

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ALEXANDRE TORRES

**Essential Notation for Object-Relational
Mapping**

Thesis presented in partial fulfillment of the
requirements for the degree of Doctor in
Computer Science.

Prof. Dr. Renata Galante
Adviser

Prof. Dr. Marcelo S. Pimenta
Co-adviser

Porto Alegre, Abril, 2014.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Torres, Alexandre

Essential Notation for Object-Relational Mapping / Alexandre Torres. – 2014.

188 f.:il.

Orientadora: Renata Galante.

Co-orientador: Marcelo Pimenta.

Tese (Doutorado) – Universidade Federal do Rio Grande do Sul, Instituto de Informática, Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2014.

1. Object-Relational Mapping. 2. Model-Driven Development. 3. Patterns. 4. UML. 5. Relational Model. I. Galante, Renata, orient. II. Pimenta, Marcelo, coorient. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Carlos Alexandre Netto

Vice-Reitor: Prof. Rui Vicente Oppermann

Pró-Reitor de Pós-Graduação: Prof. Vladimir Pinheiro do Nascimento

Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb

Coordenador do PPGC: Prof. Luigi Carro

Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

Essential Notation for Object-Relational Mapping

ABSTRACT

This thesis presents the Essential Notation for Object-Relational Mapping (ENORM), a general purpose notation that represents structural concepts of Object-Relational Mapping (ORM). The goal of ENORM is to facilitate the design by the clear application of ORM patterns, document mappings with a platform independent notation, and became a repository for model-driven transformations, partial code generation, and round-trip engineering tools. ENORM is a UML profile based notation, designed to represent patterns within a domain modeling logic, with objects of the domain incorporating both behavior and data.

The notation represents patterns adopted by widespread ORM frameworks in the market (*Active Record*, of *Ruby*; *SQLAlchemy*, of *Python*; *Entity Framework*, of Microsoft *.net*; *JPA*, *Cayenne*, and *MyBatis*, of *Java*), following the Don't Repeat Yourself and Convention over Configuration principles. ENORM was evaluated by controlled experiments, comparing the modeling by students with the use of separated UML and relational models, achieving significantly more goals in the majority of the scenarios, without being significantly different in the worst experimental scenarios.

Keywords: Object-Relational Mapping, Model-Driven Development, Patterns, UML, Relational Model.

RESUMO

Esta tese apresenta a Notação Essencial para Mapeamento Objeto-Relacional (em inglês, ENORM), uma notação de propósito geral que representa os conceitos estruturais do Mapeamento Objeto-Relacional (MOR). O objetivo de ENORM é facilitar o projeto através da aplicação clara dos padrões MOR, documentação dos mapeamentos com uma notação independente de plataforma, e tornar-se um repositório para transformações dirigidas por modelos, geração parcial de código e ferramentas de engenharia *round-trip*. ENORM é uma notação baseada em perfil UML, projetada para representar padrões pertencentes a lógica de modelo do domínio, com objetos do domínio incorporando tanto comportamento como dados.

A notação representa padrões adotados por frameworks MOR difundidos no mercado (*Active Record*, do *Ruby*; *SQLAlchemy*, do *Python*; *Entity Framework*, da Microsoft *.net*; *JPA*, *Cayenne*, and *MyBatis*, do *Java*), seguindo os princípios Não se repita e Convenção sobre Configuração. ENORM foi avaliado por experimentos controlados, comparando a modelagem de estudantes com modelos UML e relacionais separados, atingindo um número significativamente maior de objetivos na maioria dos cenários, sem ser significativamente diferente nos piores cenários experimentais.

Palavras-chave: Mapeamento Objeto-Relacional, Desenvolvimento Dirigido por Modelos, Padrões, UML, Modelo Relacional.

LIST OF ABBREVIATIONS

ANOVA	Analysis of Variance
AST	Abstract Syntax Tree
BNF	Backus–Naur Form
CoC	Convention over Configuration
DA	Data Administrator
DRY	Do not Repeat Yourself principle
EBNF	Extended Backus–Naur Form
EER	Extended Entity-Relationship
ER	Entity-Relationship
FK	Foreign Key
IMP	Impedance Mismatch Problem
JPA	Java Persistence API
MDD	Model-Driven Development
OO	Object-Oriented
ORM	Object-Relational Mapping
PIM	Platform Independent Model
PK	Primary Key
PSI	Platform Specific Information
PSM	Platform Specific Model
RAR	Ruby's Active Record
RDB	Relational Data Base
Rel	Relational model
SA	SQLAlchemy
SoC	Separation of Concerns
UML	Unified Modeling Language

LIST OF FIGURES

FIGURE 1.1: TEXT ORGANIZATION.....	19
FIGURE 2.1: JAVA EXAMPLES OF TABLE MODULE PATTERN (LEFT) AND DOMAIN MODEL PATTERN (RIGHT).....	21
FIGURE 2.2: ACTIVE RECORD EXAMPLE.....	23
FIGURE 2.3: INHERITANCE TO GENERATED CLASSES.....	24
FIGURE 2.4: INSTANTIATED MAPPING EXAMPLE FOR SQLALCHEMY LOOSE COUPLING APPROACH.....	24
FIGURE 2.5: JPA LOOSE COUPLING OVERVIEW.....	25
FIGURE 2.6: DATA MAPPER SUCH AS IN MYBATIS.....	27
FIGURE 2.7: ENTITY DATA MODEL FOR MS ENTITY FRAMEWORK (SNEED, 2012).....	29
FIGURE 2.8: CAYENNE MODELS DOMAIN AND DATABASE ELEMENTS SEPARATED.....	30
FIGURE 2.9: PRIMARY KEY FIELDS (A) AND PRIMARY KEY CLASS (B)..	31
FIGURE 2.10: ASSOCIATION RELATIONSHIP BETWEEN CLASSES.....	33
FIGURE 2.11: ASSOCIATION TABLE EXAMPLE.....	36
FIGURE 2.12: UML MODEL (LEFT) AND TABLES (RIGHT) OF THE ASSOCIATION TABLE EXAMPLE.....	37
FIGURE 2.13: DEPENDENT MAPPING PATTERN EXAMPLE: CLASS MODEL (UPPER) AND DATABASE MODEL (LOWER).....	38

FIGURE 2.14: INHERITANCE EXAMPLE FOR ACCOUNT.....	40
FIGURE 2.15: SINGLE-TABLE APPROACH FOR ACCOUNT EXAMPLE.....	40
FIGURE 2.16: CLASS-TABLE APPROACH FOR ACCOUNT EXAMPLE.....	40
FIGURE 2.17: CONCRETE-TABLE APPROACH FOR ACCOUNT EXAMPLE.	41
FIGURE 2.18: INDEPENDENT KEYS IN CLASS-TABLE INHERITANCE EXAMPLE.....	42
FIGURE 3.1: MAIN VISUAL ELEMENTS AND THEIR MEANING.....	47
FIGURE 3.2: SIMPLE TRANSACTION EXAMPLE.....	48
FIGURE 3.3: SUMMARY ACCOUNT EXAMPLE.....	48
FIGURE 3.4: DATABASE MODEL OF ACCOUNT EXAMPLE.....	49
FIGURE 3.5: MAIN ELEMENTS OF THE ENORM PROFILE.....	51
FIGURE 3.6: A CLASS EMBEDDED BY TWO PERSISTENT CLASSES.....	54
FIGURE 3.7: EMBEDDED CLASSES REFERENCING PERSISTENT CLASSES.....	55
FIGURE 3.8: TRANSITIVITY OF EMBEDMENT.....	55
FIGURE 3.9: USING <<EMBED>> FOR DEPENDENT MAPPING.....	56
FIGURE 3.10: TWO COLLECTION-EMBEDMENTS EXAMPLE.....	57
FIGURE 3.11: TWO DEPENDENT COLLECTIONS TO THE SAME CLASS..	57
FIGURE 3.12: MAP WITH KEY REFERENCE.....	58
FIGURE 3.13: THREE DIFFERENT INHERITANCE EXAMPLES WITHOUT PARENT TABLE.....	58
FIGURE 3.14: INHERITED ASSOCIATION WITH PERSISTENT CLASS.....	59
FIGURE 3.15: ALTERNATIVE WAY TO EXPRESS MAPPED SPECIALIZATIONS WITH ENORM.....	59

FIGURE 3.16: ENORM PROFILE MODEL OF GENERATORS.....	60
FIGURE 3.17: PROFILE MODEL OF TABLE DEFINITIONS, FOR INDEXES AND CONSTRAINTS.....	60
FIGURE 3.18: DEFINITION EXAMPLE WITH UNIQUE INDEX CONSTRAINT DEFINITION.....	61
FIGURE 3.19: A SKETCH OF META-MODEL WITH FLEXIBLE DATA SOURCES.....	62
FIGURE 3.20: TEMPLATE PARAMETER EXAMPLE.....	63
FIGURE 3.21: EBNF SPECIFICATION FOR ENORM LABELS.....	64
FIGURE 3.22: VISUAL DISTRIBUTION OF THE ENORM NON-TERMINALS.	65
FIGURE 3.23: MODELING TOOL SCREEN SHOT.....	66
FIGURE 3.24: EXPERIMENTAL TOOL ARCHITECTURE.....	67
FIGURE 3.25: MDD SCENARIO FOR ENORM MODELS.....	67
FIGURE 3.26: UML PROFILE FOR DATA MODELING EXAMPLE (AMBLER, HARTFORD AND RUECKERT, 2003).....	69
FIGURE 4.1: PARTY PATTERN DESIGNED WITH ENORM.....	72
FIGURE 4.2: ACCOUNTABILITY, FIRST MODEL.....	78
FIGURE 4.3: ACCOUNTABILITY WITH PARTY TYPE PATTERN AND KNOWLEDGE LEVEL.....	79
FIGURE 4.4: CONCEPTUAL MODEL FOR THE RESOURCE ALLOCATION PATTERN.....	90
FIGURE 4.5: RESOURCE ALLOCATION ENORM MODEL.....	91
FIGURE 5.1: TASK 1 - ADDRESS BOOK UML AND ER MODELS.....	108
FIGURE 5.2: TASK 1 - ADDRESS BOOK ENORM MODEL.....	109

FIGURE 5.3: SCREEN SHOTS OF TREATMENT A (LEFT) AND B (RIGHT) USING THE MODELING TOOL.....	113
FIGURE 5.4: FEEDBACK OF I1.....	116
FIGURE 5.5: EXPERIENCE LEVELS AMONG GROUPS (GRAPHIC).....	118
FIGURE 5.6: MEAN DIFFICULTY LEVELS (Q1-Q3, LEFT) AND PREFERRED METHOD (Q4, RIGHT).....	119

LIST OF TABLES

TABLE 2.1: SUMMARY OF FRAMEWORKS.....	22
TABLE 2.2: ORM FRAMEWORKS SUPPORT FOR EACH PROPOSED CRITERION.....	44
TABLE 2.3: SUMMARY OF DESIGN DECISIONS BASED UPON ORM FRAMEWORKS.....	45
TABLE 4.1: MAIN CORRESPONDENCE OF ENORM CONCEPTS.....	97
TABLE 4.2: UML CLASS MAPPINGS, NON TRIVIAL CASES.....	98
TABLE 4.3: PLATFORM SPECIFIC INFORMATION.....	99
TABLE 4.4: DESIGN QUESTIONS AND RESPONSE SCOPE.....	100
TABLE 5.1: SUBJECTS AND EXPERIMENTS.....	104
TABLE 5.2: ANALYSIS PATTERNS AND TASKS.....	108
TABLE 5.3: FEATURE COVERAGE AND TASK GOALS.....	110
TABLE 5.4: TASKS AND TIME CONSTRAINTS FOR EXPERIMENT I1.....	113
TABLE 5.5: CODES USED IN THE GROUP EXPERIMENT FOR RANDOM ASSIGNMENT.....	114
TABLE 5.6: RESULTS FOR THE ANOVA OF MISSES CONSIDERING THE SEQUENCE.....	115
TABLE 5.7: LEAST SQUARE MEANS OF MISSES AT I2, WITH ADJUSTMENT TUKEY-KRAMER.....	116
TABLE 5.8: ORIGINAL DATA OF VARIABLE TIME AT I2.....	117

TABLE 5.9: GROUP STATISTICS FOR EXPERIENCE LEVELS.....	117
TABLE 5.10: T-TEST FOR EQUALITY OF MEANS (EQUAL VARIANCES ASSUMED).....	118
TABLE 5.11: MANN-WHITNEY U TEST APPLIED TO DIFFICULT LEVELS.	119
TABLE 5.12: MANN-WHITNEY RANKS, MEANS, AND DEVIATION OF G.	120

CONTENTS

ABSTRACT.....	3
RESUMO.....	4
LIST OF ABBREVIATIONS.....	5
LIST OF FIGURES.....	6
LIST OF TABLES.....	10
1 INTRODUCTION.....	16
2 ORM PATTERNS AND FRAMEWORKS.....	20
2.1 Survey organization.....	20
2.1.1 Object-Relational Patterns.....	20
2.1.2 Selected Frameworks for the Survey.....	21
2.1.3 About the examples.....	22
2.2 Transparency and Coupling.....	22
2.2.1 Discussion.....	26
2.3 Mapping Type.....	26
2.3.1 Mapping Classes to Many Tables.....	27
2.3.2 Discussion.....	28
2.4 Model-based Mapping.....	28
2.4.1 Discussion.....	30
2.5 Identity.....	30
2.5.1 Discussion.....	32
2.6 Foreign Key.....	32
2.6.1 Fetch Strategy.....	34
2.6.2 Discussion.....	35
2.7 Association table.....	35
2.7.1 Discussion.....	37
2.8 Embedded Values Support.....	37
2.8.1 Discussion.....	39

2.9 Inheritance Mapping	39
2.9.1 Discussion.....	43
2.10 Summary	43
3 ESSENTIAL NOTATION FOR ORM (ENORM)	46
3.1 Overview	46
3.2 A Simple Example	47
3.3 A not so Simple Example	48
3.4 ENORM Meta-model	50
3.5 Special Mapping Cases	53
3.5.1 Embedded Values.....	53
3.5.2 Maps.....	57
3.5.3 Inheritance.....	58
3.5.4 Auto-generated Columns.....	59
3.5.5 Constraints and Indexes.....	60
3.6 Limitations	61
3.6.1 Flexible Data Sources.....	61
3.6.2 Qualified Associations.....	62
3.6.3 Multiple Inheritance, Multiple Types.....	62
3.6.4 Association Class and “n-ary”.....	62
3.6.5 Generics and Template Parameters.....	62
3.7 ENORM Notation Reference	63
3.8 Modeling Tool	65
3.8.1 Modeling Tool for the Experiments.....	66
3.8.2 Future Steps.....	67
3.9 Other Class Models and Persistence Extensions	68
3.9.1 A UML Profile for Data Modeling.....	68
3.9.2 Information Management Meta-model (IMM).....	69
4 ENORM IN PRACTICE: APPLICATION EXAMPLES	71
4.1 ENORM and ORM Frameworks	71
4.2 Party Pattern for Accountability	72
4.2.1 Mapping Persistent class Telephone.....	73
4.2.1.1 Using JPA.....	73
4.2.1.2 Using SQLAlchemy.....	73
4.2.1.3 Using ActiveRecord of Ruby.....	74
4.2.2 Embedded classes.....	75
4.2.2.1 Using JPA.....	75
4.2.2.2 Using SQLAlchemy.....	75
4.2.2.3 Using ActiveRecord of Ruby.....	75
4.2.3 Party, Person, Company, and Flat inheritance.....	75
4.2.3.1 Using JPA.....	75
4.2.3.2 Using SQLAlchemy.....	76
4.2.3.3 Using ActiveRecord of Ruby.....	77

4.3 Accountability Type Model.....	78
4.3.1 Implementing the Associations.....	79
4.3.1.1 Using JPA.....	80
4.3.1.2 Using SQLAlchemy.....	80
4.3.1.3 Using ActiveRecord of Ruby.....	81
4.4 Account Model.....	82
4.4.1 Entry is a dependent entity.....	82
4.4.1.1 Using JPA.....	82
4.4.1.2 Using SQLAlchemy.....	82
4.4.1.3 Using ActiveRecord of Ruby.....	82
4.4.2 Account mapped by two tables.....	83
4.4.2.1 Using JPA.....	83
4.4.2.2 Using SQLAlchemy.....	83
4.4.2.3 Using ActiveRecord of Ruby.....	83
4.4.3 Vertical Inheritance of Account.....	84
4.4.3.1 Using JPA.....	84
4.4.3.2 Using SQLAlchemy.....	84
4.4.3.3 Using ActiveRecord of Ruby.....	85
4.4.4 Properties and columns with distinct names.....	85
4.4.4.1 Using JPA.....	86
4.4.4.2 Using SQLAlchemy.....	86
4.4.4.3 Using ActiveRecord of Ruby.....	86
4.4.5 Overrides and Embedded objects referencing persistent classes.....	86
4.4.5.1 Using JPA.....	86
4.4.5.2 Using SQLAlchemy.....	87
4.4.5.3 Using ActiveRecord of Ruby.....	87
4.4.6 The Account-Entry association.....	87
4.4.6.1 Using JPA.....	88
4.4.6.2 Using SQLAlchemy.....	88
4.4.6.3 Using ActiveRecord of Ruby.....	89
4.5 Resource Allocation Model.....	89
4.5.1 Horizontal Inheritance at the Resource Allocation Tree.....	92
4.5.1.1 Using JPA.....	92
4.5.1.2 Using SQLAlchemy.....	93
4.5.1.3 Using ActiveRecord of Ruby.....	93
4.5.2 Overriding inherited properties and associations.....	94
4.5.2.1 Using JPA.....	94
4.5.2.2 Using SQLAlchemy.....	95
4.5.2.3 Using ActiveRecord of Ruby.....	95
4.5.3 Association to general classes with horizontal specializations.....	96
4.5.3.1 Using JPA.....	96
4.5.3.2 Using SQLAlchemy.....	97
4.5.3.3 Using ActiveRecord of Ruby.....	97
4.6 Remarks about implementing ENORM models.....	97
4.6.1 Guidelines for MDD.....	98
5 EMPIRICAL EVALUATION.....	102
5.1 Experimental Related Work.....	102
5.2 Planning and Design.....	103
5.2.1 Subjects.....	104
5.2.2 Task Design.....	105
5.2.2.1 Individual Experiments.....	105
5.2.2.2 Group Experiment.....	106

5.2.3 Hypothesis Formulation, Factors and Variables.....	106
5.2.4 Tasks and Feature Coverage of the Individual Experiment.....	108
5.2.5 Tasks of the Group Experiment.....	112
5.2.6 Experimental Setting.....	112
5.3 Results and Analysis.....	114
5.3.1 Individual Experiment with Crossover (I1).....	114
5.3.1.1 Analysis of the Feedback.....	115
5.3.2 Individual Experiment with Time Measurements (I2).....	116
5.3.2.1 Results Regarding the Variable “Misses”.....	116
5.3.2.2 Results Regarding the Variable “Time”.....	117
5.3.2.3 Experience Level Influence.....	117
5.3.2.4 Analysis of the Feedbacks.....	118
5.3.3 Group Experiment (G).....	119
5.3.4 Analysis Summary.....	120
5.4 Validity Evaluation.....	120
5.4.1 Internal Validity.....	121
5.4.2 Construct Validity.....	121
5.4.3 External Validity.....	121
5.4.4 Conclusion Validity.....	122
6 CONCLUSION.....	123
REFERENCES.....	125
APPENDIX A – CROSSOVER EXPERIMENT (I1).....	131
APPENDIX B – NON-CROSSOVER EXPERIMENT (I2).....	138
APPENDIX C – TASKS (INDIVIDUAL EXPERIMENTS).....	142
APPENDIX D – TASKS (GROUP EXPERIMENT).....	164
APPENDIX E – EXTENSIONS TO ACTIVE RECORD.....	170
APPENDIX F – RELATED PUBLICATIONS.....	173
ANNEX A – EXPERIMENTAL REPORT – EXPERIMENT I1.....	174
ANNEX B – EXPERIMENTAL REPORT – EXPERIMENT I2.....	177

1 INTRODUCTION

Relational Databases (RDBs) are the backbone of information systems, and nobody knows when (or if) this will change (ATZENI et al., 2013). However, the Impedance Mismatch Problem (IMP) (ATKINSON and BUNEMAN, 1987; COPELAND and MAIER, 1984) continues to haunt object oriented designs that tend to underestimate the Object-Relational Mapping (ORM) difficulties.

