Database Modeling for Relational DBs

Book · January 2016

CITATIONS
0

READS
1,264

1 author:

Fernando Almeida
Instituto Superior Politécnico Gaya
112 PUBLICATIONS **280** CITATIONS

SEE PROFILE

INESC TEC

Database Modeling for Relational DBs

Fernando Almeida, PhD.

2016

ABSTRACT

This book provides a brief reference about the process of modeling a relational database. For that, it uses the UML classes diagram for identifying the classes, attributes and relationships. Then, it presents how to map the UML classes diagram in a relational model, by identifying primary keys, foreign keys, attributes and other domain constraints. Along the book we describe the concepts involved in the process of modeling a relational database and we give eight practical examples how to model a relational database.

CAMPUS DA FEUP
RUA DR. ROBERTO FRIAS

Table of Contents

Index of Figures

Acronyms

DBMS - Database Management System

EU - European Union

SQL - Structured Query Language

UML - Unified Modeling Language

Glossary

Null - empty field in a database.

Relation - an entity describing an object. In a database it has the same meaning as a table, but the "relation" term is more conceptual.

String - an object type composed by alphanumeric elements.

Trigger - a SQL procedure that initiates an action (i.e., fires an action) when an event (INSERT, DELETE or UPDATE) occurs.

Tuple - an ordered list of elements. This ordered is only determined by the positioning of attributes when the programmer creates the table.

1. Introduction

1.1 Contextualization

A database consists of logically related data stored in a single repository. Typically, it provides advantages over file system traditional management approaches by eliminating inconsistency, data anomalies, data dependency, and structural dependency problems. A database system stores data structures, relationships, and access paths.

Databases are important structures in a wide range of different domains since business to government and educational contexts. Associated to databases are an important role of advantages respectively:

- Makes data management more efficient and effective;
- Query language (e.g., SQL) allows quick answers to ad-hoc queries;
- Provides better access to more and better managed data;
- Promotes integrated view of organization's operations;
- Reduces the probability of inconsistent data.

Associated to the concept of a database appears the notion of Database Management Systems (DBMS). The DBMS essentially serves as an interface between the database and end users of applications programs, ensuring that data is consistently organized and remains easily accessible. The DBMS manages three important things: the data, the database engine, and the database schema. These three foundational elements help provide concurrency, security, data integrity, and uniform administration procedures.

Database modeling is one of the first steps when building a database system. This step includes the collection of logical constructs used to represent data structure and relationships within the database. There are generically two approaches of database modeling:

1. Conceptual modeling: logical nature of data representation;
2. Implementation modeling: emphasis on how the data are represented in the database;

In the context of this book we are only interested to anlyze and detail the first approach.

1.2 Objectives

This mini book intends to provide a brief reference guide for undergraduate students that intend to model a relational model using the UML language. For that, we provide a concise theoretical and practical reference how to model a database using UML class diagrams and how to instantiate this representation in a relational model, by identifying primary keys, foreign keys, attributes and other domain constraints.

1.3 Book Structure

The book is organized in 5 chapters as follow:

- Chapter 1 "Introduction" - gives a brief overview about relational databases and briefly presents the organization of the book;
- Chapter 2 "UML Class Diagrams" - presents and describes the main elements of a UML class diagram;
- Chapter 3 "Relational Model" - presents the main concepts associated to the relational model and presents the notion of primary and foreign keys;

- Chapter 4 "Mapping to Relational Model" - describes the different approaches that should be adopted to transform the UML classes model into the relational model;
- Chapter 5 "Examples" - gives eight examples of relational models, starting by its modeling using UML class diagrams and also presents its relational model;
- "Bibliography" - presents the adopted bibliography for this book.

2. UML Class Diagrams

2.1 Overview

Unified Modeling Language (UML) is a widely-used and reference modeling language in the field of software engineering. Users can adopt UML to analyze, design, and implement software-based systems, along with other business processes.

UML is composed by a wide range of different diagrams that are mainly categorized in three groups:

- Structure diagrams - emphasize the structural elements that must be present in the system being modeled. They are frequently used to document the architecture of a software application. Examples of such diagrams are class diagrams, object diagrams or component diagrams;
- Behavior diagrams - emphasize the necessary behavior in the system being modeled. They are often used to describe software systems' functionality. Examples of such diagrams are use case diagrams, activity diagrams or state machine diagrams;
- Interaction diagrams - provide detail about the flow of control and data throughout the modeled system. Examples of such diagrams are sequence diagrams and timing diagrams.

Within the context of this book we focus only on class diagrams. They are used to provide an overview of the target system by describing the objects and classes in the system and the relationships between them. It provides a wide variety of usages: from modeling the domain-specific data structure to detailed design of the target system.

In next sections we provide a full description of the elements used by UML class diagrams.

2.2 Class

A class or entity is a general concept (represented as a square box) that represents an object in the real world. A class defines the structural attributes and behavioral characteristics of that concept. It is shown as a rectangle labeled with the class name (Figure 1).

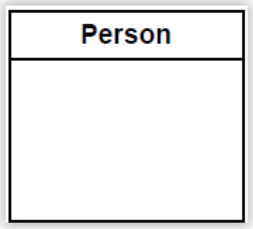

Figure 1 - Class representation

Figure 1 represents a simple class called "Person" with no attributes or behavior shown.

2.3 Association

An association indicates that the system being developed stores links of some kind between the classes. Generally, it does not imply anything about the implementation of the links, but presents what is the relationship type between the classes.

One class can be related to another in the following terms:

- One-to-one;
- One-to-many;
- One-to-zero;
- One-to-exactly n.

The multiplicity can be expressed as:

- Exactly one - 1;
- Zero or one - 0..1;
- Many - 0..* or *;
- One or more - 1..*
- Exact number - e.g., 2, 3, 4, 5, and so on;
- Complex relationship - e.g., 0..3; 3..5; 5..*, among others.

Figure 2 provides an example of two classes that are associated in a relationship one-to-one. The rlationship below states that in a given company only an employee can work there. Reading it from right to left, we can state a given employee works only in one company.

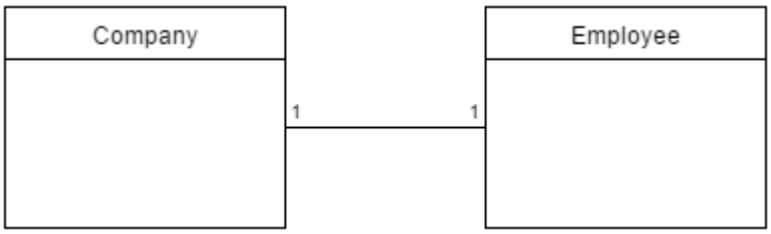

Figure 2 - Association one-to-one

In fact, the above relationship doesn't make much sense, because it expected that a company has several employees. In order to attend such situation we slightly change the above scenario for a one-to-many relationship (Figure 3). Now we can read it as a company that can have several employees and a given employee only can works in one company.

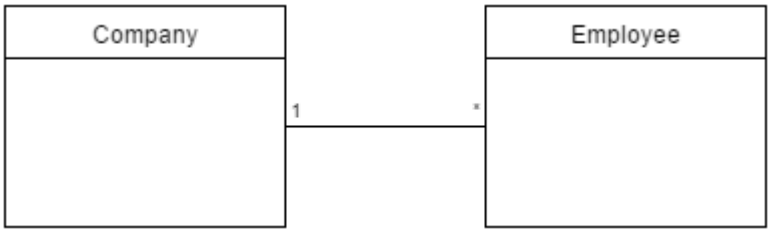

Figure 3 - Association one-to-many

If we want to express that a employee can work in more than one company we need again to change the above relationship. Then, we have a many-to-many relationship (Figure 4). Now, we can see that a company can have multiple employees and a given employee can work in several companies. The existence of a many-to-many relationships turns mandatory the existence of an additional class that will be responsible to manage the correspondences between companies and employees.

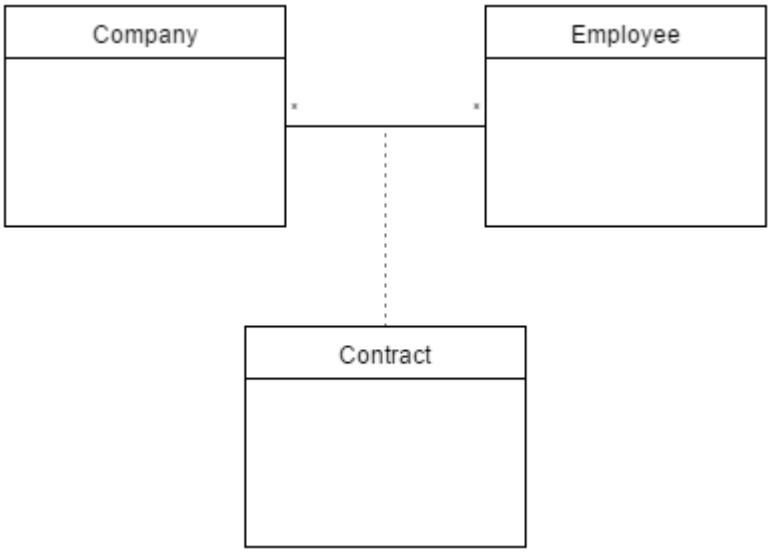

Figure 4 - Association many-to-many

There are also more possibilities of relationships between the above entities. For example we can say that an employee can work in no more than 2 companies (Figure 5). In other words, an employee can work only in one or two companies.

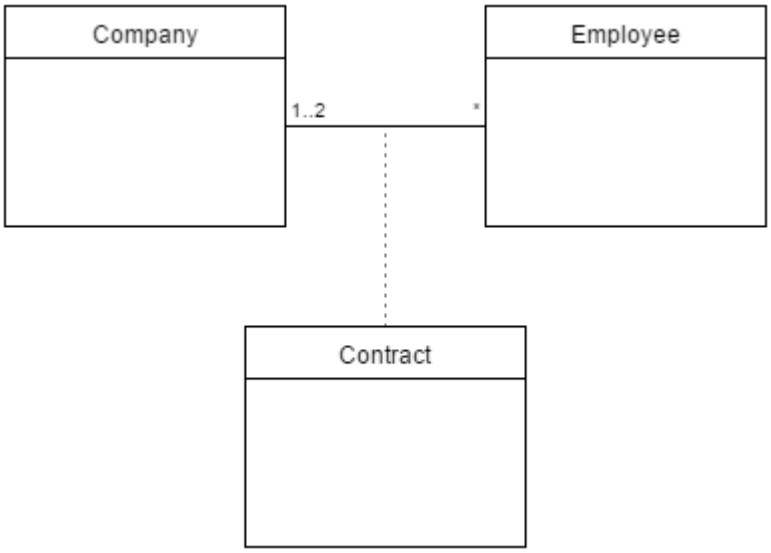

Figure 5 - Association 1 or 2-to-many

Other example could be the existence of companies without employees. In order to express that we need to explicitly state this situation in the class diagram (Figure 6).

