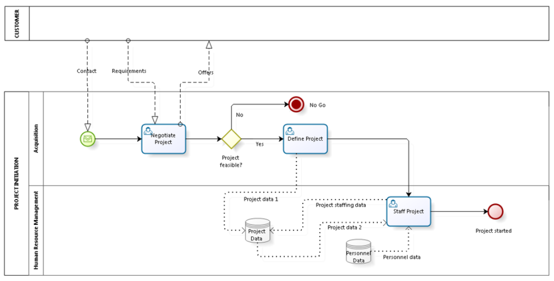
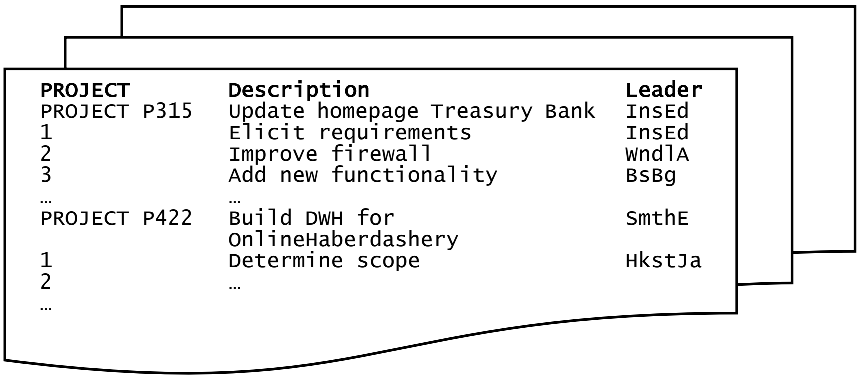
# ISE-DMDD

# Reader Data Modeling and Relational Database Structures



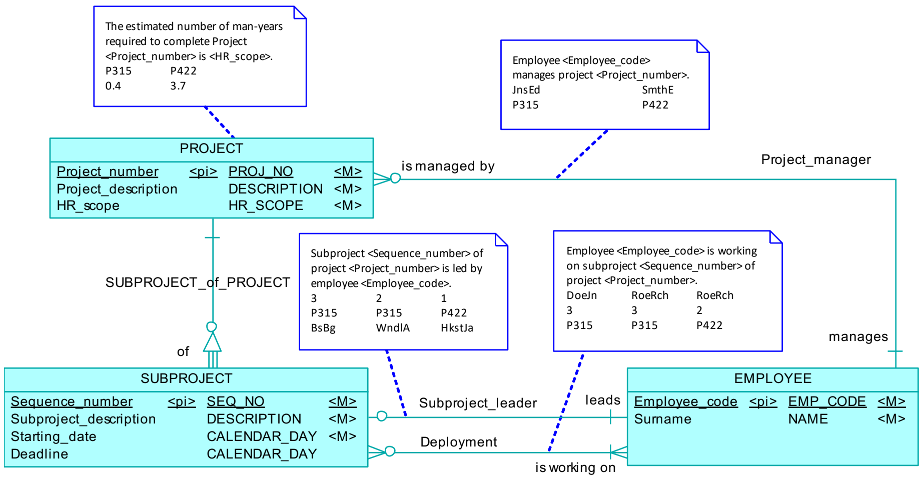


```
PROJECT        Description                     Leader
PROJECT P315   Update homepage Treasury Bank   InsEd
1              Elicit requirements             InsEd
2              Improve firewall                WndlA
3              Add new functionality           BsBg
…              …
PROJECT P422   Build DWH for                   SmthE
               OnlineHaberdashery
1              Determine scope                 HkstJa
2              …
…
```

```
Subproject 3 of project P315 is led by employee BsBg.
   "       2  "    "    P315  "  "  "     "      WndlA.
   "       1  "    "    P422  "  "  "     "      HkstJa.
ET SUBPROJECT                             ET EMPLOYEE
MATCH                                     MATCH

RT Subproject_leader between SUBPROJECT and EMPLOYEE

Subproject     <Sequence_number>     of     project
<Project_number> is led by employee <Employee_code>.
```



The estimated number of man-years required to complete Project <Project_number> is <HR_scope>.

| | |
|---|---|
| P315 | P422 |
| 0.4 | 3.7 |

Employee <Employee_code> manages project <Project_number>.

| | |
|---|---|
| JnsEd | SmthE |
| P315 | P422 |

Subproject <Sequence_number> of project <Project_number> is led by employee <Employee_code>.

| | | |
|---|---|---|
| 3 | 2 | 1 |
| P315 | P315 | P422 |
| BsBg | WndlA | HkstJa |

Employee <Employee_code> is working on subproject <Sequence_number> of project <Project_number>.

| | | |
|---|---|---|
| DoeJn | RoeRch | RoeRch |
| 3 | 3 | 2 |
| P315 | P315 | P422 |

PROJECT
Project_number <pi> PROJ_NO <M>
Project_description DESCRIPTION <M>
HR_scope HR_SCOPE <M>
is managed by
Project_manager
manages
SUBPROJECT_of_PROJECT
of
SUBPROJECT
Sequence_number <pi> SEQ_NO <M>
Subproject_description DESCRIPTION <M>
Starting_date CALENDAR_DAY <M>
Deadline CALENDAR_DAY
Subproject_leader
leads
Deployment
is working on
EMPLOYEE
Employee_code <pi> EMP_CODE <M>
Surname NAME <M>

HAN_UNIVERSITY OF APPLIED SCIENCES

## General information

| | |
|---|---|
| **Title** | ISE-DMDD Reader Data Modeling and Relational Database Structures |
| **Academic year** | 2019-2020 |
| **Training Programs** | Information Science, profile Software Development (SD) |
| | Information Science, profile Enterprise Software Solutions (ESS) |
| | Information Science, profile Information Management & Consultancy (IMC) |
| **Author** | Jan Pieter Zwart, January 2019 |

The English texts were written by Jan-Pieter Zwart, who also made all the English figures
The exercises are based on original texts by Marco Engelbart.

## Table of Contents

1 Prerequisites ..... 1

2 Normal Forms and Normalization ..... 3

2.1 Introduction ..... 3

2.1.1 Redundancy ..... 3

2.1.2 Functional dependency ..... 5

2.1.3 Normal forms and normalization ..... 5

2.2 Orders example: First Normal Form (1NF) ..... 7

2.3 Functional Dependency ..... 10

2.3.1 Connection between facts and functional dependencies ..... 12

2.4 Orders example: Second Normal Form (2NF) ..... 13

2.5 Orders example: Third Normal Form (3NF) ..... 15

2.6 Boyce-Codd Normal Form (BCNF) ..... 17

2.6.1 Nearest shops ..... 18

2.6.2 Dutch postcode ..... 22

2.7 Orders example: Boyce-Codd Normal Form (BCNF) ..... 26

2.8 Higher Normal Forms ..... 28

3 Introduction to ERM and Fact Oriented ERM (FO-ERM) ..... 30

3.1 Concepts in ERM ..... 30

3.1.1 Entity Type (ET), and entity ..... 30

3.1.2 Attribute (Att) ..... 30

3.1.3 Domain ..... 30

3.1.4 Mandatory constraint ..... 31

3.1.5 ERM Rule 1: Every ET must have a *primary identifier* ..... 31

3.1.6 Strong ET: needs only its own attributes for its own <pi> ..... 31

3.1.7 Weak ET: needs <pi>s from other ETs for its own <pi> ..... 31

3.1.8 ERM Rule 2: No ‘foreign-key’ attributes are allowed. ..... 32

3.1.9 Relationship type ..... 32

3.1.10 Cardinalities ..... 33

3.1.11 Roles ..... 33

3.2 FO-ERM: adding fact types and example populations ..... 34

3.2.1 Facts, and fact types versus ETs, Atts and RTs. ..... 34

3.2.2 Instance level versus type level ..... 34

3.2.3 Fact types and example populations clarify the meaning of the model ..... 34

4 Drawing up an ERM Diagram ........ 36
4.1 Fact Oriented Modeling ........ 36
4.1.1 Verbalizations of Concrete Examples of Facts ........ 36
4.1.2 A Good Procedure for How to Model ........ 37
4.1.3 Focus on Elementary Facts ........ 37
4.1.4 Procedure to draw up an ERM diagram in a Fact Oriented Modeling way ........ 38
4.2 Collecting: finding or drawing up concrete examples of facts ........ 39
4.3 Verbalizing ........ 40
4.3.1 Express concrete facts that can be highlighted ........ 42
4.3.2 Express exactly one fact: no clusters of facts, no incomplete facts ........ 43
4.3.3 Use sentences that do not depend on other sentences ........ 46
4.3.4 Use sentences that make the meaning of the facts as clear as possible ........ 47
4.4 Sorting and analyzing fact expressions ........ 48
4.4.1 Components, segments and predicates ........ 48
4.4.2 Procedure for sorting and analyzing fact expressions and building an ERM diagram 49
4.4.3 Example used in section 4.4.4 ........ 53
4.4.4 Sorting and analyzing the fact expressions ........ 56
4.4.5 Examples of fact types with only one segment ........ 65

5 Deriving a PDM from a CDM ........ 68
5.1 Step A: Process many-to-many relationship types (n-m RTs) ........ 69
5.1.1 Substep 1: Replace the RT with an ET ........ 69
5.1.2 Substep 2: Add two new 1-n RTs ........ 69
5.1.3 Substep 3: Make the new ET dependent on both sides ........ 69
5.1.4 Substep 4: Determine the minimal cardinalities at the side of R ........ 69
5.1.5 Substep 5: Give the new dependent RTs meaningful names ........ 70
5.1.6 Substep 6: Add roles to the new dependent RTs ........ 71
5.2 Step B: Process one-to-one relationship types (1-1 RTs) ........ 72
5.2.1 Dominance in 1-1 RTs ........ 72
5.2.2 Substep 7: Choose the dominant side in 1-1 RTs ........ 74
5.3 Step C: Process entity types (ETs) and attributes (Atts) ........ 75
5.3.1 Substep 8: Entity types become tables, attributes become columns ........ 75
5.4 Step D: Process one-to many relationship types (1-n RTs) ........ 75
5.4.1 Substep 9: 1-n RTs become references ........ 75
5.4.2 Substep 10: Add foreign key columns ........ 75
5.4.3 Substep 11: Process inheritance links of subtypes ........ 76
5.4.4 Substep 12: Add join expressions to all references ........ 76
5.4.5 Substep 13: Add cardinalities to all references ........ 76

5.5 Step E: Process constraints ... 78
5.5.1 Substep 14: translate constraints into integrity rules ... 78
5.6 Step F: Add predicates and example populations where convenient ... 78
5.6.1 Substep 15: Add predicates and example populations where convenient ... 78
6 Subtypes ... 79
6.1 Introducing subtypes and supertypes ... 79
6.1.1 Subtype and supertype ... 79
6.1.2 Inheritance link ... 82
6.1.3 Subtype defining fact type ... 83
6.2 Derivable subtypes ... 83
6.3 Declarative subtypes ... 84
6.4 Deriving a PDM from a CDM with subtypes ... 86
6.4.1 Setting the options in the Inheritance Properties Generation screen ... 86
6.4.2 Deriving a table for the supertype only ... 88
6.4.3 Substep 11 in Deriving a PDM from a CDM ... 91
6.5 Other possibilities to generate a PDM from a CDM with subtypes ... 92
6.6 Analyzing fact expressions with subtypes ... 94
7 References ... 95
Appendix A: Detailed procedure for analyzing fact expressions ... 96

# 1 Prerequisites

To study the material in this reader successfully, you must be familiar with the Relational Model for database structures. In particular you must have mastered the following concepts, which are briefly summarized below. If anything here is unclear, please refresh your knowledge using any source on relational databases.

**Table**

Synonym: Relation (now obsolete). The Relational Model was invented by E. F. Codd in the sixties, who was a mathematician and regarded tables as mathematical constructs called 'relations'. Not to be confused with the Relationship Types in ERM.

**Column**

Vertical part of a table (synonym: attribute) containing values of one particular kind; see also **Domain**.

**Row**

Horizontal part of a table (synonyms: tuple, record), containing values that belong together. Each row represents the properties of one 'item' or 'entity' in the table.

**Cell**

A single compartment (synonym: field) in a table, containing exactly one value.

**Domain**

To each column belongs a domain: a set of values appropriate for this attribute. A domain can be seen as a source from which values of a particular kind can be drawn (SURNAME, CALENDAR DAY, PERCENTAGE, …). Different columns can have the same domain. A data type is assigned to each domain (char(20), date, smallint, …). Different domains can have the same data type.

**Candidate Key**

Column (or smallest set of columns) with the property that the value (or combination of values) must be unique in each row of the table. There is always at least one candidate key.

**Primary Key**

One of the candidate keys, chosen to be the standard way of identifying the entities (or items) in the table.

**Alternative Key**

A candidate key that is not the primary key (if any).

**Natural Key**

A key that contains information about the entity it identifies, like a surname or a purchase date, often conveniently used by domain experts. See also **Surrogate Key**.

**Surrogate Key**

A key that contains no information about the entity it identifies, like a sequence number or a meaningless code, often used in information systems because it never changes. See also **Natural Key**.

**Foreign Key**

A column (or set of columns) that is a copy of the primary key from another table (or even from the same table). The values in the rows of the foreign key must also exist in the primary key of the other table. See also **Reference**.

**Reference**

A constraint that runs from a column (or a set of columns) in one table to a column (or set of columns) in another table. It specifies that the values in the from-side of the reference must also exist in (i.e.: must 'come from' or 'be a part of') the values in the to-side of the reference. A reference is usually shown graphically as an arrow (from the tail to the head). A foreign key is always at the from-side of a reference, the corresponding primary key at the to-side.

**Foreign Key Reference**

The reference between a foreign key (from-side) and its corresponding primary key (to-side).

**Parent Table**

A table at the to-side of a foreign key reference.

**Child Table**

A table with a foreign key, therefore at the from-side of a foreign key reference.

**Mandatory Child Table**

The values in a foreign key must always be a part of the values in the corresponding primary key. However sometimes the foreign key must contain all the values in the corresponding primary key. If that is the case, then the child table is a mandatory child table: every parent value (in the primary key) must have a child value (in the foreign key).

**Mandatory Child Reference**

This can be seen as an 'ordinary' foreign key reference, extended with another reference in the opposite direction. Together these references result in the two populations being equal. The 'ordinary' foreign key reference (from child to parent) can be easily implemented in modern database management systems. The one in the opposite direction however will usually have to be programmed seperately.

# 2 Normal Forms and Normalization

## 2.1 Introduction

This section starts with **redundancy** as a main problem in bad database designs. Informally, the concept of a **functional dependency** is introduced. The connection with **normal forms** and **normalization** is discussed.

### 2.1.1 Redundancy

A database is for storing and retrieving **facts** that are important to the people working in an organization or business. The most reliable way to do this, is if each fact is stored only once. Otherwise, if one fact is stored several times in different places, data integrity problems arise. If a fact is changed (updated or deleted), how can we be sure this happens in all the places the fact is stored in? If a new fact is added, how can we be sure it is added in all the right places? Experience shows that in practice **data pollution** (two facts that contradict one another, facts missing where they should be, etcetera) arises as soon as facts are stored more than once in a database. If a fact can be stored more than once, the database is said to suffer from **redundancy**. 'Redundant' is the same as 'superfluous', 'unnecessary' or 'surplus': you just don't need to store the same fact twice because once is enough. Therefore a major goal of designing a relational database has always been to avoid redundancy.

Avoiding redundancy is not so easy. One single fact is usually composed of values in several different table columns. So when does a table have redundancy? Just looking at duplicate values in columns doesn't help much, as the following example shows. The table Smartboards in figure 2.1 is used in a large school. It shows what type of smartboard is present in which classroom, and what the maximum number of simultaneous touchpoints is that the board can handle. Is there redundancy in this table?

**Table Smartboards**

| Floor Number | Room Number | Smartboard Type | Number of Touchpoints |
|---|---|---|---|
| 1 | 21 | XM | 10 |
| 1 | 23 | S75 | 16 |
| 2 | 23 | XM | 10 |
| 2 | 21 | XM | 10 |
| 3 | 12 | S86 | 16 |

**Figure 2.1 Table Smartboards**

Many values appear more than once, but that doesn't necessarily mean that whole facts are stored more than once. So we need to know what the facts are. Below, two different cases will be considered. These cases concern two different situations, with the same table Smartboards, but with different facts (based on different rules that apply to the data in the table in the two cases).

**Case 1**

An information modeler and a domain expert (a technical assistant or teacher who works in the school) draw up the following verbalizations of all the complete facts in the upper two rows of the table. These verbalizations express the meaning of exactly one complete fact each.

```
Classroom 1.21 contains a smartboard of type XM.
A smartboard of type XM can handle at most 10 touchpoints simultaneously.
Classroom 1.23 contains a smartboard of type S75.
A smartboard of type S75 can handle at most 16 touchpoints simultaneously.
```

**Figure 2.2 Verbalizations of facts in table Smartboards, case 1**

Now it is clear that the table indeed suffers from redundancy: it contains three copies of the fact:
`A smartboard of type XM can handle at most 10 touchpoints simultaneously.`
This is highlighted in figure 2.3.

**Table Smartboards**

| Floor Number | Room Number | Smartboard Type | Number of Touchpoints |
|---|---|---|---|
| 1 | 21 | ***XM*** | ***10*** |
| 1 | 23 | S75 | 16 |
| 2 | 23 | ***XM*** | ***10*** |
| 2 | 21 | ***XM*** | ***10*** |
| 3 | 12 | S86 | 16 |

**Figure 2.3 Table Smartboards with redundant facts shown**

A better table design would be the one given in figure 2.4, which has no redundancy anymore.

**Table Smartboard_in_Room**

| Floor Number | Room Number | Smartboard Type |
|---|---|---|
| 1 | 21 | XM |
| 1 | 23 | S75 |
| 2 | 23 | XM |
| 2 | 21 | XM |
| 3 | 12 | S86 |

**Table Smartboard_Touchpoints**

| Smartboard Type | Number of Touchpoints |
|---|---|
| XM | 10 |
| S75 | 16 |
| S86 | 16 |

**Figure 2.4 Tables without redundancy**

## Case 2

An information modeler and a domain expert (a a technical assistant or teacher who works in the school) draw up the following verbalizations of all the complete facts in the upper two rows of the table. These verbalizations express the meaning of exactly one complete fact each. Note the difference with case 1 for the second and fourth fact: the number of touchpoints does not depend on the smartboard type here. Apparently, even smartboards of the same type can have a different number of touchpoints (even if the examples in the table do not show this, but see also figure 2.6).

```
Classroom 1.21 contains a smartboard of type XM.
In classroom 1.21 the smartboard can handle at most 10 touchpoints simultaneously.
Classroom 1.23 contains a smartboard of type S75.
In classroom 1.23 the smartboard can handle at most 16 touchpoints simultaneously.
```

**Figure 2.5 Verbalizations of facts in table Smartboards, case 2**

If the table Smartboards in figure 2.1 is used to store the facts in figure 2.5, instead of the facts in figure 2.2, then it does not suffer from redundancy at all. What is the difference?

**Difference between cases 1 and 2**

To show the difference, in figure 2.6 a new tuple has been added to table Smartboards. This tuple is not allowed in case 1, but it is valid in case 2. The number of touchpoints depends on the smartboard type in case 1, but not in case 2, therefore it must always be the same for a smartboard type in case 1, but it can be different in case 2. The verbalizations make this implicitly clear, and the domain expert will confirm it when asked.

**Table Smartboards case 1**

| Floor Number | Room Number | Smartboard Type | Number of Touchpoints |
|---|---|---|---|
| 1 | 21 | XM | 10 |
| 1 | 23 | S75 | 16 |
| 2 | 23 | XM | 10 |
| 2 | 21 | XM | 10 |
| 3 | 12 | S86 | 16 |
| 4 | 11 | ~~..XM..~~ | ~~..12..~~ |

**Table Smartboards case 2**

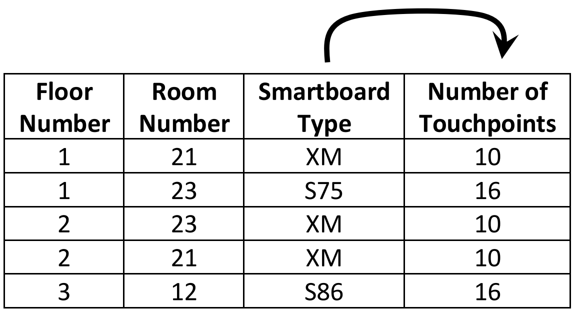
| Floor Number | Room Number | Smartboard Type | Number of Touchpoints |
|---|---|---|---|
| 1 | 21 | XM | 10 |
| 1 | 23 | S75 | 16 |
| 2 | 23 | XM | 10 |
| 2 | 21 | XM | 10 |
| 3 | 12 | S86 | 16 |
| 4 | 11 | XM | 12 |

**Figure 2.6 Table Smartboards with extra tuple not allowed in case 1 but allowed in case 2**

## 2.1.2 Functional dependency

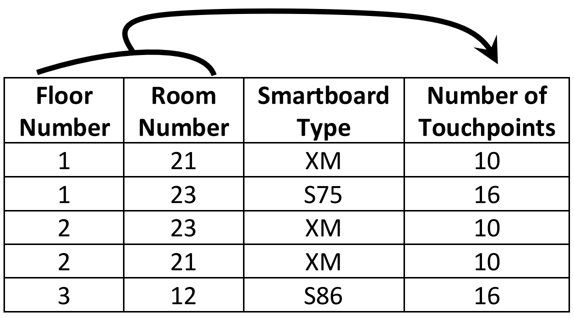
The difference between cases 1 and 2 above is caused by a difference in the dependency of column Number of Touchpoints. In case 1, the values in the column depend on the smartboard type: there can be only one number of touchpoints for each smartboard type. In case 2, the number of touchpoints does not depend on the type of smartboard, but on the particular smartboard in the classroom. The smartboard in room 1.21 has 10 touchpoints, whereas the smartboard in room 4.11 has 12, though both are of type XM. This difference in dependency can be expressed by drawing the dependency as an arrow, as is shown in figure 2.7. Section 2.3 below discusses these dependencies further; they are formally called **functional dependencies**. The head of the arrow points to the dependent column, the tail comes from all the columns that the head is dependent on.

**Table Smartboards case 1**



| Floor Number | Room Number | Smartboard Type | Number of Touchpoints |
|---|---|---|---|
| 1 | 21 | XM | 10 |
| 1 | 23 | S75 | 16 |
| 2 | 23 | XM | 10 |
| 2 | 21 | XM | 10 |
| 3 | 12 | S86 | 16 |

**Table Smartboards case 2**



| Floor Number | Room Number | Smartboard Type | Number of Touchpoints |
|---|---|---|---|
| 1 | 21 | XM | 10 |
| 1 | 23 | S75 | 16 |
| 2 | 23 | XM | 10 |
| 2 | 21 | XM | 10 |
| 3 | 12 | S86 | 16 |

**Figure 2.7 Table Smartboards with functional dependency in case 1 and case 2**

## 2.1.3 Normal forms and normalization

E.F. Codd, who invented the Relational Model for databases in the sixties, realized he could use functional dependencies (explained in detail in section 2.3 below) to check the quality of database designs. He defined several kinds of redundancy and showed how you could remove them by looking at the functional dependencies. He defined several **normal forms** for tables. A normal form is a set of rules a table should comply with to avoid some of these kinds of redundancy. The higher the normal form is, the stricter these rules are and the fewer the kinds of redundancy that still exist. Based on the concept of functional dependency (explained in detail in section 2.3 below), Codd described the First Normal Form (1NF), Second Normal Form (2NF) and Third Normal Form (3NF) in 1970. Together with his colleague R.F. Boyce he also defined the Boyce-Codd Normal Form (1974), an elegant normal form that exploits functional dependencies maximally.

Later, many other normal forms were defined by several other people (4NF, 5NF, 6NF, DKNF, ONF, etc.), which all use other concepts than functional dependencies. These will only briefly be discussed in section 2.8.

The process of improving table structures so they will comply with a higher normal form is called **normalizing**, or **normalization**. The reverse process of introducing redundancy and so lowering the normal form, is called **denormalizing**, or **denormalization.**

As a technique to *design* good table structures, normalization is now quite outdated, since far better data modeling techniques exist at present (based on facts rather than on functional dependencies), like Entity-Relationship Modeling (ERM), and Fact Oriented Modeling (FOM) techniques like Object Role Modeling (ORM) and Fully Communication Oriented Information Modeling (FCO-IM). Still, it is very useful if database designers are at least familiar with the concepts of functional dependency and normal forms. Terms like ‘3NF’ are still widely being used today, and a good information modeler will have great benefits from being able to see at once which normal form a given table is in. Many sub-optimal database structures exist in practice, perhaps as legacy from a poor design in the past, or perhaps because denormalization was intentionally done to improve performance (with the resulting redundancy hopefully controlled by the system), or for some other reason.

Therefore normal forms based on functional dependencies are discussed in depth in this chapter, even though a different fact-based method to design a database is presented in chapters 4 - 6 of this reader. Instead of first making a poor database design and then improving this afterwards step-by-step, as normalizing does, these other techniques draw up a good conceptual information model, and then derive an optimal relational database structure automatically from this model in one go.

Another serious shortcoming of normalizing is that it considers attributes in isolation, and does not take *combinations of attributes that make up elementary fact types* into account. Many ‘problems’ in table structures simply do not arise when complete facts are considered to start with (see also sections 2.6 – 2.8 on the Boyce-Codd Normal Form and higher normal forms). Therefore I have included verbalizations of facts where this helps in making structural problems more clear.

## 2.2 Orders example: First Normal Form (1NF)

Consider the following example table T1 in figure 2.8. It contains facts concerning customers who place orders for articles. Columns 'Order no.' and 'Date' contain the order number and order date, the name and address data are those of the customer who places the order, 'Article no.' and 'Art. name' contain the number and name of an ordered article, 'PPP' the price per piece of an article, and 'Number' how many of that particular article were ordered.

**Table T1**

| Order no. | Date | Cust- omer | Name | Street | House no. | Postcode | Town | Article no. | Art. name | PPP | Num- ber |
|---|---|---|---|---|---|---|---|---|---|---|---|
| O1 | 18/02/02 | C1 | Jones | Forel | 2 | 5345 DK | Oss | A1<br>A2<br>A6 | pen<br>pen<br>pad | 3.50<br>9.00<br>14.00 | 3<br>2<br>4 |
| O2 | 25/02/02 | C1 | Jones | Forel | 2 | 5345 DK | Oss | A4 | box | 25.00 | 1 |
| O3 | 25/02/02 | C2 | Smith | Forel | 14 | 5345 DK | Oss | A3<br>A5 | clip<br>box | 3.50<br>12.99 | 1<br>3 |
| O4 | 26/02/02 | C3 | Smith | Kist | 2 | 6661 ZH | Elst | A3<br>A5<br>A6 | clip<br>box<br>pad | 3.50<br>12.99<br>14.00 | 2<br>1<br>7 |

**Figure 2.8 Table T1: unnormalized**

Note: several articles can be ordered on one order. In table T1, there is a **repeating group** for all articles ordered on one order. The group of columns from 'Article no.' up to 'Number' contains **more than one value** for each value in columns 'Order no.' up to 'Town'. In the Relational Model however, there can be only one value in each cell of a table: each cell must contain an *atomic* (= single, non-compound, indivisible) value. Table T1 is called *unnormalized*, because it contains several values per cell in the group of columns 'Article no.' up to 'Number'.

**Definition 1**

**A table is in First Normal Form (1NF) if every cell contains an atomic value.**

It is possible to bring table T1 into 1NF as is shown in Table T2 in figure 2.9. The PK is on two columns. But this would clearly be a wrong way of doing it. In this way, a lot of extra redundancy would be introduced! The facts in columns 'Order no.' up to 'Town' are repeated several times in table T2. This would lead to serious data pollution in no time, in all possible changes to the population (additions: typos in values that should be the same; updates/deletes: incomplete handling; etc.).

**Table T2**

| Order no. ← | Date | Cust-omer | Name | Street | House no. | Postcode | Town | Article no. → | Art. name | PPP | Num-ber |
|---|---|---|---|---|---|---|---|---|---|---|---|
| O1 | 18/02/02 | C1 | Jones | Forel | 2 | 5345 DK | Oss | A1 | pen | 3.50 | 3 |
| O1 | 18/02/02 | C1 | Jones | Forel | 2 | 5345 DK | Oss | A2 | pen | 9.00 | 2 |
| O1 | 18/02/02 | C1 | Jones | Forel | 2 | 5345 DK | Oss | A6 | pad | 14.00 | 4 |
| O2 | 25/02/02 | C1 | Jones | Forel | 2 | 5345 DK | Oss | A4 | box | 25.00 | 1 |
| O3 | 25/02/02 | C2 | Smith | Forel | 14 | 5345 DK | Oss | A3 | clip | 3.50 | 1 |
| O3 | 25/02/02 | C2 | Smith | Forel | 14 | 5345 DK | Oss | A5 | box | 12.99 | 3 |
| O4 | 26/02/02 | C3 | Smith | Kist | 2 | 6661 ZH | Elst | A3 | clip | 3.50 | 2 |
| O4 | 26/02/02 | C3 | Smith | Kist | 2 | 6661 ZH | Elst | A5 | box | 12.99 | 1 |
| O4 | 26/02/02 | C3 | Smith | Kist | 2 | 6661 ZH | Elst | A6 | pad | 14.00 | 7 |

**Figure 2.9 Wrong way to bring table T1 into 1NF**

There is a better way to bring table T1 into 1NF:

- Split off the repeating group into a table of its own.
- Determine all the candidate keys of the new tables and choose a primary key (PK).
  **Note**: add a foreign key (FK) to the repeating group table to preserve the connection

The following two points should also be done, though they are not usually stated in texts about normalizing. However, they are necessary perform, to ensure that the resulting tables satisfy the same integrity rules that the original tables did. Integrity rules are also a vital part of a relational database structure. It is alas an aspect that the normalization process unjustly neglects.

- Add a foreign key reference from the FK in the new child table to the PK of the parent table.
- Add a mandatory child reference in the opposite direction if necessary.

The last point above can only be determined by interviewing a domain expert.

Figure 2.10 shows the result of applying the steps above to table T1. Tables T3 and T4 are now both in 1NF without introducing extra redundancy. All the facts from from T1 are now in T3 and T4. In both tables, there is only one candidate key, so it is automatically also the PK.

**Table T3**

| **Order no.** ⟷ | **Date** | **Cust-omer** | **Name** | **Street** | **House no.** | **Postcode** | **Town** |
|---|---|---|---|---|---|---|---|
| O1 | 18/02/02 | C1 | Jones | Forel | 2 | 5345 DK | Oss |
| O2 | 25/02/02 | C1 | Jones | Forel | 2 | 5345 DK | Oss |
| O3 | 25/02/02 | C2 | Smith | Forel | 14 | 5345 DK | Oss |
| O4 | 26/02/02 | C3 | Smith | Kist | 2 | 6661 ZH | Elst |

**Table T4**

| **Order no.** ⟵ | **Article no.** ⟶ | **Art. name** | **PPP** | **Num-ber** |
|---|---|---|---|---|
| O1 | A1 | pen | 3.50 | 3 |
| O1 | A2 | pen | 9.00 | 2 |
| O1 | A6 | pad | 14.00 | 4 |
| O2 | A4 | box | 25.00 | 1 |
| O3 | A3 | clip | 3.50 | 1 |
| O3 | A5 | box | 12.99 | 3 |
| O4 | A3 | clip | 3.50 | 2 |
| O4 | A5 | box | 12.99 | 1 |
| O4 | A6 | pad | 14.00 | 7 |

**FKref1:** T4(Order no.) → T3(Order no.)
**MCref1:** T3(Order no.) → T4(Order no.)
**Note:** The domain expert assured there can be no empty orders, so there must be a mandatory child reference that goes in the opposite direction of FKref1.
Whereas FKref1 can be implemented declaratively in almost all relational database management systems, MCref1 must be coded by hand (using stored procedures or so).

**Figure 2.10 Correct way to bring table T1 into 1NF: two tables, with references**

Although tables T3 and T4 are now in 1NF, the design of these tables can still be much improved. Can you spot the redundancy present in both tables? In normalization, **functional dependencies** are used to spot and handle redundancy. Therefore this concept of functional dependency will first be introduced in the next section.

## 2.3 Functional Dependency

In tables T3 and T4 in figure 2.10, the following properties (integrity rules) apply:

- An order **has only one** order date.
- A customer **has only one** name.
- On an order an article **has only one** ordered amount.

Another way of saying the same thing is:

- The order date **depends** on the order.
- The name **depends** on the customer.
- The number ordered **depends** on the order and article.

A formal way of expressing the same thing is:

- Column ‘Date’ is **functionally dependent** on column ‘Order no.’.
- Column ‘Name’ is **functionally dependent** on column ‘Customer’.
- Column ‘Number’ is **functionally dependent** on columns (‘Order no.’ + ‘Article no’).

Above, three **functional dependencies** between table columns are stated. Functional dependencies can be used to pinpoint where and why redundancy occurs in a table. Here is the formal definition in wo parts (for easy reading, although the first part is actually a special case of the second part):

**Definition 2**

- **In a given table, a column P is functionally dependent on another column A,**
  **if <u>at most one</u> value $p_0$ of P can belong to each value $a_0$ in A.**
  **Notation: A → P.**
- **In a given table, a column P is functionally dependent on a set of other columns (A, B, …),**
  **if <u>at most one</u> value $p_0$ of P can belong to each set of values ($a_0$, $b_0$, …) in (A, B, …).**
  **Notation: (A, B, …) → P.**

Please verify that the three functional dependencies stated above satisfy definition 2:

- At most one value of ‘Date’ can belong to each value of ‘Order no.’:
  **‘Order no.’ → ‘Date’.**
  Note that the reverse is not true: several different values of ‘Order no.’ (O2, O3) can belong to one value of ‘Date’ (25/02/02).
- At most one value of ‘Name’ can belong to each value of ‘Customer’:
  **‘Customer’ → ‘Name’**.
- At most one value of ‘Number’ can belong to each combination of ‘Order no.’ and ‘Article no.’:
  **(‘Order no.’ + ‘Article no.’) → ‘Number’**.

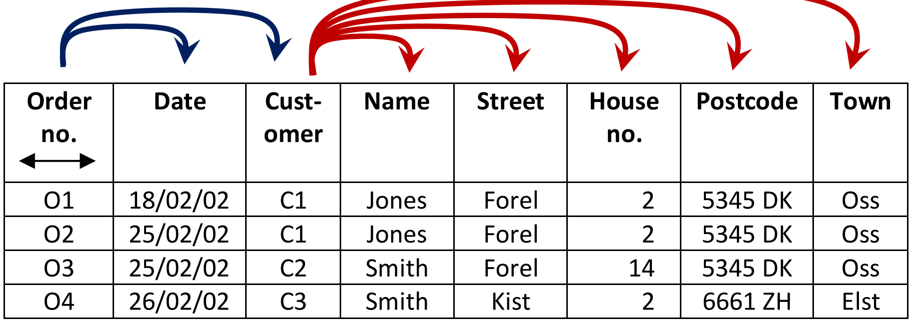
**Note:** if you find the following dependencies: P → Q and Q → R, then these imply P → R as well. Such a **transitive dependency** P → R is not listed, since it can be derived from the other two.

In short:

$$\begin{tikzcd} P \arrow[r, bend left] \arrow[rr, bend right, "\times" description] & Q \arrow[r, bend left] & R \end{tikzcd}$$

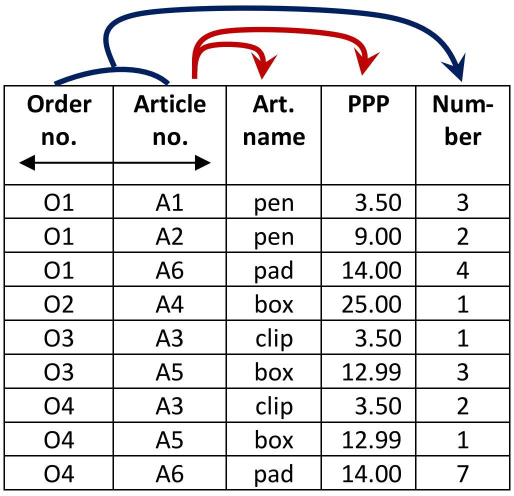
In figure 2.11, T3 and T4 are shown again, with all their functional dependencies.

**Table T3**



| Order no. | Date | Cust-omer | Name | Street | House no. | Postcode | Town |
|---|---|---|---|---|---|---|---|
| O1 | 18/02/02 | C1 | Jones | Forel | 2 | 5345 DK | Oss |
| O2 | 25/02/02 | C1 | Jones | Forel | 2 | 5345 DK | Oss |
| O3 | 25/02/02 | C2 | Smith | Forel | 14 | 5345 DK | Oss |
| O4 | 26/02/02 | C3 | Smith | Kist | 2 | 6661 ZH | Elst |

**Table T4**



| Order no. | Article no. | Art. name | PPP | Num-ber |
|---|---|---|---|---|
| O1 | A1 | pen | 3.50 | 3 |
| O1 | A2 | pen | 9.00 | 2 |
| O1 | A6 | pad | 14.00 | 4 |
| O2 | A4 | box | 25.00 | 1 |
| O3 | A3 | clip | 3.50 | 1 |
| O3 | A5 | box | 12.99 | 3 |
| O4 | A3 | clip | 3.50 | 2 |
| O4 | A5 | box | 12.99 | 1 |
| O4 | A6 | pad | 14.00 | 7 |

**FKref1:** T4(Order no.) → T3(Order no.)
**MCref1:** T3(Order no.) → T4(Order no.)

Transitive dependencies in T3
like 'Order no.' → 'Name' etc.
have not been drawn.

