POLITECNICO DI TORINO

MASTER'S DEGREE THESIS
COMPUTER ENGINEERING - SOFTWARE

A WEB-BASED APPLICATION FOR COMPUTER-AIDED DESIGN OF RELATIONAL DATABASES

Candidate: Emanuele Marchetta

Supervisor: Silvia Chiusano
Co-supervisor: Paolo Garza

April 2021

Abstract

Databases allow applications to store and retrieve data in an efficient and reliable way, among many other things. Even though recent years have witnessed a rise in the use of NoSQL database technologies, relational databases are still relevant and very useful in several respects. In order to make effective use of database technology, one has to master the skill of schema structure design. Concepts at the heart of such design have been developed and consolidated over time, but no software packages have been deployed with the precise intent of guiding and assisting users – more or less experienced – throughout the entire process. This thesis work focuses on the design and development of a tool that serves two main purposes. On the one hand, it offers functionalities specific to the domain, from ER schema drawing, restructuring, and translation, to logical schema editing and exporting. On the other hand, it allows users to learn the theoretical concepts of database design and prevents them from making conventional mistakes. Rather than just a piece of software, we strived to deliver a great, innovative product. User experience had a significant impact on development decisions, and interface design was driven by an extreme push for simplicity and efficiency. It was deployed as a Web Application for several reasons: they are quicker and easier to build, they work in the browser (always available on all devices) and they can be updated seamlessly. Although all requirements set out initially were met, there certainly is a great deal of room for improvement. Some edge cases have not been covered and a great number of features can still be implemented. Also, a lot of emphasis was put on the educational aspect, but – with some small changes – the software has everything it takes to become a fully-fledged professional product.

Contents

Abstract

1 Introduction and Motivation **1**
1.1 Relational Databases Design Process 1
1.2 Who This Tool Is For 3
1.3 Existing Alternatives 3
1.4 Organization of This Thesis 5

2 Conceptual Design **6**
2.1 Entity-Relationship Model 7
2.1.1 Entities 7
2.1.2 Relationships 7
2.1.3 Attributes 9
2.1.4 Internal and External Identifiers 10
2.1.5 Generalizations 11

3 Logical Design **13**
3.1 ER Schema Restructuring 13
3.1.1 Generalizations 14
3.1.2 Multivalued Attributes 15
3.1.3 Composite Attributes 16
3.2 Translation into the Relational Model 17
3.2.1 Entities 17
3.2.2 Relationships 18
3.2.3 Attributes 20
3.3 Towards Physical Design 21
3.3.1 Data Types 21
3.3.2 Constraints 22
3.3.3 SQL Language 22

4 Technologies Adopted **25**
4.1 Why a Progressive Web Application . . . 25
4.1.1 Mobile Apps and Web Apps . . . 25
4.1.2 Single-Page Applications . . . 27
4.1.3 Progressive Web Applications . . . 29
4.2 Web APIs . . . 31
4.2.1 Web Storage API . . . 31
4.2.2 Canvas API . . . 33
4.2.3 Touch Events . . . 34
4.3 Vue.js JavaScript Framework . . . 35
4.3.1 Overview . . . 35
4.3.2 Components . . . 36
4.3.3 Reactivity . . . 37
4.4 Two.js Drawing API . . . 38
4.4.1 Overview . . . 38

5 Product Design and User Experience **41**
5.1 User Interface . . . 41
5.1.1 Minimal UI for Maximum Impact . . . 42
5.1.2 Tool UI Appearance . . . 43
5.1.3 Responsive Web Design . . . 44
5.1.4 Editor Navigation . . . 47
5.2 User Experience . . . 48
5.2.1 Errors and Suggestions . . . 48
5.2.2 Hide Unnecessary Information . . . 49
5.2.3 Real-Time Updates . . . 51
5.3 Schema Drawing . . . 52
5.4 Utility Features . . . 53
5.4.1 Load and Save . . . 53
5.4.2 Exporting the Diagram . . . 54
5.5 Enhancing Usability . . . 55
5.5.1 Undo and Redo . . . 55
5.5.2 Autosave . . . 57
5.5.3 Offline Availability . . . 58
5.5.4 Keyboard Shortcuts . . . 60

6 Tool Overview **61**
6.1 Drawing the ER Model . . . 61
6.1.1 Entities . . . 61
6.1.2 Relationships . . . 64

6.1.3 Attributes . . . 68
6.1.4 Identifiers . . . 69
6.1.5 Generalizations . . . 72
6.1.6 ER Code Generation . . . 74
6.2 Restructuring Step . . . 75
6.2.1 Generalizations . . . 76
6.2.2 Multivalued Attributes . . . 77
6.2.3 Composite Attributes . . . 78
6.3 Translation Step . . . 79
6.3.1 Entities . . . 79
6.3.2 Relationships . . . 80
6.4 Logical Schema Editing . . . 81
6.4.1 Assigning Data Types . . . 81
6.4.2 Reordering Columns . . . 82
6.4.3 Unique Constraint . . . 82
6.4.4 SQL Code Generation . . . 82
6.5 An Example Use Case . . . 83
6.5.1 Exercise Text . . . 84
6.5.2 ER Diagram and Logical Schema . . . 84

7 Conclusions **87**
7.1 Summing Up . . . 87
7.2 Where To Go From Here . . . 88
7.2.1 Functionality Limitations . . . 88
7.2.2 New Features . . . 89
7.2.3 General Improvements . . . 89

Chapter 1

Introduction and Motivation

Databases play a central role in modern computing systems. They not only store application data in an efficient and reliable way, but also offer advanced features like transactions and concurrency control, and face issues regarding reliability, data replication and recovery from failures. Since they represent such a critical subsystem, the importance of a good design cannot be overstated. This is especially true for relational databases, where the structure and format of the data must be determined a priori.

Modern Web Applications can be run from any device, from desktop computers, to tablets and smartphones. Thanks to the numerous advantages they offer, they are a great option when it comes to software development. This thesis work aims to design and develop an engaging product to support every step of the database design process. Even though the software is not limited to any particular use or application, our main goal was to deliver a tool capable of assisting users in their learning process.

1.1 Relational Databases Design Process

Relational databases have the characteristic of requiring the definition of a schema. In this setting, data is organized in tables, and each one of these consists of a set of columns.

The process of designing a relational database has been consolidated over the years and is composed of three consecutive phases (outlined in Figure 1.1):

- *Conceptual design.* Informal requirements described in natural language are expressed in a high-level formal graphical representation, that explicitly states which entities are involved and what are the relationships among them. A possible approach to conceptual design is through the use of the Entity-Relationship (ER) model.

- *Logical design.* A series of well-defined rules allows the translation of the conceptual representation defined in the preceding phase into a set of tables, each with their columns. The output of this step is the logical schema of the database and refers to a logical data model. This representation is still independent of the physical details, although the DBMS used for the implementation must be one that supports that data model.
- *Physical design.* DBMS-specific features are exploited in such a way that efficiency and performance are maximized. This requires knowledge about statistics on data, expected workload, and DBMS operation (i.e. indexing, storage structures, query processing, etc.).

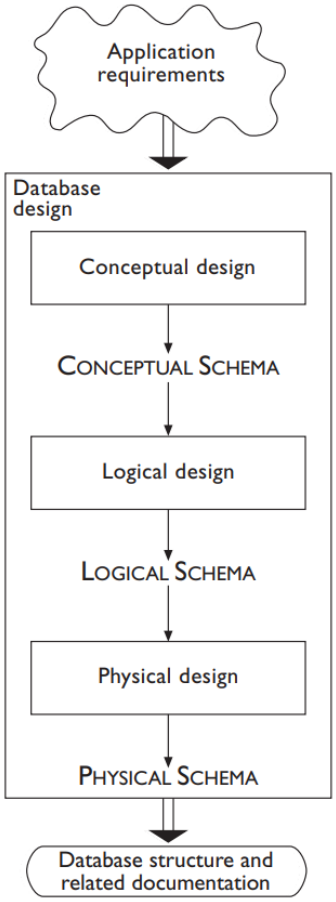

Figure 1.1: The phases of database design [1].

Clearly, physical design is strongly affected by the DBMS of choice. In order to deliver a product agnostic to the selection of any particular DBMS, this work focuses mainly on the first two steps.

1.2 Who This Tool Is For

Anyone looking for a software for conceptual and logical design will likely find this tool helpful. No particular experience or expertise is required to use the software. Every step of the process is covered: from drawing the ER model, to restructuring and translating it, and finally editing the resulting logical schema down to the last detail.

Most importantly, however, this product sets itself to provide support for students and enthusiasts who are learning how to design relational databases. Productivity and automation have been disregarded in favor of clarity and completeness. Each feature is implemented in a way that prevents users from making careless mistakes and, whenever possible, suggestions and clarifications are shown.

1.3 Existing Alternatives

For what concerns the mere purpose of drawing diagrams, a huge number of software packages are currently available. These tools provide specialized features for creating and manipulating objects like shapes, lines, and text. These are totally valid options if the intention is only to concentrate on the drawing aspect. However, while the graphical side is central to our product, our focus does not stop there.

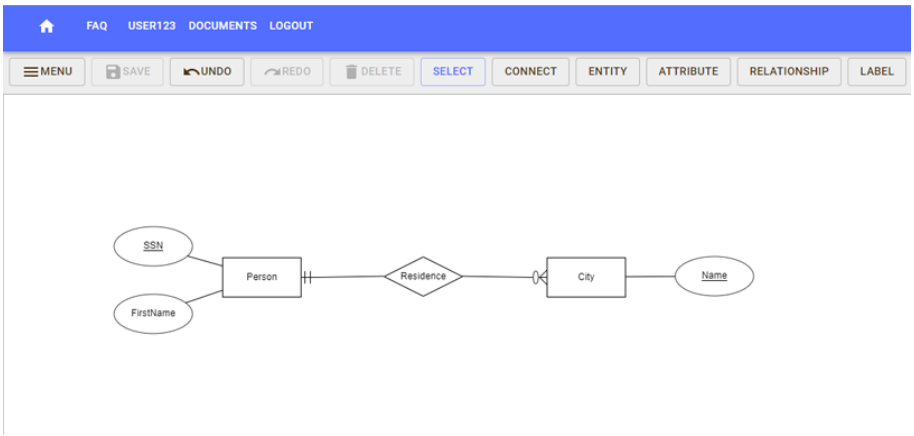

Figure 1.2: ERDPlus user interface.

On the other hand, some tools are specifically conceived for conceptual schema drawing and therefore are able to offer features best suited to the needs of database design. Below is a selection of the most promising ones, with some considerations about their selling points and the reasons why they do not meet our requirements.

ERDPlus

ERDPlus is a database modeling tool to quickly and easily create Entity-Relationship Diagrams, Relational Schemas, and Star Schemas [2]. It is free to use and quite simple to operate. The majority of ER schema constructs are supported, and automatic conversion to the relational schema is offered. Diagrams can be easily loaded from files, saved for later reuse, and exported as images. This is likely the closest thing to our idea of application for database design.

Unfortunately, some issues prevent this tool from being complete. The lack of error-checking makes it fitter for experienced users than for beginners and does not really help them learn. While being significant, the provided functionality is still limited: generalizations are not supported and model restructuring is absent.

Microsoft Visio

Microsoft Visio is a diagramming and vector graphics application and is part of the Microsoft Office family [3]. It is paid (with the possibility of a free trial) and is a complete and extended diagramming software. Among many other things, it allows creating Entity-Relationship diagrams with the possibility to specify things like relationship cardinality, identifier attributes, mandatory, multivalued, and derived attributes.

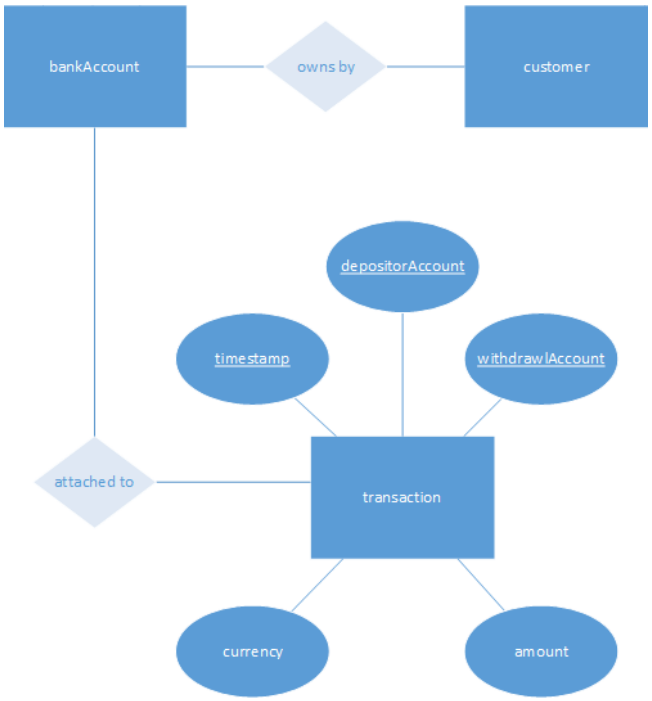

Figure 1.3: ER model designed using Microsoft Visio [3].

The software offers advanced tools like exporting to other Microsoft programs or reverse engineering to generate an ER model from an existing database schema. On the downside, it is not specific to database design and does not support both generalizations and guided model restructuring/translation. It is more appropriate for brainstorming and quick diagraming than for in-depth design of relational databases.

1.4 Organization of This Thesis

Chapter 2 discusses the basics of conceptual schema design. It focuses entirely on the Entity-Relationship (ER) model and covers its fundamental constructs, their meaning, and the rules that govern them.
Chapter 3 is a reference on conceptual schema restructuring and translation: tasks that constitute the logical design phase. It looks at how to construct a logical schema that correctly and efficiently represents all of the information described in the ER schema.
In Chapter 4, we explore the technologies used in the development and explain the rationale for the technical choices made. Some code examples are reported to showcase a particular feature's capabilities or to illustrate possible applications.
Chapter 5 digs into product design, User Interface, and User Experience. It covers best practices and guidelines of software design, and explains how we took advantage of them to make our application more usable and enjoyable.
Chapter 6 illustrates in great detail how the software works. It is a comprehensive walkthrough to the different features of the application and highlights common mistakes that are made and how to avoid them.
Finally, Chapter 7 draws the conclusions of the thesis work and provides some recommendations for future developments of the software, both in terms of functionality and technology.

Chapter 2

Conceptual Design

In general, the design of relational databases starts with the definition of a conceptual schema. Note that there is no unique formalism for that. In this work, the choice fell on the Entity-Relationship (ER) model, widely used in structured analysis and conceptual modeling. We will use the terms *conceptual schema* and *ER model* interchangeably throughout the text. In this chapter, we illustrate a structured approach to conceptual design and the data representation tools at the designer's disposal [1].

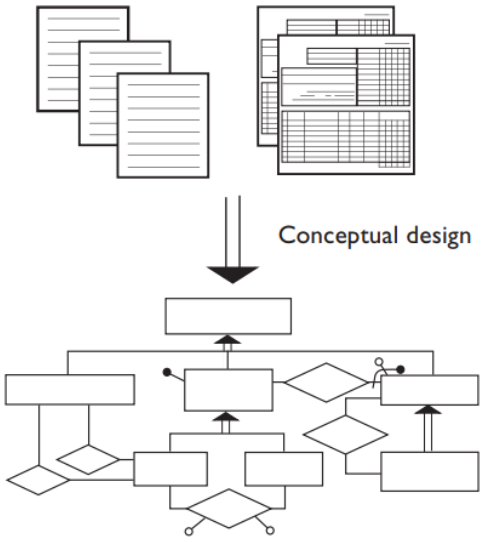

Figure 2.1: The conceptual design phase [1]

2.1 Entity-Relationship Model

The Entity-Relationship model is the most widely used representation in conceptual design. It was designed by Peter Chen and published in a paper in 1976, even though variants of the idea seemed to exist already.

The model provides a high-level view of a specific domain of knowledge by means of a set of *constructs*, namely, entities, relationships, attributes, identifiers, and generalizations. This representation is abstract and independent from any subsequent choice of implementation. The fact that constructs can be illustrated graphically allows us to define the model using a diagram.

2.1.1 Entities

An *entity* is anything that exists physically or logically, and that can be uniquely identified. It represents a class of objects that have common properties and an autonomous existence. For instance, CUSTOMER, ORDER, and INVOICE are possible examples of entities in an e-commerce domain. An occurrence of an entity can be seen as an object of the class represented by the entity itself. Customers *Stevens* and *Rodriguez* are examples of occurrences of the CUSTOMER entity.

An entity is rendered graphically as a rectangle with the name inside it (Figure 2.2). This name is unique across the whole schema: no other entity or relationship can share the same name.

Figure 2.2: Example of an entity in the ER model.

2.1.2 Relationships

A *relationship*, sometimes also called an *association*, establishes a logical link between two or more entities. In other words, it captures how entities are related to each other. RESIDENCE is an example of a relationship that associates entities PERSON and CITY, and EXAM is a possible relationship that can exist between the entities STUDENT and COURSE. When two entities are involved (it is usually said that they *participate*), the relationship is called *binary*. When three entities are linked together, it is known as *ternary*. An occurrence of a relationship is an n-tuple (a pair in the case of a binary relationship), whose elements are occurrences of the entities involved.

Figure 2.3: Example of a relationship in the ER model.

In an ER schema, a relationship is represented graphically by means of a diamond, containing the name of the relationship, and by lines that connect the diamond with each participating entity (Figure 2.3). As it was the case for entities, the name of a relationship must be unique.

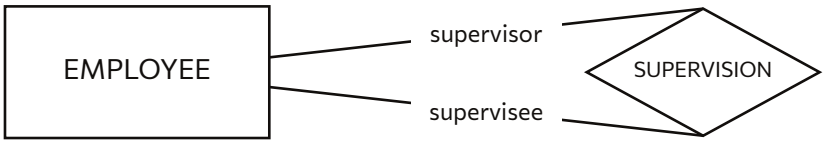

Figure 2.4: Example of a recursive relationship.

Recursive relationships are possible as well, that is, relationships between an entity and itself (Figure 2.4). In this case it is necessary to indicate the roles that the entity involved plays in the relationship.

Cardinality of Relationships

For each entity participating in a relationship, two values are specified. These describe the minimum and maximum number of relationship occurrences in which the entity occurrence can participate. In other words, they state how many times in a relationship between entities an occurrence of one of these entities can be linked to occurrences of the other entities involved.

In defining cardinalities of relationships, three values are used: zero, one and *N* (which is called "many" and indicates an integer greater than one). For the minimum cardinality, zero or one; the relationship is said to be *optional* or *mandatory*, respectively. For the maximum cardinality, one or many (N); in the first case each occurrence of the entity is associated with at most one occurrence of the relationship, while in the second case each occurrence of the entity is associated with an arbitrary number of occurrences of the relationship.

By looking at the maximum cardinalities, one can classify binary relationships in three distinct groups (illustrated in Figure 2.5):

- *one-to-one relationships*, having a maximum cardinality equal to one for both the entities involved;

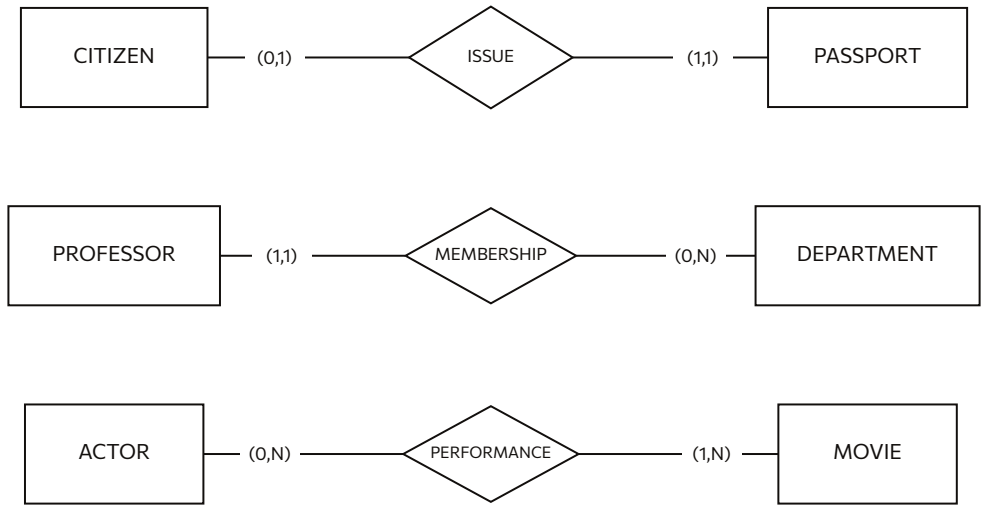

Figure 2.5: Examples of cardinality of relationships.

- *one-to-many relationships*, between an entity with maximum cardinality equal to one and another with maximum cardinality equal to N;
- *many-to-many relationships*, having a maximum cardinality equal to N for both the entities involved.

For what concerns ternary relationships, the participating entities must have maximum cardinality equal to N. Otherwise, the relationship would not be strictly ternary, as it would be possible to replace it by using only binary relationships.

2.1.3 Attributes

An *attribute* is a property associated with an entity or relationship, and whose purpose is to describe it. `LastName` and `Birthdate` are possible attributes of the CUSTOMER entity, while `Date` and `Mark` are possible attributes of the relationship EXAM between STUDENT and COURSE. An attribute can take on different values, and this set is known as the domain of the attribute.

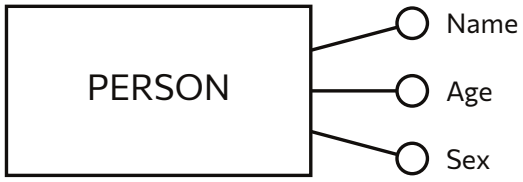

Figure 2.6: An example of attributes associated to an entity.

As depicted in Figure 2.6, when drawing the ER diagram, attributes are shaped as small circles arranged close to the entity or relationship they refer to and are connected to them by a line.

Cardinality of Attributes

For each occurrence of an entity or relationship, the associated attributes can assume a number of values between a minimum and a maximum. The cardinality of an attribute describes this range and, in most cases, it is equal to (1,1) and is omitted. If so, the attribute assumes a single value with each entity (or relationship) occurrence.

Figure 2.7: Example of an optional multivalued attribute.

However, the attribute may be null or may assume multiple values. In the former case, the minimum cardinality would be zero and the attribute is said to be *optional* (as opposed to *mandatory*, when the minimum cardinality is one). In the latter case, the maximum cardinality would be many (N) and the attribute is called *multivalued*.

Composite Attributes

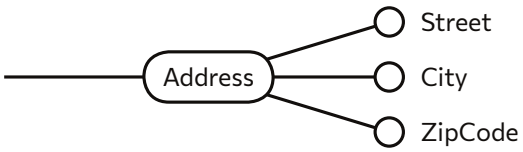

Figure 2.8: Example of a composite attribute.

It may be the case that an attribute is the natural union of a set of simpler attributes, that have connected meanings or uses. Such an attribute is qualified as *composite*. An example of a composite attribute is the attribute `Address` of `CUSTOMER`, with subattributes `Street`, `City`, and `ZipCode`. A graphical representation of a composite attribute is shown in Figure 2.8.