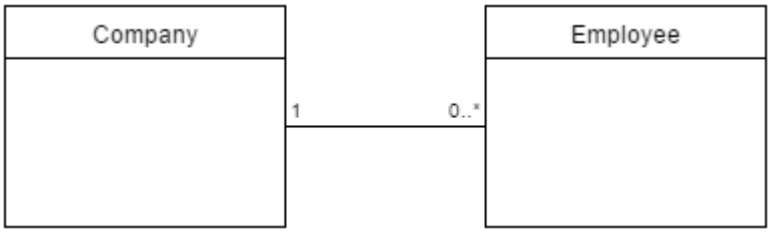

Figure 6 - Association one-to-zero until many

An association that connect a class to itself is called a self association. In this kind of association there is only one involved class. For example if we want to state that a company has employees and a single manager is responsible for up to workers, we can describe such situation as illustrated in Figure 7.

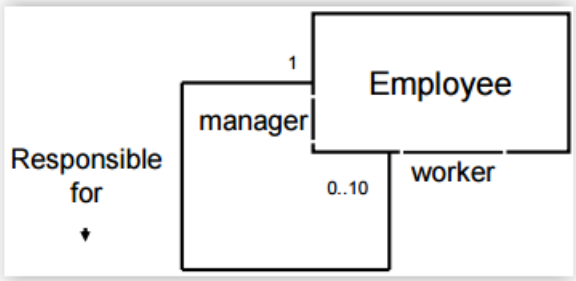

Figure 7 - Self association

2.4 Composition

The composition relationship shows that a given class has exclusive ownership over other class or classes, meaning that the container object and its parts constitute a parent-child/s relationship. In a composition relationship a Class A owns a Class B. Composition illustrates that a strong life cycle is present between the classes. Figure 8 expresses a situation where a person is composed by legs and hands. Therefore, it means that the class "Person" owns the classes "Leg" and "Hand". The existence of those two classes without "Person" doesn't make sense.

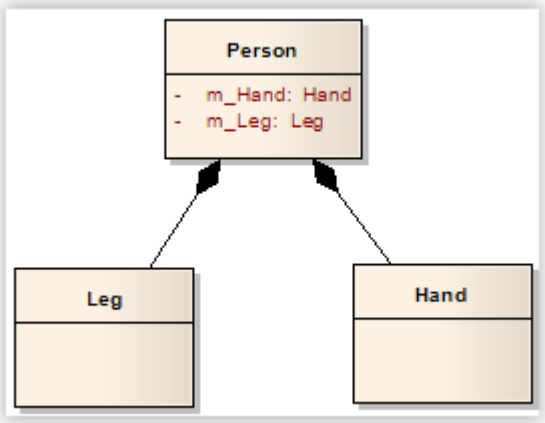

Figure 8 - Composition relationship

2.5 Aggregation

When a class is formed as a collection of other classes, it is called an aggregation relationship between these classes. It is also called a “has a” relationship. Aggregation is the weaker form of object containment (one object contains other objects). The stronger form is called Composition. In cases where there’s a part-of relationship between ClassA (whole) and ClassB (part), we can be more specific and use the aggregation link instead of the association link, highlighting that ClassB can also be aggregated by other classes in the application. Figure 9 illustrates a situation where a CD has a number of songs. The whole class is "CD" and "Song" is the part class.

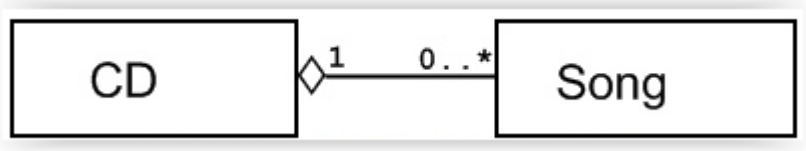

Figure 9 - Aggregation relationship

2.6 Generalization

A generalization is used to indicate inheritance. Drawn from the specific classifier to a general classifier, the generalize implication is that the source inherits the target's characteristics. Generalization is the ideal type of relationship that is used to showcase reusable elements in the class diagram. Literally, the child classes “inherit” the common functionality defined in the parent class. Figure 10 illustrates a situation where "Account" class is a generalization of "Current Account" and "Savings Account". It implies that all attributes of "Account" class are shared by his dependencies, but "Current Account" and "Savings Account" can have specific local attributes.

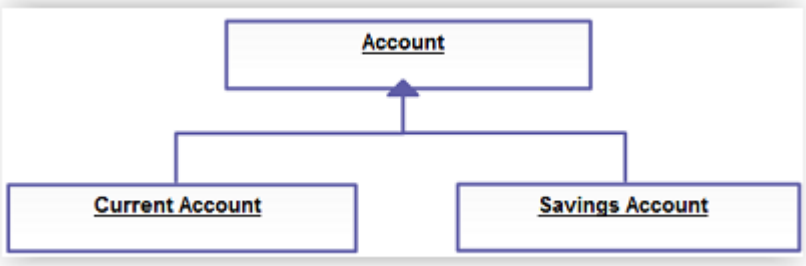

Figure 10 - Generalization relationship

2.7 Attributes

The attributes of a class represents the properties about each object of this class type. I other words, its represents the typical characteristics of an object. Figure 11 uses the same classes introduced in last section and adds attributes to classes "Company" and "Employee".

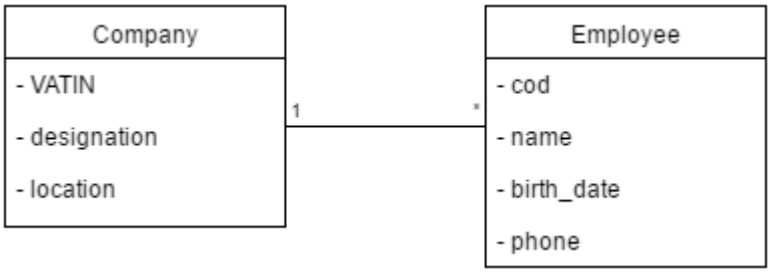

Figure 11 - Relationship of two classes with attributes inside each class

For each "Company" we know the following information:

- VATIN - an identifier used in many countries, including the countries of the European Union, for value added tax purposes;
- designation - name of registered company;
- location - city where the company is located.

For each "Employee" we register the following elements:

- cod - internal code used by the company to identify each employee;
- name - initial and surname of the employee;
- birth_date - birth date of the employee;
- phone - personal mobile phone number of the employee.

2.8 Operations

Operations is an optional feature of a class that is used to specify the operations that is supported by each class. The operations of a class typically changes the values of attributes and may interact with other classes or end-users. Figure 12 illustrates a scenario where each class offers a different number of operations or methods.

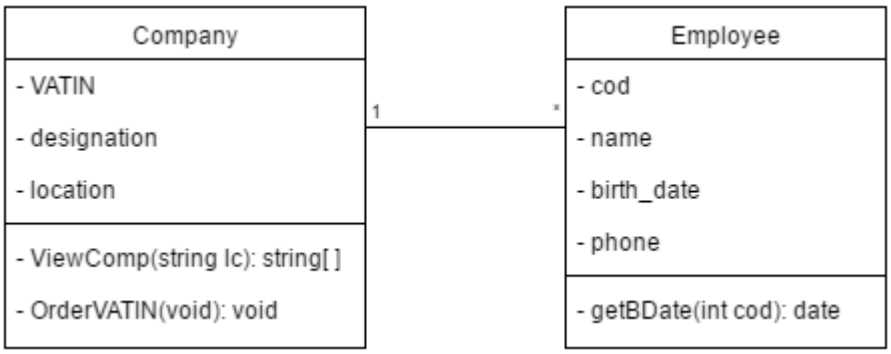

Figure 12 - Operations offered by classes

The operations/methods offered by "Company" class are:

- ViewComp -> it receives a string indicated a given location (city) and returns an array of strings of all companies located in that location;
- OrderVATIN -> this method doesn't receive or returns any data. It is used to order upwards the companies' VATIN.

The operations/methods available in the "Employee" class are:

- getBDate - it receives the cod of a given employee, which is an integer number, and returns the birth date of the given employee.

3. Relational Model

3.1 Overview

One of the most important application for computers is storing and managing information. The manner in which information is organized and stored can have a profound effect on how easy it is to access and manage. One of the simplest but most effective and versatile way to organized information is to store it in tables.

The relational model is centered on the idea that the organization of data into collections of two-dimensional tables called "relations". A relation is based in the mathematical model of set theory. In computational terms a relation is represented as a table.

The relational model was proposed by E.F.Codd in 1970 as a data model which strongly supports data independence. It was made available in commercial DBMSs since 1981. Codd establishes the following rules for the management and storage data using only its relational capabilities:

- Rule 1: Information Rule -> the data stored in a database, may it be user data or metadata, must be a value of some table cell. Everything in a database must be stored in a table format;
- Rule 2: Guaranteed Access Rule -> every single data element (value) is guaranteed to be accessible logically with a combination of table-name, primary-key (row value), and attribute-name (column value). No other means, such as pointers, can be used to access data;
- Rule 3: Systematic Treatment of Null Values -> the Null values in a database must be given a systematic and uniform treatment. This is a very important rule because a Null can be interpreted as one the following: data is missing, data is not known, or data is not applicable;
- Rule 4: Active Online Catalog -> the structure description of the entire database must be stored in an online catalog, known as data dictionary, which can be accessed by authorized users. Users can use the same query language to access the catalog which they use to access the database itself;
- Rule 5: Comprehensive Data Sub-Language Rule -> a database can only be accessed using a language having linear syntax that supports data definition, data manipulation, and transaction management operations. This language can be used directly or by means of some application. If the database allows access to data without any help of this language, then it is considered as a violation;
- Rule 6: View Updating Rule -> all the views of a database, which can theoretically be updated, must also be updatable by the system;
- Rule 7: High-Level Insert, Update and Delete Rule -> a database must support high-level insertion, updation, and deletion. This must not be limited to a single row, that is, it must also support union, intersection and minus operations to yield sets of data records;
- Rule 8: Physical Data Independence -> the data stored in a database must be independent of the applications that access the database. Any change in the physical structure of a database must not have any impact on how the data is being accessed by external applications;
- Rule 9: Logical Data Independence -> the logical data in a database must be independent of its user's view (application). Any change in logical data must not affect the applications using it. For example, if two table are merged or one is split into two different tables, there should be no impact or change on the user application;

- Rule 10: Integrity Independence -> a database must be independent of the application that uses it. All its integrity constraints can be independently modified without the need of any change in the application. This rule makes a database independent of the front-end application and its interface;
- Rule 11: Distribution Independence -> the end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only. This rule has been regarded as the foundation of distributed database systems;
- Rule 12: Non-Subversion Rule -> if a system has an interface that provides access to low-level records, then the interface must not be able to subvert the system and bypass security and integrity constraints.