In the past decade we saw a growing adoption of ORM frameworks by information system developers of distinct platforms such as Java, C#, Python, and Ruby on Rails. These frameworks have most of their resources based upon established patterns (BROWN and WHITENACK, 1996; FOWLER, 2002; KELLER, 1997), and its use spread a more standardized approach for the IMP. Nevertheless, mappings scattered in the code, annotations and/or XML files are difficult to read, understand, and reason about changes.

The Model-Driven Development (MDD) proposes that models take on the main role on the system development process (BEYDEDA, BOOK and GRUHN, 2005; OMG, 2001). For an effective MDD approach, the information represented by models should be coherent, integrated, and computable, so that automatic transformations could turn models into executable system (MELLOR et al., 2004). The UML notation lacks a specific notation for persistence, or to map classes to database. The absence of mapping information poses a challenge for developing transformations.

The problem under study at this thesis is the lack of a persistence notation that serves both to document and communicate the mappings between relations and classes, and as an artifact with the necessary information for MDD. One of the roots of IMP is the conceptual miscommunication (AMBLER, 2003), and we believe that an adequate persistence notation will contribute solving this problem.

This thesis presents a general purpose notation named *Essential Notation for Object-Relational Mapping* (ENORM). ENORM extends the UML class model, by a profile, and offers a concise set of new visual elements specific to represent the structural concepts of ORM. These essential concepts are based upon persistence patterns, and the way these patterns are adopted by distinct ORM frameworks in the market.

The goal of ENORM is to facilitate the design by the clear application of ORM patterns, document mappings with a platform independent notation, and be a repository for MDD transformations, partial code generation, and round-trip engineering tools. ENORM is designed to represent patterns within a domain modeling logic, with objects of the domain incorporating both behavior and data (FOWLER, 2002). Therefore, this

study does not encompass the design of queries, nor the specification of models that describe the behavior of the systems.

In a nutshell, the contributions here presented are a survey relating ORM patterns and frameworks; the ENORM notation, comprising graphical elements, the profile and the modeling tool; a set of examples using ENORM and implemented by selected frameworks, summarizing the mappings from model to implementation; and comparative experiments evaluating the modeling activity with ENORM.

The main modeling principle of the ENORM approach is the *Don't Repeat Yourself* (DRY) (AMBLER, 2002; HUNT and THOMAS, 1999), avoiding the duplication of domain concepts, such as separated class and table specifications for the same element. ORM providers, such as the Ruby's Active Record (RAR), and JPA, detected that classes and tables are often very similar, and that the ORM could be very straightforward most of the time. They adopted the Convention over Configuration (CoC) design pattern (CHEN, 2006), reducing the amount of configuration, and therefore code, to realize what would be an obvious mapping.

At a higher abstraction level, when modeling information systems with separated class and relational models, the designer often have to deal with duplicate definitions of the same domain objects. For instance, a class that represents a telephone, and the table that stores this information, may have a trivial mapping, predicted by convention: same names for class and table, properties and columns; or computable names, such as the table name being the plural of the class name. At a scenario that *often* can be predicted by convention, having two separated models seems to be a waste of effort and time. On the other hand, if *all* elements have one straightforward mapping, ENORM would not be needed, because no mapping would be necessary.

ENORM has a single model approach, and is focused at the class model. If the class can be mapped by convention, or in other words, without additional mapping information, it is not necessary to specify this information. Conversely, if this mapping is not trivial, the meta-model supports the detailing of this mapping. ENORM was designed to be easily understood by developers and rich enough for MDD tools, allowing the specification of the relevant persistence details, or hiding what can be inferred.

Database design concerns, such as data normalization, Primary Key (PK) and relationships are distinct from the behavior oriented forces of high cohesion and low coupling of OO design (AMBLER, 2003). The single model approach is strong when it is necessary to link the RDB specification, and the domain specification, because it enforces the habit of reasoning about the two domains together. In other words, it is a synergistic modeling, because it shows the cooperation between database and classes.

However, another principle to consider is the violation of the Separation of Concerns (SoC), by specifying two traditionally separated viewpoints, such as database and OO, together. Nevertheless, the SoC applies ideally when each concern delivers distinct functionality, that can be developed and validated independently (PRESSMAN, 2010). This rarely fully applies to applications and databases.

Moreover, SoC is not the same of ignoring the mappings between software and database. ENORM main audience are the software developers and designers that need

to understand, evaluate, and document this information, in order to deliver software that deal with both viewpoints, taking in account database and software forces.

The database model may be independent of the system, shared among various distinct systems; or serve only one system. If the database is exclusive to the system, and its design is entirely under the responsibility of the same team of this system, the mappings tend to be more conventional. This, however, depends on the performance requirements, the amount of data, and the model itself: if the database performs poorly, and the model have to change, the mapping may not be the obvious one. This is what makes all database and class models hardly ever the same, and hardly ever automatically mapped by convention.

Moreover, if the database is independent, the mappings tend to be more complicated, because changes in the RDB are complicated, involving various parties and Data Administrators (DA) (AMBLER, 2003). At this scenario, ENORM is appropriate for developers understand the connections between their particular domain and the database, and also to communicate their needs to the DA or external parties. However, ENORM diagrams are not appropriate for the DA tasks at the production database, and are not intended to replace ER models for database design.

The scenario where a system access multiple databases is a variation of the shared database scenario, but with added complexity. ENORM meta-model allows the specification of the schema and catalog, that can be used to filter the models according to a certain database.

ENORM meta-model contains all mapping information, by convention or configuration, and it is easy, for a tool, to transform ENORM models to the database viewpoint, presenting a relational model. It is also easy to present a pure class viewpoint, removing the elements introduced by ENORM.

The essential elements, contained at the ENORM meta-model, are the result of a survey relating six commercial ORM tools, and the ORM pattern literature. This survey includes representatives of four Object-Oriented (OO) programming languages, selected among the top ten most popular OO languages of 2013: Java, C#, Ruby, and Python (CARBONNELLE, 2014; DE MONTMOLLIN, 2013; O'GRADY, 2013; TIOBE, 2013).

In order to evaluate our meta-model, we followed two strategies. The first is by implementing application examples for the models, following the *prove it with code* approach of agile modeling (AMBLER, 2002). Despite the name, it is not a formal proof, but based at the comparative implementation of a set of example models, using three distinct frameworks of three distinct OO languages. The goal is to capture, by examples, how the notation, and its meta-model, relates to practical implementation.

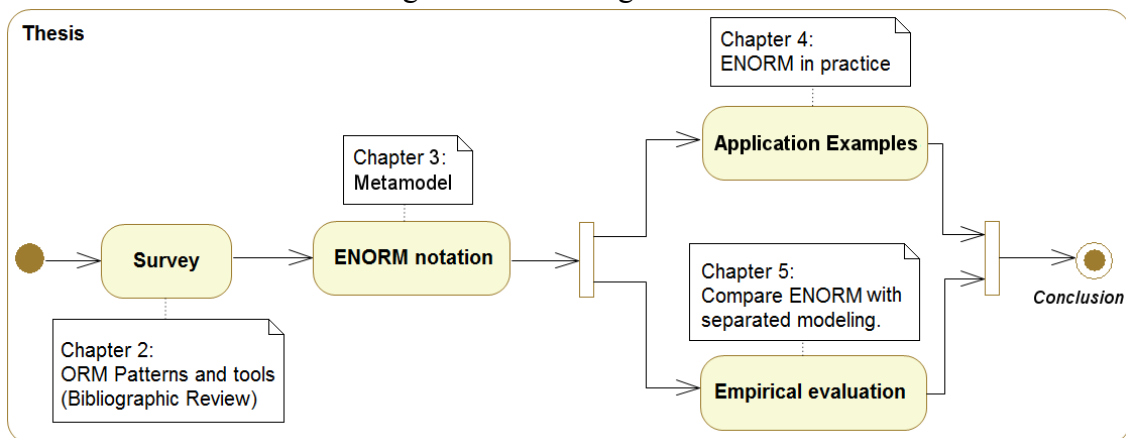
The second strategy focus at the empirical evaluation of ENORM single notation approach, in comparison with using separated class and relational models. The main issue under evaluation is if ENORM models are hard to understand and modify, due, perhaps, to a possible lack of SoC. The hypothesis tested was that ENORM single notation does not decrease the quality of models, and may perhaps increase its quality, independently of MDD, ORM framework, or actual implementation.

Two controlled experiments were performed, with computer science students, individually performing changes in models; and one experiment was performed with groups of students creating and integrating models, simulating database sharing among different systems. None of the results showed a decrease of quality, in terms of achieved goals and time, by using ENORM. In fact, in most studied scenarios, ENORM achieved significantly more goals, although no significant difference in time.

The last evaluation step would be a case study (WOHLIN et al., 2012), involving professionals and a real project, with modeling and development. This case study, unfortunately, could not be executed, and depends on establishing partnerships with the industry.

As the last contribution of this thesis, a modeling tool is under development. This tool allows the modeling using the notation, integrated with the Eclipse software development tool. Code generation, reverse engineering, and round-trip engineering are the next steps towards a MDD tool.

Figure 1.1: Text organization.



The text is organized as shown by Figure 1.1. Chapter 2 presents a bibliographic review surveying ORM patterns and tools, and organizing the knowledge in pattern based criteria and characteristics. Chapter 3 presents the ENORM notation, its meta-model, special cases, limitations, and tools under development. Chapter 4 present the application examples, based upon analysis patterns, and guidelines for developers in the context of MDD. Chapter 5 presents the empirical evaluation of the notation, in comparison with separated modeling. Chapter 6 presents the conclusion of this work.

2 ORM PATTERNS AND FRAMEWORKS

This chapter presents a bibliographic review relating ORM patterns and practices, based upon the study of representative ORM frameworks and tools. This review, presented in the format of a survey, has the purpose of conceptual organization, identifying the essential concepts necessary for our notation, and the consequences of ORM decisions at the OO design.

At the first section we explain the survey organization, introducing the domain patterns and frameworks under study. The following sections presents the survey, criteria by criteria. The last section presents a short summary relating patterns, frameworks, and design decisions.

2.1 Survey organization

At this first section we discuss the three basic patterns that guide the organization of the domain logic; the selection of ORM tools for this survey; and the examples used at this and the following chapters.

Each of the following sections describes a criteria proposed for characterizing and assessing ORM frameworks. After introducing some unifying terminology from patterns, we present and discuss the criteria in the context of the studied ORM frameworks. Some criteria are identified as coarse patterns such as *Embedded Values* or *Association Table*, but others encompass different patterns that operate over the same problem. For instance, *Coupling* and *Model Based* criteria are more architectural design oriented, while *identity/foreign key* are very close to the implementation patterns.

Under each criterion, one or more common characteristics are identified relating patterns and frameworks. The *inheritance mapping* criterion, for example, has each strategy as a characteristic, that may be available on each framework. After presenting and discussing all criteria, a short summary is presented in the end of the chapter.

2.1.1 Object-Relational Patterns

When developing enterprise applications with large and/or complex domain logic persisted by RDBs, there are different approaches on how to organize this domain logic, according to the pattern literature. The simplest *Transaction Script* pattern has a procedural approach, implementing a script for each action, business transaction, identified on the system. Conversely, the *Domain Model* pattern assigns the domain logic to the object model, hence each domain object incorporates both behavior and data (FOWLER, 2002).

A third pattern in this category is the *Table Module* pattern, that proposes a structure similar to the RDB schema. Each table, or view, has a singleton (a class that has just *one* instance object) that handles the persistence and business logic, for all rows of its table. This pattern is a middle term, between the hard to reuse transactional/procedural approach and the difficult to implement object-oriented *Domain Model* pattern.

The difference between *Table Module* and *Domain Model* may not be clear at first sight. Figure 2.1 exemplifies a simple case for a *Person* object that only has an *id* and a *name*. In the *Table Module* approach the *Person* class has one instance that deals with all *persons*, and thus has an insert method that deals with *Person* creation. There is no specific class representing the data of *Person*, hence the general purpose *DataRow* class is used to store the data for the retrieved person. In the *Domain Model* example, the code first instantiate a person object with its data, and then asks the object to *insert itself*. This is a very simple example, and the precise way to retrieve and insert information may change among different platforms, but the key difference is the general *DataRow* against the actually instantiation of *Person/Other specific class* objects.

Figure 2.1: Java examples of *Table Module* pattern (left) and *Domain Model* pattern (right).

<pre>int id=1; String name="Bob"; Person.instance.insert(id, name); DataRow row = Person.instance.retrieveById(id); assertEquals(row.getString("name"), "Bob");</pre>	<pre>int id=1; String name="Bob"; Person person = new Person(id, name); person.insert(); person = Person.retrieveById(id); assertEquals(person.getName(), "Bob");</pre>
---	---

The *Domain Model* pattern is the best approach in terms of an OO solution, because it deals with typed instances in a transparent way, better organizing complex logic. It gives access to resources such as polymorphism, relationships and inheritance, although when dealing with relational persistence, it may require more work and had a steeper learning curve (FOWLER, 2002).

The ORM solutions here studied, and our notation, are focused on the application of the *Domain Model* pattern. Several other patterns deal with how to read/persist objects within this approach, and will be identified in the following sections that analyze each ORM solution.

2.1.2 Selected Frameworks for the Survey

Due to the large number of ORM solutions in the market, we decided to select a small set of tools for our analysis, based in the access to documentation, distinguishing/unique approach to a given problem and maturity/insertion in the market. Of this list, we left out low level persistence layers that did not accomplish a minimum ORM such as *JDBC*, *ADO* (ActiveX Data Objects) and other *Record Set* layers, but included the most important tools, of four of the most popular programming languages (DE MONTMOLLIN, 2013; O'GRADY, 2013; TIOBE, 2013).

The *JPA* specification (DEMICHIEL, 2013) was the first obvious choice, encompassing a significant number of ORM solutions for the Java platform. The *MS Entity Framework* (MICROSOFT, 2012a) is both a major persistence layer, and an example of *model based* ORM tool. The *Ruby Active Record* is the major ORM solution for the *Ruby* platform (HEINEMEIER HANSSON, 2012).

Table 2.1: Summary of frameworks.

<i>Framework</i>	<i>Platform</i>	<i>URL</i>
ActiveRecord	Ruby	http://ar.rubyonrails.org/
Cayenne	Java	http://cayenne.apache.org
Entity Framework	MS .net	http://msdn.microsoft.com/en-us/library/bb399572.aspx
JPA 2	Java	http://jcp.org/aboutJava/communityprocess/final/jsr317/index.html
MyBatis	Java	http://www.mybatis.org/
SQLAlchemy	Python	http://www.sqlalchemy.org/

The *MyBatis* (formerly known as *iBatis*) is probably the most known solution with the *data mapping* approach to the ORM (BEGIN, GOODIN and MEADORS, 2007). The *SQLAlchemy* is a well known solution to the Python platform, and presents a hybrid approach to the coupling problem (BAYER, 2012). The Apache *Cayenne* is a model/generation based solution that allows some degree of organized customization (APACHE FOUNDATION, 2012). Table 2.1 presents a summary of frameworks and standards, their programming language platform, and their respective internet resources.

2.1.3 About the examples

The majority of the examples presented at this thesis were based upon Analysis Patterns (FOWLER, 1996). The models are variations of the *Party*, *Accountability*, *Knowledge level*, *Party Type*, *Account*, *Transaction*, *Quantity*, *Multilegged transaction*, *Summary Account*, and *Resource Allocation* analysis patterns, focusing at the *Accountability*, *Accounting*, and *Planning* analysis domains.

All model elements are in *italic*. References to patterns, frameworks, and tools are also in *italic*. Reference to meta-model elements, such as stereotypes or meta-classes, are in *sans serif* (Arial). Classes always starts with uppercase, while properties are written starting with lowercase.

Sometimes, an example database model will present tables that do not follow any naming convention. This is deliberate, because it is not uncommon to find databases that do not follow a convention, due to the schema longevity and resistance of the administrators to refactoring. For example, some tables are plural, other are not.

2.2 Transparency and Coupling

Coupling is a measure of interconnection among modules in a software. Software with low coupling levels tends to be easier to understand and less prone to the propagation of errors (PRESSMAN, 2001). In ORM frameworks, *transparency* is commonly referred to as the ability to keep a *loose coupling* between application and the persistence framework, mainly by keeping the domain level classes unaware of the persistence framework (BAUER and KING, 2004). *Transparency* is usually achieved by having object-relational mapping information isolated in *external configuration* files or annotations. Under *transparency/coupling* criterion, loose coupled frameworks are those that do not impose coupling between the domain classes and the persistence framework.

Lets take the *Domain Model* pattern as our context and the *Active Record* pattern as the starting point for our discussion of an ORM solution (FOWLER, 2002). Each class of the domain is responsible for retrieving and maintaining its data in the database and each instance of the domain class represents one row (or record) in the database. This frequently means implementing an interface of common CRUD (MARTIN, 1983) methods, or extending an abstract super class, responsible for encapsulating the common services among all persistent classes.

The *Ruby* platform can be used as an example of *Active Record* implementation for ORM (HEINEMEIER HANSSON, 2012), although it uses a *Metadata Mapper* as it will be examined in the mapping criteria. The persistence framework provides a base abstract class named *ActiveRecord::Base* that implements most of the SQL conversation with the database. By extending *ActiveRecord*, the classes of the domain will be automatically persisted according to the mapping contract defined within the framework.

Figure 2.2: Active Record example.

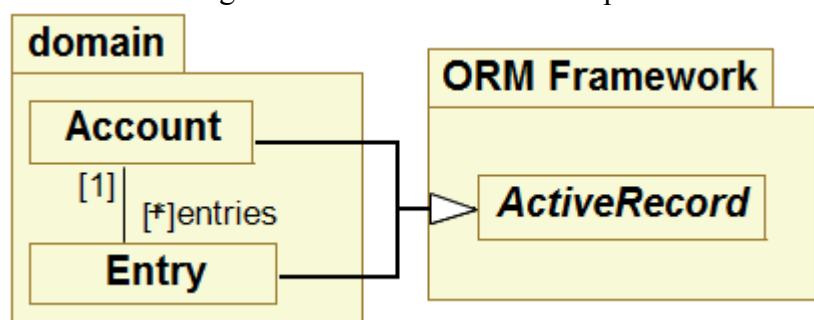


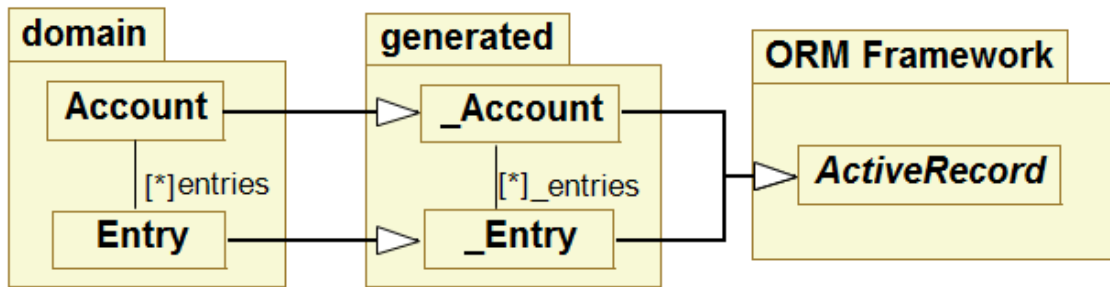
Figure 2.2 depicts an *Active Record* example on *Ruby* consisting of the *Account to Entry* relationship. Each class of the domain must specialize *ActiveRecord*, and optionally override its methods and properties, to represent persistent objects. *Ruby* is interpreted, dynamic typed, and highly reflective, characteristics that help with adopting conventions that represent mapping constructions, without losing the ability to override most of these conventions. For example, the PK in *Ruby* is, by convention, a column named “ID”, and the name of the table is the plural of the name of the class. However, the developer can override the super-class replacing the convention for one or more classes. Nevertheless, some complex mappings such as of *compound* keys are not supported by the *Active Record framework of Ruby* and difficult to override.