2.1.4 Internal and External Identifiers

Each and every entity in the schema must have an identifier, that makes it unique and distinguishable from all other entities. Often times, the identifier is formed by one or

more attributes of the entity itself, and it is known as *internal identifier* (or *key*). These attributes must necessarily have cardinality equal to (1,1). Figure 2.9 shows both the case of a single attribute and the case where multiple attributes form the identifier.

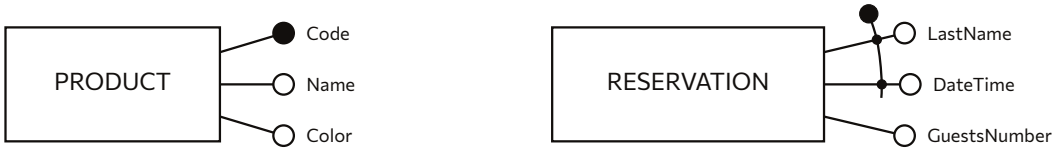

Figure 2.9: Examples of internal identifiers.

It may happen, however, that the entity attributes alone are not sufficient to identify the entity occurrences unambiguously. When this is the case, one or more other entities are involved for the identification. The entity is said to have an *external identifier* and is qualified as *weak* (Figure 2.10). This kind of identification is only possible if there exists a binary relationship between each external-identifying entity and the weak entity, and if the latter participates with cardinality equal to (1,1). Note that an external identifier can involve an entity that is in its turn identified externally, as long as cycles are not generated.

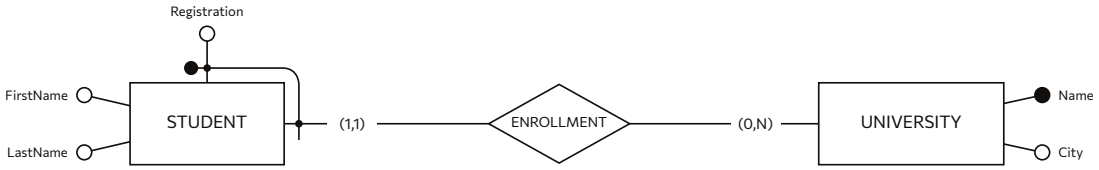

Figure 2.10: Example of an external entity identifier.

2.1.5 Generalizations

A *generalization* is a logical link between an entity E, referred to as *parent* entity, and one or more other entities $E_1, ..., E_n$, called *child* entities. We say that E is a *generalization* of $E_1, ..., E_n$, and that entities $E_1, ..., E_n$ are *specializations* of the E entity.

Every occurrence of a child entity is also an occurrence of the parent entity. In addition, every property of the parent entity (such as attributes, identifiers, relationships, and other generalizations) is also a property of each child entity. Clearly, this means that child entities in a generalization are exceptions to the rule that each entity must have an identifier (internal or external): they inherit it from the parent entity.

A generalization is said to be *total* if every occurrence of the parent entity is also an occurrence of one of the child entities, otherwise it is *partial*. A generalization is *exclusive*

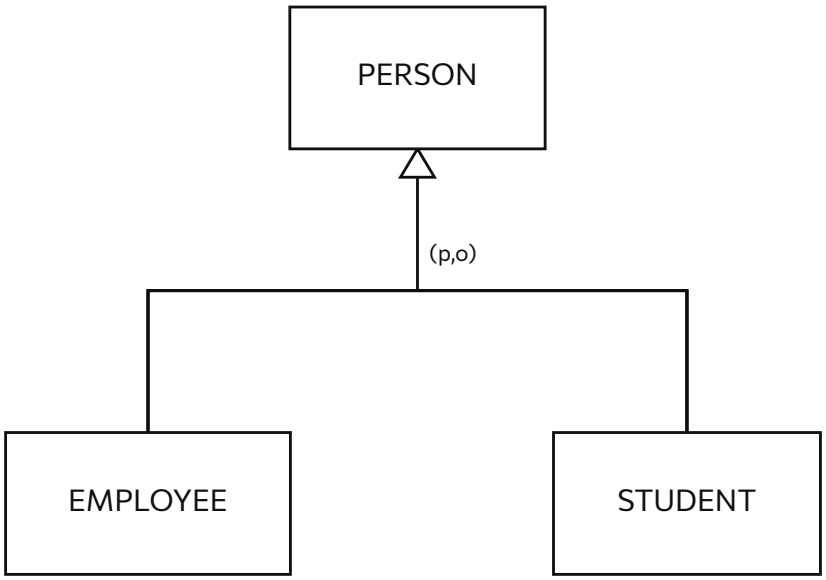

Figure 2.11: Example of a generalization (partial and overlapping).

if every occurrence of the parent entity is an occurrence of at most one of the child entities, otherwise it is *overlapping*.

In the conceptual schema, generalizations are indicated by arrows that go from child entities to the parent entity (Figure 2.11). It is also specified how the generalization is classified (total/partial and exclusive/overlapping).

Chapter 3

Logical Design

Once the conceptual design phase is complete, the next step is to construct a logical schema that correctly represents all the information described in the ER model. This activity consists of two main steps: ER schema *restructuring* and *translation* into the relational model [1]. Even though there are rules to be followed, the procedure is not automatic and requires design choices to be made.

3.1 ER Schema Restructuring

The Entity-Relationship schema is a high-level conceptual representation, and not all of its constructs naturally translate to the relational model. Generalizations and special attributes – namely, multivalued and composite attributes – need to be restructured before proceeding with the translation of the ER schema.

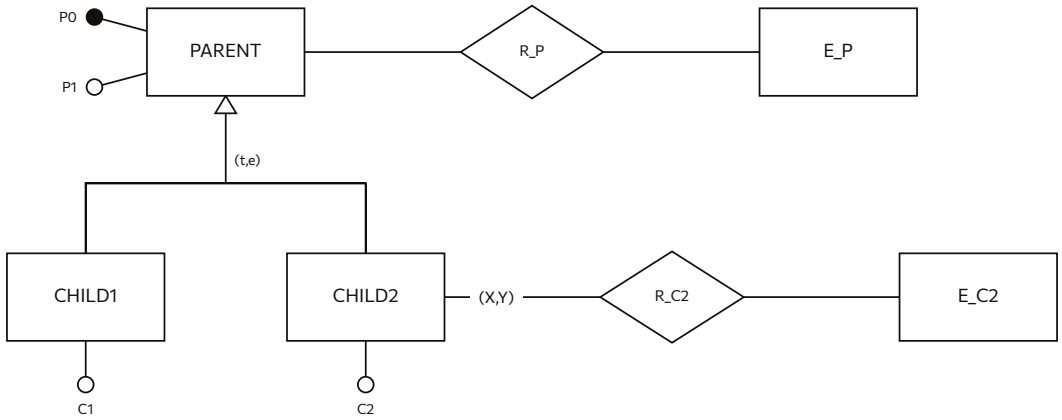

Figure 3.1: An ER schema with a generalization.

3.1.1 Generalizations

The relational model does not support the definition of complex constructs like generalizations. Therefore, it is necessary to replace them with simpler constructs that are easier to translate. It turns out that these constructs are nothing but entities and relationships, and that there are three possible options to replace a generalization. These will be demonstrated by taking as a reference the generic schema in Figure 3.1.

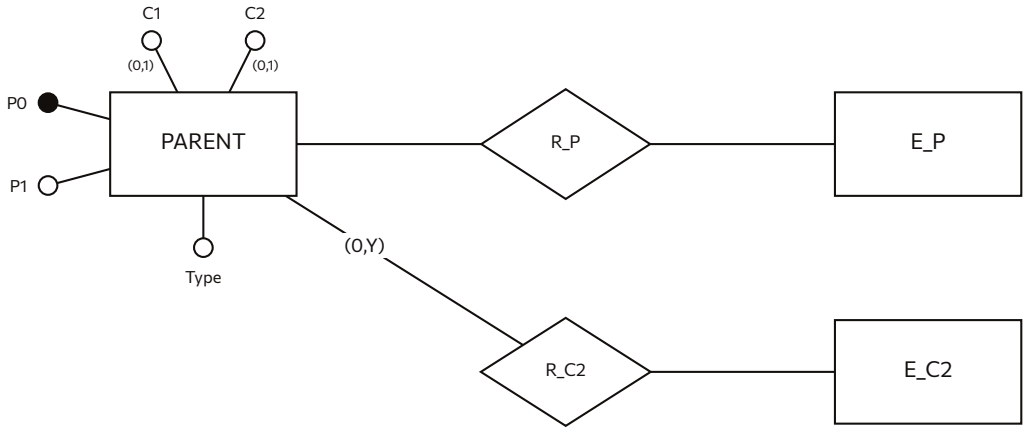

Figure 3.2: Collapsing the child entities into the parent entity.

- **Collapse the child entities into the parent entity.** Child entities are removed, and their attributes and participations in relationships are absorbed by the parent entity, with minimum cardinality set to 0. An attribute named `Type` is added to this entity, for the purpose of distinguishing occurrences of the different specializations. The restructured ER schema is shown in Figure 3.2.

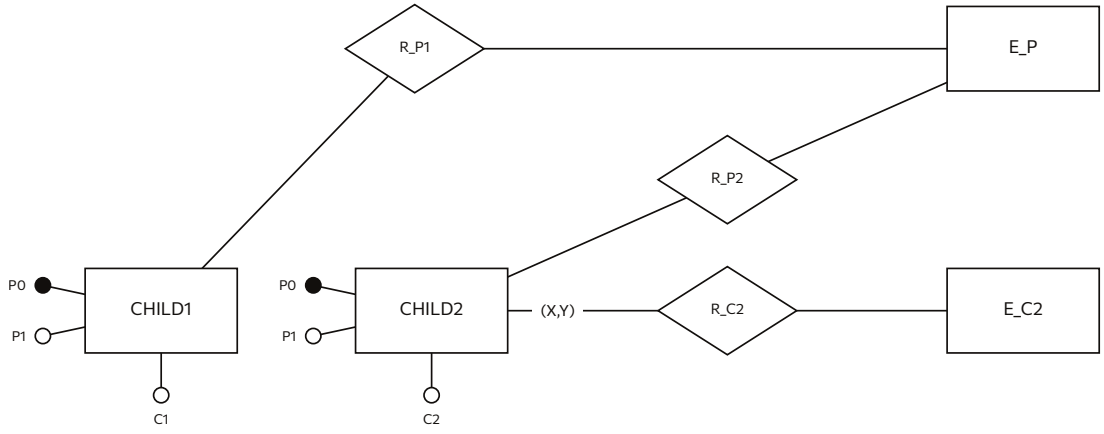

Figure 3.3: Collapsing the parent entity into the child entities.

- **Collapse the parent entity into the child entities.** The parent entity is removed, and its attributes are repeated (with the same cardinality) in each child entity. All relationships the parent entity was participating to are replicated for each child entity. This option is only available if the generalization is total and exclusive, and is illustrated in Figure 3.3.

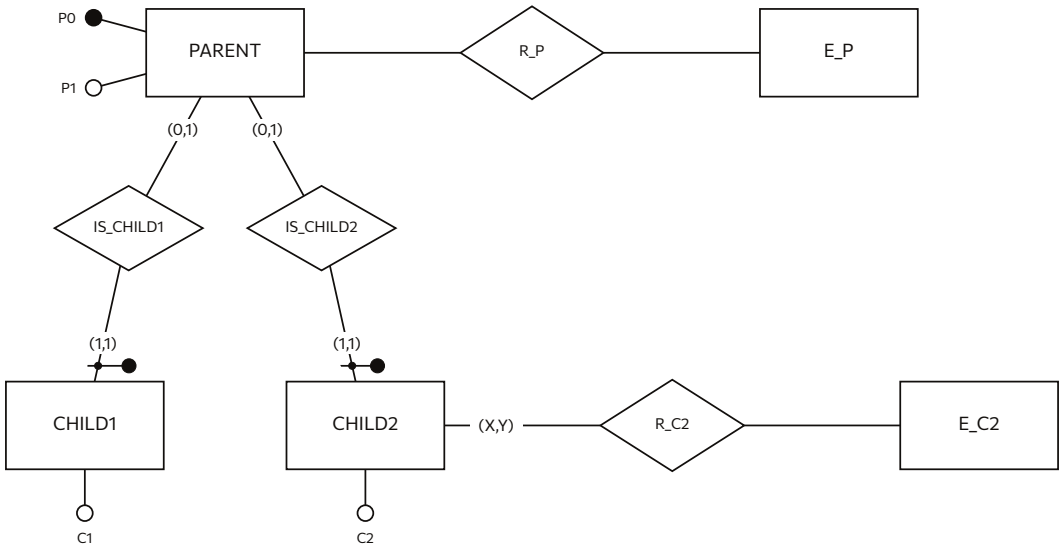

Figure 3.4: Substituting the generalization with binary relationships.

- **Substitute the generalization with relationships.** One-to-one relationships are used to connect the parent entity with each child entity, logically representing the same concept as the generalization. There are no transfers of attributes or relationships, and child entities are identified externally by the parent entity. This option, shown in Figure 3.4, is the most general and is always applicable.

3.1.2 Multivalued Attributes

In relational databases, table columns – what attributes eventually translate to – are not able to hold multiple values (e.g. arrays of strings). Consequently, among the tasks of the restructuring activity is to transform multivalued attributes in order to represent the same idea in a way compatible with the logical schema.

This is achieved by substituting the multivalued attribute with an entity-relationship combination (Figure 3.5). In particular, the attribute becomes an entity itself and is connected to the original entity through a relationship. The original entity participates to this relationship with a cardinality equal to the cardinality of the multivalued attribute, while the new entity can participate with cardinality equal to either (1,1) or (1,N).

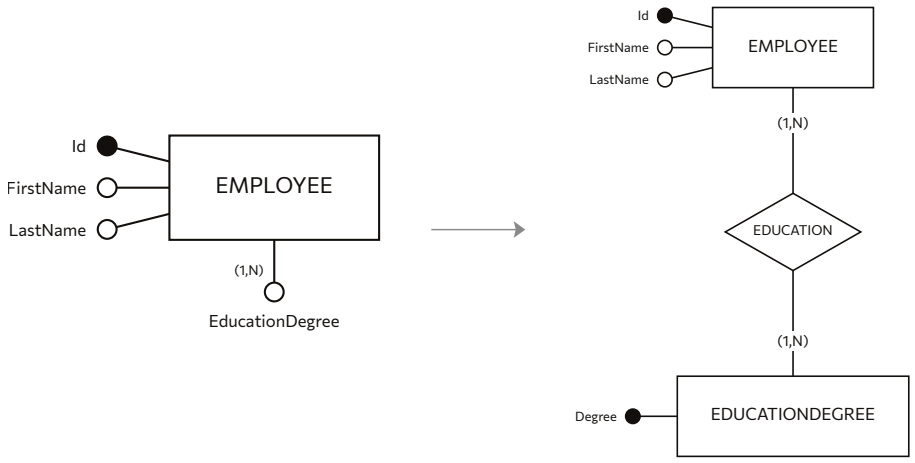

Figure 3.5: Example of a possible restructuring of a multivalued attribute.

3.1.3 Composite Attributes

Composite attributes are a conceptual formalization only possible in the ER schema. Before proceeding with the translation, they need to be restructured and converted to regular attributes.

Figure 3.6: Composite attribute restructured by merging subattributes together.

Here, two approaches are always possible. One option, illustrated in Figure 3.6, is to merge all subattributes into a single heterogeneous attribute, whose value will be the concatenation of the values of all subattributes. The alternative is to split the composite attribute and turn each subattribute into an attribute of its own (Figure 3.7).

Figure 3.7: Composite attribute restructured through decomposition.

3.2 Translation into the Relational Model

After restructuring the ER schema, all that is left are entities and relationships. There are precise rules for the translation of these fundamental constructs, and in some cases one has the possibility to choose between multiple alternatives. At the end of the translation step, a logical schema is obtained, made up of *relations* (or *tables*) possibly linked to each other by means of referential integrity constraints.

3.2.1 Entities

The way that entities are translated depends on whether they are identified internally or externally. In all circumstances, however, an entity is translated with a relation.

Entities with Internal Identifiers

In the case of an entity identified by one or more of its attributes, the translation is straightforward. The entity becomes a relation with the same name, having as attributes the same attributes as the entity and having its identifier as primary key (Figure 3.8).

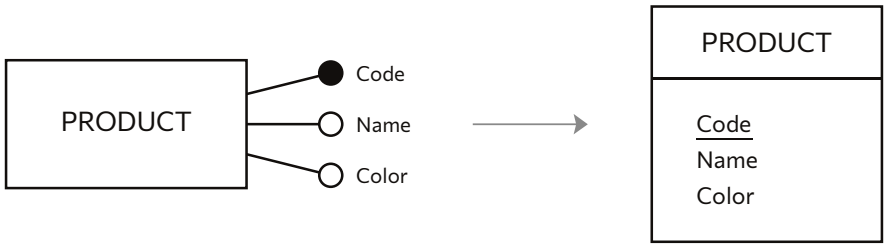

Figure 3.8: Translation of an entity identified internally.

Entities with External Identifiers

If the entity is identified externally, the approach is slightly different. For one thing, the relationship used for external identification is consumed in the translation. As in the previous case, the entity still becomes a relation with the same name, having as attributes the same attributes as the entity. This time, though, the key of the weak entity is formed putting together the identifiers of all the entities it depends on, plus any possible attribute taking part in the external identification.

Let us take the schema illustrated in Figure 2.10 as an example. The entity STUDENT is identified externally by the UNIVERSITY entity, through the ENROLLMENT relationship. The entity UNIVERSITY is identified internally and is translated as described previously.

The relationship ENROLLMENT goes away in the translation, and the weak entity STUDENT becomes a relation with the same name, having attributes FirstName and LastName. Eventually, two other attributes are included in relation STUDENT and form its key: Name (of relation UNIVERSITY, and renamed to University), since it was an identifier of entity UNIVERSITY (and the STUDENT entity depends on UNIVERSITY), and Registration (of entity STUDENT), since it was taking part in the external identification of its parent entity.

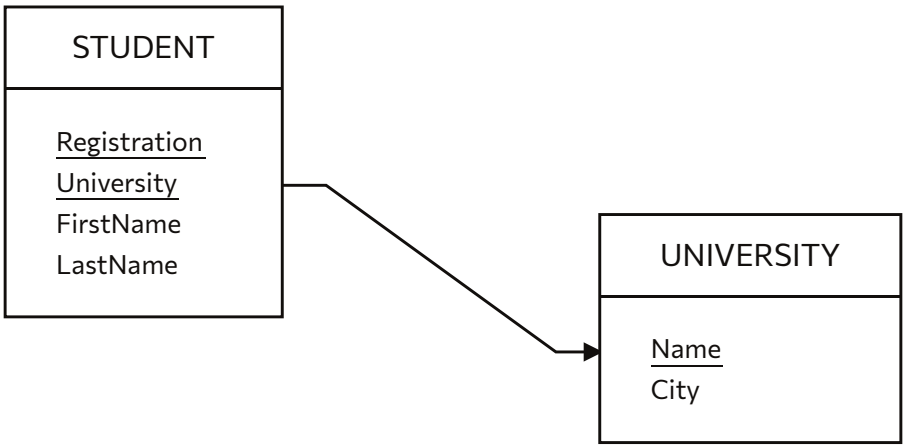

Figure 3.9: Translation of the schema in Figure 2.10.

Figure 3.9 shows the logical schema that is obtained after the translation step. When it happens that an attribute of a relation refers to another attribute, we have what is called a *referential (integrity) constraint* (further details are given in next sections of this chapter). In this case, University of STUDENT references Name of UNIVERSITY. This means that each occurrence of relation STUDENT must have a value of the University attribute for which an occurrence of relation UNIVERSITY exists with the same value in the Name attribute.

Since an external identifier can involve an entity that is in its turn identified externally, the translation of a weak entity is a recursive task. In order to avoid cascade translations happening simultaneously, as this would likely confuse users, our software enforces an order of operations: weak entities may be translated only after all entities they depend on have been translated.

3.2.2 Relationships

When translating relationships, some distinctions have to be made regarding both their type (binary or ternary) and, in the case of binary relationships, their cardinality (one-to-one, one-to-many, or many-to-many). In the following, we cover the way relationships are translated for all possible combinations of the above.

Many-to-many Binary Relationships and Ternary Relationships

As we have seen in Section 2.1.2, ternary relationships require entities to participate with maximum cardinality equal to N. Hence, it comes as no surprise that they behave in the same way as binary many-to-many relationships.

These relationships translate into a relation with the same name, having as attributes the attributes of the relationship and the identifiers of the entities involved; these identifiers, taken together, form the key of the relation.

One-to-many Binary Relationships

- **With mandatory participation: (1,1) – (*,N).**
  The entity participating with cardinality (1,1) absorbs the relationship, incorporating the attributes of the relationship and the identifiers of the (*,N) entity as external references.
- **With optional participation: (0,1) – (*,N).**
  Two alternatives are available:
  - The entity participating with cardinality (0,1) absorbs the relationship, incorporating the attributes of the relationship and the identifiers of the (*,N) entity as external references.
  - Translate it into a relation with the same name, having as attributes the attributes of the relationship and the identifiers of the entities involved; the identifiers of the (0,1) entity form the key of the relation, while the identifiers of the (*,N) entity are external references to that entity.

One-to-one Binary Relationships

- **With optional participation for one entity: (0,1) – (1,1).**
  The entity participating with cardinality (1,1) absorbs the relationship, incorporating the attributes of the relationship and the identifiers of the (0,1) entity as external references.
- **With mandatory participation for both entities: (1,1) – (1,1).**
  One can follow two approaches:
  - The first entity absorbs the relationship, incorporating the attributes of the relationship and the identifiers of the second entity as external references.
  - The second entity absorbs the relationship, incorporating the attributes of the relationship and the identifiers of the first entity as external references.

- **With optional participation for both entities: (0,1) – (0,1).**
  There are four possible choices:
  - Translate it into a relation with the same name, having as attributes the attributes of the relationship and the identifiers of the entities involved; the identifiers of the first entity form the key of the relation, while the identifiers of the second entity are external references to that entity.
  - Translate it into a relation with the same name, having as attributes the attributes of the relationship and the identifiers of the entities involved; the identifiers of the second entity form the key of the relation, while the identifiers of the first entity are external references to that entity.
  - The first entity absorbs the relationship, incorporating the attributes of the relationship and the identifiers of the second entity as external references. However, all these attributes are nullable since they come from an entity participating with (0,1) cardinality.
  - The second entity absorbs the relationship, incorporating the attributes of the relationship and the identifiers of the first entity as external references. However, all these attributes are nullable since they come from an entity participating with (0,1) cardinality.

3.2.3 Attributes

Optional Attributes

Whenever attributes of entities or relationships are described as optional, the corresponding attributes of relations can assume null values (they are said to be *nullable*). Figure 3.10 shows how, in the logical schema, these attributes are marked with an asterisk.

