Preface

If an information system is able to perform useful actions for persons working in a given domain, it is because the system knows something about that domain. The more knowledge it has, the more useful it can be to its users. Without that knowledge, the system is useless.

Most of the knowledge a system has is concrete or particular. It refers to concrete objects and the relationships they have in the domain at some point in time. Given that many systems work in domains with a very high number of objects and relationships, it is hardly surprising that the concrete knowledge they have is very large. Think, for instance, of bank management systems, where it is usual to find a large number of accounts, loans, etc. for which many details must be known (account holders, balances, transactions, etc.).

However, it is not possible to have concrete knowledge about a domain without a prior *general* knowledge about that domain. A bank management system may know the balances of accounts once it knows that there are accounts in the domain, and that accounts always have a balance. Similarly, the system may know the holders of accounts because it knows that accounts do have holders. Concrete knowledge requires prior general knowledge, which is independent of the concrete objects and relationships existing at any point in time.

This general knowledge also includes rules that must be obeyed (for instance, balances may not be negative), definitions that allow new knowledge to be obtained from existing knowledge (for instance, what is understood as the return on investment), and details of the actions that the users want the system to perform when some condition is satisfied (for example, how to calculate the interest earned by savings accounts).

In the information systems field, we use the name *conceptual modeling* for the activity that elicits and describes the general knowledge a particular information system needs to know. The main objective of conceptual mod-

eling is to obtain that description, which is called a *conceptual schema*. Conceptual schemas are written in languages called conceptual modeling languages. Conceptual modeling is an important part of requirements engineering, the first and most important phase in the development of an information system.

The elicitation of the general knowledge required by an information system is a necessary activity. Information systems cannot be designed or programmed without prior elicitation of the knowledge they need to know. This is captured by one of the principles that guide this book, called the *principle of necessity*: "to develop an information system, it is necessary to define its conceptual schema".

The only option we have is whether or not to explicitly describe that knowledge. That is, whether or not to write the conceptual schema. Sometimes, system development projects choose not to write the conceptual schema, or they do not have the time to do so. In these cases, the general knowledge is in the designers' heads only. This has many disadvantages. If there are several designers, they must share this knowledge without an explicit description. User participation is hampered. Once the system has been built, it is likely that the general knowledge will be forgotten. The future evolution of the system will require that general knowledge to be rediscovered. The explicit description of the conceptual schema brings many advantages, especially when it is done in a machine-readable language.

Furthermore, many researchers have put forward, many times, a vision in which the conceptual schema is the only important description that needs to be created in the development of an information system. According to this vision, the building of information systems is completely automated. The only things to be done are to determine the functions that the information system has to perform and to define its conceptual schema (and, probably, the design and construction of the input/output user interface). The huge potential economic benefit of this vision justifies the research and development efforts currently devoted to it, which are being made mainly in the framework of OMG's Model Driven Architecture. The progress made in other branches of computer science (especially in the field of databases) makes this vision feasible in the mid-term. On the day when the vision becomes a reality we shall be able to say that "to develop an information system it is necessary and sufficient to define its conceptual schema".

Objectives

The main objectives of this book are:

1. To describe the principles of conceptual modeling independently of particular methods and languages.
2. To describe these principles in the detail required to correctly apply them in real projects and to be able to assess the methods, languages, and tools that are most suitable in those projects.
3. To describe the formal bases of conceptual schemas. However, in this book, the logical formalization is only sketched and is not pushed too far. The book describes the formal bases with extensive use of intuitive ideas and examples.
4. To describe in detail the use of standard UML/OCL as a particular conceptual modeling language.
5. To provide exercises for readers who want to practice and deepen their knowledge by solving exercises.
6. To give bibliographical references for the concepts presented in the book and for the extensions suggested to readers, including further formalizations.

Audience

The book has two intended audiences:

1. Computer science and information systems students who, after an introduction to information systems, databases, and UML, want to know more about conceptual modeling in their preparation for professional practice.
2. Professionals with some experience in the development of information systems who feel a need to formalize their practical experiences or to update their knowledge, as a way to improve their professional activity.

Some prerequisite knowledge is assumed – and necessary – in order to benefit from the book:

1. Knowledge of the fundamental concepts of the language of first-order logic.
2. Knowledge of fundamental concepts of object technology, such as classes, operations, and inheritance.

3. Knowledge of the fundamental constructs of ER and UML for information modeling. A basic knowledge of OCL is necessary from Chap. 8 onwards.

Structure of the Book

The 18 chapters of this book are divided into five logical parts:

- *Chapter* 1: *introduction*. Here we give a general view of conceptual modeling. Readers with prior knowledge about the field may skip this chapter, but it may be useful to those who want to recall concepts and terms learnt long ago.
- *Chapters* 2–10: *structural modeling*. Here we study the concepts of entity types, relationship types, constraints, derivation rules and taxonomies.
- *Chapters* 11–15: *behavioral modeling*. Here we describe the concepts of events, their constraints, and their effects. We also describe behavioral modeling with state machines and statecharts. We include a review of the concept of the use case and its relationship to the conceptual schema.
- *Chapter* 16: *a case study*. In the preceding chapters, we follow a bottom-up approach, starting with the basic elements of entity and relationship types, and then proceeding to more complex elements until we reach state transition diagrams and statecharts. In this chapter, we provide an integrated view of conceptual modeling by means of a case study.
- *Chapters* 17 and 18: *metamodeling*. Here we introduce the main concepts of metamodeling and describe their use. We study two important standards related to metamodeling: the MOF and XMI.

Figure I.1 shows the main precedence relationships among the chapters of this book.

The book also includes a companion website (http://www-pagines.fib.upc.edu/~modeling) where students and professionals can find additional exercises, case studies, reading material and presentations on selected topics. If you have any comments on the book, any typos you have noticed, or any suggestion on how it can be improved, I would like to hear from you. The companion website includes information on how to contact me.

For the convenience of the reader, in this book I use “he” to refer to both genders.

