

## Conceptual Modeling: First Steps

- 3.1** Desirable features of conceptual modeling languages
- 3.2** Overview of ORM's conceptual schema design procedure (CSDP)
- 3.3** CSDP step 1: Verbalize information examples in terms of elementary facts
- 3.4** CSDP step 2: Draw the fact types; check that they can be populated with sample facts
- 3.5** CSDP step 3: Check for object types that should be combined; identify any fact types that can be derived arithmetically
- 3.6** Summary

### 3.1 Conceptual Modeling Language Criteria

Before discussing ORM's conceptual schema design procedure, let's review the fundamental design principles that underlie the ORM language itself. Some of these ideas were mentioned before, but I'll generalize the discussion here so that you can apply the principles to evaluate modeling languages in general. As you work your way through the book, you should consider how these principles are realized in the various languages discussed.

A modeling *method* comprises both a *language* and a *procedure* to guide modelers in using the language to construct models. This procedure is often called the *modeling process*. A language has associated *syntax* (marks), *semantics* (meaning), and *pragmatics* (use). Written languages may be graphical (diagrams) and/or textual. The terms "abstract syntax" and "concrete syntax" are sometimes used to distinguish underlying concepts (e.g., object type) from their representation (e.g., named ellipse).

Conceptual modeling portrays the application domain at a high level, using terms and concepts familiar to the application users, ignoring logical- and physical-level aspects (e.g., the underlying database or programming structures used for implementation) and external-level aspects (e.g., the screen forms used for data entry). The following criteria drawn from various sources (van Griethuysen 1982; ter Hofstede 1993; Bloesch and Halpin 1996) provide a basis for evaluating conceptual modeling languages:

- Expressibility
- Clarity
- Simplicity and orthogonality
- Semantic stability
- Semantic relevance
- Validation mechanisms
- Abstraction mechanisms
- Formal foundation

The *expressibility* of a language is a measure of what it can be used to say. The more features of the domain that the language can capture, the greater its expressive power. Ideally, a conceptual language should be able to completely model all details about the application domain that are conceptually relevant. This is called the *100% Principle* (van Griethuysen 1982). ORM is a method for modeling and querying an information system at the conceptual level, and for mapping between conceptual and logical levels. Although ORM extensions exist for process modeling, the focus of ORM is on information modeling (popularly known as data modeling), since the data perspective is more stable and provides a formal foundation on which operations can be defined. Overall, UML can express more than standard ORM, since UML use case, behavior, and implementation diagrams model aspects beyond static structures. For conceptual data modeling, however, ORM's diagram notation has much greater expressive power than UML class diagrams or ER diagrams.

The *clarity* of a language is a measure of how easy it is to understand and use. To begin with, the language should be unambiguous. The more expressible a language is,

the harder it is to maintain clarity. Ideally, the meaning of diagrams or textual expressions in the language should be intuitively obvious. This ideal is rarely achieved. A more realistic goal is that the language concepts and notations should be easily learned and remembered. To meet this goal, a language should exhibit *simplicity and orthogonality*. By avoiding attributes, ORM's role-based notation is simplified, yet easily understood by populating it with fact instances. Orthogonality allows use of an expression wherever its meaning or value may be used. ORM's constructs were designed from the ground up to be orthogonal. For example, ORM constraints can be used and combined whenever this is meaningful. As we will see later, this is not true of languages like UML.

*Semantic stability* is a measure of how well models or queries expressed in the language retain their original intent in the face of changes to the application. The more changes we are forced to make to a model or query to cope with an application change, the less stable it is. Models and queries in ORM are semantically more stable than in ER or UML since they are not impacted by changes that cause attributes to be remodeled as relationships or vice versa.

*Semantic relevance* requires that only conceptually relevant details need be modeled. Any aspect irrelevant to the meaning (e.g., implementation choices, machine efficiency) should be avoided. This is called the *conceptualization principle* (van Griethuysen 1982). Section 13.3 shows how conceptual queries in ORM meet this criterion better than queries based on attribute-based models.

*Validation mechanisms* are ways in which domain experts can check whether the model matches the application. For example, static features of a model may be checked by verbalization and multiple instantiation, and dynamic features may be checked by simulation. Unlike ER and UML, ORM models are always easily verbalized and populated.

*Abstraction mechanisms* allow unwanted details to be removed from immediate consideration. This is very important with large models (e.g., wall-size schema diagrams). ORM diagrams tend to be more detailed and larger than corresponding ER or UML models, so abstraction mechanisms are often used. For example, a global schema may be modularized into various scopes or views based on span or perspective (e.g., a single page of a data model or a single page of an activity model). Successive refinement may be used to decompose higher-level views into more detailed views. Though not a language issue, tools can provide additional support such as layering and object zoom (see Section 13.4). Such mechanisms can be used to hide and show just that part of the model relevant to a user's immediate needs. With minor variations, these techniques can be applied to ORM, ER, and UML. ORM also includes an attribute abstraction procedure to generate ER and UML diagrams as views.

A formal foundation is needed to ensure unambiguity and executability (e.g., to automate the storage, verification, transformation, and simulation of models) and to allow formal proofs of equivalence and implication between models. Although ORM's richer, graphical constraint notation provides a more complete diagrammatic treatment of schema transformations, use of textual constraint languages can partly offset this advantage. For their data modeling constructs, ORM, ER, and UML have an adequate formal foundation.

Bentley (1998) suggests the following alternative yardsticks for language design: orthogonality, generality, parsimony, completeness, similarity, extensibility, and openness. Some of these criteria (e.g., completeness, generality, extensibility) may be subsumed under expressibility. Parsimony may be treated as one aspect of simplicity. Another criterion sometimes mentioned is convenience (how convenient, suitable, or appropriate a language feature is to the user). We can treat convenience as another aspect of simplicity.

Language design often involves a number of *trade-offs* between competing criteria. One well-known trade-off is that between *expressibility and tractability*: the more expressive a language is, the harder it is to make it efficiently executable (Levesque 1984). Another trade-off is between *parsimony and convenience*. Although *ceteris paribus*, the fewer concepts the better (cf. Occam's razor), restricting ourselves to the minimum possible number of concepts may sometimes be too inconvenient. For example, it is possible to write computer programs in pure binary, using just strings of ones and zeros, but it's much easier to write programs in higher-level languages like C#. As an example from logic, it's more convenient to use several operators such as "not", "and", "or", and "if-then" even though we could use just one (e.g., "nand"). See the chapter notes for further discussion of this example.

One basic question relevant to the parsimony-convenience trade-off is whether to use the attribute concept as a base modeling construct. I've already argued in favor of a negative answer to this question. ORM models attributes in terms of relationships in its base model (used for capturing, validating, and evolving the conceptual schema), while still allowing attribute views to be displayed in derived models (in this case, compact views used for summary or implementation purposes). Traditional ER supports single-valued attributes, while UML supports both single-valued and multivalued attributes. Section 9.3 argues that multivalued attributes are especially bad for conceptual modeling, although they can be useful for logical and physical modeling.

The rest of this chapter provides an overview of ORM's conceptual schema design procedure and a detailed discussion of the first three steps in this procedure. The ORM approach has been used productively in industry for over 25 years; details of its history can be found in the chapter notes.

## 3.2 ORM's Conceptual Schema Design Procedure

When developing an information system, we first specify what is required and produce a design to meet these requirements. For reasons outlined earlier, I recommend first developing this design at the conceptual level using Object-Role Modeling. This entails describing the structure of the UoD formally in terms of an ORM conceptual schema. If an ORM context is understood, I'll often shorten "ORM conceptual schema" to just "conceptual schema".

The procedure for designing a conceptual schema that is small enough to manage as a single unit is referred to as the **conceptual schema design procedure** (CSDP). With large applications, the universe of discourse is divided into components or subsections (which may overlap). These are normally prioritized to determine which ones to

develop first, and a conceptual *subschema* is designed for each. With a team of modelers, a consensus on terminology is reached, so the same names are used for the same concepts. Later the subschemas are *integrated* or merged into a global conceptual schema that covers the whole UoD. This integration is often performed iteratively. This top-down design approach can be summarized as follows:

- Divide the universe of discourse into manageable subsections.
- Apply the CSDP to each subsection.
- Integrate the subschemas into a global conceptual schema.

For each manageably sized application, the conceptual schema design is performed in **seven steps**:

1. Transform familiar information examples into elementary facts, and apply quality checks.
2. Draw the fact types, and apply a population check.
3. Check for entity types that should be combined, and note any arithmetic derivations.
4. Add uniqueness constraints, and check arity of fact types.
5. Add mandatory role constraints, and check for logical derivations.
6. Add value, set comparison, and subtyping constraints.
7. Add other constraints and perform final checks.

Often all seven steps are performed for each model component as it is discussed with the domain expert, rather than applying step 1 to all components, then step 2, and so on.

The procedure begins with the analysis of examples of information to be output by, or input to, the information system. Basically, the first three steps are concerned with identifying the fact types. In later steps we add constraints to the fact types. Throughout the procedure, checks are performed to detect derived facts and to ensure that no mistakes have been made. In the rest of this chapter we consider the first three steps in detail.

With large applications, the preliminary segmentation and final integration add two further stages, resulting in nine steps overall. Although the CSDP is best learned in the sequence of steps shown, in practice we might apply the steps somewhat differently. For example, we might add constraints as soon as the fact types are entered, rather than waiting for all the fact types to be entered before adding any constraints.

In the commercial world, there are many existing applications that have been implemented using lower-level approaches, resulting in database designs that may be inconsistent, incomplete, inefficient, or difficult to maintain. Such systems are often poorly documented. These problems can be overcome by **reengineering** the existing applications using conceptual modeling techniques. For example, sample populations from existing database tables can be used as input to the CSDP.

Even without sample populations, an existing database schema can be *reverse-engineered* to a tentative conceptual schema by using information about constraints and domains and making simplifying assumptions about use of names. The conceptual design can then be validated and completed by communicating with a domain expert. The conceptual schema can then be *forward-engineered* by applying a conceptual

optimization procedure and then mapping to the target database system to provide an improved and maintainable implementation. This reengineering approach is discussed in Chapter 12.

### 3.3 CSDP Step 1: From Examples to Elementary Facts

To specify what is required of an information system, we need to answer the question *What sort of information do we want from the system?* Clearly, any information to be output from the system must either be stored in the system or be derivable by the system. Our first step is to begin with *familiar examples* of relevant information, and *express these in terms of elementary facts*. As a check on the quality of our work, we ask the following questions. *Are the entities well identified? Can the facts be split into smaller ones without losing information?* This constitutes step 1 of the conceptual schema design procedure.

#### ***CSDP step 1: Transform familiar examples into elementary facts, and apply quality checks.***

For process modeling, it helps to begin with examples of the processes to be carried out by the system. Such process examples have long been used in industry, but in 1987 they were coined “use cases”, which nicely suggests cases of the system being used, and this term has stuck. UML recommends use cases to drive the modeling process. Although use cases help with designing process models, in practice the move from use cases to data models is often somewhat arbitrary and frequently results in data models that need substantial reworking.

The solution is clear. If you want to get the data model right, start with examples of the *data* to be delivered by the system. By analogy with the UML term, I now call these “data use cases”, since they are cases of data being used. However, this is just another name for the “familiar information examples” concept that was introduced in step 1 of the ORM conceptual schema design procedure back in the 1970s. If you still want to use UML’s process use cases to drive the modeling process, you should at least flesh them out with associated data samples before working on the class diagrams.

If we are designing a conceptual schema for an application previously handled manually or by computer, information examples will be readily available. If not, we work with the domain expert to provide examples. Two important types of examples are output reports and input forms. These might appear as tables, forms, diagrams, or text. To verbalize such examples in terms of elementary facts, we need to understand what an **elementary fact** is.

To begin with, an elementary fact is a simple assertion, or atomic proposition, about the UoD. The word “fact” indicates that the system is to treat the assertion as being true of the UoD. Whether this is actually the case is of no concern to the system. In everyday speech, facts are true statements about the real world. However, in computing terminology we resign ourselves to the fact(!) that it is possible to have “false facts” in

the database (just as we use the word “statement” for things that aren’t really statements in languages like SQL).

We may think of the UoD as a set of *objects playing roles*. Elementary facts are assertions that particular objects play particular roles. The simplest kind of elementary fact asserts that a single object plays a given role. For example, consider a very small domain in which people can be identified by their first names. One fact about this domain might be

1. Ann smokes.

Here we have one object (Ann) playing a role (smokes). Strictly, we should be more precise in identifying objects (e.g., expand “Ann” to “the person with firstname ‘Ann’”), but let’s tidy up later. With sentences like (1), the role played by the object is sometimes called a *property* of the object. Here an elementary fact asserts that a certain object has a certain property. This is also called a *unary relationship*, since only one role is involved. Usually, however, a relationship involves at least two roles. For example:

2. Ann employs Bob.
3. Ann employs Ann.

In (2) Ann plays the role of employer and Bob plays the role of employee. In (3) Ann is self-employed and plays both roles. In general, *an elementary fact asserts that a particular object has a property, or that one or more objects participate together in a relationship*.

The adjective “elementary” indicates that the fact *cannot be “split” into smaller units of information* that collectively provide the same information as the original. Elementary facts typically do not use logical connectives (e.g., **not**, **and**, **or**, **if**) or logical quantifiers (e.g., **all**, **some**). For example, sentences (4)–(9) are not elementary facts:

4. Ann smokes **and** Bob smokes.
5. Ann smokes **or** Bob smokes.
6. Ann does **not** smoke.
7. **If** Bob smokes **then** Bob is cancer prone.
8. **All** people who smoke are cancer prone.
9. **If some** person smokes **then that** person is cancer prone.

All of these sentences express information. Proposition (4) is a logical conjunction. It should be split into two elementary facts: Ann smokes; Bob smokes. Proposition (5) is a disjunction, (6) is a negation, and (7) is a conditional fact: most database systems do not allow such information to be stored conveniently and are incapable of making relevant inferences (e.g., deducing that Bob smokes from the combination of (5) and (6)). For most commercial applications, there is no need to store such information.

Often the absence of positive information (e.g., Ann smokes) is taken to imply the negative (Ann does not smoke): this is the usual “closed-world” assumption. With an “open-world” approach, negative information can be explicitly stored using negative predicates or status object types, in conjunction with suitable constraints. For example,

the roles “smokes” and “is a nonsmoker” are mutually exclusive, and the fact type Person has SmokerStatus {‘S’, ‘NS’} requires the constraint that **each** person has **at most one** SmokerStatus. Sometimes the choice of whether to store positive or negative information depends on which occupies less space, and the borderline between positive and negative may become blurred (e.g., consider Person dislikes Food and Patient is allergic to Drug). These topics are discussed further in later chapters.

Universally quantified conditionals like (8) and (9) may be catered to either in terms of a subset constraint (see later) or by a derivation rule. Such rules can be specified readily in SQL by means of a view and are also easily coded in languages such as Prolog. For example: `cancerProne(X) if person(X) and smokes(X)`.