**Figure 2.11 T3 and T4 with functional dependencies**

In sections 2.2 through 2.5, the structure of the Dutch postcode is ignored. This structure actually gives rise to a few other functional dependencies, which will be discussed in sections 2.6.2 and 2.7.

In general, it is difficult to find functional dependencies, and I have not yet seen any method to determine them systematically. Here are a few useful guidelines. You can apply them to T3 and T4 to check the given funcstional dependencies in figure 2.11.

**1** Find all candidate keys (CKs) and choose one as the primary key (PK).
Every column that belongs to a CK (this includes the PK!) is called a **key column**.
Every column that belongs to the PK is called a **PK-column**.
Every other column is called a **non-key column**.

Consider only the non-key columns (only those that do **not** belong to one of the **candidate** keys).

**2** For each non-key column:
determine all functional dependencies between this column and the PK-columns.

**3** For each non-key column:
determine all functional dependencies between this column and other non-key columns.

**4** Delete all transitive dependencies.

### 2.3.1 Connection between facts and functional dependencies

Every functional dependency is also a **fact type** (see section 3.2.1). If you take the values from one tuple in a table from all the columns that are involved in one functional dependency (head and tail) you get exactly one complete fact. So because redundancy is about storing complete facts more than once, considering functional dependencies is a way to deal with some kinds of redundancy.

Although every functional dependency is a fact type, there are also other fact types that are not functional dependencies. This is one of the reasons that it is better to design databases using facts instead of dependencies: with facts you get the whole picture and not just the part that corresponds to functional dependencies only (see also section 2.8 on higher normal forms).

## 2.4 Orders example: Second Normal Form (2NF)

Let us first consider table T4 in figure 2.11. There are still problems with this table:

- Redundancy: facts about the name and PPP of an article are stored repeatedly.
- Loss of information: if order O2 is deleted, all info about article A4 disappears as well (if A4 is not ordered on any other order in the database)!

These problems occur because columns 'Art. name' and 'PPP' are functionally dependent on only a part of the primary key of table T4. Because the same article can appear several times in the compound PK table T4, everything that depends only on 'Article no.' is repeated as well. Since every functional dependency corresponds with a fact type (see section 2.3.1 above), this causes facts to be stored more than once. The Second Normal Form therefore forbids dependencies on only a part of the PK.

**Definition 3:**

**A key column is a column that belongs to any of the candidate keys (including the PK).**
**A non-key column is a column that is NOT a key column.**

**Definition 4:**

**A table is in Second Normal Form (2NF)**
**IF it satisfies both the following conditions:**

- **The table is in 1NF.**
- **All non-key attributes are functionally dependent on the WHOLE primary key.**

Note the distinct use in definitions 3 and 4 of the terms 'candidate keys' and 'primary key'.

Table T3 in figure 2.11 satisfies the conditions for 2NF: it is in 1NF, and the PK is only one column wide, so there can be no dependencies on only a part of it. Columns 'Name' … 'Town' also depend on the (whole) PK, via the intermediate column 'Customer' (transitive dependency).

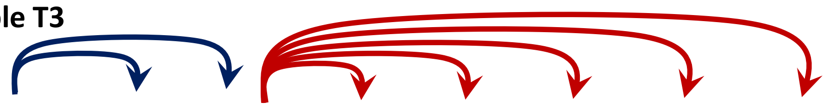
Clearly, table T4 in figure 2.11 does not satisfy the conditions of 2NF. This can be fixed as follows:

- Make a separate table (or tables) for the functional dependencies with a tail on only a part of the PK (a separate table for each different tail).
- The tail of these dependencies will be the PK of the new table.
- The part of the PK in the old table where the tail was becomes a foreign key (FK).
- Add a foreign key reference from the FK in the child table to the PK of the new parent table.
- Add a mandatory child reference in the opposite direction if necessary

The last point above can only be determined by interviewing a domain expert.

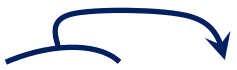
Figure 2.12 shows the result of bringing the tables in 2NF: please check they are all indeed in 2NF. There is no new mandatory child reference pointing the other way from FKref2: the domain expert confirmed there can be articles that have not been ordered yet (or anymore).

**Table T3**



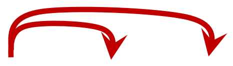
| Order no. ↔ | Date | Cust-omer | Name | Street | House no. | Postcode | Town |
|---|---|---|---|---|---|---|---|
| O1 | 18/02/02 | C1 | Jones | Forel | 2 | 5345 DK | Oss |
| O2 | 25/02/02 | C1 | Jones | Forel | 2 | 5345 DK | Oss |
| O3 | 25/02/02 | C2 | Smith | Forel | 14 | 5345 DK | Oss |
| O4 | 26/02/02 | C3 | Smith | Kist | 2 | 6661 ZH | Elst |

**Table T5**



| Order no. ← | Article no. → | Num-ber |
|---|---|---|
| O1 | A1 | 3 |
| O1 | A2 | 2 |
| O1 | A6 | 4 |
| O2 | A4 | 1 |
| O3 | A3 | 1 |
| O3 | A5 | 3 |
| O4 | A3 | 2 |
| O4 | A5 | 1 |
| O4 | A6 | 7 |

**Table T6**



| Article no. ↔ | Art. name | PPP |
|---|---|---|
| A1 | pen | 3.50 |
| A2 | pen | 9.00 |
| A6 | pad | 14.00 |
| A4 | box | 25.00 |
| A3 | clip | 3.50 |
| A5 | box | 12.99 |

| | |
|---|---|
| **FKref1:** | T5(Order no.) → T3(Order no.) |
| **MCref1:** | T3(Order no.) → T5(Order no.) |
| **FKref2:** | T5(Article no.) → T6(Article no.) |

**Figure 2.12** **Tables in 2NF**

## 2.5 Orders example: Third Normal Form (3NF)

Let us now consider table T3 in figure 2.12. There are still problems with this table:

- Redundancy: facts about the name and address of a customer are stored repeatedly.
- Loss of information: if order O3 is deleted, all info about customer C2 disappears as well (if C2 has not placed any other order in the database)!

These problems occur because columns 'Name', … 'Town' are not *directly* functionally dependent on the primary key of table T2, but only *transitively*. For instance: 'Order no.' $\rightarrow$ 'Customer' and 'Customer' $\rightarrow$ 'Name' imply 'Order no.' $\rightarrow$ 'Name', so column 'Name' is *transitively* dependent on 'Order no.'. Because the same customer can appear several times in the non-key column 'Customer', everything that depends only on 'Customer' is repeated as well. Since every functional dependency corresponds with a fact type (see section 2.3.1 above), this causes facts to be stored more than once. The Third Normal Form therefore forbids transitive dependencies.

**Definition 3** (repeated from section 2.4)**:**

> **A key column is a column that belongs to any of the candidate keys (including the PK).**
> **A non-key column is a column that is NOT a key column.**

**Definition 5:**

> **A table is in Third Normal Form (3NF)**
> **IF it satisfies both the following conditions:**
> - **The table is in 2NF.**
> - **All non-key attributes are directly (i.e.: non-transitively) functionally dependent on the primary key.**

Note the distinct use in definitions 3 and 5 of the terms 'candidate keys' and 'primary key'.

Tables T5 and T6 in figure 2.12 satisfy the conditions for 3NF: they are in 2NF, and all non-key attributes depend directly on the PK.

Clearly, table T3 in figure 2.12 does not satisfy the conditions of 3NF. This can be fixed as follows:

- Make a separate table (or tables) for the functional dependencies with a tail that doesn't come from the PK (a separate table for each different tail).
- The tail of these dependencies will be the PK of the new table.
- The column(s) in the old table where the tail was becomes a foreign key (FK).
- Add a foreign key reference from the FK in the child table to the PK of the new parent table.
- Add a mandatory child reference in the opposite direction if necessary

The last point above can only be determined by interviewing a domain expert.

Figure 2.13 shows the result of bringing the tables in 3NF: please check they are all indeed in 3NF (according to the functional dependencies given so far, see sections 2.6.2 and 2.7 for a further discussion). There is a new FKref3 between tables T7 and T8, and also a mandatory child reference pointing the other way from FKref3: the domain expert confirmed that customers cannot be registered unless they also place an order.

**Table T7**

| Order no. ⟷ | Date | Customer |
|---|---|---|
| O1 | 18/02/02 | C1 |
| O2 | 25/02/02 | C1 |
| O3 | 25/02/02 | C2 |
| O4 | 26/02/02 | C3 |

**Table T8**

| Customer ⟷ | Name | Street | House no. | Postcode | Town |
|---|---|---|---|---|---|
| C1 | Jones | Forel | 2 | 5345 DK | Oss |
| C2 | Smith | Forel | 14 | 5345 DK | Oss |
| C3 | Smith | Kist | 2 | 6661 ZH | Elst |

**Table T5**

| Order no. ⟷ | Article no. ⟷ | Number |
|---|---|---|
| O1 | A1 | 3 |
| O1 | A2 | 2 |
| O1 | A6 | 4 |
| O2 | A4 | 1 |
| O3 | A3 | 1 |
| O3 | A5 | 3 |
| O4 | A3 | 2 |
| O4 | A5 | 1 |
| O4 | A6 | 7 |

**Table T6**

| Article no. ⟷ | Art. name | PPP |
|---|---|---|
| A1 | pen | 3.50 |
| A2 | pen | 9.00 |
| A6 | pad | 14.00 |
| A4 | box | 25.00 |
| A3 | clip | 3.50 |
| A5 | box | 12.99 |

| | |
|---|---|
| **FKref1:** | T5(Order no.) → T7(Order no.) |
| **MCref1:** | T7(Order no.) → T5(Order no.) |
| **FKref2:** | T5(Article no.) → T6(Article no.) |
| **FKref3:** | T7(Customer) → T8(Customer) |
| **MCref3:** | T8(Customer) → T7(Customer) |

**Figure 2.13 Tables in 3NF**

## 2.6 Boyce-Codd Normal Form (BCNF)

The conditions for a table to be in 2NF or 3NF concern functional dependencies of non-key columns. After E.F. Codd had defined 2NF and 3NF, he and his coworker R.F. Boyce found that functional dependencies of key-columns themselves can also be a source of redundancy. They defined what is now known as the Boyce-Codd Normal Form (BCNF). This is the highest normal form that can be achieved when looking at functional dependencies only.

If a table is in 3NF but not in BCNF, then it will contain redundant facts. However, bringing the table from 3NF to BCNF to repair this redundancy comes at the cost of having to implement a difficult extra integrity rule instead. Therefore the options are:

- Keep the 3NF table and write code to ensure the redundancy doesn’t cause data pollution
- Create BCNF tables and write code to implement the extra integrity rule

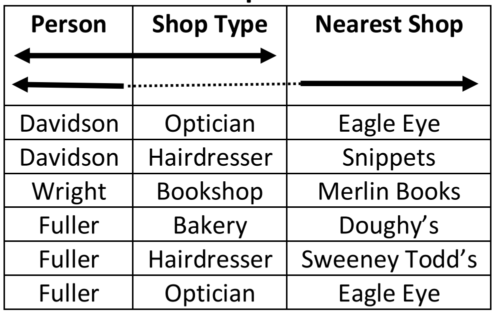
In practice, either choice is often made.

Two examples are discussed to illustrate BCNF and the consequences for database design. Section 2.6.1 contains a small example taken from the Wikipedia page about BCNF [4], and section 2.6.2 applies BCNF to the Dutch postcode (see table T8 in figure 2.13 in section 2.5).

### 2.6.1 Nearest shops

Consider the table Nearest Shops in figure 2.14. It shows which shop is nearest to a person's home.

**Table Nearest Shops**



| Person | Shop Type | Nearest Shop |
|---|---|---|
| Davidson | Optician | Eagle Eye |
| Davidson | Hairdresser | Snippets |
| Wright | Bookshop | Merlin Books |
| Fuller | Bakery | Doughy's |
| Fuller | Hairdresser | Sweeney Todd's |
| Fuller | Optician | Eagle Eye |

**Facts** (from two tuples)

```
Eagle Eye is an optician.
Doughy's is a bakery.
The nearest optician to Davidson is Eagle Eye.
The nearest bakery to Fuller is Doughy's.
```

**Figure 2.14 Table Nearest Shops**

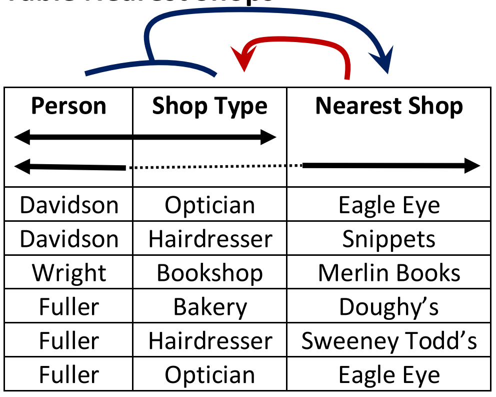
Only one shop per person per shop type is recorded. Therefore (Person + Shop Type) is a candidate key. A shop can be of only one shop type. Therefore (Person + Nearest Shop) is also a candidate key: the same shop cannot occur with the same person twice because it would then have to be of two different shop types (otherwise it would violate the other candidate key).

Table Nearest Shops is in 3NF: it is in 1NF and since there are no non-key attributes, it passes the conditions for 2NF and 3NF (check for yourself). But the table does contain redundant facts: the fact: "`Eagle Eye is an optician`" is recorded twice.

To find the source of this redundancy, in figure 2.15 the functional dependencies have been added to the table. Boyce and Codd showed that such redundancies could be removed by the extra requirement that the tail of every functional dependency must be a whole candidate key.

The dependency with a single column at the tail does not satisfy this requirenment.

**Table Nearest Shops**



| Person | Shop Type | Nearest Shop |
|---|---|---|
| Davidson | Optician | Eagle Eye |
| Davidson | Hairdresser | Snippets |
| Wright | Bookshop | Merlin Books |
| Fuller | Bakery | Doughy's |
| Fuller | Hairdresser | Sweeney Todd's |
| Fuller | Optician | Eagle Eye |

**Facts** (from two tuples)

```
Eagle Eye is an optician.
Doughy's is a bakery.
The nearest optician to Davidson is Eagle Eye.
The nearest bakery to Fuller is Doughy's.
```

**Figure 2.15 Table Nearest Shops with functional dependencies**

HAN_UNIVERSITY OF APPLIED SCIENCES

In normalization jargon, the tail of a functional dependency is called a **minimal determinant.**

**Definition 6:**

**A minimal determinant is the tail of a functional dependency.**

**Definition 7:**

**A table is in Boyce-Codd Normal Form (BCNF)**
**IF every minimal determinant is also a whole candidate key.**

If a table is in BCNF, then it is also in 3NF. The BCNF is the highest normal form that can be achieved with the use of functional dependencies only. See section 2.8 for a brief discussion of still higher normal forms.

Two solutions for handling the redundancy in table Nearest Shops are presented below, together with their consequences for implementing extra integrity rules:

- Replacing the table Nearest Shops with tables in BCNF.
- Keeping table Nearest Shops in 3NF and controlling the redundancy.

**Solution 1: Tables in BCNF**

Splitting the table in figure 2.15 yields two tables in BCNF in figure 2.16.

**Table Shop near Person**

| Person | Nearest Shop |
|---|---|
| Davidson | Eagle Eye |
| Davidson | Snippets |
| Wright | Merlin Books |
| Fuller | Doughy’s |
| Fuller | Sweeney Todd’s |
| Fuller | Eagle Eye |

**Table Shops**

| Shop | Shop Type |
|---|---|
| Eagle Eye | Optician |
| Snippets | Hairdresser |
| Merlin Books | Bookshop |
| Doughy’s | Bakery |
| Sweeney Todd’s | Hairdresser |

**Facts** (from two tuples)

```
A shop nearest to Davidson
   is Eagle Eye.
A shop nearest to Fuller
   is Doughy’s.

Eagle Eye is an optician.
Doughy’s is a bakery.
```

**FKref1: Shop near Person(Nearest Shop) → Shops(Shop)**

**Figure 2.16 Tables in BCNF**

There is no mandatory child reference: the domain expert confirmed there might be shops in table Shops that are not currently nearest to one of the persons in the other table.

The redundancy is gone in figure 2.16. But other things from figure 2.15 have disappeared as well: the second candidate key (on Person + Nearest Shop) and the other functional dependency ((Person + Shop Type) → Nearest Shop) have no counterparts in figure 2.16. If this is not compensated for, data pollution is possible. Verify carefully that adding the bottom tuple to table Shop near Person in figure 2.17 is allowed in the BCNF tables, but would violate both these 'old' constraints in the 3NF tables in figure 2.15. The reason is that both Snippets and Sweeney Todd's are hairdressers. There can be only one nearest hairdresser for each person, but this constraint is not present in figure 2.17.

**Table Shop near Person**

| Person | Nearest Shop |
|---|---|
| Davidson | Eagle Eye |
| Davidson | Snippets |
| Wright | Merlin Books |
| Fuller | Doughy's |
| Fuller | Sweeney Todd's |
| Fuller | Eagle Eye |
| **Fuller** | **Snippets** |

**Table Shops**

| Shop | Shop Type |
|---|---|
| Eagle Eye | Optician |
| Snippets | Hairdresser |
| Merlin Books | Bookshop |
| Doughy's | Bakery |
| Sweeney Todd's | Hairdresser |

**Facts** (from two tuples)

```
A shop nearest to Davidson
   is Eagle Eye.
A shop nearest to Fuller
   is Doughy's.

Eagle Eye is an optician.
Doughy's is a bakery.
```

**FKref1: Shop near Person(Nearest Shop) → Shops(Shop)**

**Figure 2.17 Tables in BCNF with wrong tuple allowed if IR1 is not implemented**

This must be repaired by adding an integrity rule to replace the missing candidate key and functional dependency. Note that implementing this integrity rule would be a difficult programming job.

**IR1: In the natural join on Shop near Person(Nearest Shop) and Shops(Shop), the values in the combination (Person + Shop Type) must be unique.**

When IR1 is implemented, the bottom tuple is no longer allowed in the BCNF tables, and so the BCNF tables are now a correct equivalent to the 3NF tables.

In the literature about normalization, the 'disappearance' of a functional dependency when upgrading tables from 3NF to BCNF (which always happens) is called 'loss of dependency'. This is a misleading term, because the functional dependency has not been lost at all, it is still there. But it acts between two tables, instead of within one table. There is nothing strange about that: for example all references (FK-references or mandatory child references) also act between two tables. But because standard normalization theory does not take constraints other than keys and functional dependencies within one table into account (a serious omission harmful to sound database design), 'loss of dependency' is seen as a problem. The real problem with normalizing however is failing to consider inter-table constraints.

**Solution 2: Tables in 3NF with controlled redundancy**

If the verbalizations from figure 2.15 are used to draw up a relational database, as is described in chapters 4 and 5 of this reader, then the tables in figure 2.18 are the result. The only difference is that table Shops is there now as well. This is better anyway, because the domein expert confirmed there might be shops in table Shops that are not currently nearest to one of the persons in the other table (such considerations are wrongfully not used in standard normalization theory).

**Table Nearest Shops**

| Person | Shop Type | Nearest Shop |
|---|---|---|
| Davidson | Optician | Eagle Eye |
| Davidson | Hairdresser | Snippets |
| Wright | Bookshop | Merlin Books |
| Fuller | Bakery | Doughy’s |
| Fuller | Hairdresser | Sweeney Todd’s |
| Fuller | Optician | Eagle Eye |

**Table Shops**

| Shop | Shop Type |
|---|---|
| Eagle Eye | Optician |
| Snippets | Hairdresser |
| Merlin Books | Bookshop |
| Doughy’s | Bakery |
| Sweeney Todd’s | Hairdresser |

**Facts** (from two tuples)

```
Eagle Eye is an optician.
Doughy’s is a bakery.

The nearest optician to
Davidson is Eagle Eye.
The nearest bakery to
Fuller is Doughy’s.
```

**FKref1: Nearest Shops(Nearest Shop) → Shops(Shop)**
**(to be replaced with Ref2, see below)**

**Figure 2.18 Tables in 3NF**

In figure 2.18, table Nearest Shops is in 3NF because the functional dependency Nearest Shop → Shop Type does not satisfy the BCNF conditions. The fact “`Eagle Eye is an optician`” is recorded twice in table Nearest Shops (as well as once more in table Shops).

Instead of bringing table Nearest Shops into BCNF, it is also possible to ensure that the redundancy cannot cause data pollution in these 3NF tables. This can be achieved by **replacing FKref1** with another reference:

**Ref2: Nearest Shops(Nearest Shop + Shop Type) → Shops(Shop + Shop Type).**

This reference says that every (Nearest shop + Shop type) combination in table Nearest Shops must come from table Shops. Therefore no data pollution because of the redundant facts in table Nearest Shops can occur anymore (try it!).

Ref2 is not a foreign key reference, because it points to more columns than the PK of table Shops. Therefore it cannot be implemented declaratively, but must be programmed separately.

**Summary of the two solutions**

In summary, the two solutions are:

- The tables in figure 2.17 together with integrity rule IR1.
- The tables in figure 2.18 together with reference Ref2 (instead of FKref1).

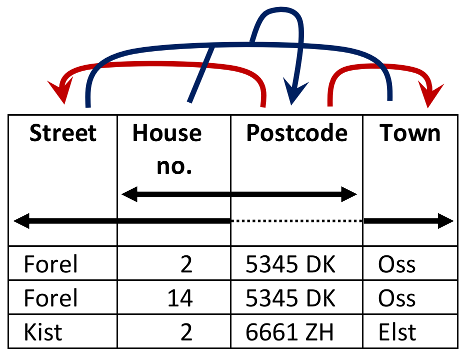
Which solution would you prefer to realize?

### 2.6.2 Dutch postcode

In the literature on normalizing, it is a standard example to consider a table of just addresses with a postcode: see table Addresses with Postcode in figure 2.19 below. In practice too, domain tables for addresses in the Netherlands are now often used: many webshops and government or business organizations use a digital version of the 'postcode book' that lists all Dutch addresses with their postcodes, either in 3NF or BCNF form. Both forms are discussed below.

In the Netherlands, every postal address has two identifiers. 'Street' + 'House no.' + 'Town' will still do, but since the postcode was introduced in 1977, only 'Postcode' + 'House no.' is enough, and the preferred way to identify a snailmail address. Therefore, table Addresses with Postcode in figure 2.19 has two candidate keys. Every postal address has only one postcode. This is the reason for the functional dependency pointing to 'Postcode' in table Addresses with Postcode. Usually, several addresses share the same postcode. but they are always on the same side of the same street in the same town (see also section 5.3 in the FCO-IM book in [2] or [3], which explains the postcode structure in more detail). Therefore addresses with the same postcode differ only in 'House no.'. This is the reason for the other two functional dependencies in table Addresses with Postcode: only one 'Street' and one 'Town' can belong to a given 'Postcode'.

**Table Addresses with Postcode**



| Street | House no. | Postcode | Town |
|---|---|---|---|
| Forel | 2 | 5345 DK | Oss |
| Forel | 14 | 5345 DK | Oss |
| Kist | 2 | 6661 ZH | Elst |

**Facts:**

```
There is an address postcode 5345 DK with house no.  2.
There is an address postcode 5345 DK with house no. 14.
There is an address postcode 6661 ZH with house no.  2.

Postcode 5345 DK belongs to street Forel in Oss.
Postcode 6661 ZH belongs to street Kist in Elst.
```

**Figure 2.19 Domain table for Dutch addresses in 3NF**

Table Addresses with Postcode is in 3NF: it meets the 3NF requirements because it is in 1NF (the values in column 'Postcode' are regarded as atomic) and there are no non-key columns. But there is still redundancy in the table: the fact "**`Postcode 5345 DK belongs to street Forel in Oss`**" is recorded twice.

The redundancy problem arises because the tail of the dependencies Postcode → Street and Postcode → Town is not a candidate key. Therefore a value in column 'Postcode' can occur more than once, together with everything that depends on it, and so cause redundancy. The Boyce-Codd Normal Form forbids this.

Two solutions for handling the redundancy in table ADDRESSES WITH POSTCODE are presented below, together with their consequences for implementing extra integrity rules:

- Replacing table Addresses with Postcode with tables in BCNF.
- Keeping table Addresses with Postcode in 3NF and controlling the redundancy.

**Solution 1: Tables in BCNF**

Here is the definition of Boyce-Codd Normal Form, repeated from section 2.6.1:

**Definition 6:**

**A minimal determinant is the tail of a functional dependency.**

**Definition 7:**

**A table is in Boyce-Codd Normal Form (BCNF)**
**IF every minimal determinant is also a whole candidate key.**

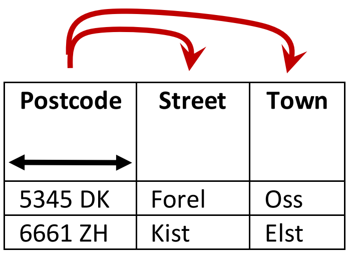
Clearly, table Addresses with Postcode in figure 2.19 is in 3NF but not in BCNF. Tables Addresses and Postcode in figure 2.20 show how table Addresses with Postcode can be split into two tables in BCNF. Note the ordinary foreign key reference FKref4, and the mandatory child reference in the opposite direction: all postcodes have at least one address.

**Table Addresses**

| House no. | Postcode |
|---|---|
| 2 | 5345 DK |
| 14 | 5345 DK |
| 2 | 6661 ZH |

**Facts:**
```
There is an address
  5345 DK, 2.
There is an address
  5345 DK, 14.
There is an address
  6661 ZH, 2.
```

**Table Postcode**



| Postcode | Street | Town |
|---|---|---|
| 5345 DK | Forel | Oss |
| 6661 ZH | Kist | Elst |

**Facts:**
```
Postcode 5345 DK
   belongs to street
   Forel in Oss.
Postcode 6661 ZH
   belongs to street
   Kist in Elst.
```

**FKref4:** Addresses(Postcode) → Postcode(Postcode)
**MCref4:** Postcode(Postcode) → Addresses(Postcode)

**Figure 2.20 Domain tables for Dutch addresses in BCNF**

The redundancy is gone in figure 2.20. But other things from figure 2.19 have disappeared as well: the second candidate key (on Street + House no. + Town) and the other functional dependency ((Street + House no. + Town) → Postcode) have no counterparts in figure 2.20. If this is not compensated for, data pollution is possible. Verify that adding the bottom tuple to tables Addresses and Postcode in figure 2.21 is allowed in the BCNF tables, but would violate both these 'old' constraints in the 3NF table in figure 2.20. The reason is that each address can have only one postcode, but this constraint is not present in figure 2.21.

**Table Addresses**

| House no. | Postcode |
|---|---|
| 2 | 5345 DK |
| 14 | 5345 DK |
| 2 | 6661 ZH |
| **2** | **6661 XX** |

**Facts:**

```
There is an address
  5345 DK, 2.
There is an address
  5345 DK, 14.
There is an address
  6661 ZH, 2.
```

**Table Postcode**

| Postcode | Street | Town |
|---|---|---|
| 5345 DK | Forel | Oss |
| 6661 ZH | Kist | Elst |
| **6661 XX** | **Kist** | **Elst** |

**Facts:**

```
Postcode 5345 DK
   belongs to street
   Forel in Oss.
Postcode 6661 ZH
   belongs to street
   Kist in Elst.
```

| | |
|---|---|
| **FKref4:** | Addresses(Postcode) → Postcode(Postcode) |
| **MCref4:** | Postcode(Postcode) → Addresses(Postcode) |

**Figure 2.21 Domain tables for Dutch addresses in BCNF**

This must be repaired by adding an integrity rule to replace the missing candidate key and functional dependency. Note that implementing this integrity rule would be a difficult programming job.

**IR2a:**
**In the natural join on Addresses(Postcode) and Postcode(Postcode), the values in the combination (Street + House no. + Town) must be unique.**

When IR2 is implemented, the bottom tuple is no longer allowed in the BCNF tables, and so the BCNF tables are now a correct equivalent to the 3NF tables.

In the literature about normalization, the 'disappearance' of a functional dependency when upgrading tables from 3NF to BCNF (which always happens) is called 'loss of dependency'. This is a misleading term, because the functional dependency has not been lost at all, it is still there. But it acts between two tables, instead of within one table. There is nothing strange about that: for example all references (FK-references or mandatory child references) also act between two tables. But because standard normalization theory does not take constraints other than keys and functional dependencies within one table into account (a serious omission harmful to sound database design), 'loss of dependency' is seen as a problem. The real problem with normalizing however is failing to consider inter-table constraints.

**Solution 2: Tables in 3NF with controlled redundancy**

In figure 2.19, table Addresses with Postcode is in 3NF because the functional dependencies pointing to column Postcode do not satisfy the BCNF conditions. The fact "`Postcode 5345 DK belongs to street Forel in Oss.`" is recorded twice.

Instead of bringing table Addresses with Postcode into BCNF, it is also possible to ensure that the redundancy cannot cause data pollution in these 3NF tables. This can be achieved by adding integrity rule IR2b:

**IR2b:**
**For any two tuples in table Addresses with Postcode: IF the values in column Postcode are the same THEN the values in column Street must also be the same AND the values in column Town must also be the same.**

This integrity rule ensures that no data pollution because of the redundant facts in table Nearest Shops can occur anymore (try it!). Note that implementing this integrity rule would be a difficult programming job.

**Summary of the two solutions**
In summary, the two solutions are:

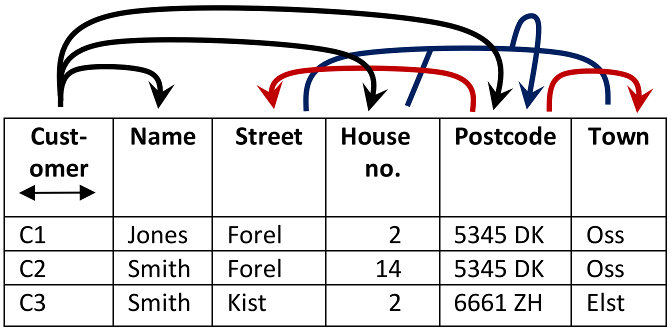
- The table in figure 2.19 together with integrity rule IR2b.
- The tables in figure 2.20 together with IR2a.

Which solution would you prefer to realize?

## 2.7 Orders example: Boyce-Codd Normal Form (BCNF)

Figure 2.22 shows table T8 from section 2.5 again, but with corrected functional dependencies in view of the structure of addresses with postcode, discussed in section 2.6.2. Note the complex dependencies and the verbalization of the facts concerning addresses.

**Table T8**



| Cust-omer ⟷ | Name | Street | House no. | Postcode | Town |
|---|---|---|---|---|---|
| C1 | Jones | Forel | 2 | 5345 DK | Oss |
| C2 | Smith | Forel | 14 | 5345 DK | Oss |
| C3 | Smith | Kist | 2 | 6661 ZH | Elst |

**Facts** (from two tuples)**:**

```
The name of customer C1 is Jones.
The name of customer C2 is Smith.
Customer C1 lives at 5345 DK,  2.
Customer C2 lives at 5345 DK, 14
Postcode 5345 DK belongs to
   street Forel in Oss.
```

**Figure 2.22 Table T8 with corrected functional depencies**

Carefully check all the following points (abbreviations for column names are used):

- There are transitive dependencies C → P, P → S and C → P, P → T, so T8 is now seen not to be in 3NF after all (only in 2NF). The remedy: split T8 into two tables T9 and T10 (see figure 2.23 below) as usual to remove the transitive dependencies. This is analogous to splitting table Addresses with Postcode, as discussed in section 2.6.2.
- In this split, the dependency (S, H, T) → P is 'lost'. So in this way, <u>even 3NF cannot be achieved without 'loss of dependency' (see section 2.6)!</u> However, T9 and T10 are now directly in BCNF.
- Another split would in theory be possible (but prove to be meaningless): split T8 into Tx(<u>C</u>, N, S, H, T) and Ty(S, <u>H, P</u>, T), with both tables in 3NF, preserving all dependencies. But: Ty is the same as table Addresses with Postcode discussed in section 2.6.2 above (in 3NF but not in BCNF). Splitting Ty (= Addresses with Postcode) into Addresses and Postcode (see section 2.6.2) would then finally produce <u>three</u> tables for T8 in BCNF, and we would still 'loose' (S, H, T) → P. This would also be meaningless from a fact point of view, because the primary identifier for an address used by the post is indeed (P + H), not (S + H + T).

So finally, the tables in BCNF would be the ones shown in figure 2.23 (note the extra integrity rule).

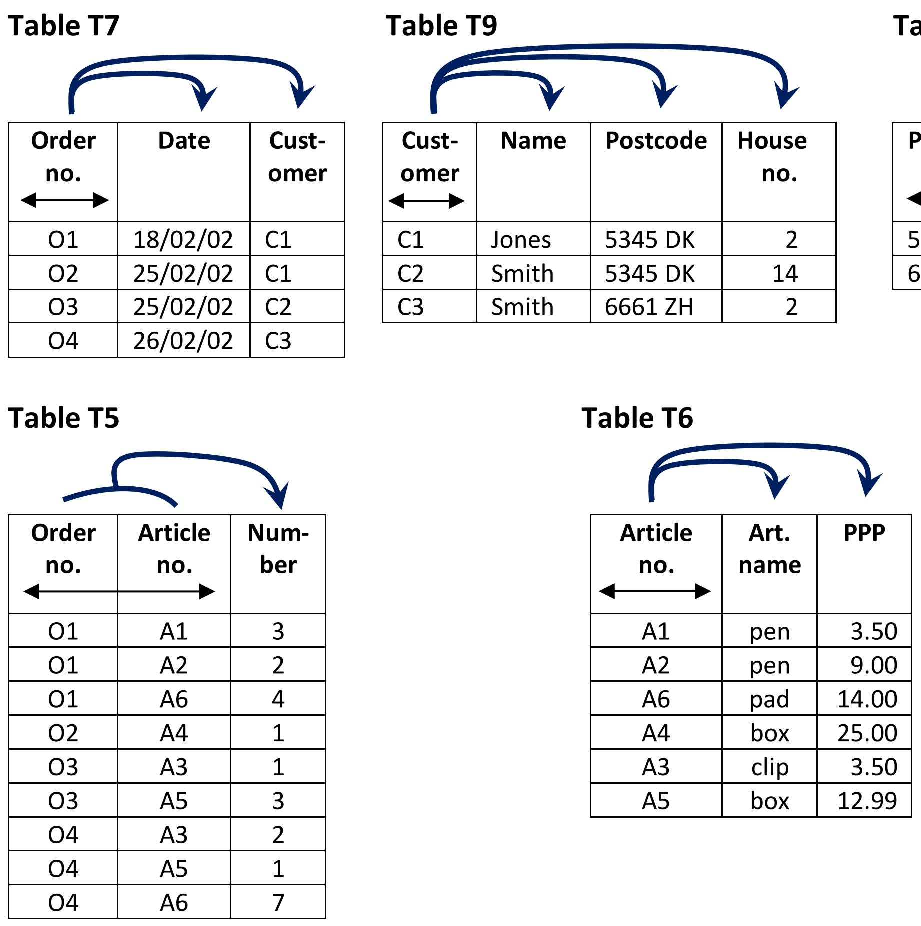


**Table T7**

| Order no. ↔ | Date | Cust-omer |
|---|---|---|
| O1 | 18/02/02 | C1 |
| O2 | 25/02/02 | C1 |
| O3 | 25/02/02 | C2 |
| O4 | 26/02/02 | C3 |

**Table T9**

| Cust-omer ↔ | Name | Postcode | House no. |
|---|---|---|---|
| C1 | Jones | 5345 DK | 2 |
| C2 | Smith | 5345 DK | 14 |
| C3 | Smith | 6661 ZH | 2 |

**Table T10**

| Postcode ↔ | Street | Town |
|---|---|---|
| 5345 DK | Forel | Oss |
| 6661 ZH | Kist | Elst |

**Table T5**

| Order no. ⟵ | Article no. ⟶ | Num-ber |
|---|---|---|
| O1 | A1 | 3 |
| O1 | A2 | 2 |
| O1 | A6 | 4 |
| O2 | A4 | 1 |
| O3 | A3 | 1 |
| O3 | A5 | 3 |
| O4 | A3 | 2 |
| O4 | A5 | 1 |
| O4 | A6 | 7 |

**Table T6**

| Article no. ↔ | Art. name | PPP |
|---|---|---|
| A1 | pen | 3.50 |
| A2 | pen | 9.00 |
| A6 | pad | 14.00 |
| A4 | box | 25.00 |
| A3 | clip | 3.50 |
| A5 | box | 12.99 |

**Figure 2.23 Tables for the Order example in BCNF**

Integrity rules between these tables:

**FKref1:** T5(Order no.) → T7(Order no.)
**MCref1:** T7(Order no.) → T5(Order no.)
**FKref2:** T5(Article no.) → T6(Article no.)
**FKref3:** T7(Customer) → T9(Customer)
**MCref3:** T9(Customer) → T7(Customer)
**FKref4:** T9(Postcode) → T10(Postcode)
**MCref4:** T10(Postcode) → T9(Postcode) (if only postcodes of existing customers are to be recorded)
**IR2a:** **In the natural join on T9(Postcode) and T10(Postcode), the values in the combination (Street + House no. + Town) must be unique.**

Note:

- Mandatory child references (or their absence) don't follow from normalization considerations, but can only be determined by other means (such as interviewing a domain expert).
- There are seven references and one other integrity rule between these tables, all of which require checks on two tables (instead of on only one). Only four of these are 'ordinary' foreign key references that can be implemented declaratively. The others require programming. All these (but especially the 'other constraints') require checking two tables simultaneously, so I can see no reason to avoid designing tables in BCNF just because a dependency will be 'lost'.

## 2.8 Higher Normal Forms

A discussion of higher normal forms would require treating the concept of 'multivalued dependencies', the definition of which is quite mathematical in nature. The definitions in the literature of 4NF, 5NF and other normal forms suffer even more seriously from the same drawback as mentioned earlier: no 'eye for elementary fact types'. This chapter will therefore conclude with only a few general remarks.