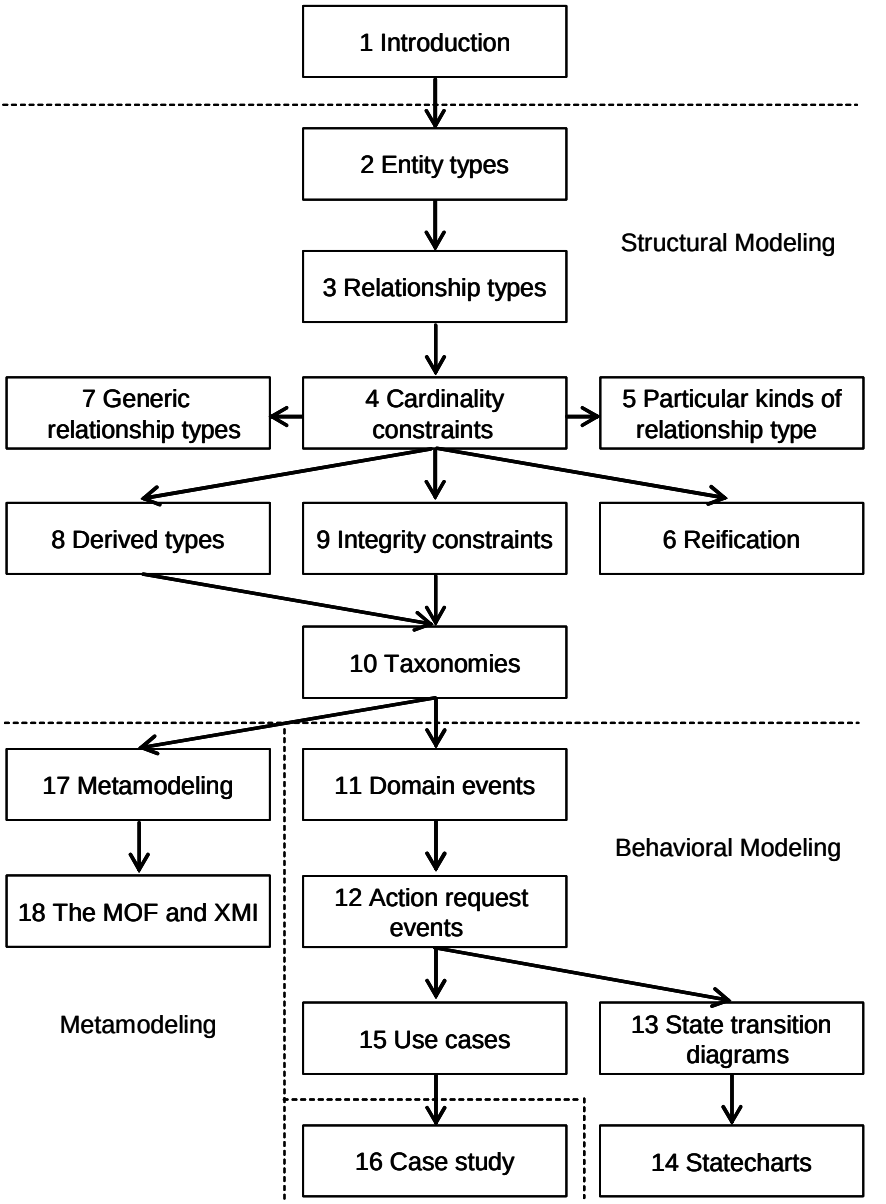

Fig. I.1. Main precedence relationships among the chapters of this book

Acknowledgements

This book explains part of what I have learnt over the years that I have dedicated to the study of conceptual modeling. I have learnt a lot from other people's work. The extensive bibliography shows clearly enough how the ideas and realizations of others have contributed to this book. My first thanks to all these people. I also express my gratitude to the members and friends of the IFIP WG8.1, CAiSE, and ER communities for the many opportunities they have given to me to share and discuss views on what this book is about.

Many people contributed directly and indirectly to bringing this text into existence. First and foremost, I would like to thank my colleagues of the GMC group (whose current composition is: Jordi Cabot, Jordi Conesa, Dolors Costal, Cristina Gómez, Enric Mayol, Joan Antoni Pastor, Anna Queralt, Ruth Raventós, Maria Ribera Sancho, and Ernest Teniente) for providing a friendly environment in which I have been able to discuss many ideas that have helped me to sharpen and focus this book, and for the many comments and suggestions they have given to me over the years.

During the thirteen-year period in which the ideas expressed in this book were developed, from their first expression as lecture notes in Catalan until their current form and language, many friends, colleagues and students have reviewed, read and commented on previous drafts of this book or its chapters. Their help is gratefully acknowledged. Among them, I specially thank Carles Farré, Oscar Díaz, Juan Ramón López, Elena Planas, Carme Quer, Sudha Ram, Keng Siau, Albert Tort and Toni Urpí. Preliminary versions of this book have been used in several courses on conceptual modeling. I deeply appreciate the many comments made by the students. Any remaining errors are, of course, mine.

I would like to thank my university, the Technical University of Catalonia, for the continued support that it has provided to me during these years, and the research agencies that have funded part of my research. Finally, I would also like to thank Ralf Gerstner, from Springer, for his continued interest and help in this book.

Barcelona, June 2007 Antoni Olivé

3 Relationship Types

Relationship types are another important element in conceptual schemas, because they also play a fundamental role in the memory, informative, and active functions of information systems. Determining the relationship types that are relevant to an information system is one of the most important tasks in conceptual modeling. In this chapter, we study the nature and general characteristics of a relationship type.

In the first section, we define relationship types similarly to the way in which entity types were defined in the preceding chapter. In Sect. 3.2, we explain how to represent relationships and their types in an information system, in both logic and UML. Attributes are a particular but important kind of relationship type, which is addressed in Sect. 3.3.

3.1 Definition

Relationship types can be defined in several ways. Here, we adopt a definition similar to that of entity types. The rationale is that there is a similarity between the operation of classifying an object into an entity type and that of classifying a relationship into a relationship type. For example, there is little difference between classifying the reader of this book as a *Person* and that of classifying what he does with the book as *Reads*. In both cases we abstract something: in *Person*, we ignore the differences that exist between people; in *Reads*, we ignore how, where, and why the book is being read, the reader's interest in the book, the level of difficulty, and so on. Thus, it would be reasonably accurate to use a definition of a relationship type that is almost identical to that of an entity type: "A relationship type is a concept whose instances at a given time are identifiable individual relationships that are considered to exist in the domain at that time."

However, the above definition is unsatisfactory because it ignores the fact that a relationship is always a relationship between objects. Although we can imagine isolated objects in a domain, it is impossible to imagine a relationship without the presence of objects. The *participants* in a relationship are the objects that participate in it. Each participant plays a *role* in

the relationship. In the above example, the participants are *The reader of this book* and *This book*. The former plays the role of reader, and the latter the role of reading.

If we move on from instances to types, we can say that a relationship type consists of a set of n participants, with $n \geq 2$. A participant is an entity type that plays a role in a relationship type. We write $R(p_1{:}E_1, \ldots, p_n{:}E_n)$ to denote a relationship type named R, with participant entity types $E_1, \ldots, E_n$ playing roles $p_1, \ldots, p_n$, respectively. Sometimes we omit the role p_i played by participant $p_i{:}E_i$, either because it is obvious or because it is the same as the name of E_i. In these cases, it is assumed that p_i is the same as E_i. For example, *Reads* (*reader*:*Person*, *Book*) is equivalent to *Reads* (*reader*:*Person*, *book*:*Book*).

We say that $R(p_1{:}E_1, \ldots, p_n{:}E_n)$ is the *schema* of the relationship type R and that $p_1{:}E_1, \ldots, p_n{:}E_n$ are its participants. Conceptually, the order of the participants in the schema is not significant. Two different participants can be of the same entity type, but there cannot be two participants with the same role. For this reason, we sometimes say simply that $p_1, \ldots, p_n$ are the participants of the relationship type.

Using the above notation and terminology, the following could be an acceptable definition: "A *relationship type* $R(p_1{:}E_1, \ldots, p_n{:}E_n)$ is a concept whose instances at a given time are distinct sets $\{<p_1{:}e_1>, \ldots, <p_n{:}e_n>\}$ formed by n entities $e_1, \ldots, e_n$ that are instances of their corresponding type $E_1, \ldots, E_n$, and are considered to have a relationship R in the domain at that time, playing the respective roles $p_1, \ldots, p_n$." The instances of a relationship type are called *relationships*.

