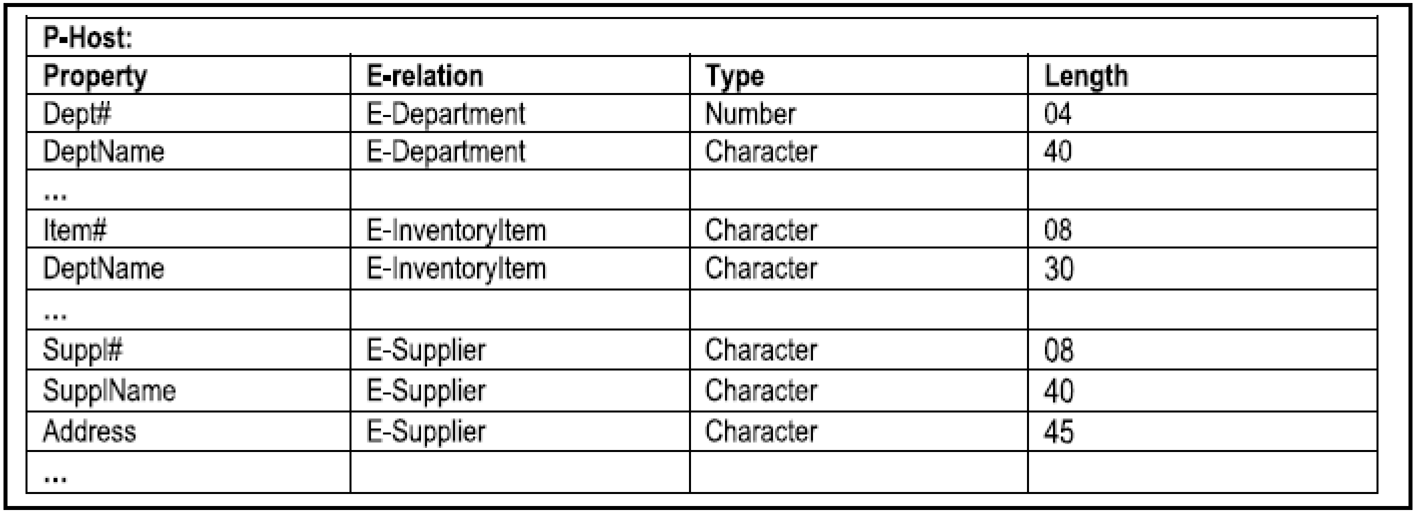
P-InventoryItem:

Item%	**Item#**	**ItemName**	
IE1	I1	HP 500 Printer	
IE2	I2	Epson 1070 Printer	
IE3	I3	Xerox Laser Printer	

P-Department:

Dept%	**Dept#**	**DeptName**	
DE1	D1	Design	
DE2	D2	Research	
DE3	D3	Synthesis	

Figure 5-6c. *Illustrating The P-relations*



P-Host:

Property	E-relation	Type	Length
Dept#	E-Department	Number	04
DeptName	E-Department	Character	40
...			
Item#	E-InventoryItem	Character	08
DeptName	E-InventoryItem	Character	30
...			
Suppl#	E-Supplier	Character	08
SupplName	E-Supplier	Character	40
Address	E-Supplier	Character	45
...			

Figure 5-6d. *Illustrating the Host P-relation*

As you will later see (in chapter 14), it is application of this methodology that assists in the implementation of sophisticated system catalogs that characterize contemporary DBMS suites. However, with this knowledge, you can actually model and design databases to mirror E-relations and P-relations as described. One obvious advantage is that if you used one P-relation instead of one for each E-relation, then in accessing the database for actual data, you would be accessing fewer relations (in fact just one relation) than if you had used another approach (such as the E-R model). The flip side to this advantage is that this relation would be extremely large for a medium sized or large database; this could potentially offset at least some of the efficiency gained from just having to access one relation for data values.

5.4.4 Integrity Rules

With this set-up, accessing and manipulating data in the database is accomplished by the DBMS through the E and P relations. For this reason, additional integrity rules must be imposed. The complete list of integrity rules follows:

1. Entity integrity rule (review section 4.1)
2. Referential rule (review section 4.1)
3. XR Model Entity Integrity: E-relations accept insertions and deletions but no updates (surrogates don't change)
4. Property Integrity: A property cannot exist in the database unless the tuple (entity) it describes exists
5. Characteristic Integrity: A characteristic entity (tuple) cannot exist unless the entity (tuple) it describes exists
6. Association Integrity: An association entity (tuple) cannot exist unless each participating entity (tuple) also exists
7. Designation Integrity: A designative entity (tuple) cannot exist unless the entity (tuple) it designates also exists
8. Subtype Integrity: A subtype entity (tuple) cannot exist except there be a corresponding super-type entity (tuple)

The following rules apply to subtypes and super-types:

1. All characteristics of a super-type are automatically characteristics of the corresponding subtype(s), but the converse does not hold.

2. All associations in which a super-type participates are automatically associations in which the corresponding subtype(s) participate, but the converse does not hold.

3. All properties of a given super-type apply to the corresponding subtype(s), but the converse does not hold.

4. A subtype of a kernel is still a kernel; a subtype of a characteristic is still a characteristic; a subtype of an association is still an association.

5.5 Database Design via the XR Model

The following approach, developed by Date and outlined in [Date, 1990], employs the basic XR method,but relaxes the requirement that surrogates, E-relations and P-relations are mandatory. The (partial)database model represented in Figure 5-2 will be used for the purpose of illustration.

Before proceeding, we now introduce a notation that has become necessary: The notation R.A will be usedto mean attribute A in relation (or entity) R. For instance, **Department.Dept#** denotes attribute **Dept#** in the relation **Department**. The database design approach involves seven steps as summarized in Figure 5-7 and clarified in the upcoming subsections.

1. Determine kernel entities.
2. Determine characteristic entities.
3. Determine Designative entities.
4. Determine associations.
5. Determine subtypes and super-types.
6. Determine component entities.
7. Determine properties of each entity.
8. Construct a relation-attribute list (RAL) for each relation.
9. By consistently following this procedure, you will obtain a set of relations that will be normalized to at least the Boyce-Codd normal form (BCNF). You can then apply your normalization theory until you achieve the desired level of normalization.

Figure 5-7. *Database Design Procedure Using the XR Model*

5.5.1 Determining the Kernel Entities

The first step involves determining the kernel entities. As mentioned, earlier, kernels are the core relations. In the example, the kernels are **Department**, **Employee**, **Supplier**, **Project**, and **InventoryItem**. Each kernel translates to a base relation. The primary key of each could be the user controlled ones, or surrogates may be introduced.

5.5.2 Determining the Characteristic Entities

The second step involves determining and properly structuring the characteristic entities. As mentioned above, a characteristic entity is existence-dependent on the entity it describes. One characteristic exists in the example, namely **Dependent**. Characteristics also translate to base relations. The foreign key in **Dependent** would be **DepnEmp#**, which would reference **Employee.Emp#**. Notice that we did not use the attribute name **Emp#** as the foreign key in **Dependent**, but **DepnEmp#**. This decision is based on the following principle:

> It is good design practice to define each attribute so that it is unique to the database (even if the attribute is a foreign key).

It should be noted that not all DBMS products support this principle (some require that a foreign key must have the same name as the attribute it references). You should therefore check to ascertain whether the product you are using supports it (Oracle and DB2 both do). Two strong arguments can be given in defense of this principle:

- Simply, it makes good sense and leads to a cleaner, more elegant database design.
- It avoids confusion when queries involving the joining of multiple relations are constructed and executed on the database. This will become clearer in division C (particularly chapter 12) of the text.