In order to allow further flexibility for the *Active Record pattern*, one solution is to introduce an intermediary abstract class. The intermediary class generalizes each domain class, encapsulating the database access details. This intermediary class can then specialize the *ActiveRecord* base class, that keeps the common database logic, and acts as a *template method* class for the domain (GAMMA et al., 1994).

It is common to have the intermediary classes automatically generated by the framework, from models or configuration files for instance. Figure 2.3 presents the *Account* domain example in this scenario. The Cayenne framework is an example of this approach, in which the *ActiveRecord* class is called *CayenneDataObject*. The intermediary, underscored classes, contains the implementation of the mapping between the domain and the database accessing the persistence layer, including the properties

and its *accessors* methods. The domain classes, implemented by the developer, may then override these properties and implement the domain logic.

Figure 2.3: Inheritance to generated classes.



Both approaches to the *ActiveRecord* pattern led to concerns, about the high coupling to the persistence framework. The inheritance relationship imposes a *strong coupling* to the general classes (PRESSMAN, 2001), but there are other approaches to persistence, based on domain classes being *loosely coupled* to ORM framework classes, that can avoid this problem.

The *SQLAlchemy* persistence framework for the *Python* platform presents a hybrid example of the previous *Active Record* examples, and a loosely coupled solution based on the *Mediator* pattern. The developer can choose among inheritance from base class, configuring a *strong/tight coupling*, or the *loose coupled*, dynamic instantiation of mapping objects that link domain classes to table definition objects.

Figure 2.4: Instantiated mapping example for *SQLAlchemy* loose coupling approach.

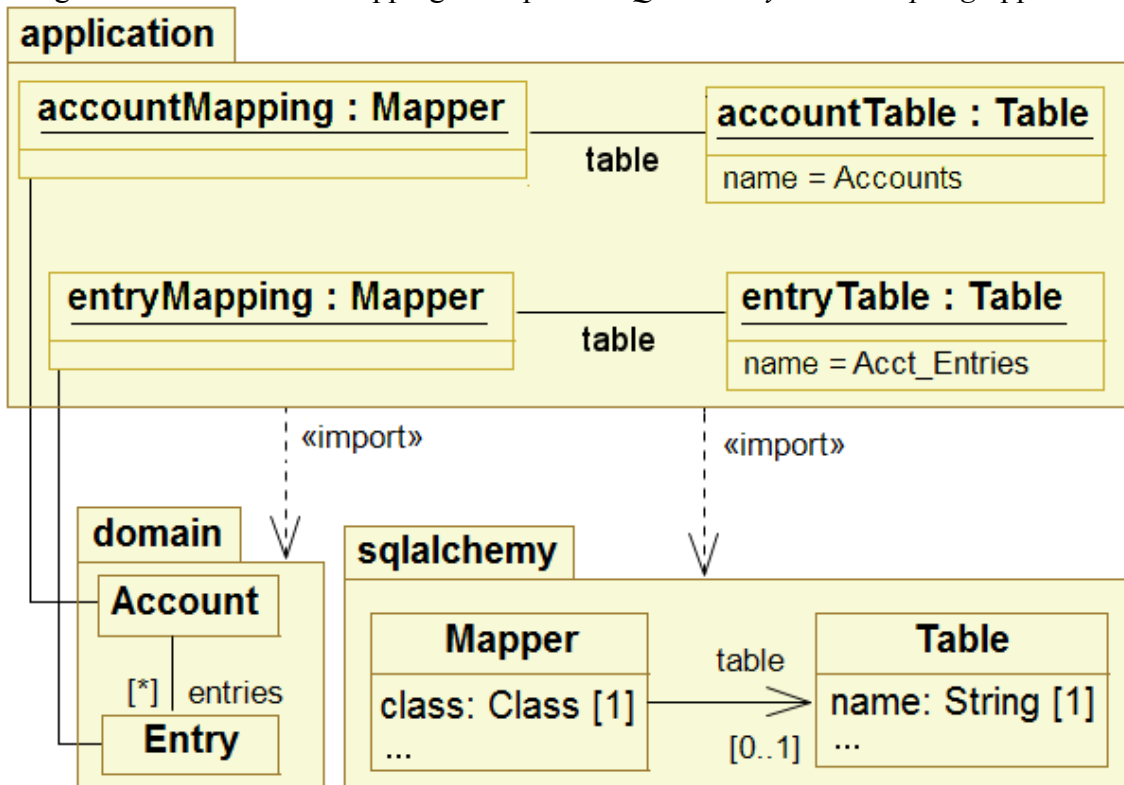
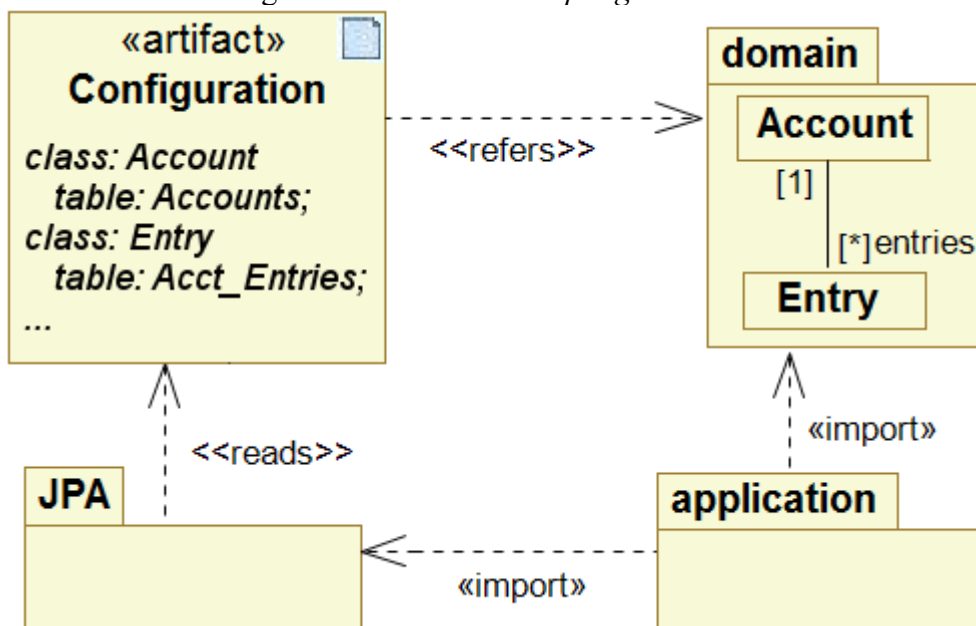


Figure 2.4 shows a simplified example of the transparent mapping, achieved in *SQLAlchemy*. The framework provides classes to specify the table (with attributes such as *name*, *column*, *column types*...) and mappings (*Mapper* class) between a *Table* and any class. In the application package, some class will instantiate the tables with its *metadata*, and the mappers binding each *Table* to the corresponding domain class. For example, the instance *accountMapping* is a *Mapper* that refers to the instance *accountTable* as its *Table* and to the domain class *Account* as its mapped class. The *accountTable* contains information about the database table, such as the name of the table being *Accounts*. This example is simplified, usually *accountTable* would contain all data necessary to create the table and its constraints. The *Entry* class is mapped in a similar way, by the *entryMapping* instance to the *entryTable*, with table named *Acct_Entries*. The framework can then use the *Mapper* instances to access the database and factory instances of the domain classes, based on application requests. This kind of loose coupling by instantiating mapping classes is named *Instantiated mapping*.

A slightly variation of the *Instantiated mapping* approach has become part of the *JPA* specification: instead of instantiating the mapping on some application class, as with *SQLAlchemy*, the mapping is done on independent files and/or code annotations, achieving a better SoC. This mapping artifact (file or annotation) is accessed by the framework, that internally builds the necessary metadata on memory. As in *SQLAlchemy*, the domain knows nothing about its persistence, and the persistence can be applied to any domain package, even without its original source code (Figure 2.5).

Figure 2.5: *JPA* loose coupling overview.



An additional advantage is that the *JPA* acts as an abstract factory (GAMMA et al., 1994) to one among various *JPA* frameworks, making it possible reduce or eliminate the *coupling* between the application and the implementing framework. In Java, domain classes with no *coupling* to the persistence framework are commonly referred as POJO (Plain Old Java Object) classes (FOWLER, 2000).

The MS Entity Framework uses an *ActiveRecord* coupled approach, like the one shown in Figure 2.2, combined with a configuration as its default solution (MICROSOFT, 2012a). The domain classes specialize the *EntityObject* abstract class (MICROSOFT, 2012b). Recently it was introduced the concept of POCO (Plain Old CLR Objects) classes which allows persistence ignorance on the domain classes (DERSTADT and VEGA, 2009).

Finally, *Mybatis* also employs *external configurations* for ORM. It achieves a *loose coupling* between the domain and the framework, encapsulating framework access at the application level.

2.2.1 Discussion

A framework that requires subclass coupling will introduce “alien” elements from the framework into the domain classes. If the platform does not support multiple inheritance, it will be impossible to have inheritance between persistent and transient application domain classes, given that each domain class already specializes its persistent counterpart and cannot inherit from another class.

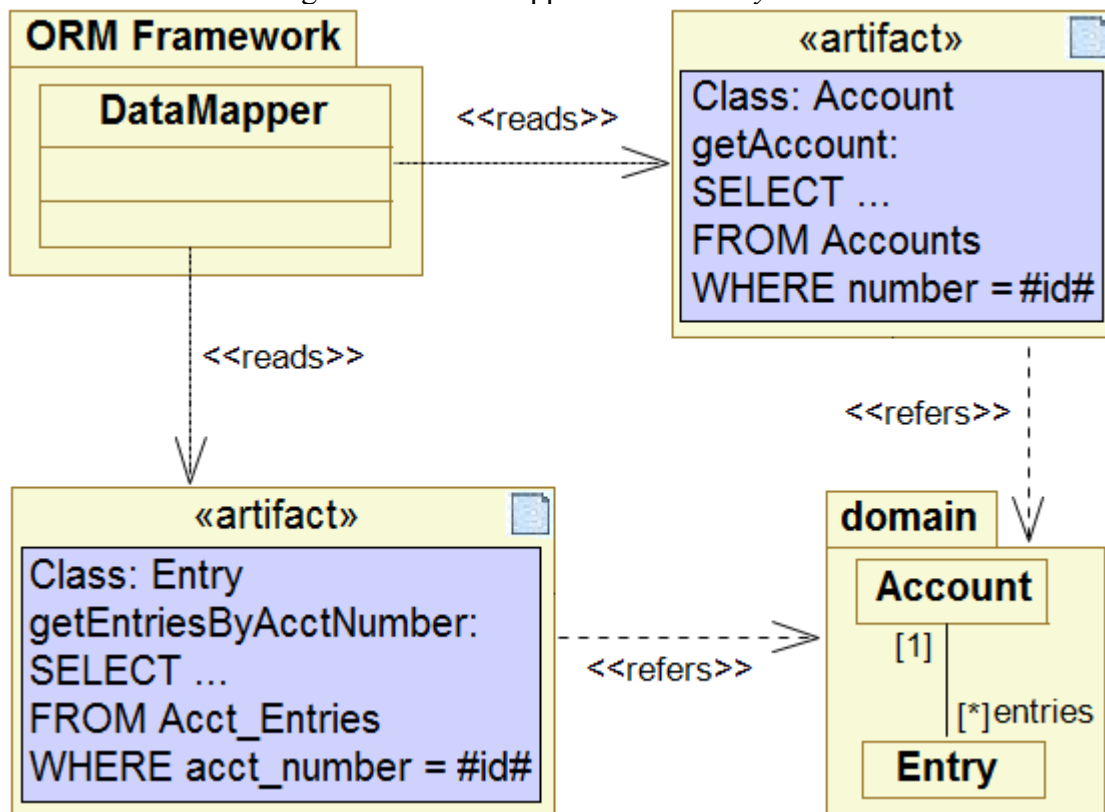
On the other hand, a framework that allows a loosely coupled domain model still have restrictions, mapping requirements and limitations. The naive assumption that *any* model can be persisted may lead to a domain model that cannot be implemented (or is unpractical) with the chosen framework or technology. If legacy databases are present (and they often are), chances are great that the design freedom on the domain model will be severely limited.

2.3 Mapping Type

An ORM framework has the responsibility of mapping data among domain objects and database tables. The mapping type criteria defines how a framework perform this mapping: the *Data Mapper* knows how to map the sets of data from SQL statements; the *Metadata Mapper* builds this SQL from *metadata* informed by the developer and/or extracted from the system. For example, this extraction may be performed by introspection and querying the database for *metadata*. Both mapping types were identified in previously published patterns with the same name (FOWLER, 2002).

Figure 2.6 illustrates how the *Data Mapper* works on environments such as the *MyBatis* framework. The mapper instance reads the configuration artifact that contains SQL statements prepared by the developer for each mapping situation of each class in the domain. For instance, the artifact *Account.xml* defines a query that is responsible for the construction of each *Account* instance. The artifact *Entry.xml* declares a query, that returns a set of *Entry* instances related to one *Account*. The developer must also provide *SQL* statements for the remaining CRUD operations.

The *Metadata Mapper* is represented by the *Mapper* class in Figure 2.4. Instead of storing developer written SQL statements, it stores information about the tables, columns and mapping options of the developer (FOWLER, 2002). The mapping options depends on the attribute and column types, including cardinality, length and precision, but may require some special configuration for Large Binary Objects (LOBs) and dates. This *metadata* is then used to dynamically assemble a SQL statement for each operation.

Figure 2.6: Data mapper such as in *MyBatis*.

A *Metadata Mapper* framework may have all *metadata* informed by the developer, by configuration artifacts and domain structure. Another approach to obtain *metadata* is to *extract* it from the database itself and combine with data informed by the developer. In such case, the database schema itself is a configuration artifact, complemented by the developer according the framework rules.

RAR uses the *metadata extraction* approach to perform all column mappings. The developers do not need to declare properties representing the database columns, all they need to do is to declare the class mapped to the table and the framework will, at run-time, query the database to obtain column *metadata* and dynamically provide properties to the classes. One drawback of this approach is that without database connection, the developer may not know what properties a persistent class have.

MyBatis is the only *Data Mapper* analyzed. All other frameworks, i.e. *SQLAlchemy*, *JPA*, *MS Entity Framework*, and *Cayenne* employ the *Metadata Mapper* approach.

2.3.1 Mapping Classes to Many Tables

Metadata mapper frameworks usually allows the mapping of one class to multiple tables. *JPA*, *Entity Framework*, *SQLAlchemy*, and *RAR* have mechanisms to map a class to more than one table, offering some mechanism to resolve the persistence of the instances. *Data mappers*, such as *MyBatis*, can easily be mapped to many tables due to its query flexibility.

JPA, *Entity Framework*, and *SQLAlchemy* allows the definition of secondary tables, joined by a common PK. The framework retrieves the data by performing inner joins,

and persist data by issuing inserts, updates, and deletes for each table in the mapping. *JPA* did not forbid implementing frameworks of improving secondary table support beyond this limitation, and in fact Hibernate allows the definition of the SQL statements that perform the joins and database changes (RED HAT MIDDLEWARE, 2014).

RAR allows secondary tables by using nested attributes (RUBYONRAILS.ORG, 2014). Differently from the other ORM frameworks, the secondary tables must first be defined as classes, with the necessary associative mappings. Therefore, the *ActiveRecord* framework do not hide the secondary tables from the domain as the above mentioned frameworks.

Another approach to map classes to multiple tables is the definition of entities over updatable views. This solution moves the mapping implementation to the database as stored procedures, views, and/or triggers. This approach gives greater flexibility, but hides the mapping inside the database.

A persistent class can often be defined over an arbitrary query, as a view can be defined by any valid SQL query, as long as it is read only. But the usual benefit of defining a domain class is its persistence. The problem of detecting what queries imposes a read-only restriction to the persistent class is similar to the problem of translating updates on views (DAYAL and BERNSTEIN, 1982; KELLER, 1985). Ultimately, the *ideal* mapping implementation would have to translate the operations affecting the tables, and these operations should be side-effect free: a change on one instance cannot affect any other instance in memory, or stored in the database.

2.3.2 Discussion

The *Data Mapper* approach requires the specification of SQL statements for each CRUD operation on each domain object. These statements are the mapping between the domain and the database, becoming sensitive to changes in both ends and dependent to the chosen database platform.

The *Metadata mapping* approach requires the specification of equivalence between domain classes and database tables, what sometimes may be difficult to achieve in legacy systems. The more resourceful is the *Metadata Mapper*, more freedom the domain model will have from the database and/or vice versa. Transferring the SQL responsibility to the framework may impact the performance and this may influence the design and require careful parametrization of the mapping. A framework may allow that part of the mapping specification is done through a *Data Mapper* approach.

One characteristic of the *Data Mapper* pattern is the higher mapping flexibility: by treating each case by individual hand crafted SQL statements, the domain model can be very distinct from the original database model (FOWLER, 2002). On the other hand, the *Metadata Mapper* encapsulates the database access, avoiding the database vendor “lock in” and its configuration is less repetitive and pretty much automatic when domain objects are similar to the tables.

2.4 Model-based Mapping

ORM frameworks often include a set of tools to specify the system, manage configuration artifacts, generate pieces of code and/or data modeling. Nevertheless, frameworks may rely on such tools to construct an abstract logical model ahead of

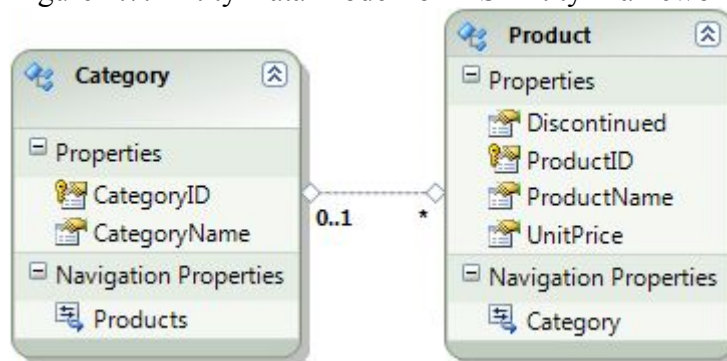
implementation, or work straight with implementation artifacts without any previous specific abstract model. *Model based* ORM involves a language with higher abstraction level (usually a visual one) to specify the two natures of entities: persistence, as a relational table, and behavior, as an OO class. This modeling language can actually represent elements from database, application and all the mappings required to overcome the IMP.

The mapping model is the primary input for any future change in the configuration, database structure and/or domain classes. Usually there are tools that can perform reverse and forward engineering between model, database and domain classes by Match and ModelGen operations, according to the model management approach (BERNSTEIN and MELNIK, 2007). The match operation tries to identify the mappings between elements from two meta-models, while the ModelGen generates elements from one model to another (such as generating DML or application code).

The Entity Data Model (EDM) is a *model based* mapping tool for the Entity Framework (ADYA et al., 2007; MICROSOFT, 2012a), influenced by the model management approach. The mapping starts with a model (with an ER logical inspired notation) that can be created from scratch, or by reverse engineering a database. This conceptual model is extended with visual elements to identify ORM patterns supported by the tool, such as navigation properties for relationships. Finally, this model is used to generate source code and configuration, by a template mechanism, that implements the domain objects and its mappings.

Figure 2.7 presents an example of EDM. The *Product* entity has a many-to-one relationship to *Category*, specified by the navigation property *Category*. The *Category* has a reverse navigation property named *Products*, that contains a collection of products.

Figure 2.7: Entity Data Model for MS Entity Framework



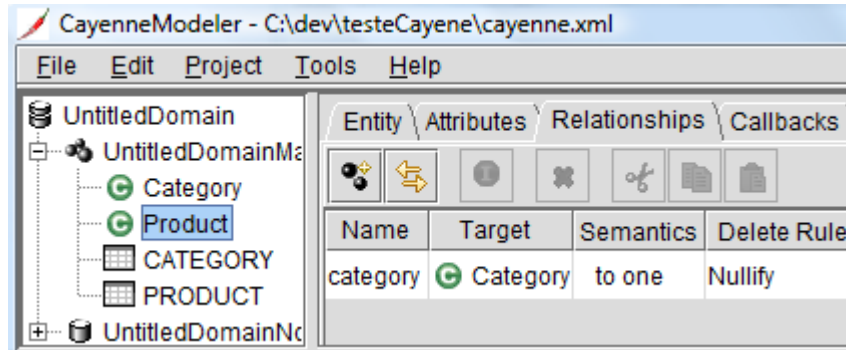
Source: (SNEED, 2012).