This definition states, in the first place, that relationship types are concepts. Therefore, the concept theories developed in cognitive science are, in theory, also applicable to relationship types.

Secondly, the definition states that a relationship is a set $\{<p_1{:}e_1>, \ldots, <p_n{:}e_n>\}$ formed by exactly one pair $<p_i{:}e_i>$ for each of its n participants. For example, the instances of *Supplies* (*Supplier*, *Part*, *user*:*Project*) must include an entity from each of the three entity types. If there were an instance of *Supplies* without, say, a *user* role then the relationship type would be incorrect. The definition requires each instance of *Supplies* to include a *user* role. Another incorrect case would occur if we were to accept that an instance of *Supplies* could consist of a supplier *S*, a part *P*, and user projects *A*, *B*, and *C*. The definition requires each instance of *Supplies* to include exactly one *user*; thus, we should consider that in this case there are three relationships, not one.

The fact that a relationship must include one entity for each participant should not be confused with the fact that two or more participants may be of the same entity type. A relationship type such as *IsParentOf* (*par-*

ent:*Person*, *child*:*Person*) is totally valid. An instance of this relationship type will include two persons, one for each participant (*parent* and *child*). A relationship type in which the same entity type plays two or more roles is called *recursive*.

Although it is rare, a relationship could include the same entity twice, as distinct participants. For example, the recursive type *Knows* (*Person*, *acquaintance*:*Person*) could have instances in which the two people are the same (if it were the case that a person knew himself).

Thirdly, the definition states that the instances of a relationship type must be distinct sets. It is not possible to have two relationships in the domain that are formed by exactly the same participants. In the *Supplies* example, there cannot be two relationships with the same *supplier*, *part*, and *project*.

Finally, the definition states that the instances of a relationship type *R* are sets of entities that are considered to have the relationship *R* in the domain at that time, playing their respective roles. It is important to note that the definition says that they "are considered to have the relationship *R*". As with entity types, this aspect does not cause difficulties in most relationship types, but there are cases that raise doubts because what some people may consider to exist at a given point in time, other people may consider not to exist (at least at that point in time). When this happens, an effort must be made to refine the definition and reach an agreement between the people involved.

An example of a relationship type that should not cause any problem in this respect is *Lives* (*resident*:*Person*, *placeOfResidence*:*Town*). If a person *p* and a town *to* are the participants in an instance of this type at some time, it is because *p* lives in *to* at that time.

An example that may raise some doubts is *IsMotherOf* (*mother*:*Woman*, *child*:*Person*). An instance of this type at time *t* involves a woman *m* and a person *p* if it is considered at *t* that *m* is the mother of *p*. Let us consider two people: Alice and Alan (see Fig. 3.1). Alice is born and dies at T_1 and T_3 respectively. At T_2, Alice gives birth to Alan, who dies at T_4. Figure 3.1 shows three possible interpretations of the relationship *IsMotherOf* (*Alice*, *Alan*):

(a) The relationship holds only while the child, *Alan*, is alive.
(b) The relationship holds only while both the mother and the child are alive.
(c) Once *Alice* has given birth to *Alan*, the relationship holds forever.

Which of these interpretations is best must be defined in the domain. The one that is chosen will have an impact on the answer to the question

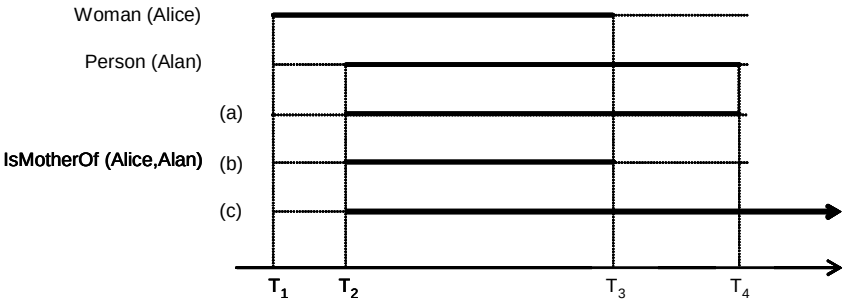

Fig. 3.1. Three interpretations, (a), (b) and (c), of the temporal existence of the relationship *IsMotherOf* (*Alice*, *Alan*) in a domain

"*who is p's mother now?*" In Fig. 3.1, the answer to this question at $T_3 < t < T_4$ and at $T_4 > t$ will depend on the interpretation chosen.

Note that the definition given above does not require that the entities participating in a relationship at time *t* must be an instance of their types at *t*. Although they usually will be, they might have been an instance at some previous time without being so at *t*. We can see the difference in the two previous examples. In *Lives* (*resident*:*Person*, *placeOfResidence*:*Town*), if person *p* lives in town *to* at time *t*, it is likely that *p* is required to be an instance of *Person* at *t*, and *to* to be an instance of *Town* at *t*. However, in *IsMotherOf* (*mother*:*Woman*, *child*:*Person*), it may be acceptable for woman *m* to be considered the mother of person *p* at time *t* even if *m* is not considered an instance of *Woman* at *t*. In interpretation (a) in Fig. 3.1, in the interval from T_3 to T_4 *Alice* is considered to be *Alan*'s mother even if *Alice* is no longer an instance of *Woman*. In interpretation (c), after T_4 *Alice* is also considered to be *Alan*'s mother, even if neither *Alice* nor *Alan* is an instance of its type at that time.

We say that $R(p_1{:}E_1, \ldots, p_n{:}E_n)$ is *synchronous* if, for each of its instances $r = \{<p_1{:}e_1>, \ldots, <p_n{:}e_n>\}$ at time *t*, the entities $e_1, \ldots, e_n$ participating in *r* are instances of their respective types at *t*; otherwise, *R* is *asynchronous*. In the examples above, *Lives* is synchronous, while *IsMotherOf* is asynchronous according to interpretations (a) and (c) in Fig. 3.1. Almost all relationship types are synchronous. Asynchronous relationship types can be defined only in temporal conceptual models. In this book, unless stated otherwise, we assume that relationship types are synchronous.

3.1.1 Degree

The *degree* of a relationship type is the number of participants in that type. Most relationship types have degree 2, and are called *binary*. Relationship types with a degree greater than 2 are called *n-ary*. The most common of the *n*-ary relationship types are those that have degree 3, which are referred to as *ternary*.

3.1.2 Pattern Sentence

Linguistically, a relationship is a fact that holds in a domain and can be expressed using a grammatical sentence. For example, an instance of *Reads* (*reader*:*Person*, *reading*:*Book*) with participants *Arnold* and *Alice in Wonderland* may be expressed using the sentence "Arnold reads Alice in Wonderland", while another with participants *Laura* and *Tirant Lo Blanc* could be expressed as "Laura reads Tirant Lo Blanc", etc.