Moving on to data integrity, we would require the following integrity constraints on the **Dependent** relation:

- Null FKs not allowed
- Deletion is cascaded from the referenced to the referencing records
- Update cascaded from the referenced to the referencing records

Two alternatives exist for choice of primary key:

a. The foreign key combined with the attribute that distinguishes different characteristics within the target entity (e.g. [**Emp#**, **DepnName**]);

b. Introduce a surrogate (e.g. **DepnRef**). The surrogate could be defined in such as way as to allow you to key solely on it; or it could be defined to allow you to key on the foreign key, combined with it (e.g. [**Emp#**, **DepnRef**]).

5.5.3 Determining the Designative Entities

This third step involves identifying and properly structuring the designative entities. As mentioned earlier, a designation is a 1:M or 1:1 relationship between two entities. In the example, designations are **ProjMgr**, **DeptEmp**, and **EmpDep**. However, **EmpDep** is a characteristic (that has already been identified above).

From the theory established in chapter 3 (section 3.5.4), a designation is implemented by the introduction of a foreign key in the relation for the designating entity. Following this principle, we would introduce foreign key, **EmpDept#** in relation **Employee** (where **EmpDept#** references **Department.Dept#**), and foreign key, **ProjManagerEmp#** in relation **Project** (where **ProjManagerEmp#** references **Employee.Emp#**).

Integrity constraints for designations would typically be:

- Null FKs allowed in the designating entity if the participation is partial
- Null FKs not allowed in the designating entity if the participation is full
- Deletion of referenced records restricted
- Update of referenced records restricted (although for some practical purposes, update could be cascaded)

Typically, the foreign key in a designative entity is a non-key attribute. Consequently, there are normally no keying issues. However, there could be exceptions to this observation (for instance in the case where an intersecting relation is introduced to implement a M:M relationship).

5.5.4 Determining the Associations

Step 4 involves identification and implementation of all associations. As mentioned earlier, associations are the implementation of M:M relationships. They translate to base relations. In the example, associations are **ProjWork**, **SuppItems**, **ItemStruct**, and **SuppSched**. Again relying on the theory established in chapter 3 (section 3.5.4) on the implementation of M:M relationships, we would introduce four base relations for the four associations — **ProjWork**, **SuppItems**, **ItemStruct**, and **SuppSched**. However, as established in chapter 4 (section 4.10), **SuppSched** should be replaced with three relations, namely **SuppItems**{Suppl#, Item#}, **ItemProj**{Item#, Proj#}, and **ProjSupp**{Proj#, Suppl#}. Additionally, and in keeping with the principle of having unique attribute names for the entire database, we will change the foreign key attribute names to names that are unique but easily traceable to the attributes they reference.

Integrity constraints for associations would typically be:

- Null FKs not allowed
- Deletion of referenced records restricted
- Update of referenced records restricted

Two alternatives exist for choice of primary key:

a. Key on the aggregation of the foreign keys.

b. Introduce a surrogate and key on it.

5.5.5 Determining Entity Subtypes and Super-types

The fifth step relates to identifying and properly implementing subtype-super-type relationships among the entities. Care should be taken here, in not introducing subtype-super-type relationships where traditional relationships would suffice (review section 3.5). Each entity type translates to a base relation. Each base relation will contain attributes corresponding to properties of the entities that apply within the type hierarchy. Again being guided by principles established in chapter 3 (section 3.5.4), each subtype will share the primary key of its super-type. Further, the primary key of a subtype is also the foreign key of the said subtype. The illustrations provided in chapter 3 (Figures 3-11 and 3.12) are also applicable here.

No subtype-super-type relationship appears in the model of Figure 5-2. However, in Figure 5-3, there are a few: **Programmer**, **Engineer** and **Accountant** are subtypes of **Employee**; **SystemProgrammer** and **ApplicationProgrammer** are subtypes of **Programmer**. Note also that in a subtype, except for the primary key (which is also a foreign key), no additional attributes of the super-type need to be repeated, since they are inherited. However, additional attributes may be specified (in the subtype). For example (still referring to Figure 5-3), the **Programmer** entity may contain the attribute **ProgLanguage** to store the programmer's main programming language; this would not apply to **Employee**.

For subtypes, integrity constraints on foreign (primary) keys may be:

- Nulls not allowed
- Deletion of referenced records restricted (in super-type)
- Deletion of referencing records allowed (in subtype but not in super-type)
- Update cascaded from the super-type

5.5.6 Determining Component Entities

Component entities were not discussed in Date's original work on the database design via the XR model. However, in the interest of comprehensive coverage, this sub-section has been added. As mentioned in the previous sub-section, care should be taken in not introducing component relationships where traditional relationships would suffice (review section 3.5). Each entity type translates to a base relation. Each base relation will contain attributes corresponding to properties of the entities that apply within the type hierarchy. Again being guided by principles established in chapter 3 (section 3.5.4), each component will include a foreign key, which is the primary key in the summary relation.

Further, this foreign key will form part of the primary key (or a candidate key) in the component relation. For examples, please refer to figures 3-11 and 3.12 of chapter 3.

For components, integrity constraints on foreign (primary) keys may be:

- Null FKs not allowed
- Deletion of referenced records restricted (in summary)
- Deletion of referencing records allowed (in component but not in summary)
- Update cascaded from the summary

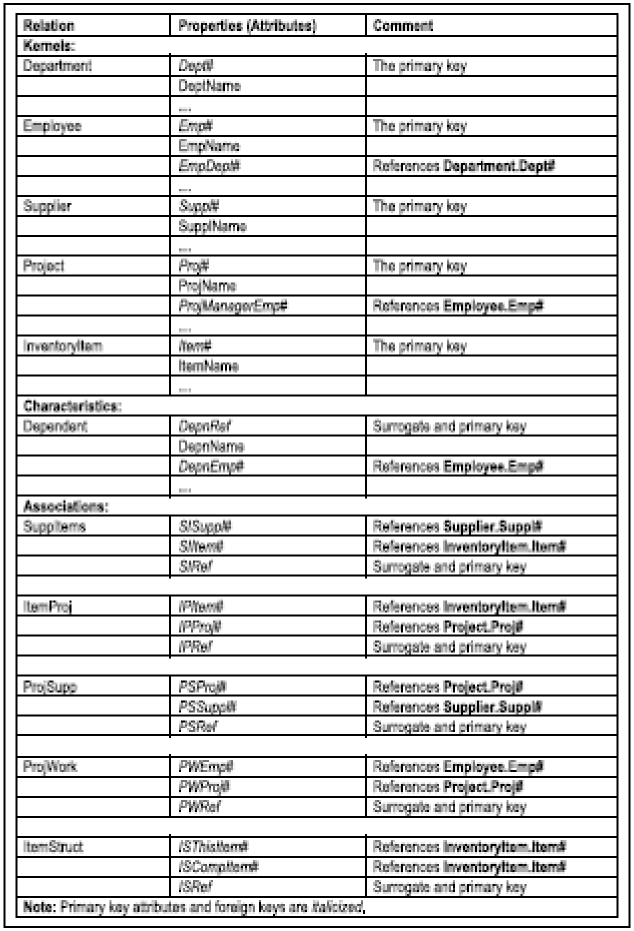
5.5.7 Determining the Properties

The final step in this (modified) XR approach is to carefully determine the properties in each relation (entity). This is actually easy, but in order to avoid mistakes, you must be diligent:

- Except for associations, in your initial system investigation (which would be part of the required software engineering or systems analysis), you would have identified the basic properties for each identified entity. That is your starting point.
- Next, go through steps 1-3, 5, and 6 above, and observe the guidelines for dealing with characteristics, designations, subtypes and components. These steps tell you when and where to introduce foreign keys.
- Next, observe step 4 above for treating associations.

By following this procedure, you will be able to confidently determine the properties for each relation in the database; in fact, you will end up with a list that is identical or very similar to the one provided in Figure 5-5 above. This finalized list is illustrated in Figure 5-8. As you examine the figure, please note the following:

1. All primary key and foreign key attributes are italicized.
2. The principle of having each attribute with a unique attribute name (including foreign keys) has been followed.
3. The database specification is presented via a Relation-Attributes List (RAL) — a technique introduced in chapter 3 (section 3.6).