Elementary facts assert that objects play roles. How are these objects and roles specified? For now we consider only basic objects: these are either *values* or *entities*. For our work it is sufficient to recognize two kinds of **values**: *character string* and *number*. These are identified by constants. Character strings are shown inside single quotes (e.g., ‘USA’). Numbers are denoted without quotes, using the usual Hindu-Arabic decimal notation (e.g., 37 or 5.2). Numbers are abstract objects denoted by character strings called numerals. For example, the number 37 is denoted by the numeral ‘37’. We assume that any information system supports strings and numbers as built-in data types. Values are displayed textually, but are internally represented by bit strings.

Conceptually, an **entity** (e.g., a particular person or car) is referenced in an information system by means of a *definite description*. For example, kangaroos hop about on an entity identified as “the Country with name ‘Australia’”. Entities may also be called “described objects”. Unlike values, some entities can change with time. An entity may be a tangible object (e.g., the City with name ‘Paris’) or an abstract object (e.g., the Subject with code ‘CS114’). We consider both entities and values to be objects that exist in the UoD. Note that object-oriented approaches use the term “object” in the more restrictive sense of “entity”. Usually we want to talk about just the entities, but to reference them we make use of values. Sometimes we want to talk about the values themselves. Consider the following sentences:

10. Australia has six states.
11. “Australia” has nine letters.

Here we have an illustration of what logicians call the use/mention distinction. In (10) the word “Australia” is being used as a label to reference an entity. In (11) the word “Australia” is being mentioned and refers to itself. In natural language, quotes are used to resolve this distinction. In everyday talk, entities are often referred to by a *proper name* (e.g., “Bill Clinton”) or by some definite description (e.g., “the previous president of the USA” or “the president named ‘Bill Clinton’”). Proper names work if we can decide what the name refers to from the context of the sentence. For example, in (10) you probably took “Australia” to refer to the country named “Australia”. However, the sentence itself does not tell you this. Perhaps (10) was talking about a dog named “Australia” who has six moods (sleepy, playful, hungry, etc.).

Since humans may misinterpret, and information systems lack any creativity to add context, we play it safe by demanding that entities be clearly identified by special kinds of definite descriptions. To begin with, the description must specify the kind of entity

being referred to: the **entity type**. A *type* is the set of all possible *instances*. Each entity is an instance of a particular entity type (e.g., Person, Subject). For a given UoD, the entity type Person is the set of all people we might possibly want to talk about during the lifetime of the information system. Note that some authors use the word “entity” for “entity type”. I’ll sometimes expand “entity” to “entity instance” to avoid any confusion. Consider the sentence

12. Lee is located in 10B.

This could be talking about a horse located in a stable, or a computer in a room, and so on. By stating the entity types involved, (13) avoids this kind of referential ambiguity. Names of object types are highlighted here by starting them with a capital letter.

13. The Patient ‘Lee’ is located in the Ward ‘10B’.

This brings to mind the old joke: “*Question*: Did you hear about the man with the wooden leg named ‘Smith’? *Answer*: No—What was the name of his other leg?” Here the responder mistakenly took the label “Smith” to refer to an entity of type WoodenLeg rather than of type Man. Sometimes, even stating the entity type fails to fully clarify the situation. Consider the following sentence:

14. The Patient ‘Lee’ has a Temperature of 37.

Now imagine that the UoD contains two patients named “Lee Jones” and “Mary Lee”. There is more than one person to which the label ‘Lee’ might apply. Worse still, there may be some confusion about the units being used to state the temperature: 37 degrees Celsius is normal body temperature, but 37 degrees Fahrenheit is close to freezing! We resolve this ambiguity by including the **reference mode** (i.e., the manner in which the value refers to the entity). Compare the following two sentences:

15. The Patient with surname ‘Lee’ has a Temperature of 37 Celsius.

16. The Patient with firstname ‘Lee’ has a Temperature of 37 Fahrenheit.

A more common way around the potential confusion caused by overlap of first names and surnames would be to demand that a longer name be used instead (e.g., “Lee Jones”, “Mary Lee”). In some cases, however, even these names may not be unique, and another naming convention must be employed (e.g., PatientNr). To avoid confusing “No.” with the word “No”, I’ll use “Nr” or “#” to abbreviate “Number”. Most entity designators involve three components:

- *Entity type* (e.g., Patient, Temperature)
- *Reference mode* (e.g., surname, Celsius)
- *Value* (e.g., ‘Lee’, 37)

This is the simplest kind of entity designation scheme. I’ll restrict our discussion of reference schemes to this simple case for quite some time. Composite identification schemes are considered later.

Now that we know how to specify objects, how do we specify the roles they play? We use logical **predicates**. In logic, a predicate is basically a *sentence with object holes in it*. To complete the sentence, the object holes or placeholders are filled in by *object*

*terms*. Each object term refers to a single object in the UoD. Object terms are also called singular terms or object designators. For us, values are designated by constants (sometimes preceded by the value type name), and entities are designated by definite descriptions that relate values to entities. Consider the following sentence:

17. The Person with firstname ‘Ann’ **smokes**.

Here the object term is “The Person with firstname ‘Ann’”, and the predicate identifier is shown in bold. The predicate may be shown by itself as

... smokes

This is a *unary predicate*, or *sentence with one object hole* in it. It may also be called a property or a unary relationship type. A *binary predicate* is a *sentence with two object holes*. Consider this example:

18. The Person with firstname ‘Ann’ **employs** the Person with firstname ‘Bob’.

Here the predicate may be shown as

... employs ...

Notice that the *order* in which the objects are placed here is important. For example, even though Ann employs Bob, it may be false that Bob employs Ann. A *ternary predicate* is a *sentence with three object holes*. For instance, the fact that Terry worked in the Computer Science Department for 10 years involves the predicate

... worked in ... for ...

In general, an *n-ary predicate* is a sentence with *n* object holes. Since the order is significant, a filled-in *n-ary predicate* is associated with a *sequence* of *n* object terms, not necessarily distinct. The value of *n* is the **arity**, or degree, of the predicate. Predicates of arity  $\geq 2$  are polyadic. An elementary fact may now be thought of as asserting a proposition of the form

$$R o_1 \dots o_n$$

where *R* is a predicate of arity *n*, and *o*<sub>1</sub> ... *o*<sub>*n*</sub> are *n* object terms, not necessarily distinct. Moreover, with respect to the UoD the proposition must not be splittable into a conjunction of simpler propositions. This definition ties in with the notation of *predicate logic*.

For naturalness, we write predicates in *mixfix* (or *distfix*) form, where the terms may be mixed in with the predicate. For example, the following ternary fact uses the predicate “... moved to ... during ...”.

19. The Scientist with surname ‘Einstein’ **moved to** the Country with code ‘USA’ **during** the Year 1933 AD.

Step 1 of the CSDP involves translating relevant information examples into sentences like this. As a simple example, consider the output report of Table 3.1. Try now to express the information in the first row in the form of elementary facts. To help with this, use the *telephone heuristic*. Imagine you have to convey the information over the

**Table 3.1** Some languages and their designers.

<i>Designer</i>	<i>Language</i>
Wirth	Pascal
Kay	Smalltalk
Wirth	Modula-2

telephone to someone. In performing this visual-to-auditory transformation, make sure you fully specify each entity in terms of its entity type, reference mode, and value, and that you include the predicate name.

In reports like this, the column headings and table names or captions often give a clue as to the object types and predicates. The column entries provide the values. Here is one way of translating row 1 as an elementary fact:

20. The Person with surname ‘Wirth’ **designed** the Language named ‘Pascal’.

Notice that the entity types and reference modes appear as nouns and the predicate as a verb phrase. This is fairly typical. In translating row 1 into the elementary fact (20), we read the row from left to right. If instead we read it from right to left, we might say:

21. The Language named ‘Pascal’ **was designed by** the Person with surname ‘Wirth’.

In reversing the order of the terms, we also reversed the predicate. We speak of “was designed by” as the *inverse* of the predicate “designed”. Although semantically we might regard sentences (20) and (21) as expressing the same fact, syntactically they are different. Most logicians would describe this as a case of two different sentences expressing the same proposition. Linguists like to describe this situation by saying the two sentences have different *surface structures* but the same deep structure.

For example, one linguistic analysis might portray the *deep structure* sentence as comprising a verb phrase (Design), various noun phrases (the object terms) each of which relate to the verb in a different case (e.g., agentive for Wirth and objective for Pascal), together with a modality (past tense). Different viewpoints exist as to the “correct” way to portray deep structures (e.g., what primitives to select), and the task of translation to deep structures is often complex. In practice, most information systems can be designed without delving further into such issues.

It is important not to treat sentences like (20) and (21) as different, unrelated facts. Our approach with binary fact types is to choose one standard way of stating the predicate, but optionally allow the inverse reading to be shown as well. For example:

22. The Person with surname ‘Wirth’ **designed / was designed by** the Language named ‘Pascal’.

Here the predicate on the left of the slash “/” is used for the standard (left-to-right) reading (20). The predicate on the right of the slash is used for the inverse reading (21). The slash visually suggests jumping over the other predicate when reading left to right,

**Table 3.2** An output report about marriages.

<i>Married couples</i>	
Adam	Eve
Jim	Mary

and jumping under the other predicate when reading right to left. Having two ways to talk about a binary fact type can help communication and can simplify constraint specification. For example, the specification “each Language was designed by some Person” is preferable to the equivalent “for each Language, some Person designed that Language”. For  $n$ -ary fact types there are many possible orderings, but only one is displayed at a time.

Now consider Table 3.2. This is a bit harder to verbalize since the columns don’t have separate names. You may assume that Adam and Jim are males and that Eve and Mary are females. As an exercise, verbalize the top row in terms of elementary facts before reading on.

Perhaps you verbalized this as shown in (23). For completeness, the inverse is included.

23. The Person with firstname ‘Adam’ **is married to / is married to** the Person with firstname ‘Eve’.

Notice that the forward predicate is the same as the inverse. This is an example of a *symmetric* relationship. Such relationships create special problems (as discussed in Section 7.3). To help avoid such problems, at the conceptual level *no base predicate should be the same as its inverse*. You can always rephrase the fact to ensure this. For example, (24) does this by highlighting the different roles played by each partner.

24. The Person with firstname ‘Adam’ **is husband of / is wife of** the Person with firstname ‘Eve’.

As another example, consider Table 3.3. This is like our earlier Table 3.1 but with an extra column added. Try to express the information on the first row in terms of elementary facts before reading on.

We might at first consider expressing this information as sentence (25), using the ternary predicate “... designed ... in ...”. Do you see any problems with this?

25. The Person with surname ‘Wirth’ **designed** the Language named ‘Pascal’ **in** the Year 1971 AD.

Recall that an elementary fact must be simple or irreducible. It cannot be split into two or more simpler facts in the context of the UoD. The appearance of the word “and” in a sentence usually indicates that the sentence may be split into simpler facts. Here there is no “and”, but “common sense” tells us that the fact can be split into the following two elementary facts with no information loss:

**Table 3.3** Origin details of some programming languages.

<i>Designer</i>	<i>Language</i>	<i>Year</i>
Wirth	Pascal	1971
Kay	Smalltalk	1972
Wirth	Modula-2	1979

26. The Person with surname ‘Wirth’ designed the Language named ‘Pascal’.

27. The Language named ‘Pascal’ was designed in the Year 1971 AD.

Here “no information loss” means that if we know (26) and (27), then we also know (25). The phrase “common sense” hides some formal ideas. In order to split (25) into (26) and (27) we probably relied on our implicit understanding that each language was designed in only one year. This constraint holds if we interpret “was designed in” to mean “had its design completed in”. Let us agree with this interpretation.

If instead we meant “had work done on its design”, then a language may be designed in many years. In this case, we could still justify the split if each language had only one designer or at least the same set of designers for each year. But this might not be true. For example, if we include UML as a language, it had different designers in different years. This illustrates the need to be clear about the meaning of our wording and to strive for significant sample data.

Later steps in the CSDP add formal checks to detect the relevant dependencies, so if our “common sense” fails us here, we will normally see this error at a later stage. For now, though, let’s work with our intuitions. Suppose we split the ternary into the two binaries: Person designed Language; Person completed design in Year. Would this be acceptable? As an exercise, use the table’s population to show that this kind of split would actually lose information.

After plenty of practice at step 1, you may wish to write the elementary facts down in abbreviated form. To start with, you can drop words such as “the” and “with” where they introduce object types and reference modes. Reference modes are placed in parentheses after the object types. Some versions of ORM append a “+” to indicate the referencing value is numeric, and hence may appear in addition (+) and other numeric operations.

You might also shorten some identifiers used for object types, reference modes, and predicates so long as the shorter names are still meaningful to you. Don’t forget to start the name of object types with a capital letter. Start the name of reference modes with a small letter, unless capitals have significance (e.g., “AD”). For example, facts (26) and (27) may be set out more concisely as (26’) and (27’).

26’. Person (surname) ‘Wirth’ designed Language (name) ‘Pascal’.

27’. Language (name) ‘Pascal’ was designed in Year (AD) 1971.

If the reference schemes for entity types are declared up front, they may be omitted in setting out the facts. For example, (26) and (27) may be specified as

*Reference schemes:* Person (surname); Language (name); Year (AD)

*Facts:* Person 'Wirth' designed Language 'Pascal'.  
Language 'Pascal' was designed in Year 1971.

Even more conveniently, a fact type may be displayed in diagram form (see next section), and fact instances may be entered into the relevant fact table simply by entering values.

The task of defining a formal grammar sufficient to capture any sentence expressed in natural language is daunting, partly because of the many ways in which objects may be referenced. For example, consider the sentence: “The next person to step on my toe will cop it”. Some artificial intelligence research is directed toward sorting out the semantics in sentences like this.

Fortunately for us, such sentences don't appear in database tables, where simple value-based schemes are used to reference objects. ORM is capable of formally capturing the relevant semantics of any fact that can be represented in a database table. Structured object terms and predicates provide the logical deep structure, independent of the natural language (English, Japanese, etc.) used to express the fact. By supporting ordered, mixfix predicates ORM enables this deep structure to be expressed in a surface structure in harmony with the ordered, mixfix nature of natural language. For example, consider Figure 3.1.

Here the two tables convey the same fact in different languages. The fact may be expressed in English as (28) and in Japanese as (29). The reference modes are italicized and the predicates are in bold.

28. The Employee with *employeeNr* '37' **works in** the Department with *name* 'Sales'.

29. Jugyo in *jugyo in bango* '37' **wa** 'Eigyo' to iu *namae* no Ka **ni shozoku suru**.

These are parsed into the structures shown in 28' and 29'. They have the same deep structure. Object terms are enclosed in square brackets. The infix predicate “... works in ...” corresponds to the mixfix predicate “... wa ... ni shozoku suru”.

28'. [Employee (*employeeNr*) '37'] **works in** [Department (*name*) 'Sales'].

29'. [Jugyo in (*jugyo in bango*) '37'] **wa** [Ka (*namae*) 'Eigyo'] **ni shozoku suru**.