The *pattern sentence* of a relationship type is a declarative sentence in which there is a placeholder for each participant. The sentence that linguistically expresses a relationship is obtained by filling in the placeholders with the names of the participants. In the above example, the pattern sentence could be

<*Person*> reads <*Book*>

where <*Person*> and <*Book*> are slots that must be filled in with (the names of) a specific person and book respectively. If we wish to express that the first participant is a person and the second a book, we may use a longer pattern sentence:

Person <*Person*> reads the book <*Book*>

which is instantiated as

The person *Arnold* reads the book *Alice in Wonderland*
The person *Laura* reads the book *Tirant Lo Blanc*

Pattern sentences help us to understand the meaning of relationship types and may be *implicit* or *explicit*. The former are built as explained below, and the latter are given by the designers. Every relationship type has one or more implicit pattern sentences. Explicit pattern sentences are optional.

Implicit pattern sentences are derived from the schema $R(p_1{:}E_1, \ldots, p_n{:}E_n)$. If we choose the names of the relationship types (R) and roles (p_1)

sensibly, the sentences derived may be enough, and we shall not need explicit pattern sentences.

The derivation of implicit pattern sentences depends on how roles are named. There are two approaches: *noun-based* and *verb-based*. In the noun-based approach, role names are nouns, while in the verb-based approach, role names are verbs. An example of the former is *R* (*reader*:*Person*, *reading*:*Book*), where the roles (*reader* and *reading*) are nouns. The same example in the latter approach could be *R* (*reads*:*Person*, *is read by*:*Book*), where the roles (*reads*, *is read by*) are verbs.

Firstly, we shall describe the derivation of implicit pattern sentences for binary types when the role names are nouns. In this case, there are three implicit pattern sentences. Before giving their general structure, we show the sentences for the example *Reads* (*reader*:*Person*, *reading*:*Book*):

The person <*Person*> reads the book <*Book*>
The person <*Person*> is (a | the) reader of the book <*Book*>
The book <*Book*> is (a | the) reading of the person <*Person*>

In the option (*a* | *the*) in the second sentence, we use *a* if a book may be read by several persons, and *the* if it can be read by one person at most.[1] In the third sentence, we use *a* if a person can read several books at the same time, and *the* if he or she can read one at most. The instantiation of these pattern sentences for a particular relationship could be as follows (assuming that a person reads several books and that the same book may be read by several persons):

The person *Laura* reads the book *Tirant Lo Blanc*
The person *Laura* is a reader of the book *Tirant Lo Blanc*
The book *Tirant Lo Blanc* is a reading of the person *Laura*

Note that the first sentence is well formed when the name of the relationship type (*Reads*) is the verb of the sentence in third-person singular and the two participants in *R* appear in an appropriate order. The other two sentences are well formed when the role names are singular nouns.

The general structure of the three pattern sentences corresponding to a binary relationship type $R(p_1{:}E_1,p_2{:}E_2)$ is as follows:

The $e_1 < E_1>$ *R* the $e_2 < E_2>$
The $e_1 < E_1>$ is (a | the) p_1 of the $e_2 < E_2>$
The $e_2 < E_2>$ is (a | the) p_2 of the $e_1 < E_1>$

[1] The information about how many people can read a book (or how many books can be read by a person) at the same time can be extracted from the cardinality constraints, which are considered in the next chapter.

We shall now go on to explain the derivation of implicit pattern sentences for binary types when the role names are verbs. In this case, the name of the relationship type is not used, and there are two implicit pattern sentences. Before giving their general structure, we show the sentences for the example *R* (*reads*:*Person, is read by*:*Book*):

The person <*Person*> reads the book <*Book*>
The book <*Book*> is read by the person <*Person*>

The instantiation of these pattern sentences for a particular relationship could be as follows:

The person *Arnold* reads the book *Alice in Wonderland*
The book *Alice in Wonderland* is read by the person *Arnold*

Note that the sentences are well formed when the role names are verbs in third-person singular.

The general structure of the two pattern sentences corresponding to a binary relationship type $R(p_1{:}E_1,p_2{:}E_2)$ is as follows:

The $e_1 < E_1 > p_1$ the $e_2 < E_2 >$
The $e_2 < E_2 > p_2$ the $e_1 < E_1 >$

For most binary types, at least one of the implicit pattern sentences captures the meaning of the corresponding relationship. When this happens, there is no need to define explicit pattern sentences.

In some cases, however, it may be difficult to find adequate names, making it necessary to define explicit pattern sentences. For example, consider the following popular relationship type found in online bookstores: "Customers who bought this book also bought this other book". It is difficult to choose names for the type and its two roles so that the implicit pattern sentences express the meaning of the relationships. A schema such as

CustomersAlsoBought (*origin*:*Book, additional*:*Book*)

does not produce expressive sentences. Therefore, in this case it would be appropriate to define an explicit pattern sentence, such as

Customers who bought book <*Book*> also bought book <*Book*>

An instance might be

Customers who bought the book *Introduction to Conceptual Modeling* also bought the book *Advanced Conceptual Modeling*

For *n*-ary types, it is difficult to find names of relationship types and roles from which we can derive expressive pattern sentences. Normally,

we need to define explicit pattern sentences for these types. For example, a pattern sentence for *Supplies* (*Supplier*, *Part*, *user*:*Project*) could be

> Supplier <*Supplier*> supplies part <*Part*> to be used in project <*Project*>

3.1.3 Unary Relationship Types

Normally, relationships are conceived of as having at least two participants. In fact, almost all conceptual modeling languages require that relationship types have a degree of at least two.

However, when we develop a conceptual schema for a domain, we find concepts that seem to be naturally modeled as unary relationship types. For example, we could have *IsThick* (*Book*) to represent the fact that a book is thick. Another example could be *IsManager* (*Person*).

In languages that do not allow unary relationship types, the above concepts can be modeled as new entity types. For example, we could define the type *ThickBook*, or the type *Manager*. This is a valid and elegant solution, but it adds new entity types to the schema.

Another solution involves modeling a unary type as binary, with an additional participant that may take two values: *true* and *false*. For example, if *Boolean* is a data type we could define *ThickBook* (*Book*, *Boolean*) and *Manager* (*Person*, *Boolean*). This is also a valid solution, and a practical one in some languages, but few people will find it elegant.

There is a third solution, which can be applied when we have two or more unary types with the same participant entity type *E*. An example of this situation might be when we have *IsManager* (*Person*) and *IsSalesman* (*Person*). In this case, we could define a binary type *P*(*E*,*E'*), where *E'* is an entity type with as many instances as the unary relationship types that we have. In this example, we could have *P* (*Person*, *JobCategory*), where *JobCategory* has the instances *Manager* and *Salesman*. In UML, *JobCategory* could be defined as an enumeration.

3.1.4 Population

The *population* of a relationship type *R* at time *t* is the set of its instances that exist in the domain at *t*.

In general, the population of a relationship type is time-varying during the lifetime of an information system. However, there are two particular cases that deserve special treatment: constant and permanent relationship types. Their definition bears some resemblance to that of entity types, but it is not the same.