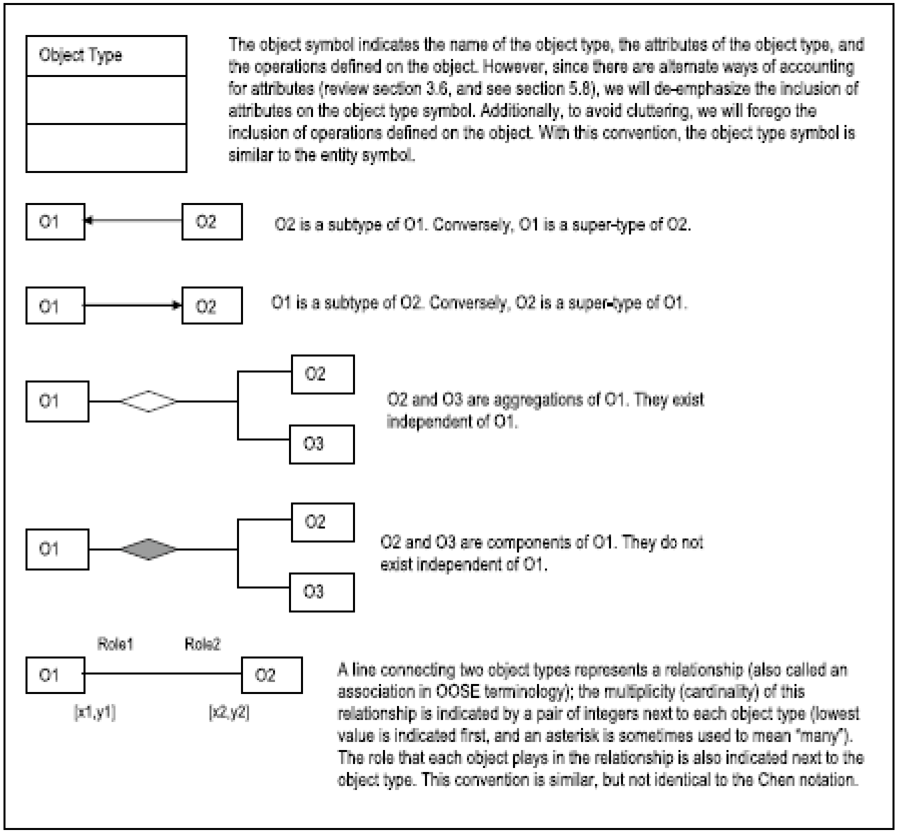
Relation	Properties (Attributes)	Comment
Kernels:		
Department	*Dept#*	The primary key
	DeptName	
	....	
Employee	*Emp#*	The primary key
	EmpName	
	EmpDept#	References **Department.Dept#**
	....	
Supplier	*Supp#*	The primary key
	SupplName	
	....	
Project	*Proj#*	The primary key
	ProjName	
	ProjManagerEmp#	References **Employee.Emp#**
	....	
InventoryItem	*Item#*	The primary key
	ItemName	
	....	
Characteristics:		
Dependent	*DepnRef*	Surrogate and primary key
	DepnName	
	DepnEmp#	References **Employee.Emp#**
	....	
Associations:		
SuppItems	*SISupp#*	References **Supplier.Supp#**
	SIItem#	References **InventoryItem.Item#**
	SIRef	Surrogate and primary key
ItemProj	*IPItem#*	References **InventoryItem.Item#**
	IPProj#	References **Project.Proj#**
	IPRef	Surrogate and primary key
ProjSupp	*PSProj#*	References **Project.Proj#**
	PSSupp#	References **Supplier.Supp#**
	PSRef	Surrogate and primary key
ProjWork	*PWEmp#*	References **Employee.Emp#**
	PWProj#	References **Project.Proj#**
	PWRef	Surrogate and primary key
ItemStruct	*ISThisItem#*	References **InventoryItem.Item#**
	ISCompItem#	References **InventoryItem.Item#**
	ISRef	Surrogate and primary key
Note: Primary key attributes and foreign keys are *italicized*.		

Figure 5-8. *Partial Database Specification for Section of Manufacturing Firm's Database*

With this additional information, you can now revisit the database specification of chapter 3 (Figures 3-4b, 3-13 and 3-14) and revise it accordingly (left as an exercise for you). In so doing, please observe that while the ERD of Figure 5-2 is similar to that of Figure 3-4b, they are not identical; they highlight different aspects of a manufacturing environment, with various areas of overlap. By examining both figures, you should come away with a better sense of what a database model and specification for such an environment would likely entail. The key is to apply sound information gathering techniques (as learned in your software engineering course and summarized in appendix 3), coupled with your database knowledge.

5.6 The UML Model

An alternate methodology for database modeling is the *Unified Modeling Language* (UML) notation. UML was developed by three contemporary software engineering paragons — Grady Booch, James Rumbaugh and Ivar Jacobson. These three professionals founded Rational Software during the 1990s, and among several other outstanding achievements, developed UML for the expressed purpose of being a universal modeling language. Although UML was developed, primarily for *object-oriented software engineering* (OOSE), it is quite suitable for database modeling. Figure 5-9 provides a description of the main symbols used in UML. Note that with UML comes a slight change in the database jargon ("entity" is replaced with "object type"), consistent with the fact that UML is primarily for OOSE.



The object symbol indicates the name of the object type, the attributes of the object type, and the operations defined on the object. However, since there are alternate ways of accounting for attributes (review section 3.6, and see section 5.8), we will de-emphasize the inclusion of attributes on the object type symbol. Additionally, to avoid cluttering, we will forego the inclusion of operations defined on the object. With this convention, the object type symbol is similar to the entity symbol.

O2 is a subtype of O1. Conversely, O1 is a super-type of O2.

O1 is a subtype of O2. Conversely, O2 is a super-type of O1.

O2 and O3 are aggregations of O1. They exist independent of O1.

O2 and O3 are components of O1. They do not exist independent of O1.

A line connecting two object types represents a relationship (also called an association in OOSE terminology); the multiplicity (cardinality) of this relationship is indicated by a pair of integers next to each object type (lowest value is indicated first, and an asterisk is sometimes used to mean "many"). The role that each object plays in the relationship is also indicated next to the object type. This convention is similar, but not identical to the Chen notation.

Figure 5-9. Symbols Used in UML Notation for Database Modeling

Note that the UML notation makes a distinction between a component relationship and an aggregation relationship. In the case of the component relationship, the constituent object types are existence dependent on a main object type. In the case of an aggregation relationship, the constituent object types are existence independent of the aggregation object type. Figure 5-10 illustrates the UML diagram for the (partial) college database model that was introduced in chapter 3 (Figure 3-11). Notice that except for the **StudentEmployee** object type, which has been omitted, the information represented is essentially similar to what was represented in Figure 3-11. The only difference here is that the appropriate UML symbols have been used.

