Model-Driven Software Engineering Assignment

Model-Driven Software Engineering

Group 2:
Fábio Araújo de Sá (up202007658@up.pt)
Filipe Rodrigues Fonseca (up202003474@up.pt)
Lourenço Alexandre Correia Gonçalves (up202004816@up.pt)
Pedro Pereira Ferreira (up202004986@up.pt)

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

November 2024

Contents

1 Introduction **2**

2 Tools and Technologies **2**

3 Modeling Domain Analysis **3**
3.1 Domain . . . 3
3.1.1 ER Model . . . 3
3.1.2 Relational Model . . . 4
3.1.3 SQL Code . . . 4
3.2 Concepts . . . 5

4 Metamodel Design **5**
4.1 ER Metamodel . . . 6
4.2 Relational Metamodel . . . 6

5 Metamodel Constraints **7**
5.1 Constraints . . . 7
5.2 Validation . . . 8

6 Conclusion **9**

7 Contributions **9**

A Annex **11**

1 Introduction

This project aims to explore techniques and technologies of Model-Driven Software Engineering (MDSE), which can be applied to automate parts of the development of applications on low-code platforms [Brambilla et al., 2017]. These platforms seek to simplify development by allowing the creation of complex solutions without the need for direct programming.

Throughout this project, we explored the different phases of constructing a pipeline that enables the transformation of ER Models into SQL code. To achieve this, we designed and implemented metamodels representing both ER models and Relational models, and these were validated through specific test cases. Afterward, constraints for the model-to-model transformation were created. Finally, a model-to-text transformation was performed to obtain valid and executable SQL code derived from the initial model.

At every stage, the main challenges were identified, and the decisions made were justified.

2 Tools and Technologies

For the implementation of the project, we used the following tools and technologies:

- **Eclipse**[1]: an integrated development environment (IDE) that served as the main platform for development and the integration of other tools;
- **Ecore**[2]: part of the Eclipse Modeling Framework (EMF), which allows the creation, manipulation, and validation of metamodels. It was used to define the formal languages for the source and target models;
- **XMI**[3]: XML Metadata Interchange, is a standard for exchanging metadata information via Extensible Markup Language.
- **Object Constraint Language (OCL)**[4]: used to define integrity constraints on metamodel instances, ensuring that the models adhere to all the necessary rules and consistencies during transformations;
- **OCLinEcore**[5]: is an extension that integrates OCL expressions into Ecore models, allowing for more expressive constraints and operations within models;
- **Atlas Transformation Language (ATL)**[6]: used for applying model-to-model (M2M) transformations, enabling the conversion of more abstract models (such as ER models) to domain-specific models (such as relational models);
- **Acceleo**[7]: a tool for applying model-to-text (M2T) transformations, used to generate code from the final models, allowing, in this case, the generation of valid SQL code.

These tools enabled the automation of the pipeline developed from the proposed models and domains. The use of metamodeling stategies ensured the validation of the models, their transformation, and the appropriate output.

[1]https://www.eclipse.org
[2]https://wiki.eclipse.org/Ecore/
[3]https://www.omg.org/spec/XMI/
[4]https://projects.eclipse.org/projects/modeling.mdt.ocl
[5]https://wiki.eclipse.org/OCL/OCLinEcore
[6]https://eclipse.dev/atl/
[7]https://projects.eclipse.org/projects/modeling.acceleo

3 Modeling Domain Analysis

At this stage, we analyzed the domain to define the source and target models that will be used throughout the project. The analysis of one concrete example helped to delineate the scope of the project, ensuring that our metamodels capture all the concepts covered in the domain and validate each one appropriately.

3.1 Domain

To understand the domain and define the project's boundaries, we analyzed practical one example of two types of models commonly used as the foundation for generating SQL code: Entity-Relationship (ER) models and Relational models. This example guided the development of the metamodels and also served as test cases in the subsequent phases.