The disadvantage of using an ER like model (such as EDM) as the source for generating the domain model is the absence of visual behavior specification for the entity classes. Even when dealing with code and mappings, currently it is only possible to define methods that call stored procedures outside the model. *Operations* on the application side are not represented on the models.

The *Cayenne* is another example of *model based* tool that generates the source code from a visual model. Instead of one model, *Cayenne* requires the specification of two models. The first is a data model and the last is a class model bound to this data model.

Figure 2.8 presents an example of *Cayenne* model for a similar *Product x Category* model. The “C” icon represents the classes (*ObjEntities*) and the boxed icon represents tables (*DbEntities*). The modeler allows the specification of how each *ObjEntity* is mapped to one or more *DbEntity*. The model itself has a “Property explorer” visual representation, far away from the ER visual language.

Figure 2.8: *Cayenne* models domain and database elements separated.



Although the *JPA* specification was not conceived as a *model based* mapping framework, nothing prevents independent vendors from building tools for modeling ORM. The Hibernate Tools project (RED HAT MIDDLEWARE, 2014) provides a graphical view of the mappings, along with other development tools. The key point here is that such tools are not intended to design, but more focused on ease the comprehension of the (eventually huge) mappings.

2.4.1 Discussion

Model based ORM frameworks are somewhat new and clearly did not encompass all design choices for mapping, with limitations on behavior specification and a lack of connection to the transient components of the system.

EDM option for ER models naturally limits behavior specification, better expressed with class models. It is somewhat unclear how manual behavior development and code generation should work together without constant overriding, if by customizing the generator (that may require maintenance when updating the framework) or by moving any behavior out of the entity classes (what feels like a procedural solution).

The separation of class and table concepts on *Cayenne*, potentially duplicates the number of elements that the developer have to deal with (elements that in a conceptual viewpoint represent the same thing). Its modeling tool is much more a visual editor than a notation itself, leaving most of mapping information hidden behind wizards and menus, instead of graphic displaying its details. Moreover, it has the same limitation of the EDM regarding the lack of *operations* specification.

A *model based* solution, with support to mappings that are sensitive to changes on both sides of the ORM artifacts, still seems to be missing and not being offered by any of the tools available for the researched frameworks.

2.5 Identity

The PK is the minimal subset of columns that uniquely identifies a row in a database table. The PK may be composed by meaningful or meaningless information to the

developer viewpoint, and it is usually immutable. PKs can be referenced in other tables by Foreign Keys (FKs).

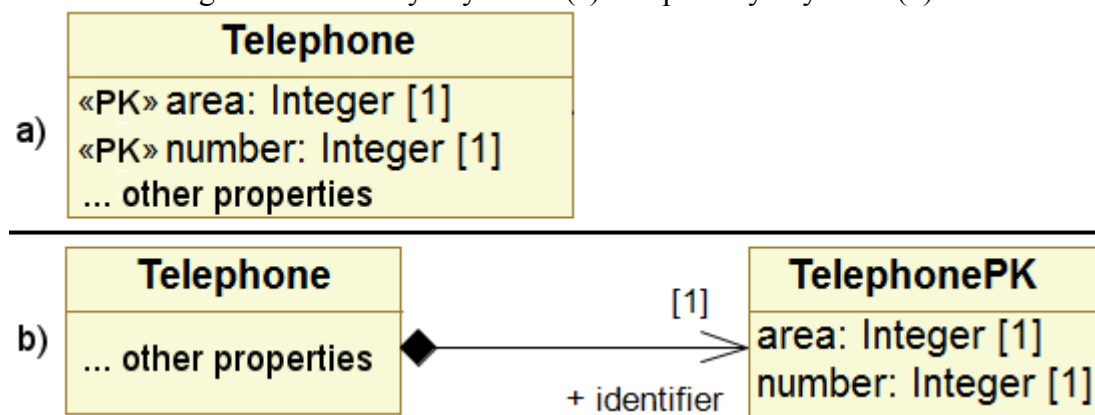
The *identity* problem refers to the mapping of the PK concept to the domain object. Objects usually do not need a declarative *identity* because they are internally identified by their memory position. From an OO viewpoint, if an object is serialized to an external system and later reconstructed to the same state, it is another instance and therefore a distinct object. Non standard equality and guarantee of uniqueness are behaviors that must be implemented on the class, such as the *equals* operation on the Java platform (GOSLING et al., 2005).

The *identity* problem was covered by the *Identity Field* pattern (FOWLER, 2002). The following characteristics were identified based on the implementation of this pattern by the ORM frameworks:

- *Identity Complexity*: can be of *Single* or *Compound* complexity: *Single* identities are formed by a single column; *Compound* identities by a set of columns. *Compound* keys are often used to implement weak entities.
- *Identity Uniqueness scope*: can be of *Table* or *Database uniqueness*. Concerns if the database is designed, to have PKs that are unique in the context of the *table*, or the entire *database*. *Table uniqueness* is the most common situation on legacy databases.
- *Identity Assignment*: can be of *Auto-generation*, *Counter* or *User Assigned*. A database can offer different resources to assign the PK: With *auto-generation* the database assigns a key for each inserted row; With *counter assignment*, the developer obtains a new key from a named unique *counter* (also called *sequence generator*), that may be shared by several tables; and with *user assignment* the application is responsible of assigning the key, asking the user for a meaningful key or generating the key by a client side solution.

ORM frameworks may not support all combinations of the above characteristics. Some ORM frameworks only support single key mappings, such as the *RAR* (HEINEMEIER HANSSON, 2012). Some characteristics are supported by all frameworks, such as *single complexity* and *table uniqueness*.

Figure 2.9: Primary key fields (a) and primary key class (b).



Composite keys may be represented by a separated *PK class*, or by *PK fields* identified within the class. On Figure 2.9, diagram *a*) shows the composite PK for *Telephone* as two properties, *area* and *number*, directly containing the columns values; and diagram *b*) depicts the PK as an *identifier* relationship, between *Telephone* and *TelephonePK* classes. Some frameworks, such as the ones that follow the *JPA* specification, allow both representations, although the definition of PK classes may be required even if the identified fields approach is used (DEMICHIEL, 2013).

Key assignment is a tricky problem to the ORM. A domain object is first created in memory and then it *can* be persisted on database. If the key is *user assigned*, the ORM must ensure that a key is provided before that object is persisted. On the other hand, if the key is generated by the database, it must not be assigned by the user. If it is an *auto-generation* field, the ORM must implement some way of capturing the generated value, to store in the object for later updates. When a *counter* is employed, the *counter* specification must be informed in the mapping, or obtained from some default naming convention.

The *user assigned identity* may be automatically generated in the ORM level. The most common techniques are based upon Globally Unique Identifier (GUID) *generation*, key tables and table scans. A framework may allow the user to create customized generations, combining different techniques.

Complexity and *assignment* are affected by the chosen *uniqueness* scope. *Composite* and *auto-generated* identities are usually related to *table* scope, while *GUID* and *single* identities are the usual combination for *database* scope. The ORM may treat all scoping as *table*, offer some facility to distinguish *database* or *table* scoping, or just require *database* scoping. By treating all scoping as *table*, the developer can still implement some sort of *database* scoping, if the ORM supports *user assignment* to generate GUIDs for instance. Conversely, requiring *database* scope turns out to be an impediment for database schemas with *table* scoping and access by legacy systems.

2.5.1 Discussion

PK mapping is often forgotten in the OO project stage, spanning unpredicted PK classes in the implementation stage. Another common problem is the absence of the PK attributes in the domain classes, since they are required only to model the database. The *assignment*, and *uniqueness* of scope, may impact in the inheritance support of the framework. Single and *compound* keys will impact the foreign key support.

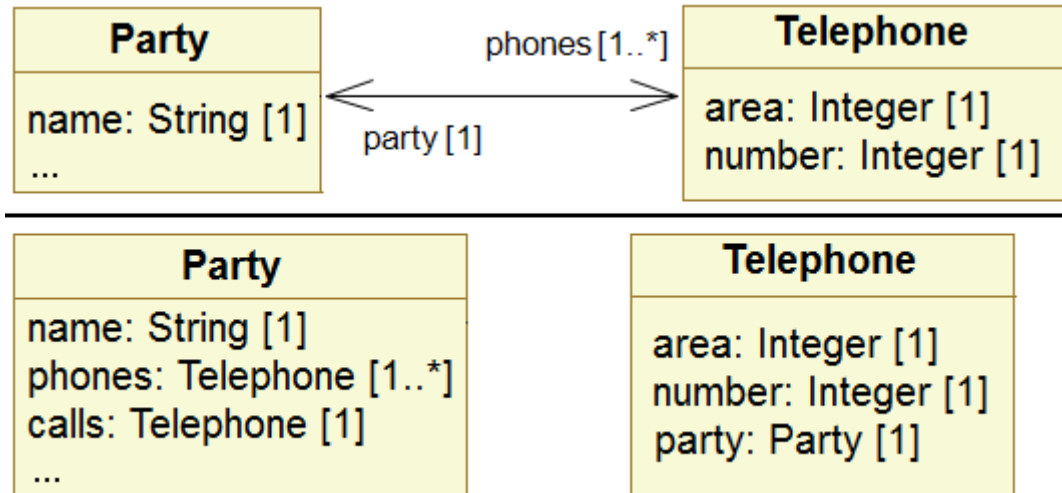
If a *metadata mapper* is employed it will require metadata about how the PK will be represented. Conversely, the *data mapper* will require the presence of "find-by-primary-key" factory methods for most domain classes.

2.6 Foreign Key

The *Foreign Key Mapping* pattern describes the common solutions for representing database relationships in classes (and vice verse) for the *one-to* and *to-one* cases (FOWLER, 2002; KELLER, 1997). One of the most important features of RDBs is the referential integrity, a mechanism that guarantees coherent relationships between rows in different tables (ELMASRI and NAVATHE, 2003; GARCIA-MOLINA, ULLMAN and WIDOM, 2008). One row references the other by having a FK, a subset of its

columns that references the PK columns (sometimes an Alternate Key) of the other table. The database must either ensure that the FK is pointing to some row that exists in the referenced table, or that it has NULL values in all of its columns.

Figure 2.10: Association relationship between classes.



In OO systems, classes may reference other classes by variables, what denotes a (usually transient) dependency relationship. An association (or structural) relationship between two classes A and B typically happens when there is an instance variable on A with type being B, or a collection of B elements (BERLER et al., 2000). This relationship is often persistent, and may be unidirectional or bidirectional, the later typically requiring another instance variable on B referencing A.

Figure 2.10 represents two diagrams representing *Party*, a pattern to represent persons and organizations, and *Telephone* (FOWLER, 1996). The upper diagram is an UML representation of a bidirectional relationship between *Party* and *Telephone*. However, in implementation terms, the relationship is only implied by instance variables *phones* and *party*, as shown in the lower diagram. It is not clear, without looking at the upper diagram, if *phones* and *party* forms a bidirectional relationship or two independent unidirectional relationships. If a second variable references the same destination class (*calls*), there is no structural information, on the classes, that indicates which property is in the opposite side of the relationship. There is nothing like a FK constraint in OO programming languages, although it can be implemented, by encapsulation of instance variables access with operations.

On RDBs the FK is placed in one table, depending on the cardinality of the relationship. The row with the FK can reference at most one row in the other table by this FK. Rows in the table referenced by the FK, often named master table, can be referenced by zero or more rows within the same constraint rule. Nevertheless, database relationships are considered bidirectional because the SQL query language allows to access both the master row knowing the detail FK, or the detail rows by knowing its master PK.

The *Foreign Key Mapping* pattern simplest version is to have a field, in the class that owns the foreign key, that references an instance of the master class. This mapping

is known as *many-to-one*. Sometimes the inverse is a more significant design, such as the party referencing its phones, in such case the field is a collection, owned by the master class, and is named *one-to-many*. *One-to-one* relationships differ very little from *many-to-one*, regarding the implementation in unidirectional situations like that.

Nevertheless, if both classes references each other, the *foreign key mapping* is named bidirectional, and it must deal with the problem of keeping both sides of the relationship updated. If one *Telephone* is added to the *phones* of *Party*, the *Telephone* itself must have its *party* reference updated, and if the *Telephone* is assigned to a different *Party*, both parties must be updated: the old *Party* collection of *phones* must have this *Telephone* removed and added to the *phones* collection of the new *Party*.

The simplest case of mapping unidirectional *many-to-one* is supported by all studied ORM frameworks. The instance variable *identity* is mapped to a FK, automatically, if the name matches the database structures. Additional mapping information is needed, when the variable name did not match with the column name, or when the identifier is *compound*.

The *bidirectional* case must support some mechanism to specify what is the opposite instance variable, as shown in the Figure 2.10 *party-phones* relationship. To represent the *to many* side, the instance variable requires some collection instance, with variable size support, such as the Java Collection framework. Mechanisms such as Templates, or Generics, allow the framework to type collections.

Some languages do not support the typing of collections, leading to some mechanism to specify the target of the relationship. One solution, employed by the *RAR*, is to match relationships by the variable name. If this match is not possible, the relationship can still be defined, by overriding the standard active record implementation. Most ORM frameworks have some customization approach using reflection, instantiation (*SQL Alchemy*), or *external configuration* (*JPA*, *Entity Framework*).

The *one-to-many* unidirectional relationship is one exception scenario. By using the *Foreign Key* pattern, the key is stored in the many side, that should supposedly not know about the relationship. This can be handled, at some ORM frameworks, by implementing *one-to-many* with the *association table pattern*, avoiding an FK in the “wrong” direction.

For the *one-to-many* direction of the relationship, some frameworks work with read only collections and add/remove methods for the elements, while others allow the direct manipulation of collections, later reflecting these changes as updates in the referring object. *Cayenne*, for instance, requires a method to add elements to the collection, while the *JPA* allows direct manipulation of collections. However, for *bidirectional* relationships, changes on one side may not be automatically reflected in the opposite side, hence add/remove methods are a good practice to encapsulate such details, when this is the case.

2.6.1 Fetch Strategy

The *fetch strategy* determines what part of a persistent graph should be retrieved (BAUER and KING, 2004). For the example in Figure 2.10, the *fetch strategy*, specified for the *phones* of a *Party*, would specify whether these phones should be loaded from

the database along with that instance, or not. Conversely, when a *Phone* instance is loaded from the database, loading the *Party*, into the *party* instance variable, is also subject to the *fetch strategy*, specified for this instance variable.

The *Lazy Load* pattern offers a flexible solution to the fetch strategy, by deferring the loading, to the first moment the information is requested by the system (FOWLER, 2002). For collections, the most common solution, is to provide a transparent wrapper around the collection, that checks if the collection was initialized, and load it only when needed.

In the *to zero/one* scenario, the solution may be a virtual *proxy*, an instance of the class that actually is a pointer to the real object, retrieved on demand from the database (GAMMA et al., 1994). The decision about using virtual proxies appears to be transparent, but it may end up neglecting polymorphism. For instance, imagine the following scenario: the *Party* class has a specialization called *Person*. If the system retrieves a *Telephone* with lazy loading strategy, a *Party* virtual *proxy* will be instantiated despite the possibility of the *party* being a *Person*. If the system access the *party* variable, the related *Person* will be instantiated and loaded, but, the already instantiated *Party proxy*, will be wrapped around this *Person* instance.

For data mappers such as *MyBatis*, the lazy loading is controlled by the *nesting type* used in the configuration. In the previous example, the *Party* class might be mapped to an outer join with *Telephone* and a nested *resultset*; or, alternatively, mapped by one query to *Party* and a nested “on demand” query for *Telephone*. All frameworks in this survey, present the possibility of *fetch strategy configuration*, but only *JPA* and *SQLAlchemy* allow the configuration of *proxies* for relationships.

2.6.2 Discussion

Relationships are best represented in visual models, rather than code or SQL, in which the reader must interpret statements, to discover the nature of one relationship. For ORM frameworks, some mapping are often needed, to specify elements such as relationship direction, type or collections, that are connected by foreign keys.

Performance is a concern in relationship mapping. The *fetch strategy* plays an important role in ORM design. Lazy fetch in to-zero/one relationships has different consequences on each ORM framework, leading to the use of proxies and impacting on the way that the domain objects behave.

2.7 Association table

The *Association Table Mapping* pattern describes the common solutions to represent OO *many-to-many* relationships in databases (FOWLER, 2002; KELLER, 1997). RDBs do not deal transparently with *many-to-many* relationships. These relationships must be implemented by a third *association table*, containing mandatory *many-to-one* relationships to both related tables.

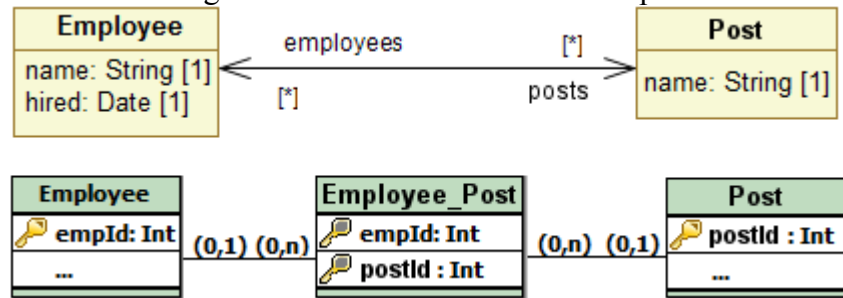
The basic idea, behind *Association Table Mapping*, is to use a link table to store the association. This table has only the foreign key IDs, for the two tables that are linked together, and it has one row, for each pair of associated objects. The link table has no corresponding in-memory object. As a result, it has no ID and its PK, if it exists, is the compound of the two PKs of the tables that are associated.

Figure 2.11 shows an example of *many-to-many* table mapping, between classes *Employee* and *Post* of the *Post* pattern. The *Employee* class has the collection field *posts* referencing *Post* objects. *Employee* and *Post* are stored in the tables with the same names, and the *employees-posts* collection is stored in the *Employee_Post* association table. If a *Post* is added into the collection, a row must be inserted in the *Employee_Post* table, with corresponding *empId* and *postId* values; if it is removed from the collection, this row is deleted in the same way. The identification of this table is usually done by joining the PKs, from the associated tables.

Tables such as *Employee_Post*, present in the database model, may not be of interest from an OO design standpoint. In OO language models, *many-to-many* relationships can be transparently mapped into object references, as long as the underlying relationship table does not contain information by itself. For example, an *employees-posts* association may be transparently mapped, as long as it does not have other information, such as *experience* in years, or *type of degree*. When the relationship has attributes, it usually ends up being a *Class*, represented in UML by an *Association Class* (OMG, 2011b).

The *Employee_Post* table is not represented in the class model, but ORM frameworks usually have to store *metadata* about this table, in order to implement the relationship. Some tools may assume the table name from the class names, but usually the developer supply the name of the *association table*.

Figure 2.11: Association table example.



For bidirectional relationships, the ORM must deal with two instance variables (*employees* and *posts* on Figure 2.11) that represent dependent collections. If a *Post* is added or removed from the *posts* collection of an *employee*, this *employee* should be added/removed from the *employees* collection of this *Post* instance.

Most persistence frameworks are able to deal with transparent association tables, but some of them may have restrictions. For instance, the *MS Entity framework* assumes that the *association table* has a PK consisting of the two FKs and is unable to recognize any other column in this table. An *association table* with a unique identifier is therefore unsupported. The *Cayenne* Framework allows some flexibility of the transparent *association table* for read-only relationships. *JPA* allows to represent the relationship as a *Map* containing the associative attribute(s), such as the *experience* level.

Transparent *association tables* are not directly supported by *MyBatis*. Nevertheless, the mapping language allows the definition of collections over nested results or nested queries that can emulate the *association table* without the existence of an association class to explicitly map the *association table*.