3.2 Tables, Rows and Columns

In a relational database, data are stored in tables. A table is composed by several rows and columns, where we have the following situations:

- Each row is called a tuple;
- Each column is called an attribute;
- The relation schema of a table is the set of its attribute names.

Figure 13 illustrates an example of a table called "Prof" that stores information about professors. Regarding each professor we have information about its pid (internal control number), name, department, ranking and salary. For example the professor with name "Name" works in "CS" department, has a ranking of assistant and a salary of 6000.

PROF

pid	name	dept	rank	sal
*p*1	Adam	CS	asst	6000
*p*2	Bob	EE	asso	8000
*p*3	Calvin	CS	full	10000
*p*4	Dorothy	EE	asst	5000
*p*5	Emily	EE	asso	8500
*p*6	Frank	CS	full	9000
		...		

Figure 13 - Contents of table "Prof"

3.3 Primary Key

Every table must have a primary key. The primary key is a column or set of columns that identifies a particular row. Each table can have only one primary key that may be composed by a single attribute or multiple attributes.

There are some rules applied to a primary key, respectively:

- Unique - two or more rules can't have the same value;
- Not Null - every row has a value that can't be "Null";
- Minimal - the primary key should be short as possible while preserving the uniqueness. The shorter the possible, the less space it takes to store in the PK columns, then the savings are multiplied by each of the foreign key (FK) references.

Considering the classes "Company" and "Employee" we will check what can be the most suitable primary keys for each table (Figure 14). For the table "Company" the most suitable primary key can be the VATIN, considering it is an unique numerical number. "Designation" and "location" are not good primary keys because it can exists companies with the same designation (i.e., in several countries or states) and in the same location obviously may exist several companies. For the table "Employee" the most suitable primary key is "cod", considering that each employee has a different internal code. "Phone" could also be the primary key, but perhaps is not a so good solution because an employee may change his/her phone number along time. "Name" and "birth-date" are definitely not good choices because may exist duplicate values.

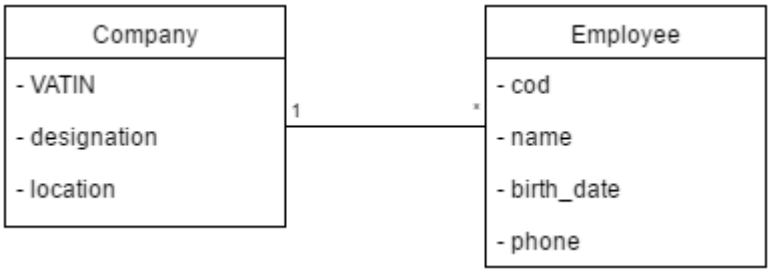

Figure 14 - Classes representation for definition of primary keys

Figure 15 illustrates another example about the choice of primary keys, but this time the solution is the adoption of a primary key composed by two attributes "ProductID" and "VendorID". Only using these two attributes together we can avoid the existence of duplicate rows.

Primary Key

ProductID	VendorID	AverageLeadTime	StandardPrice	LastReceiptCost
1	1	17	47.8700	50.2635
2	104	19	39.9200	41.9160
7	4	17	54.3100	57.0255
609	7	17	25.7700	27.0585
609	100	19	28.1700	29.5785

Figure 15 - Primary key composed by two attributes

In some situations, where there isn't a clear evidence what could be the most suitable primary key from the available attributes of a class or table, we can add an additional attribute to the table that will works as the primary key. Figure 16 illustrates a scenario where we have a table composed by four attributes. However, no one of them is a good option to be the primary key. In such conditions we can add to the table a new attribute, for example called "id", that will necessarily be unique for each row.

S_name	age	course	address

Figure 16 - Table without a good candidate primary key

3.4 Foreign Key

A foreign key is a column (or columns) that references a column (most often the primary key) of another table. The purpose of the foreign key is to ensure referential integrity of the data. In other words, only values that are supposed to appear in the database are permitted.

Figure 17 presents an example of two tables connected by a foreign key. The student table contains the "course_id" the student is attending. Another table lists the courses on offer with "course_id" being the primary key. These two tables are linked through "course_id" and as such "course_id" would be a foreign key in the student table.

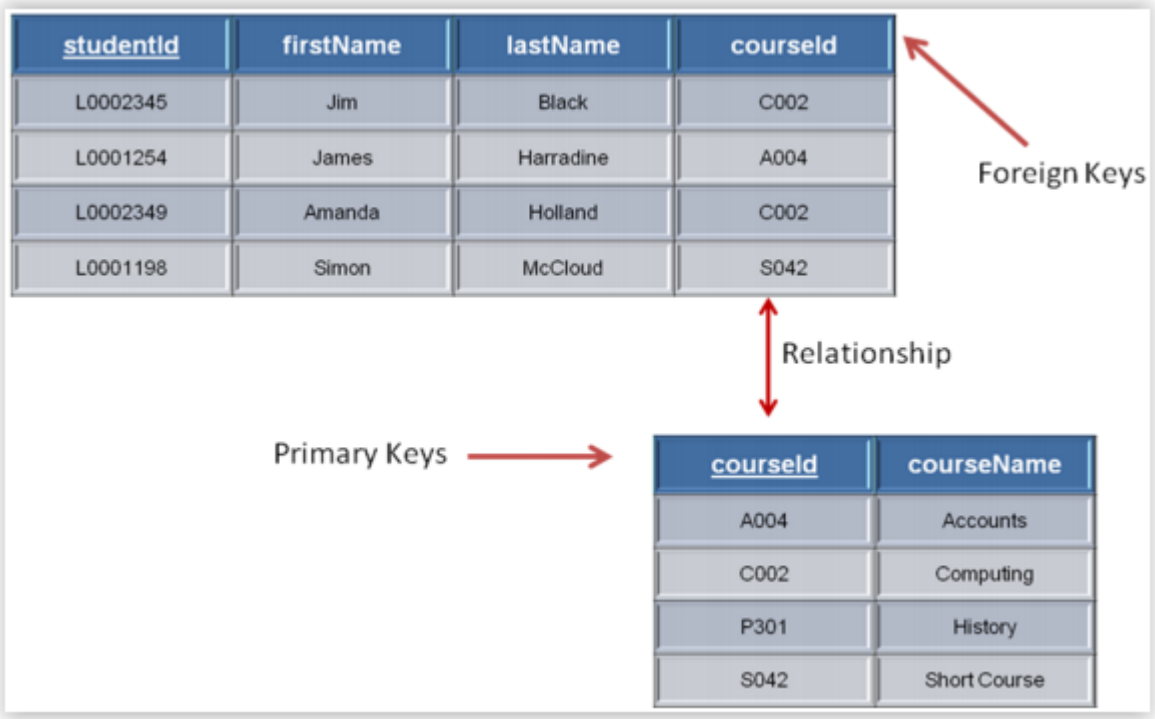

Figure 17 - Example of a foreign key

For any column acting as a foreign key, a corresponding value should exist in the link table. Special care must be taken while inserting data and removing data from the foreign key column, as a careless deletion or insertion might destroy the relationship between the two tables. Some referential actions associated with a foreign key action include the following:

- Cascade - when rows in the parent table are deleted, the matching foreign key columns in the child table are also deleted, creating a cascading delete;
- Set Null - when a referenced row in the parent table is deleted or updated, the foreign key values in the referencing row are set to null to maintain the referential integrity;
- Triggers - referential actions are normally implemented as triggers. In many ways foreign key actions are similar to user-defined triggers. To ensure proper execution, ordered referential actions are sometimes replaced with their equivalent user-defined triggers;
- Set Default - this referential action is similar to "set null." The foreign key values in the child table are set to the default column value when the referenced row in the parent table is deleted or updated;
- Restrict - this is the normal referential action associated with a foreign key. A value in the parent table cannot be deleted or updated as long as it is referred to by a foreign key in another table;
- No Action - this referential action is similar in function to the "restrict" action except that a no-action check is performed only after trying to alter the table.

4. Mapping to Relational Model

4.1 One-to-One Relationships

A one-to-one relationship is characterized to have two entities that are implicitly interconnected. An instance of one class appears only one time in the other class. An example of such situation is given in the Figure 18.

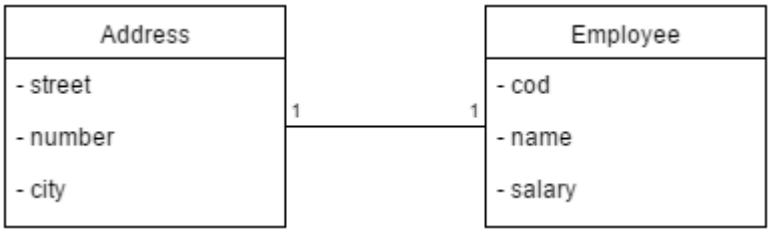

Figure 18 - Mapping one-to-one relationship

In the above situation an employee only have an associated address in the database. Reading from the left to right, we can interpret that a given address belongs only to one employee. The mapping to relational model is given below.

Employees (cod, name, salary)
Addresses (id, street, number, city, cod->Employees)

The primary key for the "Employees" table is the "cod" field. For "Addresses" table we need to establish a new field for the primary key. In the "Addresses" table we add a new foreign key that will let us to know what is the employee associated to that address. The foreign key could also be added to the "Employees" key. Typically, it is the same put the foreign key in one of the two classes, and this choice is established only by domain consistency and readability.

4.2 One-to-Many Relationships

In one-to-many relationship the two entities are connected in a way that an object of one class can appear multiple times in the other class. A one-to-many relationship is equivalent to many-to-one relationship. An example of such situation is given in the Figure 19.

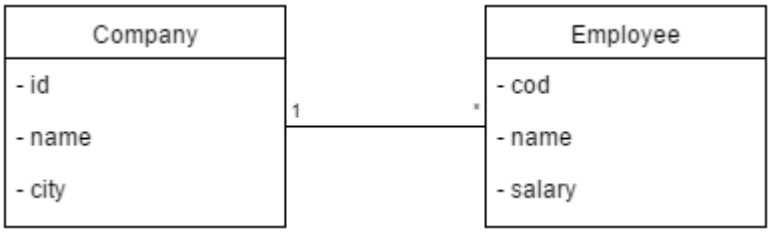

Figure 19 - Mapping one-to-many relationship

In the above situation a company can have many employees, but a given employee can only works for one company. The mapping to relational model is given below.