Although ordered, mixfix predicates are preferred for naturalness, you could also treat a fact as a named set of (object, role) pairs:  $F\{(o_1, r_1), \dots, (o_n, r_n)\}$ . Here each object  $o_i$  is paired with the role  $r_i$  that it plays in the fact  $F$ . For example, (22) might be specified as Design{ (The Person with surname 'Wirth', agentive), (The Language named 'Pascal', objective) }. Instead of the case adjectives “agentive” and “objective”, other role names could be used (e.g., “designer” and “language” or “designing” and “being

<i>EmployeeNr</i>	<i>Department</i>	<i>Jugyo in</i>	<i>Ka</i>
37	Sales	37	Eigyo

**Figure 3.1** The same fact in English and Japanese.

designed by”). By pairing objects with their roles, the order in which the pairs are listed is irrelevant. This approach is used in RIDL (Reference and Idea Language), as discussed in Section 13.3.

Now consider the output report of Table 3.4, and try to express the information contained in its top row in terms of one or more elementary facts. Here the table itself has a name (“Result”) that can help us with the interpretation.

To save writing later, let us declare the reference schemes: Student (name); Subject (code); and Rating (nr). Because this table looks similar to Table 3.3, you might have been tempted to try to split the information into two facts. For example:

- 30. The Student ‘Bright S’ **studied** the Subject ‘CS112’.
- 31. The Student ‘Bright S’ **scored** the Rating 7.

This approach is incorrect because it results in loss of information. Since Bright studies more than one subject, we don’t know for sure from these two facts that Bright’s 7 rating is for CS112. In some cases a ternary that is not splittable into two facts may be split into three facts. Here we might try to split the information into the above two facts as well as

- 32. The Rating 7 **was obtained for** the Subject ‘CS112’.

However, even these three facts don’t guarantee that Bright got a 7 for CS112. For example, Jones studied CS112, Jones scored a 7, and a 7 was obtained for CS112, but Jones did not score a 7 in CS112. So the whole of the first row should be expressed as one elementary fact. For example, using the predicate “... for ... scored ...” we obtain

- 33. The Student ‘Bright S’ **for** the Subject ‘CS112’ **scored** the Rating 7.

Notice that we chose to think of a rating not as a number, but as an entity that is referenced by a number. This has two advantages. Suppose that students are rated numerically on their exam performance and are also rated numerically on their popularity. In this case we have the object types ExamRating and PopularityRating. Is an exam rating of 7 the same thing as a popularity rating of 7? I would answer, “No, only the numbers are the same”.

If we treat ratings as numbers we would have to answer “Yes” to the previous question. I prefer to think of ratings as entities rather than as numbers. Another advantage of this approach is that decisions about preferred reference schemes can be delayed until

**Table 3.4** A relational table for storing student results.

*Result:*

<i>student</i>	<i>subject</i>	<i>rating</i>
Bright S	CS112	7
Bright S	CS100	6
Collins T	CS112	4
Jones E	CS100	7
Jones E	CS112	4
Jones E	MP104	4

**Table 3.5** A relational table storing facts about lecturers.

<i>Lecturer:</i>	<i>name</i>	<i>birthyear</i>	<i>age</i>	<i>degrees</i>
	Adams JB	1946	46	BSc, PhD
	Leung X	1960	32	BE, MSc
	O'Reilly TA	1946	46	BA, BSc, PhD

**Table 3.6** An output report indicating tutorial allocations.

<i>Tute group</i>	<i>Time</i>	<i>Room</i>	<i>StudentNr</i>	<i>Student name</i>
A	Mon 3 p.m.	CS-718	302156 180064 278155 334067 200140	Bloggs FB Fletcher JB Jackson M Jones EP Kawamoto T
B1	Tues 2 p.m.	E-B18	266010 348112	Anderson AB Bloggs FB
...	...	...	...	...

the conceptual schema is to be mapped to a logical schema. For example, exam ratings are often given in both numbers and codes. A 7 might correspond to “HD” (high distinction). With the suggested approach, the same rating is involved no matter which way we reference it.

As another example, consider the report shown in Table 3.5. Try step 1 yourself on this table before reading on. One of the tricky features of this table is the final column. Entries in this column are sets of degrees, not single degrees.

You should phrase your sentences to include only one degree at a time. When applied to the first row, step 1 results in four facts that may be set out as follows.

34. The Lecturer with name ‘Adams JB’ **was born in** the Year 1946 AD.
35. The Lecturer with name ‘Adams JB’ **has** the Age 46 years.
36. The Lecturer with name ‘Adams JB’ **holds** the Degree with code ‘BSc’.
37. The Lecturer with name ‘Adams JB’ **holds** the Degree with code ‘PhD’.

Here the entity types and reference modes are Lecturer (name), Year (AD), Age (years), and Degree (code). There are three fact types: Lecturer **was born in** Year; Lecturer **has** Age; Lecturer **holds** Degree. The entity types Year and Age are semantically different. For example, Year involves a starting point in time, whereas Age is merely a duration of time.

A harder example is shown in Table 3.6. This is an extract of a report listing details of tutorial groups. Perform step 1 yourself before reading on.

Since verbalization of an example involves interpretation, it is important that the kind of example is *familiar* to us or another person (e.g., the domain expert) who is

assisting us with step 1. Since domain experts often lack technical expertise in modeling, we should not expect them to do the whole of step 1 themselves. It is sufficient if they verbalize the information correctly in their own terms. This is **step 1a: *Verbalize the information***. As modelers we can then refine their verbalization by ensuring the facts are elementary and the objects are well identified. This is **step 1b: *Verbalize the information as elementary facts***. For example, a domain expert might perform step 1a by verbalizing the information on the top row of this report as

Tute group A meets at 3 p.m. Monday in Room CS-718.  
Student 302156 belongs to group A and is named “Bloggs F”.

We might then perform step 1b by refining these informal sentences into the following four elementary sentences:

38. The TuteGroup with code ‘A’ **meets at** the Time with dayhrcode ‘Mon 3 p.m.’.
39. The TuteGroup with code ‘A’ **meets in** the Room with roomNr ‘CS-718’.
40. The Student with studentNr 302156 **belongs to** the TuteGroup with code ‘A’.
41. The Student with studentNr 302156 **has** the StudentName ‘Bloggs FB’.

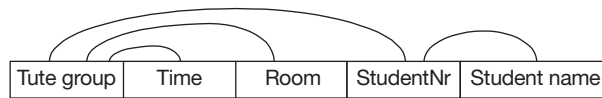
Many features in this example rely on interpretation. For instance, I assumed that StudentNr and StudentName refer to students, and that a student number and student name on the same row refer to the same student. I also filled in the associations as “meets at”, “meets in”, “belongs to”, and “has”. The report itself does not tell us this. We use our background familiarity with the situation to make such assumptions.

Decisions were made about entity types and reference schemes. For example, I chose to think of Time as an entity type referenced by a day-hour code rather than introducing Day and Hour as separate entity types. A similar comment applies to Room and StudentName.

StudentNr was chosen rather than StudentName to identify Student. The report helps here since StudentNr appears first (on the left) and ‘Bloggs FB’ appears with two different student numbers. But we are still making assumptions (e.g., that students have only one student number or that students belong to only one group). Another major assumption is that each tutorial group meets only once a week. We need to know this to justify using separate facts for the time and room (rather than verbalizing this as TuteGroup meets at Time in Room). Of course the fact that group A is not repeated in the report helps us with this decision, but this still assumes the sample is representative in this regard.

Since interpretation is always involved in the initial step, if we are not familiar with the example we should resolve any doubts by asking a person who is familiar with the UoD. Communication with domain experts should use examples familiar to them. Although we as modelers might be expert in expressing ourselves at a formal, type level, the same cannot be said of the average domain expert. By working with examples familiar to the subject matter experts, we can tap their implicit understanding of the UoD without forcing them to abstract and express, perhaps incorrectly, the structure we seek.

Sentence (41) expresses a relationship between an entity (a student) and a value (a name). When verbalizing facts in this way, the **value type** is stated just before the value (e.g., StudentName ‘Bloggs FB’). Unlike entity terms, value terms do not include a



**Figure 3.2** Drawing connections for the verbalized relationships.

CS114 Tutorial Preferences Form		
Please complete this form to assist in tutorial allocation. Tutorials are of 1-hour duration and are available at these times:		
Monday	Tuesday	Thursday
		10 a.m.
		11 a.m.
		12 noon
	2 p.m.	2 p.m.
3 p.m.		3 p.m.
<i>Student number:</i> _____ <i>Student surname:</i> _____ <i>Initials:</i> _____ <i>Tutorial preference 1:</i> _____ <i>Tutorial preference 2:</i> _____ <i>Tutorial preference 3:</i> _____		

**Figure 3.3** An input form for collecting tutorial preferences.

reference mode. The next section discusses the connection between value types and reference modes.

With many kinds of reports it is sometimes useful to draw a connection between the relevant fields as we verbalize the corresponding fact. For example, we might add links between the columns of Table 3.6 as shown in Figure 3.2. This informal summary of the fact types may help us to see if some connections have been missed (each field is normally involved in at least one connection).

Besides tables, *forms* are a common source of information examples. These are more often used for input than output, but may be used for both (e.g., the personnel forms considered in the previous chapter).

As another example, consider the input form shown in Figure 3.3. Suppose students studying the subject CS114 use this form to indicate up to three preferences regarding which tutorial time is most suitable for them. This form is used to help decide which students are assigned to which groups and which groups are eventually used. If this information is to be taken into account in determining tutorial allocations, it must be stored in the system.

Let's assume that the previous output report (Table 3.6) shows the tutorial allocations for CS114, made after all the student preferences are considered. The preference

input form lacks some of the information needed for the allocation report. For example, it does not show how groups are assigned to rooms and times. This helps to prevent students from entering wrong data (they enter the times they prefer directly rather than indirectly through associated group codes) and allows flexibility in offering many tutorials at the same time.

To perform step 1 here, you should first fill out the form with some examples, as shown in Figure 3.4. These facts may now be verbalized as Student (nr) 302156 has Surname ‘Jones’; Student 302156 has Initials ‘ES’; Student 302156 has first preference at Time (dh) ‘Mon 3 p.m.’; Student 302156 has second preference at Time ‘Thurs 11 a.m.’.

Taken individually, the output report and the input form reveal only partially the kinds of information needed for the system. In combination, however, they might be enough for us to arrive at the structure of the UoD. If so, the pair of examples is said to be significant. In general, a set of examples is **significant** or adequate with respect to a specific UoD only if it illustrates all the relevant sorts of information and constraints required for that application.

With complex UoDs, significant example sets are rare. With our current application, if a student can be allocated to only one group, then Table 3.6 is significant in this respect. However, if more than one group can be held at the same time, Table 3.6 is not significant in this other respect. A further row is needed to show this possibility (e.g., a row indicating that group B2 meets at Tues 2 p.m.).

When using information examples to extract facts, we need to decide *which aspects should be modeled*. This helps to determine the scope of the UoD. A useful heuristic is to ask the question *Which parts may take on different values?* Look again at the input form in Figure 3.4. The header section contains information that we may or may not wish to model. The first item we see is “CS114”. Is it possible to have other values in

CS114 Tutorial Preferences Form		
Please complete this form to assist in tutorial allocation. Tutorials are of 1-hour duration and are available at these times:		
Monday	Tuesday	Thursday
		10 a.m.
		11 a.m.
		12 noon
	2 p.m.	2 p.m.
3 p.m.		3 p.m.
Student number: <u>302156</u> Student surname: <u>Jones</u> Initials: <u>ES</u> Tutorial preference 1: <u>Mon 3 p.m.</u> Tutorial preference 2: <u>Thurs 11 a.m.</u> Tutorial preference 3: _____		

**Figure 3.4** The input form populated with sample data.

place of it, within our overall application? If the only subject of interest is CS114, then the answer is “No”. However, if we wish to cater to other subjects as well, we might require another form with a different value here (e.g., “CS183”). In this case we need to introduce Subject (or some equivalent term) as an object type within our UoD.

Different places use different names to refer to a unit of study in which a student may enroll. I use “Subject” for this concept, but you might prefer “Course” or “Unit” or some other term. If the domain experts all prefer the same term, you should use that. In large projects, different people might use different terms for the same concept. In that case, you should get them to agree upon a *standard term*, and also note any *synonyms* that they might still want to use.

Returning to the header of our preferences form, we see the word “Tutorial”. Could this change (e.g., to “Lecture”)? If we wish to capture preferences for lectures as well as tutorials, the answer is “yes” and we could model this as data. But let’s assume that this is not the case. The rest of the form header contains other information (e.g., duration of tutorials), but let’s assume this doesn’t need to be modeled.

The middle section of the form contains information about the tutorial times. If our UoD has only one subject (CS114), we could model this information as unary facts (e.g., Time ‘Mon 3 p.m.’ is available). If we need to cater to other subjects as well, then we need to treat the “CS114” at the top of the form as data, and hence verbalize the schedule as binary facts (e.g., Subject ‘CS114’ has a tutorial slot at Time ‘Mon 3 p.m.’). A completed tutorial preferences form for a different subject is shown in Figure 3.5.

Here the layout of the five weekdays into columns makes it more obvious that the times are to be treated as data. The first fact from this section reads: Subject ‘CS183’ has a tutorial slot at Time ‘Tues 2 p.m.’. If you reformat the structure of the earlier CS114 example (Figure 3.4) to agree with this structure, and place the two forms side by side, you can see what aspects are to be modeled as data by looking at what changes between the two forms (subject code, times, student details).

CS183 Tutorial Preferences Form				
Monday	Tuesday	Wednesday	Thursday	Friday
			9 a.m.	
			11 a.m.	
	2 p.m. 3 p.m.	3 p.m.	2 p.m.	
Student number: <u>211780</u> Student surname: <u>Smith</u> Initials: <u>JA</u> Tutorial preference 1: <u>Tues 2 p.m.</u> Tutorial preference 2: <u>Thurs 11 a.m.</u> Tutorial preference 3: <u>Tues 3 p.m.</u>				

**Figure 3.5** A completed tutorial preferences form for a different subject.

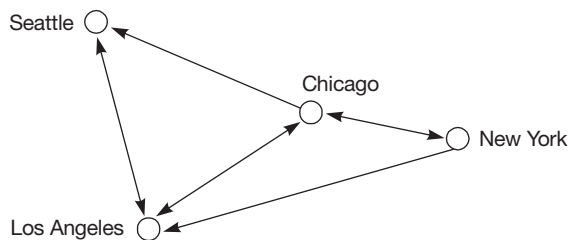
In this larger UoD, assuming that students may enroll in many subjects, the facts about student preferences now need to take the subject into account. Instead of binaries, the preference facts are now verbalized as ternaries. For example: Student 302156 has first tutorial preference for Subject ‘CS114’ at Time ‘Mon 3 p.m.’; Student 211780 has third tutorial preference for Subject ‘CS183’ at Time ‘Tues 3 p.m.’. Tutorial allocations would also need to indicate the relevant subject (e.g., Table 3.6 would need a header showing the subject). Tutorial groups would then need a composite reference scheme that included both the subject code and the group code. For example, the first fact from the CS114-headed version of Table 3.6 would now read as TuteGroup ‘A’ of Subject ‘CS114’ meets at Time ‘Mon 3 p.m.’. Composite reference schemes are discussed in detail later in Section 5.4.