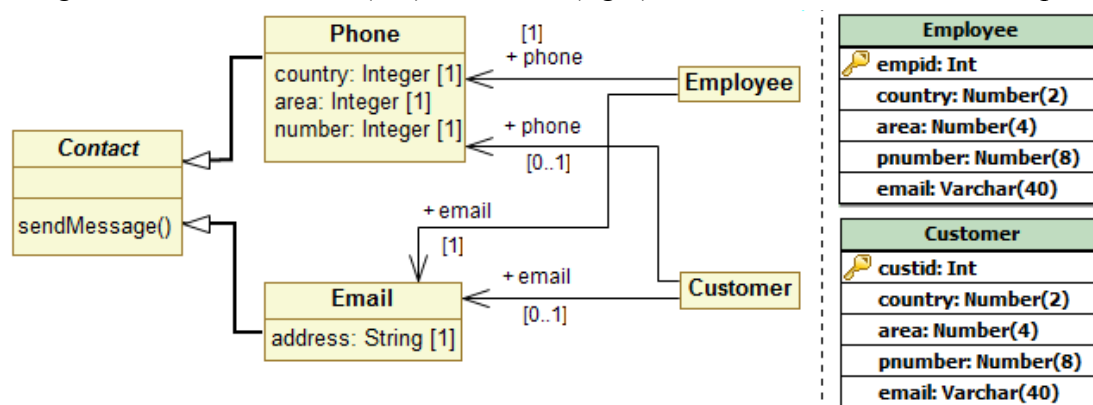
2.7.1 Discussion

Transparent association tables are a great abstraction for modeling systems, and may simplify the implementation of domain classes, by removing association classes required only to emulate *many-to-many* relationships. However, support for *association tables* is often limited in ORM frameworks, while it is common to find several variations in legacy databases that are unsupported by *transparent association tables*, regarding, for instance, surrogate keys in *association tables*.

2.8 Embedded Values Support

Not all classes in the OO design model will make sense as database tables. Some classes may represent persistent data only when related to another persistent class. In cases where two classes relate in a *one-to-zero/one* basis, the dependent classes are the best candidates to be stored within the owner table, by the application of the *Embedded Value* or *Single-Table Aggregation* patterns (FOWLER, 2002; KELLER, 1997).

Figure 2.12: UML model (left) and tables (right) of the *Association table* example.



Imagine the situation where the system deals with contacts for its employees and customers, within the *Accountability* domain (Figure 2.12). A *Contact* may be an *Email* or a *Phone* with SMS support. Both *Employee* and *Customer* can have one *Phone* and one *Email* contact, but the customer may not inform the contact information. To reuse and encapsulate the contact behavior, phone and email are separated classes, but from the database viewpoint these are simple attributes of the parent tables.

Phone and *Email* can be seen as regular value objects (like *String* or *Date*) and are only persisted when related to an owner instance, such as employee or customer. Usually the relationship is unidirectional from the owner to the *embedded* class, what denotes an “*attribute of*” relationship (OMG, 2011b). This relationship may be a composition, although it is common to see the owner reassignment allowed by the persistence framework.

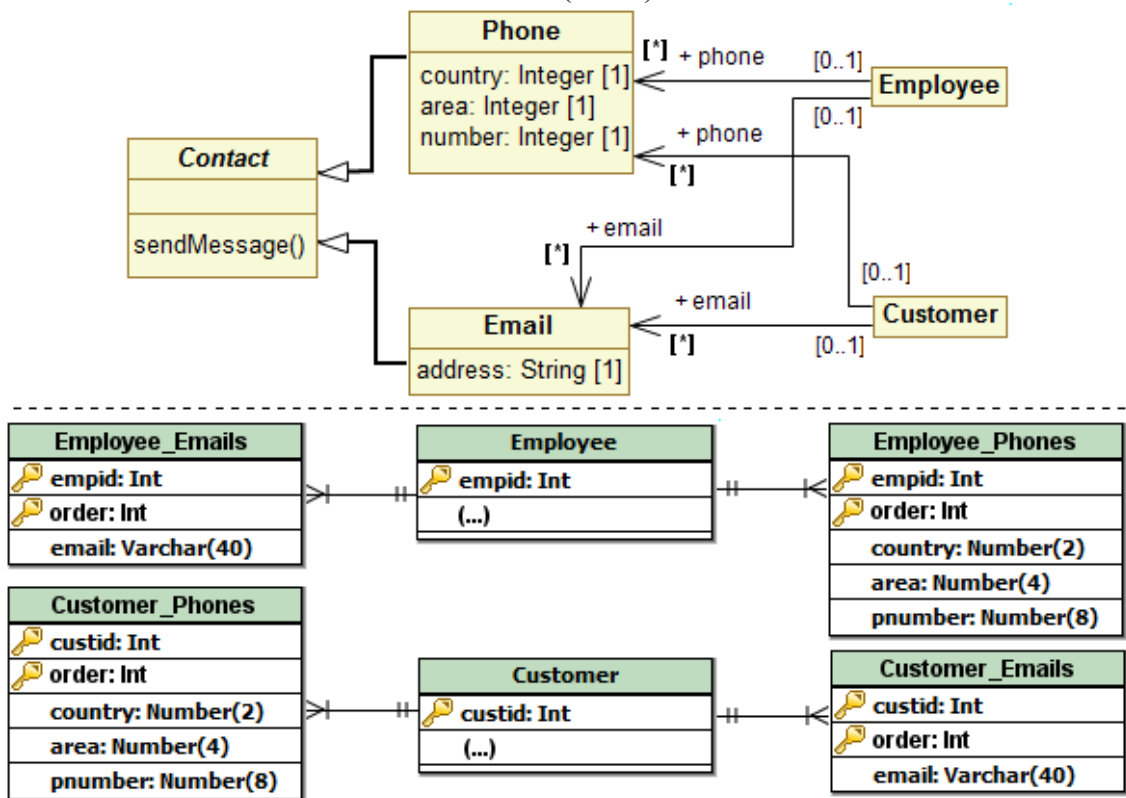
Cayenne and *JPA* have support for *embeddable* classes. In the *MS Entity Framework* the *Embedded Value* is named *Complex Type*; in the *SQL Alchemy* it is named *Composite Column*; and in *RAR* its named *Aggregated Value Object*. *MyBatis* does not need to distinguish embeddable classes, because for each class, the developer have to provide the SQL for persistence.

Some frameworks, such as *JPA* and *MS Entity Framework*, allow the definition of standard mappings for embeddable classes, such as the preferred database types. Others, such as *RAR* and *SQL Alchemy*, do not support default mappings, requiring a specific mapping of each reference to an embeddable class.

Another pattern that falls under this *embeddable* criteria is the *Dependent mapping*. In fact, as a pattern, *Dependent mapping* is a generalization of *embedment*, that deals with classes that are persistent due to the relationship to other persistent classes, regardless of the nature or cardinality of the association (FOWLER, 2002).

Figure 2.13 revisits the *embeddable* example replacing the *one-to-zero/one* with *one-to-many* relationships between *Employee/Customer* and *Phone/Email* contacts. In order to store more than one associated element, a new table is required, as what happens with the *Foreign Key* pattern. The key difference here is, that the table that stores the *phones* of *Customer*, is not the same table that stores the *phones* of *Employees*. The table that stores a dependent mapping should do it exclusively for one owner.

Figure 2.13: Dependent mapping pattern example: class model (upper) and database model (lower).



In the example of Figure 2.13, the dependent tables are designed with composite keys with one column referencing the owner table, and the other registering the *order* of the element in the relationship. The dependent relationship simplify the persistence changes, in a collection of dependents that can all be safely deleted, and reinserted, when the owner is persisted.

The *Dependent Mapping pattern* described by Fowler states that the owner class is responsible to the persistence of the owned objects. Considering ORM tools taking over this work, or with *metadata* mappings performing general persistence, this pattern can be used to describe *element collections* such as collections of *embeddable objects* and basic types.

The support of more generic dependent mapping is not yet widely supported by ORM solutions. *JPA* supports collections of embeddable objects (named *element collections*), mapping such collections into a separate table that refers back to one owner entity table. The element collection mapping allows the mapping of *one-to-many* dependent mappings.

2.8.1 Discussion

Embedded values are a valuable asset to bridge persistent and transient elements of the domain model. Reuse and encapsulation are great advantages of OO, and the forces that lead OO design to break a class in two or more pieces are often contrary to the database normalization forces that put that information together.

The possibility of defining default mappings for the *embedded* classes can facilitate the design, by reducing repetitive mapping. Information such as field types and lengths will probably be the same for all *embedded* mappings that target to the same class.

2.9 Inheritance Mapping

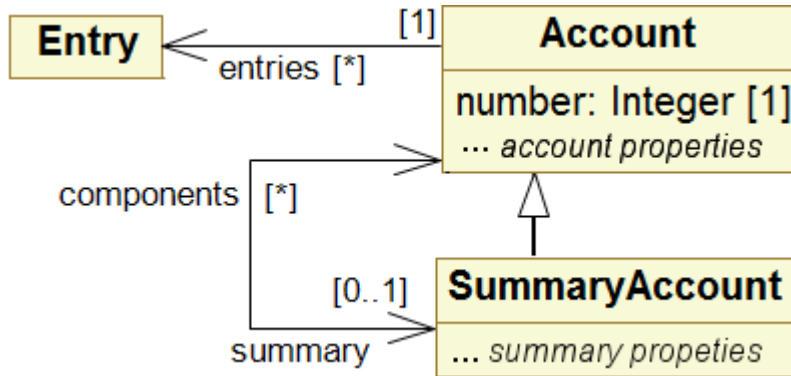
Inheritance is not supported in pure RDB models. It must somehow be emulated with optional fields, discriminators and/or table joins. There are three preferred ways, identified as patterns, for mapping inheritance (FOWLER, 2002; KELLER, 1997).

- “*single-table*” or “one inheritance tree one table”, meaning that one table contains all possible attributes of the class tree.
- “*Class-table*”, “Vertical inheritance”, or “one class one table”. Each class is mapped to one table containing only the attributes for that class.
- “*Concrete-table*”, “Horizontal inheritance”, or “one inheritance path one table”. Each concrete class is mapped to one table, but each table contains the sum of all attributes of the class hierarchy.

The choice of patterns depends on implementation and platform specific issues. The balance of performance forces, such as *update and write access* versus *polymorphic read*, may be more determinant than maintenance and ease of writing queries, in the decision about what pattern should be followed. Resources of the database system, such as NULL compression that saves space for the *single-table* pattern, must be also taken into account before deciding the best strategy (KELLER, 1997).

The *Account* example is expanded in Figure 2.14 to illustrate a specialization of accounts, named *SummaryAccount*, which represent accounts that are composed by other accounts. *SummaryAccount* inherits properties from *Account*, such as *name*, and adds its own properties. The *components* relationship may connect a *SummaryAccount* to any *Account*, including another *SummaryAccount*. The *Entry* class has a polymorphic association with the *Account* class: it may refer to one *Account* or to one *SummaryAccount* object.

Figure 2.14: *Inheritance* example for *Account*.



A *single-table* pattern solution for the *components* model is shown in Figure 2.15. The *Accounts* table contains the sum of all attributes of the hierarchy and the sum of all associations. The *type* attribute (marked with *) was added to discriminate between an *Account* and a *SummaryAccount* row, although it could be replaced by using the *synthetic object identifier* pattern (KELLER, 1997). Mandatory associations are mapped as optional associations, but our example association was already optional. Nevertheless, the standard database integrity mechanisms cannot prevent an *Account*, from being referenced by another *Account* which is not a *SummaryAccount*. It would be necessary to check the discriminator value, in order to determine that such constraint was violated.

Figure 2.15: *Single-table* approach for *Account* example.

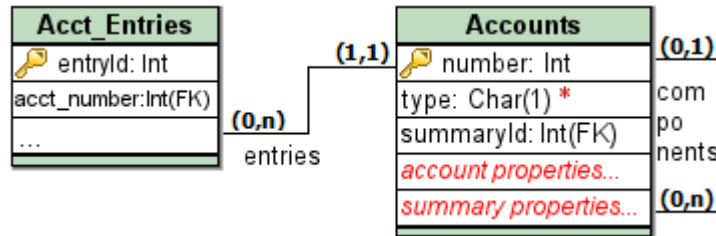
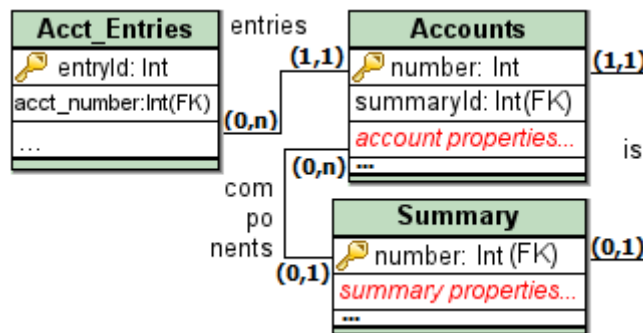


Figure 2.16: *Class-table* approach for *Account* example.

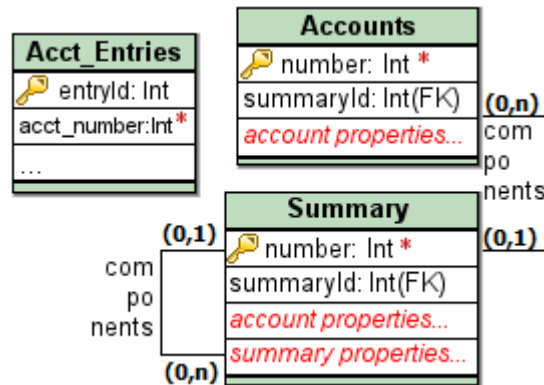


A *class-table* pattern mapping is shown in Figure 2.16. *Accounts* and *Summary* represent *Account* and *SummaryAccount* classes, with a similar attribute/association distribution. To retrieve a *SummaryAccount*, the system must join tables *Summary* and *Accounts*, by its common PK. The association is now placed in the *Summary*, allowing

the database to enforce the *components* association. One visible downfall for this approach is the cost of joining *Accounts* and *Summary*, to obtain a single *SummaryAccount* instance.

A *concrete-table* pattern mapping is shown in Figure 2.17. Each concrete class of the hierarchy is mapped to one table, but the retrieval must be done without the requirement of joining tables. The consequence is that specialization tables will contain columns for the inherited attributes.

Figure 2.17: Concrete-table approach for *Account* example.



Even without the foreign key constraint, in the specialized concrete tables, it is highly desirable that the *number* (marked with ***) be unique for the entire hierarchy, because otherwise, it would be impossible to represent a polymorphic association to a super class. In the example, *Acct_Entries* refers to an *acct_number* that can be a row at *Accounts* or *Summary* tables. This kind of relationship cannot be enforced by a FK and should be enforced by the persistence layer.

Both *Account* and *SummaryAccount* can be components of a *SummaryAccount*, requiring two column references, to represent the *components* association of Figure 2.14. The *summaryId* column, of *Accounts*, can reference the *Summary* PK, but a *Summary* PK can also be referenced by a *summaryId* column, of another *Summary*. In the *concrete-table* strategy, it is not uncommon to have one association become two, or more, column references between tables.

Besides problems related to relationship mapping, the *concrete-table* has a severe performance problem for polymorphic queries (KELLER, 1997). If one needs to query the entire hierarchy, for a specific condition, an expensive UNION operation is issued to all tables.

Surely, this example does not explore all possible combinations of problems adapting each inheritance pattern to the other relationship, PK, FK, and so forth patterns. If, in this scenario, the *concrete-table* seems to be a bad choice, it may be turned into a good choice, depending on the relationships, the number of estimated account records, what classes are persistent in the class tree, and/or presence of a legacy database schema. Some flexibility, to change between each approach, is a valuable asset for this solution.

Most ORM frameworks have some kind of inheritance mechanism, and these three patterns are usually supported. *JPA*, *MS Entity framework* and *SQLAlchemy* let the

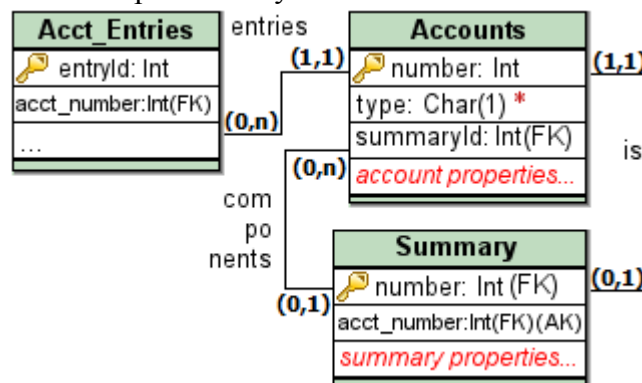
developer choose between the three patterns, although the *concrete-table* support is optional in *JPA*. *Cayenne* supports *single-table* and *class-table* (named *vertical inheritance*) but has the *concrete-table* (to be named *horizontal inheritance*) under development. The *RAR* implements only the *single-table* strategy, by introducing a discriminating column named *inheritance_column* (HEINEMEIER HANSSON, 2012).

MyBatis framework *data mapper* approach, requires the specification of the SQL expressions for each mapping, and offers the *discriminator* mapping, that uses one table column to discriminate the correct class of the hierarchy. The discriminator can be utilized to implement a *single-table* strategy, but it is possible to emulate the *class-table*, by manually providing adequate SQL expressions. However, the discriminator column is required for all “type select” situations.

The discriminator column may be required depending of ORM and pattern in use. With *JPA*, a discriminator is required for *single-table*, but optional for *class-table* inheritance. In *Ruby*, the discriminator may not exist for *single-table*, but it means that the instance type will not be automatically detected. It is sometimes possible to identify the record type in the *single-table* strategy, by the presence of null values, but it is not really supported by the researched tools, due to its poor performance.

SQL-Alchemy allows fine control on retrieving *class-table* objects, but requires a *discriminator* column in the root table. When retrieving *Account*, for instance, only the *Accounts* table may be queried, and the remaining attributes are loaded on demand, avoiding the expensive join. With the *discriminator*, the persistence framework can determine which type must be instantiated, without performing an outer join to each possible sub-type.

Figure 2.18: Independent keys in *class-table* inheritance example.



Another issue rises in the joining of inheritance tables. The MS Entity Framework requires that, the PK of each table, must be the same, and a FK to the master table (Figure 2.16). The *JPA* specification states that the PK should be specified only in the root entity, but at same time, allows the FK to the super-class to be redefined by subclasses, using the *PrimaryKeyJoinColumn* annotation. The effect of this redefinition, in *JPA*, may have unexpected consequences depending on the implementing framework, because it is not possible to declare in the mappings the PK of the subclass. The *SQL-Alchemy* explicitly allows user defined FK relationship between each class-table.

Figure 2.18 shows an example of *class-table* mapping with independent PK and a discriminator (marked with *) column. Flexibility in the FK *inheritance mapping* is important to map legacy database relationships as inheritance relationships, when the FK relationship is not in the PK, but an alternate key.

2.9.1 Discussion

Inheritance is one key feature of ORM, and the most difficult issue to deal with. The mapping strategy to be chosen is a decision that affects behavior, performance, and design limitations of the domain model. Some of these limitations are dependent of the ORM tool, and others are inherent to the strategy itself.

Mapping inheritance in legacy *schemas* may be difficult, because certain design conditions must be met to support inheritance, and these conditions change from tool to tool. Moreover, not all constraints can be enforced by the database for every strategy, and if there are other non ORM based applications, using the same schema, they must ensure these constraints are satisfied within their code.

2.10 Summary

Table 2.2 presents a summary, with each analyzed criterion, and its characteristics, related to each ORM framework analyzed over the previous sections. This summary is organized as criteria, subdivided by characteristics previously presented, and its relationship with each studied ORM framework.

The first criterion is the domain *coupling*, in which some frameworks offer *loose coupling* and others are tied by *inheritance coupling*. The entity framework is usually *not loose coupled*, although with some customization it is possible to design a loose coupled domain, hence its marked with *depends*. Another characteristic analyzed is if some of the mapping is done by *external configuration* files.

The *mapping* criterion is determinant for the mapping abstraction level. *Data Mappers* deal directly with SQL statements, therefore some of the criteria can be satisfied by emulating their mapping requirements. Entries in Table 2.2 marked with the *Emul.* abbreviation indicate these situations.

The *model* criterion is focused on model first tools, in which *entity class models* represent the mapping between the relational and OO artifacts. As show in Table 2.2, none of the tools are yet focused on the structural behavior specification of *operations*.