Companies (id, name, city)
Employees (cod, name, salary, id->Companies)

The primary key is underlined identified for both tables. In "Employees" table we need to add the foreign key of the table "Company" in order to know in which company a given employee works. The foreign key must be placed in the table that has "*" multiplicity associated. This happens always in a one-to-many relationship.

4.3 Many-to-Many Relationships

In many-to-many relationship the two entities are connected in a way that any object from any of the two classes appear many times in the other class. An example of such situation is given in the Figure 20.

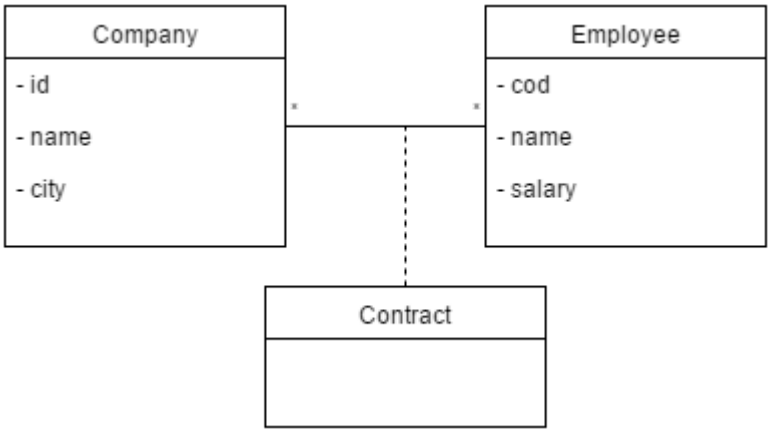

Figure 20 - Mapping many-to-many relationship

In the above situation a company can have many employees, and a given employee can only works in many companies. In a many-to-many it is mandatory the creation of a new derived class. The mapping to relational model is given below.

Companies (id, name, city)
Employees (cod, name, salary)
Contracts (id->Companies, cod->Employees)

The primary key is underlined identified for both tables. In "Contracts" table we use the foreign keys from companies' table and employees' table. The primary key for "Contracts" table is both fields.

Typically, it is useful the use of local attributes for the derived class in order to better characterize the existence relationship between the two principal classes. In our example we could add two new local fields that will give us more information about the contracts established between companies and employees. The Figure 21 depicts this situation.

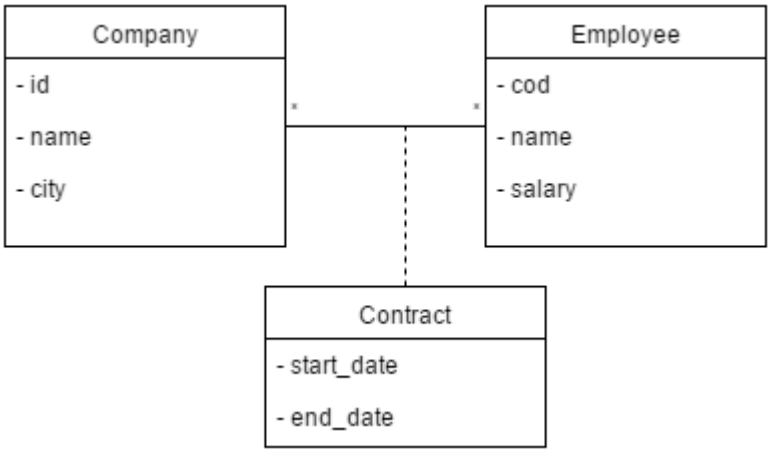

Figure 21 - Mapping many-to-many relationship with local attributes

We included two fields in the "Contract" class that will let us know where a given employee started and ended to work in each company. The mapping to relational model is given below.

Companies (id, name, city)
Employees (cod, name, salary)
Contracts (id->Companies, cod->Employees, start_date, end_date)

The relational model in this situation is very similar to previous example. In fact, we only added the two local attributes of the "Contract" class to the "Contracts" table.

However, the mapping process to relational model can become more complex and these simple rules don't work in all situations. Consider a slight different scenario where we have an employee that worked for the same company two times. For example, he left the company in 2006, but there years later in 2009 we was contracted again for the same company. In our previous example of relational model we couldn't represent this situation, because the primary key should be unique. A good alternative to avoid this situation is the creation of a new field to use as primary key. The new mapping to relational model is given below.

Companies (id, name, city)
Employees (cod, name, salary)
Contracts (idc, id->Companies, cod->Employees, start_date, end_date)

4.4 Generalizations

A generalization is the relationship between the base class that is named as “superclass” or “parent” and the specific class that is named as “subclass” or “child”. An example of such situation is given in the Figure 22.

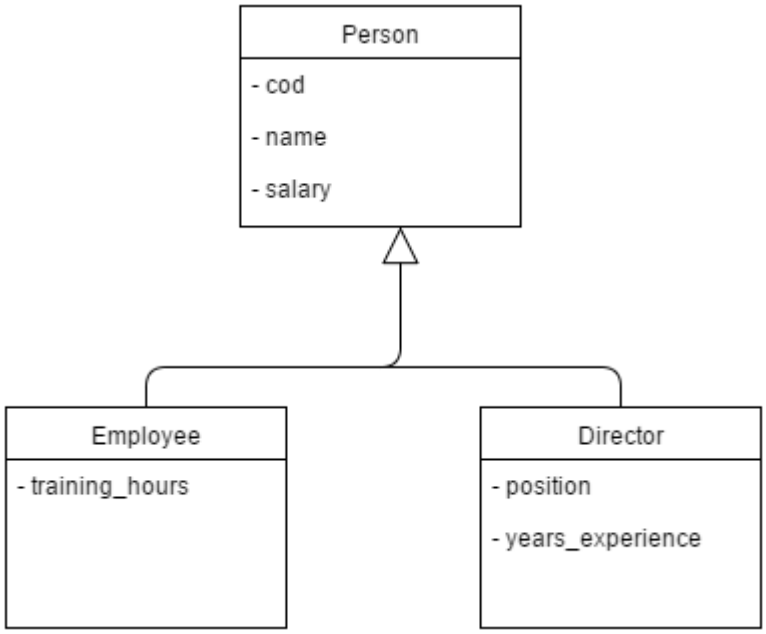

Figure 22 - Mapping a generalization relationship

In the above situation a person is a superclass entities that offers global attributes that are common to both subclasses. However, the "Employee" and "Director" classes have local attributes that are not visible to the "Person" superclass. There are three alternatives for mapping a generalization in the relational model. The choice depends on the specific circumstances of each situation.

The first approach is given below.

Persons (cod, name, salary)
Employees (cod->Persons, training_hours)
Directors (cod->Persons, position, years_experience)

The primary key for "Persons" table is the internal code ("cod") used to identify each person. The "Employees" and "Directors" table have as foreign key the "cod" of the person. This foreign key is used also as primary key for both subclasses table. This approach is the most common mapping strategy adopted in the majority of models.

The second approach is given below.

Employees (cod, name, salary, training_hours)
Directors (cod, name, salary, position, years_experience)

In the second approach there are only two tables. The attributes of the superclass are mapped in the subclasses. This strategy is good when there are many subclasses and the superclass only has a small number of attributes.

The third approach is given below.

Persons (cod, name, salary, type, training_hours, position, years_experience)

In the third approach we only have one table. We add a new field called "type" to distinguish if the person is an employee or director. This strategy originated that some tuples will have null values. It can be a good strategy when there are a small number of subclasses with few attributes.

5. Examples

In order to understand the practical implication of modeling databases using UML and adopting the relational model we consider the use of five different scenarios. We start by describing each scenario, followed by the presentation of UML class diagram and corresponding relational model.

5.1 Financial Database

A government agency intends to create a small database to register electronic invoices. The database may contain information about consumers and traders. Each consumer and trader have associated a tax card, that is unique for each citizen. For each consumer we know its full-name and each trader has information regarding its company name and city. Besides that, we know that for each invoice we have information about the type of operation, taxes and associated consumer and trader. It is also important to record for each invoice its value before and after the application of government taxes.

5.1.1 UML Class Diagram

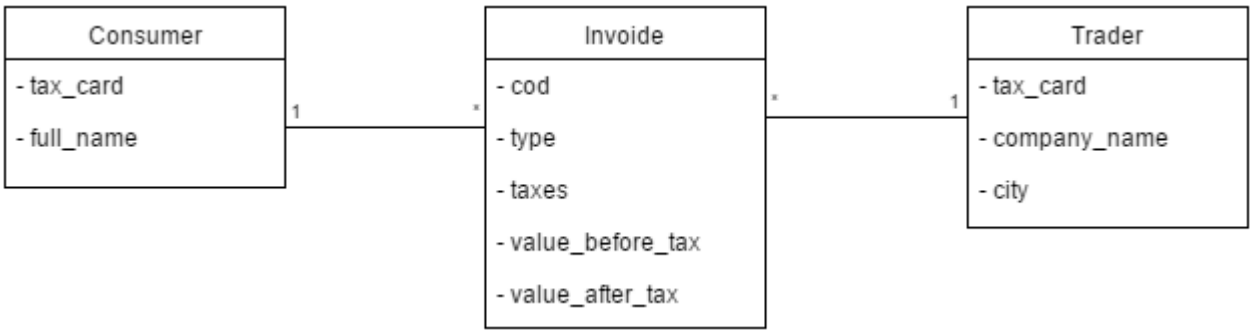

Figure 23 - UML Class Diagram - Financial Database

The model depicted in Figure 23 is composed by three entities (consumer, invoice and trader). It is a good rule to write the name of classes in singular and the name of tables in plural. For each invoice we have always associated just one consumer and trader. The class invoice offers two attributes to register the value of the invoice before and after the application of government taxes (as requested). The multiplicity of the relationship between Consumer and Invoice is 1..*, meaning that each consumer can have several invoices, but each invoice only has an associated consumer. Looking on the right side of the relationship between Invoice and Trader we have another 1..* multiplicity, meaning that a trader can have multiple invoices, but each invoice only has associated one trader. Other thing that we should notice, is that we don't have any date associated to invoice. Of course, it makes sense to have a data associated to the invoice, but we don't represent it because the enunciation doesn't explicitly refers it.

5.1.2 Relational Model

Consumers (<u>tax_card</u>, full_name)
Traders (<u>tax_card</u>, company_name, city)
Invoices (<u>cod</u>, type, taxes, value_before_tax, value_after_tax, tax_card->Consumers, tax_card->Traders)

Each table must have a primary key that is represented underlined. The foreign keys are represented with the symbol "->", where on the left side we write the name of the field, and on the right side we specify the name of the table.