**Definition 8**

**A table is in 5NF, if it cannot be replaced by two (or more) smaller tables without loss of information.**
Note: a table is also in 4NF if it is in 5NF.

- It is very difficult to find realistic examples of tables that are in 4NF, but not in 5NF (or even of tables in BCNF but not in 4NF). Reasons are:
    - Examples in the literature are often very silly from an elementary fact type point of view.
    - Constraint considerations (Not Null or Optional columns?) are almost always left out in the discussion in the literature, yet another sign that 'fact-based thinking has not yet gained ground here. All the examples I have seen in the literature fail to consider these.
- The definition of BCNF makes no reference to lower normal forms, and is therefore a beautiful and final statement about the structure of tables when only functional dependencies are considered.
- The definition of 5NF makes no reference to lower normal forms, and is therefore a beautiful and final statement about the structure of tables when functional and multivalued dependencies are considered.

Section 3.3.1.3 in the 2002 FCO-IM book [3] discusses the 'projection/join' test (see this book for elaborate examples). This test is designed to check definition 8 above. It works properly provided one essential condition is met (see below). When carried out, it ensures that a table is in 5NF (and therefore also in 4NF, BCNF and lower normal forms as well).

The projection/join test starts from a table with N attributes, with an example population that is big enough. The test first splits this table (and its population) into all N possible smaller tables with N-1 attributes. Then, using N-1 natural joins, these N smaller tables are rejoined into the original table with N attributes. If we find at the end of this process that we have *more tuples* than we started out with, then the original table is *not* in 5NF.

To work properly, this test requires a *significant* population, i.e. a population that contains *every allowed combination of values*. However, since there exists no way of telling whether a population is significant, the test is a bit circular in nature: a table is in 5NF if it passes the test, but you can only be sure that the test works properly if the population is significant. To know whether a population is significant, you practically have know in advance that the table is in 5NF!

**Definition 9**

> **A set of tables is in Optimal Normal Form (ONF) w.r.t. a given information model, if all tables are in 5NF and the number of tables is minimal.**

The algorithms to derive a relational schema from a conceptual data model (in ERM, ORM or FCO-IM) yield a result that is almost certainly free of all forms of redundancy (there are a few known exceptions not yet implemented). They therefore yield ONF database structures in the vast majority of cases.
It is not meaningful to consider a relational model with a 'minimal number of tables' without referring to the 'given information model' from which it was derived. In such a data model, certain design choices can influence the number of generated tables (see also section 5.1 in [3]).

# 3 Introduction to ERM and Fact Oriented ERM (FO-ERM)

## 3.1 Concepts in ERM

The concepts of ERM will be introduced using the example diagram in figure 3.1.

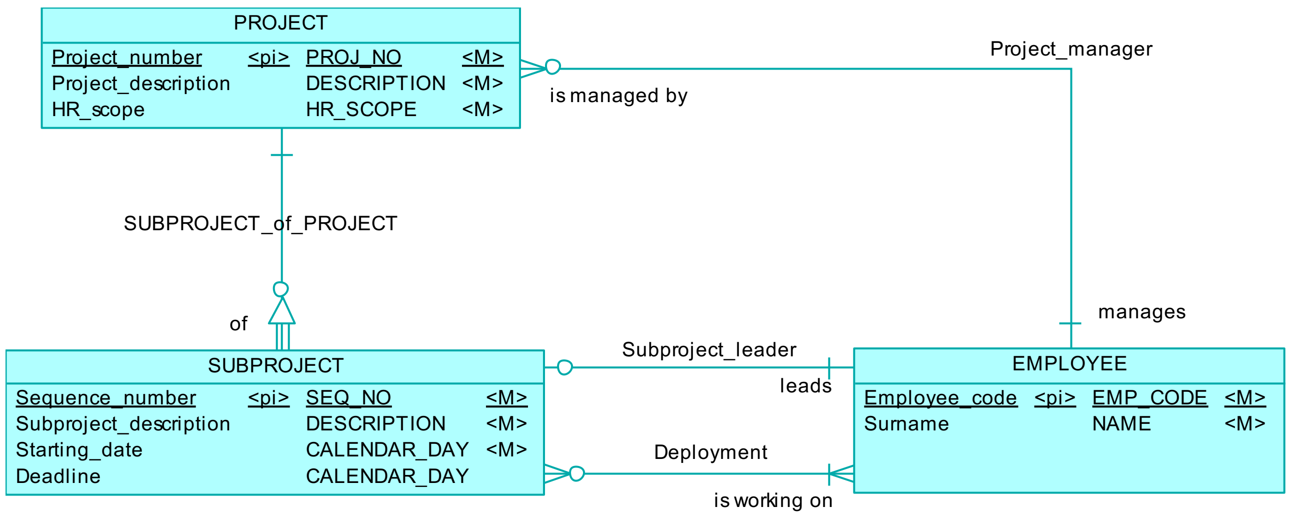

PROJECT
Project_number <pi> PROJ_NO <M>
Project_description DESCRIPTION <M>
HR_scope HR_SCOPE <M>
Project_manager
is managed by
SUBPROJECT_of_PROJECT
of
manages
SUBPROJECT
Sequence_number <pi> SEQ_NO <M>
Subproject_description DESCRIPTION <M>
Starting_date CALENDAR_DAY <M>
Deadline CALENDAR_DAY
Subproject_leader
leads
Deployment
is working on
EMPLOYEE
Employee_code <pi> EMP_CODE <M>
Surname NAME <M>

**Figure 3.1** **Classic ERM diagram**

### 3.1.1 Entity Type (ET), and entity

PROJECT, SUBPROJECT and EMPLOYEE are *entity types*. An entity type represents a *kind of person, thing or concept* about which data are to be stored. An Entity Type (ET) in ERM corresponds roughly to a *table* in the Relational Model. However, there are a few essential differences between entity types and tables, discussed below (section 3.1.8). In this reader, the convention is used to write names of entity types in capital letters, but this is no requirement.

Entity type PROJECT stands for the collection (class, kind, type) of all particular projects. One individual project, like 'project P315', is one *entity instance*, or *entity* for short. Do not confuse 'entity' and 'entity type': the former term refers to one concrete instance, the latter to the abstract class. So 'project P315' is one entity that belongs to entity type PROJECT.

### 3.1.2 Attribute (Att)

Project_number and HR_scope are *attributes* of PROJECT. Likewise, Deadline is an attribute of SUBPROJECT and Surname an attribute of EMPLOYEE. An attribute represents a property of an entity type that is to be recorded. An Attribute (Att) in ERM corresponds to a *column* in the Relational Model. In this reader, the convention is used to write names of attributes with an initial capital letter, and underscores instead of blanks, but this is no requirement.

### 3.1.3 Domain

PROJ_NO is the domain of attribute Project_number of entity type PROJECT. Likewise, NAME is the domain of attribute Surname of entity type EMPLOYEE, etc. A domain in ERM corresponds to a domain in the Relational Model. In this reader, the convention is used to write names of domains in capital letters, but this is no requirement.

A data type can be assigned to a domain (see also chapter 1, **Domain**), but is not shown in figure 3.1. That is because in this reader domains are considered to belong to the *conceptual level* of data modeling (ERM), and data types to the *physical level* of data modeling (an implemented relational database).

### 3.1.4 Mandatory constraint

The code '<M>' added to an attribute means: this attribute is *mandatory*, i.e.: this attribute must have a value for every entity recorded in the database. A mandatory constraint on an attribute in ERM corresponds to a NOT NULL integrity rule on a column in the Relational Model.

The *absence* of a mandatory constraint in an attribute means the attribute is *optional*: a value can be recorded in this attribute for any entity, but does not have to be recorded for every entity. This corresponds to a NULL-column in the Relational Model.

### 3.1.5 ERM Rule 1: Every ET must have a *primary identifier*

An *identifier* of an entity type in ERM corresponds to a *key* of a table in the Relational model (see chapter 1). An identifier is an attribute (or smallest set of attributes) with the property that the value (or combination of values) must be unique for each entity of the entity type. Every ET must have at least one identifier, and may have several. One identifier is chosen as the *primary identifier*, any other ones then become *alternative identifiers*. In ERM the identification of an ET is more complex however than in the relational model. See the section on strong and weak ETs below.

### 3.1.6 Strong ET: needs only its own attributes for its own <pi>

A *strong entity type* is an ET that has a primary identifier that consists only of its *own* attributes. These attributes are marked with '<pi>'. In figure 3.1, EMPLOYEE and PROJECT are strong ETs: The <pi> of EMPLOYEE is Employee_code, the <pi> of PROJECT is Project_number. A <pi> could also consist of two or more attributes together (the combination of their values is then unique).

### 3.1.7 Weak ET: needs <pi>s from other ETs for its own <pi>

A *weak entity type* is an ET that needs to 'borrow' the <pi> of *another* ET for its own <pi>. In figure 3.1, the complete <pi> of SUBPROJECT is the combination Sequence_number + Project_number. Sequence_number is one of its own attributes, and Project_number is the <pi> of PROJECT. The small triangle on the line between SUBPROJECT and PROJECT, pointing from SUBPROJECT to PROJECT, means that SUBPROJECT *is dependent on* PROJECT. This tells us that SUBPROJECT needs the <pi> of PROJECT for its own identifier as well. Apparently, subprojects are numbered sequentially *in each project*, like in 'subproject P315, 1; subproject P315, 2; subproject P422, 1; subproject P422, 2; etc. SUBPROJECT is therefore a *child ET* of PROJECT, and PROJECT is a *parent ET* of SUBPROJECT. In a large ERM diagram, there can be several levels of parents and children (a child ET can have several parents, which in turn can be children of higher parents, etc.)

### 3.1.8 ERM Rule 2: No ‘foreign-key’ attributes are allowed.

Why not include an attribute Project_number in SUBPROJECT, and give it a <pi>-indicator? This is not allowed in a conceptual ERM model, and this is one of the major differences between ERM and the Relational Model. In the Relational Model, a foreign key is a column (or set of columns) that is a copy of the primary key from another table (see also chapter 1). This means that in the Relational Model there is redundancy in the table structures: the primary key of one table can have been copied to many other tables. If this primary key would change, then all foreign keys must be changed likewise. This is redundancy at the table structure level (metadata level). ERM avoids this redundancy by forbidding to make such copies: an ET is not allowed to have an attribute (or set of attributes) that is the <pi> of another ET. Instead, it is indicated from which ET the <pi> is needed by the small triangle on the connecting line between the ETs. Now if this borrowed <pi> would change (for example replace Project_number with two new attributes Project_type and Project_code, which together become the new <pi> of PROJECT), then nothing has to change in SUBPROJECT.

### 3.1.9 Relationship type

Lines connecting two Entity Types (ETs) represent a *relationship type* between those ETs. See figure 3.2 below, which repeats figure 3.1, with some extra text added. The Relationship Type (RT) Project_manager between EMPLOYEE and PROJECT models that a project is managed by an employee. Between EMPLOYEE and SUBPROJECT there are two different RTs, one for employees leading subprojects, one for employees working on subprojects. Finally RT SUBPROJECT_of_PROJECT models to which project a subproject belongs. In this reader, the convention is used to write names of non-dependent RTs with an initial capital letter (like attributes), and names of dependent RTs in the format ETNAME1_of_ETNAME2 or ETNAME1_in_ETNAME2, but this is no requirement.

Note: very often only the word ‘relationship’ is used instead of ‘relationship type’. Strictly speaking, a relationship concerns only two concrete entities, and is at the instance level, whereas a relationship *type* concerns two entity *types*, and is on the type level. But if the context is clear this should not be a problem.

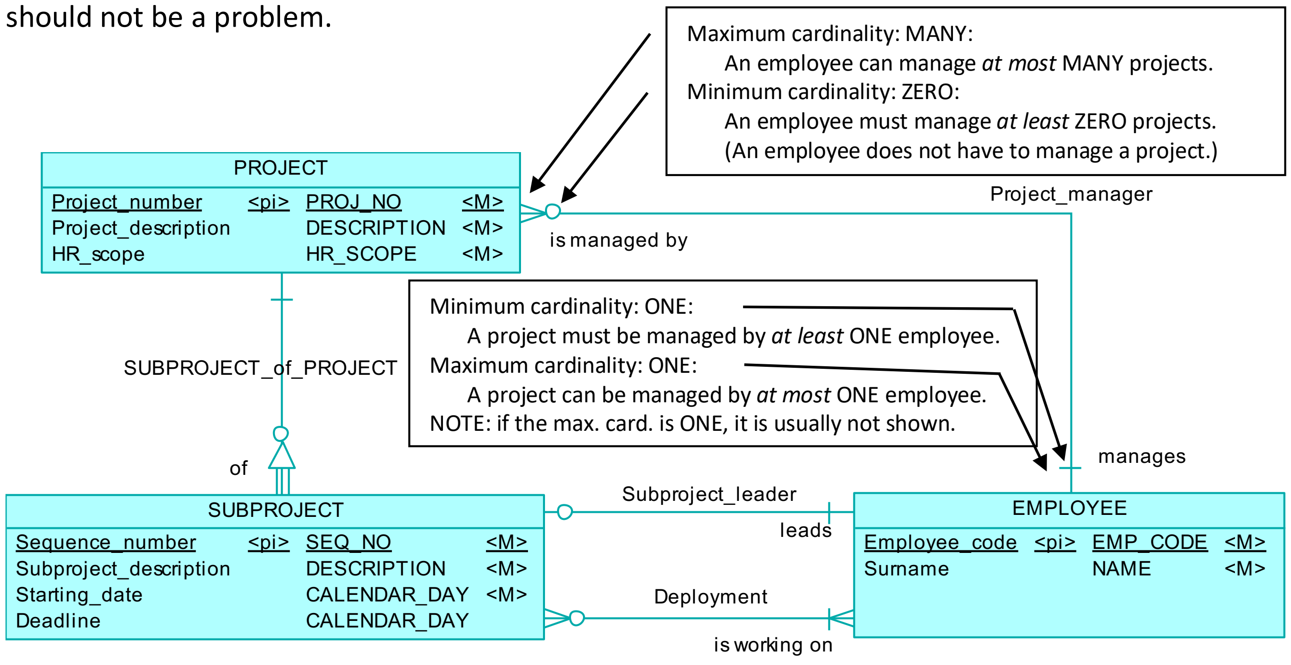

Maximum cardinality: MANY:
An employee can manage *at most* MANY projects.
Minimum cardinality: ZERO:
An employee must manage *at least* ZERO projects.
(An employee does not have to manage a project.)
PROJECT
Project_number <pi> PROJ_NO <M>
Project_description DESCRIPTION <M>
HR_scope HR_SCOPE <M>
is managed by
Project_manager
Minimum cardinality: ONE:
A project must be managed by *at least* ONE employee.
Maximum cardinality: ONE:
A project can be managed by *at most* ONE employee.
NOTE: if the max. card. is ONE, it is usually not shown.
SUBPROJECT_of_PROJECT
of
manages
Subproject_leader
leads
SUBPROJECT
Sequence_number <pi> SEQ_NO <M>
Subproject_description DESCRIPTION <M>
Starting_date CALENDAR_DAY <M>
Deadline CALENDAR_DAY
EMPLOYEE
Employee_code <pi> EMP_CODE <M>
Surname NAME <M>
Deployment
is working on

**Figure 3.2** **Cardinalities in Project_manager highlighted**

### 3.1.10 Cardinalities

Every Relationship Type (RT) has four constraints that must be determined very carefully: a *minimum cardinality* and a *maximum cardinality* at each side of the RT. See figure 3.2, which makes the meaning of the cardinalities clear in RT Project_manager, and also how and where they are displayed.

A minimum cardinality can be either *at least ZERO*, displayed by a small zero-symbol, or *at least ONE*, displayed by a small stroke. A maximum cardinality can be either *at most ONE*, displayed by a small stroke, or *at most MANY,* displayed by three strokes, either as a crow's foot (⋖) or as parallel lines (≡). If the maximum cardinality is *at least ONE*, it is usually omitted (as it is in figure 3.2).

The other cardinalities in figure 3.2 are (please check how they are displayed):
RT Subproject_leader:
- At the side of EMPLOYEE:
  - Min. card.: ONE. A subproject must be led by at least one employee.
    Max. card: ONE. A subproject can be led by at most one employee.
    Note: this max. card. of ONE is not displayed explicitly.
- At the side of SUBPROJECT:
  - Min. card.: ZERO. An employee does not have to lead a subproject.
    Max. card.: ONE. An employee can lead at most one subproject.
    Note: this max. card. of ONE is not displayed explicitly.

RT Deployment:
- At the side of EMPLOYEE:
  - Min. card.: ONE. A subproject must have at least one employee working on it.
    Max. card: MANY. A subproject can have many employees working on it.
- At the side of SUBPROJECT:
  - Min. card.: ZERO. An employee does not have to work on a subproject.
    Max. card.: MANY. An employee can work on many subprojects.

RT SUBPROJECT_of_PROJECT:
- At the side of SUBPROJECT:
  - Min. card.: ZERO. A project does not have to have a subproject.
    Max. card: MANY. A project can have many subprojects.
- At the side of PROJECT:
  - Min. card.: ONE. A subproject must belong to at least one project.
    Max. card.: ONE. A subproject can have at most one subproject.
    Note: this max. card. of ONE is not displayed explicitly.
    **NOTE: if one of the ETs in the RT is a weak entity type** (SUBPROJECT is weak here), **then both the minimum and maximum cardinalities at the other side must be ONE.** This is because the weak ET needs the <pi> of the other ET for its own identification, so each entity from the weak ET must have a relationship with **exactly one** entity of the other ET.

### 3.1.11 Roles

The text fragments in figure 3.2 'manages', 'is managed by', 'leads', 'of' etc. that are written next to the RT lines are called *roles*. They serve to make the meaning of the RT more clear. It is strongly recommended to always add at least one role to each RT.

## 3.2 FO-ERM: adding fact types and example populations

### 3.2.1 Facts, and fact types versus ETs, Atts and RTs.

Classic ERM diagrams do not clearly show the most important units of information, namely the *fact types* that are being modeled. Storing something new in a database (or deleting something old from it) almost never concerns a single attribute, but almost always several attributes at the same time, which together make up a complete fact. It is the complete facts that are the most important structures in a database.

A *fact* is one concrete indivisible unit of information that is stored in the database, for example (see figure 3.2): Fact 1: The surname of employee RoeRch is Roberts. or Fact 2: Employee RoeRch is working on subproject 2 of project P422. Please note that these facts each concern several attributes, from the same ET (Fact 1) or even different ETs (Fact 2).

A *fact type* is one abstract indivisible kind of information that is stored in the database, for example surnames of employees. A fact type can be shown by giving a *predicate* for it. For example: Fact Type 1: The surname of employee <Employee_code> is <Surname>, or Fact Type 2: Employee <Employee_code> is working on subproject <Sequence_number> of project <Project_number>. So a predicate is a sentence, in which the concrete instance values have been replaced by blanks, indicated with pointed brackets ('<' and '>') that enclose the name of the attribute that contains the values that are to be filled in to obtain a concrete fact. Please note that these fact types each concern several attributes, from the same ET (Fact Type 1) or even different ETs (Fact Type 2).

So an ET is usually a collection of several fact types (every <pi> + one other Att makes up one fact type), and an Att is half a fact type at best. In addition, a RT is also a part of a fact type (the other parts are the complete <pi>s of the ETs in the RT). It is a weakness of ERM that it models one concept (fact type) in such different ways.

### 3.2.2 Instance level versus type level

Classic ERM diagrams only show the type level of an information model, no instance level values. This has the advantage of yielding a concise type level model. However, in such concise ERM diagrams, the *meaning* of the modeled information is not very clear. Adding a fact type with a few examples of instance level facts can make the meaning of (the facts in) the ETs, Atts and RTs much more clear. Such examples of concrete facts are called an *example population* of the diagram. See for example Figure 3.1. Can you explain what is the meaning of the attributes in ET PROJECT from looking at this diagram (HR_scope for instance)? And how is this in the diagram in figure 3.3?

### 3.2.3 Fact types and example populations clarify the meaning of the model

In figure 3.3, several (not all) fact types and example populations have been added. A diagram showing at least some fact types and example populations is called a Fact-Oriented ERM diagram.

In practice, capturing the *semantics* (i.e.: the meaning) of the modeled information is very important and highly valued by stakeholders. Information modelers devote over half of their time to 'getting to know the data', or 'understanding what this is all about'. A good information model cannot be drawn up if the modeler does not understand what it is he/she is modeling.

Indeed, it is impossible to verify (check) and validate (approve) the correctness of a model if it does not make sufficiently clear what it is modeling exactly. Even experiended modelers can overlook serious errors in an abstract classic ERM model.

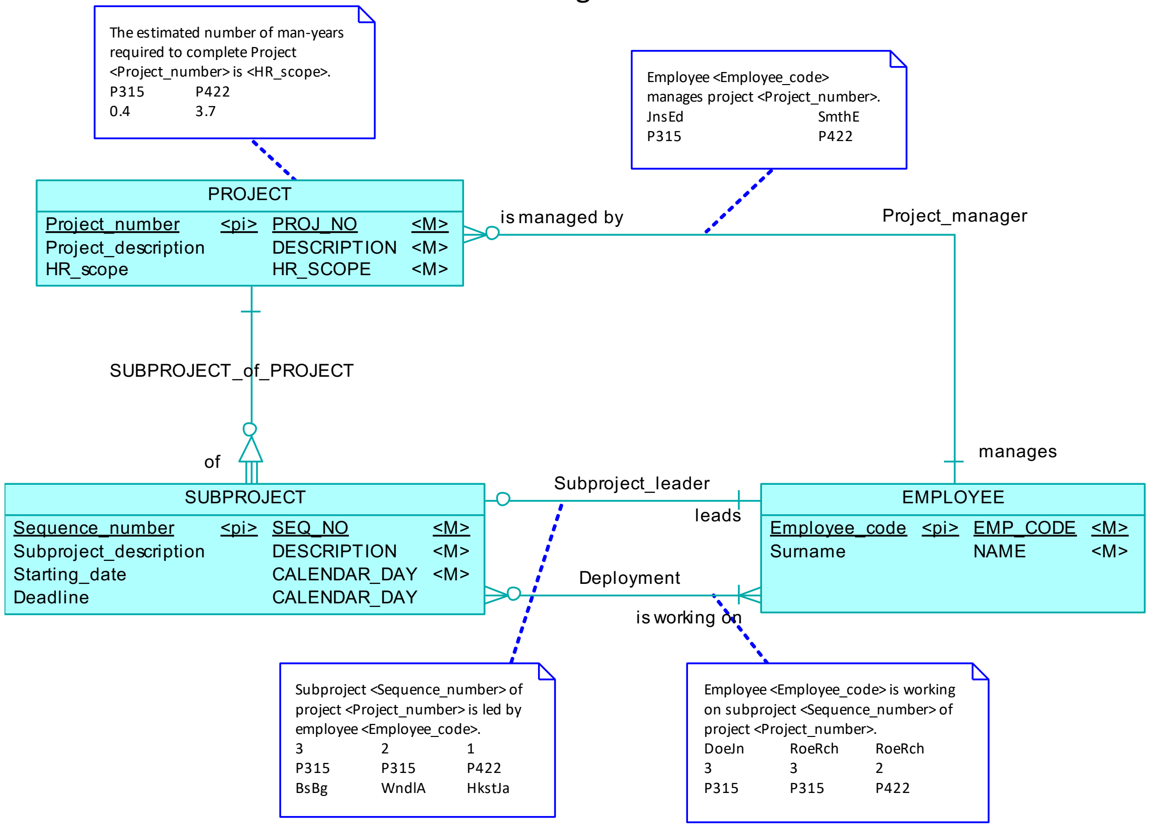
On the other hand, many attributes are quite clear: everyone is familiar with personal particulars like surname, address, shoe size, etc., or clear properties of everyday entity types like article name, article cost price, article color, etc. This is the strength of ERM: it offers a fast way to model many trivial properties of obvious entity types. The trouble starts where arcane properties or complex relationships between ETs are concerned. Therefore we recommend the following way of modeling.

**Recommendation for adding fact types and example populations**

- **Always add a fact type and example population for**
  - **Each RT that does not have a dependency if the meaning of the RT is not self-explanatory.**
  - **Each <pi>+Att combination within one ET if the meaning of the Att is not self-explanatory.**
- **Otherwise: only add a fact type and example population if this greatly clarifies the model**

RTs are fact types in disguise, usually with a large number of components if weak ETs are involved. Therefore it is good practice to almost always add the semantics there. This does not hold for RTs with a dependency: these mostly only serve to indicate where the parent ET is, and do not model storable facts types (for example: try to find a fact type for SUBPROJECT_of_PROJECT in figure 3.3).

Exceptions to these recommendations can exist, so please regard them as guidelines, not as hard rules, and apply them wisely (but it is better to have one example too many than one too few).
Thes above recommendations were followed in figure 3.3.



The estimated number of man-years required to complete Project <Project_number> is <HR_scope>.
P315 P422
0.4 3.7
Employee <Employee_code> manages project <Project_number>.
JnsEd SmthE
P315 P422
PROJECT
Project_number <pi> PROJ_NO <M>
Project_description DESCRIPTION <M>
HR_scope HR_SCOPE <M>
is managed by
Project_manager
SUBPROJECT_of_PROJECT
of
manages
SUBPROJECT
Sequence_number <pi> SEQ_NO <M>
Subproject_description DESCRIPTION <M>
Starting_date CALENDAR_DAY <M>
Deadline CALENDAR_DAY
Subproject_leader
leads
EMPLOYEE
Employee_code <pi> EMP_CODE <M>
Surname NAME <M>
Deployment
is working on
Subproject <Sequence_number> of project <Project_number> is led by employee <Employee_code>.
3 2 1
P315 P315 P422
BsBg WndlA HkstJa
Employee <Employee_code> is working on subproject <Sequence_number> of project <Project_number>.
DoeJn RoeRch RoeRch
3 3 2
P315 P315 P422

**Figure 3.3** **Fact Oriented ERM diagram, with fact types and example populations.**

# 4 Drawing up an ERM Diagram

Sections 4.1 – 4.3 below are edited and adapted excerpts from the book in reference [1].

## 4.1 Fact Oriented Modeling

The most important hallmarks of Fact Oriented Modeling (abbreviated as FOM from here on) are:

- Use of *verbalizations of concrete examples of facts* as basis for modeling instead of a general description (the latter is always vague and incomplete).
- A good *procedure* (cookbook) telling you *how* to model instead of just *what* to model.
- Focus on *elementary facts* instead of attributes (half facts) or entities (clusters of facts).

Each point is explained further below.

### 4.1.1 Verbalizations of Concrete Examples of Facts

So what is meant by 'verbalizations of concrete examples of facts'? They are sentences in natural language used by people exchanging information about their work (they're called 'domain experts'), as in a telephone conversation. For example, "The price of a single ticket for performance 108 is $27.50." is such a verbalization. Such sentences that express one actual fact are used by domain experts many times daily and mean much more to them and to information modelers than abstract generalizations like 'performance tickets have prices'. They are a *big* help in making domain experts (pros in their own field, but usually unfamiliar with information modeling) and information modelers (pros in information modeling, but usually unfamiliar with the domain) understand each other.
Technically, 'performance 108' (one specific thing) is called an *instance*, whereas 'performance' (without a number) is a kind of thing, or a *type*. Similarly, 'Jungle Town' is an instance of the type 'performer', just as 'Fido' can be an instance of the type 'dog'. Practically speaking, type level descriptions like 'performance tickets have prices' ideally give you what you need to create *empty* tables in a database. That is fine in itself (the type level is crucial in FOM as well), but it leaves out the explicit link with the instance level communicational payload. By including instances in the modeling and providing concrete means to analyze instance level items (examples of real fact communication), it is assured that there is a full, well described match between instance level and type level.

So what makes FOM approaches Fact Oriented is that they take the instance level very seriously and incorporate it into their modeling. Other techniques of information modeling (for example ERM, or UML Class Diagrams) do not do this. This has implications for the logical technicalities underlying FOM, but it also, and perhaps primarily, reflects a *way of looking at information structures*. Using concrete and realistic examples is stressed because it is considered indispensable for analyzing domain communication and designing information systems in order to support business communication.

When communicating with business stakeholders (i.e. the people in the business who may not know much about designing information systems but surely know a lot about their business!), it is highly recommendable to keep a solid footing in the concrete world of the communication that really takes place in the domain at all times in everyday work and interactions. In other words, always complement your abstractions with plenty of *real or realistic examples*.

In short, using concrete examples has the following advantages:

- They make the *meaning* of the data much clearer than general descriptions.
- They are the common ground between domain experts and information modelers.
- They serve as the firm basis (instance level) on which all modeling decisions (type level) are based.
- They automatically bring out many modeling details (how a performer is identified, etc.).
- They make validation of the model easy.

This is the core idea behind Fact Oriented Modeling, but it is not the whole story. Without going into as much detail, here are some other features typical to FOM.

### 4.1.2 A Good Procedure for How to Model

Less typically in FOM, but still a strong characteristic of all FOM approaches, is that an extensive *process of modeling* is included in them. Many forms of modeling heavily emphasize the modeling language as such: its concepts (entities, relationships, classes, attributes, tables, keys, etc.), its syntax and semantics, its notation. While the FOM approaches are, without exception, fully worked out in this respect (including underlying math), they also provide an extensive, stepwise description of *how* to perform an analysis and create an information model (the modeling process or procedure). Of course, all information modeling techniques are clear in *what* a model should contain, but students and other interested parties are rather left in the cold if they aren't shown *how* this can be achieved.

### 4.1.3 Focus on Elementary Facts

Fact Oriented Modeling techniques consider *one complete fact at a time*. This contrasts sharply with other modeling techniques like ERM, UML class diagrams or Relational tables. These techniques use concepts like *attribute* (at best, half a fact) and *entity*, *class* or *table* (clusters of several facts). Historically, these concepts may have been quite useful from a software engineering perspective, but for considering information structures, they are rather arbitrary and do not match how language and logic are actually structured. By enforcing the attribute-entity distinction, a software engineering view (lens) is imposed on information structures. The scientists behind the creation of the FOM family of information modeling techniques did not approve of this as it needlessly fouls up the essence of what information modeling stands for: entities overshoot the mark and attributes fall short. The essential building blocks for information structures are solitary whole facts. One single complete fact is called an *elementary fact*: it is the smallest unit of information that can be meaningfully exchanged.

Using elementary facts has the following benefits:

- They are the simplest to model; if clusters are modeled in one go, the risk of errors is much greater. If incomplete facts are modeled, information loss might occur.
- They don't need extra concepts, like NULL values in tables, which you do need if you want to model clusters. So FOM techniques are simpler and leaner than other techniques, yet have the same modeling power.

- They can be easily combined in desired clusters later, whether on a computer interface screen (an elementary fact is either completely on it or not at all) or in entities, classes or table structures without making mistakes (even automatically).
- They prevent modelers and others from continually considering just fact *parts* (attributes). Seeing *pairs of columns* or attributes (or larger groups, depending on the number of components in the elementary facts) as indivisible units of information is far more fruitful.

### 4.1.4 Procedure to draw up an ERM diagram in a Fact Oriented Modeling way

The ‘cookbook’ for making an information model in ERM contains the following steps, which will each be discussed in a separate section of this chapter:

- Collecting concrete examples of facts (section 4.2)
- Verbalizing the concrete examples of facts (section 4.3)
- Sorting the verbalizations (section 4.4)
- Analyzing a fact type and adding the result to an ERM diagram (section 4.5)

## 4.2 Collecting: finding or drawing up concrete examples of facts

When you start to make an information model, it is often very helpful to make a rough (i.e. without too many details) process model first, for example in BPMN. Such a model serves to capture the important data flows and data stores as well, and can therefore be used as a starting point to collect concrete examples of the relevant information to be modeled. For every data flow or data store in the process model, samples of the actual data in the flow or store should be collected. This also helps to ensure that the information model will contain all the relevant data.

If other data sources turn up, during interviews or otherwise, samples should be collected immediately from these as well. It is usually easy to find concrete examples like this. However, sometimes the organization doesn't yet have concrete examples because the data concerned is to be used in the future and is not available yet. In such a case, the modeler and domain expert should make up realistic artificial concrete examples, as in the dialogue shown in figure 4.1.

**Domain expert:** *In the future, we'd like to offer discounts for different types of customers in the form of a percentage of the total cost for reservations, independent from the usual arrangement for season ticket prices.*

**Modeler:** *Can you give me an example of such a discount?*

**Domain expert:** *Well, something like VIP customers get a 10% reduction.*

**Modeler:** *OK, then you'd have to know which customers are VIPs as well, I suppose.*

**Domain expert:** *Yes, of course.*

**Modeler:** (Quickly scribbling two tables on a piece of paper) *Like this?*

| Customer type | reduction (%) |
|---|---|
| Ordinary | 0 |
| Frequent | 5 |
| VIP | 10 |

| Customer | is of type |
|---|---|
| 111 | VIP |
| 112 | Ordinary |
| 113 | Ordinary |

**Domain expert:** *Exactly! Good idea to use 'Ordinary', too; then we're sure we didn't forget to specify a type for a customer. Let's make that the default type.*

**Modeler:** *That's why I always like to make examples; they often help to make things a lot more clear.*

**Figure 4.1** **Drawing up concrete examples of facts that are new**

## 4.3 Verbalizing

Reasons why verbalizations are good to have are given first. This is followed by guidelines for how to verbalize properly. These guidelines will be discussed and illustrated in separate sections below.

- Verbalizing concrete examples prevents vagueness and misunderstandings.
- A good verbalization makes the *meaning* of the facts clear.
- Verbalizing clarifies how persons, things or concepts are *identified*.

**Figure 4.2 Why verbalize?**

One of the main problems with data models is that they are very hard to *validate*: How can domain experts verify that an information model is correct and give their approval? They usually cannot read such arcane models, and while the information modelers may be experts in data structures, they are usually not familiar with the data itself. If this problem is not dealt with, an information model can have serious flaws and nobody will notice until the implementation fails to work.

One reason is that information models contain a lot of *metadata* (i.e. statements *about* the data to be modeled), in terms of entity types, attributes, cardinalities and other such abstract data structure concepts, but they usually do not show the ‘ordinary’ data itself. This leaves the door wide open to vagueness (the abstract concepts seem quite plausible) and misunderstandings (people can interpret the abstract concepts differently and so talk at cross-purposes without realizing it).

Using verbalizations of concrete examples of the data to be modeled helps to avoid such problems in two important ways:

- The first way is that good verbalizations make the *meaning* of the data clear. Data examples are often in abbreviated form (diagrams, tables, etc.) in which much of their meaning is tacitly understood by the domain experts, but unknown to the information modeler. This ‘hidden’ semantics is brought out by verbalizing them. Because verbalizations are expressions of facts in natural language, they put both domain expert and modeler on common ground.
- The second way is that these verbalizations are the foundation on which the abstract information model will be built. All abstract data structures are gradually added to this concrete layer in a constant dialogue between the information modeler and the domain expert.

Another benefit from using such verbalizations is that several crucial information modeling issues are automatically and inextricably dealt with as well. For instance, how are all the persons/things/concepts that the facts refer to *identified*? Verbalizing a concrete fact forces you to use the actual numbers, names, codes, etc. that unambiguously make clear which individual things, persons or concepts the fact is about. Identification is one of the most important aspects of information models, and is so taken along automatically in a natural way.

If the verbalizations are incorporated lock, stock and barrel into the information model, the connection between the concrete ordinary data of the domain expert and the abstract metadata of the information modeler is always clear. This makes validation a lot easier and greatly reduces the risk of vagueness and misunderstandings.

Guidelines for how to verbalize concrete examples properly are:

A good verbalization is a grammatically correct complete sentence that:

- expresses a concrete fact that can be highlighted in the concrete examples,
- expresses exactly one fact (no clusters of facts, no half facts),
- does not refer to other sentences and
- expresses the meaning of this fact as clearly as possible.

**Figure 4.3 Guidelines for verbalizing**

All verbalizations must always be stated in complete sentences with correct spelling and grammar. This basic requirement is assumed to be fulfilled throughout this reader. Such sentences, which express exactly one fact, are called *fact expressions* from now on.

Figure 4.4 shows a few examples of the body weight data of members of the SlimSlim Fitness and Weight Reducing Club. These examples will be used to illustrate the guidelines for verbalizing.