On the *identity* criterion, the PK representation as a field, or as an embeddable class, has limitations in some of the frameworks. In *JPA*, for example, when mapping a composite PK as fields in the class, the developer must also define a PK class with these PK fields. Also, support to table independent keys exists in all frameworks, and unique identifiers for databases can be achieved by application or user assigned keys.

Foreign keys, *transparent association tables*, and *embeddable values* all represent relationship criteria, the first two between persistent entities, and the third with transient reusable classes. *Mybatis* can emulate some of these constructs, as explained earlier.

Finally, on the inheritance criterion, *JPA* allows, but does not require, the *concrete-table* strategy, hence it is marked as implementation dependent (*Dep*). *MyBatis* supports

single inheritance, but may emulate other strategies through some additional mapping work.

Table 2.2: ORM Frameworks support for each proposed criterion.

<i>Criteria \ Frameworks</i>		<i>RAR</i>	<i>Cayenne</i>	<i>Entity Framework</i>	<i>JPA 2</i>	<i>MyBatis</i>	<i>SQL- Achemy</i>
Coupling	Loose	No	No	Depends	Yes	Yes	Yes
	External conf.	No	No	Yes	Yes	Yes	No
Mapping	Metadata Mapper	Yes	Yes	Yes	Yes	No	Yes
	Extract metadata	Yes	No	No	No	No	No
Model	Entity class	No	Yes	Yes	No	No	No
	Operations	No	No	No	No	No	No
Identity	Compound keys	No	Yes	Yes	Yes	Yes	Yes
	Table uniqueness	Yes	Yes	Yes	Yes	Yes	Yes
	Auto-generation	Yes	Yes	Yes	Yes	Yes	Yes
	Counter gen.	Yes	Yes	No	Yes	Yes	Yes
	Application gen.	No	Yes	Yes	Yes	Yes	Yes
	User assignment	Yes	Yes	Yes	Yes	Yes	Yes
	PK as Field(s)	Yes	Yes	Yes	Yes	Yes	Yes
	PK as Class	No	No	No	Yes	Yes	No
Foreign Key	Bidirectional	No	Yes	Yes	Yes	Emul	Yes
	Collection update	Yes	Yes	Yes	Yes	Emul	Yes
	Fetch config.	Yes	Yes	Yes	Yes	Yes	Yes
	Proxy option	No	No	No	Yes	No	Yes
Transparent Assoc. Table		Yes	Yes	Yes	Yes	Emul	Yes
Embed. Value	Map container	Yes	Yes	Yes	Yes	Emul	Yes
	Map aggregated	No	Yes	Yes	Yes	No	No
Inheritance	Single-table	Yes	Yes	Yes	Yes	Yes	Yes
	Class-table	No	Yes	Yes	Yes	Emul	Yes
	Concrete-table	No	No	Yes	Dep.	Emul	Yes
	Join class-table	No	Flex	PK	Flex	Emul	Flex

The join of inheritance class tables may be done by PK, or may be flexible if user defined. *JPA* allows FK redefinition, but not the mapping of distinct PK in subclasses, what may bring unexpected problems for some mappings.

Table 2.3 presents a summary of the discussed topics that affect the design of systems based upon ORM frameworks. The topics were organized by UML element, such as models, classes, attributes, relationships and inheritance, helping designers to document their decisions on class models. The information summarized by tables 2.2 and 2.3 relates resources and decisions common to the studied persistence frameworks, helping with designing and documenting applications, as well as porting application across distinct ORM frameworks.

Table 2.3: Summary of design decisions based upon ORM frameworks.

<i>Domain Level</i>		<i>What to observe/question during project design</i>	
Model		If the framework has a data-model that controls persistence, how does it work with object-oriented domain models?	
		Are operations assigned to classes that extend generated classes or domain classes do not support operations?	
Class	All domain classes	Are inheritance to ORM framework classes mandatory?	
		Is there some dependency to framework classes on domain classes? Can it be decoupled?	
	If persistent	Which table(s) (if data mapper, what SQL statements) are mapped to that class?	
		Is the identity mapping defined? one or more attributes?	
		How the identity will be assigned? Will generation parameters for the identity be necessary? (such as table of ids, sequences, auto-columns...)	
	If embeddable	Is it necessary to distinguish a class as embeddable?	
		Is it possible to define preferred mapping for attributes?	
		Is it used as identity for persistent classes? If so, should the class follow specific rules required by the ORM framework?	
		Relationships <i>from</i> embeddable values to other domain classes should be avoided, and may be unsupported according to each framework.	
	Attribute (persistent)		Does it match a database column type or should it be an embeddable value?
			If it is part of a composite Identity, does it represent an embeddable value class containing PK fields or each attribute of the identity is an individual attribute of the class?
			Type parameters such as length and precision were defined? Dates and LOBs may need specific parametrization. Is the cardinality matching NULL/NOT NULL constraints?
Association (persistent)	zero/one-to-many	Should a collection type be defined ? How to deal with element ordering?	
		Can/Should the fetch configuration be specified?	
	many-to-zero/one	Will the relationship attribute be loaded by a proxy?	
	bidirectional	The collection must have a reverse attribute or collection in the opposite class that maintains a bidirectional relationship. Is it defined and documented?	
	to-one	The FK that implements the relationship may be a class attribute or may be hidden by the framework. The relationship may be represented only by a reference to the related object. Is the relation Attribute-Collection-FK clear and well documented?	
	many-to-many	Is a join table clearly defined, with FKs to the tables mapped to the related classes?	
		Do your ORM framework transparently support Join Tables? In what cases a join table must be explicitly implemented as an association class?	
other	Maps and element collections are supported by few frameworks.		
Inheritance		What strategy will be employed (among those available in the chosen framework) ?	
		Is a discriminator column necessary? This column is not an attribute of the domain model, but it is required to map inheritance. Discriminator requirements change from one framework to another.	
		Does the persistence framework support classes in an inheritance hierarchy with distinct Identities (PKs) ?	

3 ESSENTIAL NOTATION FOR ORM (ENORM)

This chapter presents the Essential Notation for Object-Relational Mapping (ENORM) and the artifacts related to this notation. ENORM extends the UML class model, offering a concise set of visual elements specific for ORM designs. These essential concepts, introduced at chapter 2, reflect persistence patterns of the literature adopted by distinct ORM frameworks in the market. The goal of ENORM is to facilitate the design by the clear application of ORM patterns, document mappings with a platform independent notation, and be a repository for MDD transformations, partial code generation, and round-trip engineering tools.

This chapter begins presenting the main visual elements, followed by examples explaining the key features. After a few examples, the *meta-model* is presented, followed by an analysis of special cases, known limitations, and a reference, in BNF, for element naming. The chapter ends by presenting the main features of the modeling tool, and related notations specialized in persistence.

3.1 Overview

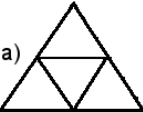


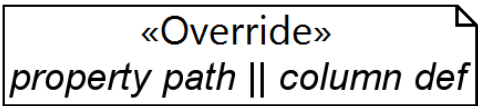
The notation here proposed is a lightweight UML profile, represented by a set of graphical extensions for class models, encompassing the essential structural concepts of ORM. ENORM was designed to be easily understood by developers and rich enough for MDD tools, allowing the specification of the relevant persistence details, but hiding what can be inferred.

ENORM elements (Figure 3.1) are derived from ORM patterns following the *Domain Model* pattern (FOWLER, 2002). Besides, ENORM reflects common practices of various ORM frameworks, such as *activerecord* for *Ruby (RAR)*, *JPA*, and *SQLAlchemy (SA)* for Python (BAYER, 2012; DEMICHIEL, 2013; HEINEMEIER HANSSON, 2012).

A *Persistent* class (marked with “||”) represents a class implemented as an *Active Record*, *Data Mapper*, or mapped in such a way by a framework. The class is persisted by a table with the same name; or one or more specified tables. Each property of a persistent class maps to a column, that can be detailed in the model when necessary.

Associations between persistent classes are implemented with Foreign Keys (FKs) detailed by join columns and tables. Inheritance can be *flat* for single table pattern; *vertical*, for joined table pattern; or *horizontal* for the concrete table pattern. Non persistent classes can be persisted within *persistent* classes, by associations marked as *embed*. A persistent class can have transient properties by using the transient symbol.

Figure 3.1: Main visual elements and their meaning.

Graphical Element	Description
class <i>table1, table2, ...</i>	Persistent class. Optional tables
property <i>column def</i>	Definition of the column mapping
«PK»	Part of the primary key
«Embed»	Dependent or embedded
(a)  (b)  (c) 	Inheritance types: (a) Flat, (b) Vertical or (c) Horizontal. Discriminators can be specified.
join table = <i>table</i>	Association table name
join columns = <i>col1, col2, ...</i>	Columns that implement association
	Override an inherited or embedded property or association mapping
<i>property or assoc. end</i> θ	Transient feature should not be mapped
«Map»	Set key property of qualified associations

The specification of ENORM follows the principles bellow:

- **Do not Repeat Yourself (DRY):** By representing concepts that are the same together: classes and tables, properties and columns, etc...
- **Convention over configuration (CoC):** Keep the notation short, and hide what can be inferred. For example, any association between persistent classes are persistent; if no PK is specified, use a simple non-meaningful column as PK.
- **Keep It Short and Simple (KISS):** Do not introduce completely new visual elements, but decorate existing elements of UML. Use of formatted comment boxes, or braces, to display stereotype details, as suggested by the UML specification.
- **Models as central artifacts:** All the essential mapping information should be stored at the model, and follow the ENORM meta-model.
- **Platform independence:** The notation is pattern centric, the majority of the resources are established ORM patterns in the literature. In a few exceptions, they represent trends revealed by our survey of chapter 2.
- **MDD aimed:** The ENORM is a UML profile, stored with XMI, and ready to be used as input for transformation languages that read UML models.

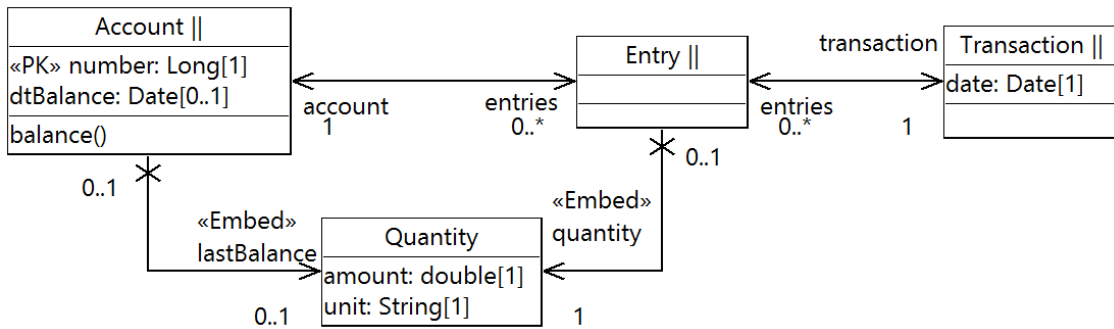
3.2 A Simple Example

Figure 3.2 shows a simple design for the *Accounting* patterns (FOWLER, 1996). *Account*, *Entry*, and *Transaction* are persistent classes, each persisted by tables with the

same name. *Account* has a meaningful PK named *number*. *Entry* and *Transaction* will also have PKs, but they are not specified (inferred by convention).

Quantity is not persistent and does not correspond to a table. However, each *Entry* instance refers to a *Quantity* with the *Embed* stereotype. Since the upper multiplicity is one, quantity association is persisted along the *Entry* table, by columns *amount* and *unit*. *Quantity* is similarly embedded by *Account*.

Figure 3.2: Simple Transaction example.



Finally, the associations between persistent classes are mapped as FKs connecting the PKs of each table. *Entry* will have a column referencing *account number* and a column referencing the PK of *Transaction*.

3.3 A not so Simple Example

Database information systems usually refer to centralized databases serving multiple systems, that must adapt to the existing schema. Often that means a break between nomenclature used by the system and the database, and a more complicated mapping.

Figure 3.3: Summary account example.

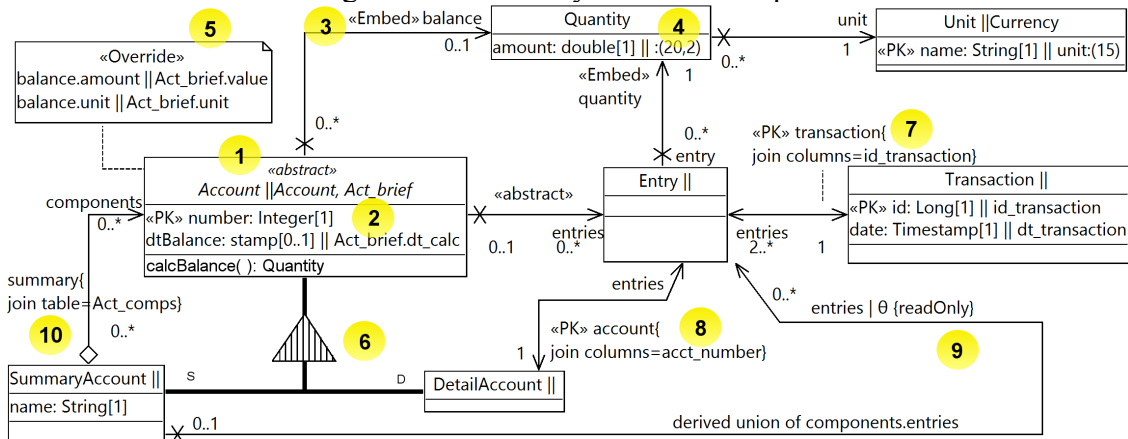
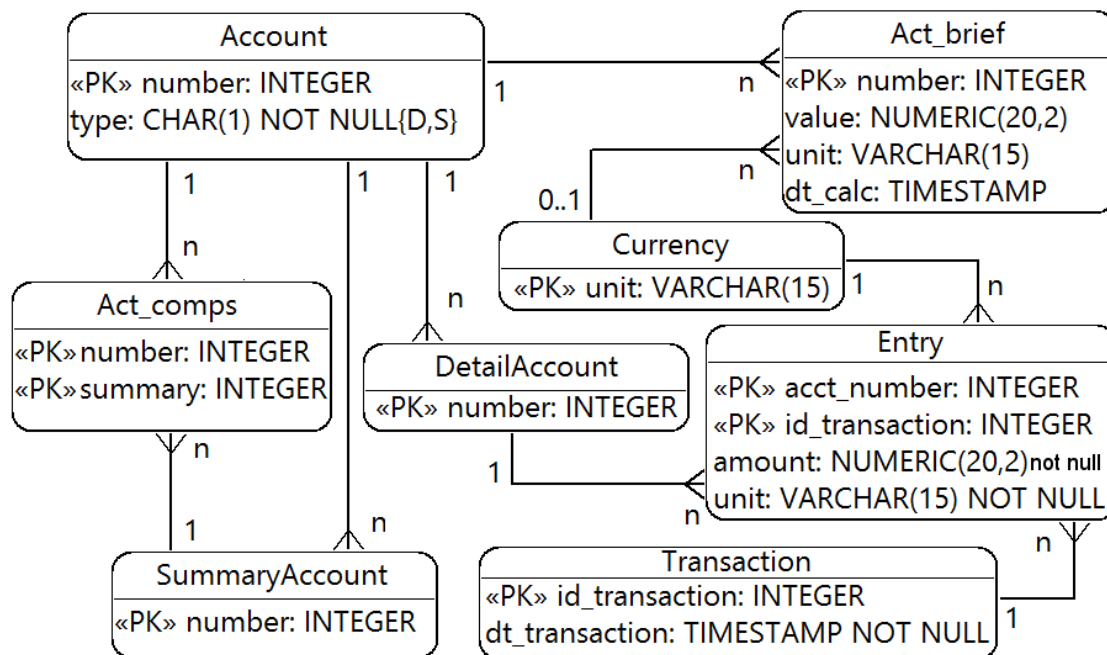


Figure 3.3 introduces the *SummaryAccount* class, that aggregates accounts implementing multiple summary accounts (FOWLER, 1996). Each account can be part of one or more summary accounts, and the *entries* of the summary was redefined as the union of all underlying *DetailAccount* instances. The *Unit* class now replaces the free

text unit property. Figure 3.4 presents the database derived from the model. Several changes were introduced in the mapping:

1. *Account* is mapped to two tables: *Account* and *Act_brief*. The table *Act_brief* has an FK to *Account*, that is also a PK, and each instance of *Account* is retrieved by a join. *Account* is the main table, because it is the first in the list.
2. Property *dtBalance* is mapped to column *dt_calc* on table *Act_brief*. Mapping information of a property can be specified using the persistence symbol (||).
3. *Quantity* now refers to a *Unit* persisted by the *Currency* table. When *Account* references a *Quantity* instance, it stores a reference (FK) to the *Currency* table.

Figure 3.4: Database model of account example.



4. Property *amount* with default SQL precision/scale of (20,2).

5. *Account* overrides the *quantity*: *amount* is persisted by the column *value* of table *Act_brief*; the association end *unit* is stored by the column *unit* in table *Act_brief*, that references the table *Currency*. By default, all columns would have been stored along the primary table *Account*.

6. The account inheritance tree is persisted with the *joined table* pattern. Each class has its own tables, and each PK of the specializations refers to the *Account* PK. The discriminator column can assume 'S', for summary accounts, or 'D', for detail accounts.

7-8. *Entry* refers to *Transaction* with a column named *id_transaction*, and refers to *DetailAccount* with a column named *acct_number*. Both relationships are marked as PK, setting a composite PK of *Entry*.

9. *Account* defines the association *entries* as *abstract*, therefore it is not persisted. *DetailAccount* redefines *entries* as bidirectional, with an FK, and a concrete association. The *SummaryAccount* redefines the association as derived from its components, with

the transient symbol marking that this association should not be stored by an FK column.

10. The *components* association is *many-to-many*, and therefore is mapped by an association table. The *join table* specifies that this table is *Acct_Comps*. By default, it will have FK columns referring to *Account* and *SummaryAccount*.

3.4 ENORM Meta-model

The profile package of UML contains a set of mechanisms providing the ability to tailor the UML meta-model for different platforms (OMG, 2011b). A profile is mainly comprised of stereotypes, each of them extending a meta-class of UML, such as classes, properties, or use cases.

The UML meta-model works as a tree, where each leaf is owned by its parent in a composite relationship. For instance, the *Property* is owned by one *Class*, that is owned by one *Package*. Each object has one, and only one, parent owner, or is the root. Each stereotype can have properties representing scalar values, or other meta-objects. When defining new meta-classes, they must also belong in the owning tree, tracking back to the root UML element. This ownership is denoted by making the association between the stereotype, and this meta-class, a composition.

Backing up the visual notation of ENORM, there is a profile providing compatibility between ENORM and UML implementations. Figure 3.5 summarizes the stereotypes, the extended UML elements, meta-classes, and its properties and relationships.

The *Persistent* stereotype is applicable to a *Class*, marking this class with the double bars (||) of Table 3.1. The *source* property allows the direct definition of one *Table*, a reference to an already defined table by *TableRef*, or a *JoinedSource* comprising two or more tables connected by *JoinColumn* objects. If *source* is unspecified, the class is persisted by a table with the same name of the class.