The primary key for "Consumers" and "Traders" is the same. The field "tax_card" looks a good option for primary key because it unequivocally represents each entity. For "Invoices" table we need to guarantee that the "cod" field is unique for each invoice. The "invoices" must also have the information regarding the involved consumer and trader in the commercial transaction. For that, each invoice will have associated the tax_card number of the consumer and trader.

5.2 Business Database

A R&D center intends to creates a database about the location of Portuguese companies. The database should contain information about:

- Companies - each company is characterized by its name, website, foundation year and activity sector. Each company only has an activity sector, but an activity sector can be shared by several companies. It should be possible to calculate the number of activity years based in the foundation year for each company;
- Delegations - it is known that each company can have several delegations. For each delegation we intend to register information regarding the number of employees, incomes, expenses and EBITDA (Earnings Before Interest, Taxes, Depreciation and Amortization). It should possible to know where is located the company headquarters;
- Countries - each delegation is located in a given country. For each country we should register its name and an indication if the country is located inside the European Union (EU).

5.2.1 UML Class Diagram

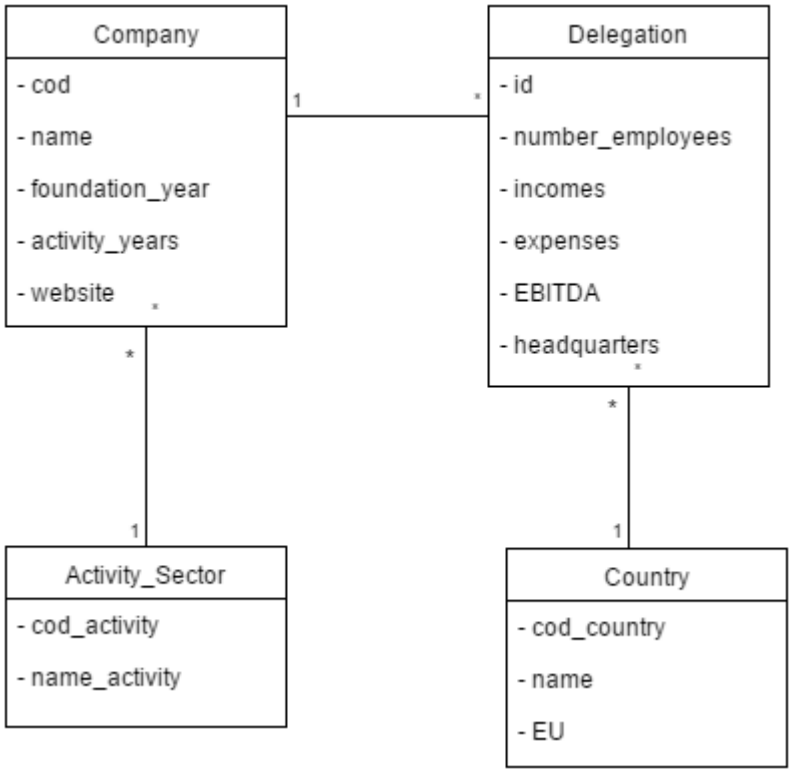

Figure 24 - UML Class Diagram - Business Database

The model depicted in Figure 24 is composed by four entities (company, delegation, activity_sector and country). For each company we register simultaneously the foundation year and the number of activity

years. In fact, the existence of these two fields is redundant, considering that it is possible to calculate in real time the number of activity years based in the current year and the foundation year of a given company. However, and in order to save this information in the database, we included also the "activity_years" field directly in the database. The delegation has an attribute called "headquarters" that indicates if a given delegation is the headquarters of the company. The multiplicity between "Company" and "Delegation" is 1..*, meaning that a company can have several delegations, but a given delegation only belongs to a company. The company can have only one activity sector and a delegation is positioned in only one country. The "EU" field in the class "Country" indicates if active (true) that the country belongs to European Union.

5.2.2 Relational Model

Companies (cod, name, foundation_year, activitiy_years, website, cod_activity->Activities_Sector)
Delegations (id, number_employees, incomes, expenses, EBITDA, headquarters, cod->Company, cod_country->Country)
Activities_Sector (cod_activity, name_activity)
Countries (cod_country, name, EU)

Each table has a primary key that corresponds to an internal code. The foreign key of table "Companies" is the code of activity, considering that a company can only have one sector of activity. In "Delegations" table, we have two foreign keys: the cod of the company and the code of the country. The other two tables ("Activities_Sector", "Countries") don't have any foreign key. It is important to highlight that the existence of a primary key in a table is mandatory, but the existence of foreign keys in a table is optional.

5.3 Store Database

A store intends to create a database to manage the items sold along the year. Each sell has an associated date, hour, quantity (if applied) and the final price. Each sell is composed by several products. For each product we have an internal code (which is unique), the name, the brand (if applied), the available stock, and the unit price. It is important to register information regarding the worker that performed a sell and the client that receives the products. For the worker we only know the name, and for the client, we know the name, tax_card and email (if available).

5.3.1 UML Class Diagram

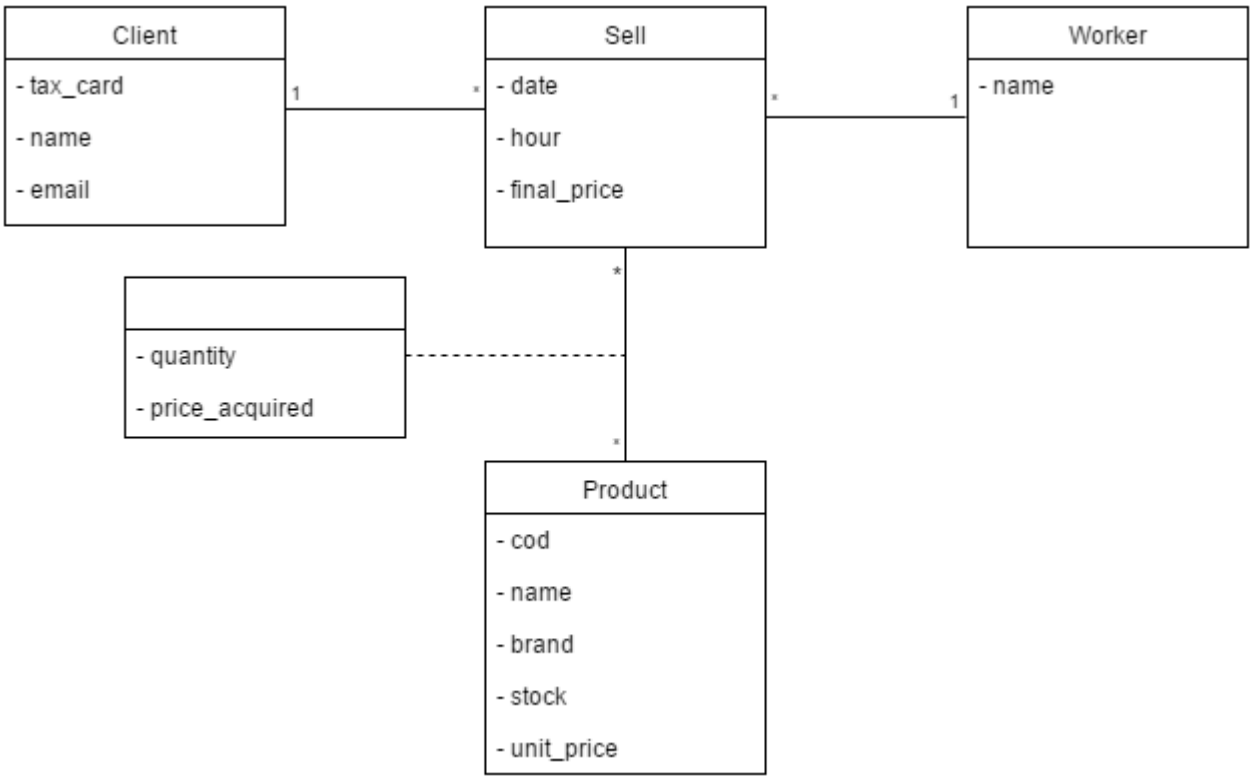

Figure 25 - UML Class Diagram - Store Database

The model depicted in Figure 25 is composed by four entities (client, sell, worker, and product). There is also a derived class that appears from a many-to-many relationship between the "Product" and "Sell" classes. It is mandatory the existence of a third class when there is in the system a many-to-many relationship. In this case, this derived class is used to record information about the quantity of each product in a given sell. It is also important to distinguish the fields "final_price" and "unit_price". The former is used to calculate the final price of a given sell considered all the acquired items; the later is used to store the unit price of each product. If we look carefully, we can realize that the "name" field is used in three different classes (Client, Worker, and Product) but with different goals. In "Client" class it is used to store the name of the client, in "Worker" class it contains information regarding the name of the worker, and in "Product" class it represents the name of the product. Finally, for each sell we only have one associated client and worker. For example, it is not possible to have a sell associated with two or more clients.

5.3.2 Relational Model

Clients (tax_card, name, email)
Workers (codWorker, name)
Sells (codSell, date, hour, final_price, tax_card->Clients, codWorker->Workers)
Products (cod, name, brand, stock, unit_price)
Items (codSell->Sells, cod->Products, quantity, price_acquired)

Each table must have a primary key. Looking to the corresponding classes, there are two tables ("Sell" and "Worker" that don't have obvious primary keys. Therefore we need to create a primary key inside those two tables. In "Sells" table we have two foreign keys that tells us which is the client and worker involved

in a given sell. In fact, we have a 1..* relationship between "Clients" and "Sells", and a similar relationship between "Workers" and "Sells". Finally there is a *..* relationship between "Sells" and "Products". It means that a sell may be composed by several products, and a given products can be acquired in many sells. Therefore, we created a derived table called "Items" that represents this situation. The "Items" table must have a primary key like the others table. The primary key is composed by the two foreign keys involved in the relationship ("Sells" and "Products"). Finally, "quantity" is a normal field that has the quantity acquired for each item. That's look an example (Figure 26) of the content of the table "Items" to explain what kind of information this table can have.

codSell	cod	quantity	isValid(?)
1	1	1	YES
2	1	4	YES
2	2	1	YES
2	5	2	YES
3	1	1	YES
3	1	2	NO
3	2	1	YES
4	8	2	YES

Figure 26 - Vision of the "Items" table

All the lines presented in Figure 26 are valid except the sixth line that is invalid. This line is invalid because the primary key must be unique, and we already have in the table a record with "codSell" equal to 3 and "cod" equal to 1. All the other lines are valid, because we can have sells composed by just one item or multiple items.