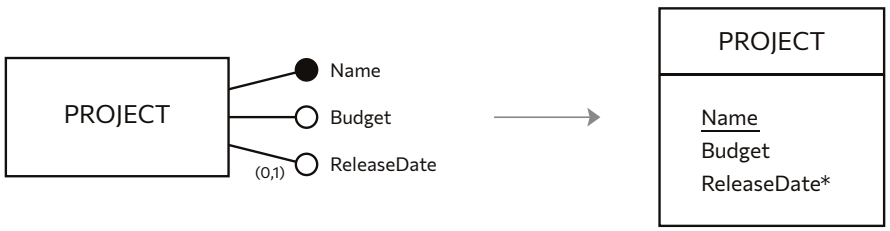

Figure 3.10: Optional columns of a table are indicated with an asterisk.

3.3 Towards Physical Design

The last phase of database design is the physical design. This process produces the physical schema of the database, which encompasses the definitions of the relations and of the physical access structures used, with the related parameters. The activity of physical database design can be very complex and is certainly outside the scope of this work, primarily because it very much depends on the DBMS of choice.

Nevertheless, some aspects are common to all implementations. In the following, we will have a look at the data types of table columns, the different constraints that can exist in a relational database schema, and the (standard) SQL language.

3.3.1 Data Types

The SQL standard groups predefined data types into types with similar characteristics:

- Character Types
  - Character (`CHAR`)
  - Character Varying (`VARCHAR`)
  - Character Large Object (`CLOB`)
- Binary Types
  - Binary (`BINARY`)
  - Binary Varying (`VARBINARY`)
  - Binary Large Object (`BLOB`)
- Numeric Types
  - Exact Numeric Types (`NUMERIC`, `DECIMAL`, `SMALLINT`, `INTEGER`, `BIGINT`)
  - Approximate Numeric Types (`FLOAT`, `REAL`, `DOUBLE PRECISION`)
- Datetime Types (`DATE`, `TIME`, `TIMESTAMP`)
- Interval Type (`INTERVAL`)
- Boolean
- XML
- JSON

3.3.2 Constraints

A *constraint* can be seen as a predicate that associates a *true* or *false* value with a database instance. In general, a collection of constraints is defined for a database schema and we consider *correct* (or *legal*) the instances that satisfy all such constraints. It is possible to classify the constraints in two categories, according to the elements of the database that are involved in it: *intra-relational constraints* and *inter-relational constraints*. Intuitively, the former ones involve a single relation and the latter ones take into account several relations.

3.3.3 SQL Language

SQL is an acronym for Structured Query Language, and it was one of the first commercial languages to utilize Codd's relational model. SQL has been standardized in the 1980s and has become the reference language for relational databases.

SQL is more than a mere query language. It consists of different classes of statements, informally referred to as sublanguages: the main ones are the *Data Definition Language* (DDL) and the *Data Manipulation Language* (DML). The first one has commands to define a relational database schema, while the second one has commands to modify and query a database instance.

Schema Definition

With SQL it is possible to define a database schema as a collection of *domains*, *tables*, *indices*, *assertions*, *views* and *privileges*. It is not necessary for all the components to be defined at the same time as the schema is created: this can take place in several successive phases. A schema has a name and an owner, and is created with the `CREATE SCHEMA` command.

Domain Definition

A database schema may have zero or more domains. A user-defined domain is characterized by its own name, by an elementary domain (either be predefined or previously user-defined), by a possible default value, and finally by a (potentially empty) set of constraints that represent the conditions that must be satisfied by legal domain values. It is possible to create a user-defined domain by means of the `CREATE DOMAIN` statement.

Table Definition

A table in SQL has a name and consists of an ordered set of attributes and of a (possibly empty) set of constraints. Each attribute, in turn, has a name and domain and possibly

a set of constraints. The table being created is initially empty and the creator holds all the privileges associated with it. In this case, the SQL command `CREATE TABLE` is used.

For example, the schema of a table DEPARTMENT is defined by means of the following SQL statement:

```
CREATE TABLE Department (
    Name     CHAR(20) PRIMARY KEY,
    Address  CHAR(50),
    City     CHAR(20)
);
```

Intra-Relational Constraints

The simplest intra-relational constraints are *not null*, *unique*, and *primary key*.

- **Not Null.**
  The null value is a special value, which indicates the absence of information. The *not null* constraint indicates that the null value is not admissible as the attribute value. If this is the case, the attribute must always be specified at the insertion stage. However, if a default value is associated with the attribute, it is possible to carry out an insertion without providing a value for the attribute, since the default value will be assigned to it automatically.

- **Unique.**
  A *unique* constraint imposes that one or more attributes of a table are a (super) key. Thus, it ensures that different rows do not possess the same values. Things are different for the null value, which can appear in various rows without violating the constraint, as it is assumed that each null value represents an unknown actual value different from that of another null value.
  Note that the definition of multiple unique constraints on single attributes is very different from the definition of a unique constraint on the set of those attributes.

- **Primary Key.**
  The *primary key* is the most important identifier for a relation. SQL allows a primary key constraint to be specified only once for each table. The primary key constraint can be directly defined on a single attribute, or by listing the several attributes that make up the primary key. It is not possible for these attributes to assume the null value, since the primary key constraint implies the not null constraint.

Inter-Relational Constraints

The most important inter-relational constraints are referential integrity constraints (introduced in Section 3.2.1). In the SQL language, the construct used to define them is the *foreign key constraint*. This constraint creates a link between the values of the attribute(s) of one table and the values of the attribute(s) of another table. The tables involved are referred to as *internal* and *external*.

Basically, for every row of the internal table the value of a given attribute, if not null, must be present among the values of a given attribute among the rows of the external table. The only requirement is that the attribute referred to in the external table has a unique constraint – that is, it identifies the tuples in the external table. This attribute is typically the primary key of the table, for which the unique constraint is guaranteed. The same goes for the case where a set of attributes are involved in the constraint.

Chapter 4

Technologies Adopted

This chapter is about the technical choices made in this project and aims to provide insights into the role of each technology in the final application. Discussions about APIs, framework features, or third-party libraries are kept general and are only meant to be an overview of the subjects. In some cases, code examples are reported to demonstrate possible usages of the features offered by such technologies.

4.1 Why a Progressive Web Application

Software applications can be deployed and delivered in a variety of forms, mainly through desktop, mobile, or Web applications. This decision likely depends on several factors, such as the target audience or the intended use of the software. While desktop applications have no competition in some use cases and mobile apps are unrivaled in many aspects, Web Applications have become increasingly popular and widely used for how easy it is to develop, distribute and update them.

Based on the intended users of our application, we could have used both the Web environment and the mobile one for the development of the design tool. The following sections investigate these two alternatives and provide a rationale for why it was decided to opt for the former in this work.

4.1.1 Mobile Apps and Web Apps

Clearly, seeking to determine the absolute winner among these two contenders is not the point. Note that they are not mutually exclusive, and in some cases it may be wiser to develop the product in both environments. Both have strengths and weaknesses, and either one may be the best fit in a particular scenario. Let us have a closer look at the main factors that differentiate them.

Native Mobile Apps

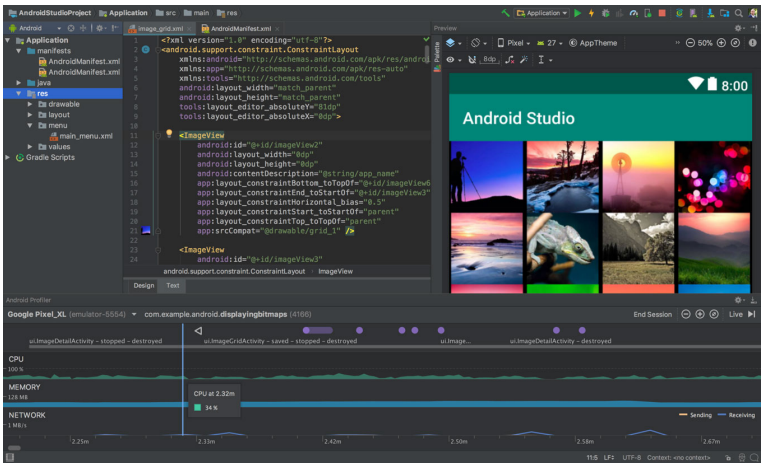

Figure 4.1: Native Android Apps are built using Android Studio [4].

Native mobile apps are built entirely in technologies that are specifically designed to leverage mobile operating system and hardware functions, without requiring additional layers to bridge gaps. These applications are distributed and downloaded by users through the *app store* specific to that mobile OS. The two dominating operating systems for smartphones and tablets are Android (Figure 4.1) and Apple iOS (Figure 4.2), which together constitute a 99% market share worldwide [5].

Figure 4.2: Apple XCode allows developers to build native iOS Apps [6].

Since they are targeted to a particular mobile platform, native apps only work on devices from that specific vendor and are incompatible with other environments. However, this specificity allows them to take full advantage of the functionality built into the operating system, leading to better performance, consistency, and an unparalleled user experience.

Web Apps

A Web-based app is an Internet-enabled application that is accessible via the browser and is coded in HTML, CSS, and JavaScript. Users are not required to download and install the app on their device in order to access it. Web Applications are cross-platform, meaning that they are not designed and developed to be used only on a particular operating system. The browser is their only requirement, so they can be accessed and used from all sorts of devices: desktop computers, smartphones, and tablets (Figure 4.3). However, this limits their capabilities as far as accessing the device features is concerned.

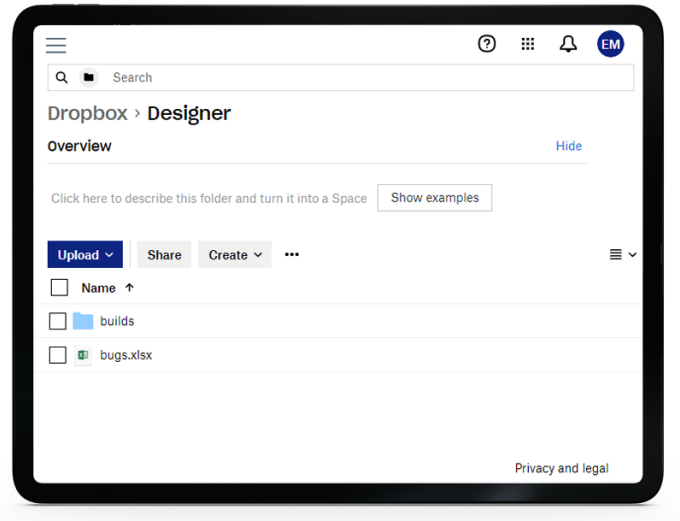

Figure 4.3: A Web Application used on a tablet through a browser.

Web Applications are evolving rapidly, becoming more and more powerful. In the next sections, we will see that a lot of the limitations in typical Web Applications have been addressed. New development technologies and philosophies like Single-Page Applications (SPAs) and Progressive Web Applications (PWAs) bridge several major functional gaps and enable richer and almost native-like mobile experiences.

4.1.2 Single-Page Applications

Single-Page Applications are JavaScript-driven Web Applications that interact with the user by updating the page dynamically, instead of performing a full reload or transferring control to another page. The intention is to make transitions faster and give the user the impression of a more organic and immersive experience, as is the case for native apps.

In this kind of applications, all resources needed to run the software (JS, CSS, and HTML) are downloaded by the browser in a single page load. After the initial page load, no more HTML gets sent over the network; instead, only content and data are requested

from the server (or are sent to the server). Exchanging only data instead of entire Web pages is clearly an advantage in terms of both time and bandwidth.

The architecture of modern Single-Page Applications is illustrated in Figure 4.4.

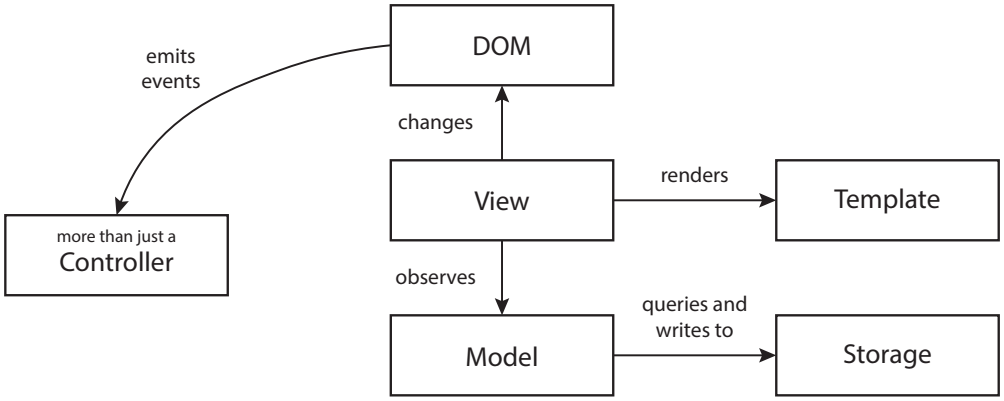

Figure 4.4: High-level architecture of Single-Page Applications.

More specifically, this structure is based upon the following fundamental principles [7]:

- **Write-only DOM.**
  No state or data is read from the DOM. Storing state in the DOM is cumbersome and gets difficult to manage very quickly. A much better alternative is to store the data in a single place, separate from the presentation layer, and to render the UI from this data.
- **Models as the single source of truth.**
  There is a single source of truth, the model (not the DOM, and not random objects spread across the application). Changes happen in one place only and these updates propagate to the entire application. In this way, state management is easier and less error-prone, and the possibility of duplicated or inconsistent data is removed.
- **Views observe model changes.**
  The views reflect exactly the data in the models. It may be the case that multiple views depend on a single model. When this model changes, it is not the job of the model to update the dependent views, nor to keep track of them. Instead, an event system makes sure that views receive notifications when the data in the model changes, and then these views update and redraw themselves accordingly.
- **Decoupled modules that expose small external surfaces.**
  Components and subsystems should be designed and implemented to be as much

reusable and independent as possible. Dependencies increase overall complexity and make the code hard to test. On the other hand, low coupling and small external surfaces make refactoring easy and code easier to maintain.

4.1.3 Progressive Web Applications

Progressive Web Applications are the new phenomenon of application landscape. They are Web Applications that use emerging Web browser APIs and features along with traditional progressive enhancement strategy to bring a native app-like user experience to cross-platform Web Applications. In order to call a Web Application a PWA, technically speaking it should have the following features: secure contexts (HTTPS), one or more Service Workers, and a manifest file [8].

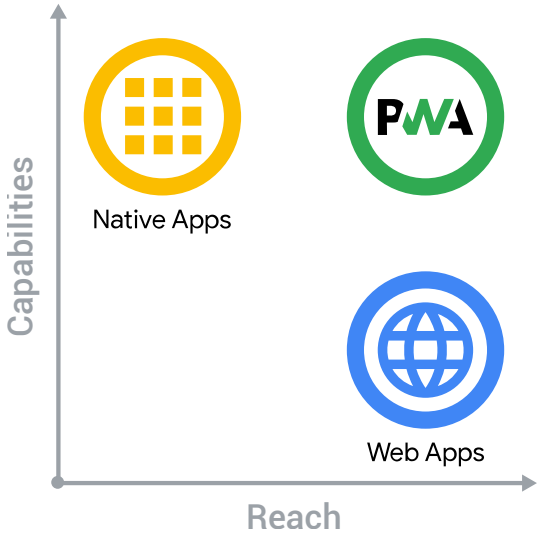

Figure 4.5: Capabilities vs reach of native apps, web apps, and progressive web apps [9].

PWAs are not created with a single technology. They represent a new philosophy for building Web Applications, involving some specific patterns, APIs, and other features. In particular, there are some key principles Web Applications should try to observe to be identified as Progressive Web Applications. They should be:

- **Discoverable.** Are identifiable as "applications" thanks to W3C Manifests and Service Worker registration, can be easily discovered by search engines, are easier to expose, catalog and rank, and have metadata usable by browsers to give them special capabilities.
- **Installable.** Can be installed on the home screen via browser-provided prompts (a feature called *Web app installation*, shown in Figure 4.6). Installing a PWA allows it to look, feel, and behave like all other installed apps: it is launched from the same

place users launch their other apps, it runs in its own window (separate from the browser), and it appears in the task list.

Figure 4.6: The Add to Home Screen feature allows PWAs to be installed on devices.

- **Linkable.** Can be easily shared via a specific URL, without the need for an App store or complex installation process.
- **Network independent.** Can work when the network is unreliable, or even non-existent, thanks to Service Workers. Where connectivity is not strictly required, the application works the same offline as it does online: an offline PWA provides a true app-like experience for users.
- **Progressive.** Work for every user, regardless of browser choice, using best practices such as progressive enhancement. They provide an excellent experience to fully capable browsers and an acceptable experience to less capable browsers.
- **Re-engageable.** Can use push notifications to maintain engagement with the user, sending them updates and new content even when they are not looking at the application or using their devices.
- **Responsive.** Work on any screen size and all of the content is available at any viewport size. Devices come in a range of sizes, and applications may be used at a range of sizes, even on the same device. Therefore, it is critical to ensure that the

UI will fit any form factor: desktop, mobile, or tablet.

- **Safe.** Are served via HTTPS (a Service Worker requirement) to prevent snooping and ensure content has not been tampered with. Also, it is easy for users to know whether they are installing the right app, because its URL will match the product website domain. This is different from applications in App stores, which may have a number of similarly-named apps and mislead the user. Web Applications eliminate that confusion and ensure that users get the best possible experience.

The main technology required for Progressive Web Applications is Service Worker support. Fortunately, Service Workers are now supported on all major browsers on desktop and mobile. Other features such as Web App Manifest, Push Notifications, and Add to Home Screen functionality have wide support too.

4.2 Web APIs

When writing code for the Web, there are a large number of Web APIs available. All browsers have a set of built-in Web APIs to support complex operations, to help accessing data and to perform various useful tasks. These can be accessed using JavaScript code, and follow the recommendations and open standards developed by the World Wide Web Consortium (W3C).

In this work, we made use of three of these technologies: the Web Storage API, the Canvas API, and the Touch Events specification. In the following is explained what they are about and how they were used in this project. Occasionally, some simple usage examples are also reported.

4.2.1 Web Storage API

The Web Storage API provides mechanisms by which browsers can store key/value pairs, in a much more intuitive fashion than using cookies [10]. Indeed, Web Storage differs from cookies in some key ways:

- Purpose. Cookies are primarily used when communicating with the server and are automatically included in requests; they can be accessed by both the client and the server. In contrast, Web Storage is purely designed for use on the client-side: the server can neither directly read from nor write to it.
  As an example, the user's preferred language could be saved in a cookie (since this information is needed by the server to provide content in the right language), while the preference about the page font size could be stored using Web Storage.

- Storage size. While cookies are usually limited to 4096 bytes, Web Storage capacity can be up to about 5MB or 10MB, depending on the browser.
- Interface. Web Storage provides a more organic programmatic interface than cookies. It is possible to access values like an object or, even better, through the use of the dedicated methods `getItem()` and `setItem()`.

Current versions of all major browsers support this technology. The two mechanisms within Web Storage are `sessionStorage` and `localStorage`, behaving similarly to session cookies and persistent cookies respectively:

- `sessionStorage` maintains a separate storage area for each given origin that's available for the duration of the page session (as long as the browser is open, including page reloads and restores).
- `localStorage` does the same thing, but persists even when the browser is closed and reopened.

Example

The Web Storage API allows to securely store key/value pairs. The keys and the values are always strings, so it may be necessary to convert or serialize data which is not in string format. Since the objects `windowStorage` and `localStorage` are exposed as properties of the global object `window`, the JavaScript code to use the Web Storage API is straightforward:

```
// save color preference
localStorage.setItem('colorSetting', '#f5df4d');

// read color preference
const color = localStorage.getItem('colorSetting');
// use white if no preference was set
setBgColor(color || '#fff');
```

How We Use It

At the current stage, our project exploits the Web Storage API to implement the autosaving feature described in Section 5.5.2. Whenever it is required to save the user progress, the state of the application is serialized in JSON format and stored in the `localStorage` object. Correspondingly, at application startup, the software checks whether a record is present in the local storage and possibly restores the previously saved state.

4.2.2 Canvas API

The Canvas API provides a means for drawing graphics via JavaScript and the HTML <canvas> element. Among other things, it can be used for animation, game graphics, data visualization, photo manipulation, and real-time video processing [11].

The Canvas API largely focuses on 2D graphics. It is a low-level, procedural model that updates a bitmap and does not have a built-in scene graph. Although extremely powerful, it is not always simple to use. For this reason, a number of libraries exist that can make the creation of canvas-based projects faster and easier.

This API covers every aspect of two-dimensional graphics: from basic ones like drawing shapes, applying styles and colors, drawing text, and using images to more advanced ones such as transformations, compositing and clipping, animations, pixel manipulation, hit regions and accessibility.