A relationship type $R(p_1{:}E_1, \ldots, p_n{:}E_n)$ is *constant* with respect to a participant p_i if the instances of R in which an instance e_i of E_i participates are the same during the temporal interval in which e_i exists. We shall illustrate this definition with two examples. The first is

WasBorn (*native*:*Person*, *birthplace*:*Town*)

We assume that *Town* is constant and that *Person* is not permanent, meaning that persons are born and die. *WasBorn* is constant with respect to *native* because the set of instances r = {<*native*:*p*>, <*birthplace*:*to*>} in which a person *p* participates is constant during *p*'s life. Note that *WasBorn* is not constant with respect to *birthplace*, because the set of people born in a town may change over time.

The second example is:

Equivalence (*source*:*Unit*, *conversionRate*:*Decimal*, *target*:*Unit*)

with the explicit pattern sentence

A <*Unit*> is equivalent to <*Decimal*> <*Unit*>

which produces sentences such as

An *inch* is equivalent to 2.54 *centimeters*

Equivalence is constant with respect to its three participants. The set of instances of *Equivalence* corresponding to a pair of *units* and a *decimal* is the same at any time.

A relationship type is *constant* if it is constant with respect to all its participants. *Equivalence* is constant, but *WasBorn* is not.

A relationship type $R(p_1{:}E_1, \ldots, p_n{:}E_n)$ is *permanent* with respect to a participant p_i if the instances of R in which an instance e_i of E_i participates never cease to exist during the temporal interval in which e_i exists. In the example above, if *Person* were permanent, *WasBorn* would be permanent with respect to *birthplace*, because the set of people born in a town would never decrease. As another example, consider

HasVisited (*visitor*:*Person*,*Town*)

If we assume that *Town* is constant and that *Person* is permanent then *HasVisited* is permanent with respect to *visitor* and *town* because once a person has visited a town, he has visited it forever.

A relationship type is *permanent* if it is permanent with respect to all its participants. *HasVisited* is permanent, but *WasBorn* is not.

3.1.5 Subsumption

In general, the population of a relationship type is independent of that of the other types defined in a schema. However, in some cases the population of a relationship type must necessarily be included in that of another type. Using a definition similar to that of entity types, we say that R_2 subsumes R_1 or that R_1 is a subtype of R_2 if all instances of R_1 must also be instances of R_2. A formal definition is provided in the next section. For example, consider the following relationship types:

Works (*employee*:*Person*, *employer*:*Company*)
Manages (*manager*:*Person*, *Company*)

If we assume that the managers of a company are employees of that company, then *Manages* is a subtype of *Works*. We shall study the subsumption of relationship types in Chap. 10.

3.2 Representation in an Information System

As we already know, in order to be able to perform their functions, information systems must have a representation of their domain. In terms of the elements considered in this chapter, this means that an information system must have a representation of:

- the relationship types;
- the relationships in the domain;
- the classification of the relationships into relationship types.

The relationship types are represented in the conceptual schema. The schema contains a symbol (which is generally chosen arbitrarily) for each relevant relationship type. The concrete form of the symbol is irrelevant at the conceptual level. We need only to assume that the symbols exist and that there is a biunique correspondence between the relationship types represented in the information system and the symbols representing them in the conceptual schema. In this book, we normally designate these symbols with words beginning with a capital letter. When confusion is unlikely, however, we do not distinguish between relationship types and their symbols.

The relationships that exist in the domain and the classification of relationships according to their types are represented in the information base.

Below, we sketch the representation in first-order logic and describe the use of UML in more detail.

3.2.1 State of the Information Base

In the previous chapter, we defined the state of an information base at a given time as the set of facts it contained at that time. In that chapter, the facts were entity facts. We have now seen that there are also relationship facts. Therefore, the state of an information base consists of the entity and relationship facts represented in the information system. There are no other fact types.

3.2.2 Logical Representation

In logic, we represent a relationship type $R(p_1{:}E_1, \ldots, p_n{:}E_n)$, with degree n, using a predicate R with the same degree, where the n arguments are symbols denoting objects or values. The order of the arguments is conventional. We assume the order used in the schema. If the name R is unique, then the name of the predicate is also R. Otherwise, given that there cannot be two predicates with the same name, we use some variation of the name R. We write the predicate's name starting with a capital letter and without blanks. Note that in this representation, the role names disappear.

A relationship $r = \{<p_1{:}A_1>, \ldots, <p_n{:}A_n>\}$ that is an instance of $R(p_1{:}E_1, \ldots, p_n{:}E_n)$ is represented using a formula $R(A_1, \ldots, A_n)$, where R is the predicate corresponding to the relationship type and $A_1, \ldots, A_n$ are the symbols that denote the entities or the values of the participants. The formula $R(A_1, \ldots, A_n)$ indicates simultaneously that $A_1, \ldots, A_n$ are related in the domain and that the relationship they have is of type R. Note that, in logic, there cannot be duplicate formulas and therefore there cannot be duplicate relationships. In logic, the formulas consisting of a simple predicate with constant arguments are called facts. For this reason, formulas $R(A_1, \ldots, A_n)$ are called *relationship facts*, or simply facts.

For each synchronous relationship type $R(p_1{:}E_1, \ldots, p_n{:}E_n)$, the schema must include *n referential* integrity constraints

$$R(e_1, \ldots, e_n) \rightarrow E_1(e_1)$$
$$\ldots$$
$$R(e_1, \ldots, e_n) \rightarrow E_n(e_n)$$

These constraints guarantee that each participant entity is an instance of its corresponding type. The referential constraint is the most important kind of constraint in conceptual modeling.

In logic, we represent that $R_1(p_{1,1}{:}E_1, \ldots, p_{1,n}{:}E_n)$ is a subtype of $R_2(p_{2,1}{:}E_1, \ldots, p_{2,n}{:}E_n)$ using the formula

$$R_1(e_1, \ldots, e_n) \rightarrow R_2(e_1, \ldots, e_n)$$

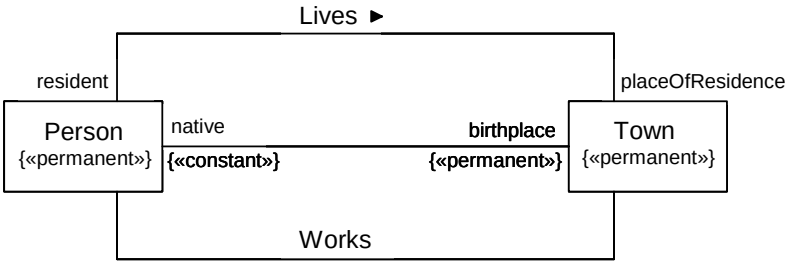

Fig. 3.2. Graphical representation of three relationship types as UML associations

For example, to state that

Manages (*manager*:*Person*, *Company*)

is a subtype of

Works (*employee*:*Person*, *employer*:*Company*)

we write

$Manages(p,c) \rightarrow Works(p,c)$.

3.2.3 Representation in UML

In UML, binary relationship types can be represented in two ways: either as associations or as attributes. We consider associations here and attributes in the next section.