5.4 Sports Database

A national league of handball wants to register information regarding a national competition of handball that occurs once a year. It is not important to register information regarding past years, but only current year. The league is composed by 16 teams divided per two divisions. About each team, we know only its name. A handball team has between 12 and 15 players, but only 7 players can play at the same time in the field. It is important to know which were the five started players on a given team. About each player we know the name, number in the shirt, position (GR, PD, PE, LD, LE, C, P), country and value. There are always involved two teams in a given match, where is important to save information regarding the top scorer and assistance of the game. Finally, it is important to register the information about the team of the year. The team for each year is always composed by 7 players, one for each position.

5.4.1 UML Class Diagram

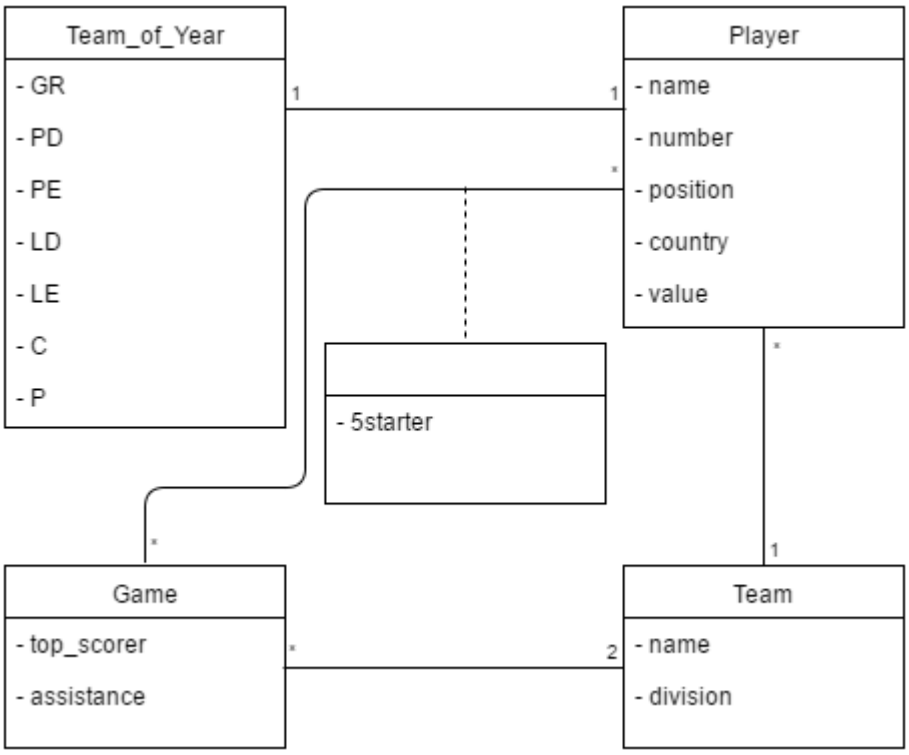

Figure 27 - UML Class Diagram - Sports Database

The model depicted in Figure 27 is composed by four entities (team, player, game, and team of the year). There is also a derived entity that appears from a many-to-many relationship between game and player, because in there are involved several players in a game and a player can play along the competition in several games. The information about the division is placed inside the "Team" class. Another option, and perhaps more versatile, is the inclusion of a new class with the divisions. In the "Game" class the "top_scorer" field has information about the "id" of the player. However, we don't know how many goals scored the top scorer, but we only have information about the player. The "5starter" field will be a boolean attribute that will be true if a given player belongs to the initial team. Finally, the team of the year has 8 field. The first 7 fields will contain the "id" of the player for each position. Finally, the last field records the year associated to the team of the year.

5.4.2 Relational Model

Teams (cod, name, division)
Players (id, name, number, position, country, value, cod->Teams)
Games (codGame, top_scorer, assistance, team1->Teams, team2->Teams)
Teams_of_Years (idTY, GR->Players, PD->Players, PE->Players, LD->Players, LE->Players, C->Players, P->Players)
Enrolled (id->Players, codGame->Games, 5starter)

The analysis of this relational model is more complex than in previous examples. We have the following relevant situations:

- Teams - this is the easiest table. We need only care about the choice of the primary key to guarantee that each team is unique in the database;

- Players - the "number" field can't be the primary key of the table, because a given number could be used by several players from different teams. Therefore, we need to create a new field for the primary key ("id"). Additionally, we need to create a foreign key to associated players to the teams;
- Games - as in previous situation the best solution is to create a new field for the primary key. Besides that, we need to include two foreign keys: "team1" is the team that plays at home; "team2" is the team that plays as away;
- Teams_of_Years - a good choice for primary key is the "year" field, because we can guarantee that there is only one team of the year per each year. Therefore, it won't be possible to have in this table two entries with the same year. In each position (e.g., GR, PD, PE, etc) we have the "id" of the player;
- Enrolled - it is used to know the players that have played in each game. The "5starter" field, as explained before, is just a boolean field.

In Figure 28 we present an example of possible contents for the "Enrolled" table.

id	codGame	5starter	isValid(?)
1	1	True	YES
2	1	True	YES
3	1	False	YES
7	1	True	YES
1	2	True	YES
1	2	False	NO
2	2	True	YES
3	2	True	YES
4	2	False	YES
5	2	True	YES

Figure 28 - Vision of "Enrolled" table

The sixth row is not valid, because the previous line has already a tuple with the same "id" and "codGame". Additionally, it doesn't make sense to say that a given player was in 5starter of a game, and say the opposite in the following line.

5.5 Health Database

A health clinic intends to create a new database to register information regarding her activity. The most important event for the clinic is the doctor's appointments. For each appointment the clinic gets information about the date and involved patients and doctors. Each appointment is characterized to have involved just one patient and doctor. The information that we know about them is the following:

- Patient - social security number, name and birth date;
- Doctor - name and medical specialty.

For each appointment is important to have information about the diagnosis, the severity of the symptoms and the date where those symptoms started to appear. An appoint can give origin to a prescription, which can be composed by several medicines. Each medicine has information about the designation/name, laboratory and price. Finally, it is important to know the pharmacy (name, address, and technical director) where patients use their prescription. A given prescription can only be used by one pharmacy, and cannot be partially fulfilled.

5.5.1 UML Class Diagram

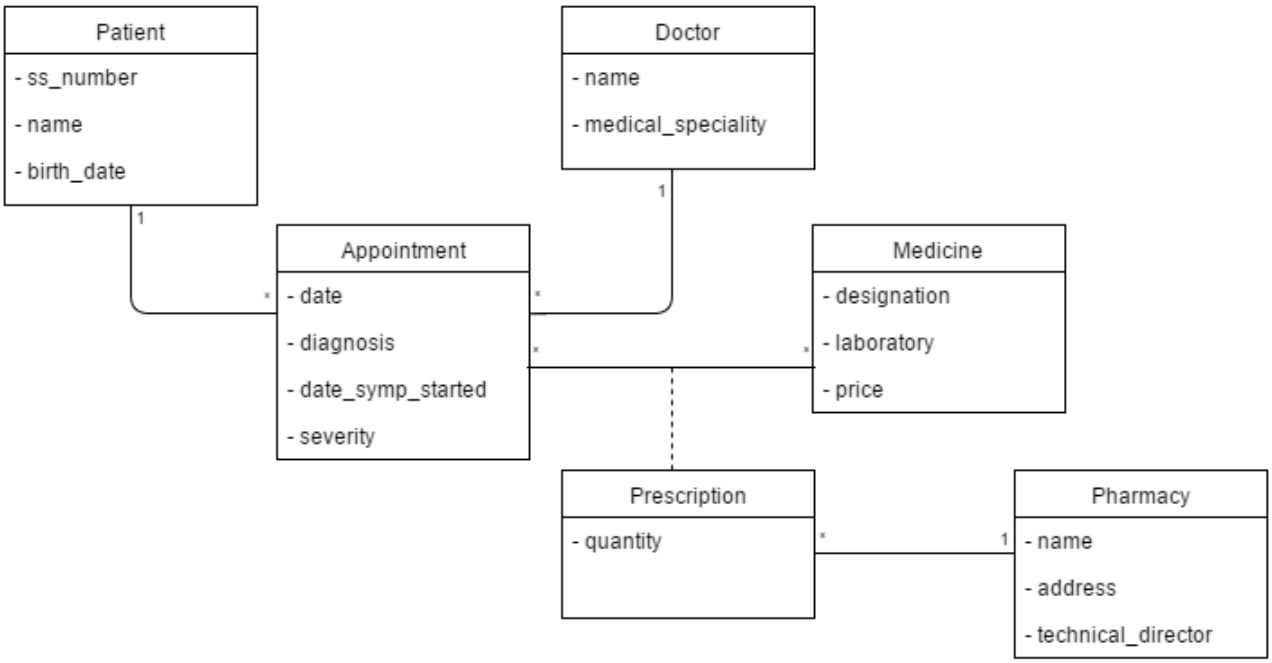

Figure 29 - UML Class Diagram - Health Database

The model depicted in Figure 29 is composed by six entities (patient, doctor, appointment, medicine, prescription, and pharmacy). There "Prescription" class is a derived class that appears from the many-to-many relationship between appointments and medicines. The appointment is characterized to have two dates but with different goals: "date" field has the date where occurred the appointment; "date_symp_started" has indication about the date where a given symptom started to appear. This second field can be null, but the date of appointment is mandatory. A given appointment involves only one patient and one doctor, as demonstrated by the multiplicity 1..*. The prescription has always an associated appointment and medicines. Finally, the same pharmacy can be used to fulfill several prescriptions.

5.5.2 Relational Model

Patients (ss_number, name, birth_date)
Doctors (id, name, medical_speciality)
Appointments (cid, ss_number->Patients, id->Doctors, date, diagnosis, date_symp_started, severity)
Medicines (ref, designation, laboratory, price)
Prescriptions (cid->Appointments, ref->Medicines, quantity)
Pharmacies (cod, name, address, technical_director)

In relational model we have the following relevant situations:

- Patients - the social security number is the primary key;
- Doctors - we create a new "id" field to be the primary key;
- Appointments - the best alternative is the creation of a new "cid" field to work as primary key. Another alternative is the use of three combined fields (ss_number, id, date) as primary key. However, this alternative is more complex, particularly because we need to use the foreign key of appointment in other tables;
- Medicines - a good option is the creation of a new field "ref" for the primary key;

- Prescriptions - we use the foreign keys from "Appointments" and "Medicines" tables. The primary key is these two fields together;
- Pharmacies - we add a new field "cod" to work as primary key.

5.6 University Database

A university intends to create a new database to register information about the academic path of students. Each course has several classes, which can be shared among different courses. A given class can be placed in different semesters according each course. The course is characterized to have a name, coordinator and department. The department has always information about is location and phone number. Each class has information about its name, ECTS and bibliography. The same class can be given by several instructors. Students may attend at least one class and we have information about the students' name, address, city and email. It is also important to know the date where a student enrolled in a given class. Finally, a class has no more than 5 assignments. For each assignment we know the date, if it is an exam or project, and the percentage of the grade in the final mark. It is also important to register the academic score obtained by each student in the assignment.