Example

If a <canvas> element exists in the DOM, it is enough to obtain its context to start drawing onto it. For instance, drawing a red 100x100 rectangle, requires only a few lines of code:

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');
ctx.fillStyle = 'red';
ctx.fillRect(20, 20, 100, 100);
```

How We Use It

Drawing of Entity-Relationship and Logical schemas was not accomplished using the Canvas API directly; rather, as presented later in this chapter, a third-party drawing library has been adopted. However, the diagram exporting feature has only been possible because of this interface. Once the schema is drawn to the canvas, the `toBlob()` method of the `HTMLCanvasElement` interface creates a `Blob` object representing the image contained in the canvas and can then be downloaded as an image file.

Also, pixel manipulation capabilities were exploited in order to give exported images a transparent background (instead of the default white one). This was achieved with the `getImageData()` and `putImageData()` methods of the canvas context, that allow to directly read and write a data array to manipulate pixel data.

4.2.3 Touch Events

To provide quality support for touch-based user interfaces, the Touch Events specification offers the ability to interpret finger (or stylus) activity on touch screens or trackpads [12].

The touch events interfaces are relatively low-level APIs that can be used to support application-specific multi-touch interactions such as a two-finger gesture. A multi-touch interaction starts when a finger (or stylus) first touches the contact surface. Other fingers may subsequently touch the surface and optionally move across the touch surface. The interaction ends when the fingers are removed from the surface. During this interaction, an application receives touch events during the start, move, and end phases.

Touch events are similar to mouse events except they support simultaneous touches and at different locations on the touch surface. The `TouchEvent` interface encapsulates all of the touchpoints that are currently active. The `Touch` interface, which represents a single touchpoint, includes information such as the position of the touch point relative to the browser viewport.

Example

The user action of moving a finger inside an element in the page can be easily intercepted by adding the appropriate event handler:

```
const panel = document.getElementById('panel');
const position = document.getElementById('position');
panel.addEventListener('touchmove', event => {
    // get finger coordinates
    const x = event.touches[0].clientX;
    const y = event.touches[0].clientY;
    position.innerHTML = `${x}, ${y}`;
});
```

The `event` variable contains the property `touches`, a `TouchList` of all the `Touch` objects representing all current points of contact with the surface. A `Touch` object has several properties, for example `identifier` (same finger means same identifier), `target` (the element being touched), `clientX` and `clientY` (the coordinates relative to the browser window, regardless of scrolling), etc.

How We Use It

The editor supports panning and zooming, and does this by interpreting mouse events (click and move, or scroll). To make this behavior available also in mobile devices like

smartphones and tablets, the resources provided by the Touch Events interfaces have been used.

Figure 4.7: Examples of smartphone interaction gestures (designed by Freepik).

These touch events mechanisms allowed for the implementation of a cutting-edge navigation system, that lets users move around in a smooth, precise, and intuitive way. This state-of-the-art interface is able to respond to gestures like *drag*, *pinch*, *spread* (Figure 4.7), and even a combination of drag and pinch/spread.

4.3 Vue.js JavaScript Framework

When it comes to building a Web Application, one of the first choices to be made is about the core library that will support the software package. Several different frameworks are available, each with its own vision, core concepts, characteristics (strengths and, inevitably, weaknesses), and community support. An alternative choice, totally legitimate, is to use the JavaScript language directly, without any intermediation from third-party libraries. This option goes by the name of *vanilla JavaScript* and has its advantages and disadvantages: it is faster and lighter, but it is also harder to maintain and one runs the risk of reinventing the wheel.

Vue.js is a progressive framework for building user interfaces [13]. While its core library focuses on the view layer only, its ecosystem of supporting libraries makes it capable of powering large and complex Single-Page Applications. It is an MIT-licensed open source project, and its latest version (Vue 3) brings new features and breaking changes. In the following are described some characteristics of Vue, its most recurring idioms, and features that distinguish it from other frameworks.

4.3.1 Overview

At the core of Vue.js is a system that enables us to declaratively render data to the DOM using straightforward template syntax. The data and the DOM become linked,

and everything is *reactive*. It is possible to bind data to not only text and attributes, but also the structure of the DOM (taking advantage of features like *conditional rendering* and *class and style bindings*).

Vue.js uses an HTML-based template syntax and, under the hood, compiles the templates into Virtual DOM render functions. Combined with the reactivity system, Vue is able to intelligently figure out the minimal number of components to re-render and apply the minimal amount of DOM manipulations when the application state changes.

Let us have a look at a few concrete examples. The most basic form of data binding is text interpolation using the "mustache" syntax (double curly braces):

```
<span>Message: {{ msg }}</span>
```

The mustache tag will be replaced with the value of the `msg` property from the corresponding component instance. It will also be updated whenever the `msg` property changes. And here is how the `v-bind` directive is used to reactively update an HTML attribute:

```
<a v-bind:href="url"> ... </a>
```

This will bind the element's `href` attribute to the value of the expression `url`. Finally, the `v-on` directive, typically shortened to the `@` symbol, is used to listen to DOM events and run some JavaScript when they are triggered:

```
<button @click="counter += 1">Add 1</button>
```

The next sections explore the topics of *components* and *reactivity*. While the first one is common to several Web Application frameworks, the unobtrusive reactivity system is one of Vue's most distinct features.

4.3.2 Components

Components are reusable instances with a name. It is common for an app to be organized into a tree of nested components (Figure 4.8). Components are one of the most powerful features of Vue: they allow extending basic HTML elements to encapsulate reusable code.

Components are meant to be used together, most commonly in parent-child relationships: Component A may use Component B in its own template. They inevitably need to communicate to one another: the parent may need to pass data down to the child, and the child may need to inform the parent of something that happened in the child. However, it is also very important to keep the parent and the child as decoupled as

possible via a clearly-defined interface. This ensures each component's code can be written and reasoned about in relative isolation, thus making them more maintainable and potentially easier to reuse.

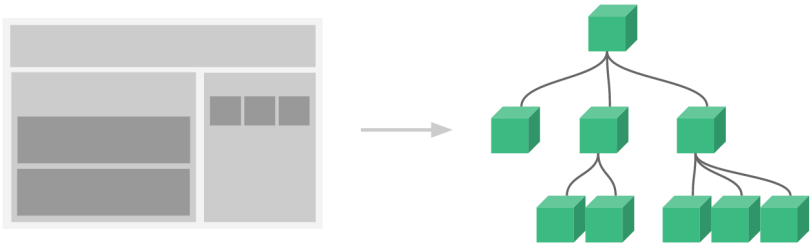

Figure 4.8: An app is organized in smaller units, called components [14].

In Vue, the parent-child component relationship can be summarized as *props down*, *events up*. The parent passes data down to the child via props, and the child sends messages to the parent via events.

4.3.3 Reactivity

Reactivity is a programming paradigm that allows us to adjust to changes in a declarative manner (Figure 4.9). In JavaScript, it is implemented using proxies: a `Proxy` is an object that encases another object or function and allows to intercept and redefine fundamental operations for that object.

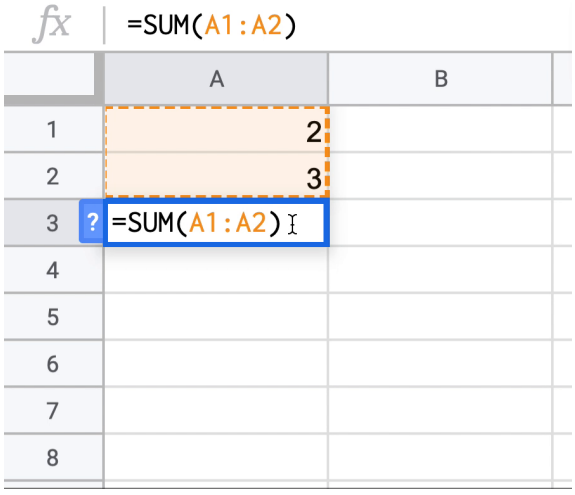

Figure 4.9: A spreadsheet formula is great example of reactivity.

To create a reactive state from a JavaScript object, one can use the *reactive* method. The essential use case for the reactive state in Vue is that it can be used during render. Thanks to dependency tracking, the view automatically updates when the reactive state changes.

Sometimes we need state that depends on other state. In Vue, this is handled with *computed properties*. To directly create a computed value, the `computed` method is used: it takes a getter function and returns an immutable reactive `ref` object for the returned value from the getter.
While computed properties are more appropriate in most cases, there are times when a custom *watcher* is necessary. That is why Vue provides a more generic way to react to data changes through the `watch` option. This is most useful when performing asynchronous or expensive operations in response to changing data.

4.4 Two.js Drawing API

Figure 4.10: The Two.js logo, made with Two.js.

Two.js is a two-dimensional drawing API geared towards modern Web browsers [15]. It offers the same set of renderer-agnostic functions to draw in multiple contexts: `svg`, `canvas`, and `webgl`. It is deeply inspired by flat motion graphics and aims to make the creation and animation of flat shapes easier and more concise.

4.4.1 Overview

At its core, Two.js relies on a *scenegraph*. This means that when an object (a `Two.Path` or `Two.Group`) is drawn or created, Two actually stores and remembers that. After its creation, and until the object is removed from the scene, it is possible to modify it and apply a number of operations to it (e.g. change its rotation, translation, scale, etc.).

To start with, an instance of Two has to be created and attached to an element in the page (it is possible to specify properties and construction parameters as needed):

```
const canvas = document.getElementById('canvas');
const params = { width: 285, height: 200 };
const two = new Two(params).appendTo(canvas);
```

The simplest way to use Two.js is to draw shapes and organize them in groups for easier management. In the next paragraphs, we explore the main features of Two and provide some simple examples of its usage. Of course, the actual tool is much more powerful and complex than what is shown here, and complete documentation can be found on the project website.

Drawing Shapes

After creating a Two instance it is possible to start drawing shapes using the convenience functions provided by the library. For instance, making a 150x100 semitransparent dark-navy rectangle boils down to a few trivial instructions:

```
// Two has convenience methods to create shapes
let rect = two.makeRectangle(50, 50, 150, 100);

// the object returned has many stylable properties
rect.fill = 'rgb(2, 7, 93)';
rect.opacity = 0.5;
rect.noStroke();

// render everything to the screen
two.update();
```

Whenever object properties are changed, as in the example above, it is necessary to call the `update()` method to tell Two that the scene needs to be redrawn.

Arranging Elements in Groups

Adding shapes to groups makes them easier to handle. Groups provide a simple way to move content around through the `translation`, `rotation`, and `scale` properties. To add some shapes to a group the `makeGroup()` function is used, that returns a `Two.Group` object:

```
// groups can take an array of shapes and/or groups
let group = two.makeGroup(rect, text);

// and have translation, rotation, and scale like all shapes
group.translation.set(two.width / 2, two.height / 2);
group.rotation = Math.PI / 4;
group.scale = 1.15;

// it is also possible to set the same properties that a shape has
group.linewidth = 4;

two.update();
```

It is worth noting that all rendered objects in Two are children of a group. Every Two instance has the `scene` property, which is a root-level `Two.Group` and can act as a camera by means of the same transformations described above.

Transforms and Masks

Groups and shapes can be transformed using their `translation`, `rotation`, and `scale` properties. These operations emit from the coordinate space `(0,0)`. These properties are pretty self-explanatory: `translation` is a `Two.Vector` that represents the x, y translation of the object in the drawing space; `rotation` is a number that represents the rotation of the object in the drawing space, in radians; and `scale` is a number that represents the uniform scale of the object in the drawing space.

Another powerful feature of Two is masking. At its most basic level, masking is a way of making parts of a group invisible. But the beauty of masking is that it is non-destructive – meaning, we can make something invisible but still be able to make it visible again at any time. In Two, a group has the `mask` property that can be set to a `Two.Path` object that masks the content within the group.

Chapter 5

Product Design and User Experience

Developing a software product is much more than just writing code. It is the process of creating an artifact that aims to integrate the needs of users and the possibilities of technology. Simplicity and intuitiveness must go hand in hand when designing a piece of software, so as to create a product that is functional and appealing.

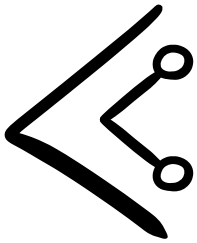

Figure 5.1: The Designer logo.

Designer is the result of our efforts to create a user-friendly, captivating software tool for computer-aided database design. We have realized that there was no choice but to completely redesign the user experience from the ground up, putting ourselves in the users' shoes and anticipating their needs. As part of the visual identity of the product, the software has its own logo (depicted in Figure 5.1), that evokes the shape of some basic constructs of the Entity-Relationship model.

5.1 User Interface

User Interface design is a subset of a field of study called *human-computer interaction*. Human-computer interaction is the study of how people and computers work together so that a person's needs are satisfied in the most effective way. The User Interface is the part of a software that people can see, hear, touch, talk to, or otherwise understand or

direct. It has essentially two components: input and output. A proper interface design is the one that provides a mix of well-designed input and output mechanisms that satisfy the user's needs, capabilities, and limitations in the most effective way possible. The best interface is one that is not noticed, and one that allows the user to focus on the information and task at hand instead of the mechanisms used to present the information and perform the task [16].

5.1.1 Minimal UI for Maximum Impact

Nick Babich, in his article *The Art of Minimalism in Mobile App UI Design* [17], writes that *"minimalism is a perfect marriage of form and function. Its greatest strength is clarity of form – clean lines, generous whitespace, and minimal graphical elements brings simplicity to even the most confounding subject matter. That is, of course, if it is used effectively."*

In order to achieve beautiful design and great usability, everything has to be kept concise, clear, and consistent. A few key principles guide designers on how to shift towards a simpler interface, removing all unnecessary visual design details in the UI. Among them are:

- **Simple color scheme.**
  A simple color scheme has a positive effect on user experience, while having too many colors could be detrimental. Also, it is important to have a balanced saturation and enough contrast. A three color combination is a good starting point: it is enough to create variation and visual interest. There are many good ways to choose a color palette, for example monochromatic, analogous, triadic, or split complementary. The important thing is to not use colors in equal amounts, and instead exploit them to visually prioritize elements and highlight important details.
- **Consistent typeface.**
  Multiple typefaces rarely result in a better user experience. In reality, mixing several different fonts can make the app seem fragmented and inconsistent. Reducing the number of fonts on a screen can reveal the power of typography: a great user interface should be delivered by playing with font weight, style, and size, not by using different typefaces.
- **Visual hierarchy and emphasis.**
  Visual hierarchy concerns the arrangement of elements in a way that implies importance. Visual contrast (emphasis) can be achieved by size, proximity, color, opacity, and actual tonal contrast between elements. For example, things like obscuring backgrounds behind popups or modal boxes, or using neutral colors for the general scheme and adding contrasting colors for calls to action, help the user focus on the action that should be taken.

5.1.2 Tool UI Appearance

With all these considerations in mind, we strived to conceive a graphical interface that would be as clean and direct as possible. On a desktop window, the application looks like shown in Figure 5.2. The space available is almost entirely devoted to the model editor, without any panels or windows for element properties/outlines.

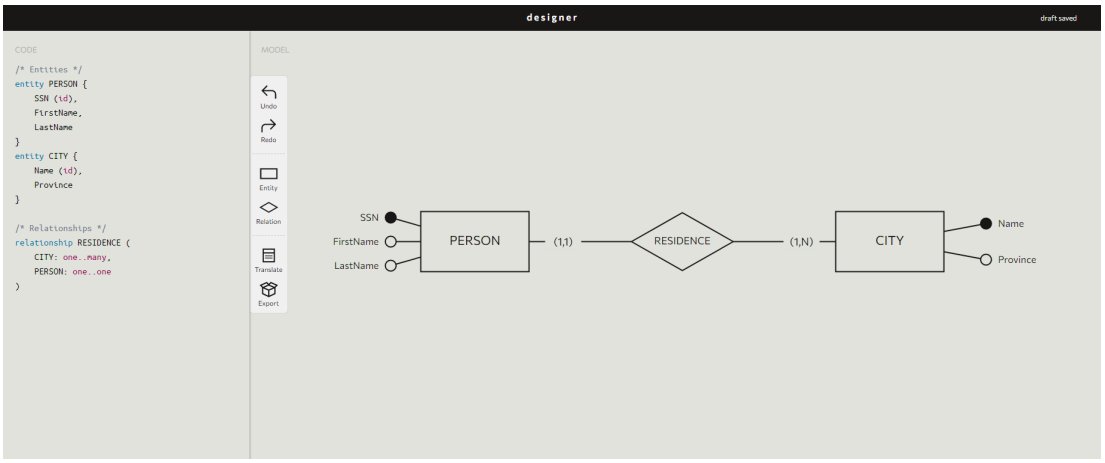

Figure 5.2: The User Interface of the Designer app.

Minimal UI, careful color usage, clean shapes, and proper typeface selection make the software look smart and engaging. The interface often includes icons that simplify the usage and understanding of the tool, especially for inexperienced users. The main parts of the graphical interface are the editor, the toolbar, and the code panel.

Editor

The editor is the beating heart of the whole application: this is where the user creates the design and where the diagram is rendered graphically. Here the schema drawing process happens continuously, reflecting the model changes observed in the application state and handling user interactions such as mouse clicks and touch gestures. The drawing area is not limited to the size of the window panel but extends indefinitely and can be navigated with ease (see Section 5.1.4 for more details).

Toolbar

The vast majority of actions are performed using the toolbar on the left-hand side of the editor (Figure 5.3). As explained at greater length in Section 5.2.2, this is not a static component. Instead, different commands are available depending on the current step of the design and the type of the selected element in the model (if any). Furthermore,

when an action cannot be performed, the relative button is disabled and does not respond to user input (this condition is also represented graphically by making the button semitransparent). Additional insights about the toolbar are given when discussing User Experience design (Section 5.2).

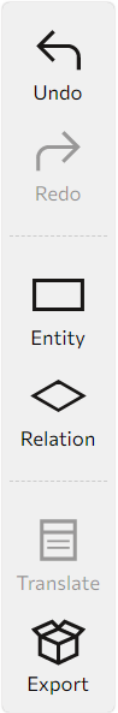

Figure 5.3: The application toolbar (as it appears in the ER design step).

Code Panel

In certain steps of the design process, a code section is present, which provides a textual representation of the model and reflects changes that happen in the editor. In the conceptual design step, it shows a sort of *pseudo-code* that describes the Entity-Relationship model in textual form (we called it *ER code*). In the logical schema editing step, it shows SQL code in a generic dialect, for exporting the design to a DBMS of choice. A syntax highlighting feature (powered by the Prism library [18]) was added, to make the code more readable and easily comprehensible.

5.1.3 Responsive Web Design

In the early days of Web design, pages were built to target particular screen sizes. If the user had a larger or smaller screen than expected, results ranged from unwanted

scrollbars to overly long line lengths, and poor use of space. As more diverse screen sizes became available, the concept of *Responsive Web Design* (RWD) appeared, a set of practices that allow Web pages to alter their layout and appearance to suit different screen widths, resolutions, etc. [19].

One of the core qualities of Progressive Web Applications (presented in Section 4.1.3) is their ability to adapt to any viewport size. This requires focusing on only the most important data and actions in an application, as there is simply no room for extraneous, unnecessary elements. Figure 5.4 gives a glimpse of how the Designer app adapts to a smartphone screen: as there is not enough room to fit both the code and the model sections, the user is given the opportunity to show them alternatively by toggling a blue sidebar.

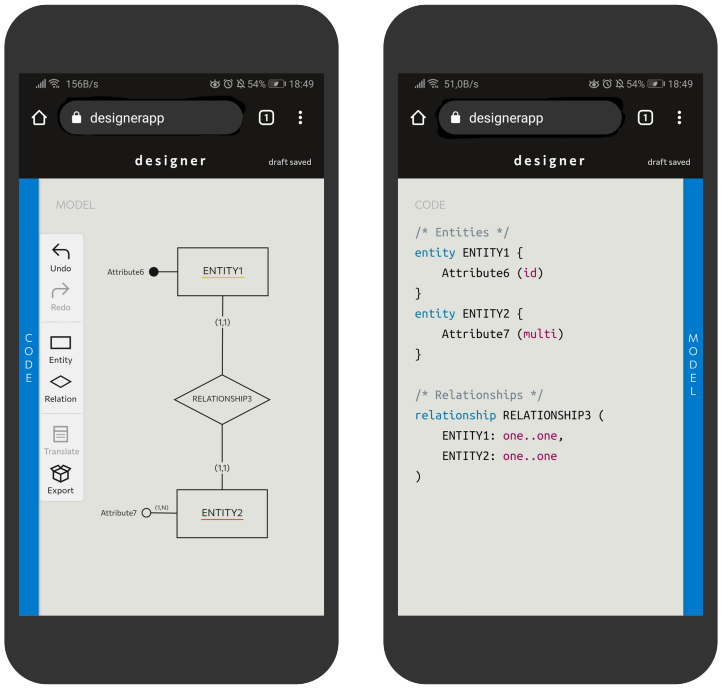

Figure 5.4: How the Designer app adapts to fit a mobile viewport.

Note that responsive Web design is not a separate technology – it is a term used to describe an approach to Web design or a set of best practices, used to create a layout that can respond to the characteristics of the device being used to view the content. Modern CSS layout methods are inherently responsive, and new features have been built into the Web platform to make designing responsive applications easier. Techniques like *media queries*, *flexible grids*, and *flexbox* are central in Responsive Web Design.

Media Queries

Media queries enable developers to run a series of tests (e.g. whether the user's screen is greater than a certain width, or has a certain resolution) and apply CSS selectively to style the page appropriately for the user's needs. It is possible to add multiple media queries within a stylesheet, tweaking the whole application layout or parts of it to best suit the various screen sizes. The points at which a media query is introduced, and the layout changed, are known as *breakpoints*.

For example, the following media query tests to see if the current web page is being displayed as screen media (therefore not a printed document) and the viewport is at least 600 pixels wide. The CSS for the `.box` selector will only be applied if these two things are true:

```
@media screen and (min-width: 600px) {
    .box {
        width: 500px;
        margin: 0.5em 1em;
    }
}
```

Flexible Grids

Rather than changing their layout between breakpoints, responsive applications can be built on flexible grids. A flexible grid means that there is no need to target every possible existing device size and build a pixel-perfect layout for it, as that approach would be unfeasible. By using a flexible grid, breakpoints and changes in the design only need to happen when the content starts to look bad. For instance, if the line lengths become unreadably long as the screen size increases, or a box becomes squashed with two words on each line as it narrows.

For example, if the target column size is 120 pixels, and the context (or container) it is in is 960 pixels, the percentage of the column width with respect to the container width is calculated to get a value that can be used in the CSS:

```
.col {
    width: 12.5%;
}
```

Flexbox

In flexbox, flex items will shrink and distribute space between the items according to the space in their container, as their initial behavior. By changing the values for `flex-grow`

and `flex-shrink` one can indicate how the items should behave when they encounter more or less space around them.

To have items (elements with the `.item` selector) take an equal amount of space in a container, it is possible to use the `flex: 1` shorthand:

```
.container {
    display: flex;
}
.item {
    flex: 1;
}
```

Flexbox is a powerful modern method that allows building a simple responsive layout without the need to specify percentage values for column sizes.

5.1.4 Editor Navigation

The editor, which displays the rendered model, is obviously interactive: objects can be selected, modified, and moved around. Even if the editor panel is constrained by the window size, reducing the drawing space to that area only would have been a significant limitation. For this reason, we added controls to navigate the editor. Both with the mouse in a desktop client and with fingers in a mobile/tablet environment, it is possible to interact with the editor to zoom and pan.

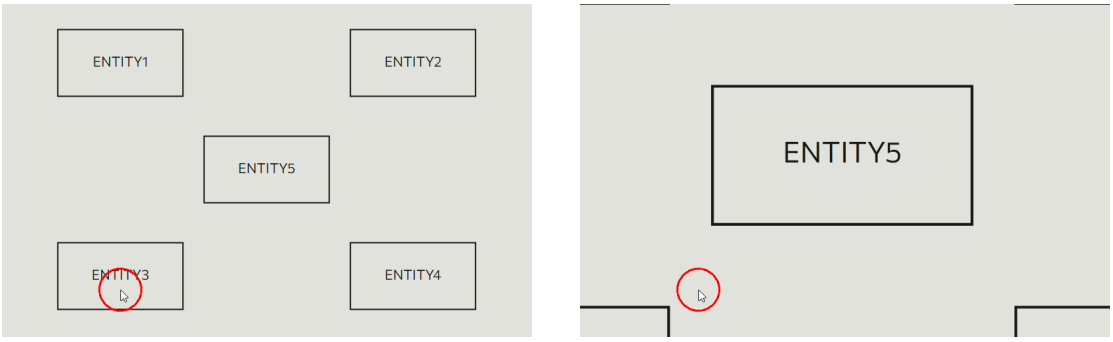

Figure 5.5: Mouse position is not taken into account when zooming.

Implementing the pan function is trivial, as it can be easily accomplished by a geometric translation (or offset) of the model in the drawing space. Conversely, there are two main ways to deliver zoom functionality. One is straightforward to implement but gives mediocre results (Figure 5.5), while the other is slightly more sophisticated to program but makes the interaction feel way more natural and intuitive (Figure 5.6). The easier path is to merely apply a scale transform in response to the user-generated event, without taking into account the mouse/finger position. The more precise solution, the one

used in our software, is to zoom using the mouse pointer (or the centroid of the fingers positions) as the pivot of the scale transform. As a matter of fact, both a translation and a scaling are applied to achieve this effect.

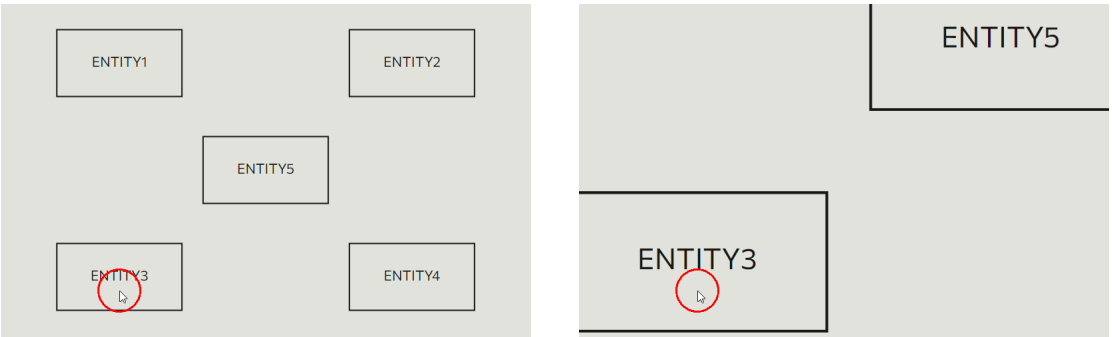

Figure 5.6: The zoom action enlarges the area pointed by the mouse.

5.2 User Experience

User Experience is how a person interacts with and experiences a product, system, or service. It includes the user's perceptions of utility, ease of use, and efficiency. Plenty of guidelines and best practices have been refined over time to give users a great experience. The following paragraphs shed light upon a few key elements that make Designer really stand out.

5.2.1 Errors and Suggestions

Since the software is primarily intended for educational purposes, setting up an effective and meaningful system for showing errors and suggestions was paramount. This is a very prominent aspect of the tool's ability to assist users in designing a database.

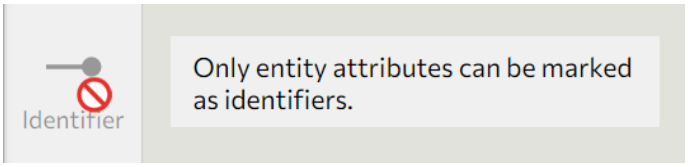

Figure 5.7: An explanation is given every time an action is not allowed.

Figure 5.7 shows a concrete example of how one can learn theory concepts by simply using the software. Whenever an action from the ones available in the toolbar cannot be performed, the button is disabled and the cursor switches to the "forbidden" symbol. But most importantly, a message is displayed next to the button with a brief explanation of why that precise action is not feasible or makes no sense. An alternative would be to

let the user click the button and then display a popup with an error message. However, we believe that our approach is more user-friendly, more impactful, and less frustrating.

When an error condition affects a specific schema item, the error is shown directly in the diagram editor. The user can learn more about the issue by selecting the problematic item, and an error bar will appear on the bottom part of the interface with a more detailed explanation. Chapter 6 dives deeper into potential errors that may happen in the design process and how to fix them.

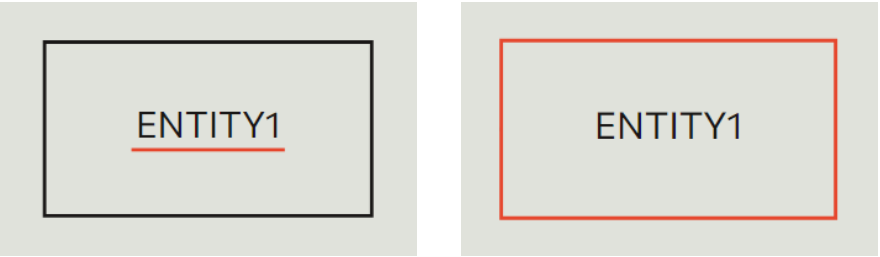

Figure 5.8: Different graphics are used to represent different kinds of error conditions.

Figure 5.8 depicts how different representations are used in different situations: while the entity on the left has a problem, the entity on the right is itself the problem.

5.2.2 Hide Unnecessary Information

It is crucial to keep users focused on the important information and functions. Since our mental focus is finite, unnecessary elements should be removed from the UI or deemphasized. The fact that a piece of software is packed full of features does not mean that the user should be overwhelmed and confused by being shown all of them together. In Designer, we applied this concept on several occasions.

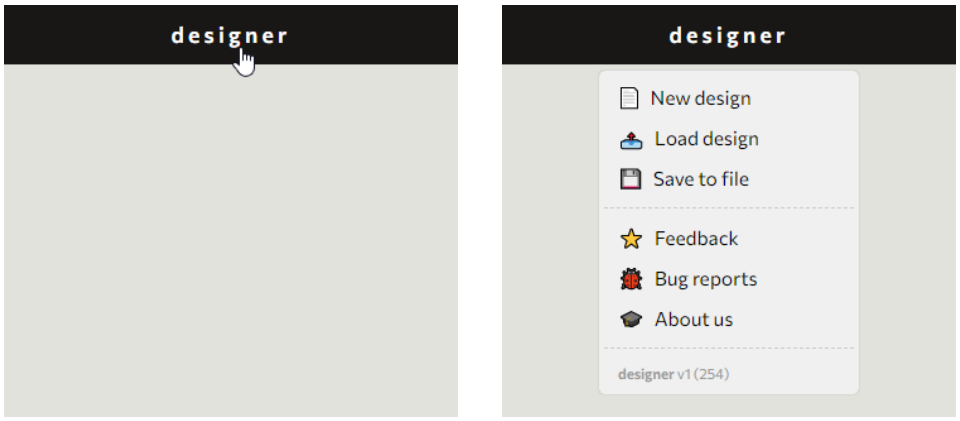

Figure 5.9: The main menu is shown by clicking on the textual logo.

For a start, the application menu is not constantly displayed on the interface. Instead, it is hidden by default and can be revealed by clicking on the textual app logo (as shown in Figure 5.9). The menu contains functions for creating a new design, loading one from disk, or saving the current one to a file, as well as other extra functions for giving feedback, reporting bugs, and visiting the DBDMG website. These functions are estimated to be used no more than five or ten times in a typical working day, as opposed to more design-related features (the ones exposed in the toolbar), which are expected to be used hundreds of times. This is the rationale behind the decision of concealing the main menu while keeping the toolbar always visible and ready.

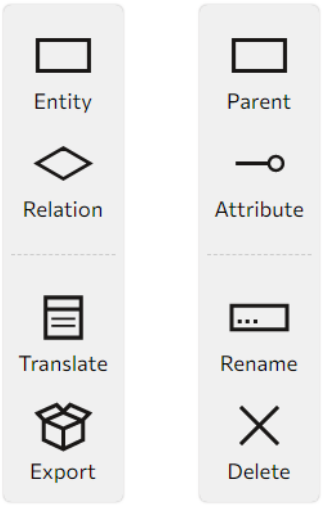

Figure 5.10: Available toolbar functions change depending on the situation.

The toolbar itself is another great example of application of the "hide unnecessary information" principle. Overall, the database design process in our software employs a total of about forty functions. Besides being a relevant space management issue, showing that many buttons all at once can be disorienting. Consequently, the toolbar was designed in such a way that it only shows the software functions relevant to the current design step and to the selected item. An example of this concept is portrayed in Figure 5.10, which shows a portion of the toolbar under two different conditions. On the left, the one we encounter in the conceptual design step when no item is selected: it contains buttons to create entities and relationships, to proceed with the translation, and to export the ER schema. On the right, the toolbar shown in the conceptual design step when an entity item is currently selected: it features buttons to create a generalization, to add an attribute, and to rename or delete the selected entity.

5.2.3 Real-Time Updates

By taking advantage of Vue's reactivity features, our software is capable of updating in real-time all interface components, in response to changes that happen in the model. The code panel, for example, does not need to be generated by activating a certain function. Instead, all changes in the design model are instantly reflected in the code.

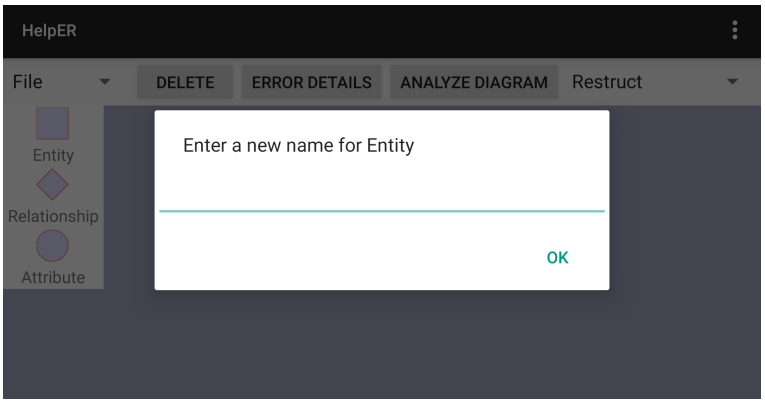

Figure 5.11: Popup dialogs fail to provide a great user experience.

The renaming of an item in Designer is a great example of real-time updates in action. The usual popup dialogs (Figure 5.11) have several disadvantages: they hide a large portion of the workspace, preventing users from looking at the bigger picture; one must click the *Save* or *OK* button in order to see changes applied; and any errors, inconsistencies, or conflicts can only be displayed when the popup for renaming is closed.

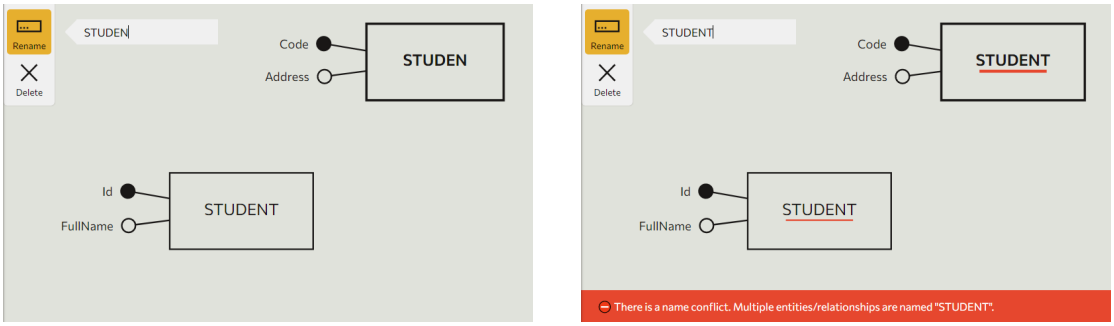

Figure 5.12: Renaming is not obtrusive and allows to immediately spot conflict errors.

Instead of using popup dialogs, the *Rename* button in the toolbar shows a textbox "in place" in a way that the schema remains fully visible. Any issues occurring due to the renaming action can be spot right away and corrected on-the-fly. Figure 5.12 illustrates a STUDENT entity and another entity being renamed in the same way. Without leaving the rename control, the user can figure out the mistake and fix it.

5.3 Schema Drawing

An important part of the development effort was put into the creation of a high-level schema drawing library. This library is built upon Two.js and uses its lower-level functions to render complex diagrams on screen. Several such libraries are already available, but none of them felt appropriate for our standards and for the fresh look we wanted to give to our software.

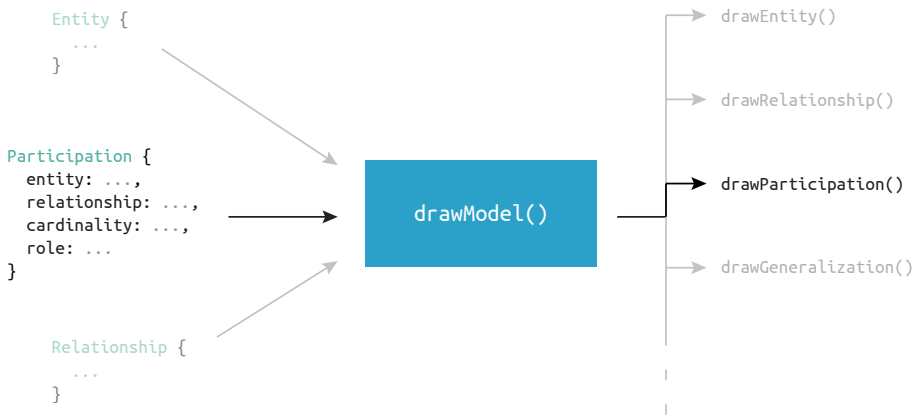

Figure 5.13: Operation of the `drawModel()` function.

Everything starts with the `drawModel()` function. It has the job of populating the scene with all the objects in the model and maintaining them. To provide separation of concerns and to keep the logic open for extension, the `drawModel()` function does not contain any knowledge on how to render schema elements graphically. Instead, as illustrated in Figure 5.13, each element type simply defines the way it should be drawn, and the model-drawing logic only takes care of calling these methods.

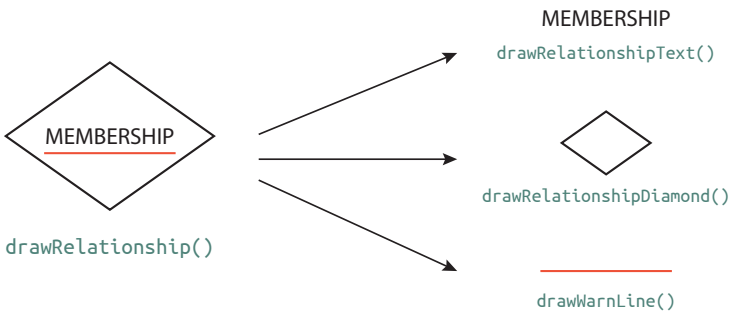

Figure 5.14: The schema drawing library abstracts low-level drawing details.

The different `draw()` functions include detailed instructions for turning each construct specification into a `Two.Group` that contains all the appropriate basic graphical elements needed for that particular schema construct. Figure 5.14 shows what is inside the `drawRelationship()` function, used to graphically represent a `Relationship` object: a `drawRelationshipText()` function to display the name of the relationship, a `drawRelationshipDiamond()` function to draw its characteristic diamond shape, and a `drawWarnLine()` function to show a red/yellow line in case of errors/warnings.

5.4 Utility Features

Besides all software functionalities devoted to schema drawing and model restructuring and translation, we also deployed some features to support portability, durability and to allow sharing. Among these, functions to load design files from the user's device or to save them, and functions for exporting diagrams as images.

5.4.1 Load and Save

Figure 5.15: The saved file contains details of the design in JSON format.

As we will see in Section 5.5.2, our software is equipped with an autosave functionality that automatically stores the current progress of the design in the browser. Even so, there is still the need to allow the user to save the design file that is currently being

edited. This is the only way to share one's work with other people, but is also useful for backup purposes. The design file contents are in plain text and hold the complete application state in JSON format (Figure 5.15).

Similarly, a feature for users to load a design file from disk and resume working where they left off was developed. Both actions are available in the main menu of the application (Figure 5.9). As soon as a design is loaded from a file, its contents will replace the current work and the autosave routine will start using the new file as "draft": every new change will be stored locally in the browser, according to the mechanism explained in the relative section.

5.4.2 Exporting the Diagram

Even though taking a screenshot of the editor is always a viable option, a feature for exporting the diagram as an image was deemed necessary. It allows saving the model being edited as a transparent PNG image, for use in presentations and articles (Figure 5.16).

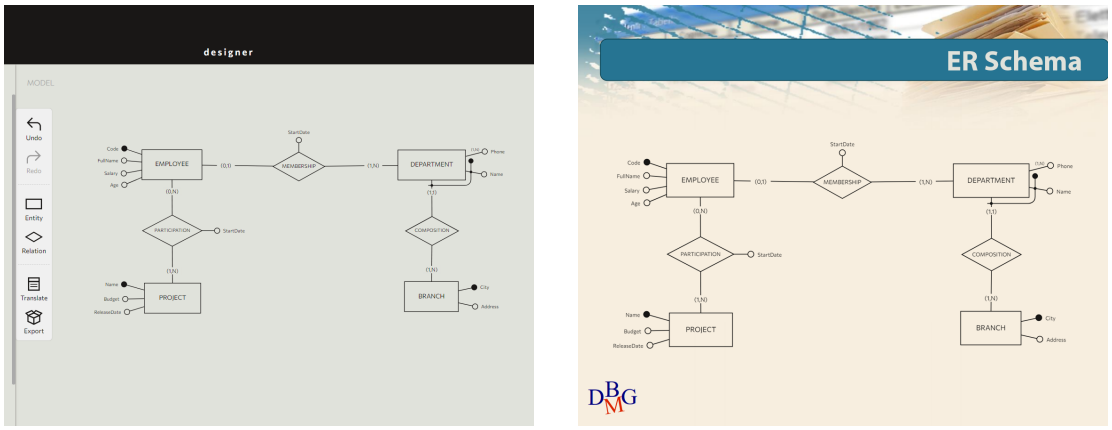

Figure 5.16: Exporting the diagram for use in lecture slides.

The export functionality is available when drawing both the Entity-Relationship model and the logical schema, and can be activated by clicking the *Export* button in the toolbar (Figure 5.17). The image is then generated on the fly and downloaded to the user's device. Any errors present in the diagram are not included in the final image.

We took advantage of the Canvas API (Section 4.2.2) and the Two.js drawing library (Section 4.4) in order to deploy this feature. The schema is drawn to a canvas, whose contents are then converted to a `Blob` object and downloaded as `image/png`.

Figure 5.17: The Export button in the toolbar.

5.5 Enhancing Usability

The set of features that will be covered in this Section are not strictly related to database design. Though not explicitly required, they were implemented for the sole purpose of enhancing usability and making the software more desirable to the final user.

5.5.1 Undo and Redo

The ability to undo an operation in a computer program was independently invented multiple times, in response to how people used computers. It is an interaction technique currently implemented in the majority of software applications. The *undo* command erases the last change done to the document, reverting it to an older state. On the other hand, the *redo* command restores changes that have been reverted using the undo command (Figure 5.18). With the possibility of undo, users can explore and work without fear of making mistakes, because every action can easily be undone [20].

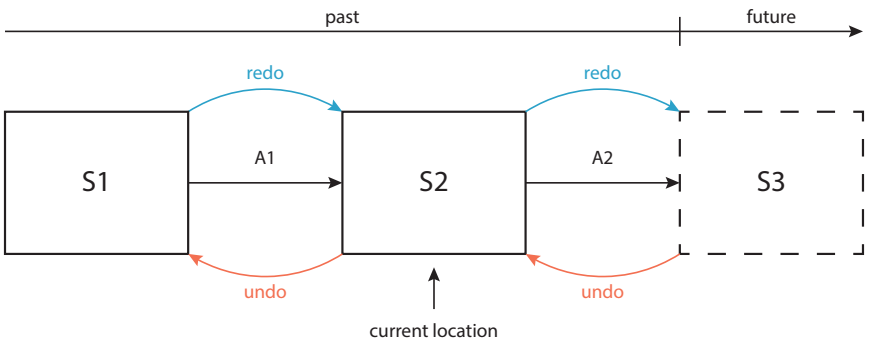

Figure 5.18: An overview of how the Undo/Redo mechanism works.

Usually undo is available until the user undoes all executed operations, or until a certain number of operations has been undone (this number is commonly known as *history buffer size*, or simply *history size*). In addition, there are some actions that are not stored in the undo buffer, and thus they cannot be undone. Such actions include file saving, navigation (pan and zoom), selecting and deselecting elements.

Two categories of undo models exist: linear and non-linear. The undo model adopted in our project is a restricted linear model. In particular, it is:

- *linear*, because it is implemented with a stack (LIFO structure) that stores the history of all executed commands. When a new command is executed it is added on top of the stack. Hence only the last executed command can be undone and removed from the history. Undo can be repeated as long as the history is not empty.
- *restricted*, because the history list size is limited. That is, when a defined size is reached, the first executed command is deleted from the list.

Undo can be implemented through different patterns. The most common patterns are the *Command Pattern* and the *Memento Pattern*. In our project, we use a very trivial implementation of the second option.

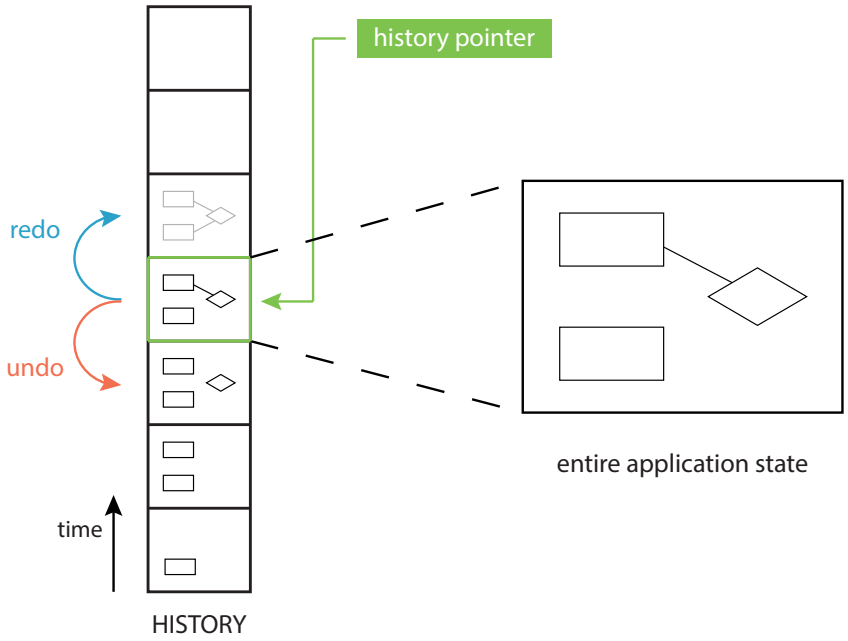

Figure 5.19: The Undo/Redo feature is implemented using a history array.

In a nutshell, an array emulates the file history, and undo and redo commands allow the user to move around. Regarding the array contents, the *full checkpoint* strategy is adopted: after each command is executed, the complete state is saved. This implementation, although not highly efficient, is straightforward and is still a good choice in the case of a limited history size.

5.5.2 Autosave

As explained previously, users have the possibility to save their work for later reuse or for sharing with other people. Nonetheless, a feature was included in the application which automatically saves the progress in the database design (commonly referred to as "draft", as shown in Figure 5.20). This technique is widely used in the vast majority of current web-based editing tools and helps to reduce the risk or impact of data loss in case of a crash or freeze.

Figure 5.20: *Left:* There are unsaved changes. *Right:* The design draft is saved.

This feature makes use of the Web Storage API technology described in Section 4.2.1. The entire state of the application at a given moment is serialized in JSON format and stored using the `localStorage` mechanism, in order to persist design data even after the browser is closed. Each time the application is launched, it looks for the presence of a design draft locally and – if it is there – restores it, allowing the user to resume work seamlessly.

Some applications save the user's progress at regular time intervals, while others do it after an action is performed or a task is completed. In our software, all changes to the design are saved continuously. Having such kind of autosave function in place removes the need for saving design files entirely, except for backup or sharing purposes. Clearly, this implementation poses problems in terms of application performance: user actions triggering an autosave may occur many times per second and this computational overload may eventually cause the application to be unresponsive or to freeze.

To avoid this kind of problem, we use a technique called *debouncing*. In essence, the debounce function delays processing of an event for a certain amount of time and, if another event is fired during this delay, the old one is discarded (and the delay timer is reset). The way that debounce operates is summarized in Figure 5.21. In the beginning, Event `a` happens and it is delayed. Then Event `b` happens and, while waiting for the delay time to go by, Event `c` happens. This causes the reset of the delay timer, so Event `c` *overrides* Event `b`. Finally, Event `d` happens and it is delayed normally, since no other events happen in the meantime. In our case, circles at the top represent changes to the design document, and circles at the bottom represent activations of the autosave function.

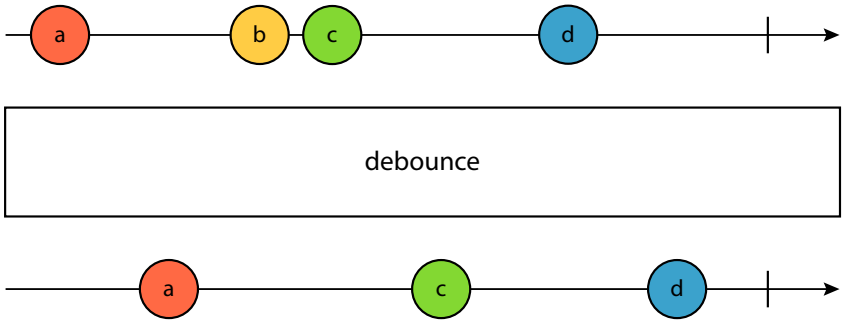

Figure 5.21: Example of operation of the debounce function.

This same mechanism is used in Web pages that have autocomplete search boxes, where not every keystroke results in an API request. As a rule of thumb, one should wrap any interaction that triggers excessive calculations or API calls with a debounce.

5.5.3 Offline Availability

Platform-specific apps show some content and provide some sort of limited interaction even when no network connection is available. It might not be anything particularly meaningful, and the user could even be unable to achieve what he or she wanted to achieve, but at least one gets the feeling that the app is in control. In contrast, on the Web, traditionally nothing happens when the browser is offline. Most likely an error message appears reading "There is no Internet connection", and there is the chance an offline dino game is available to fill the time (Figure 5.22).

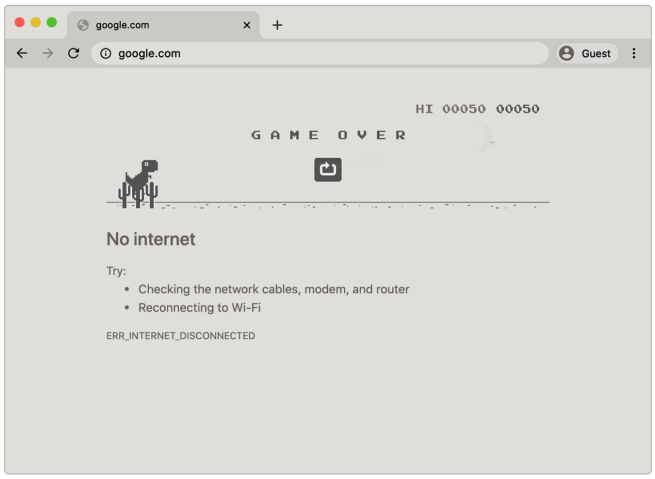

Figure 5.22: On the Web, not much can be done without an Internet connection.

As described in Section 4.1.3, Progressive Web Applications are network independent: they are decoupled from the backend and they work offline in the same way as they do online. Data is stored in the frontend and synchronized with the server whenever possible. In other words, the lack of connectivity is not treated as an error, but only as a temporary situation where the user is still able to work with the application. Web Applications that require an Internet connection to work may show a simple branded page with the information that the user is currently offline, and here there is no limit to creativity.

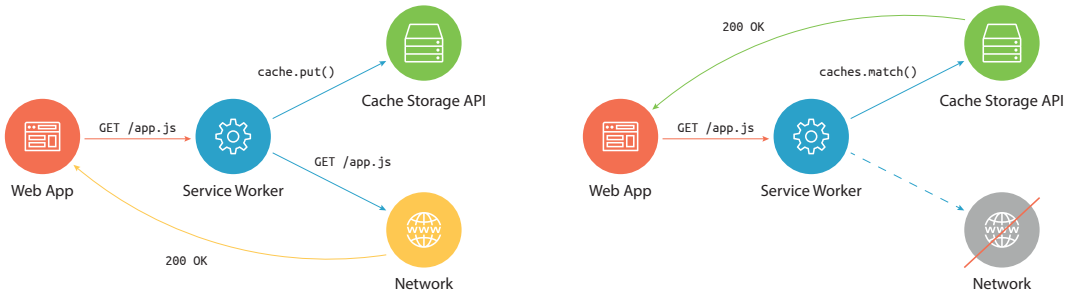

Figure 5.23: Enabling technologies: Service Workers + Cache Storage API.

On the technical side, this is made possible through the use of Service Workers and the Cache Storage API (Figure 5.23). Service Workers are a virtual proxy between the browser and the network. They run on a separate thread from the main JavaScript code of the page and do not have any access to the DOM structure. Since Service Workers are very powerful, they can only be executed in secure contexts (that is, through HTTPS).

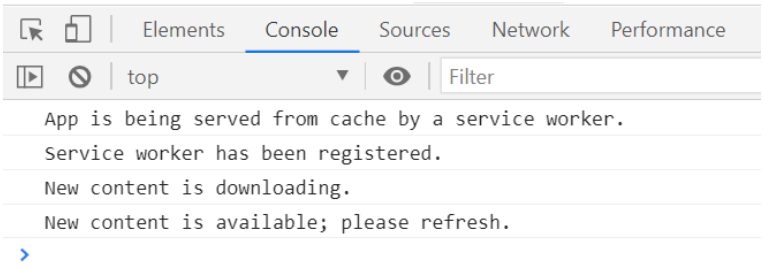

Figure 5.24: The Service Worker checks if new content is available and updates the cache.

Here is how it works: the Service Worker is registered by the application, and this means it is automatically downloaded, then installed, and finally activated. Upon installation, the static app files – HTML, CSS, JavaScript, and image files – are cached using the Cache Storage API. The Service Worker then listens for `fetch` events (which fire every time an HTTP request is made by the app) and is able to intercept requests and respond to them with custom responses. This allows us to serve content from the cache instead of

the network, and therefore users can access the app also when no Internet connection is available. When a new version of the app is released, a new service worker is installed in the background which will then add all application files (including the new static assets) to a new cache (Figure 5.24). The updated Service Worker is then activated and takes over management of the page from the old one.

Usually, the most recent Web Application frameworks do all the work themselves thanks to special plugins. Vue.js is no exception and uses the `@vue/cli-plugin-pwa` package to take care of all the PWA-related duties, including the registration of a Service Worker that will precache the site's local assets.