Binary associations are represented graphically by means of a line connecting the two entity types. The name of the association is shown near the line. The names of the roles are placed near their corresponding entity types. Although it is not explicitly prescribed in the official documentation, most users of UML define role names using nouns. Figure 3.2 shows the graphic representation of the associations corresponding to the following types:

Lives (*resident*:*Person*, *placeOfResidence*:*Town*)
WasBorn (*native*:*Person*, *birthplace*:*Town*)
Works (*Person*, *Town*)

In the case of the first example, we show the names of the relationship types and those of the two roles. The solid arrowhead next to the name of the association that points toward a participant indicates the order of the

participants used in the derivation of implicit pattern sentences. When the order is left to right or top to bottom, we usually omit the arrowhead. In this example, the implicit pattern sentences are:

Person <*Person*> lives in town < *Town*>
Person <*Person*> is a resident of town <*Town*>
Town <*Town*> is the place of residence of person <*Person*>

These sentences are reasonably expressive.

In the case of the second example, we omit the name of the relationship type. There will only be two implicit pattern sentences, which are expressive enough:

Person <*Person*> is a native of town <*Town*>
Town <*Town*> is the birthplace of person <*Person*>

In the case of the third example, the names of the roles are the same as those of the entity types. The single implicit pattern sentence is also expressive:

Person <*Person*> works in town <*Town*>

Given that the order of the participants is from left to right, an arrowhead is not needed in this case.

In UML, the name of an association is optional. When there is no name and we need to refer to an association, we use the names of the roles. An example is shown in the case of the association *native-birthplace* in Fig. 3.2.

Two or more associations may have the same name, but, conceptually, each association shown in a diagram is unique. The role names are optional; when they are missing, they are assumed to be the name of the entity type starting with a lowercase character.

In UML, it must be possible to unambiguously navigate from one entity type to the others with which it is connected using only the role names. In Fig. 3.2, we can navigate from *Person* to *Town* using the role names *placeOfResidence*, *birthplace*, and *town*. An ambiguity arises if we add a new association between *Person* and *Town*, such that the name of the role played by *Town* is one of the other three. For example, the following would not be admissible:

HasVisited (*Person*, *Town*)

It must be defined with different role names, such as

HasVisited (*visitor*:*Person*, *visitedTown*:*Town*)

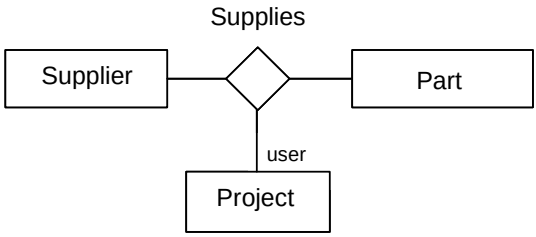

Fig. 3.3. UML representation of a ternary relationship type

Similarly, we can navigate from *Town* to *Person* using the role names *resident*, *native*, and *person*. As before, an ambiguity arises if we define a new association between *Person* and *Town*, with *Person* playing a role named *placeOfResidence*, *birthplace*, or *person*. Sometimes it is difficult to find good role names.

UML represents *n*-ary relationship types as associations. An association is drawn as a diamond, with a solid line for each participant connecting the diamond to the corresponding entity type. The name of the association is placed inside or near the diamond. The names of the roles are placed near their corresponding entity types. An example is shown in Fig. 3.3.

In UML, it is not necessary to explicitly define the referential integrity constraints. The symbol used to represent an association (either a line or a diamond and lines) connects the participating entity types, and from here those constraints are defined implicitly.

In UML, it is not possible to formalize whether a relationship type is constant or permanent.[2] However, as we did for entity types, we can assume that there are two constraint stereotypes (named *constant* and *permanent*), whose formalization is implicit. On the basis of this assumption, we have only to attach the constraint stereotype to the corresponding participant or association. In Fig. 3.2, we have defined that the association *native-birthplace* is constant with respect to *native* and permanent with respect to *birthplace*.

As with entity types, in UML we represent association R_1 as a subtype of R_2 using a solid-line path from R_1 to R_2 and a large hollow triangle at the end of the path where it meets R_2.

Normally, concrete relationships are not shown in diagrams. However, if needed, UML provides a notation that can be used for showing relation-

[2] UML 2.1 has a concept of changeability of association participants and attributes, but it is not expressive enough to capture the semantics of constant and permanent constraints.

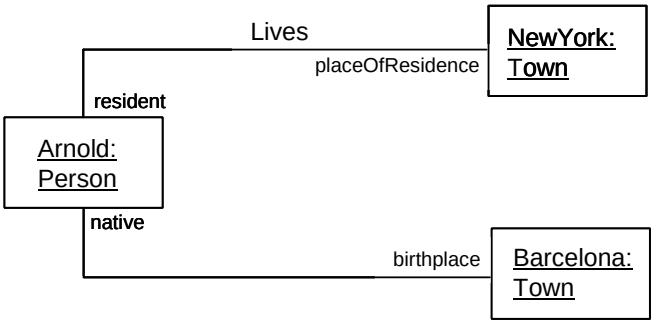

Fig. 3.4. Graphic representation of two links

ships in object diagrams. A concrete relationship is shown using the same notation as for an association, but the solid path or paths connect entities rather than entity types. In UML, instances of associations are called *links*. The graphic representation of relationships is useful to illustrate a fragment of complex schemas. Figure 3.4 shows two links of the associations defined in Fig. 3.2.

The UML graphical representation of relationships may not be practical when we want to show many links, because the figures become large. In these cases, a tabular representation may be a better choice.

3.2.3.1 Ordered Participants

Consider the relationship type

WaitingList (*Flight*, *passenger*:*Person*)

shown in Fig. 3.5, with the pattern sentence

The person <*Person*> is in the waiting list of flight <*Flight*>

If a given flight has several people in its waiting list, there will be an instance of *WaitingList* for each passenger in the list, but these instances are unordered. We cannot assume that there is a first instance corresponding to the first passenger in the list, a second one corresponding to the second, and so on.

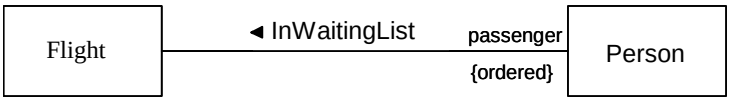

Fig. 3.5. The passengers in the waiting list of a flight are ordered

If we are interested in representing the order of people in the list, we can add a third participant to the relationship type,

WaitingList (*Flight*, *passenger*:*Person*, *order*:*Ordinal*)

which now has a pattern sentence such as

> The person <*Person*> is the <*Order*> in the waiting list of flight <*Flight*>

This is an acceptable solution, but it is difficult to maintain in an information base. One of the main problems is that when a passenger leaves a waiting list, the order of the people that follow him changes. Another problem is that we need to define an integrity constraint requiring that, in a list, the orders must be consecutive.

UML provides a construct that is useful in cases such as this one: *ordered* participants. In the above example, we would define a binary association and indicate that the passenger participant is ordered, using the keyword *ordered* in braces. The meaning is that the passengers in the waiting list of a flight are ordered. The passengers of a flight can be obtained in the order in which they are in the list. A passenger can be added in any position of the list. When a passenger leaves a list, the order of the people that follow him in the list is updated automatically. Using this construct, the above constraint is not needed.