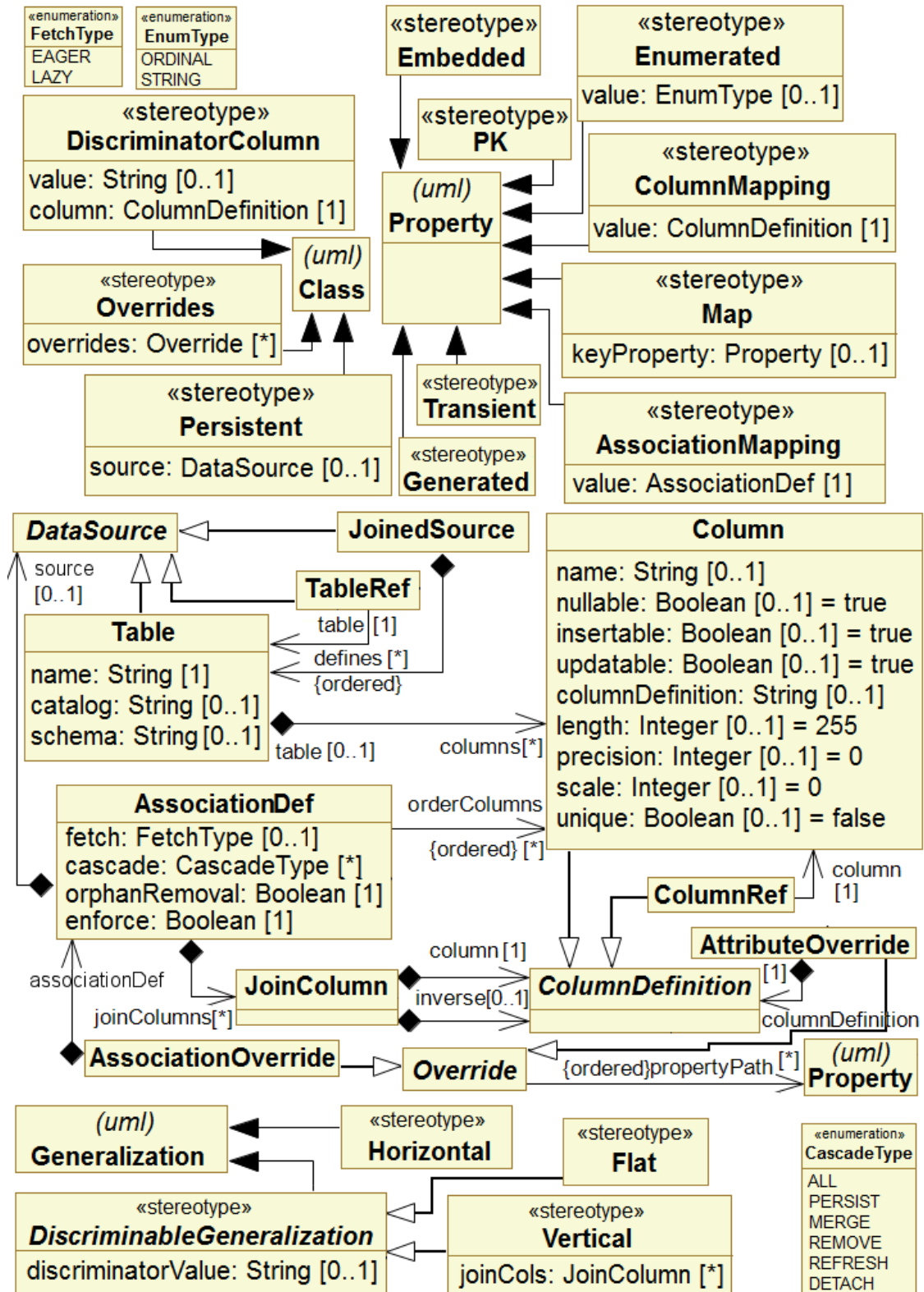
The use of *Table* or *TableRef* determines the class that “owns” the table definition, preventing duplicate specification of tables. This follows the *proxy* pattern, where the owner has a composite association with an abstract meta-class, that can be the object itself (table, column, and so on) or just a reference to an object, owned by some other stereotype/class (GAMMA et al., 1994).

Following the example of Figure 3.3, the *Account* class has the *Persistent* stereotype applied, with the *source* property referencing a *JoinedSource*, because it is mapped to two joined tables. The *JoinedSource* will have two *Table* definitions at the *defines* ordered composition, the first being *Account*, and the second *Act_brief*. Both tables are defined by the *Account* class. The *Unit* class will have the *Persistent* stereotype, with the *source* directly referencing a *Table* named *Currency*. Classes such as *Transaction* can have unspecified *source*, because the table has the same name of the class.

Properties owned by a persistent class are, by default, persisted, and scalar values are stored as columns. The *ColumnMapping* stereotype allows the definition of these columns, informing column name, if it accept nulls, length, precision, scale, unique constraint, database type and so on. The column can be owned by a *Table*, but the table may be inferred, if the *Persistent* class does not define a table, or if the class is not

persistent. Again, a *ColumnDefinition* can be a *Column* owned by the property, or a *ColumnRef proxy* that references a *Column*. A property without mapping will have a column with an inferred definition from its meta-information (type, multiplicity, etc...).

Figure 3.5: Main elements of the ENORM profile.



The *Embedded* stereotype is applied to association ends, or simple properties, whose types are not persistent classes. This means that this class is persisted as a dependent table (if *to-many*), or embedded in the table (if *to-one*). Properties of non persistent classes can have the *ColumnMapping* stereotype applied, in order to specify how is its preferred way of being persisted, such as *length*, *precision*, and so on. These definitions will not be owned by a *Table*.

For example, the *dtBalance* property of *Account* will have the *ColumnMapping* stereotype with a *ColumnRef* value, referencing a *Column* with name *dt_calc*. This *Column* is owned by the *Table Act_brief*, defined in the mapping of *Account*. The property *amount*, of *Quantity* class, will have the *ColumnMapping* stereotype referencing a *Column*, with precision of 20, and scale of 2, because *Quantity* is not persistent and this column does not exist at any specific table. However, when *Quantity* is embedded by *Entry*, this information is used to know what is the default precision and scale of this property.

The *AssociationMapping* stereotype allows the definition of mapping details for one association, by the application in one of the association ends. The *AssociationDef* class allows the definition of fetch strategies, cascade delete, orphan removal policy, columns used by an *order by* clause, join columns, and a join table. The *JoinColumn* class defines the FK column in the detail side, and optionally the corresponding PK in the master side (for multiple PK, or ad hoc joins). If *enforce* is true, there will have a FK constraint for the join columns. The *source*, of *AssociationDef*, is usually defined on *many-to-many* situations, to specify the table(s) that implements the relationship.

For example, the class *Entry* has an association end named *transaction*, that refers to the *Transaction* class, with the *AssociationMapping* stereotype applied. This defines an *AssociationDef*, owning one *JoinColumn*, that owns a *Column* named *id_transaction*. Notice that if *Entry* defines the mapping table, the *JoinColumn* could own a *ColumnRef* referencing a *Column* owned by this table. Another example is the *components* association end, that has the *AssociationMapping* stereotype with an *AssociationDef* that owns, by the association *source*, the *Table* named *Act_Comps*. This table implements the *many-to-many* relationship.

The *PK* stereotype marks a property as part of the PK of some persistent class. It can be applied on association ends, such as *transaction* and *account* of *Entry*, meaning that the FK columns are also part of the PK. *PK* can be combined with *ColumnMapping*, *AssociationMapping* and so on. *Generated* marks a column with generated values.

Horizontal, *Flat*, and *Vertical* stereotypes can be applied to a generalization, to specify which pattern will be used to emulate inheritance on the database. With *Flat*, all columns necessary to represent the inheritance tree are stored in the same table. Usually, the instance type is determined by a discriminator column, that can be defined by applying the *DiscriminatorColumn* stereotype at the general class, and filling the property *discriminatorValue* for each generalization with the *Flat* application.

The *Vertical* stereotype stores each class along its properties in a distinct table, that is by default joined by a common PK. It is possible to specify what columns perform the join by the *joinCols* property. It is also possible to define a discriminator. Finally, the *Horizontal* stereotype stores each concrete class independently, and the origin table determines the type.

In the example of Figure 3.3, the generalizations connecting *Account* and its specializations have the *Vertical* stereotype applied. The generalization from *SummaryAccount* has the *discriminatorValue* equals to “S”, and the generalization from *DetailAccount* has the value of “D”. We could have defined the discriminator column by applying the *DiscriminatorColumn* stereotype to *Account*, detailing the *column* property. If *Account* was not abstract, we could had specified the discriminator value of *Account* too.

A class may specify an inheritance pattern, even when it inherits from a non persistent class. In this situation (and only this), the properties and associations of the general class will be persisted along the persistent specializations. The *Overrides* stereotype allows a class to override such properties (*AttributeOverride*) and associations (*AssociationOverride*), defining the *columns*, *join columns*, *join tables*, among other details.

A class may also override properties and associations of embedded/dependent classes. The tricky part here is that one class can embeds a class, that embeds another class. The property path of embedded overrides is represented by the ordered association *propertyPath*.

In the example of Figure 3.3, the *Overrides* stereotype was applied on the *Account* class, containing one *AttributeOverride* and one *AssociationOverride*. The override with the path “*balance.amount*” refers to the *AttributeOverride* with the sequence {*Account.balance*, *Quantity.amount*} as *propertyPath*. This allows the override to differentiate when the class has more than one relationship to the same class.

The *Enumerated* stereotype allows the definition of how enumerations are mapped (*STRING* or *ORDINAL* values). The *Transient* stereotype marks a property, or association end, to be ignored on persistence mapping.

3.5 Special Mapping Cases

This section presents a series of special cases that exemplify other applications of the notation supported by the profile presented by section 3.4.

3.5.1 Embedded Values

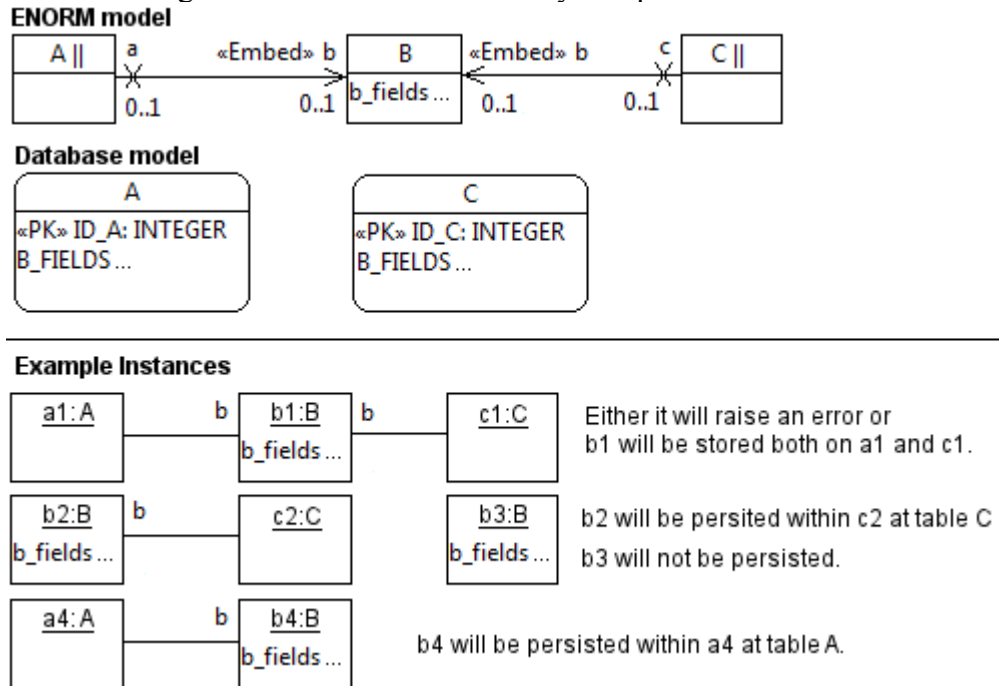
Class embedding denotes a class from which its instances are persisted embedded at a related class table. Only the instances referenced by persistent classes with an embed association can be persisted; and will be persisted only when the owning class is persisted. This kind of relationship suits well to aggregations, compositions and classes that behave like a database domain.

The embedding is represented in the relationship by the <<embed>> annotation, just after the association end pointing the embedded class. On Figure 3.2, the *Account* class is persistent, and related to the *Quantity* class, that is not persistent. The embedded relationship allows the information of *Quantity* to be persisted along with *Account* records.

The embedding on ENORM is a stereotype for the *association end*. The class do not need to be marked as *embeddable*, but it will implicitly be mapped that way, according to the framework used for persistence.

Typically, the attributes of *Quantity* became attributes of the table(s) of *Account* (*Act_brief*, at the example), such as *value* and *unit*. However, the *nullable* characteristic of each column depends on the relationship cardinality. An *Account* may not have a *balance*, and therefore both columns must allow nulls on the database, even if *amount* has minimal cardinality of *one*. Notice that a *Quantity* not referenced by an *Account*, or an *Entry*, will not be persisted.

Figure 3.6: A class embedded by two persistent classes.

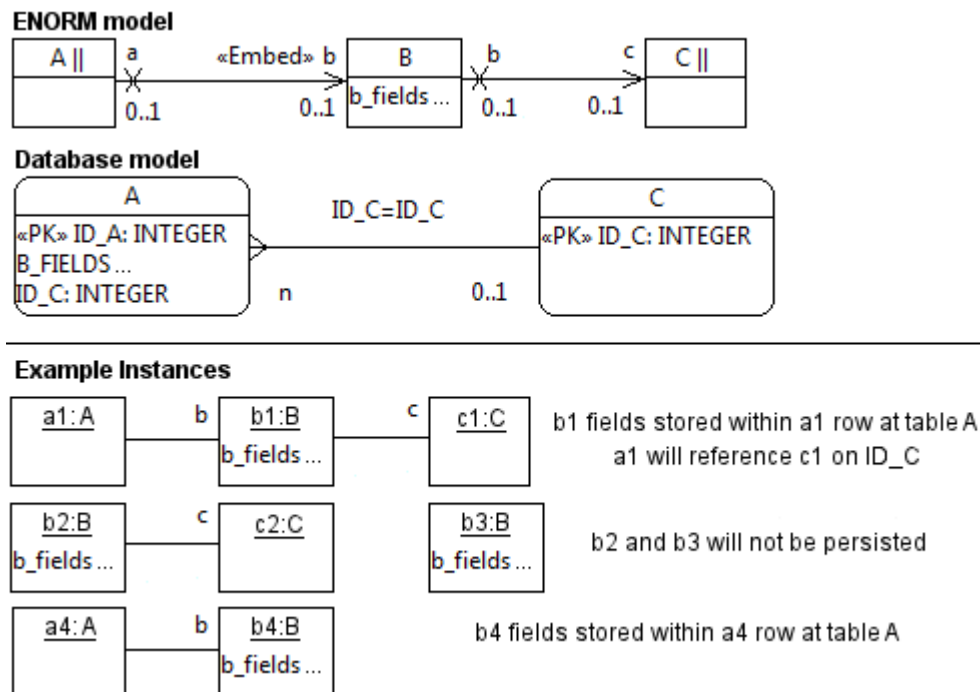


One class may be embedded by various persistent classes, and that is the usual scenario. Figure 3.6 shows classes *A*, *B* and *C*, where both *A* and *C* embed *B*. *A* and *C* are responsible of persisting related *B* instances, and therefore their tables will have the columns necessary to persist *B*. Individual instances of *B* will be persisted on *A*, *C*, or will be dumped, according to the references held by either *A* or *C*. If an instance, such as *b1*, is referenced by more than one entity, and it is persisted, there is no guarantee that, when restored, the same instance (in memory terms) of *b1* will be referenced by *a1* and *c1*. Thus, the implementation should ensure an adequate equality operation(s) (such as overriding the Java *equals* method).

A non-persistent class may reference a persistent class without being embedded. Instances of this class will not be persisted, even when related to instances of this persistent class. However, they can be persisted if embedded by another class.

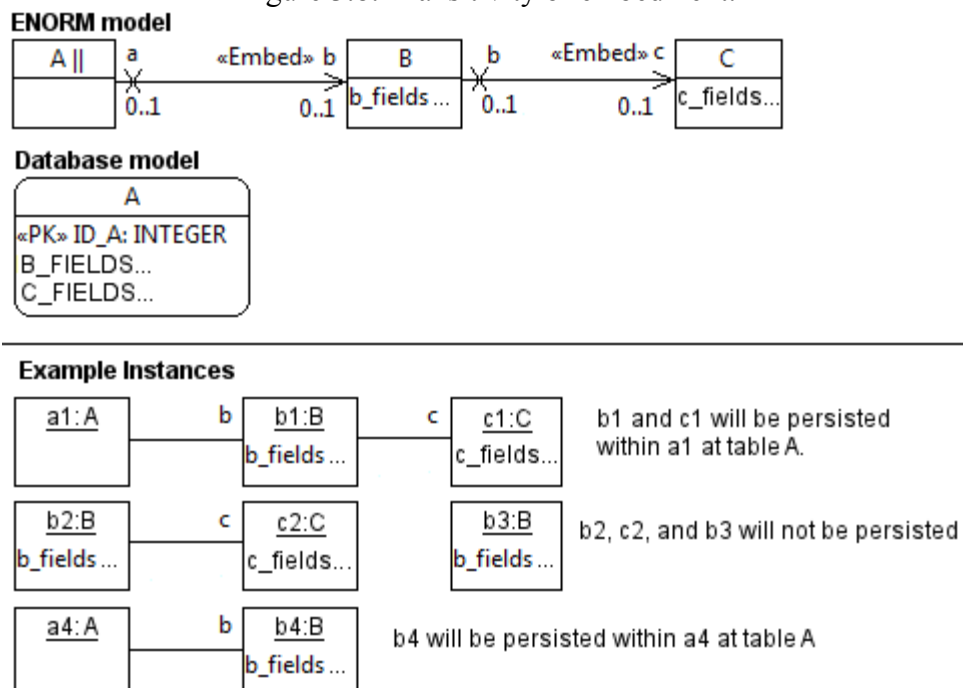
For instance, take the classes *A*, *B*, and *C* of Figure 3.7. *A* and *C* are persistent, *A* is associated to *B*, *B* to *C*, and *A* embeds *B*. An instance *b2* of *B* referencing *c2*, but unreferenced by any *A*, will not be persisted. An instance *b4* of *B*, referenced by an instance *a4* of *A*, will be persisted. An instance *b1* of *B*, referenced by an instance *a1* of *A*, that references an instance *c1* of *C*, will be persisted, and also the reference to *C*. This will inevitably link *A* and *C* tables in the database. If the cardinality of *A* to *B* is one, and *B* to *C* is also one, *A* will be persisted in a table that has a foreign key (*ID_C*) to *C*.

Figure 3.7: Embedded classes referencing persistent classes.



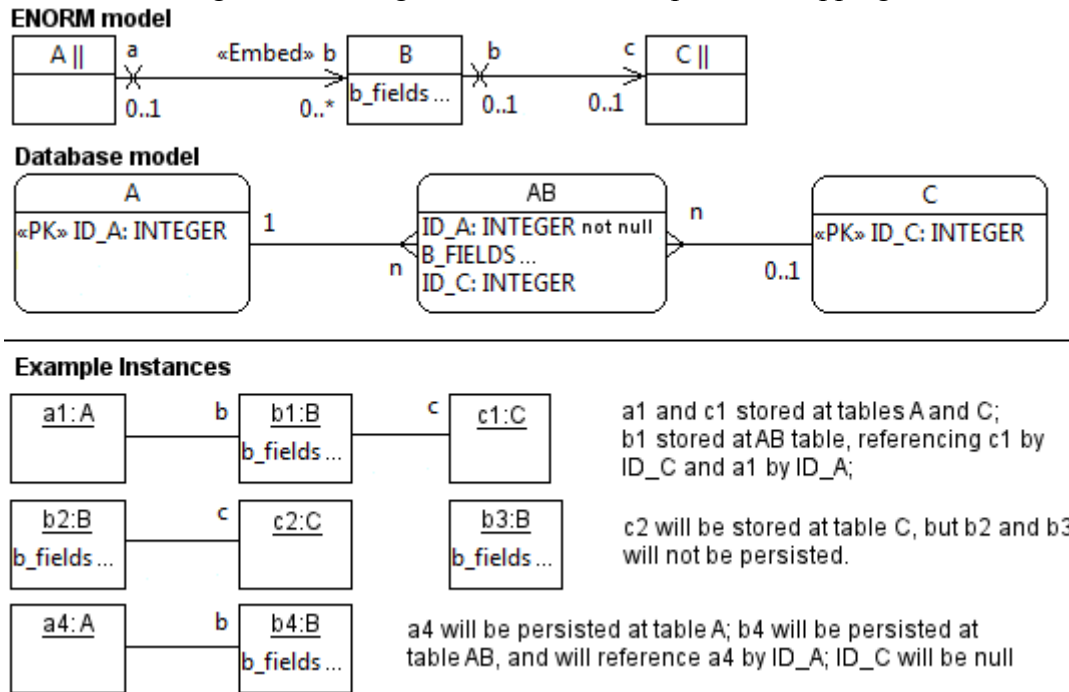
This is what happens between the classes *Entry*, *Quantity*, and *Unit* of Figure 3.3. The table *Entry*, that stores the embedded *Quantity*, has a FK to the table *Currency*, that stores *Unit*, as shown by Figure 3.4.