We will see later that no set of examples can be significant with respect to derivation rules or subtype definitions. In such cases the use of a domain expert is essential. With the current application, we made no mention of the rules used to arrive at the tutorial allocations. If in addition to storing information about preferences and allocations, the information system has to compute the allocations in a nearly optimal way, respecting preferences and other practical constraints (e.g., size of groups), the design of the derivation rules becomes the challenging aspect of the schema. While this can be automated, an alternative is to divide the task between the human expert and the system. High-level languages facilitate such cooperative solutions.

Often, the information examples used for verbalization in step 1 apply to the way the business or application domain currently works. From these examples we can build the *as-is model* to reflect the current practice (as it is now). Some changes may also be needed to expand or improve the way the business operates. For example, we might have started with separate applications to administer tutorials for just one subject, and then realized it would be better to integrate these into a single application capable of handling all the subjects (as discussed above). By including examples of the new data requirements, we are then able to build the *to-be model*, which reflects the way we want the business to be in the future. A proper understanding of the as-is model is a great assistance in designing the to-be model. As you gain more experience as a modeler, you will be able to draw upon lessons learned from prior modeling projects to help spot ways to improve things on future projects. Modeling is not just a science. It’s an art as well, and that makes it more fun.

A comprehensive set of output reports (covering intermediate stages) may include all the information on input forms. Output reports tend to be easier to interpret, especially if the input forms have been poorly designed. Care is needed in the design of the input forms to make them clear and simple for users.

Information can appear in lots of ways. Apart from many kinds of tables and forms, information may be expressed graphically in all kinds of *diagrams*, *charts*, *maps*, and *graphs*. Harris (1996) discusses several hundred different ways of presenting information graphically. Regardless of how it’s presented, information can always be verbalized as facts. Because practice helps a lot with verbalization skills, I’ve included lots of varied examples in the book to prepare you for performing step 1 in practical situations. As a simple graphical example, Figure 3.6 might be used to display information about nonstop flight connections provided by a particular airline, with the arrowheads



**Figure 3.6** A graph showing flight connections between cities.

indicating the direction of the flights. As an exercise, perform step 1 for this graph before reading on.

How did you go? There is only one entity type: City (name). There is also only one fact type: City has a flight to City. For instance the arrow from Chicago to Seattle may be verbalized as City ‘Chicago’ has flight to City ‘Seattle’. The “to” in the predicate is important, since it conveys direction and avoids the symmetry problem with the earlier marriage example.

In this UoD, not all the connections are two-way. If this was an as-is model, and we wanted also to talk about the flight connections or to include many airlines, we should add flight numbers to the arrows on the graph. This to-be model leads to a different verbalization, which you might like to try for yourself. A later exercise returns to this example.

By now you may have some sense of the power of verbalizing examples in terms of elementary facts. No matter what kind of example you start with, if you or an assistant understands the example, then you should be able to express the information in simple facts. This does require practice at the technique, but this is fun anyway—isn’t it? If you can’t do step 1, there is little point in proceeding with the design—either you don’t understand the UoD or you can’t communicate it clearly.

Although it might sound hard to believe, if you have performed step 1 properly, you have completed most of the “hard part” of the conceptual schema design procedure. The remaining steps consist of diagramming and constraining the fact types. Apart from the problem of detecting unusual constraints and derivation rules, once you learn the techniques you can carry out those steps almost automatically. With step 1, however, you always need to draw upon your interpretation skills.

### Exercise 3.3

1. Assuming suitable entity types and reference modes are understood, which of the following sentences express exactly one elementary fact?
  - (a) Adam likes Eve.
  - (b) Bob does not like John.
  - (c) Tom visited Los Angeles and New York.
  - (d) Tom visited Los Angeles or New York.
  - (e) If Tom visited Los Angeles, then he visited New York.

- (f) Sue is funny.  
 (g) All people are funny.  
 (h) Some people in New York have toured Australia.  
 (i) Brisbane and Sydney are in Australia.  
 (j) Brisbane and Sydney are in the same country.  
 (k) Who does Adam like?
2. Indicate at least two different meanings for each of the following sentences, by including names for object types and reference modes.  
 (a) Pluto is owned by Mickey.  
 (b) Dallas is smaller than Sydney.  
 (c) Arnold can lift 300.

Perform step 1 of the CSDP for the following output reports. In verbalizing the facts, you may restrict yourself to the top row of the table unless another row reveals a different kind of fact.

3.

<i>Athlete</i>	<i>Height</i>
Jones EM	166
Pie QT	166
Smith JA	175

4.

<i>Athlete</i>	<i>Height</i>
Jones EM	400
Pie QT	450
Smith JA	550

5.

<i>Person</i>	<i>Height (cm)</i>	<i>Birth year</i>
Jones EM	166	1955
Pie QT	160	1970
Smith JA	175	1955

6.

<i>Person</i>	<i>Height (cm)</i>	<i>Birth year</i>
Jones EM	160	1970
	166	1980
	166	1990

7.

<i>Advisory panel</i>	<i>Internal member</i>	<i>External member</i>
Databases	Codd	Ienshtein
Logic programming	Kowalski	Spock
	Colmerauer	Robinson
	Kowalski	
	Spock	

8.

<i>Parents</i>	<i>Children</i>
Ann, Bill	Colin, David, Eve
David, Fiona	Gus

9.

<i>Country</i>	<i>Friends</i>	<i>Enemies</i>
Disland Hades	Oz	Oz Wundrland
Wundrland Oz	Oz Disland Wundrland	Hades Hades

10.

<i>Apple</i>	<i>Australia</i>	June, July, Aug
	<i>America</i>	Oct, Dec, Jan
	<i>Ireland</i>	Oct, Dec
<i>Mango</i>	<i>Australia</i>	Nov, Dec, Jan, Feb
<i>Pineapple</i>	<i>America</i>	June, July
	<i>Australia</i>	Oct, Nov, Dec, Jan

### 3.4 CSDP Step 2: Draw Fact Types, and Populate

Once we have translated the information examples into elementary facts and performed quality checks, we are ready for the next step in the conceptual schema design procedure. Here we *draw* a conceptual schema diagram that shows all the *fact types*. This illustrates the relevant object types, predicates, and reference schemes. Once the diagram is drawn, we check it with a *sample population*.

#### ***CSDP step 2: Draw the fact types, and apply a population check.***

Consider the sample output report of Table 3.7. Let us agree that the information in this report can be expressed by the following three elementary facts, using “regNr” to abbreviate “registration number”:

The Person named ‘Adams B’ **drives** the Car with regNr ‘235PZN’.

The Person named ‘Jones E’ **drives** the Car with regNr ‘235PZN’.

The Person named ‘Jones E’ **drives** the Car with regNr ‘108AAQ’.

Before looking at the relevant conceptual schema diagram, it may help to explain things if we first view an *instance diagram* for this example (see Figure 3.7). Instance diagrams illustrate particular instances of objects and relationships.

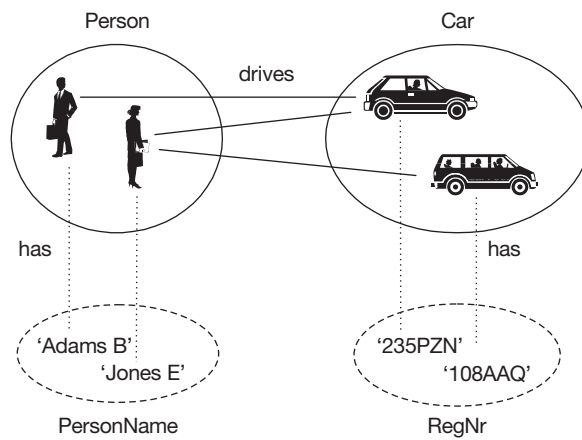
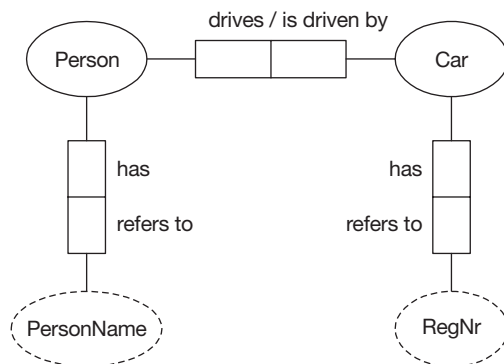
Taking advantage of the concrete nature of the entities in this example, cartoon drawings denote the actual people and cars. The values are shown as character strings. A particular fact or relationship between a person and a car is shown as a solid line. A particular reference between a value and an entity is shown as a broken line. Figure 3.8 shows a *conceptual schema diagram* for the same example.

Instance diagrams and conceptual schema diagrams depict an *entity type* as a *named, solid ellipse*, which may be a circle (the simplest form of an ellipse). A *value*

**Table 3.7** A relational table indicating who drives what cars.

*Drives:*

<i>person</i>	<i>car</i>
Adams B	235PZN
Jones E	235PZN
Jones E	108AAQ

**Figure 3.7** An instance diagram.**Figure 3.8** A conceptual schema diagram (constraints omitted).

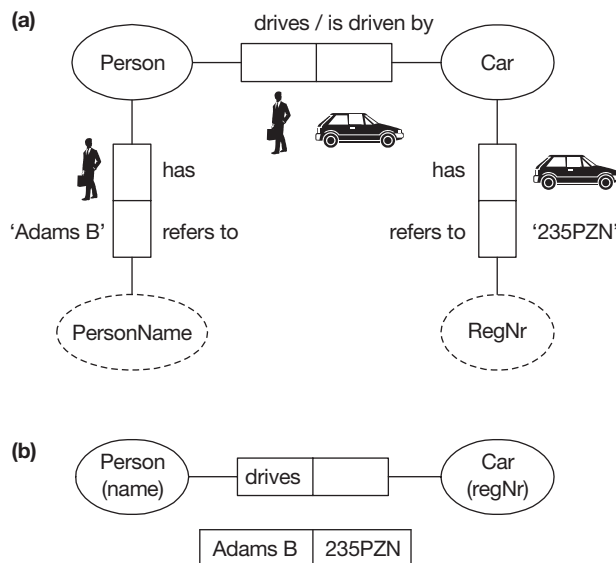
*type* is shown as a *named, broken ellipse*. The object type's name is written inside or beside the ellipse.

On an instance diagram, individual objects of a given population are explicitly portrayed (by cartoon figures or other symbols). However, on a conceptual schema diagram, individual objects are omitted (unless we reference them in associated tables). Recalling that a type is the set of permitted instances, we may imagine that objects of a particular type are represented as points inside the ellipse.

On a conceptual schema diagram, the *roles* played by objects are explicitly shown as *boxes*. Each *n-ary predicate* ( $n > 0$ ) is depicted as a *named, contiguous sequence of n role boxes* ("contiguous" means the boxes are adjacent, with no gaps in between). Predicates are ordered from one end to the other, with their name starting inside or beside their first role box (which must be an end role). For binary predicates (two roles), both forward and inverse readings may be shown. If shown combined, forward and inverse predicates are separated by a slash "/".

Each role is connected to exactly one object type by a *line*, indicating that the role is played only by objects of that type. A complete conceptual schema diagram includes the relevant constraints. We'll see how to add these later.

A relationship used to provide an identification scheme is called a *reference*. All other relationships are called *facts*. Typically, facts are relationships between entities, and references are relationships between entities and values. References provide the bridge between the world of entities and the world of values. This is clearly seen in Figure 3.9(a), where an instance has been added to populate each relationship. The



**Figure 3.9** Using reference modes for 1:1 reference: (a) model; (b) abbreviation.

relationship between the person and car (depicted with icons) is a fact. The relationship between the name ‘Adams B’ and the person is a reference, and so is the relationship between the registration number ‘235PZN’ and the car.

Although both reference predicates are displayed with the name “has”, they are different predicates. Internally a CASE tool may identify the predicates by surrogates (e.g., “P2”, “P3”) or expanded names (e.g., “PersonHasPersonName”, “Personhas-RegNr”). Although the predicate name “has” may also be used with fact types, it is best avoided if there is a more descriptive, natural alternative. For example, “Person drives Car”, if accurate, is better than “Person has Car”, which could mean many things (e.g., Person owns Car).

For this example, each person has exactly one name, and each person name refers to at most one person. Moreover, each car has exactly one registration number, and each registration number refers to at most one car. This situation is seen clearly in the earlier instance diagram (Figure 3.7). Each of the two reference types is said to provide a *simple 1:1 reference scheme*. We read “1:1” as “one to one”. Later we’ll see how to specify this on a conceptual schema diagram using uniqueness and mandatory role constraints.

When a simple 1:1 naming convention exists, we may indicate the *reference mode* simply by placing its name in *parentheses* next to the name of the entity type, and use the values themselves to depict entity instances in associated fact tables. Assuming appropriate constraints are added, the populated schema of Figure 3.9(a) may be displayed more concisely by Figure 3.9(b). Unless we want to illustrate the reference schemes explicitly, this concise form is preferred because it’s closer to the way we verbalize facts and it simplifies the diagram. The fact table is omitted if we wish to display just the schema.

Reference modes indicate the mode or manner in which values refer to entities. Using reference modes, we rarely need to display value types explicitly on a schema diagram. However, to understand the abbreviation scheme, we need to know how to translate between reference modes and value types. Different versions of ORM have different approaches to this. One method for doing this is now outlined. Let the notation “ $E(r) \rightarrow V$ ” mean “Entity type  $E$  with reference mode  $r$  generates the Value type  $V$ ”. Reference modes may be partitioned into three classes: popular, unit based, and general. The *popular reference modes* are *name*, *code*, *title*, *nr*, *#*, and *id*. To obtain the value type name, a popular reference mode has its first letter shifted to upper case and is then appended to the name of entity type. For example: Person (name)  $\rightarrow$  PersonName; Item (code)  $\rightarrow$  ItemCode; Song (title)  $\rightarrow$  SongTitle; Rating (nr)  $\rightarrow$  RatingNr; Room (#)  $\rightarrow$  Room#; Member (id)  $\rightarrow$  MemberId.

The *unit-based reference modes* include a built-in list of physical units (e.g., cm, kg, mile, Celsius), monetary units (e.g., EUR, USD), and abnormal units (e.g., AD), as well as user-defined units. Value type names are generated from unit-based reference modes by appending the word “Value”. For example: kg  $\rightarrow$  kgValue; USD  $\rightarrow$  USDValue.

All other reference modes are called *general reference modes*. These generate value type names simply by shifting their first letter to upper case. For example: surname  $\rightarrow$  Surname; empNr  $\rightarrow$  EmpNr.