5.6.1 UML Class Diagram

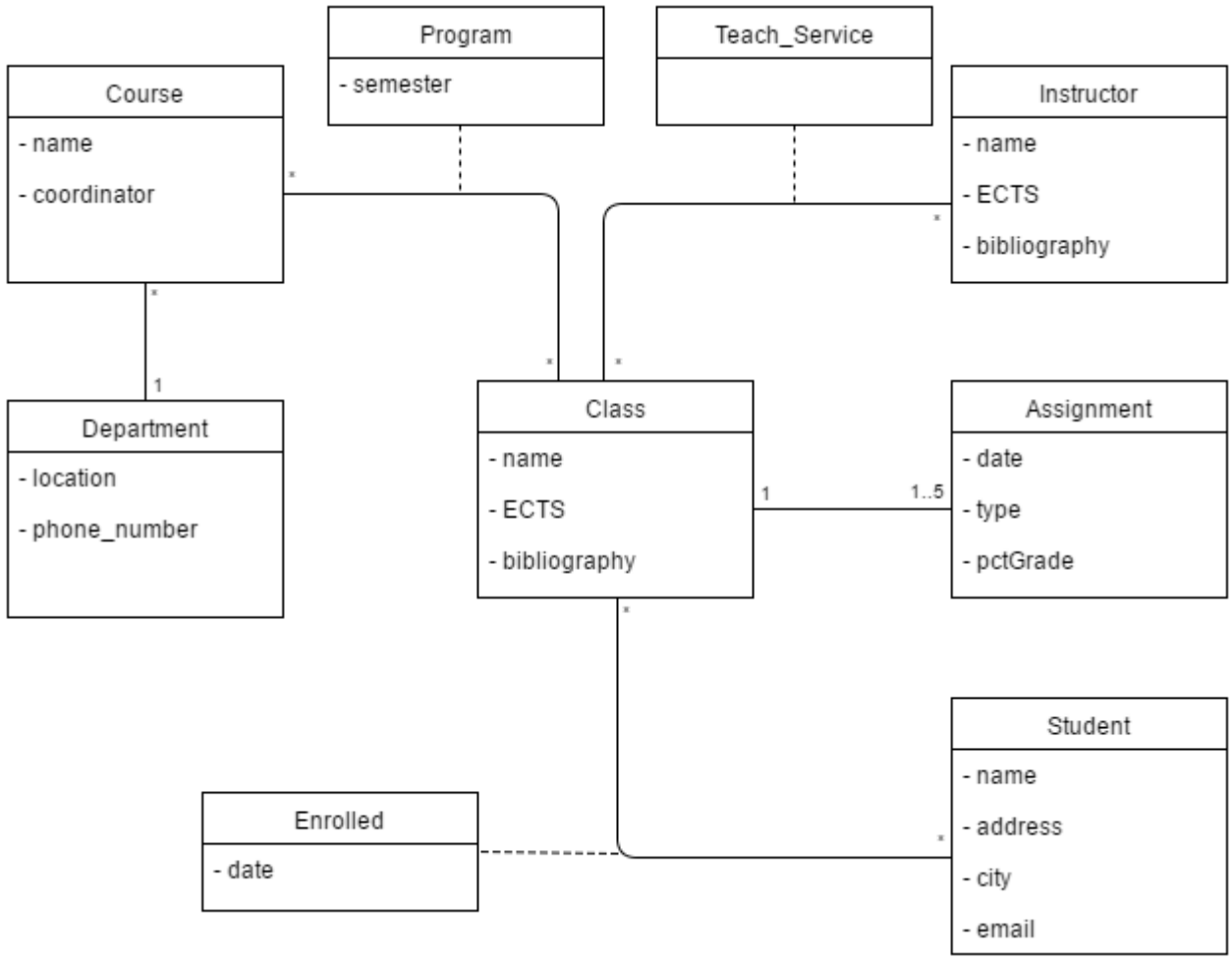

Figure 30 - UML Class Diagram - University Database

The model depicted in Figure 30 is composed by six entities (course, department, instructor, class, assignment, and student). There are also three derived entities (program, teaching service and enrolled).

These derived classes appeared from three many-to-many relationships. The information regarding the department of each course is registered as a separated class, because for each department we know its location and phone number. The existence of separated classes adds to our model more flexibility and easy to perform changes if needed. A class can have until 5 assignments, but it is not possible to have a class without at least one assignment. This is illustrated by the "1..5" relationship. However, a given assignment corresponds only to one class. Finally students can attend several classes and, therefore, the enrolled class is used.

5.6.2 Relational Model

Departments (cod, location, phone_number)
Courses (cid, name, coordinator, cod->Departments)
Classes (ref, name, ECTS, bibliography)
Programmes (cid->Courses, ref->Classes, semester)
Instructors (id, name, ECTS, bibliography)
Teach_Services (ref->Classes, id->Instructors)
Assignments (idA, date, type, pctGrade, ref->Classes)
Students (codStd, name, address, city, email)
Enrolled (ref->Classes, codStd->Students, date)

In relational model we have the following relevant situations:

- Departments - we create a new field "cod" that is unique for the primary key;
- Courses - like in table above we need to create a new field for the primary key. Additionally, we need to include the foreign key for the department;
- Classes - the primary key is also a new created field. There aren't foreign keys in this table;
- Programmes - the primary keys are also foreign keys. The foreign key is the primary key of "Courses" and "Classes" tables;
- Instructors - we need to define the primary key;
- Teach_Services - we need to use the foreign keys of the involved tables in the many-to-many relationship;
- Assignments - we add a new field for the primary key. We need to include the foreign key of classes to know the associated class of each assignment;
- Students - the primary key need to be created. A good option is to use an internal unique number that characterizes each student along his/her academic path at university;
- Enrolled - needs to use as foreign key the primary keys of "Classes" and "Students" table. The proposed relationship establishes that a student only can be enrolled in the same class just one time. If we want to guarantee that a student may be enrolled in the same class more than one time, we need to include the "date" field in the primary key too.

5.7 Library Database

A library intends to create a database to store information about their books and loan process. Each book has information about ISBN, title, publisher, authors and number of pages. We only know the name of the authors. A publisher has information about its name and city, and can be responsible for the edition of several books. It is important to allow searches by the subjects of a given book. The same book can have multiple copies, which can be loaned. It is important to store information regarding the acquired date of each book' copy. For the loans it is important to know the due date. Finally, for the users we must know the name, address, phone and city.

5.7.1 UML Class Diagram

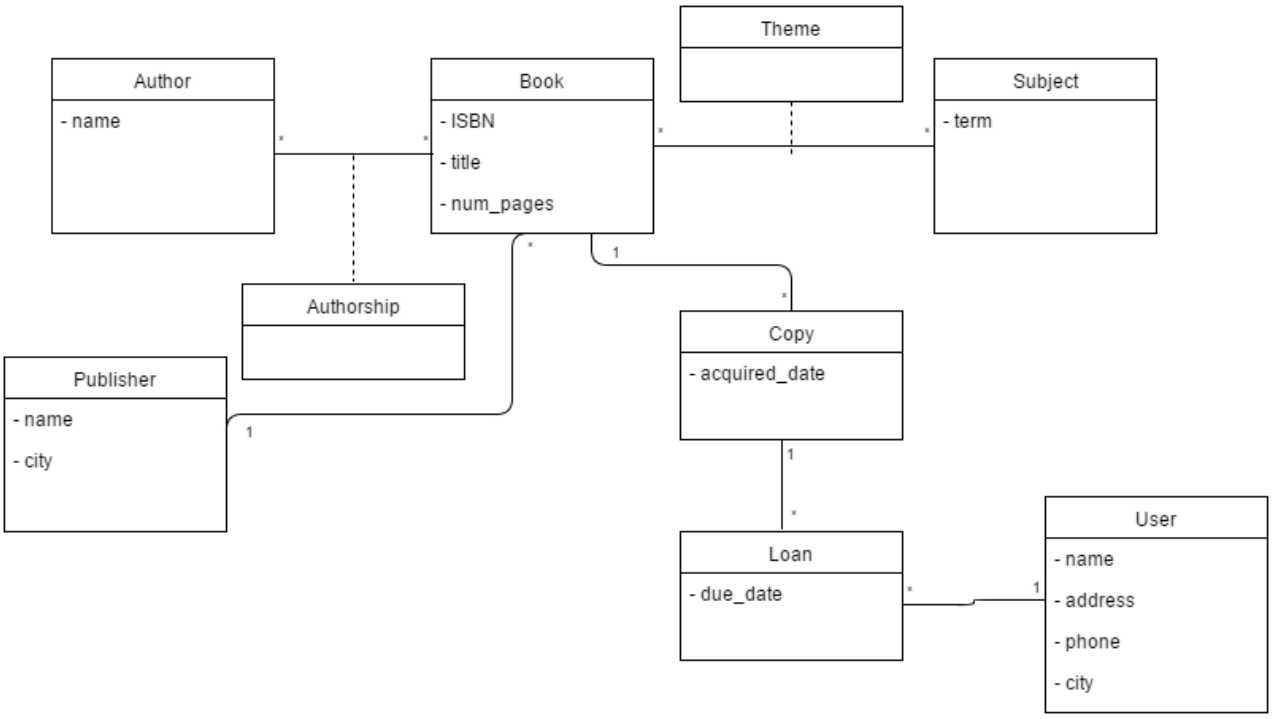

Figure 31 - UML Class Diagram - Library Database

The model depicted in Figure 31 is composed by seven entities (author, publisher, book, copy, subject, loan, and user). There are also two derived classes (authorship, and theme) that results from many-to-many relationships. A book has one or more copies, but a copy can only belong to a given book. The loans are performed in relation to copies and not books. However, the relation between authors and books is not performed with a copy, because all the copies share the same information about authors. The same situation applies to publishers. Finally we only have one user responsible for the loans of a book.

5.7.2 Relational Model

Authors (codA, name)
Books (ISBN, title, num_pages, codP->Publishers)
Subjects (id, term)
Themes (ISBN->Books, id->Subjects)
Publishers (codP, name, city)
Authorships (codA->Authors, ISBN->Books)
Copies (codC, acquired_date, ISBN->Books)
Loans (idLoan, due_date, codC->Copies, idUser->Users)
Users (idUser, name, address, phone, city)

In relational model we have the following relevant situations:

- Authors - the primary key is an internal code that is unique for each author;
- Books - the primary key is the ISBN of book (unique reference). For each book we need to know what is the publisher. For that, we use the foreign key of the publisher;
- Subjects - we create an internal id associated to each term;
- Themes - uses the foreign keys of the "Books" and "Subjects" tables. The primary key is these two fields together;

- Publishers - each publisher has an unique code;
- Authorships - the same situation like happened to "Themes" table;
- Copies - each copy has an unique identifier. We also need to know what is the book associated to each copy. For that we use the foreign key of the "Books" table;
- Loans - each loan has an unique identifier to work as primary key. We also need to include two foreign keys: one to connect to "Copies" table; the other to connect to "Users" table;
- Users - a new id has been added for the primary key.