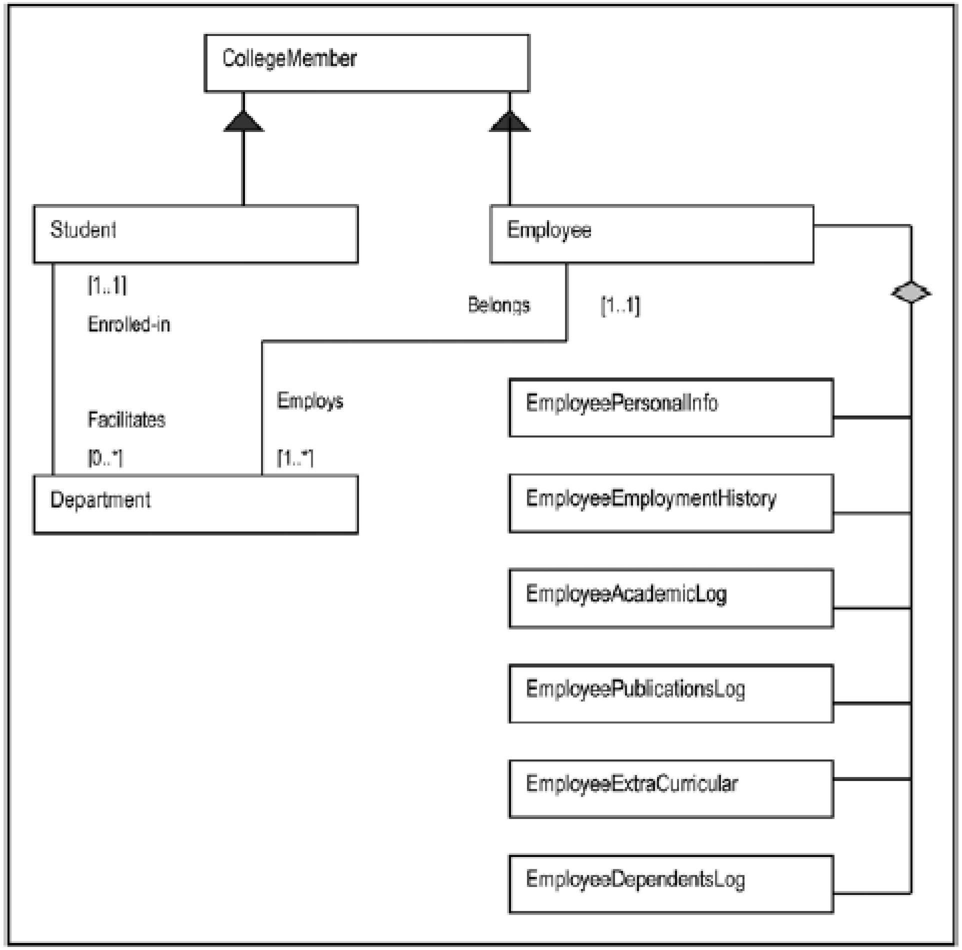
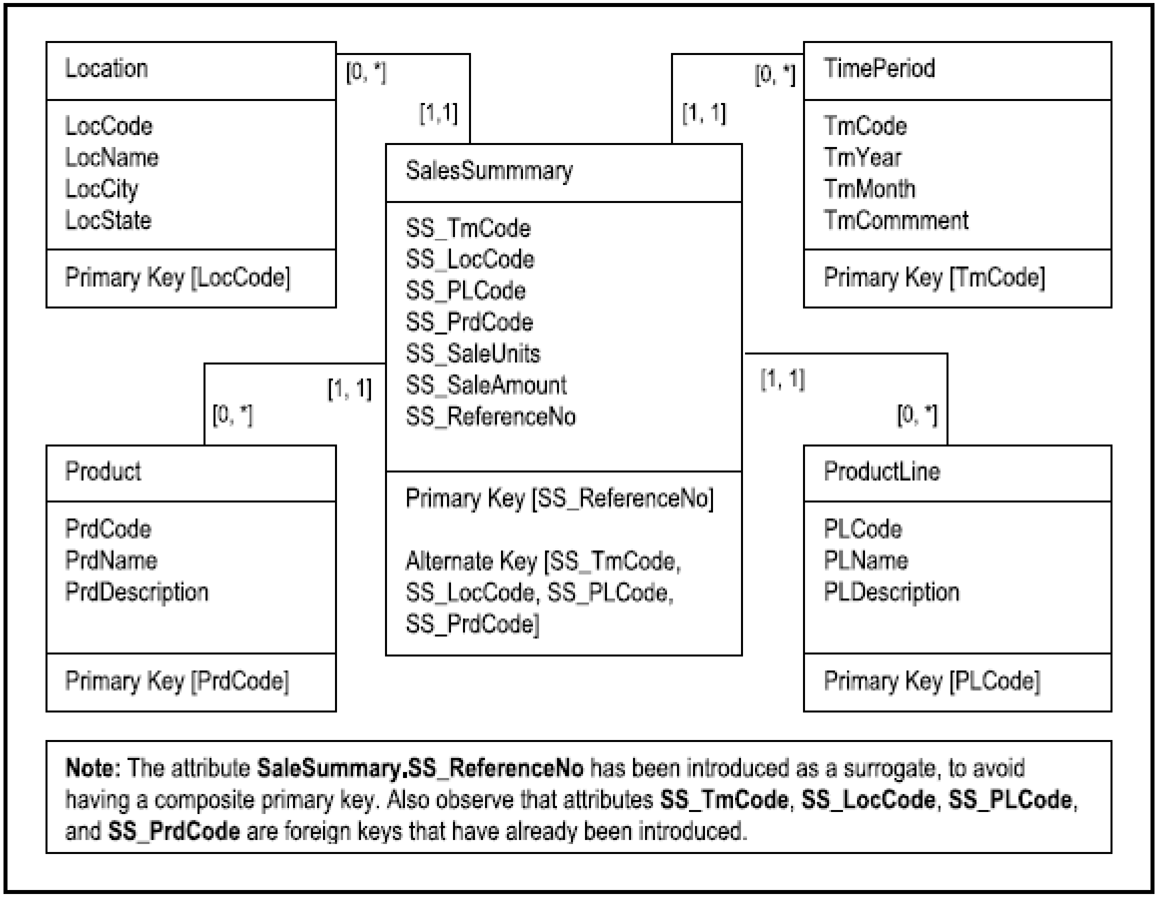


***Figure 5-10.** UML Diagram for a Partial College Database Model*

Let us examine another example: Suppose that you were hired at a large marketing company that needs to keep track of its sales of various products and product lines over time. Suppose further that the company operates out of various offices strategically located across the country and/or around the world. How would you construct a database conceptual schema that would allow the company to effectively track its sales? One way to solve this problem is to employ what is called a *star schema* — a central relation (or object type) is connected to two or more relations (or object types) by forming a M:1 relationship with each. Figure 5-11 illustrates such a schema for the marketing company. The central relation (often referred to as the *fact table*) is **SalesSummary**. The surrounding relations (often referred to as *dimensional tables*) are **Location**, **TimePeriod**, **ProductLine**, and **Product**. Each forms a 1:M relationship with **SaleSummary,** the central relation. Notice that consistent with the theory, **SalesSummary** has a foreign key that references each of the referenced relations (object types). Finally, observe also that in this illustration, the attributes for each relation (object type) have been included in the diagram.

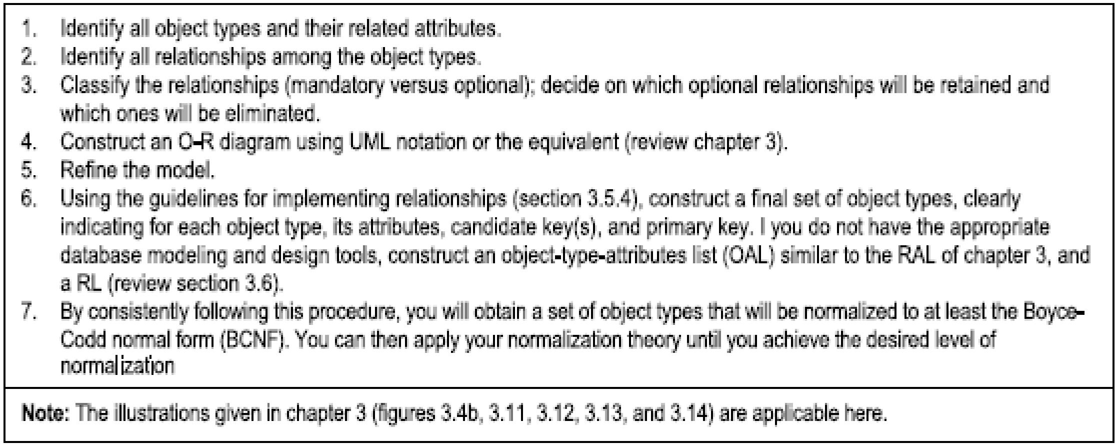


Note: The attribute **SaleSummary.SS_ReferenceNo** has been introduced as a surrogate, to avoid having a composite primary key. Also observe that attributes **SS_TmCode**, **SS_LocCode**, **SS_PLCode**, and **SS_PrdCode** are foreign keys that have already been introduced.