3.2.4 Properties of the Representation

In the previous chapter, we saw that, independently of the language used, the representation of entities and their types in an information system must satisfy a few properties, mainly completeness, correctness, nonempty population, and nonredundancy. The same applies to the representation of relationships and their types. In the following, we briefly describe each of these properties.

3.2.4.1 Completeness

This property states that if a relationship *r* in the domain is considered to be an instance of *R* at a given time, and we want to represent this fact in an information system, then the information base must contain a representation of *r* and a representation of the classification of *r* into *R* at that time.

3.2.4.2 Correctness

This property states that if at some instant the information base contains a classification of a relationship *r* into relationship type *R*, then in the domain it must be considered that *r* is an instance of *R* at that instant, and this should be represented in the information system.

3.2.4.3 Nonempty Population

It must be possible for any relationship type defined in the schema to have a nonempty population. Otherwise, that type would not have any instance during the system's lifetime. An example of a relationship type that would always have an empty population is *Writes* (*author*:*Person*, *Machine*). Assuming that nobody writes a machine, *Writes* would always have an empty population. No valid instance of *Writes* may exist in the domain such that its second participant is a machine.

A relationship type is *satisfiable* if it may have a nonempty population at a certain time. Similarly, a relationship type is *unsatisfiable* if it must always have an empty population.

3.2.4.4 Nonredundancy

Two relationship types are redundant if they must always have the same population. A schema should not include redundant relationship types. For example, a schema could include the following types:

Manages (*boss*:*Employee*, *subordinate*:*Employee*)
Supervises (*supervisor*:*Employee*, *Employee*)

If, using the logical representation, the following property must hold in the information base,

$$Manages(b,s) \leftrightarrow Supervises(b,s)$$

then *Manages* and *Supervises* are redundant. One of them must be removed from the schema. Redundancy must not be confused with the fact that a given relationship type may have several alternative names (synonyms).

3.3 Attributes

Besides relationship types, most conceptual models contain the concept of an attribute of an entity type. Attributes are not strictly needed at a concep-

tual level,[3] and they are very similar to binary relationship types. Thus it is not clear whether attributes should be used or when.

In a binary relationship type there are two participants, each of which is an entity type playing a role in that type. The two participants must be considered as "colleagues" in the relationship type, because they perform the same function, and neither of them is subordinated to the other. This can be illustrated by a type such as *Reads* (*reader*:*Person*, *reading*:*Book*): a person cannot read without a text, nor can a book be read without a reader. The order of the participants in the schema does not imply a relationship of priority or subordination between them.

However, there are some relationship types in which users and designers may consider a participant as a "characteristic" of the other. For example, in the case of *HasBalance* (*Account*, *balance*:*Money*) someone might argue that the participant *balance* is a characteristic of *account* and is thus subordinate to *Account*. The concept of an attribute allows this (subjective) subordination of one participant to another to be defined.

An *attribute* is a binary relationship type $R(p_1{:}E_1,p_2{:}E_2)$ in which participant p_2 is considered to be a characteristic of E_1, or p_1 a characteristic of E_2. Therefore, an attribute is like a binary relationship, except that users and designers add the interpretation that one participant is a characteristic of the other. Sometimes we say that E_1 has attribute $(p_2{:}E_2)$, that E_2 is the value of the attribute p_2 of E_1, or that $(p_2{:}E_2)$ is an attribute of E_1.

We denote the schema of an attribute using $P(E_1,E_2)$, which must be understood as equivalent to a relationship type $R(E_1,p{:}E_2)$. In the above example, attribute *Balance* (*Account*, *Money*) is equivalent to the relationship type *HasBalance* (*Account*, *balance*:*Money*).

Data types may have attributes too. An attribute of a data type is considered to be an immutable characteristic of its instances (values).

3.3.1 Conceptual Models Based on Attributes

Some conceptual models use attributes instead of relationship types. The rationale, as we shall see in Chap. 6, is that all relationship types can be transformed into binary ones and that attributes are binary relationship types.

For any relationship type $R(p_1{:}E_1,p_2{:}E_2)$ we can define one or two attributes, $P_2(E_1,E_2)$ and/or $P_1(E_2,E_1)$. If we define two, then we must indicate that they correspond to the same relationship type; this can be done by de-

[3] There are conceptual modeling languages that do not use attributes. The most prominent one is ORM (Halpin 2001).

claring that one is the *inverse* of the other. For example, A_1 = *Reader* (*Book*, *Person*) and A_2 = *Reading* (*Person*, *Book*) would be the two attributes that correspond to *Reads* (*reader*:*Person*, *reading*:*Book*). To this we should add that A_1 is the inverse attribute of A_2, and vice versa.

3.3.2 Attribute Pattern Sentence

Linguistically, instances of attributes can also be expressed by grammatical sentences. The pattern sentence of an attribute gives the general structure of those sentences. For example, a pattern sentence of *Balance* (*Account*, *Money*) could be

The balance of account <*Account*> is the money <*Money*>

This produces sentences such as

The balance of account 12345 is the money 30_*euros*.

As we did for relationship types, we can also distinguish here between implicit and explicit pattern sentences. For attributes, there is only one implicit pattern sentence. If we choose the name of the attribute (*P*) sensibly, the implicit pattern sentence may be enough in most cases. The general structure of the implicit pattern sentence of attribute $P(E_1,E_2)$ is

(A | The) *P* of $e_1 < E_1>$ is $e_2 < E_2>$

In this sentence, we use the indefinite article if an instance of E_1 could have several attribute values and the definite article otherwise.

Alternatively, the following structure might be preferable:

$e_2 < E_2>$ is (a | the) *P* of $e_1 < E_1>$

which, applied to the previous example, gives

The money <*Money*> is the balance of account <*Account*>

3.3.3 Representation in UML

UML shows attributes in the middle compartment of the corresponding entity type. Thus, the attribute $P(E_1,E_2)$ is represented by including the expression $p:E_2$ in the middle compartment of E_1. Figure 3.6 shows an entity type *Customer* with three attributes. Textually, we sometimes use the notation E_1::*p* to refer to attribute *p* of E_1.

In UML, attributes may be marked as read-only, using the keyword {*readOnly*} in braces. In our terminology, this keyword corresponds ap-

Fig. 3.6. Representation of attributes in UML

proximately to attributes constant with respect to E_1. In Fig. 3.6, we indicate that the lengths of rivers are constant. We define that an attribute is permanent with respect to E_1 by attaching a constraint stereotyped *permanent* to it. Attributes of data types are always constant, and we may assume that there is a constraint stereotyped *constant* attached to them.

The attributes of a given entity type must have different names. For navigation purposes, the name of an attribute of entity type *E* should not be the same as the role name of any of the participants of the associations in which *E* participates.

When we want to depict a particular entity in an object diagram we can also show the value of its attributes, as illustrated in Fig. 3.7. The text attribute *name* = *value* defines the concrete attribute values of the entity.

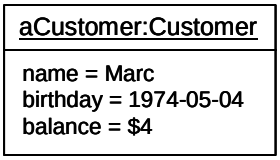

Fig. 3.7. Representation of concrete entities and attribute values in UML

3.3.4 On the Use of Attributes

As we have seen, in the case of UML, when a conceptual model uses both relationship types and attributes their graphical representation is different. The graphical representations of relationship types show the entity types that participate in them. The whole schema shows clearly all the relationship types in which an entity type participates. This representation helps users and designers to understand the schema, especially when it is large.