**SlimSlim body weight data:**

| **Member:** | **M47 John** |
|---|---|
| **Date** | **Weight (lb)** |
| May 4, 2015 | 198.4 |
| May 5, 2015 | 197.9 |
| May 8, 2015 | 198.3 |
| May 9, 2015 | 197.9 |

| **Member:** | **M58 Lisa** |
|---|---|
| **Date** | **Weight (lb)** |
| April 29, 2015 | 201.3 |
| May 5, 2015 | 198.3 |
| May 10, 2015 | 196.6 |

**Figure 4.4 Examples of body weight data**

### 4.3.1 Express concrete facts that can be highlighted

Express only what you see *directly* in the concrete examples. You should be able to highlight every component in the fact expressions in the concrete examples with a marker.

| **Good verbalizations** | **Why good?** |
|---|---|
| `The first name of member M47 is John.` | Both concrete components 'M47' and 'John' are listed directly in the examples in Figure 4.4. |
| `Member M58 weighed 198.3 lb on May 5, 2015.` | All concrete components 'M58', '193.3 lb' and 'May 5, 2015' can be highlighted in the examples in Figure 4.4. |

**Figure 4.5 Good verbalizations**

| **Bad verbalizations** | **Why bad?** |
|---|---|
| `Weights are to be recorded for members.` | This fact expression expresses a generality, not a concrete fact. It is *metadata*: an expression *about* the data, not an expression *of* the data in Figure 4.4. |
| `Three measurements have been recorded for M58.` | The 'three' can't be highlighted directly in the examples in Figure 4.4. This is a conclusion drawn from looking at the basic facts and counting them. A very large number of other conclusions can be drawn from them as well (like: `Lisa lost 1.7 pounds in 5 days.`), and it would be absurd to state them all. That's what we have query languages for: to derive all sorts of interesting things from the basic facts. In information modeling we only model the indispensable basic facts. |

**Figure 4.6 Bad verbalizations**

### 4.3.2 Express exactly one fact: no clusters of facts, no incomplete facts

The goal of modeling *elementary* facts is to have facts that each express exactly one concrete fact. The reasons are:

- **Elementary facts are the simplest to model**. Modeling the smallest units of information one at a time prevents biting off more than you can chew. If you try to model clusters of facts in one go, the risk of making mistakes is much bigger. Moreover, you would need to use extra, more complicated concepts. For example, in FCO-IM (Fully Communication Oriented Information Modeling) only one concept is needed (namely Fact Type), whereas ERM (Entity-Relationship Modeling) needs *three* concepts (entity type, attribute, relationship type) to model the same information.
- **Elementary facts can be easily combined into desired clusters later**. It is easy to group elementary facts into a table or on a user interface screen. When these 'Lego' building blocks are all available, any desired compound structure can be built with them. Clustering elementary facts prematurely reduces flexibility in making clusters and introduces extra concepts (like NULL values) that are not needed at the conceptual level of modeling; NULL values can arise only when elementary facts are combined. In a single elementary fact, all components are always present, or the fact simply doesn't exist at all, so no NULL values are needed.
- **Considering incomplete facts (partial facts, half facts) leads to information loss**. See the examples in figure 4.7 below, and the counterexamples in figure 4.8.
- **An attribute is almost always an incomplete fact and not enough to use as an independent unit**. Using elementary facts prevents modelers from considering just fact *parts*. An attribute is only half a fact type at best, so it cannot be used as a separate building block. Seeing *pairs of columns* (or sets of three or more, depending on the number of components in an elementary fact type) as indivisible units of information is far more fruitful.

| Good verbalizations 👍 | Why good? |
|---|---|
| `The first name of member M47 is John.` | A complete elementary fact with two concrete components 'M47' and 'John'. |
| `Member M58 weighed 198.3 lb on May 5, 2015.` | A complete elementary fact with three concrete components 'M58', '198.3 lb' and 'May 5, 2015'. |

**Figure 4.7 Good elementary verbalizations**

| Bad verbalizations 👎 | Why bad? |
|---|---|
| `The first names of members M47 and M58 are John and Lisa.` | This expression combines two separate facts. It should be split into two separate fact expressions, each stating only one single fact:<br>`The first name of member M47 is John.`<br>`The first name of member M58 is Lisa.`<br>In addition, in the compound sentence it isn't entirely clear which member has which first name. Adding 'respectively' would solve that but not the bigger problem of expressing two facts in one go. |
| `S1: Member M58 was weighed on May 5, 2015.`<br>`S2: Member M58 was weighed on May 10, 2015.`<br>`S3: Member M58 weighed 196.6 lb.`<br>`S4: Member M58 weighed 198.3 lb.` | Fact expressions S1 and S2 seem OK, but if you look at S3 and S4, you can't tell *when* Lisa weighed 196.6 lb. Does fact expression S3 go together with S1 or S2, neither or both? S4 has the same problem. It is necessary to have the member, the weight and the date in one fact expression, otherwise the connection is lost and only partial facts remain.<br>`S14: Member M58 weighed 198.3 lb on May 5, 2015.`<br>`S23: Member M58 weighed 196.6 lb on May 10, 2015.`<br>In fact expressions S14 and S23, the correct members, weights and dates are combined, so no information is lost. S14 and S23 can replace S1, S2, S3 and S4.<br>It is also possible to keep S1, S2, S14 and S23, even though S1 and S2 give no extra information when S14 and S23 are there as well, but S3 and S4 surely are incomplete. |

**Figure 4.8 Bad verbalizations, compound or incomplete**

How can you tell whether a fact is elementary? Here's a test:

**Test for elementaryness**

Suppose you have a verbalization in the form of a sentence.
Count the number of components in this sentence (e.g. it is four).

| | |
|---|---|
| IF | you can split this sentence into two other complete sentences that *both* have fewer components (e.g. you can split it into two sentences, each having three components) |
| THEN | the sentence is not elementary. Split it into these smaller sentences and run the test again. |
| ELSE | the sentence is indeed elementary, and it is a genuine fact expression. |

**Figure 4.9 Test for elementariness**

Figure 4.8 above shows an example of applying this test. Suppose you have only S14, and you wonder whether it is elementary. S14 has three components (member, weight, date):

```
S14: Member M58 weighed 198.3 lb on May 5, 2015.
```

You can split it into S1 and S4, which *both* have fewer components (namely two).

```
S1: Member M58 was weighed on May 5, 2015.
S4: Member M58 weighed 198.3 lb.
```

But then you have an incomplete fact: S4. This is even clearer if you consider S14 and S23, split them into S1, S2, S3, and S4, and find you no longer can tell when Lisa weighed what.

Alternatively, you can split S14 into S1 and S4, but repair S4 by adding the date to it and so get:

```
S1: Member M58 was weighed on May 5, 2015.
S5: On May 10, 2015, member M58 weighed 198.3 lb.
```

But S5 is just the same as S14, only with a different order of the three components. So this is not a split after which *both* new sentences have fewer than three components, and S1 and S5 together fail the test condition.

All other possibilities of splitting sentence S14 into two sentences with less than three components also yield incomplete facts, so it is an elementary fact expression after all.

How can you tell whether a sentence that expresses a fact is incomplete? Here's a test:

**Test for completeness**

Suppose you have verbalized all the facts from a concrete example.
Try to reconstruct the example *from these verbalizations alone*.

| | |
|---|---|
| IF | you can reconstruct the example |
| THEN | these verbalizations do not express any incomplete facts, and therefore, if they are not compound (see the test for elementariness in figure 4.9), they are genuine fact expressions. |
| ELSE | the verbalizations are incomplete, and you have to verbalize the underlying facts with more components. |

**Figure 4.10 Test for completeness**

As an example of applying this test, suppose you have verbalized the right half of figure 4.4, and you wonder whether these verbalizations express complete facts or not. All verbalizations (sentences) are given in figure 4.11 for completeness, though only S1, S2, S3 and S4 would be enough to carry out the test. Now make an empty example that looks like the ones in Figure 4.4. There is no problem in reconstructing the part of the figure that corresponds to sentences S6, S0, S1 and S2 (see figure 4.11).

| | |
|---|---|
| `S0: Member M58 was weighed on April 29, 2015.`<br>`S1: Member M58 was weighed on May 5, 2015.`<br>`S2: Member M58 was weighed on May 10, 2015.`<br>`S3: Member M58 weighed 196.6 lb.`<br>`S4: Member M58 weighed 198.3 lb.`<br>`S5: Member M58 weighed 201.3 lb.`<br>`S6: The first name of member M58 is Lisa.` | **Member:** **M58 Lisa** |

| **Date** | **Weight (lb)** |
|---|---|
| April 29, 2015 | |
| May 5, 2015 | |
| May 10, 2015 | |

**Figure 4.11 Applying the test for completeness**

But when you try to add S3, S4 and S5, the problem becomes clear: You can't tell where the weights have to go. So the test shows that these last three sentences express incomplete facts. Figure 4.12 discusses the same sentences S1, S2, S3 and S4 more abstractly but arrives at the same conclusion.

### 4.3.3 Use sentences that do not depend on other sentences

Every verbalized fact must be a stand-alone sentence, which needs no other sentences to be understood.

| **Good verbalizations** | **Why good?** |
|---|---|
| `S1: The first name of member M47 is John.`<br>`S2: Member M47 weighed 198.3 lb on May 8, 2015.` | Two complete stand-alone verbalizations of elementary facts. |

**Figure 4.12 Good verbalizations**

| **Bad verbalizations** | **Why bad?** |
|---|---|
| `S1: The first name of member M47 is John.`<br>`S3: This member weighed 198.3 lb on May 8, 2015.` | Sentence S3 does not state explicitly which member it refers to. S3 needs S1 (as immediate predecessor) to know which member is meant, so it depends on S1. It should be corrected by explicitly identifying which member is meant (see S2 in figure 4.12). |

**Figure 4.13 Bad verbalizations, referring to other sentences**

### 4.3.4 Use sentences that make the meaning of the facts as clear as possible

The clearer a fact is expressed, the better. It is well worth the time and effort to search for phrases that make the *meaning* of the expressed facts as clear as possible. Here are a few general guidelines, which should be used judiciously as rules of thumb, not as strict laws, each followed by some examples of poor and better ways of verbalizing.

**G1: Where possible and convenient, use the name of the *kind of person/thing/concept* together with the concrete identifier**.

This provides context and so leads to a much better understanding. Units of physical quantities such as length (inches, feet, miles, etc.), weight (pounds, kilograms, metric tons, etc.) and time duration (seconds, minutes, years, etc.) must of course always be given.

| **Poor verbalizations** 👎 | **Better verbalizations** 👍 |
|---|---|
| M47 weighed 198.3 on May 8, 2015. | Member M47 weighed 198.3 lb on May 8, 2015. |
| 926R785 is held by Zwrtj. | Staff card 926R785 is held by employee Zwrtj.” |

**Figure 4.14 Poor and better verbalizations**

**G2 Use verbs that are as specific as possible.**

A common lazy way of verbalizing is to use ‘has’ in almost every expression. This only satisfies the requirement for grammatically correct sentences, but doesn’t make the meaning any clearer. It also makes the expressions indistinguishable on the metalevel (because they all look like “…has…”). Choose ‘has’ only if you really can’t think of anything else (try hard!).

| **Poor verbalizations** 👎 | **Better verbalizations** 👍 |
|---|---|
| Member M47 has first name John. | The first name of member M47 is John. |
| Member M47 has city Miami, Florida, USA. | Member M47 presently lives in the city of Miami, Florida, USA. |
| Employee E1257 has room A3.12. | Employee E1257 uses room A3.12 for experiments. |
| The car with license plate Colorado 521 ACJ has service date 3 March 2014. | The car with license plate Colorado 521 ACJ was last serviced on 3 March 2014. |

**Figure 4.15 Poor and better verbalizations**

## 4.4 Sorting and analyzing fact expressions

Section 4.4.1 defines three terms needed for analyzing fact expressions. Section 4.4.2 gives a detailed working procedure for how to build an ERM diagram from verbalizations of concrete examples of facts. The use of this procedure is illustrated in the subsequent sections 4.4.3 – 4.4.5. In these sections the procedure is applied to a small example: the diagram in figure 3.1 will be drawn up. These sections also explain the reasons for some of the instructions in the procedure, give further examples, etc. Students are advised to read sections 4.4.3 – 4.4.5 first, before studying this procedure in detail, and to consult section 4.4.2 mainly as reference and for overview.

### 4.4.1 Components, segments and predicates

**Component**

A component is a part of fact expressions of the same kind where the text can vary. For example, there are three components in the following four fact expressions that are all of the same kind (fact type), about the starting dates of subprojects. The text that does not vary is indicated with ditto marks (").

```
Subproject 1 of project P315 starts on 20160201.
    "      2  "    "    P315    "    " 20160201.
    "      3  "    "    P315    "    " 20160208.
    "      1  "    "    P422    "    " 20160101.
```

**Segment**

A *segment* is a part of a fact expression that represents an entity or an attribute. A segment can contain one or more components. For example: ‘subproject 3 of project P315’ is a segment that identifies a subproject, which is an entity. It contains two components. A segment is usually one connected sentence part, but can also be a set of disconnected sentence parts. For example: both sentences S1 and S2 below express the same fact, and the segment for a subproject is underlined in both of them. The dots indicate the gap in the segment for the subproject in S2. There is another segment for an attribute Starting_date (marked by a single underlining), which only contains one component.

```
S1: Subproject 1 of project P315 starts on 20160201.
    ET SUBPROJECT                          Att Starting_date

S2: In project P315, 20160201 is the starting date of subproject 1.
       ET SUBPROJECT
                     ________
                     Att Starting_date
```

**Predicate**

A predicate is a sentence type in which all components are replaced by blanks (see section 3.2.1).

```
Predicate for S1:  Subproject  <Sequence_number>  of  project  <Project_number>
                   starts on <Starting_date>.
Predicate for S2:  In project <Project_number>, <Starting_date> is the starting
                   date of subproject <Sequence_number>.
```

The blanks, between the pointed brackets (‘<’ and ‘>’), contain the names of the attributes from which concrete values are to be filled in, to obtain a concrete fact expression (one single fact) So predicates are at the type level, and fact expressions are at the instance level.

### 4.4.2 Procedure for sorting and analyzing fact expressions and building an ERM diagram

Here is a short working procedure to draw up an ERM Diagram from verbalizations of concrete examples of facts. It deals with the most common cases first, and briefly treats unusual or more complex cases later. A more detailed procedure that covers all cases systematically and completely is given in Appendix A.

Students are advised to consult the short procedure here regularly while reading section 4.4.4 and while doing their own analyses, and only turn to Appendix A for specific details if the short procedure doesn't help them enough.

**Starting point:** a set of verbalizations of concrete examples of facts, approved by the domain experts (see section 4.3 for how to verbalize).

**Step 1: Sort and order the fact expressions.**
The verbalizations are sorted into fact types and ordered: fact types with few components first, and fact types with many components last.

1 **Sort the fact expressions into fact types and order these fact types.**
   a **Place sentences of the same kind together in a group (fact type).**
   b **Count the number of components (see section 4.4.1) in each fact type**.
   c **Arrange the fact types in ascending order of the number of components.**
   Order the fact types:
      first the ones with one component (if any),
      then the ones with two components (if any),
      then the ones with three components (if any),
      and so on.
   The order of fact types with the same number of components doesn't matter.

**Steps 2-5: Analyzing the fact types.** One by one, each fact type will be analyzed, starting with the fact types with the smallest number of components, and ending with the fact types with the most components. Steps 2-5 below are to be carried out completely for one fact type, before the next fact type is considered.

**Step 2: Determine segments.**

First, the segments (see section 4.4.1) must be determined. There is a firm rule about segments for analyzing fact types in ERM:

**Segment rule:**
**In ERM, a fact type can contain either two segments, or only one segment, never three or more.**

No matter how many components a fact type has, two segments can be marked at most (only one segment is actually a rare case, see section 4.4.5 for a few examples, and Appendix A, step 2 b i). If there are more than two components, there are several possibilities to choose two segments. By treating the fact types with the fewest components first, this should cause no serious difficulties, because components that belong together then have already been determined earlier and can be easily recognized.

2 **Underline two segments, associate one with an ET and the other with an ET or Att.**
- a **Underline two segments** (if there is only one segment: see Appendix A, step 2 b i)
  There are now two possibilities: either both segments belong to an ET, or one segment belongs to an ET and the other to an Att of this same ET.
- b **If one segment belongs to an ET, and the other to an Att of this ET:**
  - i **Write 'ET' and the ET-name directly below the underlining of the ET-segment.**
  - ii **Write 'Att' and the Att-name directly below the underlining of the Att-segment.**
  - iii **Do step 3 below for the ET found here (determine the identifier).**
  - iv **Give the predicate as the last line in the analysis.**
  - v **In step 4, determine whether the Att is <M> or not, and assign a domain to it.**
- c **If both segments belong to an ET:**
  - i **Write 'ET' and the ET-name directly below the underlining of both segments.**
  - ii **Do step 3 below for both ETs found here (determine the identifiers).**
  - iii **Add a Relationship Type between the two ETs.**
    **Give the result in the format:**
    **RT <RTname> between <ETname1> and <ETname2>.**
  - iv **Give the predicate as the last line in the analysis.**
  - v **In step 4, determine all four cardinalities carefully.**

**Step 3: Determine the identification of ETs.**

The first thing to consider as soon as an ET is encountered, is how it is identified.
The following possibilities will be discussed (starting with the simplest case and building up in complexity):

- The ET was already found earlier, so its identifier is already known: step 3a.
- The ET is new, and strong (see section 3.1.6): steps 3b and 3c.
- The ET is new, and weak (see section 3.1.7): steps 3b and 3d.

Clearly step 3d is the most complex, and students are advised to study examples of its application carefully in section 4.4.4 and in other answers to exercises.

**3 For each ET found in the analysis**

- **a IF the ET was already found earlier**
  - **i Write 'MATCH' directly below the ET-name**, and we're done for this ET.

The following steps 3b – 3d are therefore only for new ETs:

- **b Determine how the ET is identified.**
- **c IF the ET is identified <u>only</u> by one or more of its <u>own</u> attributes**
  - **i Write 'ID: Att' and the Att-name**
    **followed by '+ Att' and the Att-name of any other identifying attributes**
    **directly below the ET-name,** and we're done for this ET.
  - **ii All these attributes are mandatory (<M>). In step 4, determine their domains.**
- **d IF the ET is identified by at least one other ET (that is: it is a weak ET)**
  then the identifier in general consists of one or more ETs,
  and zero, one or more of its own Attributes.
  - **i Write 'ID: ET' and the ET-name**
    **followed by '+ ET' and the ET-name of any other identifying ETs**
    **followed by '+ Att' and the Att-name of any other identifying attributes**
    **directly below the ET-name.**
  - **ii Any attribute found here is mandatory (<M>). In step 4, determine their domains.**

  **For each ET in the identifier found:**
  - **ii Do step 3 for these ETs as well (determine their identifiers).**
  - **iii Add a Relationship Type between the new ET and the identifying ET.**
    **This RT is <u>dependent</u> at the side of the new RT. Give the result in the format:**
    **RT <RTname> between <new ETname>(dependent) and <ETname2>.**
    **In step 4, determine the two cardinalities at the side of the new ET carefully (the other two are both ONE automatically, see section 3.1.10).**

**Step 4: Determine constraints for Atts and RTs.**
For all attributes, determine the domain and whether they are mandatory. For relationship types, determine all the cardinalities.

**4 Determine domains, <M> and cardinalities**

- **a For each attribute found:**
  - **i Determine the domain.**
  - **ii Determine whether it is <M> or not.**
- **b For each RT found:**
  - **i Determine all four cardinalities carefully.**
  - **ii Account for each cardinality in the documentation.**

**Step 5: Adding the results of the analysis to the ERM diagram.**

This should be easy to do, and is not explained in detail here. See the examples in section 4.4.4, and Appendix A, step 5 for details.

**5 Add the results to the diagram**

- **a Add the new ETs, Atts and RTs to the diagram, together with their constraints.**
- **b Add at least one role (see section 3.1.11) to each RT (two roles is also OK).**
- **c Add the predicate and an example population if this helps to clarify the meaning (semantics) of the modeled information.**

### 4.4.3 Example used in section 4.4.4

A company carries out projects for customers. It wants a database to store all the project management data in. A small part will be considered here, which concerns the definition and staffing of a new project. See figure 4.16, which shows the process model for initiating a project.

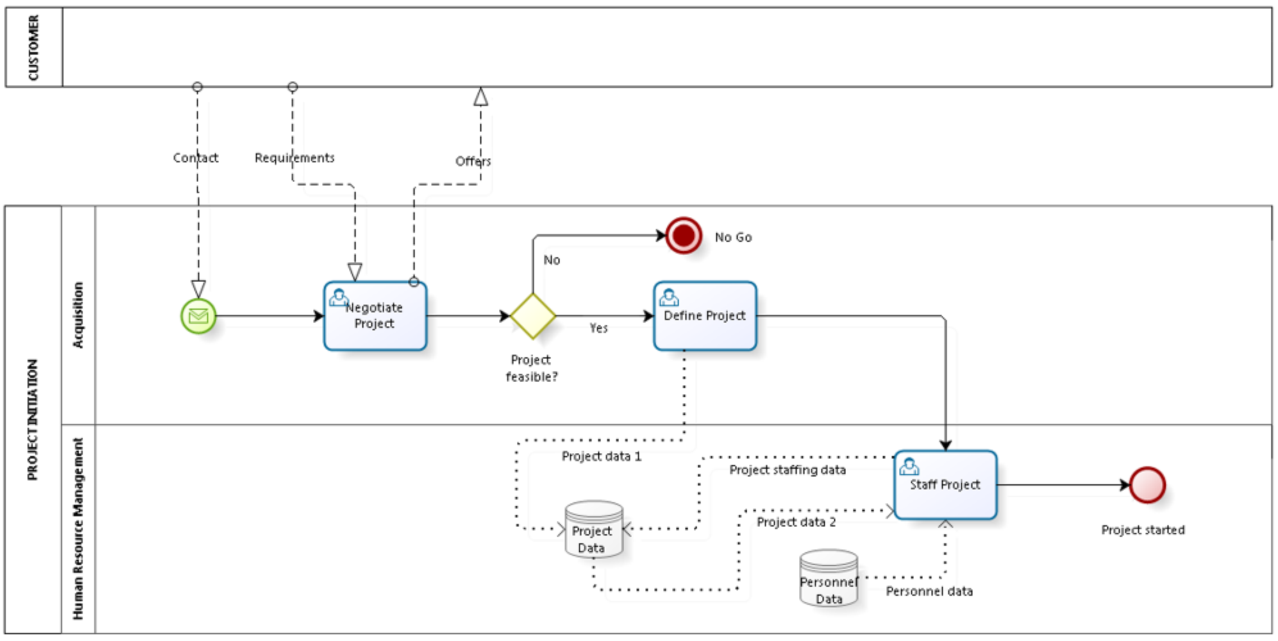


**Figure 4.16 Example process model**

The process PROJECT INITIATION starts with a contact with a customer. The acquisition department then establishes the requirements and negotiates offers with the customer (task Negotiate Project). If an agreement is reached, the project is defined (task Define Project), including the structure (subprojects, deadlines, etc.) and the result stored (flow Project data 1 and store Project Data). The HRM department then sets out to assign employees to work on the new project (task Staff Project).

In section 4.4, only (parts of) the flows Project data 2, Personnel data and Project staffing data are considered. Concrete examples of facts from these flows are given in figures 4.17 – 4.19. They actually contain many more facts, but only the ones to be used in this section are shown.

Verbalizations of the facts in figures 4.17-4.19 are given below figure 4.19. These verbalizations were approved (validated) by the domain experts, who have declared that the fact expressions capture the meaning of the data very well.

| **PROJECT** | **Description** | **From** | **To** | **…** |
|---|---|---|---|---|
| PROJECT P315 | Update homepage Treasury Bank | … | … | |
| 0.4 FTE | | | | |
| 1 | Elicit requirements | 20160201 | 20160205 | |
| 2 | Improve firewall | 20160201 | 20160301 | |
| 3 | Add new functionality | 20160208 | 20160301 | |
| … | … | | | |
| PROJECT P422 | Build DWH for OnlineHaberdashery | … | … | |
| 3.7 FTE | | | | |
| 1 | Determine scope | 20160101 | 20160201 | |
| … | … | | | |

**Figure 4.17 Examples from flow Project data 2**

| **Employee code** | **Surname** | **First name** | **Date of Birth** | **Date of entrance** | **…** |
|---|---|---|---|---|---|
| DoeJn | Doe | … | … | … | … |
| RoeRch | Roberts | … | … | … | … |
| InsEd | Insula | … | … | … | … |
| SmthE | Smith | … | | | |
| BsBg | Bose | … | | | |
| WndlA | Wendle | … | | | |
| HkstJa | Hukster | … | | | |
| … | … | … | | | |

**Figure 4.18 Examples from flow Personnel data**

| **PROJECT** | **Description** | **Leader** | **Coworker** | **…** |
|---|---|---|---|---|
| PROJECT P315 | Update homepage Treasury Bank | InsEd | -- | |
| 1 | Elicit requirements | InsEd | … | |
| 2 | Improve firewall | WndlA | … | |
| 3 | Add new functionality | BsBg | RoeRch<br>DoeJn | |
| … | … | | | |
| PROJECT P422 | Build DWH for OnlineHaberdashery | SmthE | -- | |
| 1 | Determine scope | HkstJa | … | |
| 2 | … | … | RoeRch | |
| … | … | | | |

**Figure 4.19 Examples from flow Project staffing data**

Verbalizations of the concrete examples:

From figure 4.17:
`Project P315 is described as: Update homepage Treasury Bank. The estimated number of man-years required to complete project P315 is 0.4. The description of subproject 1 of project P315 is: Elicit requirements. Subproject 1 of project P315 starts on 20160201. Subproject 1 of project P315 must be completed by 20160205. The description of subproject 2 of project P315 is: Improve firewall. Subproject 2 of project P315 starts on 20160201. Subproject 2 of project P315 must be completed by 20160301. The description of subproject 3 of project P315 is: Add new functionality. Subproject 3 of project P315 starts on 20160208. Subproject 3 of project P315 must be completed by 20160301. Project P422 is described as: Build DWH for OnlineHaberdashery. The estimated number of man-years required to complete project P422 is 3.7. The description of subproject 1 of project P422 is: Determine scope. Subproject 1 of project P422 starts on 20160101. Subproject 1 of project P422 must be completed by 20160201.`

From figure 4.18:
`The surname of employee DoeJn is Doe. The surname of employee RoeRch is Roberts.`
Other facts from this example have similar verbalizations.

From figure 4.19:

The descriptions have already been verbalized above and are not repeated here.
`Employee InsEd manages project P315. Subproject 1 of project P315 is led by employee InsEd. Subproject 2 of project P315 is led by employee WndlA. Subproject 3 of project P315 is led by employee BsBg. Employee RoeRch is working on subproject 3 of project P315. Employee DoeJn is working on subproject 3 of project P315. Employee SmthE manages project P422. Subproject 1 of project P422 is led by employee HkstJa. Employee RoeRch is working on subproject 2 of project P422.`

### 4.4.4 Sorting and analyzing the fact expressions

**Sorting**

Here are the results of step 1 of the procedure (see section 4.4.2). To highlight the number of components, the fixed text parts in each fact type are not repeated but indicated with ditto marks ("). The fact types have been given short names (like 'FT2') for easy reference.

There are no fact types with only one component.

**Fact types with two components**

```
FT1
Project P315 is described as: Update homepage Treasury Bank.
   "    P422  "     "      "  Build DWH for OnlineHaberdashery.
```

```
FT2
The estimated number of man-years required to complete project P315 is 0.4.
  "    "       "    "    "    "      "      "    "        "    P422  " 3.7.
```

```
FT3
The surname of employee DoeJn is Doe.
 "     "     "    "     RoeRch " Roberts.
```

```
FT4
Employee InsEd manages project P315.
   "     SmthE    "       "    P422.
```

**Fact types with three components**

```
FT5
The description of subproject 1 of project P315 is: Elicit requirements.
 "      "        "     "      2  "    "    P315  "  Improve firewall.
 "      "        "     "      3  "    "    P315  "  Add new functionality.
 "      "        "     "      1  "    "    P422  "  Determine scope.
```

```
FT6
Subproject 1 of project P315 starts on 20160201.
    "      2  "    "    P315    "    " 20160201.
    "      3  "    "    P315    "    " 20160208.
    "      1  "    "    P422    "    " 20160101.
```

```
FT7
Subproject 1 of project P315 must be completed by 20160205.
    "      2  "    "    P315   "   "     "      " 20160301.
    "      3  "    "    P315   "   "     "      ".20160301.
    "      1  "    "    P422   "   "     "      ".20160201.
```

```
FT8
Subproject 1 of project P315 is led by employee InsEd.
    "      2  "    "    P315  "  "   "    "     WndlA.
    "      3  "    "    P315  "  "   "    "     BsBg.
    "      1  "    "    P422  "  "   "    "     HkstJa.
```

```
FT9
Employee RoeRch is working on subproject 3 of project P315.
   "     DoeJn   "    "     "     "      3  "    "    P315.
   "     RoeRch  "    "     "     "      2  "    "    P422.
```

**Analyzing**

All the fact types above are analyzed one by one below. These analyses make frequent references to the steps in the working procedure in section 4.4.2 (from here on referred to as ‘the procedure’), and include further explanatory remarks. These elaborate references and remarks are only for didactical purposes here, in practice the bare analyses will suffice (i.e.: everything written in `Lucida Console size 10 font` in this section. See also worked answers to exercises (not in this reader).

Please note that all analyses below can only be carried out in elaborate dialogues with domain experts. Modelers need input from domain experts to determine identifiers, choose good names for Atts or ETs, determine <M> constraints and cardinalities of RTs, etc. etc.

**Analysis of FT1**

```
Project P315 is described as: Update homepage Treasury Bank.
   "    P422  "     "       "  Build DWH for OnlineHaberdashery.
```

The first component identifies a project, and the second is a description of a project. So there are two segments, one for an ET PROJECT and the other for its Att Project_description. Meaningful names are written below the segments. (If you follow the procedure in section 4.4.2: steps 2a and 2 b i-ii apply.)

```
Project P315 is described as: Update homepage Treasury Bank.
   "    P422  "     "       "  Build DWH for OnlineHaberdashery.
ET PROJECT                     Att Project_description
```

The ET is new, so the <pi> of ET PROJECT must be determined. This must be a strong ET: ‘P315’ and ‘P422’ are project numbers that identify a project (the modeler determines this together with a domain expert or at least verifies this with a domain expert), so a <pi>-Att Project_number is given. The predicate is given to complete the analysis. (In the procedure: steps 2 b iii and 3 b-c apply (it is a new strong ET), followed by step 2 b iv.)

```
Project P315 is described as: Update homepage Treasury Bank.
   "    P422  "     "       "  Build DWH for OnlineHaberdashery.
ET PROJECT                     Att Project_description
ID: Att Project_number

Project <Project_number> is described as: <Project_description>.
```

The ERM diagram resulting from this analysis is easily made: the fact type is modeled as two Atts of an ET (new), one of which is the <pi>: a <pi>-Att pair. According to the domain expert, every project must have a description, so the other Att is <M> as well. Domains were also assigned to the Atts. The result is shown in figure 4.20. The predicate was not added to the diagram because the meaning of the two Atts is obvious. (In the procedure: steps 4a and 5a apply, and step 5c is omitted.)

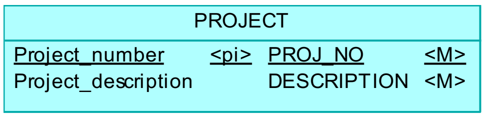

PROJECT
Project_number <pi> PROJ_NO <M>
Project_description DESCRIPTION <M>

**Figure 4.20** **Result of analyzing FT1**

## Analysis of FT2

```
The estimated number of man-years required to complete project P315 is 0.4.
  "    "       "     "    "     "    "       "     "        "      P422   "  3.7.
                                                             ET PROJECT   Att HR_scope
                                                             MATCH
```

```
The estimated number of man-years required to complete project <Project_number>
is <HRscope>.
```

The first component is again a project, and the second is clearly another Att of a project. The domain expert informs the modeler that the ususal jargon for a number of man-years is ‘HRscope’, from ‘Human Resource scope’. So an Att HR_scope is modeled. The new attribute can be easily added to the diagram. The fact type is modeled as two Atts of an ET (old), one of which is the <pi>. The domain expert declares that the HRscope must be known for all projects, so the Att is <M>. See figure 4.21. The predicate and an example population were added to the diagram because the meaning of the new attribute is not obvious at all.

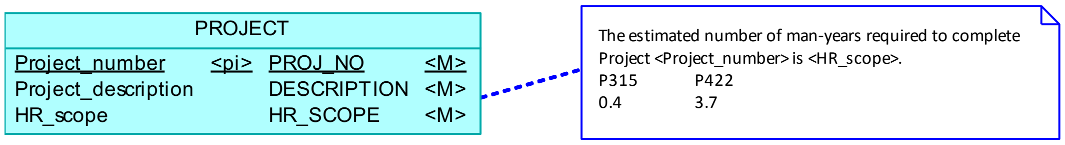

PROJECT
Project_number <pi> PROJ_NO <M>
Project_description DESCRIPTION <M>
HR_scope HR_SCOPE <M>
The estimated number of man-years required to complete Project <Project_number> is <HR_scope>.
P315 P422
0.4 3.7

**Figure 4.21 Result of analyzing FT2**

(If you follow the procedure in section 4.4.2: steps 2a, 2 b i-iii, 3a, 2 b iv, 4a, 5a and 5c apply.)

## Analysis of FT3

```
The surname of employee DoeJn is Doe.
 "      "     "     "      RoeRch "  Roberts.
                ET EMPLOYEE       Att Surname
                ID: Att Employee_code
```

```
The surname of employee <Employee_code> is <Surname>.
```

The analysis is analogous to that of FT1. The first component identifies an employee (the domain expert asserts every employee has a unique employee code), and the second is his/her surname. So there are two segments, for an ET EMPLOYEE with its Att Surname. The fact type is modeled as two Atts of an ET (new), one of which is the <pi>: another <pi>-Att pair. The domain expert agrees that the surname must be known for all employees, so the Att is <M>. No predicate was added because the meaning is obvious. See the result in figure 4.22. (If you follow the procedure in section 4.4.2: steps 2a, 2 b i-iii, 3b-c, 2 b iv, 4a and 5a apply.)

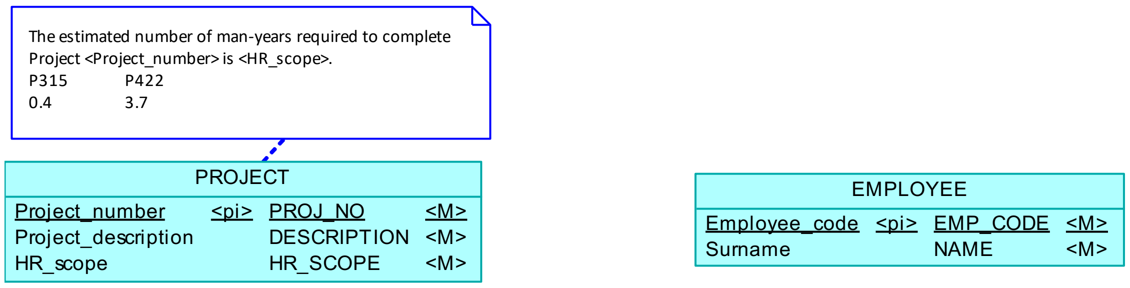

The estimated number of man-years required to complete Project <Project_number> is <HR_scope>.
P315 P422
0.4 3.7
PROJECT
Project_number <pi> PROJ_NO <M>
Project_description DESCRIPTION <M>
HR_scope HR_SCOPE <M>
EMPLOYEE
Employee_code <pi> EMP_CODE <M>
Surname NAME <M>

**Figure 4.22 Result of analyzing FT3**

**Analysis of FT4**

```
Employee InsEd manages project P315.
   "     SmthE     "       "    P422.
ET EMPLOYEE            ET PROJECT
MATCH                  MATCH
```

In this fact type, the two previously modeled ETs are easily recognized. So here the modeler chooses option 2c in the procedure in section 4.4.2: two ET segments. Both ETs are old and strong, so a RT without dependencies is added, and its name is chosen as clearly as possible. The cardinalities are determined carefully in a dialogue with the domain expert (see the result in figure 4.23). The predicate is written down to complete the analysis, and the results are added to the diagram (see figure 4.23). The predicate and an example population were added as well. (This is not really necessary, because the roles also make the meaning quite clear. However, for didactical purposes it is nice to show here how a RT can have a predicate and population.)

```
Employee InsEd manages project P315.
   "     SmthE     "       "    P422.
ET EMPLOYEE            ET PROJECT
MATCH                  MATCH
```

RT Project_manager between EMPLOYEE and PROJECT.

```
Employee <Employee_code> manages project <Project_number>.
```

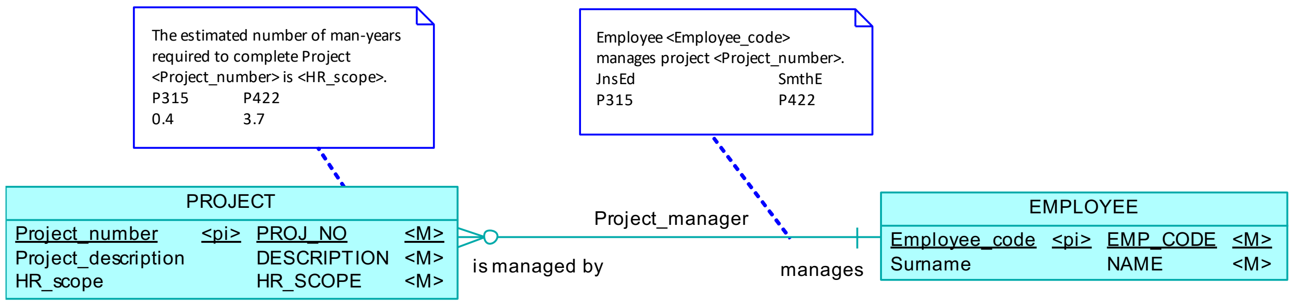

The estimated number of man-years required to complete Project <Project_number> is <HR_scope>.
P315 P422
0.4 3.7
Employee <Employee_code> manages project <Project_number>.
JnsEd SmthE
P315 P422
PROJECT
Project_number <pi> PROJ_NO <M>
Project_description DESCRIPTION <M>
HR_scope HR_SCOPE <M>
Project_manager
is managed by
manages
EMPLOYEE
Employee_code <pi> EMP_CODE <M>
Surname NAME <M>

**Figure 4.23** **Result of analyzing FT4**

(If you follow the procedure in section 4.4.2: steps 2a, 2 c i-ii, 3a, 2 c iii-iv, 4a-b, and 5a-c apply.)

**Analysis of FT5**

```
The description of subproject 1 of project P315 is: Elicit requirements.
 "      "        "     "     2  "     "   P315  "  Improve firewall.
 "      "        "     "     3  "     "   P315  "  Add new functionality.
 "      "        "     "     1  "     "   P422  "  Determine scope.
                   ET SUBPROJECT                   Att Subproject_description
```

This is the first fact type with three components to be analyzed. It states what the description of a subproject is. So the modeler marks two segments, one for an ET SUBPPROJECT, one for an Att Subproject_description.