5.5.4 Keyboard Shortcuts

As we have seen, the design is built by interacting with the editor and by using the application toolbar. To speed up the operation of the software for desktop users, some actions have an associated keyboard shortcut. JavaScript offers two event handlers that make this feature easy to implement, which are the `keyup` and `keydown` events. The `keydown` event is fired when a key is pressed and the `keyup` event is fired when a key is released. Some keyboard shortcuts in Designer are `Ctrl+Z` to undo, `Ctrl+Y` to redo, `Canc/Delete` to delete an item.

In our case, shortcuts are desired to be caught within a particular area of the page (namely, the editor) rather than across the entire document, so listeners for the `keydown` and `keyup` events should be set on the root DOM element of that area. Event listeners on an element catch all events within that element, including events fired from children elements. When listening for events, it is often useful to call the `preventDefault()` function to prevent the default actions from happening when they are not desired.

Chapter 6

Tool Overview

After having discussed the technologies at the core of this software, and after having explained the ideas and principles that inspired its design, and after having shown the main components of its interface, we now proceed to illustrate its functionality. Rather than selecting a particular scenario and performing the whole database design process using that as a reference, we prefer to showcase the software functionality on smaller and more refined use cases.

6.1 Drawing the ER Model

When the application starts up, it is ready for conceptual design. If no design draft is saved locally, the editor appears empty and the user can start building the design without needing to configure or set anything. One usually draws basic constructs first, and then proceeds with more advanced idioms.

The rest of this section shows how to use the software to design a complete, full-grown Entity-Relationship schema from scratch. For each construct, we will provide an explanation on how to create it, how to modify its properties, what errors could arise, and how to avoid them.

6.1.1 Entities

Entities can be created in the diagram through the use of the *Entity* button in the toolbar (Figure 6.1). Once this tool is selected, a semitransparent entity rectangle will appear behind the mouse cursor (on desktop devices only) and will follow its movements. At this point, the user should move the cursor to the desired position in the schema where the newly-created entity should appear.

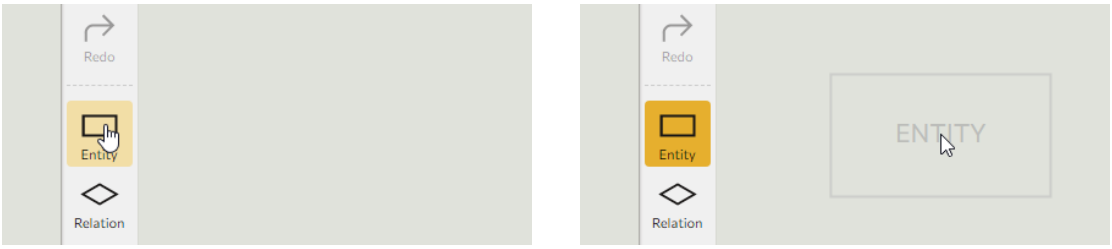

Figure 6.1: Entities can be drawn by selecting the Entity button.

Finally, with a mouse click or a touch on the screen, the entity is added to the diagram. It gets assigned a default name, that is, the word *ENTITY* followed by the object identifier (a progressive integer number).

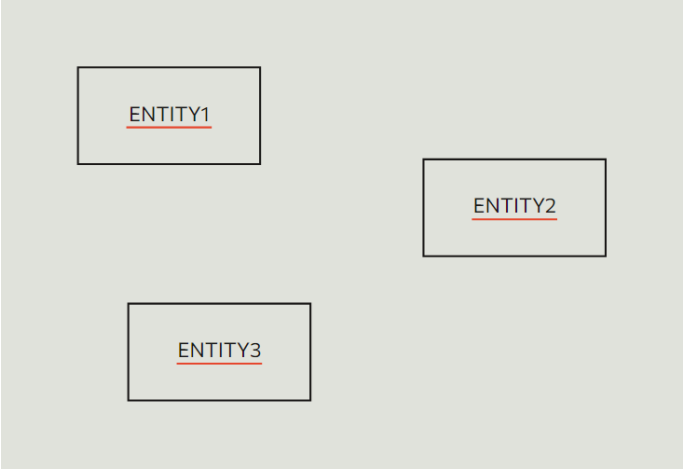

Figure 6.2: Entities have been created, but they seem to have a problem.

As soon as an entity is created, a red line appears below its name (Figure 6.2). That is the way that our software indicates that there is a problem with that element (as we will see, the same happens with other constructs as well). To figure out what the problem is, it is enough to select the entity by simply clicking on it (or by touching it).

Figure 6.3 shows two things. The first one is that items appear in bold when they are selected, to be properly differentiated from the rest of the elements. The second one is that when we select an item that has some problems (either errors or warnings) the software displays an explanatory message giving more details on the issue.

Of course, errors and warnings are two very different things. While warnings are nothing more than suggestions or best practices, errors are out-and-out flaws in the design that prevent the user from proceeding to the next steps and that should be corrected immediately.

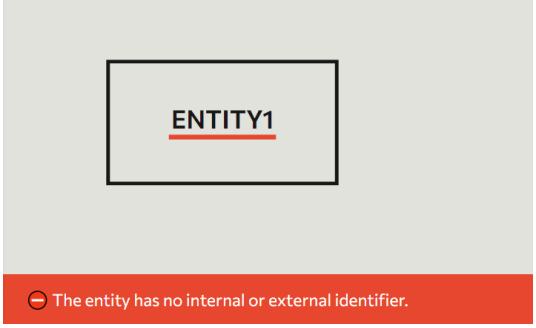

Figure 6.3: A detailed error description appears when the item is selected.

When dealing with entities, one may encounter the following errors:

- **"There is a name conflict. Multiple entities/relationships are named 'X'."**
  Entities must have unique names in the schema. To fix this error, the user can give a different name to one conflicting entity/relationship.
- **"The entity has multiple attributes with the same name."**
  Attributes of an entity must have unique names. This issue can be solved by renaming one of the conflicting attributes.
- **"The entity has no internal or external identifier."**
  An entity must necessarily have an identifier, either internal or external. One option is to mark one or more entity attributes as identifiers, thereby creating a key for the entity. Another option is to create an external identifier by including other entities in the identification (see Section 2.1.4 for more details).
- **"A child entity must not have any internal or external identifier."**
  The only exception to the rule that an entity must have an identifier is when the entity is a child entity in a generalization. Since they inherit the attributes of their parent, and the parent entity necessarily has an identifier, they automatically have one too. Thus, they do not require – in fact, cannot have – their own identifier explicitly defined.

On the other hand, an entity may show the following warnings:

- **"The entity has no attributes (apart from identifiers)."**
  An entity whose all attributes are identifiers is completely fine from the theoretical point of view. However, it is not much use in the real world and it would be better if some more attributes were added to the entity.

Some actions are available to different kinds of constructs, but we will only discuss them here: they are the *Rename* action and the *Delete* action (Figure 6.4). These buttons

appear in the toolbar whenever an item that supports the relative actions is selected.

Figure 6.4: The Rename and Delete buttons in the toolbar.

When the user clicks the *Rename* button, an input field appears that allows entering the new name for the selected item. Only alphanumeric characters and the underscore symbol (_) are allowed (even though a name cannot begin with a digit). Whitespaces are automatically converted to underscores. In the case of entities and relationships, the name is forced to be uppercase.

Figure 6.5: Renaming an item.

The *Delete* button lives up to its name and removes from the diagram the selected item. To avoid ending up with a cluttered schema, deleting an item may entail the removal of other items that depend on it. Indeed, it should be no surprise that deleting an entity causes its attributes to be disposed of as well, or that deleting a relationship results in the elimination of related participations.

6.1.2 Relationships

It is possible to add relationships to the conceptual schema by means of the *Relationship* button in the toolbar (Figure 6.6). Similarly to what happens with entities, a semitransparent diamond appears behind the mouse cursor, indicating that a relationship is about to be created. Again, with a mouse click (or a touch) the relationship is drawn in the desired location.

Deciding whether to start designing a schema with entities or relationships is left to the discretion of the user. The software does not enforce any particular order: one can draw

all entities first, all relationships first, or mix the two things.

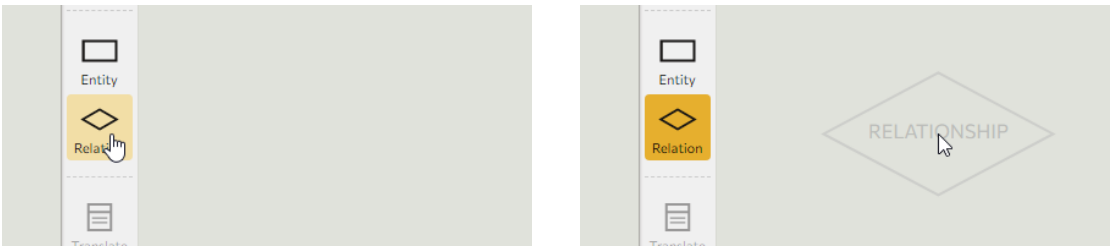

Figure 6.6: Relationships can be drawn using the Relationship button.

Once the schema contains both entities and relationships, it is possible to start connecting these two types of constructs and have entities *participate* in relationships. To do that, we select the relationship and then click the *Connect* button (Figure 6.7).

Figure 6.7: The Connect button appears when a relationship is selected.

After selecting the *Connect* tool, a message appears saying that the user should pick an entity to participate in the selected relationship (as illustrated in Figure 6.8).

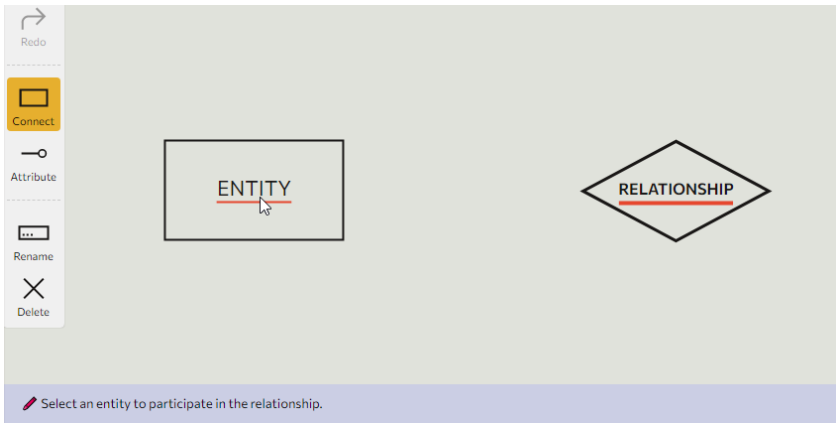

Figure 6.8: An entity should be selected to create a participation.

Once the desired entity is clicked or touched (Figure 6.9), a participation is created between this entity and the relationship from which the *Connect* tool was activated. The participation is considered to be an item in and of itself, that can be for example selected and deleted.

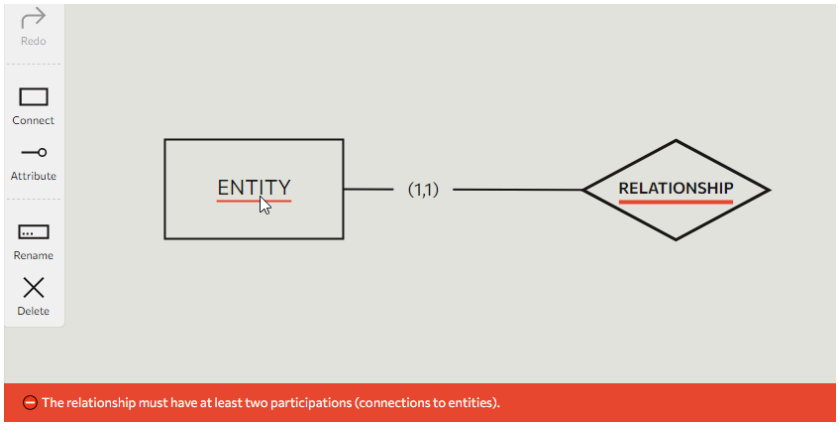

Figure 6.9: Selecting an entity (when the Connect tool is active) creates a participation.

A participation has some properties that can be modified, namely the cardinality and the role. These can be accessed and edited from the application toolbar when a participation item is selected. The *Cardinality* button opens a menu (Figure 6.10) that allows choosing the participation cardinality among four possible options (with minimum cardinality equal to zero or one, and maximum equal to one or many).

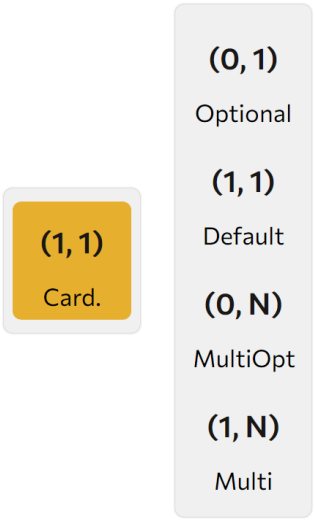

Figure 6.10: The menu for specifying the cardinality of a participation.

Editing the role of a participation (using the button in Figure 6.11) is very similar to renaming an item, with the exception that the role can be an empty string. Specifying the role for a participation is optional, with the exception of recursive relationships.

To create a recursive relationship, it is enough to connect the same entity to the relation-

Figure 6.11: The Role button is used to set a participation's role.

ship twice. In that case, the participation lines are drawn in a slightly different way, as depicted in Figure 6.12. Should any of the two participation items be deleted, the way participation lines are drawn would go back to the normal representation.

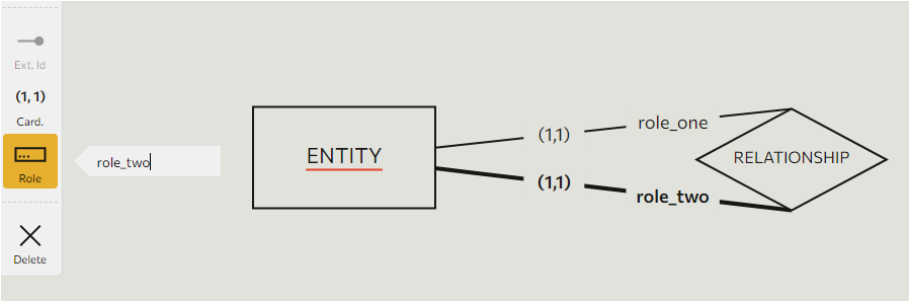

Figure 6.12: Participation lines are drawn differently in recursive relationships.

When working with relationships, it is possible to observe the following errors:

- **"There is a name conflict. Multiple entities/relationships are named 'X'."**
  Relationships must have unique names in the schema. To fix this error, rename one of the conflicting entities/relationships.
- **"The relationship must have at least two participations."**
  It does not make sense for a relationship to have one participation only. This error can be fixed by connecting the relationship to more than just one entity. In the case of recursive relationships, one participation is still not enough and the relationship should be connected *twice* with the entity.
- **"The relationship has multiple attributes with the same name."**
  Attributes of a relationship must have unique names. This issue can be solved by giving one of the conflicting attributes a different name.
- **"A relationship involved in external identification cannot be recursive."**
  It would not be a proper external identification without an outside entity. That is why recursive relationships (those of an entity with itself) and external identification are not compatible with each other and give rise to this error. To overcome the

problem, one can either give up on external identification or make the relationship not recursive.

- **"A recursive relationship requires participation roles to be specified."**
  Participation roles are always optional, except for recursive relationships. In that case, roles must be specified since they are needed in the translation step.
- **"A ternary relationship requires all participation cardinalities to have maximum value N."**
  This is a requirement of ternary relationships. The issue is easily fixed by setting the maximum cardinality to N in all the participations.
- **"A relationship involved in external identification cannot have any attributes."**
  Attributes are not allowed in relationships that serve for external identification. To solve this problem, the attributes of the relationship should simply be removed.

6.1.3 Attributes

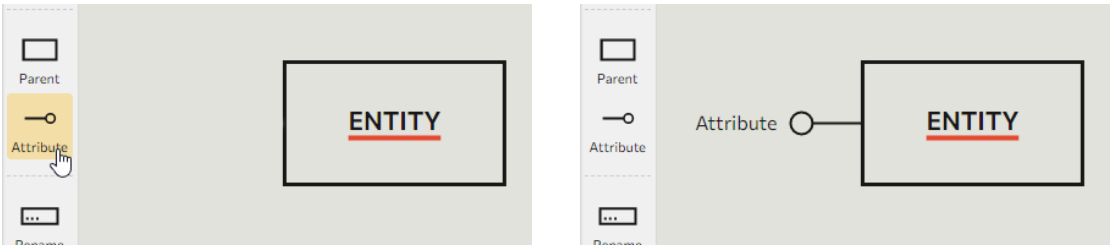

Figure 6.13: Attributes are created from an entity or a relationship.

Unlike entities and relationships, attributes are not independent, thus they cannot exist as standalone items. An attribute either refers to an entity or to a relationship and is created from the parent item, by means of the *Attribute* button (Figure 6.13). The attribute is then created in the proximity of the item it refers to, and is arranged in a way that does not obstruct or cover other items of the schema.

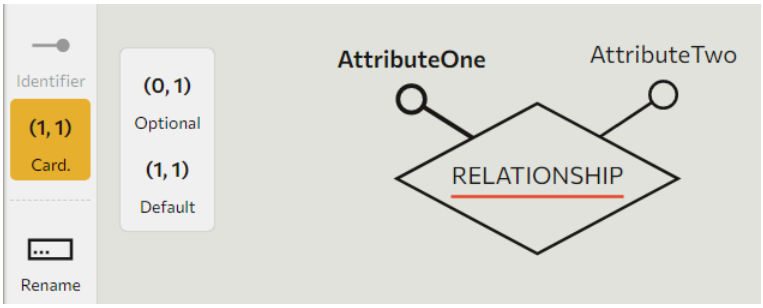

Figure 6.14: Relationship attributes cannot be multivalued.

Similar to what happens with participations, attributes can have a cardinality specified as well. An attribute is optional when its minimum cardinality is zero, and it is multivalued when its maximum cardinality is N. Modifying the attribute cardinality is done using a menu that is cut from the same cloth as the one used for participations (Figure 6.10). However, as shown in Figure 6.14, relationship attributes cannot be multivalued, so their cardinality value can only be one of two options: (0,1) or (1,1).

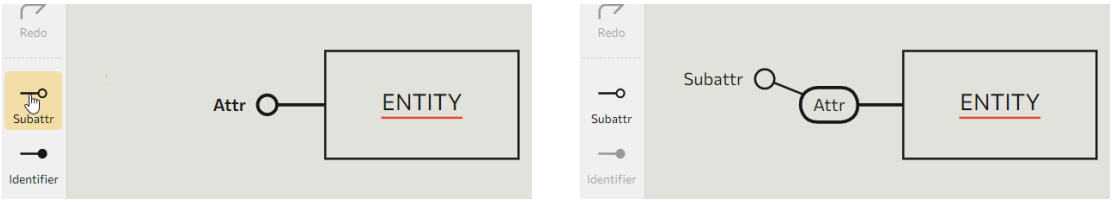

Figure 6.15: An entity attribute can be turned into a composite attribute.

In Section 2.1.3, we have seen that it can sometimes be convenient to group attributes of the same entity that have closely connected meanings or uses and that the set of attributes obtained in this fashion is called a composite attribute. Figure 6.15 illustrates how to create a composite attribute using Designer: it is enough to select an attribute of an entity and click the *Subattr* button in the toolbar (Figure 6.16).

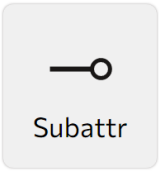

Figure 6.16: The Subattr button allows creating composite attributes.

The following errors may arise when dealing with attributes:

- **"The attribute has multiple subattributes with the same name."**
  This error can only occur with composite attributes and is fixed by giving one of the conflicting attributes a different name.

6.1.4 Identifiers

Identifiers must be specified for each entity in the schema, with the only exception of child entities in generalizations. Section 2.1.4 describes identifiers thoroughly and claims that they can be divided into internal identifiers and external identifiers. We will address them separately in the following paragraphs.

Internal Identifiers

Internal identifiers are the ones formed by one or more attributes of the entity itself (this kind of identifier is commonly referred to as key). To create such an identifier, the *Identifier* button is used (Figure 6.17).

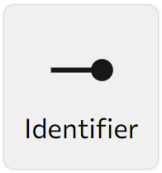

Figure 6.17: The Identifier button toggles the identifier property.

The button works as a toggle switch: it turns normal attributes into identifier attributes and the other way around. Figure 6.18 shows how it is enough to mark one or more entity attributes as identifier in order to create a key for the entity (and how the "no identifier specified" error goes away).

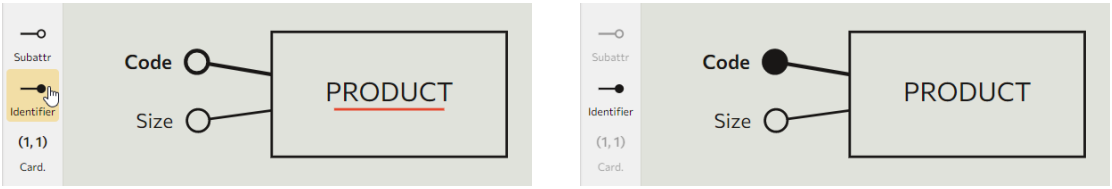

Figure 6.18: Creation of an entity key by marking one of its attributes as identifier.

When two or more attributes are marked as identifiers, their graphical representation is slightly different, as illustrated in Figure 6.19. The attribute circles do not appear filled anymore, but instead a curve goes through all of them – with small dots at the intersections – and is closed with a filled circle at one end.

Figure 6.19: Graphical representation of multiple-attribute internal identifiers.

External Identifiers

When the attributes of an entity are not sufficient to identify its occurrences unambiguously, we resort to external identifiers. These can involve one or more entities associated with the entity being identified through a relationship, and the entity being identified must participate in such a relationship with cardinality (1,1).

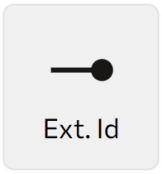

Figure 6.20: The External Identifier button in the application toolbar.

To put an external identifier in place, we first have to select the participation that connects the entity to identify with the relationship with the entity used for external identification. At that point, by clicking on the button in Figure 6.20, we obtain the external identifier we were looking for. Other participations may be added to the external identification, as well as any attributes of the entity being identified.

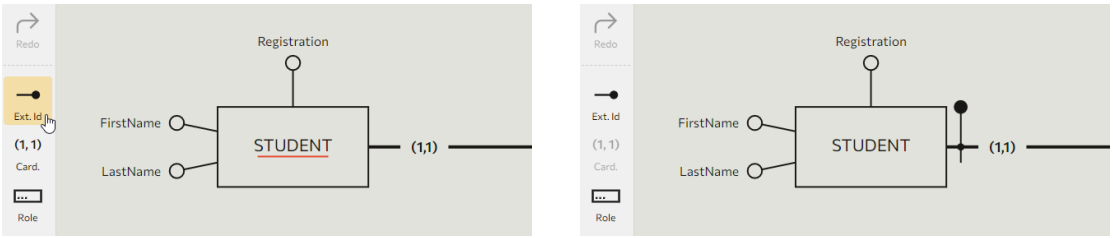

Figure 6.21: Creation of an external identifier.

Let us take once more the schema in Figure 2.10 as a reference. The STUDENT entity has to be identified externally, so we select the participation with (1,1) cardinality that connects it to the ENROLLMENT relationship and we click the *External Identifier* button (Figure 6.21).

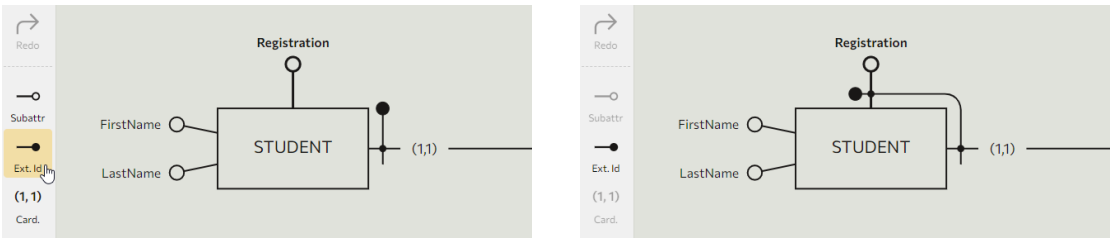

Figure 6.22: Entity attributes may be included in external identification.

Any entity attributes that contribute to the external identification (as in the case of the `Registration` attribute of entity STUDENT) are included in a similar fashion, as illustrated in Figure 6.22.

6.1.5 Generalizations

Generalizations are logical links between entities of the schema. Setting them up in Designer is straightforward, using the *Parent* button (Figure 6.23).

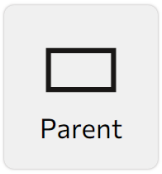

Figure 6.23: Generalizations are created using the Parent button.

Since generalizations only concern entity items, we need to select an entity to make the *Parent* button appear in the toolbar. Once the tool is activated, a message tells the user to pick an entity to be the parent of the current entity (Figure 6.24).

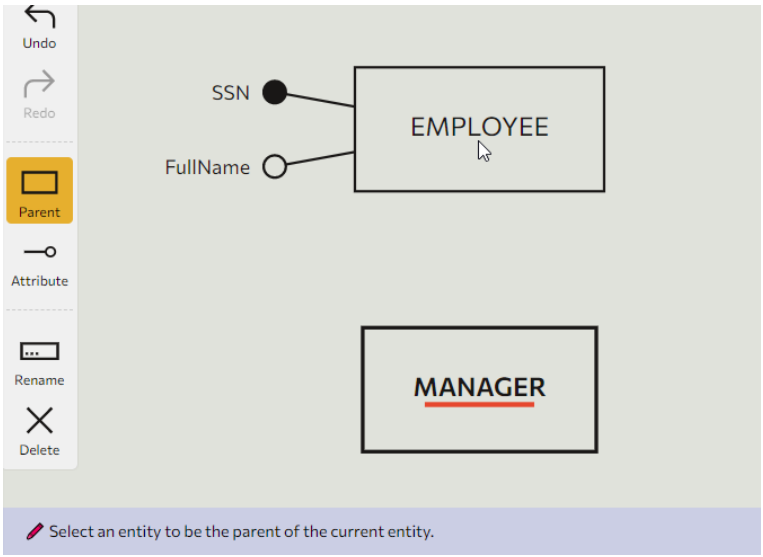

Figure 6.24: To produce a generalization item, a parent entity should be designated.

As soon as an entity is clicked or touched, it becomes the parent entity in the generalization, while the entity that was selected when the *Parent* tool was activated becomes the child entity in the generalization (Figure 6.25). A generalization may exist between a parent entity and multiple child entities. To do that, it is enough to repeat the process

above for each child entity: we select the child, activate the *Parent* tool, and click on the parent entity.

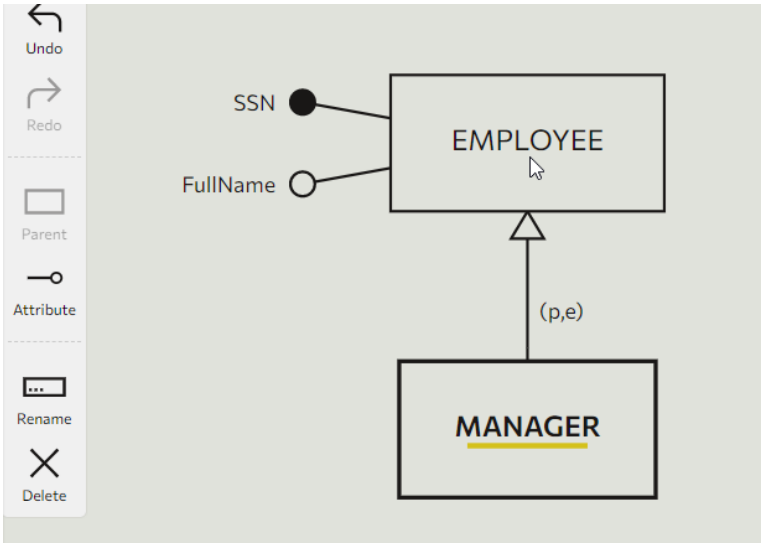

Figure 6.25: To produce a generalization item, a parent entity should be indicated.

The generalization is itself an item, composed of smaller subitems: the generalization parent (the arrow), and multiple generalization children (lines connecting child entities to the generalization parent item). While deleting the generalization parent item completely removes the generalization from the schema, deleting a generalization child item only removes the corresponding child entity from the generalization.

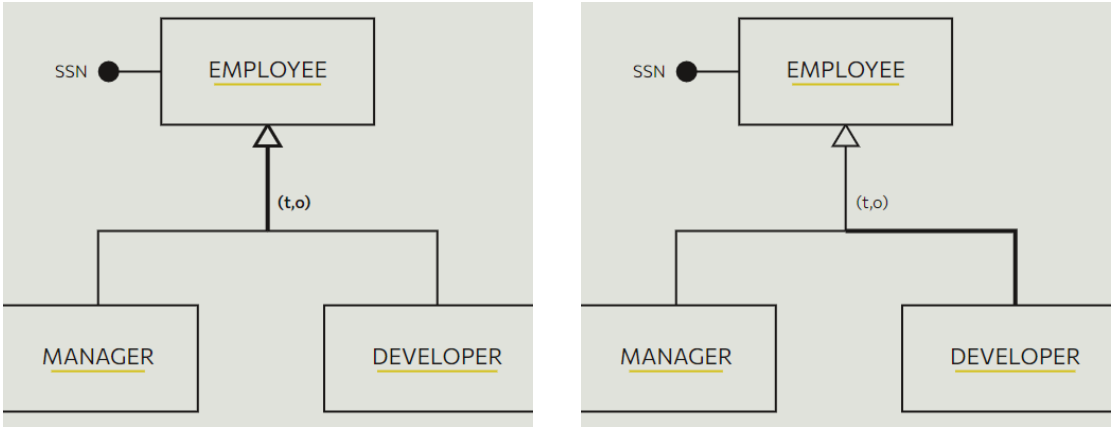

Figure 6.26: The generalization parent item (*left*) and a generalization child item (*right*).

A generalization is categorized by default as partial and exclusive, but this property can be easily modified using the *Generalization Type* button. A menu will appear (Figure

6.27), where one can choose among four possible combinations: partial or total, and exclusive or overlapping.

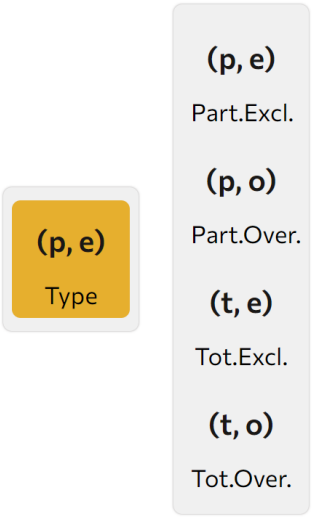

Figure 6.27: The menu for specifying the type of a generalization.

6.1.6 ER Code Generation

The code section in the conceptual design step reflects the model in the editor, as shown in Figure 6.28. As there was no standard language specification to describe an Entity-Relationship schema, we did come up with a possible language ourselves. The code is divided into three parts: entities, relationships, and generalizations.

As mentioned in Section 5.1.2, the code features syntax highlighting using the third-party JavaScript library Prism. The ER language was of our own invention, so Prism could not offer off-the-shelf support for it. Thankfully, the library allows defining new languages or extending existing ones. This is achieved by defining – using patterns known as *regular expressions* – the new language's keywords, tags, comment styles, etc.

For example, generalizations can be defined in the ER language by means of the following syntax:

```
<ParentEntityName> <= {
    <ChildEntityName>
    {, <ChildEntityName>}
} (partial|total, exclusive|overlapping)
```

```
CODE

/* Entities */
entity STAFF_MEMBER {
    SSN (id),
    FirstName,
    LastName
}
entity DOCTOR {
    Qualification (multi)
}
entity VOLUNTEER {
    Organization (optional)
}
entity DEPARTMENT {
    Name (id),
    Phone
}

/* Relationships */
relationship MEMBERSHIP (
    STAFF_MEMBER: one..one,
    DEPARTMENT: one..many
)

/* Generalizations */
STAFF_MEMBER <= {
    DOCTOR,
    VOLUNTEER
} (total, exclusive)
```

Figure 6.28: ER code automatically generated during conceptual design.

6.2 Restructuring Step

Once the conceptual schema is complete, one can proceed with the logical design phase, which includes the restructuring step and the translation step. This is achieved by clicking on the *Translate* button shown in Figure 6.29: the software will automatically detect if a restructuring action is necessary for the current ER diagram.

If this is the case, the user is presented with a screen very similar to the one used for conceptual design. The only two differences are the absence of the ER code panel and the fact that now some constructs are colored in red, as illustrated in Figure 6.30. This is the restructuring step, and the highlighted items are the ones that need our attention.

As we know from Section 3.1, model restructuring is about removing generalizations,

Figure 6.29: The Translate button in the application toolbar.

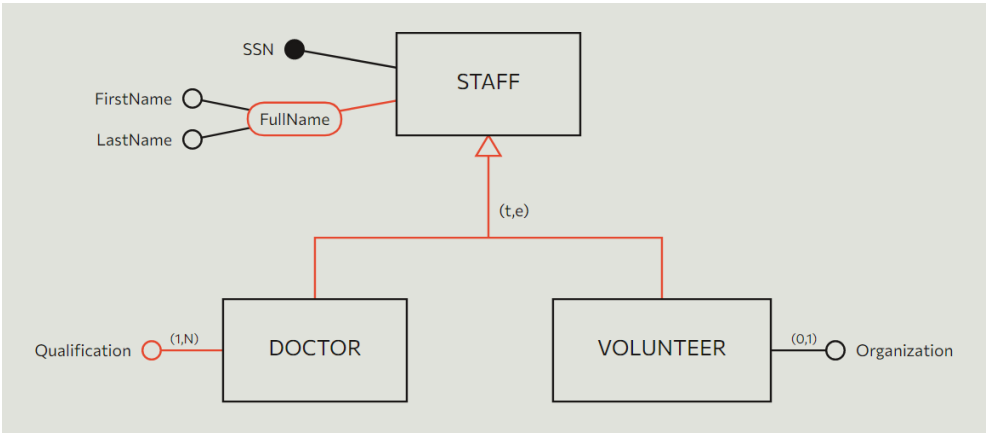

Figure 6.30: Items highlighted in red need to be restructured.

multivalued attributes, and composite attributes. The user is requested to start by restructuring generalizations first, before continuing with the two special kinds of attributes (that do not require to be processed in any particular order). All operations are executed using the *Fix* button in Figure 6.31, and later we will see how this same button is also used in the translation step.

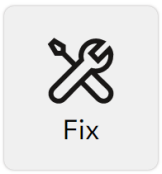

Figure 6.31: Restructuring and translation are done using the Fix button.

6.2.1 Generalizations

The constructs that must be restructured first are generalizations. This step makes no distinction between the generalization parent item and the generalization child items (unlike the conceptual design phase), and selecting the generalization arrow or one of its lines to child entities will select the entire generalization item (Figure 6.32).

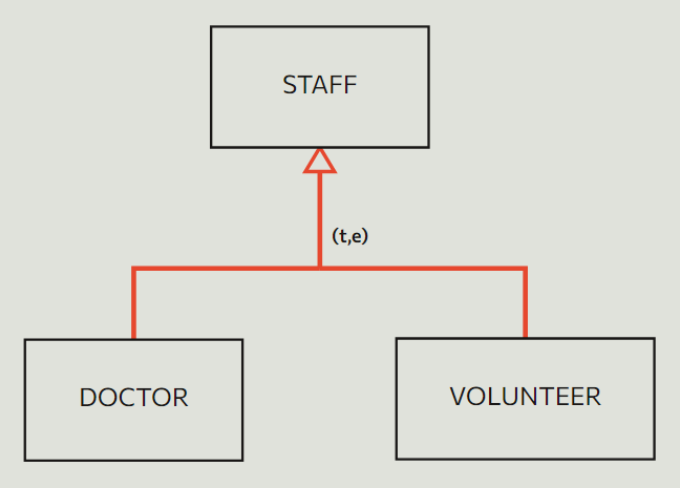

Figure 6.32: The generalization item is selected and must be restructured.

When a generalization item is selected and the *Fix* button is clicked, the menu reported in Figure 6.33 is shown to the user. The restructuring methods available are: collapsing the child entities to the parent entity, collapsing the parent entity into the child entities, or substituting the generalization with parent-child relationships.

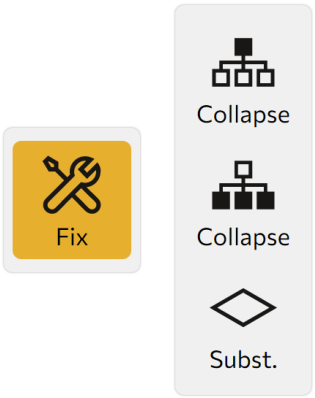

Figure 6.33: The three available options for fixing generalizations.

Recall from Section 3.1.1 that the choice of collapsing the parent entity into the child entities is only allowed when the generalization is total and exclusive.

6.2.2 Multivalued Attributes

Attributes with maximum cardinality equal to N have no equivalent representation in the logical schema. For this reason, they need to be replaced with a new entity connected

to the attribute's parent entity through a relationship (this matter was discussed in more detail in Section 3.1.2).

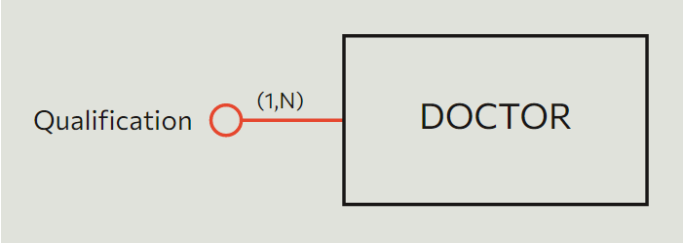

Figure 6.34: Multivalued attributes cannot be directly translated.

These attributes are substituted by choosing one of the two alternatives in the menu of Figure 6.35: the *Unique* button makes the new entity participate in the relationship with a (1,1) cardinality, while the *Shared* button assigns a (1,N) cardinality instead.

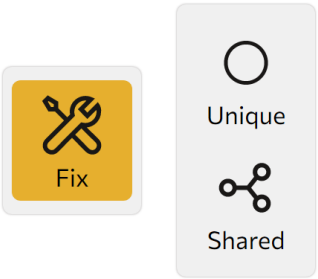

Figure 6.35: The two ways of restructuring a multivalued attribute.

6.2.3 Composite Attributes

Composite attributes are among the constructs that cannot be directly translated, and therefore need to be restructured. A composite attribute is highlighted in red in the restructuring step, as depicted in Figure 6.36.

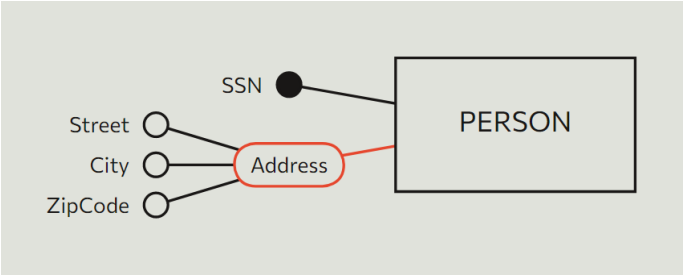

Figure 6.36: Composite attributes require restructuring.

From Section 2.1.3, we know this kind of construct has two possible restructurings: one where each subattribute becomes an attribute on its own, and another where all subattributes are merged to form a single attribute. When a composite attribute is selected, and the *Fix* button is clicked, the user is presented with the two options (Figure 6.37).

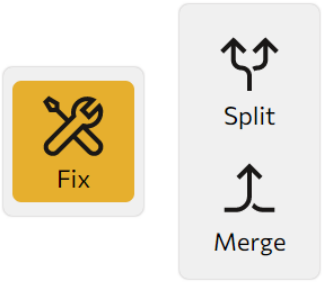

Figure 6.37: The two ways of restructuring a composite attribute.

The outcomes of the restructuring activity concerning composite attributes have been presented in Section 3.1.3.

6.3 Translation Step

Upon completion of the restructuring phase, or if the conceptual schema is such that no restructuring is needed, the ER schema is to be translated into a logical schema. When the user enters this phase, the model only consists of entities, relationships, and attributes. Translation starts from strong entities, proceeds with weak entities (those that are identified externally), and is concluded with relationships. The application prescribes this particular order of operation to help the less experienced to better understand what goes on behind the scenes.

6.3.1 Entities

The translation of entities, both the ones identified internally and those identified externally, happens in a completely automated manner. It is enough to select the entity and click the *Fix* button, and the software will do the rest. For clarity purposes, we decided to let the user control this step even if no choices about the translation have to be made. This avoids overwhelming users with many changes occurring all at once and helps them learn more effectively.

Figure 6.38 illustrates an entity before translation and the corresponding table obtained after clicking on the *Fix* button. The software autonomously figures out what the columns of the new table will be, its primary key, as well as any nullable columns and external references.

Figure 6.38: There are no choices to be made when translating entities.

While in the process of translating entities and relationships, the diagram is neither a conceptual schema nor a logical one. Tables may appear connected to each other or to entities by means of relationships, even if that makes little sense and has no counterpart in the literature. Note that this is to be intended as an intermediate step, whose sole function is to produce a logical schema. Our idea is that the advantage of empowering users with the ability to see first-hand the effects of the translation activity greatly overcomes the benefits of the unyielding pursuit of academic rigor.

6.3.2 Relationships

Relationships translate in many different ways depending on their cardinality, as described in Section 3.2.2. Hence, the *Fix* button in the toolbar does not exhibit a uniform behavior. If only one option is available, the relationship is translated directly. Otherwise, a menu with all the possible solutions is shown.

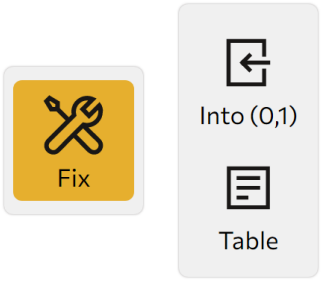

Figure 6.39: Possible translations of (0,1) - (*,N) relationships.

For example, in the case of a one-to-many binary relationship with optional participation, the two options displayed in Figure 6.39 are given to the user. The first one, named *Into (0,1)*, suggests that the (0,1) entity incorporates the relationship. The second one, called *Table*, indicates that the relationship is translated into a new table.

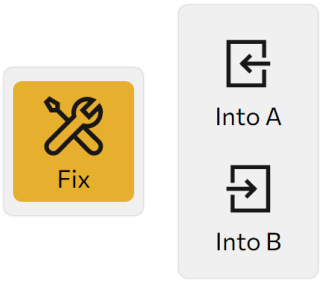

Figure 6.40: Possible translations of (1,1) - (1,1) relationships.

Figure 6.40 shows a different situation. When the relationship is one-to-one with mandatory participation for both entities, it is absorbed by one of the two entities at the discretion of the user. That is why the *Fix* button shows a menu with two options named *Into A* and *Into B*.

6.4 Logical Schema Editing

What is obtained as a result of restructuring and translation of the conceptual schema is a genuine, complete logical schema. However, the software offers some fine-tuning capabilities oriented towards physical design, but still agnostic to any specific choice of database management system. The user can specify actual data types of table columns, reorder fields as desired, define unique constraints, and export SQL code.

6.4.1 Assigning Data Types

Given that the logical schema editing step produces DBMS-ready SQL code, it is necessary to assign a data type to each table column. This is done by clicking the *Data Type* button, which shows the dropdown box illustrated in Figure 6.41.

Figure 6.41: Assignment of a data type to a column.

While it is true that each DBMS makes available its own set of data types, the list provided in our software is the most generic and standard. The aim is to offer, to the fullest extent possible, cross-platform SQL code generation.

6.4.2 Reordering Columns

The order in which columns appear in a table is decided arbitrarily by the software during the translation step. The user may change such order using the *Move* button (Figure 6.42), as long as columns belonging to the primary key of the table are always positioned above all other columns.

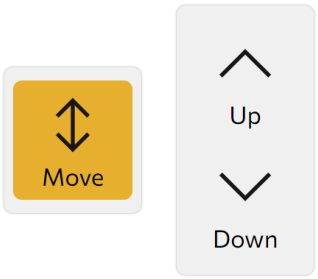

Figure 6.42: Table columns can be reordered with the Move button.

This function comes in handy in cases where the attributes of an entity or relationship – arranged meticulously in the conceptual schema – are then discombobulated when translating the parent item.

6.4.3 Unique Constraint

The software gives the option to create single-column unique constraints by means of the *Unique* button (Figure 6.43). For obvious reasons, this option is not available to the columns that form the primary key of a table.

Figure 6.43: The Unique button in the toolbar.

The *Unique* button works as an on/off switch, in a similar way to the *Identifier* and *External Identifier* buttons: it marks normal columns as unique and transforms unique columns back to normal.

6.4.4 SQL Code Generation

The code panel that was showing ER code in the conceptual design phase reappears in the logical schema editing step. This time it contains standard SQL code for porting the

database schema to a DBMS, In particular, it includes DDL statements for the definition of the database tables, with their columns, primary keys, unique constraints, and foreign key constraints. Figure 6.44 shows an example of the contents of the SQL code panel.