Figure 5-11. *UML Diagram for Tracking Sales Summary for a Large Marketing Company*

5.7 Database Design via the UML Model

Database design with the UML model is somewhat similar to database design with the E-R model. The points of divergence relate to the differences in notation between the two approaches as well as the semantic jargon used. The rules that prescribe bow to treat various types of relationships (section 3.5.4) are still applicable. However, in order to be consistent with object-oriented (OO) terminology, you would replace the term relation (or entity) with the term object type. With this in mind, we may construct a procedure for database design via the UML model as follows:



1. Identify all object types and their related attributes.
2. Identify all relationships among the object types.
3. Classify the relationships (mandatory versus optional); decide on which optional relationships will be retained and which ones will be eliminated.
4. Construct an O-R diagram using UML notation or the equivalent (review chapter 3).
5. Refine the model.
6. Using the guidelines for implementing relationships (section 3.5.4), construct a final set of object types, clearly indicating for each object type, its attributes, candidate key(s), and primary key. I you do not have the appropriate database modeling and design tools, construct an object-type-attributes list (OAL) similar to the RAL of chapter 3, and a RL (review section 3.6).
7. By consistently following this procedure, you will obtain a set of object types that will be normalized to at least the Boyce-Codd normal form (BCNF). You can then apply your normalization theory until you achieve the desired level of normalization

Note: The illustrations given in chapter 3 (figures 3.4b, 3.11, 3.12, 3.13, and 3.14) are applicable here.

Figure 5-12. Database Design Procedure Using the UML Model

Note: Due to the inherent behavior of typical OO software products, introducing primary keys and foreign keys into object types (implemented as classes) as we have prescribed, may be unnecessary. In a purely OO environment, these links are automatically introduced by the OO software and implemented as pointers — a feature called encapsulation; however, they are internal and cannot be tracked by the user. For this reason, many OO pundits argue that normalization and data independence run counter to inheritance and encapsulation. The debate as to when to use an OO database versus a relational database and vice versa, is likely to be ongoing for some time into the foreseeable future. It will be revisited in chapter 23.

5.8 Innovation: The Object/Entity Specification Grid

This section introduces an innovative approach to database design specification that has been successfully used by the author on a number of major projects. The approach may be construed as an extension of the UML model, but is applicable to any database model.

As mentioned in chapter 3 (section 3.6), for large complex projects (involving huge databases with tens of entities or object types), unless a CASE or RAD tool which automatically generates the ORD or ERD is readily available, manually drawing and maintaining this important aspect of the project becomes virtually futile. Even with a CASE tool, perusing several pages of O-R (E-R) diagram may not be much fun. In such cases, an *object/entity specification grid* (O/ESG) is particularly useful. In a relational database environment, the term *entity specification grid* (ESG) is recommended; in an object-oriented environment, the term *object specification grid* (OSG) is recommended.

The O/ESG presents the specification for each object type (or entity) as it will be implemented in a database consisting of normalized relations. The conventions used for the O/ESG are shown in Figure 5-13; a summary of these conventions follows:

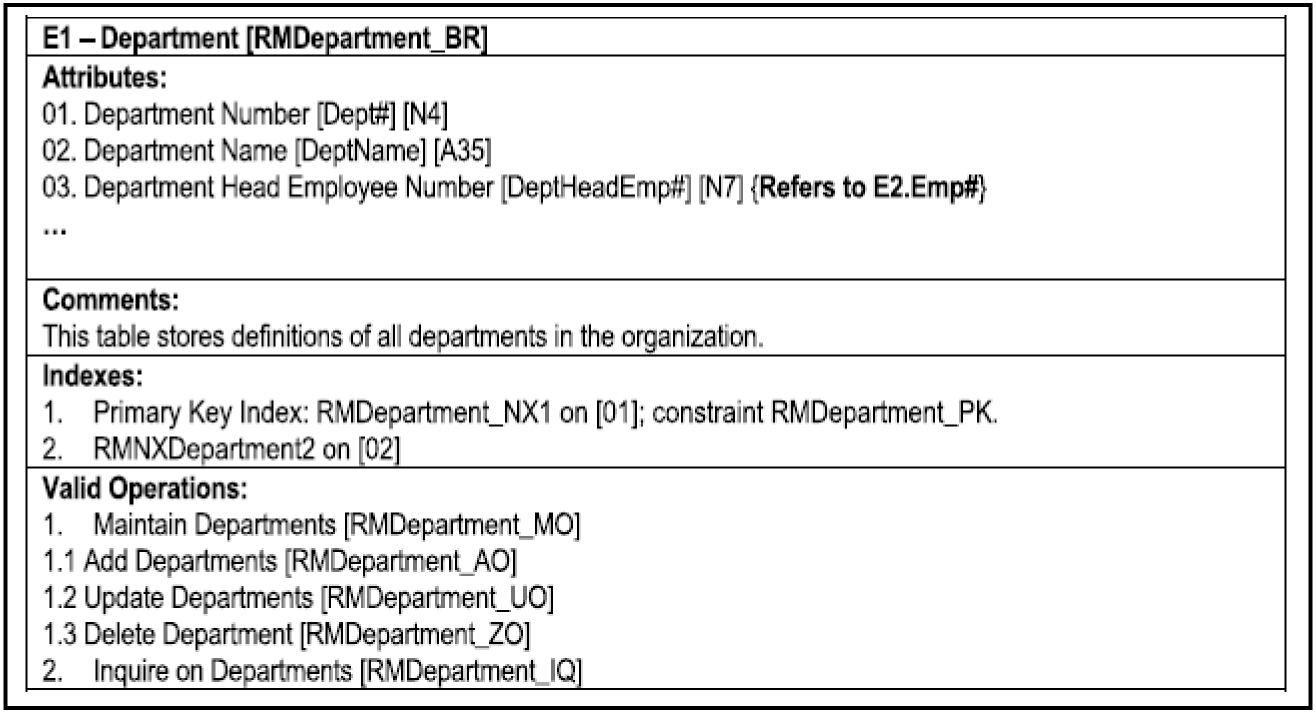
- Each object type (or entity) is identified by a reference code, a descriptive name, and an implementation name (indicated in square brackets).
- For each object type (or entity), the attributes (data elements) to be stored are identified.
- Each attribute is specified by its descriptive name, the implementation name indicated in square brackets, a physical description of the attribute (as described below), and whether the attribute is a foreign key.
- For physical description, the following letters will be used to denote the type of data that will be stored in that attribute, followed by a number which indicates the maximum length of the field: (A) alphanumeric, (N) numeric, or (M) memo. This is specified within square braces. For instance, the notation [Dept#] [N4] denotes a numeric attribute of maximum length 4 bytes. In the case of an attribute that stores a memo (M), no length is indicated because a memo field can store as much information as needed. If a real number value is being stored with a decimal value, two numbers will be used: the first number will indicate the length for the whole number part and the second number will indicate the field length of the decimal part (e.g. [N(9,2)]).
- An attribute that is a foreign key is identified by a comment specifying what object type (or entity) is being referenced. The comment appears in curly braces.
- For each object type (or entity) a comment describing the data to be stored is provided.

- An itemized specification of indexes to be defined (starting with the primary key) is provided for each object type (or entity).
- Each operation to be defined on an object type (or entity) will be given a descriptive name and an implementation name, indicated in square brackets.