The ET segment contains the first two components, because they *together* identify a subproject. Clearly the first component (with the numbers 1, 2 and 3) by itself cannot identify a subproject (the number ‘1’ is used in two different subprojects): the project it belongs to is needed as well. The SUBPROJECT segment contains an already familiar part (ET PROJECT was modeled earlier) and a new part with numbers 1, 2 etc. The domain expert agrees these are sequence numbers used to distinguish the several subprojects of a project, and they start with ‘1’ within each project. So the identifier of ET SUBPROJECT is a new Att Sequence_number together with the <pi> of ET PROJECT: SUBPROJECT is a weak ET that needs the <pi> of PROJECT for its own identification.

```
The description of subproject 1 of project P315 is: Elicit requirements.
 "      "        "     "     2  "     "   P315  "  Improve firewall.
 "      "        "     "     3  "     "   P315  "  Add new functionality.
 "      "        "     "     1  "     "   P422  "  Determine scope.
                   ET SUBPROJECT                   Att Subproject_description
                   ID: Att Sequence_number + ET PROJECT
                                              MATCH
```

Because SUBPROJECT is a weak ET, a RT between SUBPROJECT and PROJECT must be added, which has a dependency at the side of SUBPROJECT. This completes the analysis, and the predicate is written down to finalize it.

```
The description of subproject 1 of project P315 is: Elicit requirements.
 "      "        "     "     2  "     "   P315  "  Improve firewall.
 "      "        "     "     3  "     "   P315  "  Add new functionality.
 "      "        "     "     1  "     "   P422  "  Determine scope.
                   ET SUBPROJECT                   Att Subproject_description
                   ID: Att Sequence_number + ET PROJECT
                                              MATCH
RT SUBPROJECT_of_PROJECT between SUBPROJECT(dependent) and PROJECT

The description of subproject <Sequence_number> of project <Project_number> is:
<Subproject_description>.
```

In the diagram, a new weak ET SUBPROJECT must be added, connected to PROJECT with a dependent RT. The two cardinalities at the child ET side are determined carefully in a dialogue with the domain expert. There are two new attributes: Sequence_number (part of the <pi> and therefore <M>) and Subproject_description. The domain expert declares that every subproject must have a description, so this Att is <M> as well.

So this fact type is modeled as a <pi>-Att pair, with a complex <pi> because of the weak ET. No predicate was added to the diagram for this fact type because the meaning is clear enough.

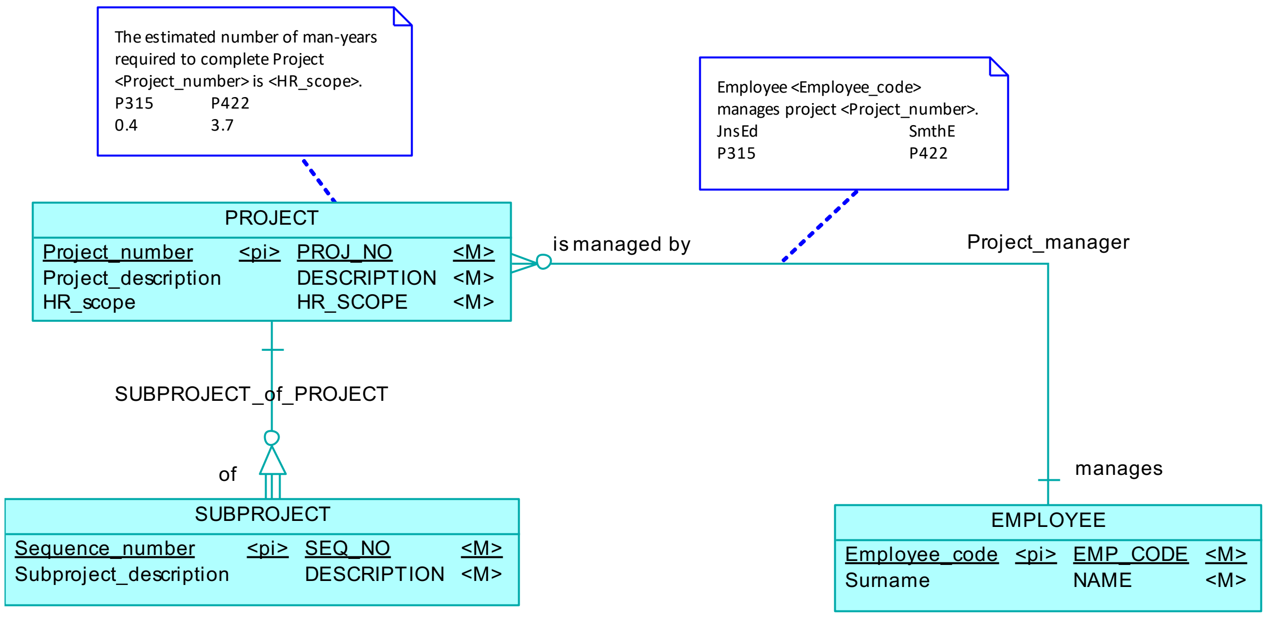

The estimated number of man-years required to complete Project <Project_number> is <HR_scope>.
P315 P422
0.4 3.7
Employee <Employee_code> manages project <Project_number>.
JnsEd SmthE
P315 P422
PROJECT
Project_number <pi> PROJ_NO <M>
Project_description DESCRIPTION <M>
HR_scope HR_SCOPE <M>
is managed by
Project_manager
manages
SUBPROJECT_of_PROJECT
of
SUBPROJECT
Sequence_number <pi> SEQ_NO <M>
Subproject_description DESCRIPTION <M>
EMPLOYEE
Employee_code <pi> EMP_CODE <M>
Surname NAME <M>

**Figure 4.24 Result of analyzing FT5**

(If you follow the procedure in section 4.4.2: steps 2a, 2 b i-iii, 3b, 3d, 2 b iv, 4, and 5a-b apply.)

## Analysis of FT6 and FT7

The analysis of these fact types is completely analogous to that of FT5, except that ET SUBPROJECT now already exists. So the analyses are given here without further comment. The fact types are both modeled as <pi>-Att pairs. Constraints domains and names of attributes were all determined in dialogues with domain experts. See the results in figure 4.25.

```
FT6
Subproject 1 of project P315 starts on 20160201.
    "      2  "    "    P315    "    " 20160201.
    "      3  "    "    P315    "    " 20160208.
    "      1  "    "    P422    "    " 20160101.
ET SUBPROJECT                          Att Starting_date
MATCH

Subproject   <Sequence_number>   of   project   <Project_number>   starts   on
<Starting_date>.
```

```
FT7
Subproject 1 of project P315 must be completed by 20160205.
    "      2  "    "    P315  "   "     "      " 20160301.
    "      3  "    "    P315  "   "     "      ".20160301.
    "      1  "    "    P422  "   "     "      ".20160201.
ET SUBPROJECT                                     Att Deadline
MATCH

Subproject <Sequence_number> of project <Project_number> must be completed by
<Deadline>.
```

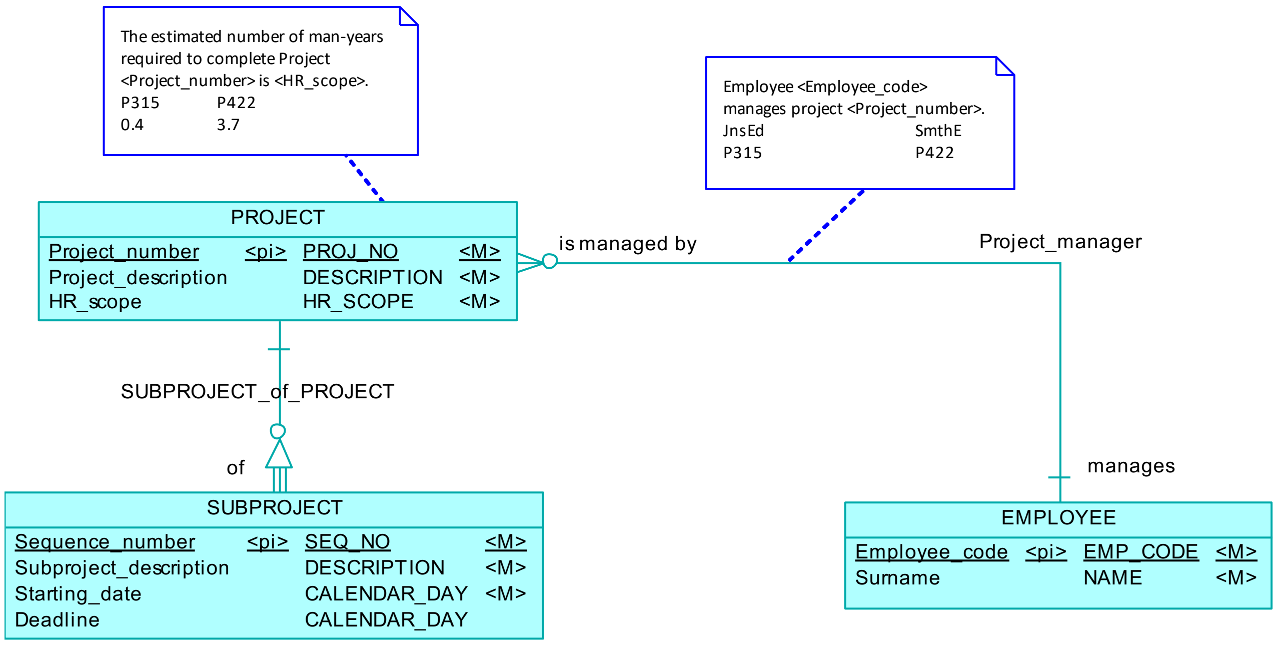

The estimated number of man-years required to complete Project <Project_number> is <HR_scope>.
P315 P422
0.4 3.7
Employee <Employee_code> manages project <Project_number>.
JnsEd SmthE
P315 P422
PROJECT
Project_number <pi> PROJ_NO <M>
Project_description DESCRIPTION <M>
HR_scope HR_SCOPE <M>
is managed by
Project_manager
SUBPROJECT_of_PROJECT
of
manages
SUBPROJECT
Sequence_number <pi> SEQ_NO <M>
Subproject_description DESCRIPTION <M>
Starting_date CALENDAR_DAY <M>
Deadline CALENDAR_DAY
EMPLOYEE
Employee_code <pi> EMP_CODE <M>
Surname NAME <M>

**Figure 4.25 Result of analyzing FT6 and FT7**

### Analysis of FT8

There are three components in this fact type, and the previously determined ETs SUBPROJECT and EMPLOYEE are easily recognized. So the two segments are both for an ET, and the fact type is modeled as a RT (without dependencies) between these ETs.

```
Subproject 1 of project P315 is led by employee InsEd.
    "      2  "    "    P315  "  "   "     "     WndlA.
    "      3  "    "    P315  "  "   "     "     BsBg.
    "      1  "    "    P422  "  "   "     "     HkstJa.
ET SUBPROJECT                          ET EMPLOYEE
MATCH                                  MATCH
```

```
RT Subproject_leader between SUBPROJECT and EMPLOYEE
```

```
Subproject <Sequence_number> of project <Project_number> is led by employee
<Employee_code>.
```

The four cardinalities were determined together with a domain expert, and the predicate with a small example population was added to the diagram to make the components more clear. See figure 4.26 for the results. (If you follow the procedure in section 4.4.2: steps 2a, 2 c i-ii, 3a, 2 c iii-iv, 4 and 5 apply.)

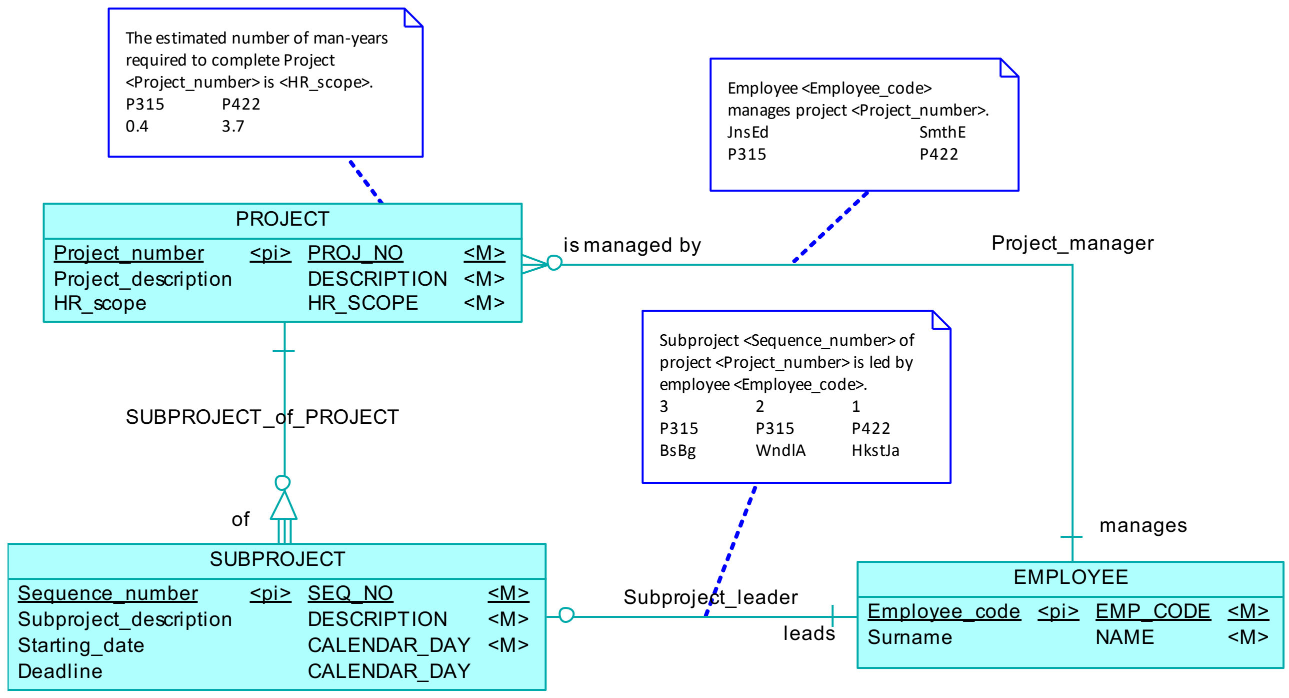

The estimated number of man-years required to complete Project <Project_number> is <HR_scope>.
P315 P422
0.4 3.7
Employee <Employee_code> manages project <Project_number>.
JnsEd SmthE
P315 P422
PROJECT
Project_number <pi> PROJ_NO <M>
Project_description DESCRIPTION <M>
HR_scope HR_SCOPE <M>
is managed by
Project_manager
Subproject <Sequence_number> of project <Project_number> is led by employee <Employee_code>.
3 2 1
P315 P315 P422
BsBg WndlA HkstJa
SUBPROJECT_of_PROJECT
of
manages
SUBPROJECT
Sequence_number <pi> SEQ_NO <M>
Subproject_description DESCRIPTION <M>
Starting_date CALENDAR_DAY <M>
Deadline CALENDAR_DAY
Subproject_leader
leads
EMPLOYEE
Employee_code <pi> EMP_CODE <M>
Surname NAME <M>

**Figure 4.26** **Result of analyzing FT8**

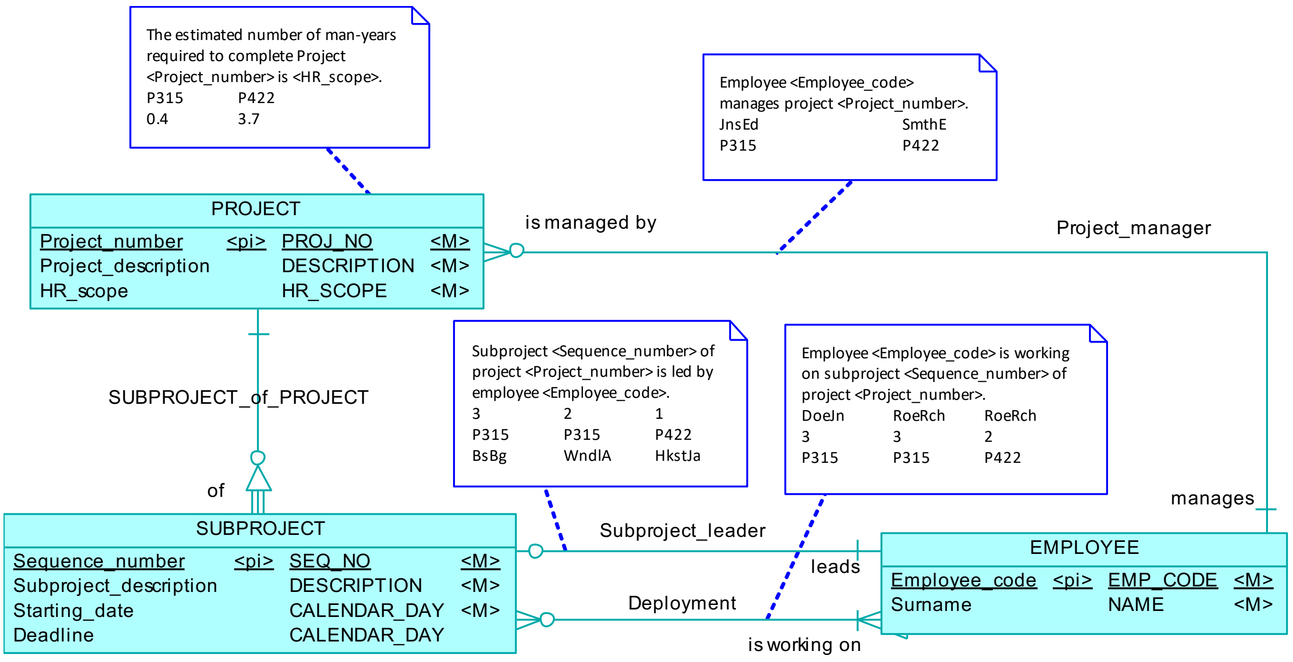
## Analysis of FT9

The analysis of this fact type is completely analogous to that of FT8. So it is given here without further comment. Note that the fact type is modeled as a second RT between the same two ETs, and has different cardinalities. The predicate and a small example population were added as well. See the result in figure 4.27.

```
Employee RoeRch is working on subproject 3 of project P315.
   "     DoeJn   "     "    "      "     3  "    "    P315.
   "     RoeRch  "     "    "      "     2  "    "    P422.
ET EMPLOYEE                    ET SUBPROJECT
MATCH                          MATCH
```

```
RT Deployment between SUBPROJECT and EMPLOYEE
```

```
Employee <Employee_code> is working on subproject <Sequence_number> of project
<Project_number>.
```


PROJECT
Project_number <pi> PROJ_NO <M>
Project_description DESCRIPTION <M>
HR_scope HR_SCOPE <M>
is managed by
Project_manager
manages
SUBPROJECT_of_PROJECT
of
SUBPROJECT
Sequence_number <pi> SEQ_NO <M>
Subproject_description DESCRIPTION <M>
Starting_date CALENDAR_DAY <M>
Deadline CALENDAR_DAY
Subproject_leader
leads
Deployment
is working on
EMPLOYEE
Employee_code <pi> EMP_CODE <M>
Surname NAME <M>

The estimated number of man-years required to complete Project <Project_number> is <HR_scope>.

| | |
|---|---|
| P315 | P422 |
| 0.4 | 3.7 |

Employee <Employee_code> manages project <Project_number>.

| | |
|---|---|
| JnsEd | SmthE |
| P315 | P422 |

Subproject <Sequence_number> of project <Project_number> is led by employee <Employee_code>.

| | | |
|---|---|---|
| 3 | 2 | 1 |
| P315 | P315 | P422 |
| BsBg | WndlA | HkstJa |

Employee <Employee_code> is working on subproject <Sequence_number> of project <Project_number>.

| | | |
|---|---|---|
| DoeJn | RoeRch | RoeRch |
| 3 | 3 | 2 |
| P315 | P315 | P422 |

**Figure 4.26 Result of analyzing FT9**

This completes the analysis of the verbalizations, and the final ERM diagram is in figure 4.26.

### 4.4.5 Examples of fact types with only one segment

Two examples of fact types with only one segment are given below. Both are typical examples of when you might encounter one of those.

**Domain Lists**
In a database, it may be very convenient to have a list of allowed or applicable values for a domain, so it can be implemented by a drop-down list in a user interface screen. This would also be a user-friendly way of preventing typos or reducing tedious typing. For example: a list of the existing departments in an organization (like wards in hospitals), or a list of all the states in the USA, or of all the towns in the Netherlands.

Such value lists could be implemented as constraints on domains (on the metadata level), but also as ordinary tables (on the data lavel). This last option is often to be preferred, especially if the list can change over time. A simple table update is easily done without requiring any changes in the programming of constraints.

Such a table will typically have only one column that contains the allowed values, and is known as a *domain table*. Let's see how to model an ET in ERM that will result in such a domain table.

Here are verbalizations for the currently existing wards in a hospital:

```
There is a ward Ophthalmology.
  "    " "    "  Neonatology and Perinatal Care.
  "    " "    "  Internal Medicine.
```

Clearly, there is only one component in these verbalizations. It should then be analyzed as an ET with an identifying attribute. In the diagram, a new ET with only the <pi> is added. See figure 4.27.

```
There is a ward Ophthalmology.
  "    " "    "  Neonatology and Perinatal Care.
  "    " "    "  Internal Medicine.
                ET WARD
                ID Att Ward_name

There is a ward <Ward_name>.
```

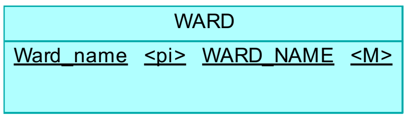


**Figure 4.27 Result of domain list**

(Following the procedure in Appendix A: Option 2 a i → 2 b ii → 3a → 3b → 3e → 3 e ii (the ET is strong) → return to 2 b ii → 2c → 2d → 5 → 5c → 5 c iii and we are done (5d and 5f do not apply)).

It is strongly recommended to use such domain tables and to model them in ERM like this. Such ETs will usually take part in several RTs, most (if not all) with a minimum cardinality of ZERO at the other side.

**All components of the fact expressions are needed to identify a (weak) ET.**
Suppose a take-away restaurant wants to start a web page and needs a database to store its menus in. A menu is identified by a name that should whet the appetite of the potential customers. Each menu consists of several courses and for each course several facts are to be recorded (like the number of calories per person, etc). Here are verbalizations of one fact type:

```
The menu Mexican Chilies contains a course Starters.
 "    "  Mexican Chilies     "    "    "   Main dish.
 "    "  Easter Delight      "    "    "   Main dish.
"    "   Easter Delight      "    "    "   Dessert.
```

Could there be two segments, one for an ET MENU and one for an Att Course? For an ET MENU there is no problem, the domain expert acknowledges that each menu has a unique name. But this ET MENU cannot have an Att Course, because a menu can have more than one course, as the examples clearly show. Therefore COURSE must be an ET of its own (with its own attributes).

But then we have another problem: obviously, the course name (Starters, or Main dish) is not enough to identify a course with: the main dish in the Mexican Chilies menu is not the same as the main dish in the Easter Delight menu. The name of the menu is needed as well to identify the course. So both components of this fact expression are needed to identify an ET COURSE.

So we cannot mark two segments in this fact expression: ET + Att fails (this would result in multiple values for the attribute), and ET + ET fails as well (the course name is not enough to identify a course with). Therefore only one segment can be marked (the dots connect the two parts of the same segment). See the complete analysis and the resulting diagram in figure 4.28 below (suppose that the ET MENU was already determined, perhaps from facts expressions like “The menu Mexican Chilies costs € 25,00 per person.”). Since the identification of ET COURSE contains the <pi> of another ET, a dependent RT must be added between COURSE and this ET. See the result in figure 4.28.

```
The menu Mexican Chilies contains a course Starters.
 "    "  Mexican Chilies     "    "    "   Main dish.
 "    "  Easter Delight      "    "    "   Main dish.
"    "   Easter Delight......"....."...."...Dessert.
         ET COURSE
         ID: ET MENU + Att Course_name
             MATCH

RT COURSE_in_MENU between COURSE(dependent) and MENU

The menu <Menu_name> contains a course <Course_name>.
```

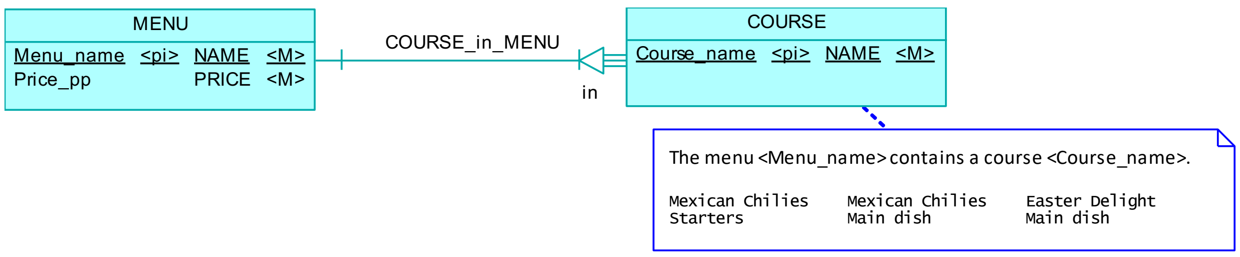


**Figure 4.28** **Result of weak ET**

(Following the procedure in Appendix A: 2 a i → 2 b i → 2 b ii (1) → 3a → 3b → 3e → 3 e iv (the ET is weak) → 3 e iv (4) (a) → 3c (for MENU) → 3d → return to 3 e iv (4) (a) → 4 → 4c → return to 2 b ii (1) → 2c → 2d → 5 → 5b → 5c i, ii, iii, iv, v → 5b (for MENU) and returns to 5 c v → 5 c v (2), (3), (4) and (5) and we are done.)

# 5 Deriving a PDM from a CDM

This chapter shows how to derive a Logical Relational Database Schema (LRS) from a conceptual data model in ERM. This will be illustrated using the Conceptual Data Model (CDM) and Physical Data Model (actually a LRS) from PowerDesigner.

Most software tools with which an ERM data model can be built, like PowerDesigner, can also generate a LRS automatically. However, no automatic transformation from a CDM into a PDM is complete, and sometimes the generated model even contains errors that need to be corrected. Therefore students must be able to evaluate and correct, tweak or extend a generated LRS. A good way to achieve this ability is to be able to derive a PDM from a CDM by hand.

An Entity-Relationship model (ERM) can be transformed into a relational schema in a systematic way, which is described here in six main steps:

- A Process many-to-many relationship types (n-m RTs):
  replace each n-m RT with a weak entity type (ET)
- B Process one-to-one relationship types (1-1 RTs):
  in each 1-1 RT, choose the *dominant* side (which ET will be parent, which will be child)
- C Process entity types (ETs) and attributes (Atts)
- D Process one-to many relationship types (1-n RTs):
  add foreign key columns (FKs) and references
- E Process constraints
- F Add predicates and example populations where convenient

To illustrate the process, a relational database schema for the ERD in figure 5.1 will be derived.

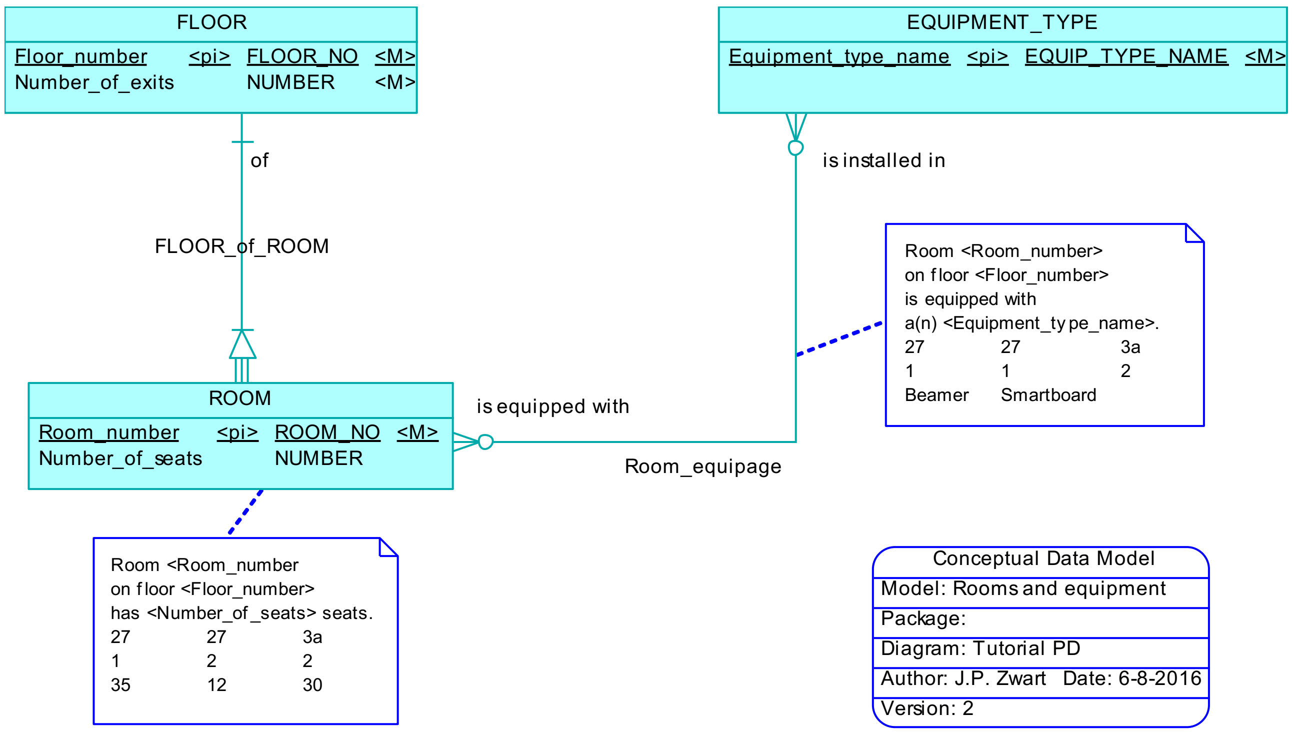


**Figure 5.1 CDM to derive a LRS from**

## 5.1 Step A: Process many-to-many relationship types (n-m RTs)

Many-to-many relationship types (n-m RTs) cannot be translated directly to the Relational Model. Broadly speaking, entity types (ETs) transform into tables, and 1-n RTs or 1-1 RTs into references, but many-to-many 'references' between tables cannot exist. So an alternative for n-m RTs must be supplied, namely a weak ET with two dependent 1-n RTs.

Carry out substeps 1 through 6 for each n-m RT. Starting point is an abstract n-m RT called R between two ETs A and B. See figures 5.2 – 5.4 below.

### 5.1.1 Substep 1: Replace the RT with an ET

Replace RT R with a new ET, and give this ET the same name as the original RT (here: R).

### 5.1.2 Substep 2: Add two new 1-n RTs

Connect the two ETs A and B each with a 1-n RT to the new ET R, with the MANY-side at R.

### 5.1.3 Substep 3: Make the new ET dependent on both sides

The new ET R is weak, and dependent on both old ETs A and B.

### 5.1.4 Substep 4: Determine the minimal cardinalities at the side of R

The maximum cardinalities at the side of R are both MANY (the old RT was many-to-many). Determine the minimum cardinalities using the three cases shown below in figures 5.2, 5.3 and 5.4.

**Case a:** minimal cardinalities of old RT R: ZERO at both sides.

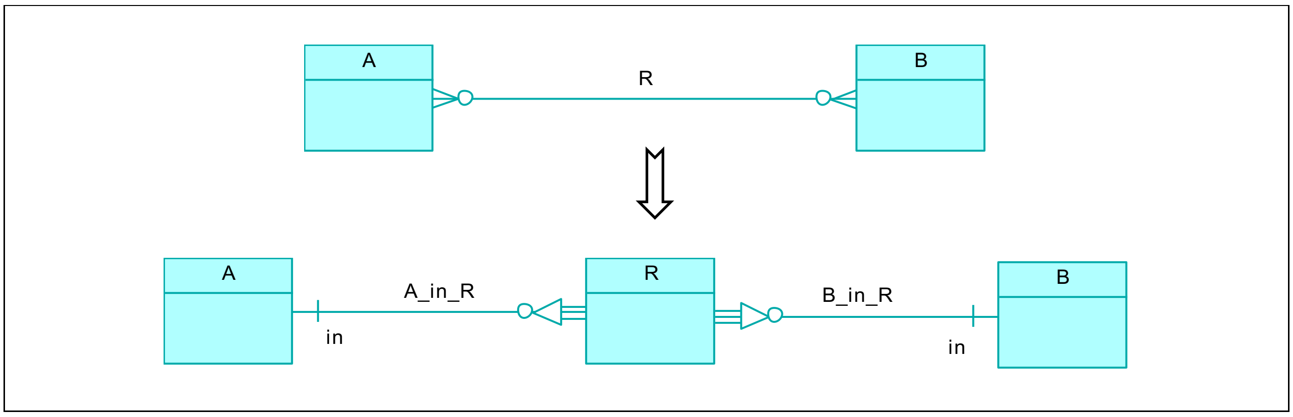


**Figure 5.2** n-m RT with two minimal cardinalities of ZERO

**Case b:** minimal cardinality of old RT R: ZERO at the side of A, ONE at the side of B.
***Take careful note of where the ZERO and ONE end up after replacing the RT with the new ET!***

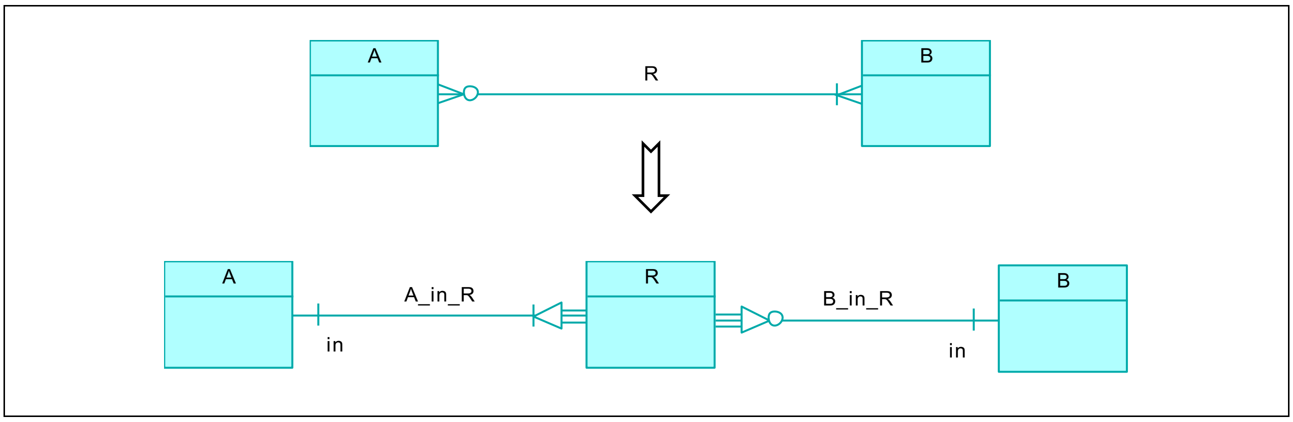


**Figure 5.3** **n-m RT with minimal cardinalities ZERO and ONE**

**Case c:** minimal cardinalities of old RT R: ONE at both sides.

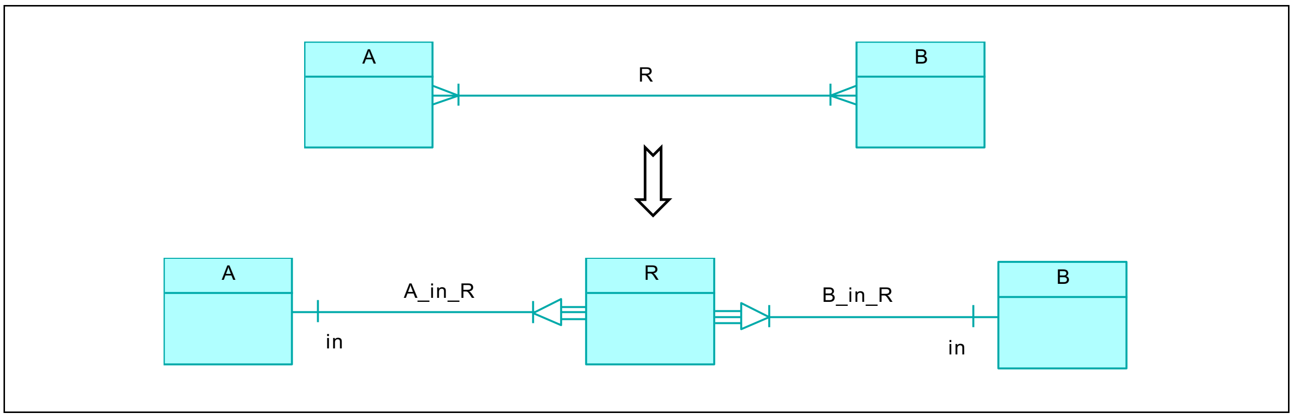


**Figure 5.4** **n-m RT with two minimal cardinalities of ONE**

### 5.1.5 Substep 5: Give the new dependent RTs meaningful names

Dependent RTs are best named like ‘A_in_R’, with R the dependent child ET and A a parent ET (or sometimes as ‘R_of_A’).

### 5.1.6 Substep 6: Add roles to the new dependent RTs

Add at least one role to each new dependent RT. A short text like ‘in’ or ‘of’ is usually sufficient.

Figure 5.5 shows the result of applying step A (substeps 1-6) to the ERD in figure 5.1.