```sql
SQL

/* Tables */
CREATE TABLE EMPLOYEE (
    Code VARCHAR(8) NOT NULL,
    FullName VARCHAR(30) NOT NULL,
    Age INTEGER NOT NULL,
    Salary INTEGER NOT NULL,
    Department VARCHAR(60),
    PRIMARY KEY (Code),
    FOREIGN KEY (Department) REFERENCES DEPARTMENT(Name)
);
CREATE TABLE DEPARTMENT (
    Name VARCHAR(60) NOT NULL,
    Phone VARCHAR(16) NOT NULL,
    PRIMARY KEY (Name)
);
CREATE TABLE PROJECT (
    Name VARCHAR(30) NOT NULL,
    Budget INTEGER NOT NULL,
    ReleaseDate DATE NOT NULL,
    PRIMARY KEY (Name)
);
CREATE TABLE PARTICIPATION (
    Employee VARCHAR(8) NOT NULL,
    Project VARCHAR(30) NOT NULL,
    StartDate DATE NOT NULL,
    PRIMARY KEY (Employee, Project),
    FOREIGN KEY (Employee) REFERENCES EMPLOYEE(Code),
    FOREIGN KEY (Project) REFERENCES PROJECT(Name)
);
```

Figure 6.44: SQL code automatically generated during logical schema editing.

6.5 An Example Use Case

Thus far, the tool overview has been focusing on the individual functionalities as if they were independent of each other. In this section, we take advantage of all the features presented previously in this chapter and apply them to a real-world example. We will be dealing with a database design exercise very similar to the ones that students may be facing in an undergraduate course in databases. For the sake of brevity, we do not explain the process in detail but we only report the final resulting conceptual and logical schemas.

6.5.1 Exercise Text

The AirQ company monitors and analyzes the air quality in cities and wants to design a database to manage its activities.

- AirQ analyzes the concentration of various pollutants such as carbon monoxide and nitrogen dioxide. Each pollutant is identified by a code and characterized by its name, a brief description, the measurement unit, and a list of the main causes of the production of the pollutant (e.g. vehicular traffic, heating systems, industrial activity).
- AirQ employees are identified by their social security number (SSN). For each employee, the name, the hire date, the mobile phone number, and the e-mail address (if available) are known. Employees are classified as administrative staff, technical staff, and analysts. For analysts, the qualification is recorded.
- A network of fixed stations is exploited to monitor the concentration of pollutants. Each station is characterized by a unique code and its geographical position, expressed in terms of latitude and longitude. Each station includes different sensors. Each sensor is identified by a unique code and characterized by the station where it is placed and the monitored pollutant.
- The sensor measurements are stored in the database. Each measurement has a value and is identified by the sensor, the date, and the time when it was collected. Each month a report summarizing measured concentrations for various pollutants is generated. Each report is identified by a unique code and is characterized by the release date and the list of measurements included in the report itself. Each measurement is used in at most one report. For each report, the analyst who performed the verification and validation of the report is also recorded.
- In case of a station failure, a maintenance operation for the station is carried out. For each maintenance operation, the database stores the date, the start and end time of the intervention, the station involved, the technician who carried out the maintenance, and the issues that caused the failure. Different maintenance operations can be performed for the same station on the same date. However, a technician cannot perform two or more maintenance operations simultaneously.

6.5.2 ER Diagram and Logical Schema

Figure 6.45 depicts the Entity-Relationship diagram describing the conceptual schema of a database for the above application. After performing the restructuring and translation steps, and after a few finishing touches, we obtain the logical schema in Figure 6.46.

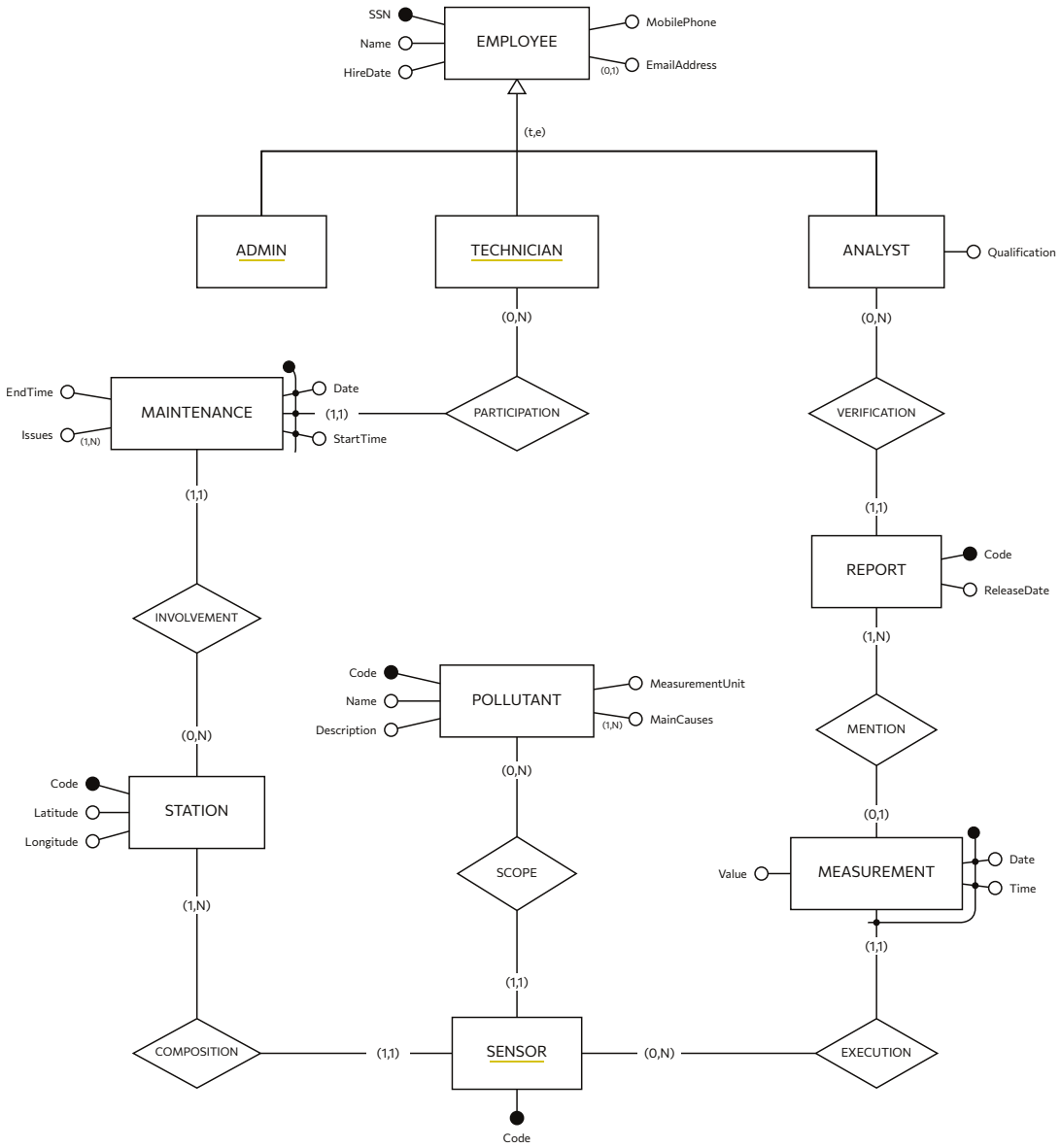

Figure 6.45: The ER diagram obtained using Designer.

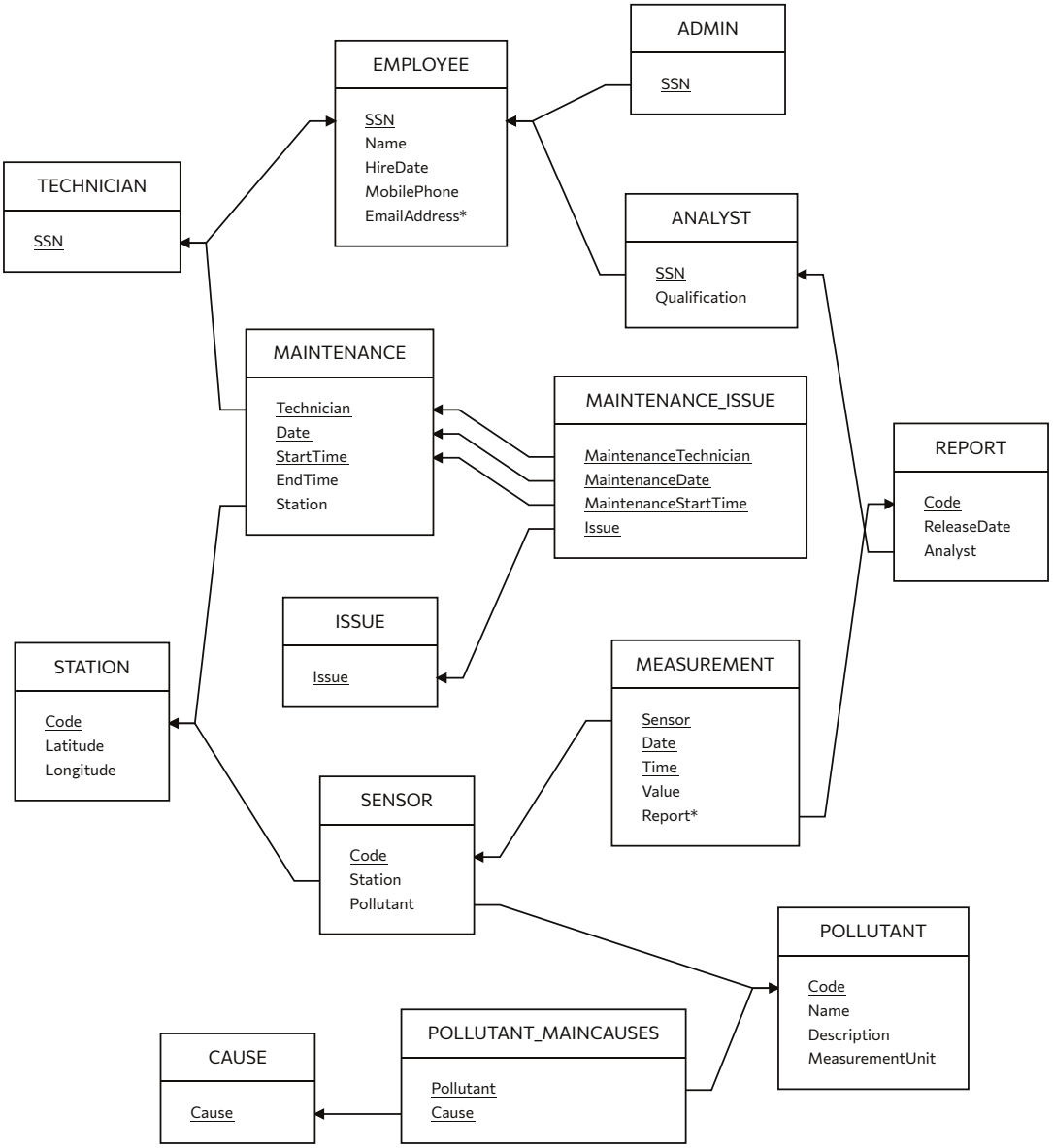

Figure 6.46: The derived relational logical schema.

Chapter 7

Conclusions

7.1 Summing Up

The database design process demands a great deal of effort and requires many delicate decisions to be taken. Building a software application for computer-aided database design means first of all to limit as much as possible the burden on users, both newcomers and experienced ones. The tool developed in this thesis work focuses entirely on this task, giving strong emphasis to specific conceptual and logical design features, and providing the most fine-grained functionality.

To the untrained eye, our software may look like a diagram drawing application. On some level, it certainly is. A significant amount of effort was devoted to implementing schema design features on a browser environment, balancing efficiency and appearance. The choice of developing a brand-new schema drawing library has definitely paid off in the long term, as no compromises had to be made at any point. We had complete freedom to decide how constructs would be represented graphically and how they would respond to user interactions. Our drawing subsystem does not target low-level graphics-related browser technologies directly. Instead, it is built upon the formidable Two.js API to take advantage of its cross-browser, renderer-agnostic capabilities. It was crucial to adopt a library that was at the same time easy to use and powerful enough to create stunning and complex graphics.

However, Designer is much more than just a database-design-specific drawing tool. It offers plenty of features to support users throughout the whole design process. Undergraduate students and inexperienced users may find it especially helpful to *learn by doing*. The full de-facto Entity-Relationship schema specification is supported, including all of its constructs – each with its own properties – and graphical representations. In addition, model restructuring and translation are completely covered. Ad hoc functions

have been deployed to deal with constructs that cannot be directly translated, and all possible restructuring options are shown to the user. While the translation step may have been implemented in a declarative way, a more visual alternative was favored: turning entities and relationships into tables only requires a few clicks. Everything happens in front of the user's eyes and it is easy to understand the logic behind each function.

The ultimate goal of this thesis work was to create the best educational tool for conceptual data modeling. What makes the design *computer-aided* is the constant presence of error-checking features, the step-by-step approach to the process, and the comprehensive explanations and suggestions. Putting aside the functional point of view, the creation of a top-quality user experience and a captivating user interface has been at the core of our agenda from day one. We followed up-to-date UX guidelines and adopted several best practices to engage users and make our product lovable and easy to use. The choice of a Web Application was not random: we wanted to make the tool accessible to the widest possible audience, and we did that by supporting a vast range of devices.

7.2 Where To Go From Here

Even though we have achieved the goals we set for ourselves, the product is not at all complete. It has great potential, yet much still needs to be done in terms of functionality, technology infrastructure, and browser support.

7.2.1 Functionality Limitations

The conceptual design phase presents some limitations. For example, the software does not support entities with more than one (internal or external) identifier, nested generalizations, and composite attributes in relationships. Also, in the case of external identification, no checks are made to ensure that the chain does not contain any loops.
Also, the restructuring activity does not currently include: an analysis of redundancies, to delete or retain possible redundancies present in the schema; the partitioning and merging of entities and relationships, to partition concepts in the schema into more than one concept or to merge several separate concepts into a single one; the selection of primary identifiers, to choose an identifier for those entities that have more than one.
In the logical schema editing step we have not implemented – primarily to avoid confusing non-expert users – features to have a column be part of the primary key, to make it optional, or to delete it. While it is true that these can be directly derived from previous steps, including such fine-tuning features would make the software more attractive to advanced users.

7.2.2 New Features

There are countless ways to improve the application by adding new and interesting features. The important thing is to always keep the users' needs in mind.
An idea would be to make the ER code panel *interactive*: that is, not only having changes in the editor being reflected in the code but also the other way around. Changing properties of schema items in the code would simultaneously update the model accordingly. To go even further, the application could allow generating a conceptual design in a completely declarative way, with the user eventually using the editor only to adjust small graphical details.
The logical design could be extended to cover other important aspects, such as the *normalization* procedure. Normalization is a verification tool used to evaluate the quality of a relational database, so as to avoid redundancies and undesirable behavior during update operations.

7.2.3 General Improvements

The application requires the browser of the user to be updated in order to work correctly. If for whatever reason older browsers should be supported, a series of adjustments from the compatibility point of view would be necessary. These would be achieved mainly through the use of *polyfills*, JavaScript codes that provide a certain API wherever the browser does not have it natively.
The overall performance of our application is quite good, but one can always do better. The use of SVG as rendering target is a good choice if the number of drawn items is limited. In our case, we hardly expect this number to go beyond the threshold of one hundred items, so the performance remains at an acceptable level. Should this ever become a usability concern, other rendering alternatives – for instance, Canvas or WebGL – need be considered. Since several advantages of the SVG approach were taken advantage of in the current implementation, some effort would be necessary to obtain similar results with other technologies.

Bibliography

[1] Paolo Atzeni et al. *Database Systems - Concepts, Languages and Architectures*. McGraw-Hill Book Company, 1999. ISBN: 0-07-709500-6.

[2] *ERDPlus*. URL: https://erdplus.com/.

[3] *Microsoft Visio*. URL: https://www.microsoft.com/en/microsoft-365/visio/flowchart-software.

[4] *Android Studio*. URL: https://developer.android.com/studio.

[5] *Mobile OS market share*. URL: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/.

[6] *Apple XCode*. URL: https://developer.apple.com/xcode/.

[7] *Modern Web Applications: An Overview*. URL: http://singlepageappbook.com/goal.html.

[8] *Progressive Web Apps*. URL: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps.

[9] *What Are Progressive Web Apps?* URL: https://web.dev/what-are-pwas.

[10] *Web Storage API*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API.

[11] *Canvas API*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API.

[12] *Touch Events*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Touch_events.

[13] *Vue.js Introduction*. URL: https://v3.vuejs.org/guide/introduction.html.

[14] *Components Basics*. URL: https://v3.vuejs.org/guide/component-basics.html.

[15] *Two.js*. URL: https://two.js.org/.

[16] Wilbert O. Galitz. *The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques*. John Wiley & Sons, Inc., 2007. ISBN: 0470053429.

[17] *The Art of Minimalism in Mobile App UI Design*. URL: https://uxplanet.org/the-art-of-minimalism-in-mobile-app-ui-design-b21aa671dd7f.

[18] *Prism*. URL: https://prismjs.com/.

[19] *Responsive design*. URL: https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Responsive_Design.

[20] *Undo*. URL: https://en.wikipedia.org/wiki/Undo.