3.1.1 ER Model

The Entity-Relationship (ER) model[8][9] is a conceptual approach to data modeling that visually represents the logical structure of an information system and the relationships between the data within that system. Taking the concrete example of a chain of gyms with equipment, where each gym can have multiple customers, and them can have some characteristics in the system, we can derive the ER model as represented in Figure 1:

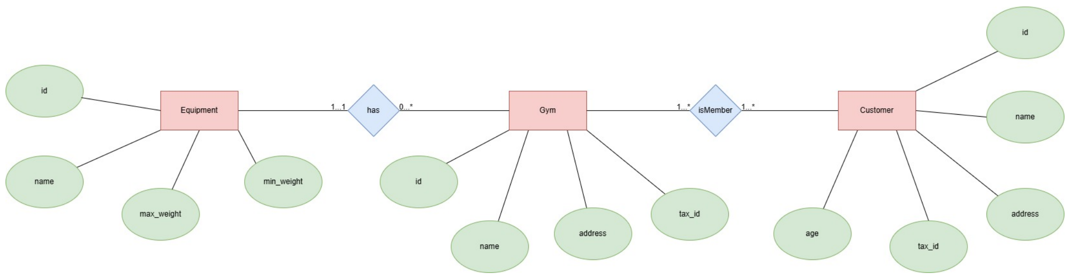

Figure 1: Gym ER Model example

In this example, as well as in similar models, it is essential to ensure the existence of:

- Entities;
- Attributes with associated entities;
- Relationships with various multiplicities (one-to-one, one-to-many, many-to-many) between entities.

For the sake of simplifying future validations, the system will only be designed to handle attributes of type string and integer. Thus, the model is considered valid if it allows the creation of attributes with these types, entities with attributes, and relationships of any cardinality between the entities in the system.

[8]https://en.wikipedia.org/wiki/Entity–relationship-model
[9]https://www.geeksforgeeks.org/introduction-of-er-model/

3.1.2 Relational Model

The Relational Model[10] is a data modeling approach that organizes information into tables, or relations, where each table consists of columns (its attributes) and rows (the actual data). Unlike the Entity-Relationship (ER) model, which is more conceptual and visual, the relational model focuses on the effective implementation of data in Database Management Systems (DBMS), thereby aligning more closely with the ultimate goal of generating SQL code. Using the previous example of the chain of gyms, a translation of the above into a Relational Model can be represented as shown in Figure 2:

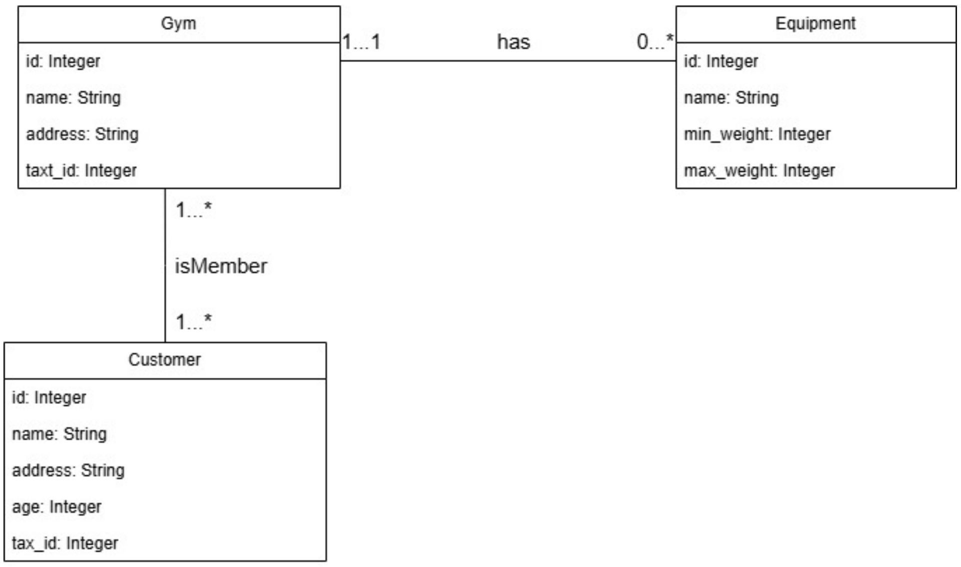

Figure 2: Gym Relational Model example

Thus, the relational models to be manipulated will include:

- Varied multiplicities between tables (one-to-one, one-to-many, many-to-many);
- Tables with multiple attributes, including the necessary primary and foreign keys;
- Relationships between tables, with the corresponding keys and associated multiplicities.

In the project, generalizations between tables, self-dependencies, and composite foreign keys were not considered. Any model with these functionalities in our project must be invalidated in the corresponding metamodel.