Commercial ORM tools may support different, more flexible schemes for mapping between the names of reference modes and corresponding value types, allowing you to choose different options, such as whether to insert underscore separators when appending.

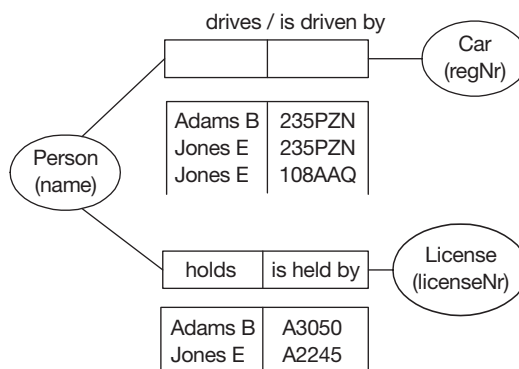
As a check that we have drawn the diagram correctly, we should populate each fact type on the diagram with some of the original fact instances. We do this by adding a **fact table** for each fact type and entering the values in the relevant columns of this table. In ORM, a fact table is simply a table for displaying instances of an elementary fact type. The term “fact table” is used in a different sense in data warehousing (see Chapter 13). A diagram that includes both a schema and a sample database is called a *knowledge base diagram*.

Consider the output report of Table 3.8. Here “LicenseNr” refers to the person’s driver’s license. Performing step 1 reveals that there are two binary fact types involved (check this for yourself).

We can now draw the conceptual schema diagram. As a check, we populate it with the original data (see Figure 3.10). If desired, the inverse predicate may be included, as shown in this figure. At this stage the diagram is incomplete because constraints are not shown. At least one fact from each fact table should be verbalized to ensure the diagram makes sense. Populating the schema diagram is useful not only for detecting schema diagrams that are nonsensical, but also for clarifying constraints (as we see later).

**Table 3.8** Driver details.

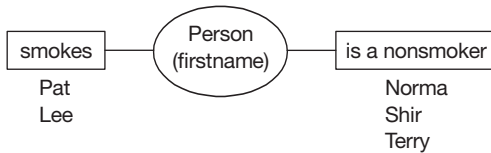
<i>Person</i>	<i>LicenseNr</i>	<i>Cars driven</i>
Adams B	A3050	235PZN
Jones E	A2245	235PZN, 108AAQ



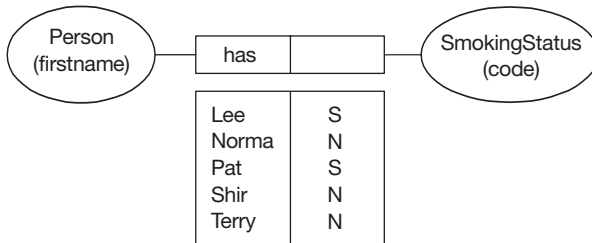
**Figure 3.10** A knowledge base diagram for Table 3.8 (constraints omitted).

**Table 3.9** Smokers and nonsmokers.

<i>Smokers</i>	<i>Nonsmokers</i>
Pat Lee	Norma Shir Terry



**Figure 3.11** Knowledge base diagram for Table 3.9 (unary version).

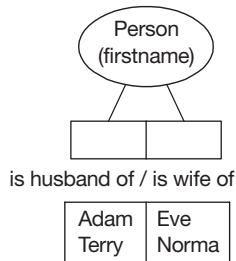


**Figure 3.12** Knowledge base diagram for Table 3.9 (binary version).

Nowadays most nonsmokers prefer a smoke-free environment in which to work, travel, eat, and so on. So for some applications, a report like Table 3.9 is relevant. Please perform step 1 on this table before reading on.

One way to express the facts on row 1 is Person (firstname) ‘Pat’ smokes; Person (firstname) ‘Norma’ is a nonsmoker. Each of these facts is an instance of a different *unary fact type*. With a unary fact type, there is only one role. The knowledge base diagram is shown in Figure 3.11.

Here the two roles belong to different fact types. This is shown by separating the role boxes. If desired, the two unaries may be transformed into a single binary by introducing SmokingStatus as another entity type, with codes “S” for smoker and “N” for nonsmoker. So the first row of Table 3.9 could be rephrased as Person (firstname) ‘Pat’ has SmokingStatus (code) ‘S’; Person (firstname) ‘Norma’ has SmokingStatus (code) ‘N’. This approach is shown in Figure 3.12. Schema transformations are discussed in depth in Chapter 12.



**Figure 3.13** A ring fact type with sample population.

Each of the binary examples discussed had two different entity types. Fact types involving different entity types are said to be *heterogeneous fact types*. Most fact types are of this kind. However, the fact type in Figure 3.13 has one entity type—Person. If each role in a fact type is played by the same object type, we have a *homogeneous fact type*. The binary case of this is called a *ring fact type* since the path from the object type through the predicate loops back to the same object type, forming a ring.

Here both the forward predicate (is husband of) and inverse predicate (is wife of) are shown. In some versions of ORM, role names are used instead of predicates (e.g., husband, husband of, being husband of). Although this can be useful for some styles of queries, it is generally better to use full predicates, since this leads to better verbalization of both facts and constraints.

To make diagrams more compact, you may abbreviate names for predicates, object types, and reference modes if their expanded versions are obvious to users of the system. But this relaxed policy should be used with care. ORM tools often allow you to attach descriptions of model elements as notes, which can compensate for such name shortening.

Apart from communication with humans, conceptual schemas provide a formal specification of the structure of the UoD, so that the model may be processed by a computer system. Hence the schema diagrams we draw must conform to the formation rules for legal schemas. They are not just cartoons.

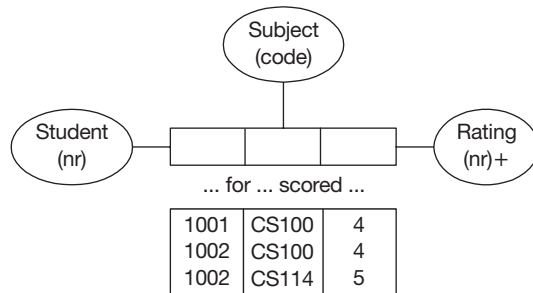
Now consider the output report of Table 3.10. This is similar to an example discussed previously, but to be more realistic, students are now identified by their student number. Here we have a *ternary fact type*. The object types and reference schemes are Student (nr), Subject (code), Rating (nr)+. Given this, we may express the fact on the first row as Student ‘1001’ for Subject ‘CS100’ scored Rating 4.

On a conceptual schema diagram, a ternary fact type appears as a sequence of three role boxes, each of which is attached to an object type, as shown in Figure 3.14. When names for ternary and longer predicates are written on a diagram, the placeholders are included, each being depicted by an *ellipsis* “...”. Figure 3.14 includes a sample population. No matter how high the arity (number of roles) of the fact type, we can easily populate it for checking purposes. Each column in the fact table is associated with one role in the predicate.

**Table 3.10** A relational table storing student results.

*Result:*

<i>studentNr</i>	<i>subject</i>	<i>rating</i>
1001	CS100	4
1002	CS100	4
1002	CS114	5


**Figure 3.14** A populated ternary fact type for Table 3.10.

An earlier example transformed unaries into a binary. Another kind of schema transformation is **nesting**. *This treats a relationship between objects as an object itself.* Consider once more the top row of Table 3.10. Instead of expressing this as a single sentence, we might convey the information in the following two sentences:

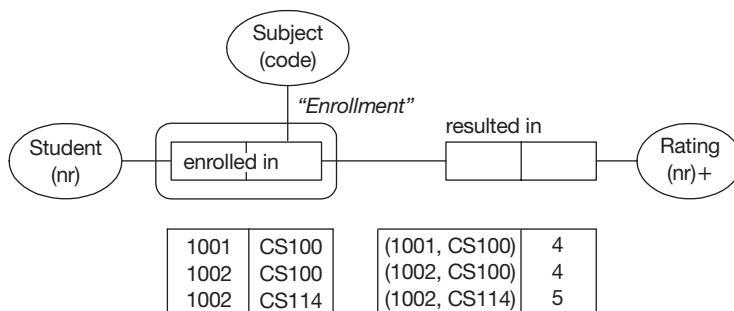
Student '1001' enrolled in Subject 'CS100'.

*This Enrollment* resulted in Rating 4.

Here "This Enrollment" refers back to the enrollment relationship between the specific person and the specific subject mentioned in the first sentence. Any such enrollment may be treated as an object in its own right. The act of making an object out of a relationship is called *objectification* and corresponds to the linguistic act of nominalization (making a noun out of a verb phrase). An object formed by objectification is called an *objectified relationship* or a *nested object* (since other objects are nested within it). The type of object so formed is called an *objectified association*, an *objectified relationship type*, or a *nested object type*. Recall that the term "association" means the same as "relationship type".

An objectified association is depicted by a *soft rectangle* around the predicate being objectified (see Figure 3.15). Soft rectangles have rounded corners and are sometimes called frames or fillets. A name for the objectified association is added beside it, in *double quotes*. An objectified association usually has two roles, but may have more: . An alternative notation uses an ellipse instead of a frame.

Entries in fact columns for nested objects may be shown as bracketed pairs (triples, etc.) of values. For example, the enrollment of student 1001 in CS100 appears as



**Figure 3.15** Knowledge base for Table 3.10 (nested version).


“(1001, CS100)” in the fact table for the resulted in predicate. Note that *nesting is not the same as splitting*. Figure 3.15 does not show two independent binaries. The resulted in predicate cannot be shown without including the enrolled in predicate. So the ternary in Figure 3.14 is still elementary. Figure 3.14 is said to be the *flattened*, or *unnested*, version.

Chapter 12 deals with the notion of schema equivalence in detail. The nested and flattened versions are not equivalent unless the role played by the objectified association is mandatory. With our current example, this means that a rating must be known for each enrollment. In this case the flattened version is preferred, since it is simpler to diagram and populate. As discussed later, nesting is often preferred if the nested object type has an optional role or more than one role to play. For example, suppose we widen the UoD to include information about when the enrollments occurred. With the flattened solution, we need to add another ternary: Student enrolled in Subject on Date. With the nested solution, we simply add the binary: Enrollment occurred on Date. As we’ll see later, the nested solution also simplifies constraint specification in this case, and hence would now be preferred.

Now consider the travel record example depicted in Figure 3.16. Using the telephone heuristic, the modeler verbalizes the first row of data as an instance of the ternary fact type Politician visited Country in Year.

The sample data indicates that many politicians may visit many countries in many years. Because of this symmetry, there are many options for nesting. We could objectify Politician visited Country as Visit, and then add Visit occurred in Year. We might instead objectify Politician traveled in Year as Travel and then add Travel visited Country, or objectify Country was visited in Year as Visit, and add Visit was by Politician. With no strong reason for one nesting choice over the other, it is simpler to leave it as a ternary.

Although we display only one reading for ternary and longer fact types on the diagram, there are many possible readings depending on the order in which we traverse the roles. In principle, an  $n$ -ary predicate has  $n!$  ( $n$  factorial, i.e.,  $n \times (n - 1) \times \dots \times 1$ ) readings. So a ternary has 6 possible readings, a quaternary has 24 possible readings,



<i>Politician</i>	<i>Country</i>	<i>Year</i>
Clinton	Australia	1994
Clinton	Italy	1994
Clinton	Australia	1995
Keating	Italy	1994

The Politician with surname 'Clinton' **visited** the Country named 'Australia' **in** the Year 1994 AD.

**Figure 3.16** One way of verbalizing the first row.

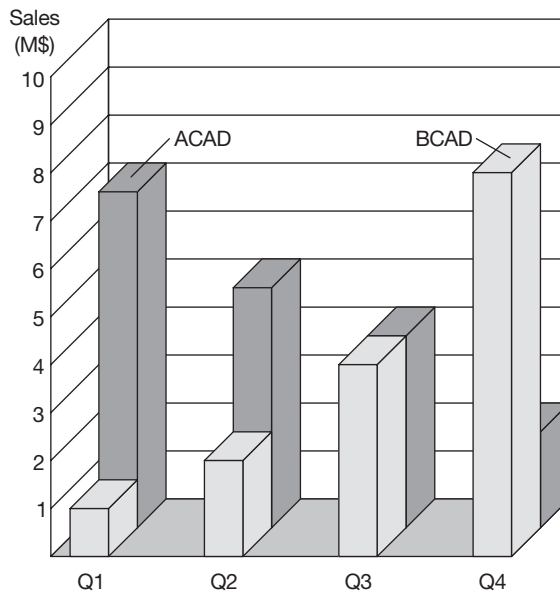
and so on. For example, the ternary fact type above could be specified by any of the following six readings: Politician visited Country in Year; Politician in Year visited Country; Country was visited by Politician in Year; Country in Year was visited by Politician; Year included visit by Politician to Country; Year included visit to Country by Politician.

In practice, one reading is enough for modeling purposes. However, if we wish to query the schema directly, it is handy to be able to navigate from any given role. To cater to this we would need to supply, for each role, a reading that starts at that role (the order of the later roles doesn't matter). So we never need any more than  $n$  readings for any  $n$ -ary fact type. This is a lot fewer than  $n$  factorial. ORM tools usually accept  $n$  readings for any  $n$ -ary, but display only one on the diagram.

Now suppose we need to design a database for storing sales data that can be displayed graphically as shown in Figure 3.17. This three-dimensional bar chart shows the sales figures for two computer-aided drafting products code-named "ACAD" and "BCAD". As an exercise in steps 1 and 2, try to verbalize the sales information and then schematize it (on a conceptual schema diagram) before reading on.

Consider the first bar on the left of the chart. A person familiar with the application might verbalize the fact as "BCAD in the first quarter had sales of one million dollars". This completes step 1a. As modelers, we complete step 1b by refining this into one or more well-formed elementary facts. In this case, we may verbalize it as a single ternary: "The Product with code 'BCAD' in the Quarter numbered 1 had sales of MoneyAmount 1000000 USD".

I chose to identify quarters using numbers (e.g., 1) but you can use codes (e.g., 'Q1') if you like. I used the object type "MoneyAmount" instead of "Sales" because I wanted to make the underlying domain explicit. This makes it clear that we can compare sales figures with other monetary values (e.g., costs and profits). I often abbreviate this to "MoneyAmt". Assuming the chart applies to the USA, I chose USD (United States dollar) for the monetary unit. This distinguishes it from other dollars (e.g., AUD



**Figure 3.17** Can you verbalize this bar chart?

for Australian dollar). This is good practice, but if there is no danger of confusion, you could simply show the unit as “\$”.

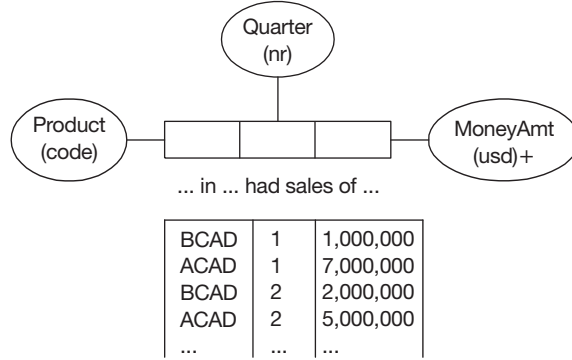
Each remaining bar may be verbalized similarly. This completes step 1. In preparation for step 2, we could set the fact type out with reference modes in parentheses, as follows:

Product (code) in Quarter (nr) had sales of MoneyAmt (usd)+.