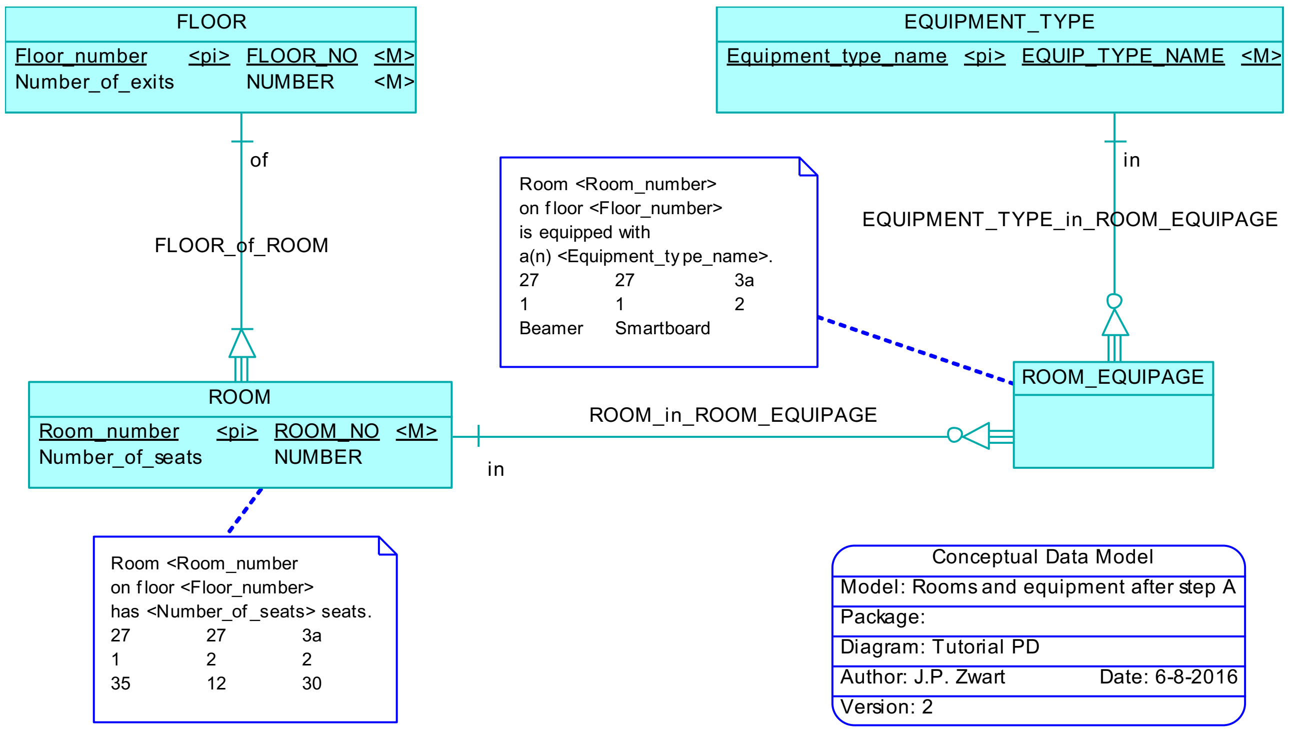


**Figure 5.5 CDM after applying step A**

## 5.2 Step B: Process one-to-one relationship types (1-1 RTs)

### 5.2.1 Dominance in 1-1 RTs

The ERD in figure 5.5 contains no 1-1 RT. Therefore, to introduce the idea of *dominance* in a 1-1 RT, consider the ERD in figure 5.6, which needs no further explanation.

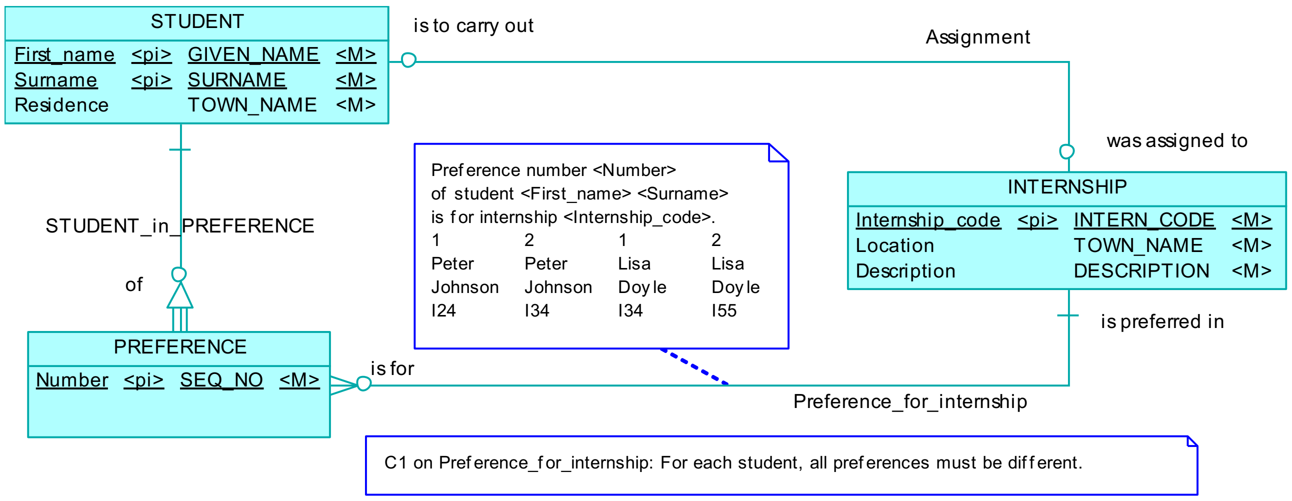

STUDENT
First_name <pi> GIVEN_NAME <M>
Surname <pi> SURNAME <M>
Residence TOWN_NAME <M>
is to carry out
Assignment
was assigned to
INTERNSHIP
Internship_code <pi> INTERN_CODE <M>
Location TOWN_NAME <M>
Description DESCRIPTION <M>
STUDENT_in_PREFERENCE
of
PREFERENCE
Number <pi> SEQ_NO <M>
is for
Preference_for_internship
is preferred in
Preference number <Number>
of student <First_name> <Surname>
is for internship <Internship_code>.
1 2 1 2
Peter Peter Lisa Lisa
Johnson Johnson Doyle Doyle
I24 I34 I34 I55
C1 on Preference_for_internship: For each student, all preferences must be different.

**Figure 5.6 Students and internships**

There is a 1-1 RT Assignment in figure 5.6. Such a RT offers two choices for a relational database schema: would it be preferred to have a FK-column ‘Assigned_internship’ in the table Student, or to have two FK columns ‘Assigned_student_first_name’ and ‘Assigned_student_surname’ in the table Internship? (It would even be possible to have both possibilities at the same time, but this would introduce redundancy, which would have to be protected against possible data pollution by writing routines that prevent this). The choice is arbitrary, so let’s decide to have the first possibility: the table Student should have a column ‘Assigned_internship’ (the result is shown in figure 5.8). To achieve this, the 1-1 RT Assignment has to be made dominant at the side of INTERNSHIP. This means that ET INTERNSHIP is appointed as the *parent* ET in RT Assignment (with STUDENT as the child). The future child table STUDENT will then contain a FK to its parent table INTERNSHIP. See figure 5.7 for the notation in the CDM (the ‘(D)’ at the side of INTERNSHIP), and the result in the PDM in figure 5.8 (see steps C – E for details on how the rest of the PDM is derived; here the focus is only on the effect of making one side of the 1-1 RT dominant).

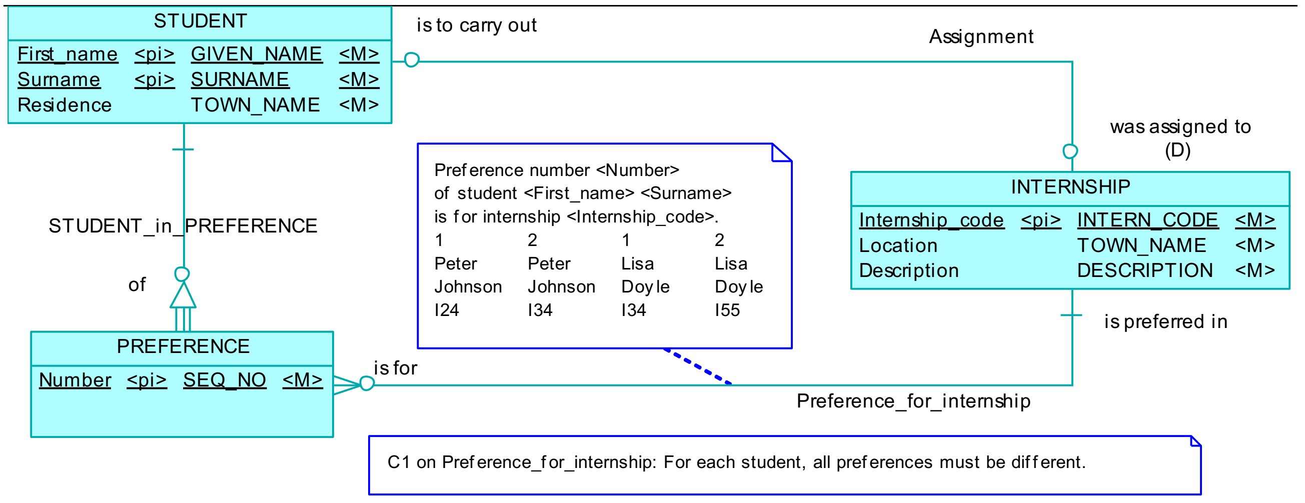

STUDENT
First_name <pi> GIVEN_NAME <M>
Surname <pi> SURNAME <M>
Residence TOWN_NAME <M>
is to carry out
Assignment
was assigned to (D)
INTERNSHIP
Internship_code <pi> INTERN_CODE <M>
Location TOWN_NAME <M>
Description DESCRIPTION <M>
is preferred in
STUDENT_in_PREFERENCE
of
PREFERENCE
Number <pi> SEQ_NO <M>
is for
Preference_for_internship
Preference number <Number> of student <First_name> <Surname> is for internship <Internship_code>.
1 2 1 2
Peter Peter Lisa Lisa
Johnson Johnson Doyle Doyle
I24 I34 I34 I55
C1 on Preference_for_internship: For each student, all preferences must be different.

**Figure 5.7** RT Assignment dominant at the INTERNSHIP side

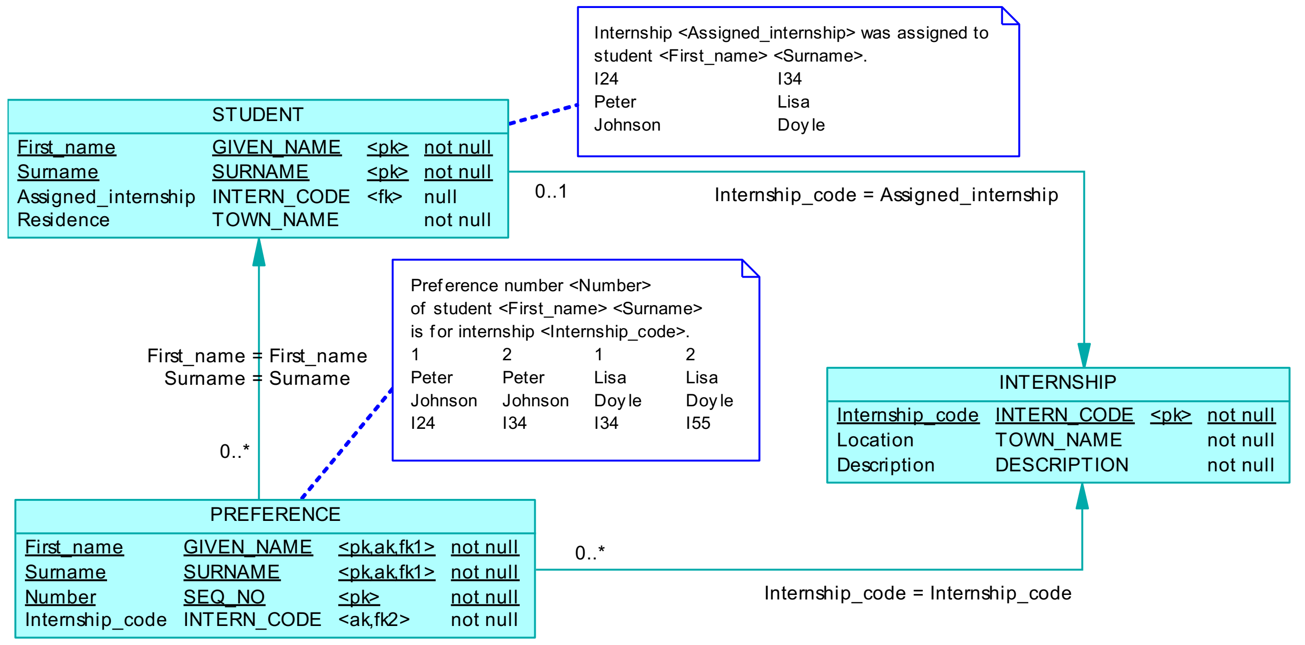

Internship <Assigned_internship> was assigned to student <First_name> <Surname>.
I24 I34
Peter Lisa
Johnson Doyle
STUDENT
First_name GIVEN_NAME <pk> not null
Surname SURNAME <pk> not null
Assigned_internship INTERN_CODE <fk> null
Residence TOWN_NAME not null
0..1
Internship_code = Assigned_internship
First_name = First_name
Surname = Surname
0..*
Preference number <Number> of student <First_name> <Surname> is for internship <Internship_code>.
1 2 1 2
Peter Peter Lisa Lisa
Johnson Johnson Doyle Doyle
I24 I34 I34 I55
INTERNSHIP
Internship_code INTERN_CODE <pk> not null
Location TOWN_NAME not null
Description DESCRIPTION not null
PREFERENCE
First_name GIVEN_NAME <pk,ak,fk1> not null
Surname SURNAME <pk,ak,fk1> not null
Number SEQ_NO <pk> not null
Internship_code INTERN_CODE <ak,fk2> not null
0..*
Internship_code = Internship_code

**Figure 5.8** PDM derived from CDM in figure 5.7

For completeness, the result of making the other choice for the dominant side is shown in figures 5.9 and 5.10.

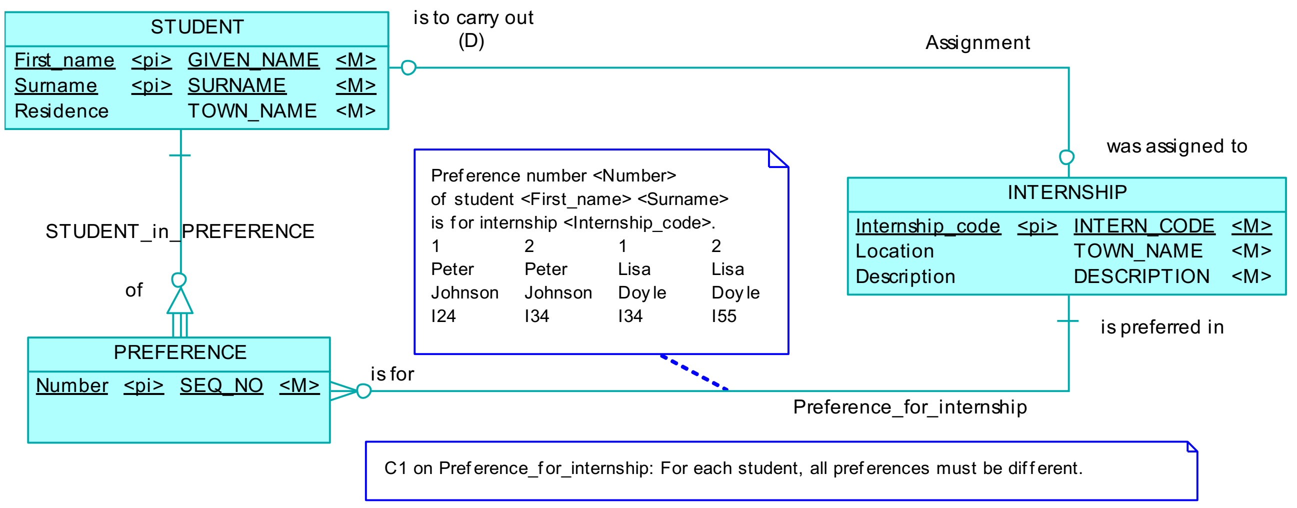

STUDENT
First_name <pi> GIVEN_NAME <M>
Surname <pi> SURNAME <M>
Residence TOWN_NAME <M>
is to carry out (D)
Assignment
was assigned to
INTERNSHIP
Internship_code <pi> INTERN_CODE <M>
Location TOWN_NAME <M>
Description DESCRIPTION <M>
STUDENT_in_PREFERENCE
of
PREFERENCE
Number <pi> SEQ_NO <M>
is for
is preferred in
Preference_for_internship
Preference number <Number> of student <First_name> <Surname> is for internship <Internship_code>.
1 2 1 2
Peter Peter Lisa Lisa
Johnson Johnson Doyle Doyle
I24 I34 I34 I55
C1 on Preference_for_internship: For each student, all preferences must be different.

**Figure 5.9** **R_Assignment dominant at the STUDENT side**

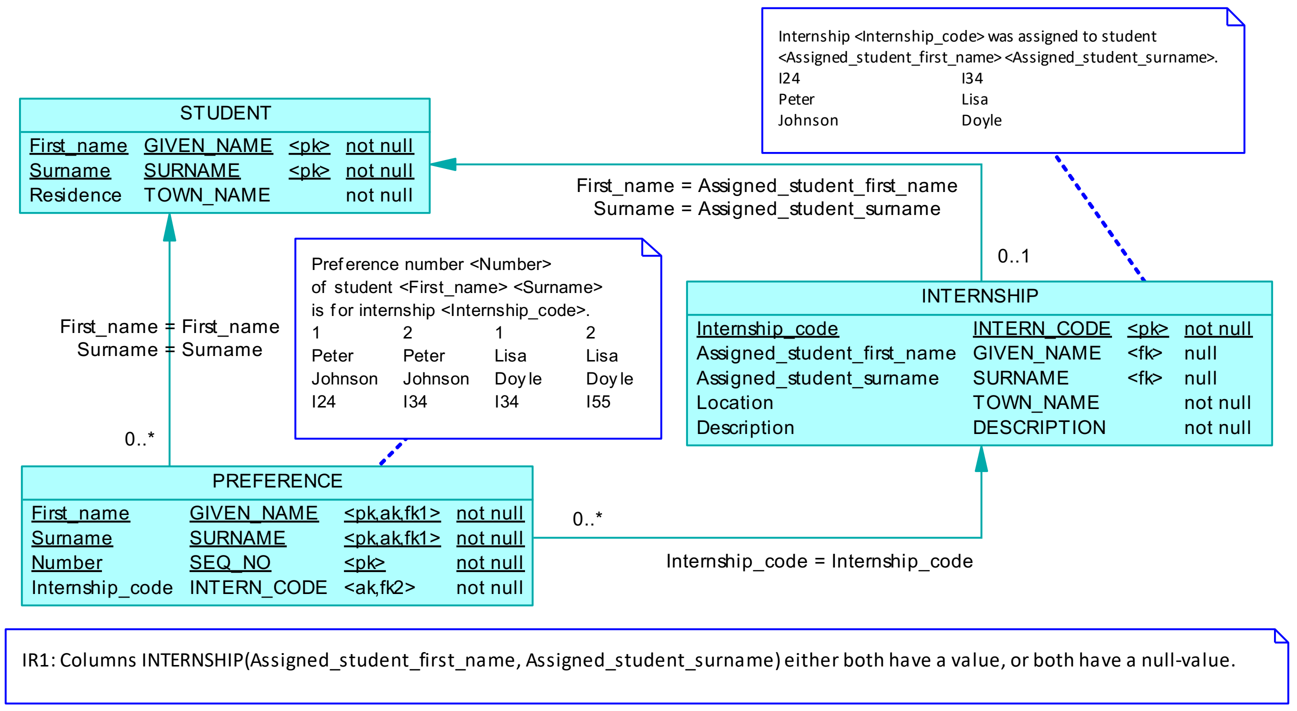

Internship <Internship_code> was assigned to student <Assigned_student_first_name> <Assigned_student_surname>.
I24 I34
Peter Lisa
Johnson Doyle
STUDENT
First_name GIVEN_NAME <pk> not null
Surname SURNAME <pk> not null
Residence TOWN_NAME not null
First_name = Assigned_student_first_name
Surname = Assigned_student_surname
0..1
INTERNSHIP
Internship_code INTERN_CODE <pk> not null
Assigned_student_first_name GIVEN_NAME <fk> null
Assigned_student_surname SURNAME <fk> null
Location TOWN_NAME not null
Description DESCRIPTION not null
First_name = First_name
Surname = Surname
0..*
Preference number <Number> of student <First_name> <Surname> is for internship <Internship_code>.
1 2 1 2
Peter Peter Lisa Lisa
Johnson Johnson Doyle Doyle
I24 I34 I34 I55
PREFERENCE
First_name GIVEN_NAME <pk,ak,fk1> not null
Surname SURNAME <pk,ak,fk1> not null
Number SEQ_NO <pk> not null
Internship_code INTERN_CODE <ak,fk2> not null
0..*
Internship_code = Internship_code
IR1: Columns INTERNSHIP(Assigned_student_first_name, Assigned_student_surname) either both have a value, or both have a null-value.

**Figure 5.10** **PDM derived from CDM in figure 5.8**

So for each 1-1 RT, substep 7 below is to be carried out.

### 5.2.2 Substep 7: Choose the dominant side in 1-1 RTs

In each 1-to-1 relationship type, mark one of the two entity types as the dominant (parent) ET. (This is not strictly necessary, but it is the best practice in most cases; otherwise redundancy arises that has to be controlled by procedures.) The choice is arbitrary; perhaps the domain expert will have a preference for one or the other (ask!). In the resulting relational database schema, the child table that will be generated from the <u>other</u> entity type will contain a foreign key to the parent table.
If the minimum cardinality is ZERO at one side and ONE at the other, then it is almost always best to make the ET at the ONE-side dominant. In other cases the choice is completely arbitrary.

## 5.3 Step C: Process entity types (ETs) and attributes (Atts)

### 5.3.1 Substep 8: Entity types become tables, attributes become columns

Each entity type is translated into a table. Add to each table all the attributes of the entity type as columns (a column is also called an attribute in the Relational Model).

Replace the <pi> indicator (for 'primary identifier') with a <pk> indicator (for 'primary key').

Copy the domain of every attribute to the corresponding column.

Replace the '<M>'-indicator of the original attribute with 'not null'for the corresponding column, otherwise enter 'null'.

The result of this step is not given separately, see figure 5.11 for the final result of applying this step and step D to the ERD from figure 5.5.

## 5.4 Step D: Process one-to many relationship types (1-n RTs)

### 5.4.1 Substep 9: 1-n RTs become references

Replace each one-to-many relationship type by a child-to-parent reference, i.e.: a reference pointing from the child table (the table on the 'many'-side) to the parent table (table on the 'one'-side). Do not name this reference yet (in substep 12 extra elements will be added to the reference).

### 5.4.2 Substep 10: Add foreign key columns

For each 1-n RT from substep 9: add one or more new foreign key columns to the child table, one for each identifying attribute of the parent table. It is easiest to start with tables that come from ETs that do not have weak parent ETs (otherwise: be sure to include all the attributes from the dependent relationships of these parent ETs!), together with their domains.

Mark these foreign key columns with '<fk>'. Include a sequence number with the foreign key (like <fk1>, <fk2>, …) if there is more than one foreign key in the same table. Note: what is meant here is different FKs, not different columns in the same FK. If there are two different FKs, one consisting of a single column, and another consisting of two columns, then the column of the first FK will get <fk1>, and both columns of the second FK get <fk2>.

Mark these foreign key columns also with 'pk' (so the combined indication '<pk, fk>' is obtained) if the relationship type is dependent (on the child side).

Mark these foreign key columns as 'not null' if the minimum cardinality at the parent-side is ONE, and mark them as 'null' if that cardinality is ZERO.

### 5.4.3 Substep 11: Process inheritance links of subtypes

This substep is discussed in chapter 6, section 6.4.4, after the introduction of subtypes. For completeness, it is stated her as well, but without explanation:

- For each subtype, decide whether or not a separate table for it is desired, and set the Inheritance Link Generation and Children properties correspondingly.
- For each declarative subtype, for which no separate table is desired:
  add specifying attributes to prevent information loss.
- Translate all subtype-related constraints from the CDM to integrity rules in the PDM. This must usually be done by hand.

### 5.4.4 Substep 12: Add join expressions to all references

In the notation of a PowerDesigner PDM, it is not always clear which columns in a child table refer to which columns in a parent table. Although many FK references are indeed obvious, there are situations in which the explicit join information cannot be done without.

Therefore: include the 'join expressions' for each reference (in PowerDesigner this is easily done automatically: see the Tutorial Power Designer in the course materials). These join expressions are given in the form: Column x = Column y (with the child table column (Column x) first and the parent table column (Column y) last).

### 5.4.5 Substep 13: Add cardinalities to all references

PowerDesigner uses a peculiar notation for references that differs from the standard one in the Relational Model. In particular, PowerDesigner uses only a single arrow, both for ordinary FK-references and for FK + mandatory child references. But whereas there is only one reference in an ordinary FK-reference, namely from the child to the parent, there are actually <u>two</u> references in a mandatory-child case: the usual FK-reference and a second reference that points in the opposite direction (see the example CDM in section 3.2.3 of the Tutorial PowerDesigner, in which table ROOM is a mandatory child of table FLOOR: the ordinary reference means: every room must be on an existing floor (FK Floor_number in table ROOM), and the mandatory-child reference means: every floor must have at least one room (no Floor_number from table FLOOR can be missing in table ROOM)). In PowerDesigner however both ordinary and mandatory-child references are shown with only one arrow, so there is no direct way to tell them apart. Only by showing the cardinalities at the foot of the arrow can it be made clear whether there is only an ordinary FK reference (0..*) or also a mandatory-child reference (1..*).

Therefore: include both cardinalities at the foot (child side) of the arrow that indicates a reference (in PowerDesigner this is easily done automatically: see the Tutorial Power Designer in the course materials).

Figure 5.11 shows the result of applying steps C – E (substeps 8-13) to the CDM in figure 5.5.

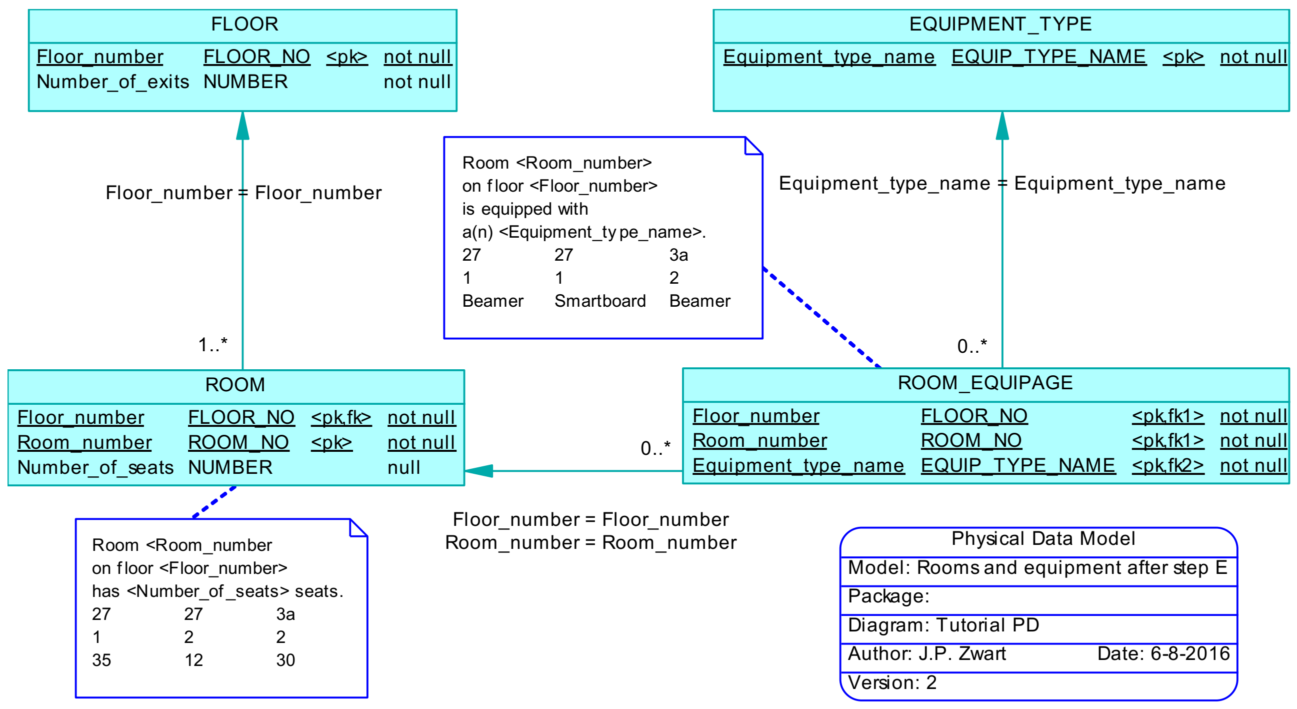

FLOOR
Floor_number FLOOR_NO <pk> not null
Number_of_exits NUMBER not null
EQUIPMENT_TYPE
Equipment_type_name EQUIP_TYPE_NAME <pk> not null
Floor_number = Floor_number
Room <Room_number> on floor <Floor_number> is equipped with a(n) <Equipment_type_name>.
27 27 3a
1 1 2
Beamer Smartboard Beamer
Equipment_type_name = Equipment_type_name
1..*
0..*
ROOM
Floor_number FLOOR_NO <pk,fk> not null
Room_number ROOM_NO <pk> not null
Number_of_seats NUMBER null
0..*
ROOM_EQUIPAGE
Floor_number FLOOR_NO <pk,fk1> not null
Room_number ROOM_NO <pk,fk1> not null
Equipment_type_name EQUIP_TYPE_NAME <pk,fk2> not null
Floor_number = Floor_number
Room_number = Room_number
Room <Room_number on floor <Floor_number> has <Number_of_seats> seats.
27 27 3a
1 2 2
35 12 30
Physical Data Model
Model: Rooms and equipment after step E
Package:
Diagram: Tutorial PD
Author: J.P. Zwart Date: 6-8-2016
Version: 2

**Figure 5.11 PDM derived from the CDM in figure 5.5**

## 5.5 Step E: Process constraints

### 5.5.1 Substep 14: translate constraints into integrity rules

Most contraints in the CDM will translate easily into integrity rules in the PDM. For example: <pi> and <ai> will translate into <pk> and <ak>, <M> will translate into 'not null', and most cardinalities will translate into similar integrity rules (minimum cardinality ONE → not null or mandatory child reference, etc.). However: most relational database management systems cannot (yet?) handle mandatory child references automatically. Ordinary FK-references are usually no problem, but the references in the opposite direction have to be implemented with procedures.

A few examples of other constraints are shown in the figures above.

- In figures 5.6 and 5.7 there is a constraint C1 that specifies that a student can have several preferences for internships, but there cannot be two preferences for the same internship (see also the example population in figures 5.6. and 5.7). Constraint C1 translates into the <ak> of table PREFERENCE in figure 5.8. This is a nice example of an alternative identifier for an ET that cannot be shown graphically: the dependency mechanism of RTs does not cater for alternative ways of identification. In the relational table, the FK columns allow an 'ordinary' <ak> tp be specified inst=stead of C1.
- In figure 5.10, there are two FK-columns in table INTERNSHIP, containing the name of the student to whom the internship was assigned. Both columns are optional, but it cannot be that only one of the two has a value and the other doesn't have a value. Therefore, integrity rule IR1 was added to make sure either both names are present in a tuple, or no name at all.

It is not possible to give a general way of translating all constraints into integrity rules, and the translation can be quite complex sometimes. This is one of the areas where data modeling is still more an art than a science, and experience is a good teacher in practice.

## 5.6 Step F: Add predicates and example populations where convenient

As usual, only unclear columns have to be clarified. Note that in a PDM, all fact types are modeled as <pk> or <pk>+column, there are no RTs left (only references, but these do not model fact types).

### 5.6.1 Substep 15: Add predicates and example populations where convenient

See figure 5.11 for the result. Note that the same predicates and example populations are present in the PDM in figure 5.11 as are in the original CDM in figure 5.1, but now only attached to tables.

# 6 Subtypes

## 6.1 Introducing subtypes and supertypes

### 6.1.1 Subtype and supertype

A common kind of business rules often encountered in practice concerns business rules like:

| | |
|---|---|
| BR1 | PSA-levels can only be recorded for male patients. |
| BR2 | Only executive staff members are entitled to an indoor parking space. |
| BR3 | A discount percentage only applies to orders with a total amount greater than € 100. |

Such business rules are of the general form **We record […] only for […]**.

Most data modeling techniques (FCO-IM, ERM, UML, …) include a way to model such business rules, namely in the form of *subtypes.* In each case above, there is an entity type (PATIENT, STAFF MEMBER, ORDER), and there are facts (PSA level, indoor parking space, discount) that only apply to a *special subset* of this entity type (MALE_PATIENT, EXECUTIVE_STAFF_MEMBER, BIG_ORDER). Such a special subset of an entity type is called a *subtype*, and the entity type it belongs to is called its *supertype*.

There is an essential difference between an optional attribute in an ET, and an attribute in a subtype. Suppose the surname of patients is recorded if possible, but sometimes the surname is not known (traffic accident victim brought in unconsciously, confused demented old lady, …). So Att Surname in ET PATIENT is optional (not <M>) and it cannot be known in advance for which PATIENT-entities a value in Surname is available: there is no special class of patients with a surname. But the PSA level is only recorded for male patients. So it is indeed known in advance there is a special class of patients for which Att PSA_level applies, and this can be modeled using a subtype MALE_PATIENT to contain this attribute. The difference therefore is in random versus systematic absence or presence of data. The examples above clearly concern such systematic situations.

This chapter discusses the modeling of subtypes in ERM, and the consequences of deriving a relational schema (PDM) from a CDM with subtypes. The following small example is used throughout. Suppose a database must store the following data about cars (see figure 6.1):

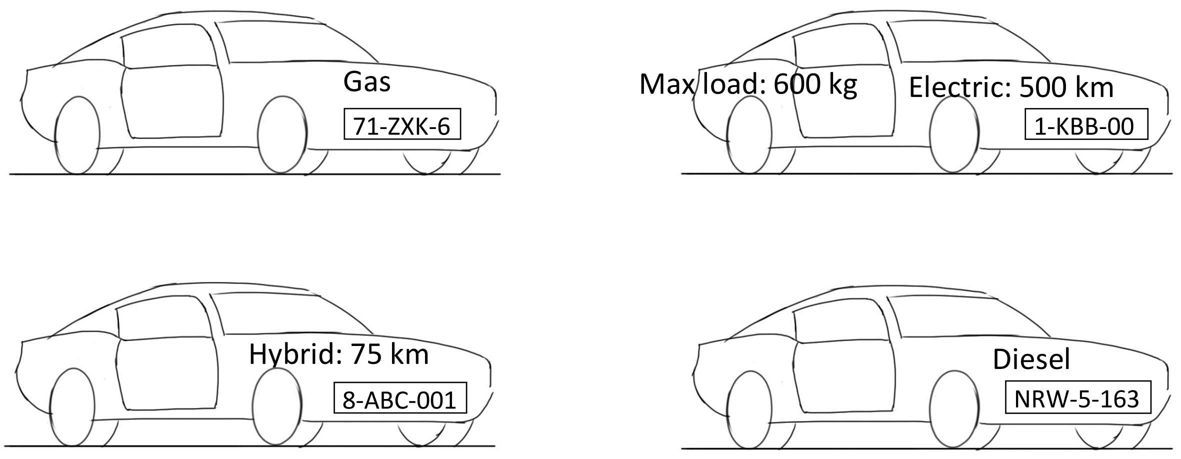


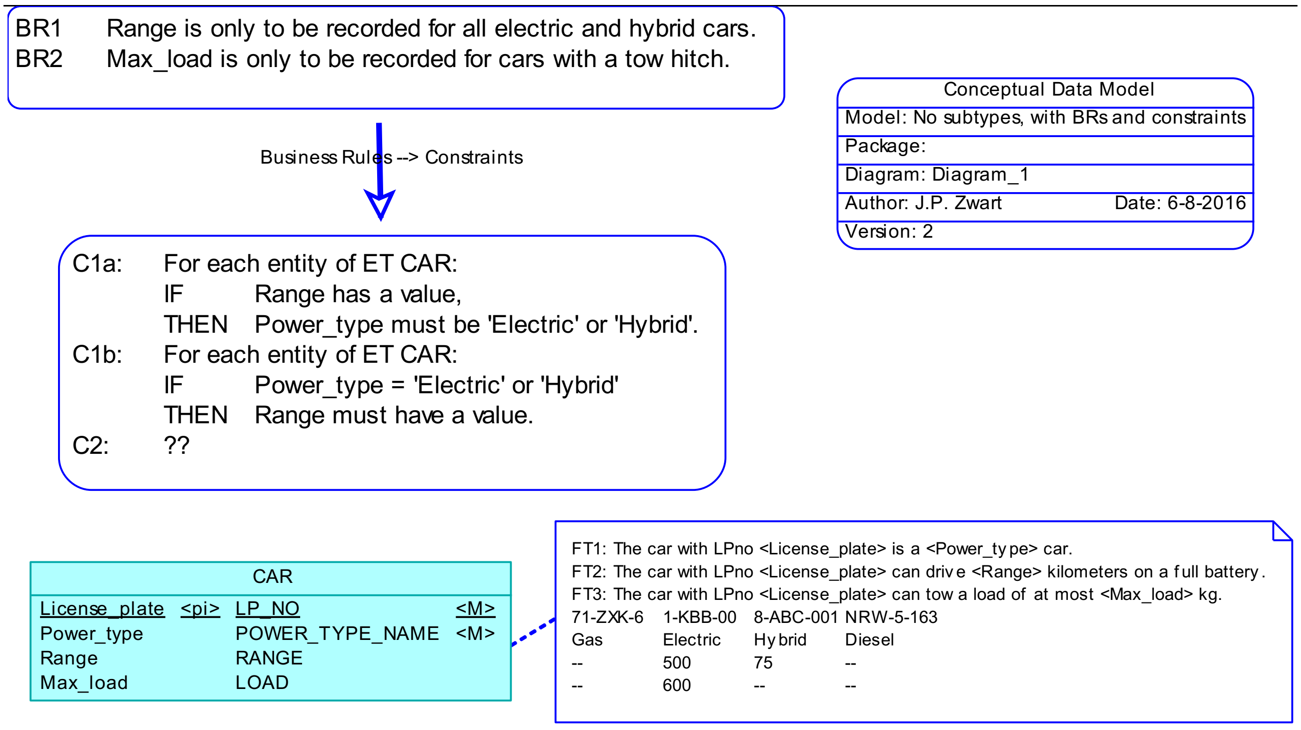
**Figure 6.1 Example car data**

The verbalizations were easily made with the domain expert and need no further explanation (only one fact per fact type given here):

```
      FT1   The car with LPno 1-KBB-00 is an electric car.
      FT2   The car with LPno 1-KBB-00 can drive 500 km on a full battery.
      FT3   The car with LPno 1-KBB-00 can tow a load of at most 600 kg.
```

The domain expert made it clear that the distance a car can drive on a full battery (the range, FT2) is only to be recorded for all electric and hybrid cars (because they use electricity as power source to drive), and that the maximum towing load (FT3) is only to be recorded for cars that have a tow hitch (pulling hook fitted at the back of the car), but that this figure is not always available.