3.1.3 SQL Code

The final objective of the project is to ensure a standard, valid Structured Query Language (SQL) code that can be executed, for example, in one database engine like PostgreSQL[11]. For the final code to be considered valid, the following requirements must be met:

- The output file must have the **.sql** extension;

[10] https://www.geeksforgeeks.org/relational-model-in-dbms/
[11] https://www.postgresql.org

- There must be a schema definition, **CREATE SCHEMA**, in the file header;
- There must be **DROP TABLE** statements in the header of the file to ensure the database can be reset without key collisions or duplicate IDs;
- The definition of each table, attributes and relationships;
- The explanation of primary keys and foreign keys, if available.

The code snippet in Annex A demonstrates the expected result from the gym example. It is important to note that in SQL, all generated tables explicitly define their primary keys and, where applicable, the corresponding foreign keys. For standardization purposes, and since not all database engines operate with the same keywords, implementations of internal primary keys using keywords such as **AUTOINCREMENT**, **ROWID** or **SERIAL** have been excluded. Instead, it is assumed that all keys are inserted manually. Conversely, to ensure final code validation in database engines, the **NOT NULL** constraint should be applied to all primary and foreign key declarations.

3.2 Concepts

After defining the problem domain and to gain a more high-level perspective for preparing the corresponding metamodels, the key concepts of both models were gathered and explored.

In the case of the ER model, the following components were identified:

- **Entity**: represents a distinct object or concept within the system;
- **Attribute**: defines the characteristics of an entity;
- **Relation**: represents the connections between two or more entities.

Regarding constraints, it is important to note that each entity can have multiple attributes, and each attribute belongs to only one entity. On the other hand, a relation is the result of the connection between a pair of entities. The multiplicity of each relation can be one-to-one, one-to-many, or many-to-many.

In contrast to the previous model, the Relational Model includes other concepts that are more closely aligned with the properties found in SQL:

- **Table**: corresponds to the entity in the previous model;
- **Column**: corresponds to the attribute of each entity;
- **Primary Key**: column that uniquely identifies each record in the table;
- **Foreign Key**: column in a table that references another existent table.

In terms of constraints, we can conclude that a table in this Relational Model has columns, and each column belongs to a single table. Additionally, there is only one primary key per table, but multiple foreign keys can exist, provided they have a valid reference to another table.

Each of these identified components, along with their constraints, was implemented using Ecore classes. In the following section, we will express the process of creating the corresponding Metamodels.

4 Metamodel Design

Having established the key components and constraints of both the Entity-Relationship model and the Relational model, we were ready to delve into the design of the metamodels. Using Ecore, we defined the classes and relationships that encapsulate the identified components, ensuring that they adhere to the constraints we had outlined.

4.1 ER Metamodel

The ER Metamodel was formed by:

- **Entity**: has a ename (EString) and can be associated with multiple Attributes;
- **Attribute**: has a aname (EString), a data type (EString), isIdentifier (EBoolean) and a correspondent Entity;
- **Relation**: has a unique rname (EString) and a multiplicity, defined by two integer pairs (EInt) representing the lower and upper bounds, which describe the number of instances of each entity that can participate in the relationship.

For many-to-many multiplicity cases, we considered the standard Ecore value: -1. To ensure Ecore compatibility, we added an **Model** component that can contain all entities and relationships in the system and has a name (EString). Since ER models do not inherently indicate the potential primary keys of entities, and this information is crucial for defining the future relational model, we chose to add a boolean attribute, **isIdentifier**, which marks attributes that, being unique and non-null, can independently identify the entity within the system.

The Ecore framework generated a visual representation of the metamodel. The Figure 3 provides an overview of how entities, attributes, and relations are interconnected:

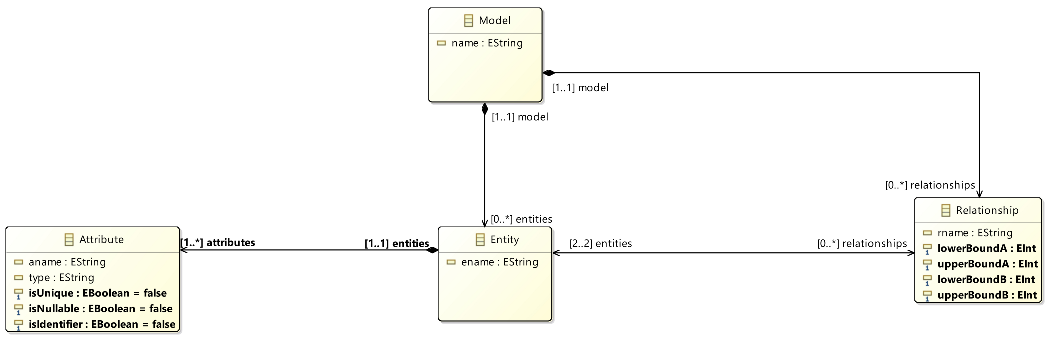

Figure 3: ER Metamodel

4.2 Relational Metamodel

The Relational Metamodel was formed by:

- **Schema**: has a name (EString) and can be associated with all system Tables;
- **Table**: has a name (EString) and can be associated with multiple Columns;
- **Column**: has a name (EString), a data type (EString), isNullable (EBoolean), isUnique (EBoolean), and has a correspondent Table;
- **Primary Key**: a column that uniquely identifies each record in the table;
- **Foreign Key**: a column in one table that creates a link to the primary key of another table, establishing a relationship between the two tables;

The Ecore framework generates a visual representation of the Relational metamodel, illustrating the relationships between tables, columns, and keys. The Figure 4 clarifies how the components interact within the relational structure:

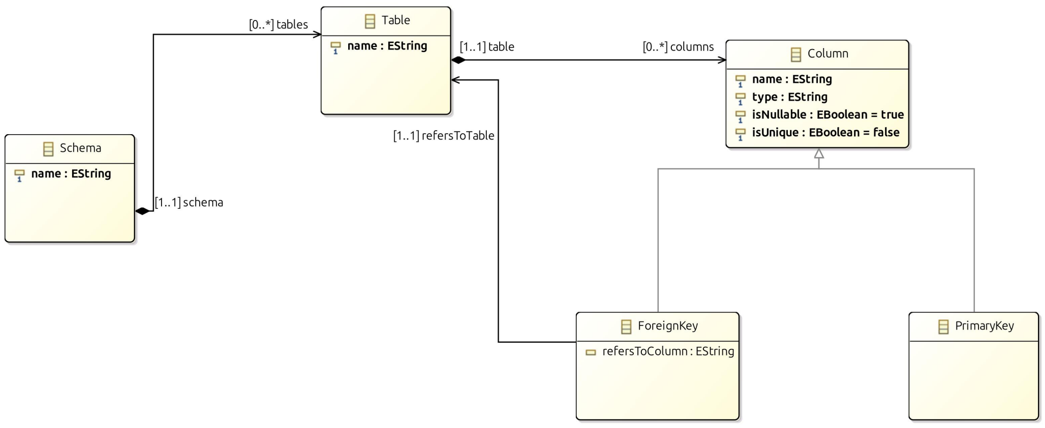

Figure 4: Relational Metamodel

Note that the generalization of Column into Primary Key and Foreign Key is an incomplete and disjoint generalization. In terms of implementation, three attributes were added to Column:

- **isPrimaryKey**: (boolean - EBoolean), which defaults to False and indicates whether this column is a primary key in the corresponding table;
- **isForeignKey**: (boolean - EBoolean), which defaults to False and indicates whether this column is a foreign key in the corresponding table;
- **refersToColumn**: (string - EString), which defaults to Null and represents the name of the referenced table if this column is a foreign key.

Multiplicities do not need to be explicitly defined in the metamodel, as they are expressed by the existence of foreign keys between tables or by adding new tables, which will be ensured by the model-to-model transformation discussed in later sections.

5 Metamodel Constraints

In this section, we explored the constraints and validations of the metamodels. This process involved applying various integrity constraints and invariants to ensure that the generated models adhere to the defined rules. The metamodels were subjected to various tests that checked each of the programmed constraints. We used the Object Constraint Language (OCL) within the Ecore framework to express these constraints, providing a mechanism for validating the integrity of our inputs.

5.1 Constraints

To maintain data integrity throughout the modeling process, we implemented the following OCL constraints:

- **C1**: The model/schema must contain entities/tables with unique, not-empty names;
- **C2**: All entities/tables must have attributes/columns with unique, not-empty names;
- **C3**: All entities/tables must have an identifier/primary key;
- **C4**: The lower multiplicity must be less than or equal to the upper bound, and both values must be non-negative;
- **C5**: The primary key must be unique and not null;
- **C6**: Columns in tables that are foreign keys must refer to an existing table in the system, other than itself;
- **C7**: No schema, table or column name can be an SQL Reserved Word;
- **C8**: The column type must a valid SQL data type;
- **C9**: The column referenced by a foreign key must be UNIQUE;
- **C10**: The referenced table of a foreign key must have the referenced column;
- **C11**: The column type of a foreign key must be of the same type as the one it references.