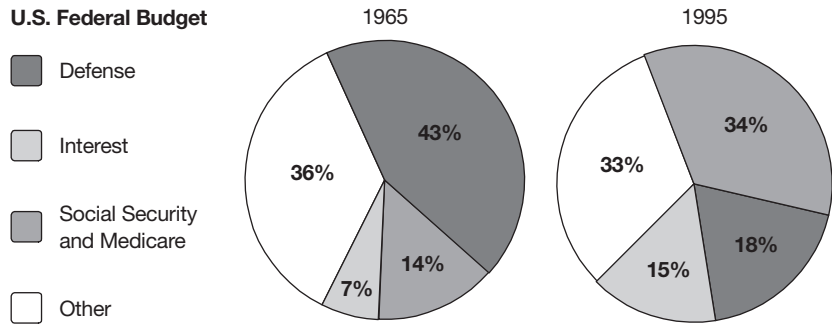
As a general rule, I use lower case for reference modes whenever practical; hence the “usd” instead of “USD”. This is largely a matter of taste. A “+” is appended to the monetary unit to indicate that we can perform arithmetic on it. This is a minor aspect and can be ignored if at some stage you associate MoneyAmt with a system data type such as Numeric or Money that implies this—Microsoft’s ORM tool actually requires you to specify an underlying numeric data type before you can use the numeric “+” marker. It is now a simple task to draw the conceptual schema diagram. As a check, we populate it with some sample fact instances (see Figure 3.18).

Another common way for presenting numeric data is the pie chart. A legend is often provided beside the chart to indicate the items denoted by each slice. Each slice of the pie indicates the portion of the whole taken up by that particular item. An example is given in Figure 3.19. Try to schematize this yourself before reading on.

Applying step 1a to the defense slice of the first pie, we could verbalize the fact represented as follows: “In 1965 defense consumed 43% of the budget”. To complete step1, we may refine this to “In Year 1965 AD the BudgetItem named ‘Defense’



**Figure 3.18** A conceptual schema for the sales data, with a sample population.



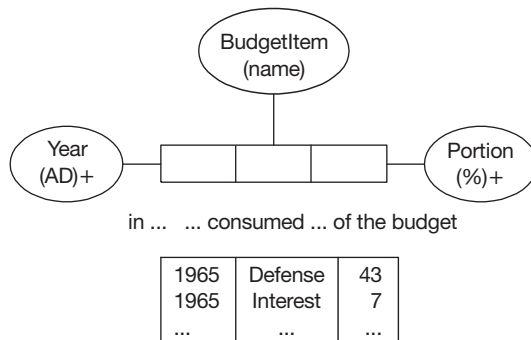
**Figure 3.19** Can you schematize this pie chart?

consumed Portion 43% of the budget”. All the other slices denote the same kind of fact. So we may generalize from the fact instances to the following ternary fact type:

in Year (AD) BudgetItem (name) consumed Portion (%) of the budget.

Here, the predicate “in ... .. consumed ... of the budget” has leading text and two object holes adjacent to one another. Though fairly rare, this is a legal mixfix predicate. This kind of flexibility makes verbalization easier than it would have been otherwise. To complete step 2, the resulting schema and sample data are shown in Figure 3.20.

To conclude this section, let’s review some terminology. Three terms for objects have now been introduced. Entities are the objects in the UoD that we reference by means of descriptions. Values (character strings or numbers) are depicted as entries in database tables and are used to refer to entities. Finally, relationships between objects may be treated as objects themselves: these are objectified relationships (or nested objects).



**Figure 3.20** A conceptual schema for the budget data, with a sample population.

**Table 3.11** Classification of predicates according to number of roles.

<i>Nr roles</i>	<i>Main descriptor</i>	<i>Alternate descriptor</i>
1	unary	monadic
2	binary	dyadic
3	ternary	triadic
4	quaternary	tetradic
5	quinary	...
6	senary	
7	septenary	
8	octanary	
9	nonary	

There are two commonly used notations for describing the arity or “length” of a predicate. The preferred notation, shown as the main descriptor, is set out for the first nine cases in Table 3.11. The alternate descriptor tends to be restricted to the first four cases, as shown. In practice it is extremely rare for any elementary predicate to exceed five roles.

Although we should populate conceptual schema diagrams for checking purposes, fact populations do not form part of the conceptual schema diagram itself. In the following exercise, population checks are not requested. However, I strongly suggest that you populate each fact type with at least one row as a check on your work.

#### **Exercise 3.4**

1. The names and gender of various people are indicated below:

*Male:* Fred, Tom

*Female:* Ann, Mary, Sue

- (a) Express the information about Fred and Ann in unary facts.
- (b) Draw a conceptual schema diagram based on this choice.
- (c) Express the same information in terms of binary elementary facts.
- (d) Draw a conceptual schema diagram based on this choice.

**Note:** For the rest of this exercise, avoid using unary facts.

- 2. Draw a conceptual schema diagram for the fact types in the following questions of Exercise 3.3:
  - (a) Question 3
  - (b) Question 4
  - (c) Question 5
  - (d) Question 6
  - (e) Question 7
  - (f) Question 8
  - (g) Question 9
  - (h) Question 10

Perform steps 1 and 2 of the CSDP for the following output reports.

3.

<i>Retailer</i>	<i>Item</i>	<i>Quantity sold</i>
CompuWare	SQL+	330
	Zappo Pascal	330
	WordLight	200
SoftwareLand	SQL+	330
	Zappo Pascal	251

4.

<i>Item</i>	<i>Retailer</i>	<i>Quantity sold</i>
SQL+	CompuWare	330
	SoftwareLand	330
Zappo Pascal	CompuWare	330
	SoftwareLand	251
WordLight	CompuWare	200

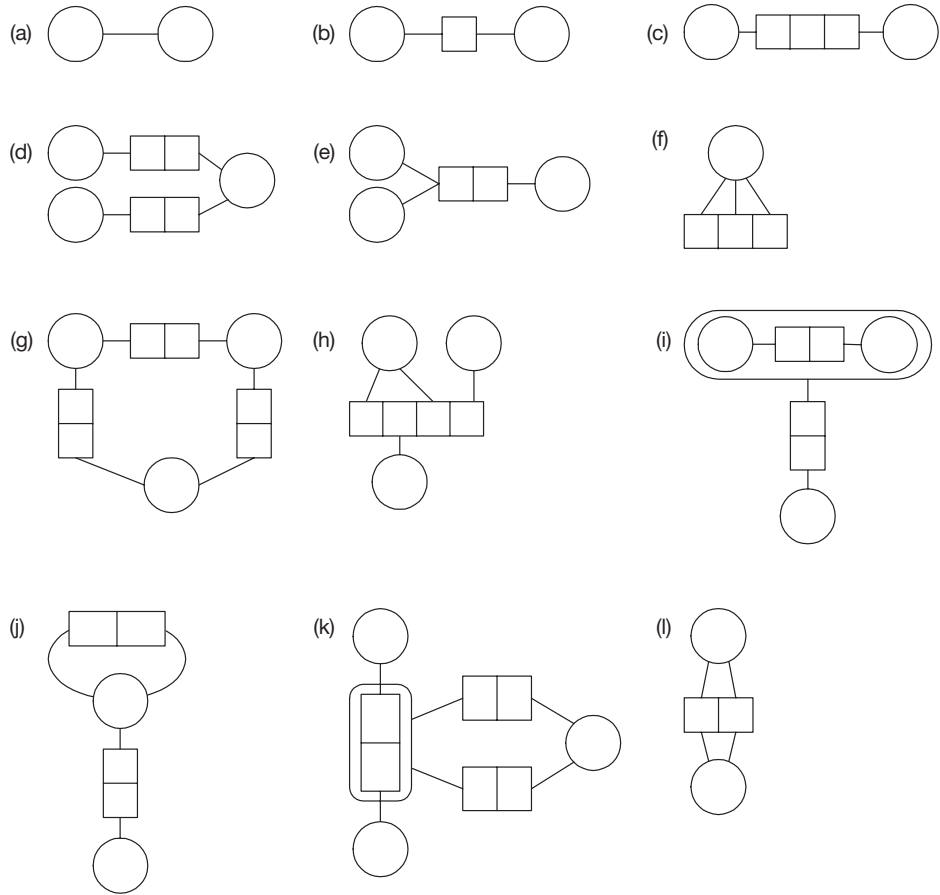
5.

<i>Tute group</i>	<i>Day</i>	<i>Hour</i>	<i>Room</i>
A	Mon	3 p.m.	69-718
B	Tue	2 p.m.	42-B18
C1	Thu	10 a.m.	69-718
C2	Thu	10 a.m.	67-103

6. *Hint:* Make use of nesting.


<i>Subject</i>	<i>CreditPts</i>	<i>Semester</i>	<i>Enrollment</i>	<i>Lecturer</i>
CS100	8	1	500	DBJ
CS102	8	2	500	EJS
CS114	8	1	300	TAH
CS115	8	2	270	TAH
CS383	16	1	50	RMC
CS383	16	2	45	PNC

7. Assuming appropriate names are supplied for entity types, reference modes, and predicates, and that appropriate constraints are added, which of the following conceptual schema diagrams are legal? Where illegal, briefly explain the error.



8. The following interactive voting form is used to input votes by cruise club members on various motions (proposals that have been officially moved by a club member). This example shows a screen shot of one completed form after a member has selected his/her voting choices. Although passwords are not displayed on-screen, they are captured by the information system. Perform CSDP steps 1 and 2 to schematize this UoD.

**Cruise Club Voting Form**

 **Member Nr:**  **Password:**

**Motions**

*Motion 52:* Ban smoking in restaurant  Approve  Reject

*Motion 53:* Change ship name to "Titanic"  Approve  Reject

### 3.5 CSDP Step 3: Trim Schema; Note Basic Derivations

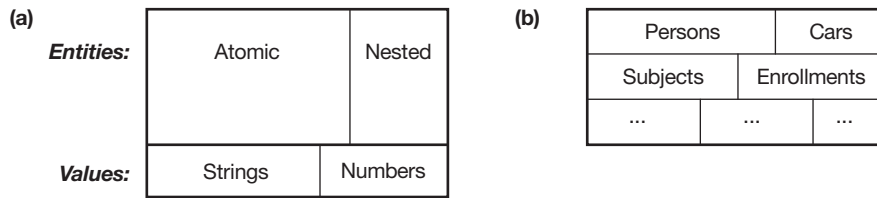
Having drawn the fact types and performed a population check, we now move on to step 3 of our design procedure. Here we check to see if there are some entity types that should be combined. We also check to see if some fact types can be derived from the others by arithmetic computation.

***CSDP step 3: Check for entity types that should be combined; note any arithmetic derivations.***

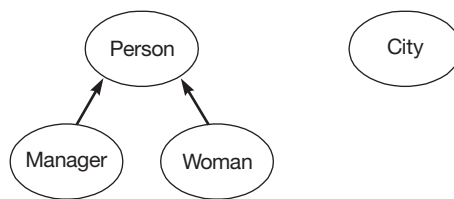
To understand the first part of this step, we need to know how the objects in the UoD are classified into types. Figure 3.21(a) shows the basic division of objects into entities (nonlexical objects) and values (lexical objects). *Entities* are identified by definite descriptions and may typically change their state, whereas *values* are simply constants (character strings or numbers). Values might include other objects directly representable on a medium (e.g., sounds), but such possibilities are ignored in this text.

*Atomic entities* are treated as individuals in their own right (e.g., persons, cars, engines). For modeling purposes, an atomic entity is treated as having no internal structure—any portrayal of structure must be depicted externally in terms of roles played by the entity. For example: Car (regNr) contains Engine (engineNr).

*Nested entities* are those relationships that we wish to think of as objects (i.e., objectified relationships; e.g., enrollments). Unlike atomic entities, nested entities are portrayed as having an internal structure (composed of the roles in the relationship). Relationships are not objects unless we think of them that way and want to talk about them.



**Figure 3.21** Partitioning into types of (a) objects and (b) primitive entities.



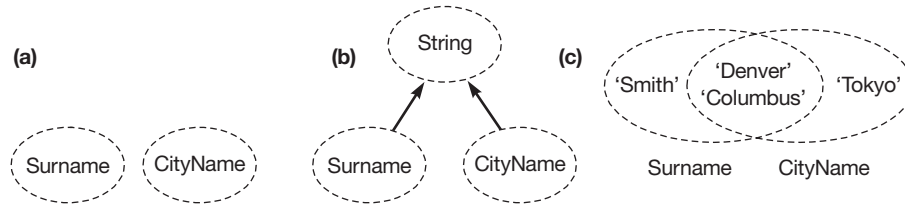
**Figure 3.22** Person and City are primitive entity types and hence mutually exclusive.

These subdivisions are **mutually exclusive** (i.e., they have no instance in common). For example, no string can be an entity. The division of a whole into exclusive parts is called a **partition**. You may think of it as cutting a pie up into slices. The slices are *exclusive* (they don't overlap) and *exhaustive* (together they make up the whole pie).

Figure 3.21(b) gives an example of how the entities might be further divided into entity types for a particular UoD. Only a few entity types are listed. Which kinds of entities exist depends on the UoD. Basically, entities are grouped into the same type if we want to record similar information about them.

For any UoD, there is always a top-level partitioning of its entities into exclusive types: these are called **primitive entity types**. We may introduce *subtypes* of these primitive types, if they have some specific roles to play. In ORM, subtypes are shown connected by an arrow to their supertype. Though shown separately, it is possible that subtypes of a given entity type may overlap. However, *primitive entity types never overlap*. For example, no person can be a city. On a conceptual schema diagram, the visual separation of primitive entity types indicates that these types are mutually exclusive. The same is not true of subtypes. In Figure 3.22, for example, Person and City are mutually exclusive but Manager and Woman need not be. Subtypes are discussed in detail in Chapter 6.

Value types often overlap, but are still shown separately on a schema diagram (e.g., Figure 3.23(a)) since they are implicitly assumed to be a subtype of String or Number (e.g., Figure 3.23(b)). In this example, Surname and CityName overlap because they may have common instances (Figure 3.23(c)). Although the explicit depiction of value subtyping or value type overlap may clarify the situation, for compactness we leave this



**Figure 3.23** String types (a) are implicit subtypes (b) and hence may overlap (c).

**Table 3.12** Some motion pictures.

<i>Movie</i>	<i>Stars</i>	<i>Director</i>
Awakenings	Robert De Niro, Robin Williams	Penny Marshall
Backdraft	Kurt Russell, Robert De Niro, William Baldwin	Ron Howard
Dances with Wolves	Kevin Costner, Mary McDonnell	Kevin Costner
...	...	...