The analysis of the three fact types without using subtypes is straightforward, and not shown here. The resulting ERD is shown in figure 6.2.



BR1 Range is only to be recorded for all electric and hybrid cars.
BR2 Max_load is only to be recorded for cars with a tow hitch.

```
C1a:  For each entity of ET CAR:
      IF     Range has a value,
      THEN   Power_type must be 'Electric' or 'Hybrid'.
C1b:  For each entity of ET CAR:
      IF     Power_type = 'Electric' or 'Hybrid'
      THEN   Range must have a value.
C2:   ??
```

| Conceptual Data Model | |
|---|---|
| Model: No subtypes, with BRs and constraints | |
| Package: | |
| Diagram: Diagram_1 | |
| Author: J.P. Zwart | Date: 6-8-2016 |
| Version: 2 | |

| CAR | | | |
|---|---|---|---|
| License_plate | <pi> | LP_NO | <M> |
| Power_type | | POWER_TYPE_NAME | <M> |
| Range | | RANGE | |
| Max_load | | LOAD | |

FT1: The car with LPno <License_plate> is a <Power_type> car.
FT2: The car with LPno <License_plate> can drive <Range> kilometers on a full battery.
FT3: The car with LPno <License_plate> can tow a load of at most <Max_load> kg.

| 71-ZXK-6 | 1-KBB-00 | 8-ABC-001 | NRW-5-163 |
|---|---|---|---|
| Gas | Electric | Hybrid | Diesel |
| -- | 500 | 75 | -- |
| -- | 600 | -- | -- |

**Figure 6.2 ERD for car data: no subtypes**

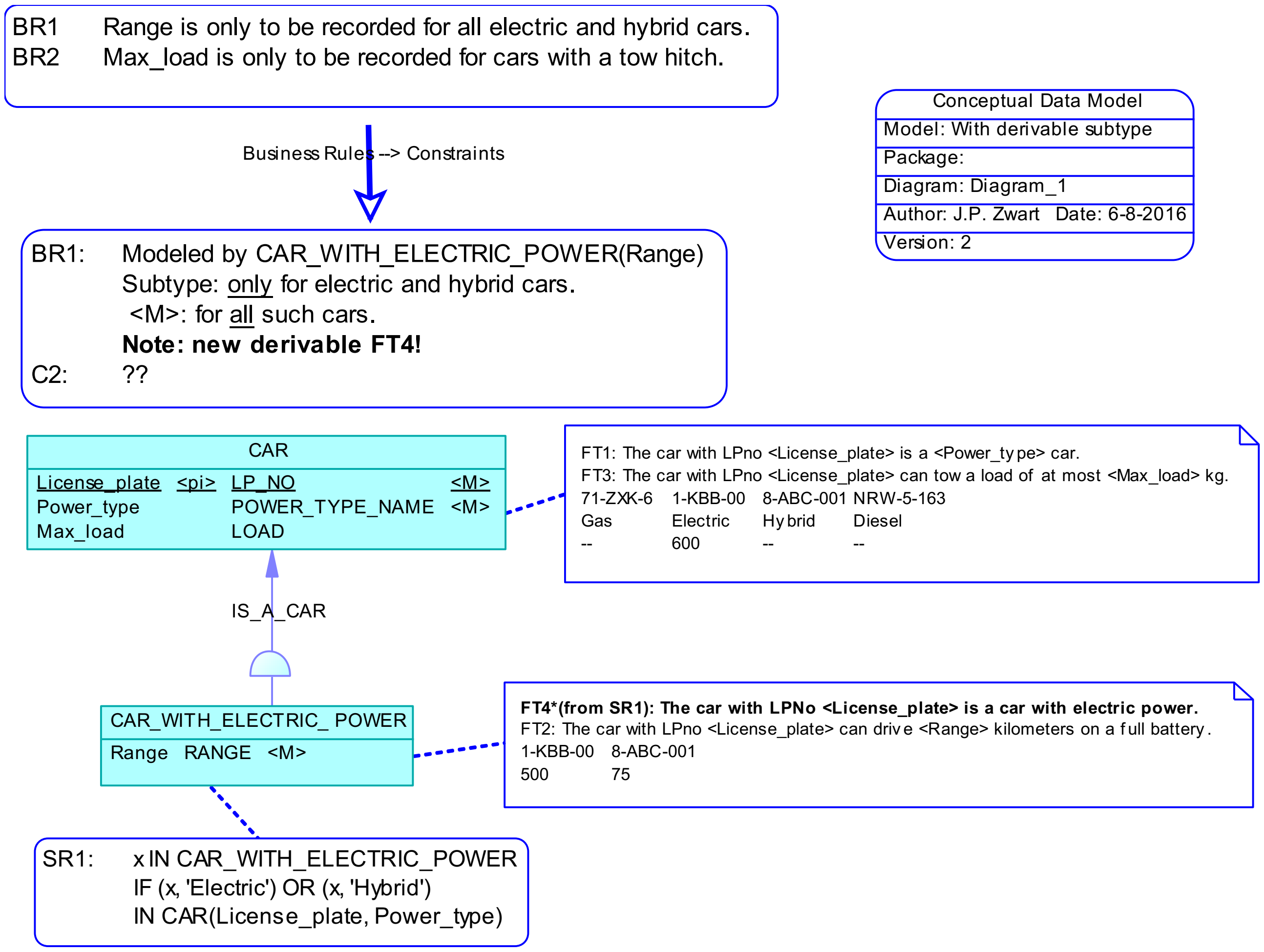
Business Rules BR1 and BR2, which capture the remarks of the domain expert, are given at the top of figure 6.2. They must be translated into constraints in the conceptual data model. C1a captures that Range is only to be recorded for electric and hybtid cars, and C1b captures that this must de done for all such cars. The notation of the constraints is informal, only meant to make the meaning clear (there are several more or less complete constraint notation languages, but these are outside the scope of this reader). The predicates and example populations were added as well.

It is not clear at this point how BR2 should be dealt with. This is postponed to section 6.4. But in section 6.1.2, let's consider BR1 and C1a and C1b first. There is a more visual way to model these constraints that makes it a lot easier to see that the range is only to be recorded for electrically powered cars.

### 6.1.2 Inheritance link

In figure 6.3, a *subtype* CAR_WITH_ELECTRIC_POWER is introduced. In ERM, a subtype is modeled as an entity type (ET) without an own primary identifier<pi>. There is no need for a separate identifier, because a subtype always has the *same* identifier as its parent supertype.

Instead, as you can see in figure 6.3, the new subtype is connected to its *supertype* CAR by an *inheritance link*: the arrow named IS_A_CAR pointing from the subtype to the supertype with the half-circle symbol in the middle. In this reader, inheritance links are named starting with 'IS_A_' and followed by the supertype name, but this is an arbitrary convention, not a requirement.


Business Rules --> Constraints
Conceptual Data Model
Model: With derivable subtype
Package:
Diagram: Diagram_1
Author: J.P. Zwart Date: 6-8-2016
Version: 2
CAR
License_plate <pi> LP_NO <M>
Power_type POWER_TYPE_NAME <M>
Max_load LOAD
IS_A_CAR
CAR_WITH_ELECTRIC_ POWER
Range RANGE <M>

BR1 Range is only to be recorded for all electric and hybrid cars.
BR2 Max_load is only to be recorded for cars with a tow hitch.

BR1: Modeled by CAR_WITH_ELECTRIC_POWER(Range)
Subtype: only for electric and hybrid cars.
<M>: for all such cars.
**Note: new derivable FT4!**
C2: ??

FT1: The car with LPno <License_plate> is a <Power_type> car.
FT3: The car with LPno <License_plate> can tow a load of at most <Max_load> kg.

| | | | |
|---|---|---|---|
| 71-ZXK-6 | 1-KBB-00 | 8-ABC-001 | NRW-5-163 |
| Gas | Electric | Hybrid | Diesel |
| -- | 600 | -- | -- |

**FT4*(from SR1): The car with LPNo <License_plate> is a car with electric power.**
FT2: The car with LPno <License_plate> can drive <Range> kilometers on a full battery.

| | |
|---|---|
| 1-KBB-00 | 8-ABC-001 |
| 500 | 75 |

SR1: x IN CAR_WITH_ELECTRIC_POWER
IF (x, 'Electric') OR (x, 'Hybrid')
IN CAR(License_plate, Power_type)

**Figure 6.3 ERD for car data: derivable subtype**

The link between a subtype and its supertype is called an *inheritance* link, because all the attributes of the supertype – including its complete <pi> – also apply for the subtype: the subtype 'inherits' all the attributes (including the <pi>) from the supertype. In contrast, the attribute(s) listed in the subtype (here: 'Range') only apply to the subtype itself, not to the supertype.

Inheritance links can also be seen as special 1-1 dependent RTs (with additional inheritance properties) in which the cardinalities at the supertype side are both ONE (because the subtype is dependent on the supertype for its <pi>) and the cardinalities at the subtype side are ZERO and ONE (why must this always be so?). Since there are no 'free' cardinalities, the cardinalities are not shown explicitly on the inheritance link.

### 6.1.3 Subtype defining fact type

If a subtype is added to an ERD, then it must be clear which entities of the supertype also belong to the subtype: which cars are also cars-with-electric-power? But for each such car, this is an ordinary *fact about the car* that can be verbalized, for example using expressions like:

```
          The car with LPno 1-KBB-00 is a car with electric power.
           "   "    "   "    8-ABC-001 " "  "    "       "     "  .
     FT4: The car with LPno <License_plate> is a car with electric power.
```

Therefore, **modeling a subtype *implies adding such a fact type* to the model**.
Such a fact type is called a subtype defining fact type because it states which entities belong to the subtype (see FT4 in figure 6.3).

Subtype defining fact types have a strange property: they cannot be seen explicitly in a classic ERD! This is because they add no new attribute to the model, but only use the identifier of the supertype (which is not shown in the subtype anyway). Still, they are really there, but their invisibility in classic ERDs is a source of errors in deriving a PDM from it. Therefore, in this reader the subtype defining fact types are always shown as predicates in the notes connected to the subtype, together with the other predicates and example populations. FO-ERM really helps here!

It would also be possible to associate the subtype defining fact type with the inheritance link (below the half circle symbol), but for convenience it is placed together with the other fact types that apply to the subtype only.

The above point is important enough to make it a general rule, shown in figure 6.4.

Subtype rule 1:
Modeling a subtype implies adding a subtype defining fact type.
A subtype defining fact type states which entities of the supertype belong to the subtype.
Every subtype must have a subtype defining fact type.

**Figure 6.4 Subtype rule 1**

## 6.2 Derivable subtypes

Is it known which entities from ET CAR also belong in the subtype CAR_WITH_ELECTRIC_POWER? Yes, namely those entities that have the value ‘Electric’ or ‘Hybrid’ in their attribute Power_type (C1a and C1b in figure 6.2). This means that it is possible to give a *Subtype Derivation Rule* that specifies which entities from CAR belong to CAR_WITH_ELECTRIC_POWER. This rule is given at the bottom of figure 6.3: SDR1, in which ‘x’ stands for an arbitrary license plate number.

This also means that the subtype defining fact type FT4 is a derivable fact type (it follows from SDR1), which is indicated by the asterisk (*) following the name FT4.

A *derivable subtype* is a subtype with a derivation rule and a derivable subtype defining fact type. By far most subtypes are derivable subtypes.

Here is the tradeoff between the model in figure 6.2 and the model in figure 6.3:

- Both ERDs model BR1 (for BR2 see section 6.3).
- The ERD in figure 6.2 uses C1a and C1b with an optional Att Range,
  the ERD in figure 6.3 uses a subtype and a derivation rule with a mandatory Att Range.

Which model is to be preferred? In practice, there is a clear preference for the model with the subtype:

- BR1 is much easier to read there (graphic subtype instead of abstract text) than in the model with C1a and C1b.
- There is not much difference in having to specify C1a and C1b on the one hand, and SDR1 on the other hand.
- The use of subtypes allows for a lot more flexibility in deriving a relational schema. This will be explained in detail in section 6.4.

This section closes with another rule for modeling subtypes. The second bullet must be strictly obeyed: otherwise logical errors will result (like tautologies).

Subtype rule 2:
- If possible, specify a Subtype Derivation Rule.
- A Subtype Derivation Rule can only refer to fact types from the supertype,
  not to fact types from the subtype itself.
- If this can be done, the subtype is a derivable subtype (normal situation).

**Figure 6.5 Subtype rule 2**

## 6.3 Declarative subtypes

In section 6.2, BR1 was modeled using a derivable subtype. Is it possible to treat BR2 in the same way? Let's try (see the ERD in figure 6.6).

Since the maximum towing load is recorded only for cars with a tow hitch, creating a subtype CAR_WITH_TOW_HITCH seems natural. It is a second subtype of CAR, and therefore also connected to CAR by the inheritance link. Inheritance links allow multiple subtypes of the same supertype, all connected to the half-circle symbol (It is also possible to specify two completely separate inheritance links, but this is not recommended, reasons for this are given in section 6.5). This attribute is not <M> however, because even for cars with a tow hitch this value is not always known, as the domain expert has stated. But the value is only recorded for cars with a tow hitch (even though not for all), so using a subtype is still justified.

However, a crucial difference with the other subtype is, that *no Subtype Derivation Rule (SDR)* can be given. There is no other information that tells us for which cars Max_load is relevant. There is no fact type in the model that tells us which cars have tow hitches. Or is there? According to subtype rule 1 in figure 6.4, modeling a subtype implies adding a subtype defining fact type. This is FT5, shown in figure 6.6 in the note associated with the new subtype. FT5 however, is *not derivable* because there is no SDR. FT5 cannot be used for a SDR, because it applies to the subtype (see figure 6.5, second bullet). Therefore, FT5 is not marked with an asterisk, like FT4.

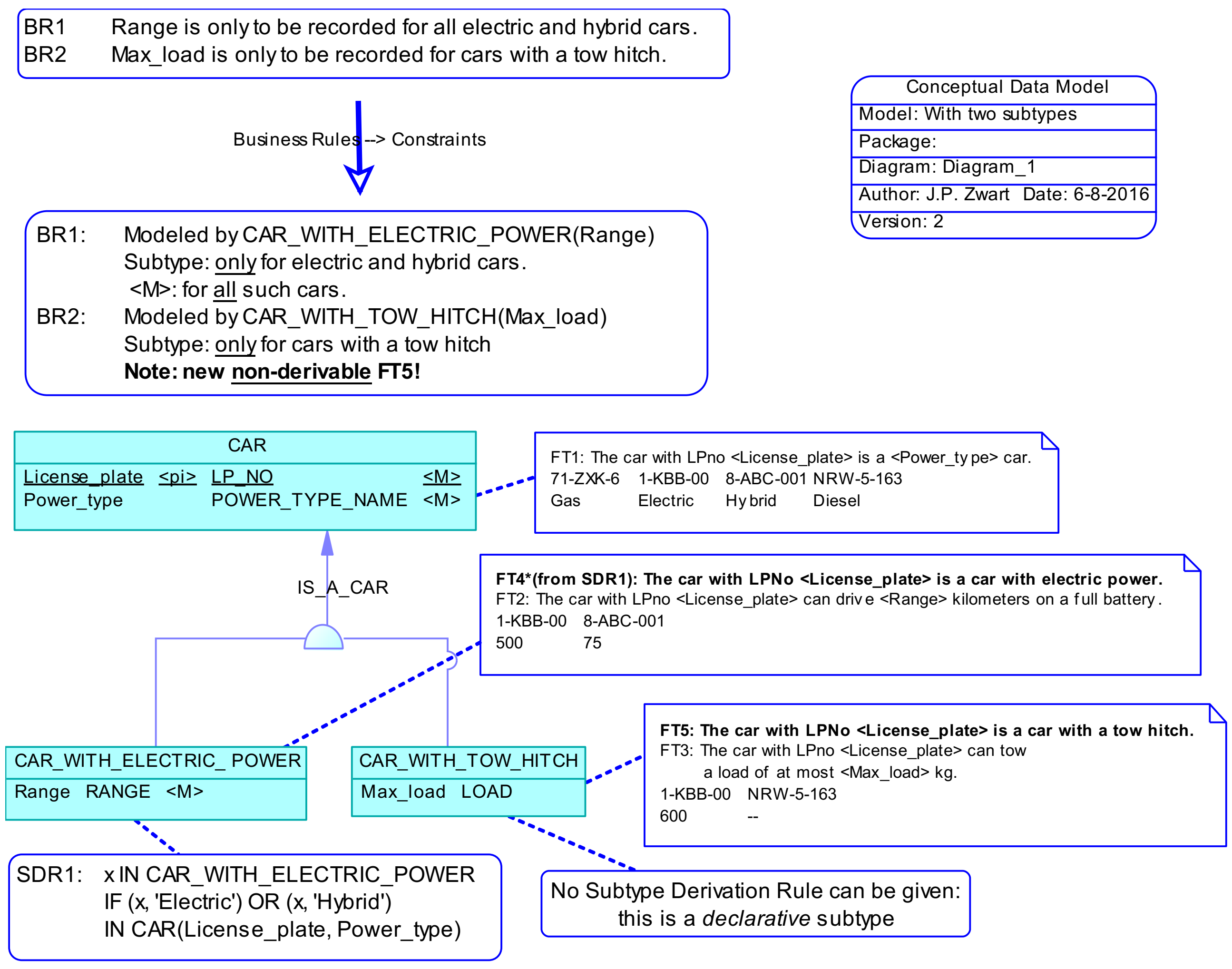

BR1 Range is only to be recorded for all electric and hybrid cars.
BR2 Max_load is only to be recorded for cars with a tow hitch.
Business Rules --> Constraints
BR1: Modeled by CAR_WITH_ELECTRIC_POWER(Range)
Subtype: only for electric and hybrid cars.
<M>: for all such cars.
BR2: Modeled by CAR_WITH_TOW_HITCH(Max_load)
Subtype: only for cars with a tow hitch
Note: new non-derivable FT5!
Conceptual Data Model
Model: With two subtypes
Package:
Diagram: Diagram_1
Author: J.P. Zwart Date: 6-8-2016
Version: 2
CAR
License_plate <pi> LP_NO <M>
Power_type POWER_TYPE_NAME <M>
FT1: The car with LPno <License_plate> is a <Power_type> car.
71-ZXK-6 1-KBB-00 8-ABC-001 NRW-5-163
Gas Electric Hybrid Diesel
IS_A_CAR
FT4*(from SDR1): The car with LPNo <License_plate> is a car with electric power.
FT2: The car with LPno <License_plate> can drive <Range> kilometers on a full battery.
1-KBB-00 8-ABC-001
500 75
CAR_WITH_ELECTRIC_ POWER
Range RANGE <M>
CAR_WITH_TOW_HITCH
Max_load LOAD
FT5: The car with LPNo <License_plate> is a car with a tow hitch.
FT3: The car with LPno <License_plate> can tow a load of at most <Max_load> kg.
1-KBB-00 NRW-5-163
600 --
SDR1: x IN CAR_WITH_ELECTRIC_POWER IF (x, 'Electric') OR (x, 'Hybrid') IN CAR(License_plate, Power_type)
No Subtype Derivation Rule can be given: this is a declarative subtype

**Figure 6.6** **ERD for car data: declarative subtype**

CAR_WITH_TOW_HITCH is not a derivable subtype, it is therefore a *declarative subtype.* Although at the conceptual level of a CDM there is nothing wrong with declarative subtypes, they are to be handled with care when a PDM is derived. See section 6.4 for an explanation of the consequences of using a declarative subtype for generating a relational schema.

At this stage however, BR2 has now been modeled using a subtype as well, and the benefits of modeling BR2 in a visually clear way are obvious, although it should ring a bell that an equivalent modeling without a subtype is not possible. In figures 6.2 and 6.3, there is no way to model BR2!

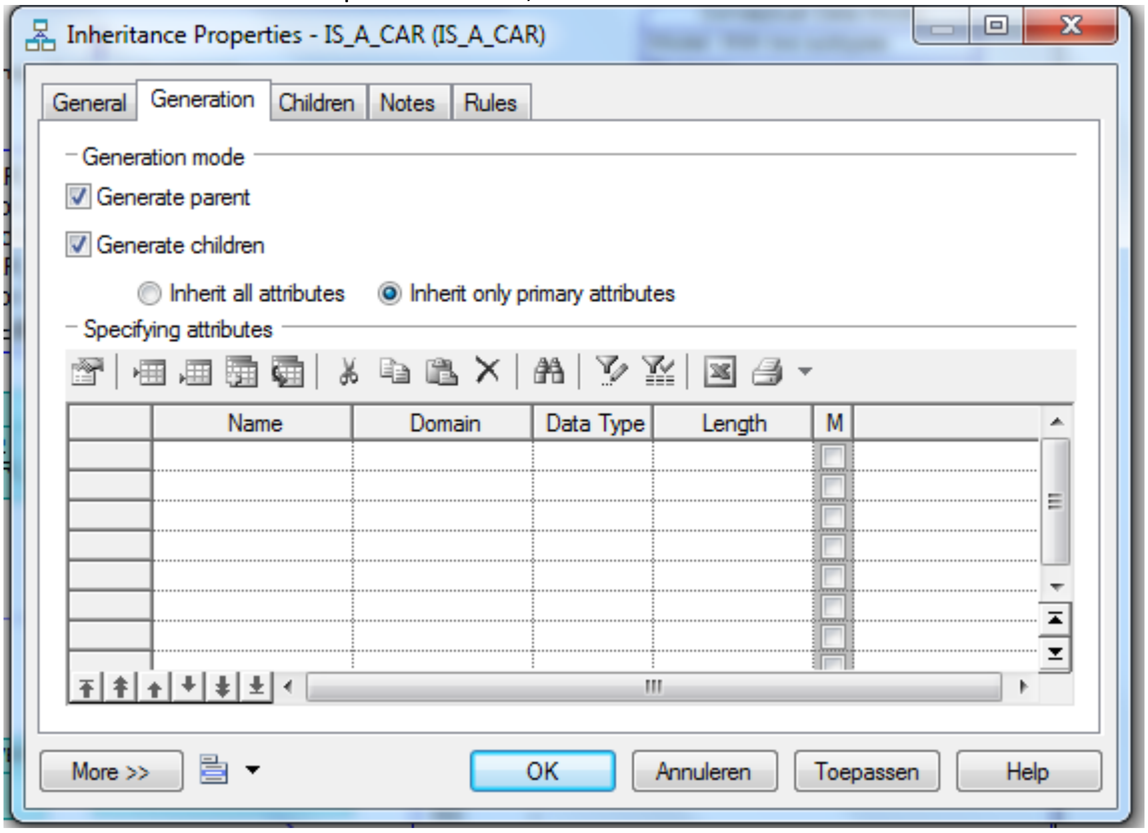
The warning above will be explained further in section 6.4. The following recommendation will also be further explained there: If it is not possible to specify a Subtype Derivation Rule, a declarative subtype can be used. However, this is not recommended. Instead, it is advised to add extra fact types, to enable a Subtype Derivation Rule to be stated and a derivable subtype to be used after all.

## 6.4 Deriving a PDM from a CDM with subtypes

One of the benefits of working with subtypes is that subtypes offer great flexibility in designing table structures, because several choices can be made for how to generate tables from subtypes. For example, from the CDM in figure 6.4, three tables can be generated (one for the supertype and one for each subtype), or only one table (for the supertype, absorbing the subtype-attributes, replacing the subtypes with constraints to keep satisfying the business rules), or two tables (absorbing one subtype but not the other). This section will discuss two main choices in detail, and the drawbacks of using declarative subtypes are shown as well. In section 6.5 a few other possibilities will be briefly indicated.

### 6.4.1 Setting the options in the Inheritance Properties Generation screen

In PowerDesigner, each inheritance link offers several possibilities for generating tables. Figure 6.7 shows the Inheritance Properties screen, tab Generation.


Inheritance Properties - IS_A_CAR (IS_A_CAR)

General | Generation | Children | Notes | Rules

Generation mode

[✓] Generate parent

[✓] Generate children

( ) Inherit all attributes [● Inherit only primary attributes]

Specifying attributes

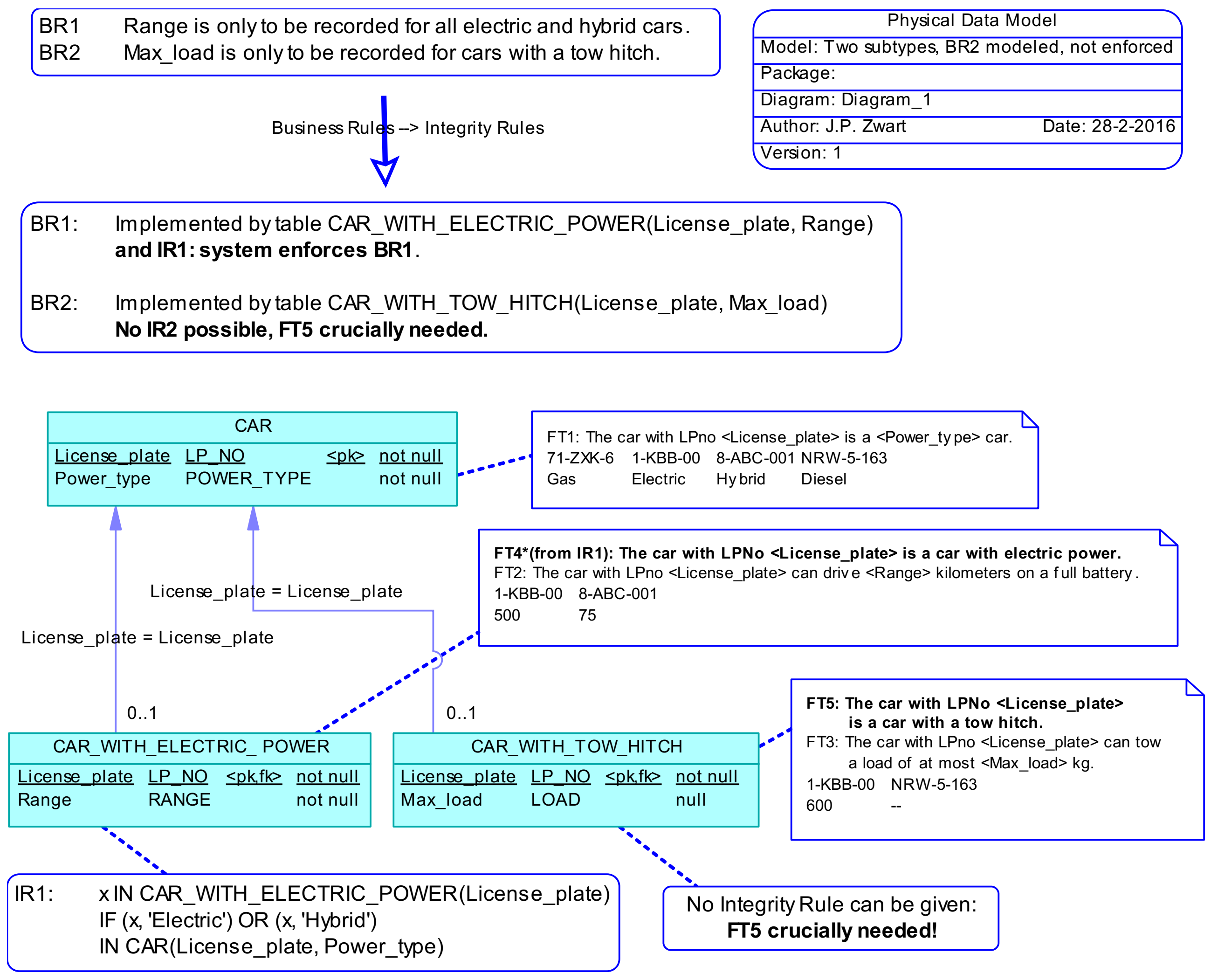
| | Name | Domain | Data Type | Length | M |
|---|---|---|---|---|---|
| | | | | | ☐ |
| | | | | | ☐ |
| | | | | | ☐ |
| | | | | | ☐ |
| | | | | | ☐ |
| | | | | | ☐ |
| | | | | | ☐ |

**Figure 6.7** **Inheritance properties to generate parents and children**

The meaning of several options:

- Generate parent: derive a table for the supertype
- Generate children: derive tables for the subtypes (can be fine-tuned using tab Children)
- Inherit all attributes: copy <u>all</u> supertype table attributes into the subtype tables (only to be chosen in a few special cases, see section 6.5)
- Inherit only primary attributes: copy only the <pk> of the supertype table into the subtype tables as <pk, fk> (default choice, valid in most cases)

Here, the checkboxes and radio button have been set so a table will be generated for the supertype and for each subtype, with the child tables only receiving a <pk,fk> to the parent table. This is a commonly made choice. Deriving a table for the supertype and for each subtype
Figure 6.8 shows the result of generating the PDM from the CDM in figure 6.6, if the generation settings are as shown in figure 6.7.



BR1 Range is only to be recorded for all electric and hybrid cars.
BR2 Max_load is only to be recorded for cars with a tow hitch.

Physical Data Model
Model: Two subtypes, BR2 modeled, not enforced
Package:
Diagram: Diagram_1
Author: J.P. Zwart Date: 28-2-2016
Version: 1

Business Rules --> Integrity Rules

BR1: Implemented by table CAR_WITH_ELECTRIC_POWER(License_plate, Range)
**and IR1: system enforces BR1.**

BR2: Implemented by table CAR_WITH_TOW_HITCH(License_plate, Max_load)
**No IR2 possible, FT5 crucially needed.**

| CAR | | | |
|---|---|---|---|
| License_plate | LP_NO | <pk> | not null |
| Power_type | POWER_TYPE | | not null |

FT1: The car with LPno <License_plate> is a <Power_type> car.

| 71-ZXK-6 | 1-KBB-00 | 8-ABC-001 | NRW-5-163 |
|---|---|---|---|
| Gas | Electric | Hybrid | Diesel |

**FT4*(from IR1): The car with LPNo <License_plate> is a car with electric power.**
FT2: The car with LPno <License_plate> can drive <Range> kilometers on a full battery.

| 1-KBB-00 | 8-ABC-001 |
|---|---|
| 500 | 75 |

| CAR_WITH_ELECTRIC_ POWER | | | |
|---|---|---|---|
| License_plate | LP_NO | <pk,fk> | not null |
| Range | RANGE | | not null |

| CAR_WITH_TOW_HITCH | | | |
|---|---|---|---|
| License_plate | LP_NO | <pk,fk> | not null |
| Max_load | LOAD | | null |

**FT5: The car with LPNo <License_plate> is a car with a tow hitch.**
FT3: The car with LPno <License_plate> can tow a load of at most <Max_load> kg.

| 1-KBB-00 | NRW-5-163 |
|---|---|
| 600 | -- |

IR1: x IN CAR_WITH_ELECTRIC_POWER(License_plate)
IF (x, 'Electric') OR (x, 'Hybrid')
IN CAR(License_plate, Power_type)

No Integrity Rule can be given:
**FT5 crucially needed!**

**Figure 6.8 PDM from CDM in figure 6.6, FT5 needed**

Apart from the usual steps in deriving a PDM from a CDM (see chapter 5, substeps 1-10 and 12-15) for the inheritance link and its subtypes the following steps were taken:

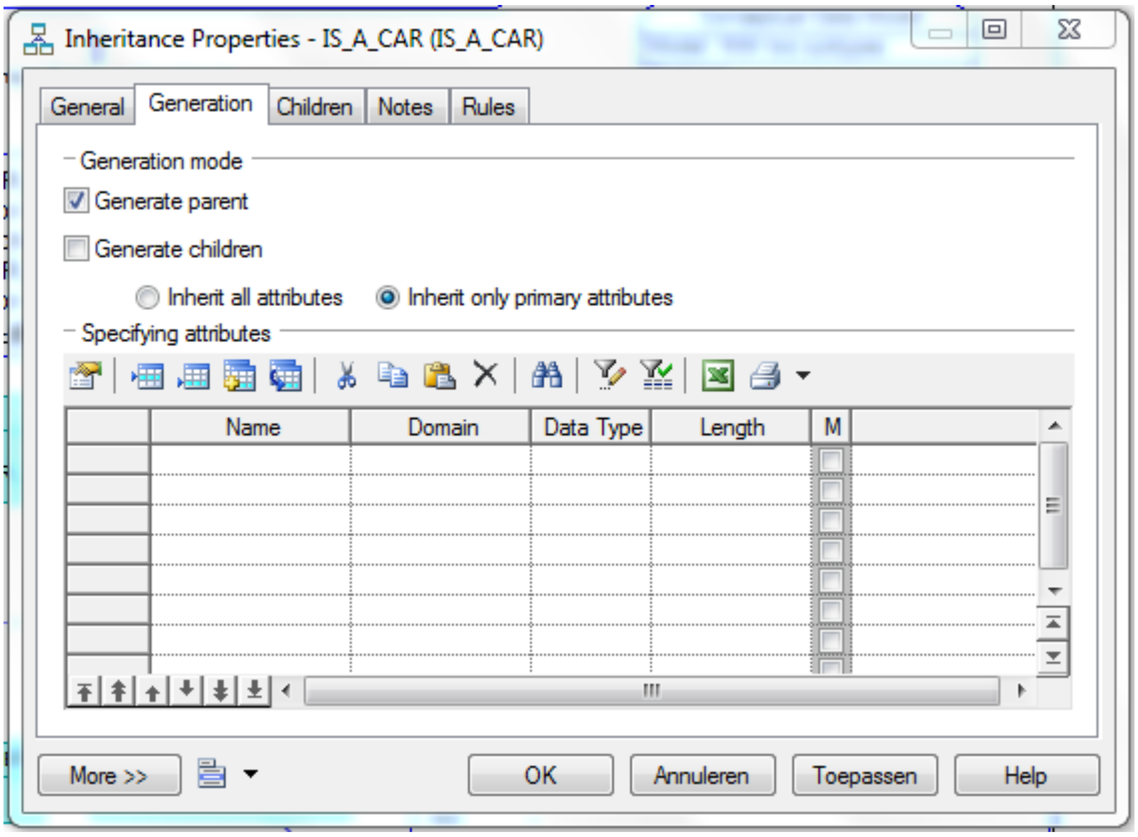
- A table was made for each subtype
- The <pk> of the parent table CAR was copied to all child tables, as <pf, fk>.
- The inheritance link was replaced by FK-references, one for each subtype.
- The cardinalities at the foot of these subtype-supertype FK references was set to 0..1 (because each CAR can belong at least ZERO CAR_WITH_TOW_HITCH, and at most ONE).
  **Note: PowerDesigner version 16.0 does this wrong and gives 0..* instead.**
  **Manual correction was needed.** An information modeler must always be able to check his/her software tools and correct its bugs.
- SDR1 was replaced with IR1

Note that the column CAR_WITH_TOW_HITCH(License_plate) is now crucially needed (with the subtype defining fact type FT5 as its verbalization). It contains as population all the cars with a towing hook, even those for which we do not know the maximum towing load.

The declarative subtype gave no problems here. Note that FT5 was needed in the model, although it was not present in the ERD of figures 6.2 and 6.3 yet, but was only added later because of the rule in figure 6.4.

### 6.4.2 Deriving a table for the supertype only

Another commonly made choice is to generate only one table for the supertype. This leads to fewer tables, but requires a more elaborate handling of constraints and integrity rules. Figure 6.9 shows the settings for this choice, which will however lead to a loss of information. The correct settings will be shown in figure 6.11, but first the arising problem is discussed.