Constraints C1 to C3 were implemented in both metamodels, while C4 applies only to the ER Metamodel. Constraints C5 to C11 are specific to the Relational Metamodel. The implementation details can be found in the **er.ecore** and **relational.ecore** files attached to the submission.

These constraints ensure that the models are well-defined and that the relationships between various components are valid, thereby preserving the integrity of the final database structure.

5.2 Validation

The validation of the modeling constraints involved encoding various test cases to verify compliance with the metamodels. This step was essential to confirm that the implemented constraints functioned as intended and that the models adhered to the specifications set in the metamodel design.

The project submission includes a tests folder containing various test cases. For instance, the file **test1-er.xmi** was used as input for the ER metamodel to test Constraint 1 (C1), while **test5-relational.xmi** served as input for the Relational Metamodel to test Constraint 5 (C5). Additional files, such as **gym-er.xmi**, **gym-relational.xmi**, **complex-er.xmi**, and **complex-relational.xmi**, represent valid models for a gym example in both metamodels, as well as a more complex but still valid example. Table 1 summarizes the expected and obtained results for each test case following validation attempts.

Targeted Constraint	**Test File**	**Expected Result**	**Obtained Result**
C1	test1-<er, relational>.xmi	Fail	Fail
C2	test2-<er, relational>.xmi	Fail	Fail
C3	test3-<er, relational>.xmi	Fail	Fail
C4	test4-er.xmi	Fail	Fail
C5	test5-relational.xmi	Fail	Fail
C6	test6-relational.xmi	Fail	Fail
C7	test7-relational.xmi	Fail	Fail
C8	test8-relational.xmi	Fail	Fail
C9	test9-relational.xmi	Fail	Fail
C10	test10-relational.xmi	Fail	Fail
C11	test11-relational.xmi	Fail	Fail
All	gym-<er, relational>.xmi	Success	Success
All	complex-<er, relational>.xmi	Success	Success

Table 1: Validation Results for Metamodel Constraints

Full coverage of all constraints was achieved, ensuring the integrity of the generated ER and Relational metamodels.

6 Conclusion

In this phase, we successfully explored and implemented key techniques and technologies within Model-Driven Software Engineering to facilitate the automation of application development on low-code platforms. By designing and validating metamodels for both Entity-Relationship and Relational models, we established a solid foundation for understanding the essential components and constraints inherent in the modeling process. This groundwork clarified the relationships between entities, attributes, and relationships, ensuring the integrity and validity of the models throughout the project.

7 Contributions

In this phase, the group members have contributed equally for the project's development. As a result, it was given the same grade in percentage, so that its sum is 100%, as shown on the list below:

- **Fábio Sá**: 25%;
- **Filipe Fonseca**: 25%;
- **Lourenço Gonçalves**: 25%;
- **Pedro Ferreira**: 25%.

In qualitative terms, all the group members have contributed in the whole project process. In other words, every group member has helped with:

- Modeling Domain Analysis;
- Modeling Language Design, contributing the development of ER and Relational models;
- Development of integrity constraints for the models;
- Models Testing;
- Writing the report.

References

[Brambilla et al., 2017] Brambilla, M., Cabot, J., and Wimmer, M. (2017). *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, San Rafael, CA, second edition.

A Annex

```
CREATE SCHEMA Gyms;

DROP TABLE IF EXISTS Gym;
DROP TABLE IF EXISTS Customer;
DROP TABLE IF EXISTS Equipament;
DROP TABLE IF EXISTS isMember;

CREATE TABLE Gym (
     id INTEGER NOT NULL,
     name VARCHAR(255),
     address VARCHAR(255),
     tax_id: Integer,
    PRIMARY KEY (id)
);

CREATE TABLE Customer (
     id INTEGER NOT NULL,
     name VARCHAR(255),
     address VARCHAR(255),
     age INTEGER,
     tax_id: INTEGER,
    PRIMARY KEY (id)
);

CREATE TABLE Equipament (
     id INTEGER NOT NULL,
     name VARCHAR(255),
     min_weight INTEGER,
     max_weight INTEGER,
     gym_id INTEGER NOT NULL,
    FOREIGN KEY (id) references Gym (id)
);

CREATE TABLE isMember (
     id INTEGER NOT NULL,
     client_id INTEGER NOT NULL,
     gym_id INTEGER NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (client_id) REFERENCES Client (id),
    FOREIGN KEY (gym_id) REFERENCES Gym (id)
);
```