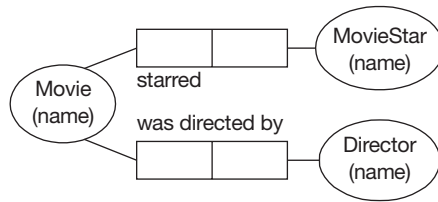
implicit (as in Figure 3.23(a)). Chapter 6 discusses how to constrain value types to a specific subtype of String or Number. For example, Surname might be restricted to strings of at most 20 characters and RatingNr to integers in the range 1..7.

Step 3 of the design procedure begins with a check to see if some entity types should be combined. At this stage we are concerned only with primitive entity types, not entity subtypes. So if you spot some entity types that do overlap, you should combine them into a single entity type. For example, consider Table 3.12, which concerns the movie application discussed in Chapter 1. Suppose that as a result of applying steps 1 and 2 we arrived at the diagram shown in Figure 3.24. Do you see what's wrong with this diagram?

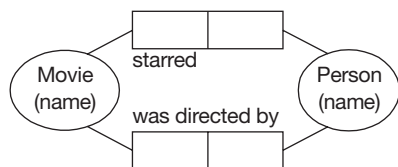
Figure 3.24 displays MovieStar and Director as separate, primitive entity types. This implies that these types are mutually exclusive (i.e., no movie star can be a director). But is this the case? Our sample population lists the value “Kevin Costner” in both the Stars column and the Director column. Does this refer to the same person?

If in doubt, you can ask a domain expert. In actual fact, it is the same person. So we must combine the MovieStar and Director entity types into a single entity type as shown in Figure 3.25. This shows it is possible to be both a star and a director. Of course, it does not imply that every movie star is a director. Chapter 6 discusses how to add subtypes later if necessary. For example, if some other facts are to be recorded only for directors, we form a Director subtype of Person for those additional facts.

One reason for suspecting that two entity types should be combined is if they both have the *same unit-based reference mode*. Here the entity is typically envisaged as a



**Figure 3.24** A faulty conceptual schema.



**Figure 3.25** The result of applying step 3 to Figure 3.24.

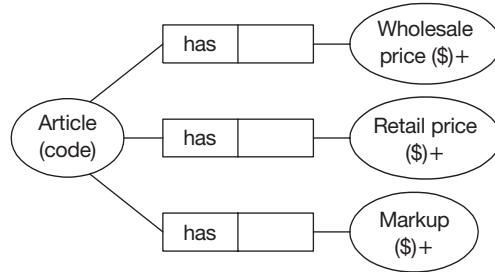
**Table 3.13** Monetary details about articles on sale.

Article	Wholesale price (\$)	Retail price (\$)	Markup (\$)
A1	50	75	25
A2	80	130	50
A3	50	70	20
A4	100	130	30

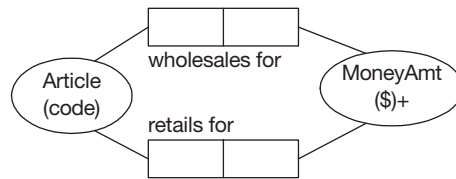
quantity of so many units (e.g., kilograms or years). Let's look at a few examples. Consider the output report of Table 3.13.

At first glance, we might consider representing the design of this UoD by the diagram shown in Figure 3.26. Note however that the entity types Wholesale price, Retail price, and Markup all have the same unit-based reference mode (\$). Here we assume that it is well understood which kind of dollar (AUD, USD, etc.) is denoted by "\$".

In the table, \$50 appears as both a wholesale price and a markup. In both cases the \$50 denotes the same amount of money. This makes it even more obvious that these entity types overlap and hence should be combined. If the table population is significant, the set of retail prices does not overlap the set of markups. Nevertheless, it is *meaningful to compare* retail prices and markups since they have the same unit (dollars). For instance, article A1 has a retail price that is three times its markup. These considerations



**Figure 3.26** Another faulty conceptual schema.



\* { markup = retail price – wholesale price }

Article has markup of MoneyAmt **iff** Article retails for MoneyAmt<sub>1</sub> **and**  
 Article wholesales for MoneyAmt<sub>2</sub> **and**  
 MoneyAmt = MoneyAmt<sub>1</sub> – MoneyAmt<sub>2</sub>

**Figure 3.27** The result of applying step 3 to Figure 3.26.

lead us to collapse the three entity types into the concept of money amount, abbreviated here as “MoneyAmt”, as shown in Figure 3.27.

There is one other point to be noted with this example. The output report satisfies the following mathematical relationship between the values in the last three columns: markup = retail price – wholesale price. Assuming this is significant, the markup value may be **derived** from the wholesale and retail values by means of this rule. To minimize the chance of human error, we have the system derive this value rather than have humans compute and store it. In this book, *derivation rules are written as text below the schema diagram*. With an ORM tool you might enter the rule in a text box or a properties sheet. If the derivation rule is captured, there is no need to include the derived fact type on the diagram itself.

An informal version of the rule may be written as a comment in braces. What about a formal version of the rule? One way of stating the derivation rule formally is shown in Figure 3.27. Here object type variables are subscripted, as in ConQuer, an ORM

query language. An alternative is to introduce the variable after the type name (e.g., Article *a*).

Notice the use of “**iff**”: this abbreviates “if and only if”, indicating the rule is a *biconditional* (it works in both directions). From a strict conceptual viewpoint, the derivation rule merely declares a constraint between the three fact types for markup, retail price, and wholesale price. Any one of the three could be derived from knowledge of the other two. Sometimes we might wish to enter the retail and wholesale prices and have the system derive the markup. At other times, we might wish to enter the wholesale price and markup and have the system derive the retail price. It is possible to build a system that supports both these choices, and this can be quite useful in practice.

In most derivation cases, however, we usually decide beforehand that one specific fact type will always be the derived one. In this case, the derivation rule may be specified as a *definition*, where the derived fact type is defined in terms of the others. For example:

```
define Article has markup of MoneyAmt as
  Article retails for MoneyAmt1 and
  Article wholesales for MoneyAmt2 and
  MoneyAmt = MoneyAmt1 – MoneyAmt2
```

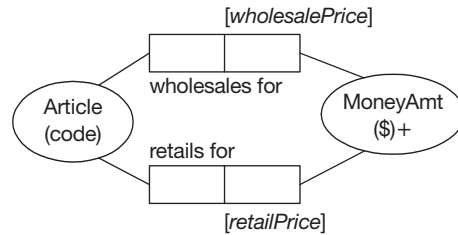
In such a definition, the derived fact type is said to be the *definiendum* (what is required to be defined). A fact type that is primitive (i.e., not defined in terms of others) is said to be a *base* fact type. *Derived* fact types are defined in terms of other fact types (base or derived).

All the ways discussed so far for setting out the derivation rule use the *relational style*. Here the fact types are set out fully as relationship types. Although this style is ideal for logical derivation rules (e.g., the uncle example from Chapter 2), it is awkward for arithmetical derivation rules, as in our current example. For such cases, the *attribute style* of rule is more compact and readable. If all the attributes are single valued, this is also called the *functional style*. In addition to the required predicate name, ORM allows you to optionally add a *rolename* for any individual role. Rolenames are used mainly for improving the readability of the attribute names automatically generated when mapping to attribute-based models such as ER, UML, or relational schemas. But they can also be used as attribute names in derivation rules.

In Figure 3.28 the rolenames “wholesalePrice” and “retailPrice” appear in square brackets next to their role. For binary associations, rolenames may be treated as attributes of the object type at the opposite end of the association. Using the dot naming convention, the qualified rolenames are Article.wholesalePrice and Article.retailPrice.

The derivation rule may now be formally but concisely expressed by the equation shown in Figure 3.28. Typing rules imply that the markup role is played by MoneyAmt, and the predicate name “has markup of” can be generated by default for the underlying fact type.

Most ORM tools do not yet support the square bracket notation for rolenames used in Figure 3.28. Displaying both predicate and rolenames on a diagram can lead to a cluttered appearance, so ideally you should be able to toggle their display on or off. The dot notation for attributes is used in UML and many other approaches, but if you are using the attribute style for derivation rules, a bracket notation or “of” notation may



\*  $\text{Article.markup} = \text{Article.retailPrice} - \text{Article.wholesalePrice}$

**Figure 3.28** Use of rolenames in an attribute style of derivation rules.

**Table 3.14** Window sizes.

Window	Height (cm)	Width (cm)	Area (cm <sup>2</sup> )
1	4	5	20
2	6	20	120
3	10	15	150
4	5	5	25

be preferable for validating rules with the domain experts. For example, the rule could also be formally declared in either of the following ways:

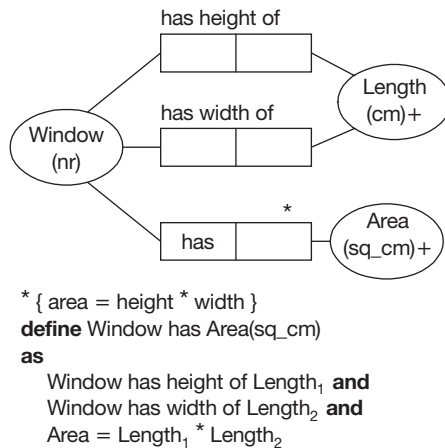
$\text{markup}(\text{Article}) = \text{retailPrice}(\text{Article}) - \text{wholesalePrice}(\text{Article})$   
 $\text{markup of Article} = \text{retailPrice of Article} - \text{wholesalePrice of Article}$

Let's consider now a somewhat similar example. The output report of Table 3.14 shows details about windows on a computer screen. Try to schematize this before reading on.

As you probably guessed, we can get by with fewer than four entity types here. Seeing that the values shown in the last three columns are all numbers, you may have been tempted to combine Height, Width, and Area into one entity type. You might argue that Height and Width overlap since the value 5 is common, and that Width and Area overlap since each includes the value 20. But notice that Area is measured in square centimeters, which is quite a different unit from centimeters.

Heights and widths may be meaningfully compared since both are lengths: a length of 5 cm may be an instance of a height or a width. But a length of 20 cm is not the same thing as an area of 20 cm<sup>2</sup>. If our final column was headed "Perimeter (cm)", we could collapse three headings into one entity type as for the previous example. But since Area is fundamentally a different type of quantity, we must keep it separate, as shown in Figure 3.29.

Derived fact types are usually specified only as rules, to avoid cluttering up the diagram. However, it is sometimes instructive to show a derived fact type on the diagram. In this case it must be marked as being derived, to distinguish it from the base fact



**Figure 3.29** Schema for Table 3.14.

types. To do this in ORM, *place an asterisk beside any derived fact type that is included on the diagram*, as shown in Figure 3.29. Whether or not a fact type is displayed on a diagram, a rule for deriving it should be declared.

The informal rule shown for area computation uses an asterisk in a different sense (for multiplication). For the formal rule, I've decided to always make Area the derived quantity, so I've used a definitional form for the rule. In principle any one of the three quantities could be derived from the other two. In cases like this, where there really is a choice as to which is the definiendum, the decision is often based more on performance than on conceptual issues. In many cases, however, there simply is no choice. For example, facts about sums and averages are derivable from facts about individual cases, but except for trivial cases we cannot derive the individual facts from such summaries.

Using rolenames, the rule could be specified more compactly as `Window.area = Window.height * Window.width`. The main advantage of predicate-based notation is that it is more stable than an attribute-based notation, since it is not impacted by schema changes such as attributes being remodeled as associations. Though unlikely in this particular example, when such changes are possible the choice between the attribute style or relational style for derivation rules involves a trade-off between convenience and stability.

It is an implementation issue whether a derived fact type is *derived-on-query* (*lazy evaluation*) or *derived-on-update* (*eager evaluation*). In the former case, the derived information is not stored, but computed only when the information is requested. For example, if our Window schema is mapped to a relational database, no column for area is included in the base table for Window. The rule for computing area may be included in a view definition or stored query, and it is invoked only when the view is queried or the stored query is executed. In most cases, lazy evaluation is preferred (e.g., computing a person's age from their birth date and current date).

Sometimes eager evaluation is chosen because it offers significantly better performance (e.g., computing account balances). In this case, the information is stored as

soon as the defining facts are entered, and updated whenever they are updated. In an ORM tool, this option might be chosen by selecting “Derived and Stored” from a predicate dialog box. As a subconceptual annotation, a double asterisk “\*\*” may be used to indicate this choice. When the schema is mapped to a relational database, a column is created for the derived fact type, and the computation rule should be included in a trigger that is fired whenever the defining columns are updated (including inserts or deletes).

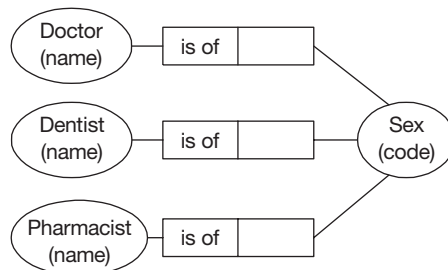
Now consider Figure 3.30. This might describe part of a UoD concerning practitioners in a medical clinic. Here the entity types Doctor, Dentist, and Pharmacist have a similar reference mode (name), but this is not unit based, so this is no reason to combine the types. If somebody could hold more than one of these three jobs, then the overlap of the entity types would normally force a combination.

However, suppose that the entity types are mutually exclusive (i.e., nobody can hold more than one of these jobs). In this case we still need to consider whether the entity types should be combined, since the *same kind of information* (their sex) is recorded for each.

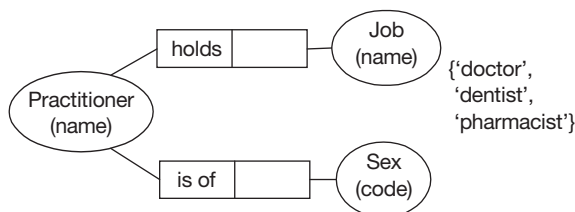
In such cases, we ask ourselves the following question: *Do we ever want to list the same kind of information for the different entity types in the same query?* For example, do we want to make the request “List all the practitioners and their sex”? If we do, then we should normally combine the entity types as shown in Figure 3.31. If we don’t, then there may be grounds for leaving the schema unchanged. Section 6.6 examines this issue in more detail.

Even if no doctor can be a dentist, the schema of Figure 3.30 permits a doctor and a dentist to have the same name (e.g., “Jones E”). In Figure 3.31, the use of “(name)” with Practitioner implies that each instance of PractitionerName refers to only one Practitioner.

Suppose we add the constraint each Practitioner holds at most one Job. A graphical notation for this kind of constraint is discussed in the next chapter. With this constraint added, Figure 3.31 would forbid any doctor from having the same name as a dentist. If the original names did overlap, we would now need to rename some practitioners to ensure their new names are distinct (e.g., “Jones E1” and “Jones E2”). Alternatively, we might choose a new simple identification scheme (e.g., practitionerNr), or identify practitioners by the combination of their original name and job. Reference schemes are discussed in detail in Section 5.4.