Reference Number – Descriptive Name [Implementation Name]
Attributes: /* Itemized specification of all attributes of this object type (or entity) */
Comments: /* A brief description of the storage purpose of this object type (or entity) */
Indexes: /* Itemized specification of anticipated indexes, starting with the primary key */
Valid Operations: /* Itemized list of operations to be defined on this object type (or entity) */

Figure 5-13. O/ESG Conventions

Figure 5-14 provides an illustration of O/ESG for four of the object types (or entities) that would comprise the manufacturing firm's database of earlier discussion (review Figure 5-7). In actuality, there would be one for each object type (or entity) comprising the system.



E1 – Department [RMDepartment_BR]

Attributes:
01. Department Number [Dept#] [N4]
02. Department Name [DeptName] [A35]
03. Department Head Employee Number [DeptHeadEmp#] [N7] {**Refers to E2.Emp#**}
...

Comments:
This table stores definitions of all departments in the organization.

Indexes:
1. Primary Key Index: RMDepartment_NX1 on [01]; constraint RMDepartment_PK.
2. RMNXDepartment2 on [02]

Valid Operations:
1. Maintain Departments [RMDepartment_MO]
1.1 Add Departments [RMDepartment_AO]
1.2 Update Departments [RMDepartment_UO]
1.3 Delete Department [RMDepartment_ZO]
2. Inquire on Departments [RMDepartment_IQ]

***Figure 5-14.** Partial O/ESG for Manufacturing Environment*

E2 — Employee [RMEmployee_BR]
Attributes: 01. Employee Identification Number [Emp#] [N7] 02. Employee Last Name [EmpLName] [A20] 03. Employee First Name [EmpFName] [A20] 04. Employee Middle Initials [EmpMInit] [A4] 05. Employee Date of Birth [EmpDOB] [N8] 06. Employee's Department [EmpDept#] [N4] **(Refers to E1.Dept#)** 07. Employee Gender [EmpGender] [A1] 08. Employee Marital Status [EmpMStatus] [A1] 09. Employee Social Security Number [EmpSSN] [N10] 10. Employee Home Telephone Number [EmpHomeTel] [A14] 11. Employee Work Telephone Number [EmpWorkTel] [A10] …
Comments: This table stores standard information about all employees in the organization.
Indexes: 1. Primary Key Index: RMEmployee_NX1 on [01]; constraint RMEmployee_PK. 2. RMEmployee_NX2 on [02, 03, 04] 3. RMEmployee_NX3 on [09] 4. RMEmployee_NX4 on [10] or [11]
Valid Operations: 1. Manage Employees [RMEmployee_MO] 1.1 Add Employees [RMEmployee_AO] 1.2 Update Employees [RMEmployee_UO] 1.3 Delete Employees [RMEmployee_ZO] 2. Inquire on Employees [RMEmployee_IO] 3. Report on Employees [RMEmployee_RO]

E3 — Supplier [RMSupplier_BR]
Attributes: 01. Supplier Number [Suppl#] [N4] 02. Supplier Name [SupplName] [A35] 03. Supplier Contact Name [SupplContact] [A35] 04. Supplier Telephone Numbers [SupplPhone] [A30] 05. Supplier E-mail Address [SuppEmail] [A30] …
Comments: This table stores definitions of all employee classifications.
Indexes: 1. Primary Key Index: RMSupplier_NX1 on [01]; constraint RMSupplier_PK. 2. RMSupplier_NX2 on [02] 3. RMSupplier_NX3 on [04]
Valid Operations: 1. Manage Suppliers[RMSupplier_MO] 1.1 Add Suppliers [RMSupplier_AO] 1.2 Update Suppliers [RMSupplier_UO] 1.3 Delete Suppliers [RMSupplier_ZO] 2. Inquire on Suppliers [RMSupplier_IO] 3. Report on Suppliers [RMSupplier_RO]

Figure 5-14. *Partial O/ESG for Manufacturing Environment (continued)*

E4 – Project [RMProject_BR]

Attributes:
01. Project Number [Proj#] [N4]
02. Project Name [ProjName] [A15]
03. Project Summary [ProjSumm] [M]
04. Project's Manager [ProjManagerEmp#] [N7] {**References E2.Emp#**}
…

Comments:
This table stores definitions of all company projects.

Indexes:
1. Primary Key Index: RMProject_NX1 on [01]; constraint RMProject_PK.
2. RMProject_NX2 on [02]

Valid Operations:
1. Manage Projects [RMProject_MO]
1.1 Add Projects [RMProject_AO]
1.2 Update Projects [RMProject_UO]
1.3 Delete Projects [RMProject_ZO]
2. Inquire on Projects [RMProject_IO]
3. Report on Projects [RMProject_RO]

Figure 5-14. *Partial O/ESG for Manufacturing Environment (continued)*

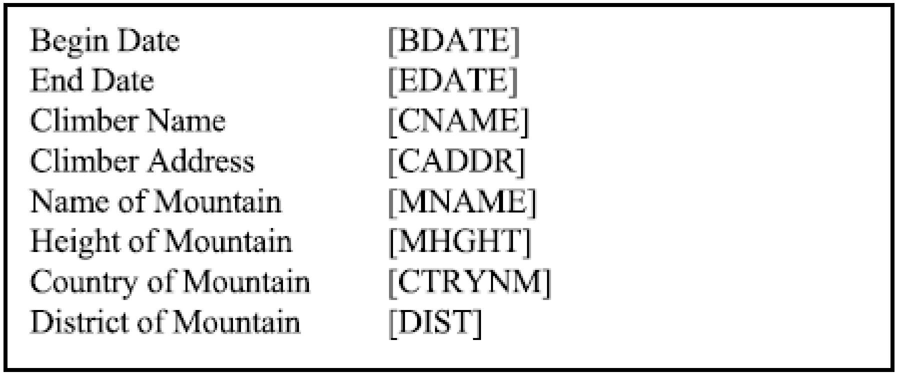
5.9 Database Design via Normalization Theory

Although this is seldom done, you can actually use the normalization theory as discussed in chapter 4 to design the basic conceptual schema (involving the structure of the base relations) of a relational database. In practice, normalization is used as a check-and-balance mechanism to acceptability of a database's conceptual schema. As such, normalization can (and should) be applied to each of the database design approaches discussed.

This section looks at two sample database design problems, and shows how the normalization theory can be used to solve them.We will advance the discussion by using two problem scenarios that will hopefully identify with.

5.9.1 Example: Mountaineering Problem

Suppose that we wish to record information about the activities of mountaineers in a relational database. Let us make the assumption that a climber can only begin one climb per day. Figure 5-15 illustrates an initial set attributes (with suggested implementation names in square brackets) for the database.



Begin Date	[BDATE]
End Date	[EDATE]
Climber Name	[CNAME]
Climber Address	[CADDR]
Name of Mountain	[MNAME]
Height of Mountain	[MHGHT]
Country of Mountain	[CTRYNM]
District of Mountain	[DIST]

Figure 5-15. *Attributes for the Mountaineering Problem*

How may we obtain an appropriate conceptual schema for the mountaineering problem? We may approach this problem in one of two ways:

- *The Pragmatic Approach*: Identify related attributes that form data entities, normalize these entities, then identify and rationalize relationships among the entities.
- *The Classical (theoretical) Approach*: Start out by creating one large 1NF relation involving all attributes, progressively decompose into relations of higher normal forms until the given requirements are met.

In the interest of illustrating application of the theory of normalization, we shall pursue the second approach. However, please bear in mind that in most real life situations, you will be advised to employ the pragmatic approach. Several of the cases in chapter 26 (for instance, assignments 1 and 2) provide an excellent opportunity for you to do this.

Step 1 — Create a large 1NF Relation

We introduce three new attributes: **Climber Identification** [CID#], **Mountain Identification** [MTN#], and **Country Code** [CTRYCD]; store all attributes as relation **M** as shown in Figure 5-16. The figure also states the observed functional dependencies.