5.8 Employees Database

A company wants to create a database to register several occurrences about their employees. An employee has information about her/his name, gender, birth date and hire date. An employee can work in several departments during a period of time. The employee can also have the role of manager in some situations. It is important to know the salary of each employee during a certain period of time, but this salary is not totally dependent in which department he/she is working. An employee can have several titles in the company. An employee participates in several events, and it is important to know who authorized the attendance of the employee in the event. An event has always associated a short description, the date when started and ended. The company has two kind of events: training (has an associated mark) and vacation (has an associated destination). A training is composed always by one course that has associated a given code and title. Finally, there are keywords associated to each course.

5.8.1 UML Class Diagram

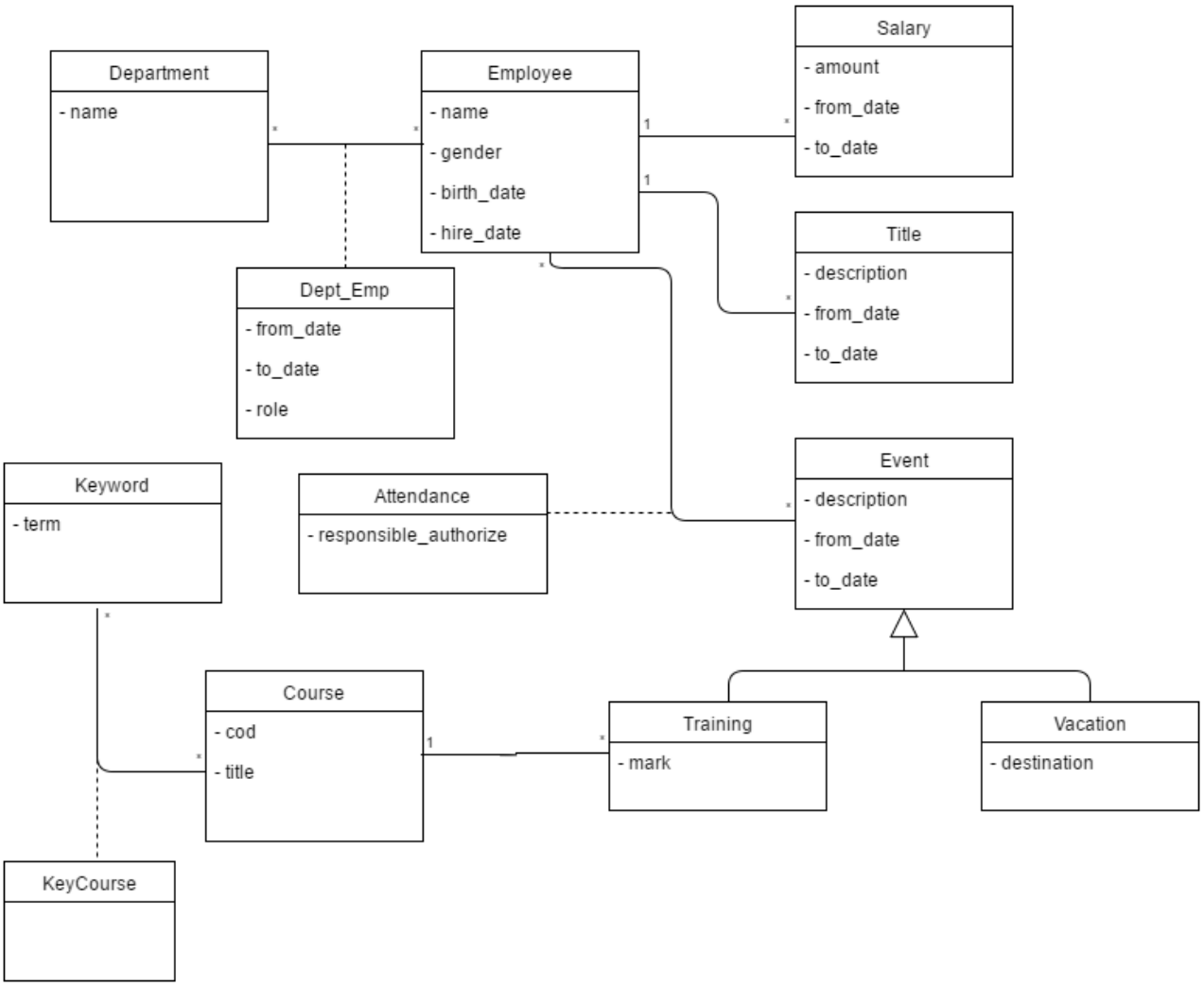

Figure 32 - UML Class Diagram - Employees Database

The model depicted in Figure 32 is composed by nine entities (department, employee, salary, title, event, training, vacation, course, and keyword). There are also two derived classes that appear from many-to-many relationships. There are some relevant situations to be highlighted:

- We don't place the salary and title as attributes of the "Dept_Emp" class because the date associated to these elements couldn't be the same as in departments. Therefore, the best option is to create three independent classes that offer us more flexibility, even if "from_date" and "to_date" is repeated in all these classes;
- The person responsible for the authorization of participation in an event is stored in the "Attendance" class, because this person can be different for each employee that participates in the same event;
- The "Training" and "Vacation" are subclasses of "Event". Therefore, these two classes share the attributes of "Event" class;
- Information about courses is stored in an independent class ("Course"), because the same course can have several editions.

5.8.2 Relational Model

Departments (idD, name)
Employees (codE, name, gender, birth_date, hire_date)
Dept_Emp (codDPE, idD->Departments, codE->Employees, from_date, to_date, role)
Salaries (idS, amount, from_date, to_date, codE->Employees)
Titles (idT, description, from_date, to_date, codE->Employees)
Events (idE, description, from_date, to_date)
Attendances (idA, codE->Employees, idE->Events, responsible_authorize->Employees)
Trainings (idE->Events, mark, cod->Courses)
Vacations (idE->Events, destination)
Courses (cod, title)
Keywords (idK, term)
KeyCourses (idK->Keywords, cod->Courses)

In relational model we have the following relevant situations:

- Departments - we create an id for the primary key;
- Employees - we only need to create an unique code for the primary key;
- Dept_Emp - we add a new field for primary key, because the employee can work in the same department more than one time along his/her career. The foreign keys appear from the many-to-many relationship;
- Salaries - we add a new field ("idS") for the primary key. We have a foreign key to know which employee is associated to the salary;
- Titles - we add a new field ("idT") for the primary key. As in above table, we have a foreign key associated to the employee;
- Events - we add a new filed ("idE") for the primary key;
- Attendances - this is the most complex table. The primary key is a new field ("idA"), because the same employee can participate in the same event multiple times. We have three foreign keys: the first one associated to the employee, the second associated to the event, and the third associated to the employee that was responsible for the authorization. Notice that the "responsible_authorize" field is connected to the "codE" field in the "Employees" table;
- Trainings - the primary key is the id of the event. It has also a foreign key to know the associated course;
- Vacations - it uses also the id of the event for the primary key;
- Courses - the primary key is the "cod" field;
- Keyworks - we create a "idK" field for the primary key;
- KeyCourses - it uses as foreign keys the primary keys of the "Keywords" and "Courses" tables. The primary key is these two fields together.

We will look in more detail to the contents of "Attendances" table. In Figure 33 we present an example of possible contents.

idA	codE	idE	Responsible_authorize	isValid(?)
1	1	1	3	YES
2	2	1	2	YES
3	2	2	3	YES
4	3	3	5	YES
5	7	3	1	YES

6	2	4	1	YES
7	1	5	5	YES
8	1	1	5	YES
9	7	6	5	YES
8	5	7	9	NO

Figure 33 - Vision of "Attendances" table

Looking to the above table we must clarify the following situations:

- Typically makes sense that the responsible to authorize the presence of the employee in a given event is not the same employee that submitted the request. However, this situation is not mandatory and not specified in the relational model. For example, in line 2 we have a situation where the "codE" is 2 and the "responsible_authorize" is also 2;
- The employee with "codE" equal to 1 participated in the events with code equal to: 1; 5; and 1. In fact, the given employee participated two times in the same event. This situation is also possible and valid;
- The last line is not valid, because the primary key ("idA") cannot be duplicated.

Bibliography

Coronel, C., & Morris, S. (2016). *Database Systems: Design, Implementation, & Management.* Course Technology.

LucidChart. (2015). *What is UML?* Retrieved 07 08, 2016, from https://www.lucidchart.com/pages/what-is-UML-unified-modeling-language

Perera, R. (2011). *Understanding UML Class Diagram Relationships*. Retrieved 07 08, 2016, from Creately: http://creately.com/blog/diagrams/understanding-the-relationships-between-classes/

Rouse, M. (n.d.). *Database Management System (DBMS)*. Retrieved 07 05, 2016, from SearchSQLServer: http://searchsqlserver.techtarget.com/definition/database-management-system

Sparx Systems. (n.d.). *UML 2 Class Diagram*. Retrieved 07 11, 2016, from http://www.sparxsystems.com.au/resources/uml2_tutorial/uml2_classdiagram.html

Tao, Y. (n.d.). *Relational Model: Tables and Keys*. Retrieved 07 17, 2016, from http://www.cse.cuhk.edu.hk/~taoyf/course/bmeg3120/notes/rel-model.pdf

TechNet. (n.d.). *PRIMARY KEY Constraints*. Retrieved 07 19, 2016, from https://technet.microsoft.com/en-us/library/ms191236(v=sql.105).aspx

Techopedia. (n.d.). *Foreign Key*. Retrieved 07 19, 2016, from https://www.techopedia.com/definition/7272/foreign-key

Tutorials Point. (n.d.). *Codd's 12 Rules*. Retrieved 07 11, 2016, from http://www.tutorialspoint.com/dbms/dbms_codds_rules.htm

UHI Millennium Institute. (n.d.). *The Database Terms of Reference*. Retrieved 07 23, 2016, from http://rdbms.opengrass.net/2_Database%20Design/2.1_TermsOfReference/r/2.2_Keys.pdf

Ullman, J., & Widom, J. (2014). *First Course in Database Systems.* Pearson.

Visual Paradigm. (n.d.). *Class diagram*. Retrieved 07 08, 2016, from https://www.visual-paradigm.com/VPGallery/diagrams/Class.html