**Figure 3.30** Should Doctor, Dentist, and Pharmacist be combined?



**Figure 3.31** An alternative schema for the UoD of Figure 3.30.

To preserve the distinction between the different kinds of practitioners, I introduced the entity type Job, and constrained job names to the set {"doctor", "dentist", "pharmacist"}. Such "value constraints" are discussed in detail in Chapter 6. As an exercise, you might like to invent a small population for this UoD and populate both schemas.

The new schema is simpler than the old one since it replaced three binary fact types with two binaries. If we had even more kinds of practitioners (e.g., acupuncturist, herbalist), the savings would be even more worthwhile. If we have only two kinds of practitioners (e.g., doctor and pharmacist), both schemas would have the same number of fact types. But even in this case, the new version is generally favored. If additional information is required for specific kinds of practitioners, subtyping should be added, as discussed much later.

In rare cases, entity types might overlap, but we are not interested in knowing this, and collapsing the types is awkward. We may then leave the types separate so long as we declare that our model differs from the real world in this respect.

In performing step 3 of the CSDP, the relevant questions to ask ourselves may be summarized thus:

1. *Can the same entity be a member of two entity types?* If so, combine the entity types into one (unless such identities are not of interest).
2. *Can entities of two different types be meaningfully compared (e.g., to compute ratios)? Do they have the same unit or dimension?* If so, combine the entity types into one.
3. *Is the same kind of information recorded for different entity types, and will you sometimes need to list the entities together for this information?* If so, combine the entity types into one, and if necessary add another fact type to preserve the original distinction.
4. *Is a fact type arithmetically derivable from others?* If so, add a derivation rule. If you include the fact type on the diagram, mark it with an asterisk "\*".

At this step, the derivation rules that concern us are of an arithmetic nature. These are usually fairly obvious. Logical derivations can be harder to spot and are considered in a later step.

Besides verbalization and population, a third way to validate a model is to see whether it enables *sample queries* to be answered, either directly from, or by derivation on, the fact populations. If you know what kinds of questions the system must be able

to answer, navigate around the ORM model to see if you can answer them. If you can't, your schema is incomplete, and you should add the fact types and/or derivation rules needed to answer the queries.

### Exercise 3.5

Perform steps 1–3 of the CSDP for the following output reports. In setting out derivation rules, you may use any convenient notation.

1.

<i>Software</i>	<i>Distributor</i>	<i>Retailer</i>
Blossom 1234	CompuWare	PCland SoftKing
SQL++ WordLight	TechSource TechSource	PCland CompuWare SoftKing

2.

<i>Project</i>	<i>Manager</i>	<i>Budget</i>	<i>Salary</i>	<i>Birth year</i>
P1	Smith J	38000	50000	1946
P2	Jones	42000	55000	1935
P3	Brown	20000	38000	1946
P4	Smith T	36000	42000	1950
P5	Collins	36000	38000	1956

3.

<i>Dept</i>	<i>Budget</i>	<i>NrStaff</i>	<i>EmpNr</i>	<i>Salary</i>	<i>Salary total</i>
Admin	80000	2	E01	40000	65000
			E02	25000	
Sales	90000	3	E03	30000	85000
			E04	25000	
			E05	30000	
Service	90000	2	E06	45000	70000
			E07	25000	

4.

<i>Employee</i>	<i>Project</i>	<i>Hours</i>	<i>Expenses</i>
E4	P8	24	200
E4	P9	26	150
E5	P8	14	100
E5	P9	16	110
E6	P8	16	120
E6	P9	14	110

5.

<i>Female staff</i>		<i>Male staff</i>	
<i>Name</i>	<i>Dept</i>	<i>Name</i>	<i>Dept</i>
Sue Bright	Admin	John Jones	Sales
Eve Jones	Admin	Bob Smith	Admin
Ann Smith	Sales		

6. The following excerpt is from the final medal tally for the 1996 Olympics. Only the top five countries are listed here.

	<i>G</i>	<i>S</i>	<i>B</i>	<i>Total</i>
United States	44	32	25	101
Germany	20	18	27	65
Russia	26	21	16	63
China	16	22	12	50
Australia	9	9	23	41
...	...	...	...	...

- (a) Schematize this using binary fact types only.  
 (b) Schematize this using a ternary fact type.

### 3.6 Summary

The following criteria are desirable characteristics for any language to be used for conceptual modeling: expressibility, clarity, simplicity and orthogonality, semantic stability, semantic relevance, validation mechanisms, abstraction mechanisms, and formal foundation. Object-Role Modeling was designed with these criteria in mind. In particular, its omission of the attribute concept from base models led to greater stability and simplicity, while facilitating validation through verbalization and population.

With large-scale applications, the UoD is divided into convenient modules, the *conceptual schema design procedure* is applied to each, and the resulting subschemas are integrated into the global conceptual schema. The CSDP itself has seven steps, of which the first three were discussed in this chapter:

- 1a. Verbalize familiar information examples as facts (domain expert's task).
- 1b. Refine these into formal, elementary facts, and apply quality checks (modeler's task).
2. Draw the fact types, and apply a population check.
3. Check for entity types that should be combined, and note any arithmetic derivations.

*Step 1* is the most important. This is seeded by *data use cases*, which are relevant information *examples* (e.g., tables, forms, diagrams) that are familiar to the domain expert. These are *verbalized in terms of elementary facts*. The domain expert should verbalize the examples informally as facts in natural-language sentences (step 1a). The modeler should complete step 1 by refining these into elementary facts (step 1b). An elementary fact is a simple assertion that an object has some property, or that one or more objects participate together in some relationship. With respect to the UoD, an elementary fact cannot be split into smaller facts without information loss.

Elementary facts are expressed as instantiated logical predicates. A logical *predicate* is a declarative sentence with object holes in it. To complete the sentence, the holes are filled in by object terms. With simple reference schemes, an entity term is a definite description that includes the name of the *entity type*, *reference mode*, and *value* (e.g., “the

Scientist with surname ‘Einstein’”). A value term comprises the name of the value type and the value (e.g., “the Surname ‘Einstein’”).

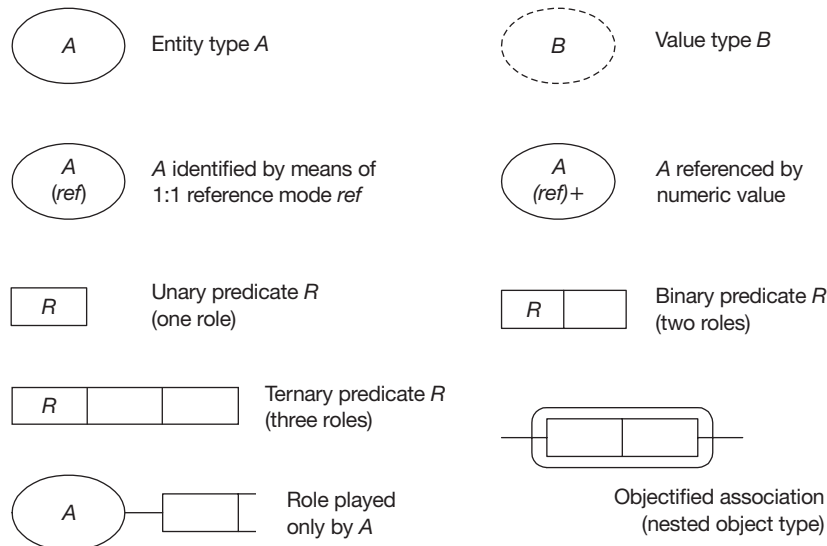
Each object hole corresponds to a *role*. A predicate with one role is *unary*, with two roles is *binary*, with three roles is *ternary*, with four roles is *quaternary*, and with  $n$  roles is  $n$ -ary. The value  $n$  is the *arity* of the predicate.

In CSDP *step 2* we draw the fact types and apply a population check. Entity types are depicted as named, solid ellipses and value types as named, broken ellipses. An  $n$ -ary predicate is shown as a named, contiguous sequence of  $n$  role boxes. Predicates are ordered; the predicate name is placed in or beside the first role of the predicate. Each role is played by exactly one object type, as shown by a connecting line.

A *simple 1:1 reference scheme* involves a reference predicate between an entity type and a value type, where each entity is associated with exactly one value, and each value is associated with only one entity. This kind of scheme may be abbreviated by enclosing the *reference mode in parentheses* next to the entity type name. If the value type is numeric, this may be indicated by adding a “+” sign or by assigning it a numeric data type.

Once a fact type is drawn, it should be checked by populating it with at least one fact and reading it back in natural language. Each column in the associated *fact table* corresponds to a specific role.

A relationship may be thought of as an object itself, if we wish to talk about it. *Objectified associations* or *nested object types* are depicted by a soft rectangle. Figure 3.32 summarizes the graphic notations met so far. Objects are classified into three main groups: atomic entities, nested entities (objectified relationships), and values (strings or numbers).



**Figure 3.32** Some basic symbols used in conceptual schema diagrams.

In *step 3* of the CSDP, we check for entity types that should be combined, and note any arithmetic derivations. For any given UoD, each entity belongs to exactly one of the *primitive entity types* that have been selected (e.g., Person, Car). Hence, if we have drawn two entity types that may have a common instance, we should combine them. Even if they don't overlap, entity types that can be meaningfully compared should normally be combined; these typically have the same unit-bearing reference mode (e.g., cm). If the same kind of information is to be recorded for different entity types, these should often be combined, and if needed, a fact type should be added to preserve the original distinction.

If a fact type is arithmetically derivable from others, an appropriate derivation rule should be provided. The rule may be declared in relational style (using predicates) or in attribute style (using attributes derived from rolenames). Usually the derived fact type is omitted from the diagram, but if included, it should be marked with an asterisk.

## Chapter Notes

The parsimony-convenience trade-off is nicely illustrated by two-valued propositional calculus, which allows for 4 monadic and 16 dyadic logical operators. All 20 of these operators can be expressed in terms of a single logical operator (e.g., nand, nor), but while this might be useful in building electronic components, it is simply too inconvenient for direct human communication. For example, “not p” is far more convenient than “p nand p”. In practice, we use several operators (e.g., not, and, or, if-then) since the convenience of using them directly far outweighs the parsimonious benefits of having to learn only one operator such as nand. When it comes to proving metatheorems about a given logic, it is often convenient to adopt a somewhat parsimonious stance regarding the base constructs (e.g., treat “not” and “or” as the only primitive logical operators), while introducing other constructs as derived (e.g., define “if  $p$  then  $q$ ” as “not  $p$  or  $q$ ”). Similar considerations apply to modeling languages.

A classic paper on linguistics that influenced the early development of a number of modeling methods is Fillmore (1968). Use case specification was central to the former Objectory approach of Ivar Jacobson, one of the main contributors to UML. For a clear overview of logicians' approaches to proper names and descriptions, see Chapter 5 of Haack (1978). Some heuristics to help with step 1 in interpreting common forms used in business are discussed in Choobineh et al. (1992). In this book, the term “line” is used informally to mean “line segment” or “edge” (in geometry, lines have no beginning or end). The following notes provide a brief history of ORM.

In the 1970s, especially in Europe, substantial research was carried out to provide high-level semantics for modeling information systems. Jean Abrial (1974), Mike Senko (1975), and others discussed modeling with binary relationships. In 1973, Eckhard Falkenberg generalized their work on binary relationships to  $n$ -ary relationships and decided that attributes should not be used at the conceptual level because they involved “fuzzy” distinctions and they also complicated schema evolution. Later, Falkenberg (1976) proposed the fundamental ORM framework, which he called the “Object-Role Model”. This framework allowed  $n$ -ary and nested relationships, but depicted roles with arrowed lines.

Shir Nijssen adapted this framework by introducing the now standard circle-box notation for object types and roles, and adding a linguistic orientation and design procedure to provide a modeling method called ENALIM (Evolving Natural Language Information Model) (Nijssen 1976, 1977). A major reason for the role-box notation was to facilitate validation using sample populations. Nijssen led a group of researchers at Control Data in Belgium who developed the method further, including Franz van Assche who classified object types into lexical object types (LOTs) and nonlexical object types (NOLOTs). Today, LOTs are commonly called “entity types”

and NOLOTs are called “value types”. Bill Kent (1977, 1978) provided several semantic insights and clarified many conceptual issues.

Robert Meersman (1982) added subtypes to the modeling framework and invented RIDL, which enabled the conceptual models to be specified, updated, and queried directly at the conceptual level. The method was renamed “aN Information Analysis Method” (NIAM) and summarized in a paper by Verheijen and van Bekkum (1982). In later years the acronym “NIAM” was given different expansions, and is now known as “Natural-language Information Analysis Method”. Two matrix methods for subtypes were developed, one (the role-role matrix) by Dirk Vermeir (1983) and another by Falkenberg and others.

In the 1980s, Falkenberg and Nijssen worked jointly on the design procedure and moved to the University of Queensland, where the method was further enhanced by various academics. It was there that I provided the first full formalization of the method (Halpin 1989b), including schema equivalence proofs, and made several refinements and extensions to the method. In 1989, Nijssen and I coauthored a book on the method. Another early book on the method was written by Wintraecken (1990).

In the early 1990s I developed an extended version of NIAM called Formal ORM (FORM), initially supported in the InfoDesigner modeling tool from ServerWare. This product later evolved to InfoModeler (at Asymetrix Corp., then InfoModelers Inc.), then VisioModeler (at Visio). This ORM technology was then recoded to work on the Visio engine, first appearing in Visio Enterprise. Anthony Bloesch and I designed an associated query language called ConQuer (Bloesch and Halpin 1997), supported in the ActiveQuery tool. Microsoft acquired Visio in 2000 and is currently extending the ORM technology for use within. A number of other commercial and academic institutions have also engaged in research and development to provide CASE tool support for the Visual Studio.net method (e.g., Ascaris Software 2000).

Many researchers have contributed to the ORM method over the years, and a full history would include many not listed here. Today various versions of the method exist, but all adhere to the fundamental object-role framework. Although most ORM proponents favor  $n$ -ary relationships, some prefer Binary-Relationship Modeling (BRM), for example, Peretz Shoval (Shoval and Shreiber 1993). Henri Habrias (1993) developed an object-oriented version called MOON (Normalized Object-Oriented Method). The Predicator Set Model (PSM) was developed mainly by Arthur ter Hofstede, Erik Proper, and Theo van der Weide (ter Hofstede et al. 1993) and includes complex object constructors. Olga De Troyer and Robert Meersman (1995) developed another version with constructors called Natural Object-Relationship Model (NORM). Harm van der Lek and others (Bakema et al. 1994) allowed entity types to be treated as nested roles, to produce Fully Communication Oriented Information Modeling (FCO-IM). The term “Object-Role Modeling” (ORM), originally used by Falkenberg for his modeling framework, is now used generically to cover the various versions of the modeling approach.

Independently of the main ORM movement, Dave Embley and others decided that using attributes in conceptual modeling was a bad idea, so they developed Object-oriented Systems Analysis (OSA) that includes an attribute-free “Object-Relationship Model” component that has much in common with ORM (Embley et al. 1992; Embley 1998).