The graphical representation of attributes, on the other hand, shows them in the context of the entity type of which they are a characteristic.

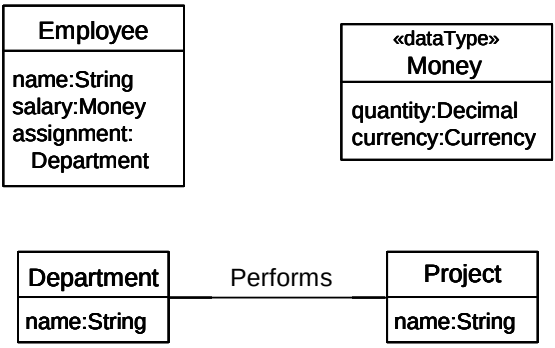

Fig. 3.8. An example of attribute misuse. The attribute *assignment* is best modeled by an association

This representation also helps one to understand the meaning of an entity type.

A problem arises when an entity type *E* is the value of the attribute of another type *E′*, because the diagram does not show a line connecting *E* and *E′*. The relationship between *E* and *E′* is not shown in the same way as the others. Figure 3.8 shows three examples. *String* is the value of the attribute *name* of *Employee*, *Department*, and *Project*. The diagram does not have three lines connecting *Employee*, *Department* and *Project* with *String*. In fact, *String* does not appear in the diagram. *Money* is the value of the attribute *salary* of *Employee*, but the diagram does not show this with a line connecting *Employee* and *Money*. There is a line connecting *Department* and *Project* to show the association *Performs*, but there is no line between *Employee* and *Department* to show the attribute *assignment*. The visual treatment of *String*, *Money*, and *Department* is different from that of *Employee* and *Project*.

This problem can be solved by distinguishing between two kinds of entity type: those that are specific to the domain being modeled and those that are independent of it. The former are entity types that must be defined completely in our schema: users and designers must reach agreement on their meaning. The latter are defined instead in other schemas, and they are only used (or reused) in our schema. In general, data types are domain-independent. In the example in Fig. 3.8, we assume that *String* and *Money* are domain-independent, while *Department* is considered particular to the domain being modeled.

On the basis of this distinction, a guideline for the use of attributes could be that the values of attributes should be entity types defined outside our schema. If we apply this guideline to Fig. 3.8, *name* can be an attribute

in the three entity types and *salary* can be an attribute of *Employee*. However, *assignment* must be defined as an association between *Employee* and *Department*.

A variant of this guideline is to use attributes for data types and associations for ordinary entity types.

This is not a strict guideline, but it does help to make schemas easier to understand. We can define attributes whose values are entity types defined elsewhere, because their meaning must be sought outside our schema. In the above example, it does not seem sensible to assume that our schema must include a definition of what is meant by *String* and *Money*. On the other hand, we should not define attributes whose values are entity types particular to our domain (such as *Department*), because this makes it more difficult to see the relationships between these types and the others.

3.4 Bibliographical Notes

As we noted in the preceding chapter, only a few books describe relationship types in detail without focusing on particular languages. Two notable exceptions are (Kent 1978) and (Tsichritzis and Lochovsky 1982). In contrast, many books addressing a particular language or method give interesting explanations of particular topics, particularly (Nijssen and Halpin 1989, Batini et al. 1992, Martin and Odell 1995, and Halpin 2001). The material presented in this chapter is a synthesis of those texts and other journal and conference papers. One of the most seminal of these was Chen's paper (1976) on the entity–relationship language, which is the basis of this chapter. Wand et al. (1999) provided a thorough analysis of the relationship type concept based on Bunge's ontology.

Chen (1983) gave one of the first analyses of the correspondence between relationship types and English sentences. Rolland and Proix (1992) discussed the correspondence in both senses: from natural language to relationship types (and other schema constructs), and the generation of natural-language sentences from a schema. Hofstede et al. (1997) discussed the uses of verbalizations of fact types in conceptual modeling.

A few languages require a pattern sentence of relationship types to be defined, including OSA (Embley et al. 1992), YSM (Yourdon 1993) and ORM (Halpin 2001). Unary relationship types are allowed in ORM (Halpin 2001) (where they are called unary fact types) and HERM (Thalheim 2000).

The first two properties of the representation of classification (correctness and completeness) are normally implicit and very few works

mention them explicitly. Two exceptions are Greenspan et al. (1994), who called them *property induction constraints*, and Martin and Odell (1995).

The nonempty population and nonredundancy properties are presented in Parsons and Wand (1997). The temporal properties of relationship types were presented by Costal et al. (1997).

The guideline on the use of attributes mentioned in Sect. 3.2.4 is well known in conceptual modeling. It has become part of the UML Reference Manual (Rumbaugh et al. 2005, p. 189)

3.5 Exercises

3.1 Define at least ten relationship types found in the domain of a library that deal with books, authors, the order of the authors of a book, titles, publishers, number of pages, and so on. At least one of them must be *n*-ary. Give the implicit (and if necessary the explicit) pattern sentences. Give the representation of the relationship types in logic and in UML. Also, give the representation in logic and in UML of the relationship "*James Rumbaugh* is the *first* author of the book *The UML Reference Manual*".

3.2 Define a schema with at least ten relationship types found in the domain of persons and their relatives. At least one of them must be *n*-ary. Give the implicit (and if necessary the explicit) pattern sentences. Give the representation of the relationship types in logic and in UML.

3.3 Determine the schema of a relationship type whose instances can be expressed by sentences such as the following:

- Sudha was a General chair of the 25th edition of the ER conference.
- David was a Program chair of the 25th edition of the ER conference.
- Oscar was the Program chair of the 17th edition of the CAiSE conference.
- Colette was the General chair of the 13th edition of the RE conference.

Show the UML representation of this relationship type and of the four instances. Give an explicit pattern sentence.

3.4 Determine the relationship types needed in a conceptual schema for a domain consisting of partially or completely filled-in Sudoku (also known as Number Place) puzzles. You will easily find the rules on the Internet. Each puzzle has a code that identifies it. Assume that the entity types

needed include *Grid*, *Row*, *Column*, *Region*, and *Cell*. Other types may also be necessary. Give the implicit (and if necessary the explicit) pattern sentences. Give the representation of the relationship types in UML.

3.5 The periodic table is a tabular display of the known chemical elements. Consider the data depicted in the standard table (to be found in Wikipedia, for example). Determine entity and relationship types in a schema that are able to represent the data depicted in the standard table. Define the schema in UML. Give the explicit pattern sentences where necessary. Show graphically the instantiation of your schema for the element californium (Cf, atomic number 98).

3.6 The population of a recursive binary relationship type $R(p_1{:}E,p_2{:}E)$ can be constrained, like any other. Give an example of each of the following cases:

1. R is permanent with respect to p_1 and constant with respect to p_2.
2. R is permanent with respect to p_1 and p_2.
3. R is constant with respect to p_1 and p_2.

Indicate in each case whether the population of E is constant, permanent, or unconstrained.