Inheritance Properties - IS_A_CAR (IS_A_CAR)

General | Generation | Children | Notes | Rules

Generation mode

[✓] Generate parent

[ ] Generate children

( ) Inherit all attributes (●) Inherit only primary attributes

Specifying attributes

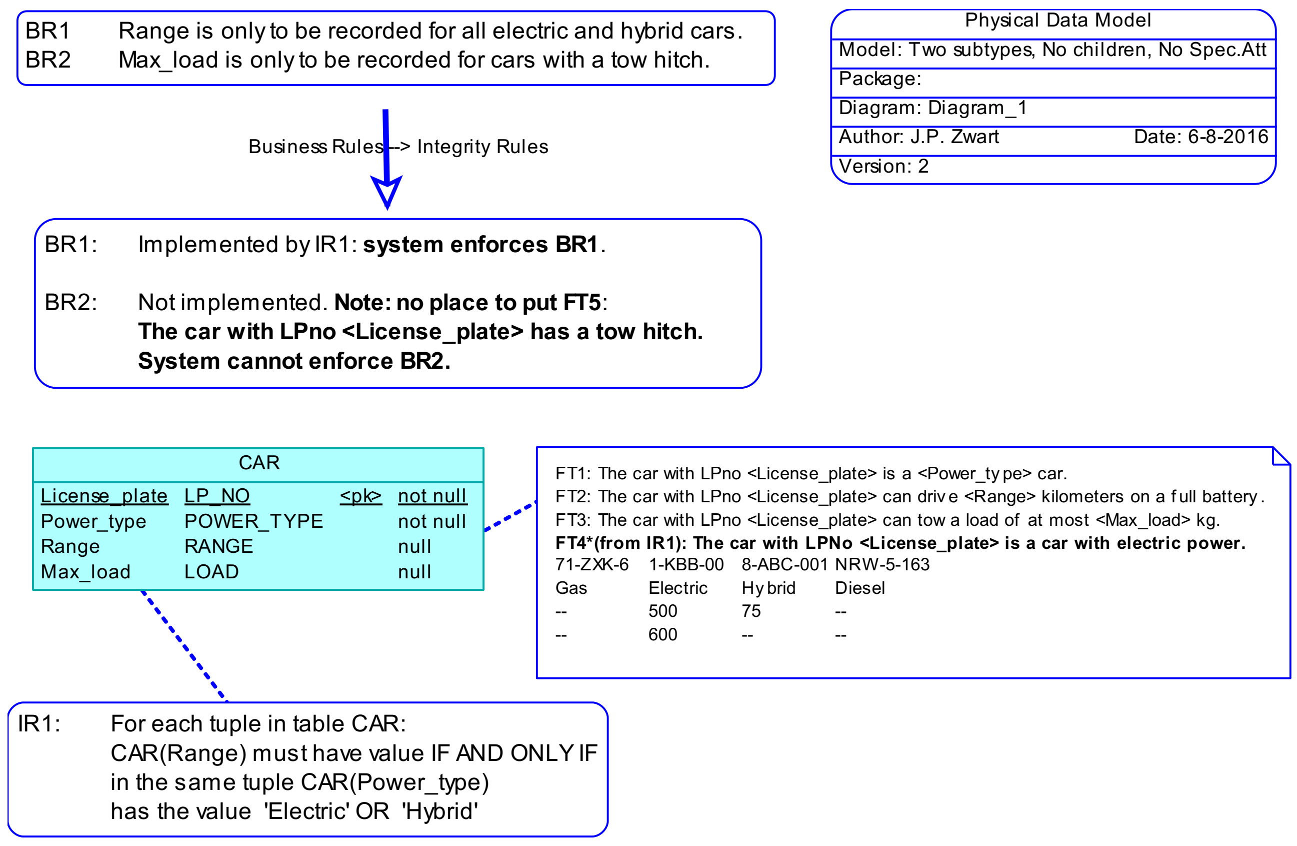
| | Name | Domain | Data Type | Length | M |
|---|---|---|---|---|---|

More >> | OK | Annuleren | Toepassen | Help

Figure 6.9 **WRONG settings to generate a table for the supertype only if there is a declarative subtype**

If checkbox 'Generate children' is not marked, the setting of the radio button for inheriting attributes doesn't matter (no subtype tables, therefore it doesn't apply).

All attributes of the subtypes will now be absorbed by the supertype as 'null'-columns. The translation of the other constraints into appropriate integrity rules must be done by hand (here especially SDR1 → IR1). The resulting PDM is shown in figure 6.10.



BR1 Range is only to be recorded for all electric and hybrid cars.
BR2 Max_load is only to be recorded for cars with a tow hitch.

Business Rules --> Integrity Rules

BR1: Implemented by IR1: **system enforces BR1**.

BR2: Not implemented. **Note: no place to put FT5**:
**The car with LPno <License_plate> has a tow hitch.**
**System cannot enforce BR2.**

| Physical Data Model | |
|---|---|
| Model: Two subtypes, No children, No Spec.Att | |
| Package: | |
| Diagram: Diagram_1 | |
| Author: J.P. Zwart | Date: 6-8-2016 |
| Version: 2 | |

CAR

| License_plate | LP_NO | <pk> | not null |
|---|---|---|---|
| Power_type | POWER_TYPE | | not null |
| Range | RANGE | | null |
| Max_load | LOAD | | null |

FT1: The car with LPno <License_plate> is a <Power_type> car.
FT2: The car with LPno <License_plate> can drive <Range> kilometers on a full battery.
FT3: The car with LPno <License_plate> can tow a load of at most <Max_load> kg.
**FT4*(from IR1): The car with LPNo <License_plate> is a car with electric power.**

| 71-ZXK-6 | 1-KBB-00 | 8-ABC-001 | NRW-5-163 |
|---|---|---|---|
| Gas | Electric | Hybrid | Diesel |
| -- | 500 | 75 | -- |
| -- | 600 | -- | -- |

IR1: For each tuple in table CAR:
CAR(Range) must have value IF AND ONLY IF
in the same tuple CAR(Power_type)
has the value 'Electric' OR 'Hybrid'

**Figure 6.10 PDM from CDM in figure 6.6 with settings from figure 6.9: information loss**

However this PDM has a problem: it is not possible to implement BR2, because there is no place to put FT5, so the information about which cars have tow hitches has disappeared. FT5 cannot be added to table CAR, because then all cars would have a tow hitch. This loss of information must be repaired.

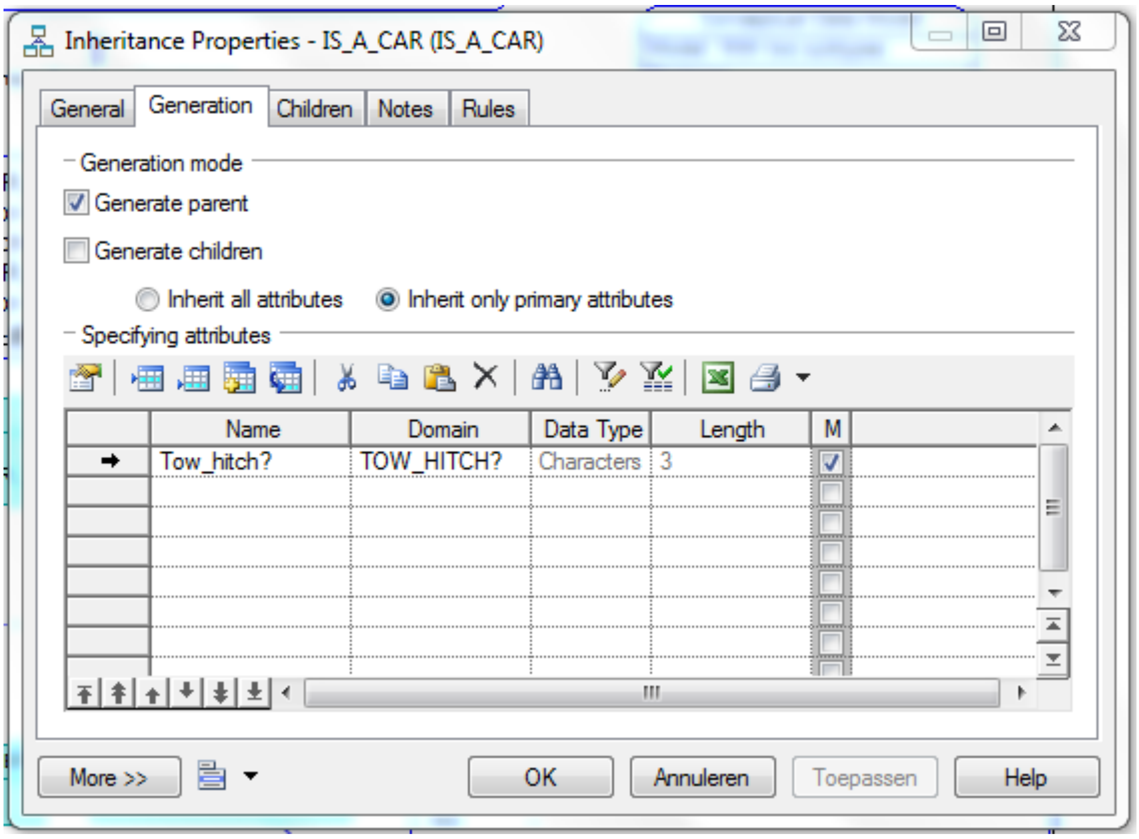
The cause of the information loss is that CAR_WITH_TOW_HITCH is a declarative subtype: the information which cars have tow hitches is stored in the subtype itself only, and cannot be derived from attributes of the supertype. The solution is then to add an attribute to the supertype that does contain this information. This can be done in the Inheritance Link Generation settings: it is necessary to enter a *specifying attribute* in the large area at the bottom, as is shown in figure 6.11.

The new attribute 'Tow_hitch?' will contain values 'Yes' or 'No' and (with the <pk>) represents facts that can be verbalized (in question-and-answer style) like:

```
          Does the car with LPno 71-ZXK-6 have a tow hitch? No.
            "   "   "    "    "  1-KBB-00    " "  "     "    Yes.
     FT6: Does the car with LPno <License_plate> have a tow hitch? <Tow_hitch?>.
```

Now the information is not lost if only one table is generated, and the result is shown in figure 6.11.

(Note: the same result is obtained if the specifying attribute is a column 'Tow_hitch?' that is optional, only contains 'Yes' for all cars with tow hitches, and NULL otherwise. Several other options are possible as well).



Inheritance Properties - IS_A_CAR (IS_A_CAR)

General | Generation | Children | Notes | Rules

Generation mode

[✓] Generate parent

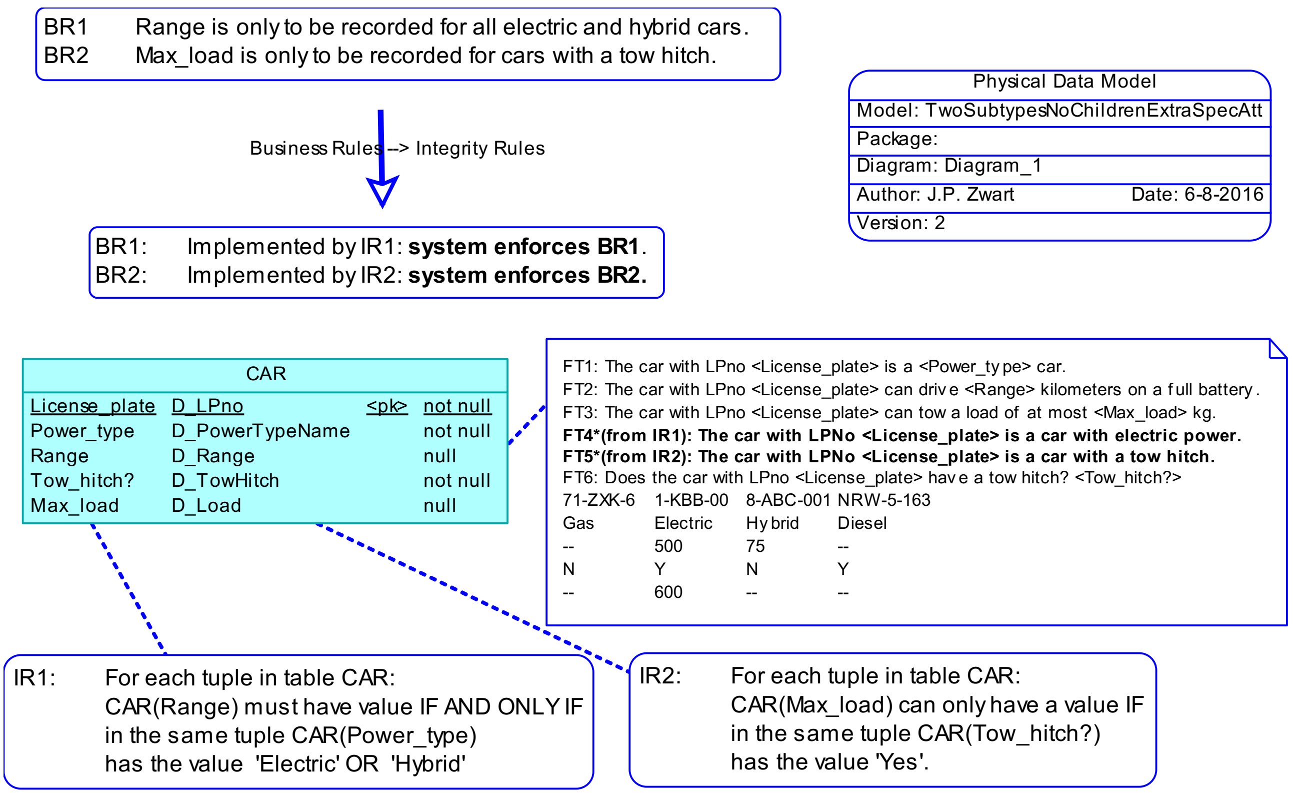
[ ] Generate children

○ Inherit all attributes ● Inherit only primary attributes

Specifying attributes

| | Name | Domain | Data Type | Length | M |
|---|---|---|---|---|---|
| ➜ | Tow_hitch? | TOW_HITCH? | Characters | 3 | ✓ |

**Figure 6.11 Correct settings to generate a table for the supertype only if there is a declarative subtype**



BR1 Range is only to be recorded for all electric and hybrid cars.
BR2 Max_load is only to be recorded for cars with a tow hitch.

Business Rules --> Integrity Rules

BR1: Implemented by IR1: **system enforces BR1**.
BR2: Implemented by IR2: **system enforces BR2**.

| Physical Data Model | |
|---|---|
| Model: TwoSubtypesNoChildrenExtraSpecAtt | |
| Package: | |
| Diagram: Diagram_1 | |
| Author: J.P. Zwart | Date: 6-8-2016 |
| Version: 2 | |

CAR

| | | | |
|---|---|---|---|
| License_plate | D_LPno | <pk> | not null |
| Power_type | D_PowerTypeName | | not null |
| Range | D_Range | | null |
| Tow_hitch? | D_TowHitch | | not null |
| Max_load | D_Load | | null |

FT1: The car with LPno <License_plate> is a <Power_type> car.
FT2: The car with LPno <License_plate> can drive <Range> kilometers on a full battery.
FT3: The car with LPno <License_plate> can tow a load of at most <Max_load> kg.
**FT4*(from IR1): The car with LPNo <License_plate> is a car with electric power.**
**FT5*(from IR2): The car with LPNo <License_plate> is a car with a tow hitch.**
FT6: Does the car with LPno <License_plate> have a tow hitch? <Tow_hitch?>

| | | | |
|---|---|---|---|
| 71-ZXK-6 | 1-KBB-00 | 8-ABC-001 | NRW-5-163 |
| Gas | Electric | Hybrid | Diesel |
| -- | 500 | 75 | -- |
| N | Y | N | Y |
| -- | 600 | -- | -- |

IR1: For each tuple in table CAR:
CAR(Range) must have value IF AND ONLY IF
in the same tuple CAR(Power_type)
has the value 'Electric' OR 'Hybrid'

IR2: For each tuple in table CAR:
CAR(Max_load) can only have a value IF
in the same tuple CAR(Tow_hitch?)
has the value 'Yes'.

**Figure 6.11 PDM from CDM in figure 6.6 with settings from figure 6.10: no information loss**

HAN_UNIVERSITY OF APPLIED SCIENCES

In figure 6.11, the new attribute 'Tow_hitch?' has been added to table CAR, and its predicate to the note associated with it. For completeness, FT4 and FT5, the two subtype defining fact types, are shown as well. Note that FT5 has now become derivable (from IR2).

Both FT4 and FT5 can be missed (they are after all derivable), but it is recommended to keep them in place to remind the modeler that the PDM actually contains subtypes, which is not easily visible from the constraints IR1 and IR2 alone.

It would have been possible to include the attribute 'Tow_hitch?' from the start, if the concrete examples in figure 6.1 had shown a tow hitch. It is left as an exercise to the reader to work this out completely: extend figures 6.2 and 6.6 with an attribute 'Tow_hitch?', and use a second derivable subtype in figure 6.6. Then generate figures 6.8 and 6.11 again, this time needing no specifying attribute at all.

The above discussion should make the reasons for the following recommendation clear:

**Recommendations and rules for working with subtypes**

- Use derivable subtypes as much as possible, unless there is a good reason not to do so. Add extra attributes to change declarative subtypes into derivable ones if needed.
- Always include a subtype defining fact type for each subtype (usually derivable) (see subtype rule 1 in figure 6.4).
- Always give a Subtype Derivation Rule for a derivable subtype, that only uses fact types from the supertype (see subtype rule 2 in figure 6.5).
- For any declarative subtype: beware of information loss when deriving a PDM and prevent this where necessary by adding specifying attributes.

**Figure 6.12 Recommendations and rules for working with subtypes**

### 6.4.3 Substep 11 in Deriving a PDM from a CDM

In chapter 5, about deriving a PDM from a CDM, substep 11 was skipped because subtypes had to be discussed first. From the discussion above, this substep is:

- For each subtype, decide whether or not a separate table for it is desired, and set the Inheritance Link Generation and Children properties correspondingly.
- For each declarative subtype, for which no separate table is desired:
  add specifying attributes to prevent information loss.
- Translate all subtype-related constraints from the CDM to integrity rules in the PDM. This must usually be done by hand.

## 6.5 Other possibilities to generate a PDM from a CDM with subtypes

Under certain conditions, other options are available too. In this section, one example is given, and the reader is encouraged to explore other avenues as well.

Considere the following context. In a hospital, patients are identified by a patient number. The surname and gender are to be recorded for all patients, the PSA-level only for male patients (not always known), the number of pregnancies only for female patients (always known). The ERD with two derivable subtypes in figure 6.13 was correctly drawn up. Only the SDRs and subtype defining fact types are shown for brevity.

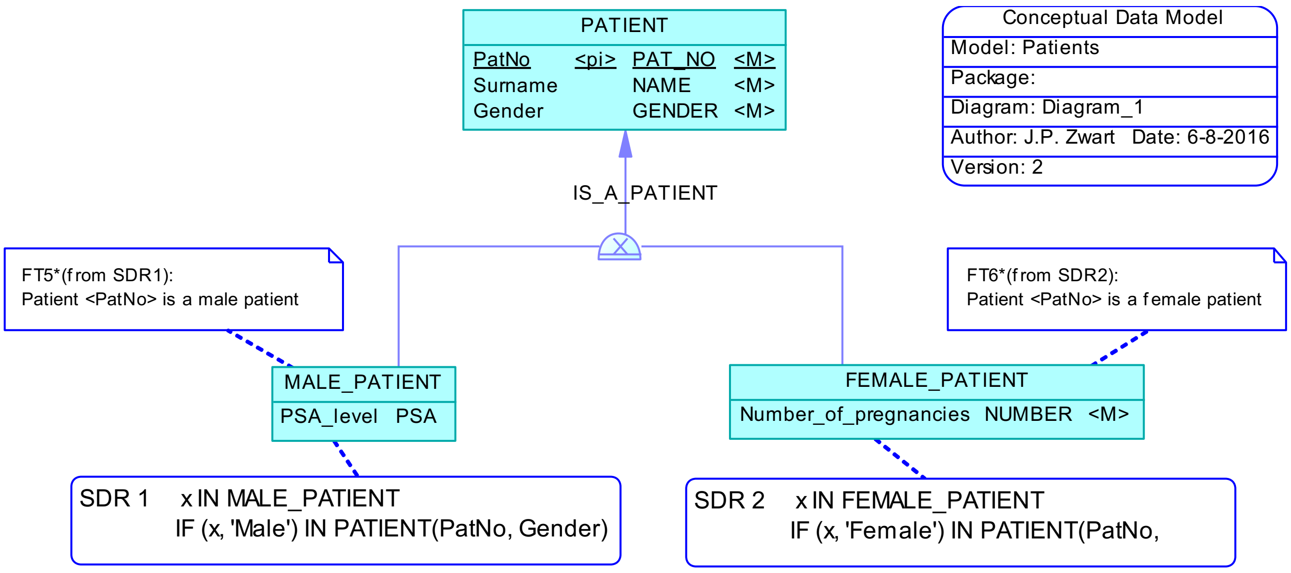

PATIENT
PatNo <pi> PAT_NO <M>
Surname NAME <M>
Gender GENDER <M>
Conceptual Data Model
Model: Patients
Package:
Diagram: Diagram_1
Author: J.P. Zwart Date: 6-8-2016
Version: 2
IS_A_PATIENT
FT5*(from SDR1): Patient <PatNo> is a male patient
FT6*(from SDR2): Patient <PatNo> is a female patient
MALE_PATIENT
PSA_level PSA
FEMALE_PATIENT
Number_of_pregnancies NUMBER <M>
SDR 1 x IN MALE_PATIENT IF (x, 'Male') IN PATIENT(PatNo, Gender)
SDR 2 x IN FEMALE_PATIENT IF (x, 'Female') IN PATIENT(PatNo,

**Figure 6.13: ERD for patients**

There are two new symbols in the inheritance link:

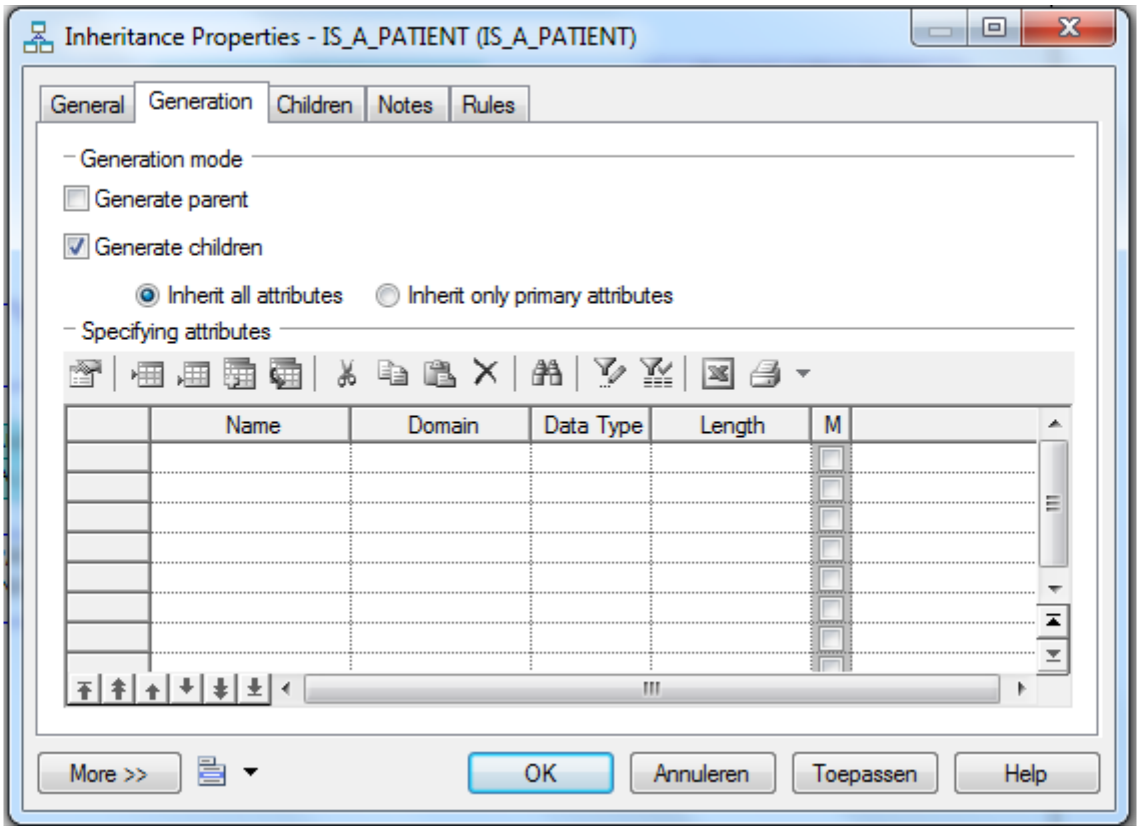
- The 'X' means that no entity from PATIENT can be in both MALE_PATIENT and FEMALE_PATIENT: the two subtypes are *exclusive*.
- The double line at the bottom of the half circle means that *every* entity from PATIENT is in one of the two subtypes: the two subtypes are *complete*.

If an inheritance link is both exclusive and complete, then it is possible to generate tables only for the subtypes, but not for the supertypes. In that case all the attributes from the supertype must be 'inherited' by the subtypes (the surname and gender would otherwise be lost because no supertype table is generated).

The settings shown in figure 6.14 will accomplish this, and the resulting PDM is shown in figure 6.15.

Several other possibilities exist as well, for instance deleting the Gender attribute from the supertype, which turns the two derivable subtypes into declarative ones, and FT5 and FT6 into non-derivable fact types needed in the resulting tables. This would improve the PDM (on which points?). This and other variants are left for the reader to explore further.

In such cases, care must always be taken to avoid loss of information. The Fact-Oriented approach (explicitly keeping track of the subtype defining fact types) is however a great help in this otherwise difficult area of information modeling.


Inheritance Properties - IS_A_PATIENT (IS_A_PATIENT)

General | Generation | Children | Notes | Rules

Generation mode

Generate parent: ☐

Generate children: ☑

[✓ Inherit all attributes] Inherit only primary attributes

Specifying attributes

| | Name | Domain | Data Type | Length | M |
|---|---|---|---|---|---|

**Figure 6.14** **Settings to generate children only (for exclusive and complete subtypes only)**

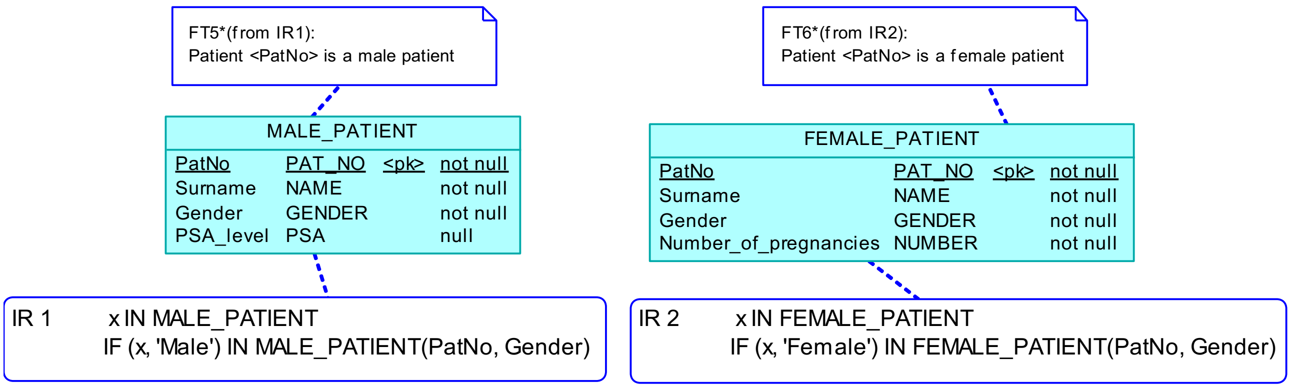

FT5*(from IR1):
Patient <PatNo> is a male patient
MALE_PATIENT
PatNo PAT_NO <pk> not null
Surname NAME not null
Gender GENDER not null
PSA_level PSA null
FT6*(from IR2):
Patient <PatNo> is a female patient
FEMALE_PATIENT
PatNo PAT_NO <pk> not null
Surname NAME not null
Gender GENDER not null
Number_of_pregnancies NUMBER not null
IR 1 x IN MALE_PATIENT IF (x, 'Male') IN MALE_PATIENT(PatNo, Gender)
IR 2 x IN FEMALE_PATIENT IF (x, 'Female') IN FEMALE_PATIENT(PatNo, Gender)

**Figure 6.15** **PDM from the CDM in figure 6.13 with the settings in figure 6.14**

## 6.6 Analyzing fact expressions with subtypes

In the previous sections, subtypes were discussed without analyzing fact expressions. Below, the verbalizations of the two fact types that concern the subtypes are analyzed, showing how this can be done. Other cases (from figure 6.13 for instance) are left as exercise to the reader.

Suppose that ET CAR has already been analyzed as ET in FT1.

```
FT2
The car with LPno 1-KBB-00 can drive 500 kilometers on a full battery.
 "   "    "    "  8-ABC-001 "    "    75     "       " "   "     "   .
ET CAR_WITH_ELECTRIC_POWER           Att Range
SUBTYPE(derivable) OF ET CAR
                      MATCH
The car with LPno <License_plate> can drive <Range> kilometers on a full
battery.
```

The derivation rule for the derivable subtype must eventually be specified as well (see figure 6.3 for the result).

```
FT5 (Note: declarative subtype defining fact type: only one segment)
The car with LPno 1-KBB-00 is a car with a tow hitch.
 "   "    "    "  NRW-5-163 " "  "    "  "  "    "  .
ET CAR_WITH_TOW_HITCH
SUBTYPE(declarative) OF ET CAR
                        MATCH

FT3
The car with LPno 1-KBB-00 can tow a load of at most 600 kg.
ET CAR_WITH_TOW_HITCH                                Att Max_load
MATCH
```

Because the subtype defining fact type of a declarative subtype is an indispensible fact type as well, it should be analyzed along with the other fact types. The derivable subtype defining fact types of derivable subtypes don't have to be analyzed. If in doubt at the analysis stage whether a subtype is derivable or not, play it safe and analyze it anyway (better one analysis too many than one too few).

# 7 References

[1] Fact Oriented Modeling with FCO-IM:
Capturing Business Semantics in Data Models
with Fully Communication Oriented Information Modeling.
Jan Pieter Zwart, Marco Engelbart, Stijn Hoppenbrouwers
2015 Technics Publications, www.technicspub.com
ISBN print ed. 978-1-63462-086-4
ISBN Kindle ed. 978-1-63462-087-1
ISBN ePub ed. 978-1-63462-088-8
ISBN PDF ed. 978-1-63462-089-5

[2] Volledig Communicatiegeoriënteerde Informatiemodellering FCO-IM
Guido Bakema, Jan Pieter Zwart, Harm van der Lek
1996 Academic Service, www.academicservice.nl
ISBN 90-395-2418-1
NUR 992

[3] Fully Communication Oriented Information Modeling (FCO-IM)
2002 English translation of [2]
Freely downloadable from www.fco-im.nl

[4] https://en.wikipedia.org/wiki/Boyce–Codd_normal_form
16 January 2019

# Appendix A: Detailed procedure for analyzing fact expressions

Here is a detailed working procedure to draw up an ERM Diagram from verbalizations of concrete examples of facts. Many fact expressions will be easy to process, and you will soon be able to do those without following this procedure. But it should also cater for the difficult ones, and you can consult the procedure if needed.

1 **Sort the fact expressions into fact types.**
   a **Place sentences of the same kind together in a group (fact type).**
   b **Count the number of components (see section 4.4.1) in each fact type**.
   c **Arrange the fact types in ascending order of the number of components.**
   Order the fact types:
      first the ones with one component (if any),
      then the ones with two components (if any),
      then the ones with three components (if any),
      and so on.
   The order of fact types with the same number of components doesn't matter.

**2 Analyze the fact types.**

For each fact type, carry out the steps 2a – 2d below.

**Process the fact types in ascending order of the number of components.**

The fact types with the fewest components are to be treated first, those with the most components last (see also step 2a below). For fact types with the same number of components, it doesn't matter much in which order they are analyzed, though some choices will prove to be more convenient than others. Experience will give you a good feeling for this, but the procedure should work in any order anyway.

**a Choose one of the three possible options below to mark segments.**

There is a firm rule about segments for analyzing fact types in ERM:

**Segment rule: In ERM, a fact type can contain at most two segments.**

No matter how many components a fact type has, two segments can be marked at most (only one segment is actually a rare case, see section 4.4.5 for a few examples). If there are more than two components, there are several possibilities to choose two segments. By treating the fact types with the fewest components first, this should cause no serious difficulties, because components that belong together then have already been determined earlier and can be easily recognized.

There are only three possibilities to mark segments in a fact type. Choose one of them.

- i **All components belong to only one segment.**
  This is actually a rare case, see section 4.4.5 for examples.
  The segment can only be for an ET, not for an Att.
- ii **The components belong to two segments: one for an ET and one for an Att.**
  This option implies that the Att belongs to the *same* ET: the Att must be one of the own Atts of the ET.
- iii **The components belong to two segments: both for an ET.**

**b Process the marked segments**

```
i    IF      Option 2ai was chosen (only one segment)
ii   THEN
     (1)     Carry out step 3 for this ET segment, then continue with step 2c.
iii  ELSEIF  Option 2aii was chosen (one ET segment and one Att segment)
iv   THEN
     (1)          Underline the Att segment with a single underlining.
     (2)          Give the Att a meaningful name
                  and write 'Att ' followed by the name under the underlining.
     (3)     Carry out step 3 for the ET segment, then continue with step 2c.
v    ELSEIF  Option 2aiii was chosen (two ET segments)
vi   THEN
     (1)   Carry out step 3 for the first segment, then continue with step 2 b vi (2).
     (2)   Carry out step 3 for the other segment, then continue with step 2 b vi (3).
     (3)   Carry out step 4 to add a RT, then continue with step 2c.
     ENDIF
```

**c Write the predicate (see section 4.4.1) as the last line in the analysis.**

**d Add the results of steps 2b and 2c to the diagram according to step 5.**

```
3   Process an ET-segment
    a   Underline the ET-segment with a double underlining.
    b   Give the ET a meaningful name
        and write 'ET ' followed by the name under the underlining.
    c   IF   it is an old ET (the ET has already been determined earlier)
    d   THEN
        i   Write 'MATCH' under the name of the ET.
            This completes step 3 for an old ET.
    e   ELSE it is a new ET (the ET has not been determined earlier)
        i   Determine its primary identifier (<pi>)
            There are two possibilities:
            •  The ET is strong, so the <pi> is simply one or more new Atts
            •  The ET is weak, so the <pi> comes from the <pi>s of one or more other ETS
               as parent ETs (usually already determined earlier),
               in combination with zero, one or more new Atts.
        ii  IF  the ET is strong
        iii THEN
            (1) Write under the ET name:
                'ID: Att ' followed by the name of the <pi>-Att
            (2) followed by '+ Att ' and the next <pi>-Att (if any), and so on.
                This completes step 3 for a new strong ET.
        iv  ELSE  the ET is weak
            (1) Write under the ET name:
                'ID: ET ' + the name of the parent ET
            (2) followed by '+ ET ' and the next parent ET (if any), and so on,
            (3) followed by '+ Att ' and the name of a <pi>-Att (if any), and so on.
            (4) FOR EACH parent ET from step 3 e iv (1) and step 3 e iv (2) (if any):
                (a) Carry out step 3c for this parent ET,
                    then continue with step 4 to add a dependent RT.
                    Note: Usually, the parent ET is old.
                This completes step 3 for a new weak ET.
            ENDIF
    ENDIF
```

```
4   Add a RT between two ETs
    a   IF      both ETs are strong OR both ETs are old
    b   THEN
            i       Add a RT without dependencies between the two ETs
            ii      Give the RT a meaningful name (starting with 'R_')
            iii     Write a line under the analysis so far according to the following template:
                    RT <RT name> between <ET name> and <ET name>.
            Note: the four cardinalities and at least one role are still to be determined. This is
            however described in step 5, which also deals with constraints.
    c   ELSE    There is one child ET and one parent ET
            i       Add a RT between the two ETs that is dependent at the side of the child ET
            ii      Give the RT a meaningful name (starting with 'R_')
            iii     Write a line under the analysis so far according to the following template:
                    RT <RT name> between <child ET name>(dependent) and <parent ET name>.
            Note: the two cardinalities at the child ET side and at least one role are still to be
            determined (the other two cardinalities are both ONE). This is however described in step
            5, which also deals with constraints.
        ENDIF
```

```
5     Add the analysis results to the diagram and determine constraints
      Read the analysis from top to bottom, and carry out the following steps in the ERM diagram:
      a     Choose an ET segment from the top level of the analysis.
      b     IF    The ET is new (it has ‘ID’ written under its name)
      c     THEN
            i     Add the new ET to the diagram
            ii    IF    The ET has own <pi>-attributes (its ID contains ‘Att’ parts)
            iii   THEN
                  (1)   Add all the <pi>-Atts to the new ET
                  ENDIF
            iv    IF    The new ET is weak (its ID contains ‘ET’ parts)
            v     THEN    FOR EACH parent ET
                  (1)   Carry out step 5b for this parent ET, then continue with step 5 c v (2).
                        Note: usually the parent ET already exists.
                  (2)   Add the RT (determined in step 4) from the child ET to the parent ET.
                        It is dependent at the side of the child ET.
                  (3)   Determine the two cardinalities at the side of the child ET.
                        The other two at the side of the parent ET are necessarily both ONE.
                  (4)   Write the reason for each cardinality in the constraint documentation.
                  (5)   Add a role (see section 3.1.11) ‘in’ or ‘of’ to the RT.
                  ENDIF
            ENDIF   The ET already exists in the diagram
      d     IF    The fact type has another segment for an Att
      e     THEN
            i     Add the Att to the ET of the first segment.
            ii    Determine whether or not the Att has a <M> constraint.
            iii   Write the reason for this decision in the constraint documentation.
            ENDIF
      f     IF    The fact type has another segment for an ET
      g     THEN
            i     Carry out steps 5b and 5c, but not 5d, for this ET, then continue with step 5 g ii.
            ii    Add the RT (determined in step 4) between the two ETs to the diagram.
            iii   Determine the cardinalities for this RT.
            iv    Write the reason for each cardinality in the constraint documentation.
            v     Give at least one role (see section 3.1.11).
                  If the RT has a dependency, then it is recommended to use only ‘of’ at the side of
                  the child ET, or ‘in’ at the side of the parent ET. Otherwise it is best to use a
                  clarifying verb.
                  ENDIF
            ENDIF
```