Relation M:

Primary Key	[BDATE & CID#]
Begin Date	[BDATE]
End Date	[EDATE]
Climber Identification	[CID#]
Climber Name	[CNAME]
Climber Address	[CADDR]
Mountain Identification	[MTN#]
Name of Mountain	[MNAME]
Height of Mountain	[MHGHT]
Country Code	[CTRYCD]
Country of Mountain	[CTRYNM]
District of Mountain	[DIST]

Functional Dependencies:

FD1: [BDATE, CID#] → EDATE, CNAME, CADDR, MTN#, MNAME, MHGHT, CTRYCD, CTRYNM, DIST

FD2: CID# → CNAME, CADDR

FD3: CTRYCD → CTRYNM

FD4: MTN# → MNAME, MHGHT, DIST, CTRYCD, CTRYNM

Figure 5-16. *Revised Initial 1NF Relation for the Mountaineering Problem*

Step 2 — Obtain 2NF Relations

The second step is to obtain a set of 2NFrelations. Because of this FD2, the relation can be non-loss decomposed via Heath's theorem to obtain:

Relation **M1***:* {CID#, CNAME, CADDR} PK [CID#]

Relation **M2***:* {BDA.TE, CID#, EDATE, MTN#, MNAME, MHGHT, CTRYCD, CTRYNM, DIST} PK [CID#, BDATE]

Step 3 — Obtain 3NF Relations

Next, we seek to obtain 3NF relations. Based on FD3 and FD4, relation **M2** is not in 3NF. Again applying Heath's theorem for non-loss decomposition, we obtain the following relations:

Relation **M3**: {CTRYCD, CTRYNM} PK [CTRYCD]

Relation **M4**: {MTN#, MNAME, MHGHT, CTRYCD, DIST} PK
[MTN#]

Relation **M5**: {BDATE, CID#, MTN#, EDATE} PK
[CID#, BDATE]

Step 4 — Obtain BCNF (and higher order) Relations

Next, we seek to obtain relations of higher order normal forms. Observe that relations **M1**, **M3**, **M4**, and **M5** are in BCNF, 4NF and 5NF.

Note: We could have forgone steps 1-3 and gone straight for BCNF relations by simply observing the FDs shown in Figure 5-16, and decomposing via Heath's theorem. As your confidence in database design grows, you will (hopefully) be able to do this.

5.9.2 Determining Candidate Keys and then Normalizing

In many cases, the database designer may be faced with the problem where basic knowledge of data to be stored is available, but it is not immediately clear how this partial knowledge will translate into a set of normalized relations. For instance, you may be able to identify an entity (or set of entities), but are not sure what the primary key(s) to this (these) entity (entities) will be. With experience, you will be able to resolve these challenges intuitively. However, what do you do in the absence of that invaluable experience? The relational model provides a theoretical approach for dealing with this problem, as explained in the following example.

Suppose that it is desirable to record the information about the performances of students in certain courses in an educational institution environment. Assume further, that a set of functional dependencies have been identified, but it is not sure what the final set of normalized relations will be and how they will be keyed. Figure 5-17 illustrates a summary of the information (assumed to be) known in the case. As usual, we start off by assuming that the relation shown (**StudPerfDraft**) is in 1NF.

Relation StudPerfDraft:

Course	[C]
Teacher	[T]
Hour	[H]
Room	[R]
Student	[S]
Grade	[G]

Functional Dependencies:

FD1:[H, R] → C
FD2: [H, T] → R
FD3: [C, S] → G
FD4: [H, S] → R
FD5: C → T

Figure 5-17. *Initial 1NF Relation for the Student Performance Problem*

Step 1 — Determine the Candidate Key

Having assumed that **StudPerfDraft** is in 1NF, our next step is to determine a candidate key of the relation. We do this by chasing the explicit and implicit dependencies. Any FD that ends up determining all attributes (directly or indirectly) constitutes a candidate key. The technique is referred to as *computing closures*.

The closure of an FD, denoted FD^+, is the set of all implied dependencies.

We shall examine each explicit FD in turn and determine all the attributes that it determines (explicitly or implicitly), bearing in mind that any attribute or combination of attributes that is a determinant, necessarily determines itself. Hence, we conclude the following:

- HR → CHR → CTHR
- HT → HTR → HTRC
- CS → CSG → CSGT
- HS → HSR → CHSR → CHSRG → CHSRGT
- C → CT

From this exercise, observe that [H,S] is the only candidate key; it is therefore the primary key (PK).

Step 2 — Obtain 2NF Relations

The next step is to obtain 2NF relations. We may rewrite the initial relation as follows:
StudPerfDraft {H, S, C, T, R, G} with PK [H,S]

Observe that **StudPerfDraft** is in 2NF.

Step 3 — Obtain 3NF Relations

Step 3 is to obtain 3NF relations. **StudPerfDraft** is not in 3NF due to FD5 (C $\rightarrow$T). To resolve this, decompose via Heath's Theorem to obtain:

R1 {H, S, C, R, G} with PK [H,S] and **R2** {C ,T} with PK [C].

Step 4 — Obtain BCNF Relations

Our fourth step is to obtain a set of BCNF relations. Observe that **R2** is in BCNF but **R1** is not, due to FD3 ([C,S] → G). To resolve this, decompose via Heath's theorem to obtain:

R3 {C, S,G} with PK [C,S] and **R4** {H, S, C, R} with PK [H,S].

R3 is now in BCNF but **R4** is not, due to FD1 ([H,R] → C). Decompose via Heath's theorem to obtain:

R5 {H, R, C} with PK [H,R] and **R6** {H, S, R} with PK [H,S].

We now have **R2**, **R3**, **R5**, and **R6** all in BCNF. Additionally, note that all the FDs have been resolved, except for FD2 ([H,T] → R). This may be resolved by introducing the relation **R7** as follows:

R7 {H, T, R} with PK [H, T].

Step 5 - Obtain Higher Order Normalized Relations

There are no MVDs or JDs, therefore relations **R2**, **R3**, **R5**, **R6** and **R7** are in 4NF and 5NF.

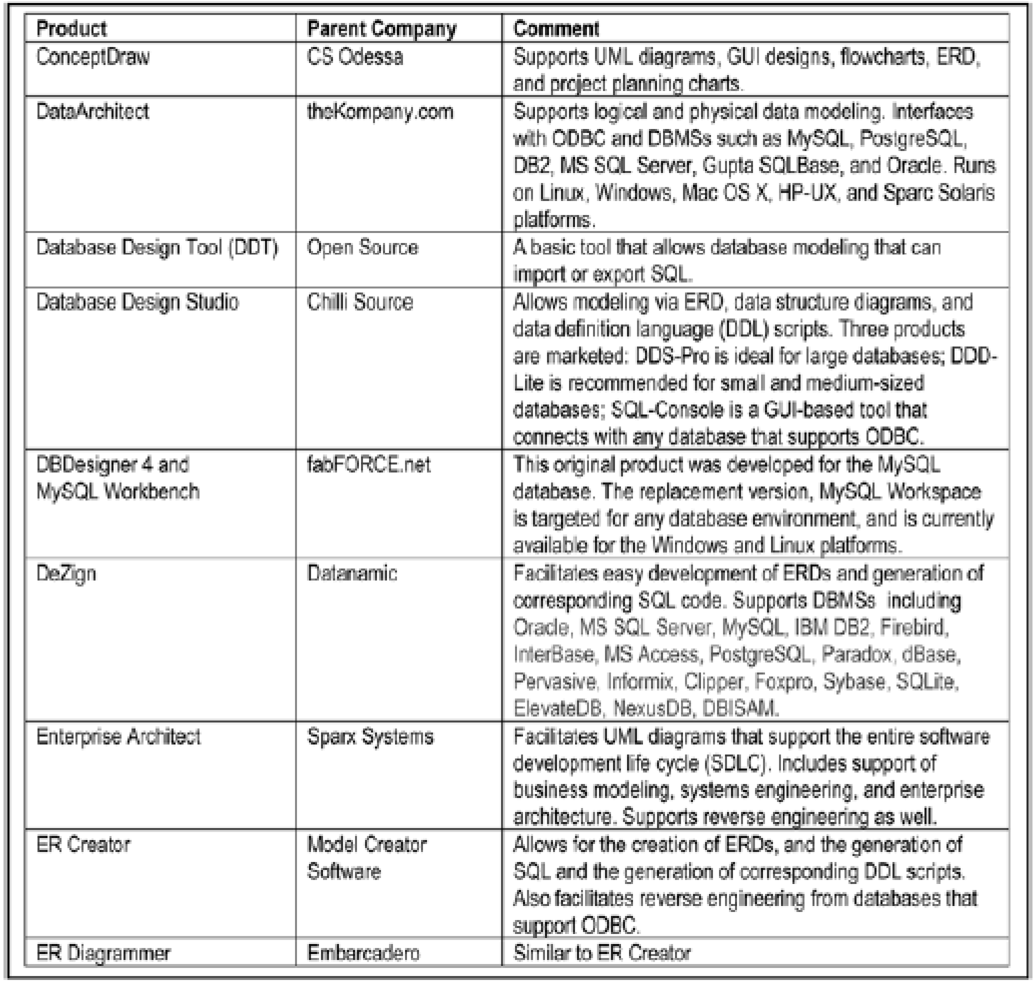
Note:

1. This is a rather trivial example; a college database is a much more complex system than the representation presented here. Moreover, questions may be raised as to the veracity of some of the FDs stated (for instance FD5). However, the representation succeeds in providing the application of the normalization theory, and that was the sole intent.

2. Relation **R6** fulfills FD4 ([H,S] → R), and **R7** fulfills FD2 ([H, T] → R). Strictly speaking, **R7** may be considered redundant, since we can determine what teacher is in a room at a given time by accessing **R5** and **R2**.

5.10 Database Model and Design Tools

At this point you must be wondering, how in the world are you supposed to model and design a complex database, and keep track of all the entities and relationships? The good news is, there are various tools that are readily available, so there's no need to panic. The standard general purpose word processors (such as MS Office, Word Perfect, Open Office, etc.) are all fortified with graphics capabilities so that if you spend a little time with any of these products, you will figure out how to design fairly impressive database model/design diagrams. Better yet, there is a wide range of CASE tools and/or modeling tools that you can use. Figure 5-18 provides an alphabetic list of some of the commonly used products. While some of the products in the list are quite impressive, the list is by no means comprehensive, so you do not have to be constrained by it. Some of the products are available free of charge; for others, the parent company offers free evaluation copies.



Product	Parent Company	Comment
ConceptDraw	CS Odessa	Supports UML diagrams, GUI designs, flowcharts, ERD, and project planning charts.
DataArchitect	theKompany.com	Supports logical and physical data modeling. Interfaces with ODBC and DBMSs such as MySQL, PostgreSQL, DB2, MS SQL Server, Gupta SQLBase, and Oracle. Runs on Linux, Windows, Mac OS X, HP-UX, and Sparc Solaris platforms.
Database Design Tool (DDT)	Open Source	A basic tool that allows database modeling that can import or export SQL.
Database Design Studio	Chilli Source	Allows modeling via ERD, data structure diagrams, and data definition language (DDL) scripts. Three products are marketed: DDS-Pro is ideal for large databases; DDD-Lite is recommended for small and medium-sized databases; SQL-Console is a GUI-based tool that connects with any database that supports ODBC.
DBDesigner 4 and MySQL Workbench	fabFORCE.net	This original product was developed for the MySQL database. The replacement version, MySQL Workspace is targeted for any database environment, and is currently available for the Windows and Linux platforms.
DeZign	Datanamic	Facilitates easy development of ERDs and generation of corresponding SQL code. Supports DBMSs including Oracle, MS SQL Server, MySQL, IBM DB2, Firebird, InterBase, MS Access, PostgreSQL, Paradox, dBase, Pervasive, Informix, Clipper, Foxpro, Sybase, SQLite, ElevateDB, NexusDB, DBISAM.
Enterprise Architect	Sparx Systems	Facilitates UML diagrams that support the entire software development life cycle (SDLC). Includes support of business modeling, systems engineering, and enterprise architecture. Supports reverse engineering as well.
ER Creator	Model Creator Software	Allows for the creation of ERDs, and the generation of SQL and the generation of corresponding DDL scripts. Also facilitates reverse engineering from databases that support ODBC.
ER Diagrammer	Embarcadero	Similar to ER Creator

***Figure 5-18.** Some Commonly Used Database Planning Tools*

Product	Parent Company	Comment
ERWin Data Modeler	Computer Associates	Facilitates creation and maintenance o data structures for databases, data warehouses, and enterprise data resources. Runs on the Windows platform. Compatible with heterogeneous DBMSs
MagicDraw	No Magic	A relatively new product that has just been introduced to the market. Appears to be similar to Enterprise Architect.
Oracle Designer	Oracle	Supports design for Oracle databases.
Oracle JDeveloper	Oracle	Supports UML diagramming.
Power Designer	Sybase	Supports UML, business process modeling, and data modeling. Integrates with development tools such as .NET, Power Builder (a Sybase Product), Java, and Eclipse. Also integrates with the major DBMSs.
SmartDraw	SmartDraw	A graphics software that facilitates modeling in the related disciplines of business enterprise planning, software engineering, database modeling, and information engineering (IE). Provides over 100 different templates (based on different methodologies) that you can choose from. Supported methodologies include UML, Chen Notation, IE Notation, etc.
TogetherSoft	Borland	Provides UML-based visual modeling for various aspects of the software development life cycle (SDLC). Allows generation of DDL scripts from the data model. Also supports forward and reverse engineering for Java and C++ code.
Toolkit for Conceptual Modeling (TCM)	University of Twente, Holland	Includes various resources for traditional software engineering methodologies as well as object-oriented methodologies based on the UML standards.
Visio	Microsoft	Facilitates modeling in support of business enterprise planning, software engineering, and database management.
Visual Thought	CERN	Similar to Visio but is free
xCase	Resolution Software	A database modeling tool that supports all aspects of the database development life cycle (DDLC): it supports ERD design, documentation, SQL code generation, logical and physical migration across multiple DBMS platforms, and data analysis.

Figure 5-18. *Some Commonly Used Database Planning Tools* (*continued*)

5.11 Summary and Concluding Remarks

It is now time to summarize what we have covered in this chapter:

- A database model is a blueprint for subsequent database design. It is a representation of the database. We have looked at three database models: the E-R model, the XR model and the UML model.
- Database design involves preparation of a database specification, which will be used to construct the database. We have discussed database design via the E-R model, the XR model, the UML model, the O/ESG, and normalization theory.

- The E-R model is the oldest model for relational databases that has been discussed. It involves the use of certain predefined symbols to construct a graphical representation of the database.
- Database design via the E-R model involves following eight steps that lead to a normalized database specification.
- The XR model is an alternate model that compensates for the weaknesses in the E-R model. It involves grouping information entities into different predefined categories that will assist in the design phase.
- Database design via the XR model involves following nine steps that lead to a normalized database specification.
- The UML model is similar to the E-R model. However, it requires taking an object-oriented approach, and employs different notations from the E-R model.
- Database design via the UML model involves following seven steps that lead to a normalized database specification.
- The O/ESG methodology describes an efficient way of developing a comprehensive, normalized database specification.
- Database design via normalization theory describes a rudimentary approach to database design that relies on mastery of the principles of normalization.

You are now armed with all the requisite knowledge needed to design quality databases. However, you will likely find that a review of this and the previous two chapters, along with practice, may be necessary until you have gained mastery and confidence with the concepts, principles and methodologies. The next chapter discusses the design of the user interface for a database system.