Figure 3.8: Transitivity of embedment.



Embedding is transitive, as in Figure 3.8. If class *A* is persistent, but classes *B* and *C* are not, and *A* embeds *B* that embeds *C*, then instances of *A* such as *a1* will persist instances of *B*, such as *b1*, and any referenced instance of *C*, such as *c1*.

Figure 3.9: Using <<Embed>> for Dependent mapping.



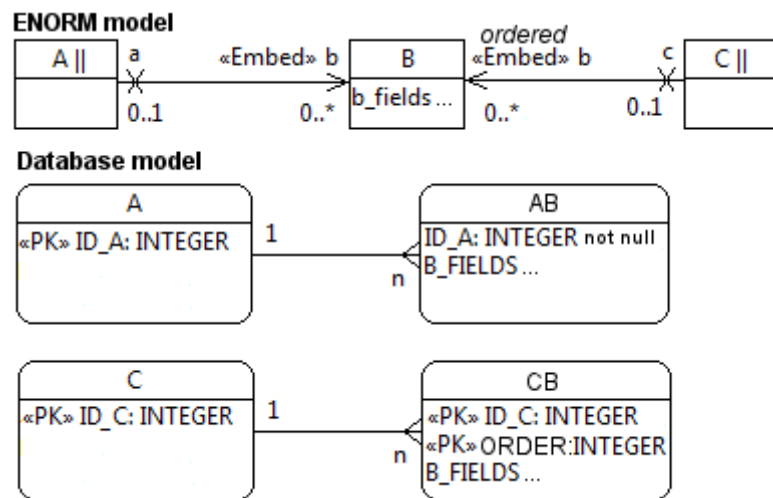
A persistent class can have more than one embedding association to the same class. The modeler can use the mapping override to give meaningful names to the columns that persist the object. Otherwise, the column names will depend of the ORM tool, that usually concatenates the association end name with each property.

The embedding association can have cardinality greater than one, in which case an exclusive table will be necessary to persist the data in this association (see Figure 2.13 in the *Embedded values support* section). Figure 3.9 shows an example of embedding association with collections for the example of Figure 3.7. The table *AB* exclusively stores zero or more *B* elements referenced by *A*. If the instance *a1* references *b1*, and *b1* references *c1*, then *b1* will be stored as an *AB* record referencing *a1* and *c1*.

The table *AB*, by default, does not have a PK, because only entities have identities. However, if the embedded property is an *ordered* collection, *AB* will have an order column, forming a composite PK with *ID_A*, as exemplified by Figure 3.10. The ordering can be toggled by the UML attribute *isOrdered*. Figure 3.10 also presents a variation of Figure 3.6, but with collections of *B* elements. Each collection is persisted by an exclusive table, *AB* for *B* related to *A*, and *CB* for ordered *B* related to *C*.

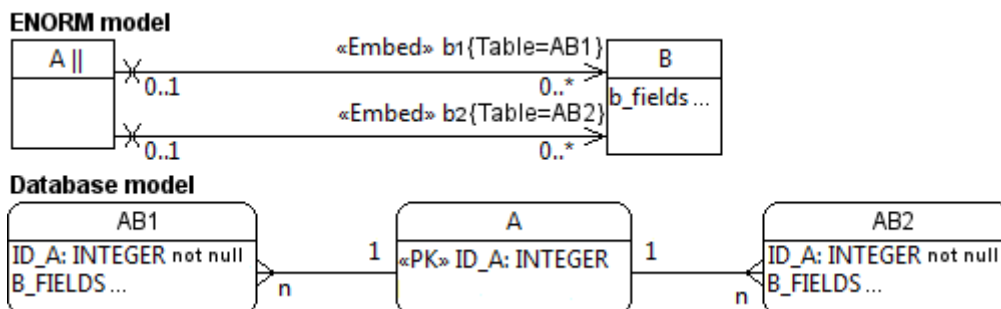
Embedding a persistent class is not allowed. If class *B* is embedded it must not be persistent. Also, the association to an embedded class must be navigable, and the reverse end is usually not navigable, because most ORM frameworks will not support this navigation.

Figure 3.10: Two collection-embedments example.



When embedding two *one-to-many* associations to the same class, the modeler may want to specify the details of the collection table, such as the table name. The *AssociationMapping* stereotype can be used with *Embedded* to specify the table (actually, a *DataSource*). Figure 3.11 shows an example of this application where *A* has two *one-to-many* embedded associations to *B*, specifying the collection table *AB1* for the association end *b1*, and *AB2* for the association end *b2*.

Figure 3.11: Two dependent collections to the same class.



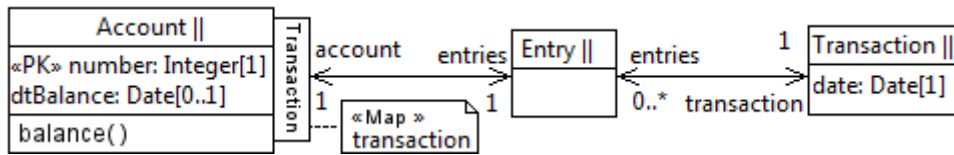
Nevertheless, embedded collections should be avoided when possible, and replaced by first class entities. They are not easy to understand, although ORM tools offer resources for this kind of mapping.

3.5.2 Maps

UML allows the specification of qualified associations, that represents partitions in the association between two classes. When the qualified property has an upper value of one, the association represents what is commonly referred as *Map* or *Dictionary* by OO languages (OMG, 2011b).

Figure 3.12 presents an example where the association end of *Account* is a map with a $\langle \text{Transaction}, \text{Entry} \rangle$ form, where the qualified variable of type *Transaction* is the key. The *Map* stereotype allows the specification that the key is, in fact, the *transaction* property of *Entry*, what is common on ORM. The goal is that, when the user adds a pair $\langle tx, ey \rangle$ to the map, it will associate *ey* both with the *account* and the *tx transaction*.

Figure 3.12: Map with key reference.

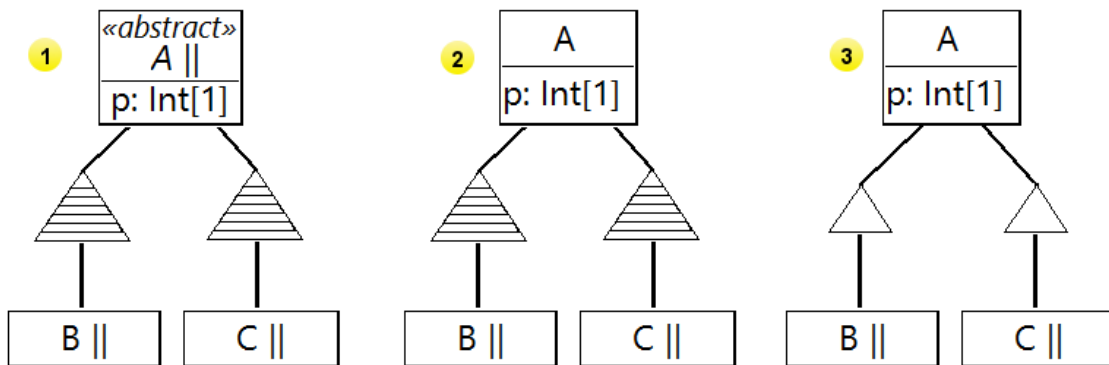


The property key can also be user defined, derived from a complex operation. In such cases, it can be a read-only map. Qualified associations without a property key are also allowed. In the example, the map would be persisted in a separate many-to-many table, instead of using the association between *entries* and *transactions*. Qualifier properties can also assume non-persistent and scalar types.

3.5.3 Inheritance

The combination of inheritance strategies, and persistence, unlocks some special situations with specific mappings by the ORM tools. Figure 3.13 presents three inheritance situations where the super class did not have a mapping table.

Figure 3.13: Three different inheritance examples without parent table.



In the first model, all three classes are marked as persistent, but the super class *A* is abstract. The horizontal inheritance states that only concrete classes will have mapped classes, and therefore there will be no need for a table that persists *A* instances.

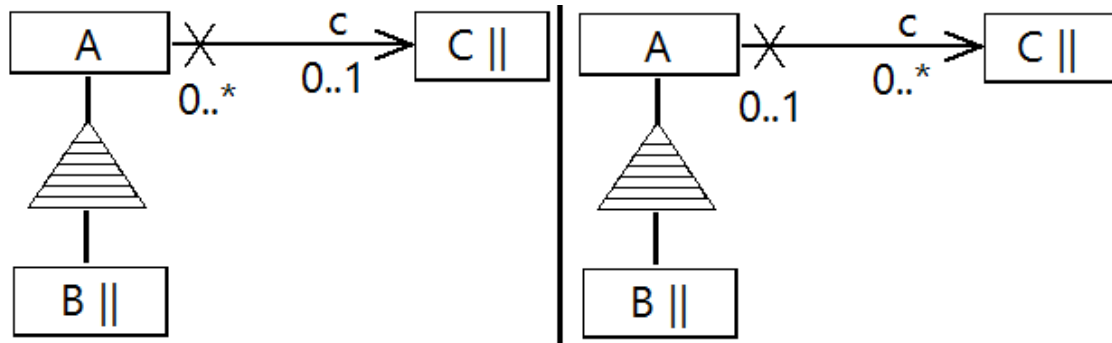
In the second and third models there are instances of *A*, but *A* is not persistent, and therefore did not have any table. However, the resulting tables persisting the specializations are distinct: At the second model, the horizontal inheritance makes the property *p* persistent at the persistent subclasses of *A*, and therefore tables *B* and *C* will have a column persisting *p*. The third model has no defined strategy, meaning that *p* should be ignored for persistence at the subclasses.

Figure 3.14 presents other cases of horizontal inheritance with non persistent parent, but this time with an association to a persistent class. Instances of *A* are not persistent, and therefore the association referencing *C* is transient. However, instances of *B* are persistent, and should persist the inherited associations of *A*, because of the horizontal inheritance.

In the left model of the figure, *B* references *zero-or-one* *C* instances by inheritance, therefore the table of *B* will have a FK pointing to *C*. However, in the right model, *B*

references *zero-or-many* *C* instances. This could be persisted by a FK from *C* to *B*, but that would require a bidirectional relationship. The preferred solution to keep the unidirectional association is to have a third table (*B_C*) that stores, like a *many-to-many* relationship, what *C* objects are related to *B*. That table would have as PK the same PK of *C*, to enforce that a *C* can have only one *B*.

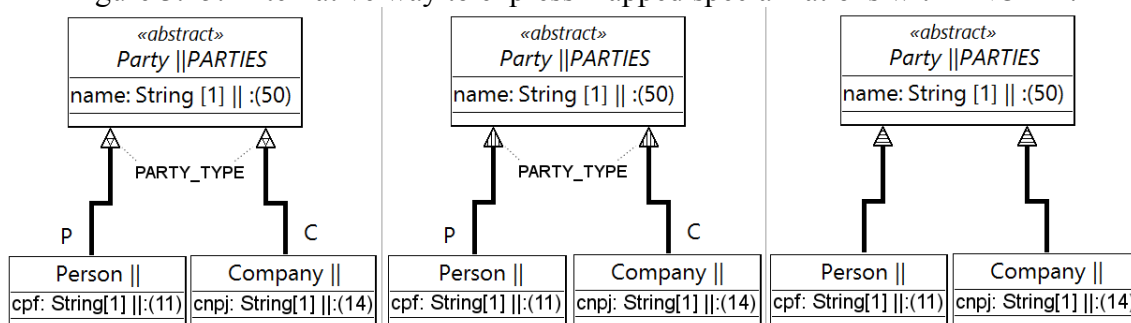
Figure 3.14: Inherited association with persistent class.



Bidirectional relationships variations of the Figure 3.14 are usually not well supported by ORM tools, because they restrict associations to transient classes. The modeler may want to create a direct relationship between *B* and *C*, overriding the original relationship. This is similar to what happens in the example of Figure 3.3, where the one directional association between *Account* and *Entry* was overwritten by the bidirectional association *DetailAccount* and *Entry*.

The inheritance symbol may be used at the end of the relationship, as exemplified at the example of Figure 3.15. The discriminator column should be displayed near the arrows, and only one time for the hierarchy. The discriminator values are represented near the end representing the class. If the discriminator applies to the general class, it should be placed after the column definition. For example, if *Party* was not abstract, and its discriminator was “Y”, then the discriminator definition would be `PARTY_TYPE="Y"`.

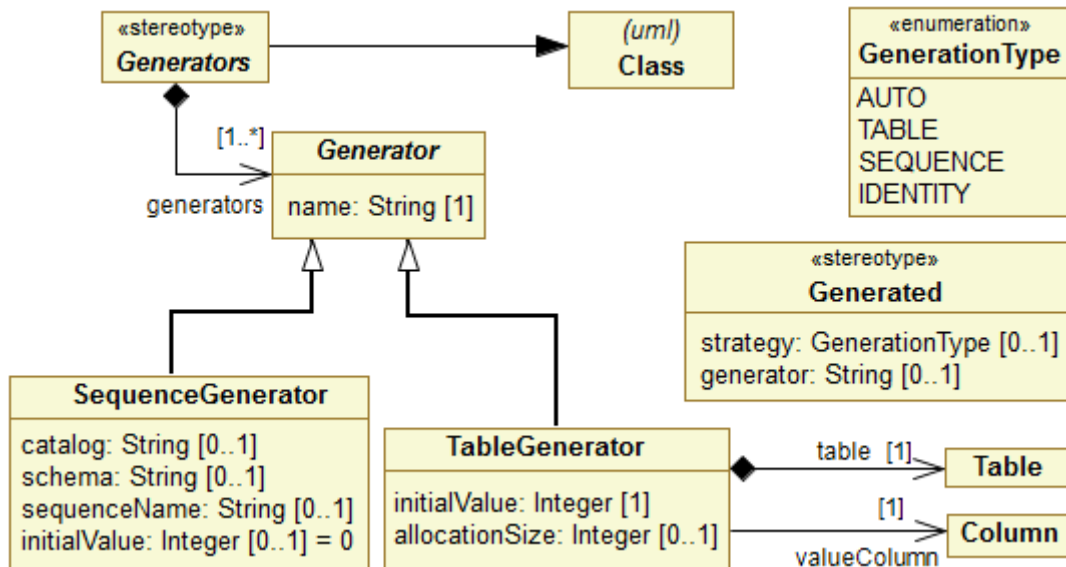
Figure 3.15: Alternative way to express mapped specializations with ENORM.



3.5.4 Auto-generated Columns

Auto-generated PKs are very common at RDBs, and a default behavior for some ORM frameworks, such as *RAR*. Generated properties are not distinguished by ENORM notation, but can be specified at the tool applying the *Generated* stereotype at the property. A class with no explicit PK will have an auto-generated PK.

Figure 3.16: ENORM profile model of generators.



The *Generated* allows the specification of the *strategy* of generation and an optional *generator*. The default strategy *AUTO* means that the strategy of the target platform should be used. This usually is the *IDENTITY*, that does not require the specification of a *generator*.

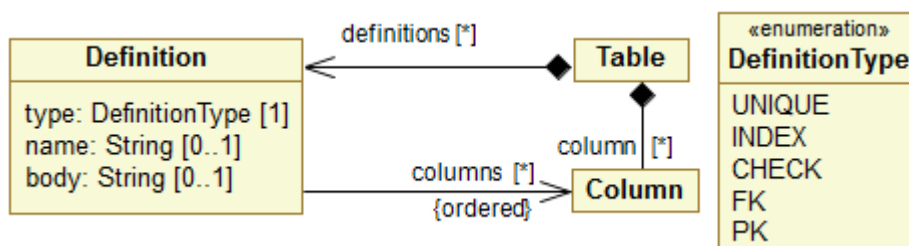
The *SEQUENCE* and *TABLE* strategies require the specification of the name of the *generator*, that can refer to a generator defined at the model, or at the database. To specify a *Generator* at the model, a class must have the *Generators* stereotype applied, specifying one or more *SequenceGenerator* or *TableGenerator* instances.

Sequence generators and table generators can be shared by more than one column, and at distinct tables. The *TableGenerator* specifies a *Table*, and one of the columns of this table should be the *valueColumn*, that holds the last generated value. In order to avoid visual pollution, information about column generation is not planned to be displayed by ENORM diagrams.

3.5.5 Constraints and Indexes

When it is necessary to specify every detail of the database, ENORM allows the specification of constraints and indexes, by specifying the *definitions* of a *Table*. Figure 3.17 shows the meta-model elements of ENORM that stores the creation of *Definition* objects.

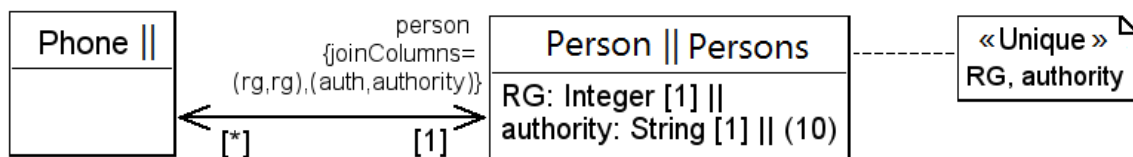
Figure 3.17: Profile model of table definitions, for indexes and constraints.



A *Definition* have a *type*, that specify if it is an index, a unique index, a check constraint, a FK, or a PK. The *Definition* may have a name, that identifies it at the database, and a *body*, if the user wants to specify the command that creates the constraint.

The *Definition* is also associated with an ordered list of *columns* that it affects. For instance, if we want to define a unique index, for an alternate key of *Person*, using *RG* and *authority*, we create a *Definition* on the *Table Persons*, referencing the *columns* *RG* and *authority*. Notice that for using the *Definition*, it is necessary to define the *Table*, and the *Column* of each referenced property, using the *ColumnMapping* stereotype, as shows the Figure 3.18.

Figure 3.18: Definition example with unique index constraint definition.



The notation used at Figure 3.18 for the unique *Definition* is a suggestion, that is not covered by the ENORM visual notation. Most ORM frameworks did not support advanced definition of constraints, but *SA* allows. If used, the title should be the type of *Definition*, and the contents are the list of *columns* or the *body*.

Notice that an *AssociationDef* can enforce this FK if it have one or more *JoinColumn* objects, each referencing one *inverseColumn* that is part of the PK, or part of a unique index. At the example, *Phone* references *Person* by two columns named *rg* and *auth*, and inverse columns pointing to the unique index columns, instead of pointing to the implicit PK of *Person*. A FK *Definition* is not necessary for *Phone*.

FK and *PK* definitions should be avoided. The preferred way of specifying FKs is by creating *JoinColumn* objects at the *AssociationDef* meta-object. For PKs, the application of the *PK* stereotype at the properties or association ends is the preferred way of specifying the constraint. However, ENORM allows the specification of tables that are not visible as classes, such as tables that implement *many-to-many* associations and *JoinedSource*, and for these cases the *Definition* is the only way to specify/customize the constraints. Moreover, unique constraints with only one column, can be specified by the *unique* property of *Column*, but unique constraints with many columns can only be specified with *Definition* objects.

3.6 Limitations

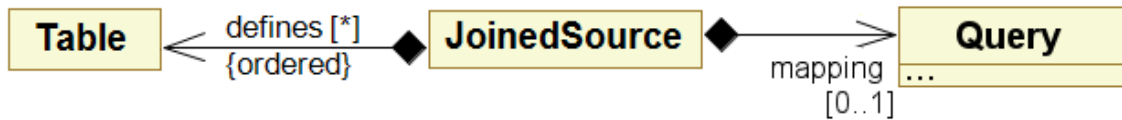
This section enumerates some known limitations of ENORM.

3.6.1 Flexible Data Sources

Currently, the profile only supports the mapping of one class, to many tables, if each table has a *one-to-one* relationship to the first table. This is an easy way to specify the data source, without caring about checking how a complex mapping would be persisted. A more flexible rule for data sources would be equivalent to a side effect free *updatable view* (DAYAL and BERNSTEIN, 1982).

This, however, does not invalidate the current ENORM meta-model. The *JoinedSource* could be extended, in the future, with the definition of the *Query* that would retrieve the object instance. The Figure 3.19 shows an example of these meta-classes, but the *Query* would have to map all the dynamic operations of the data manipulation language of SQL. This *Query* would reference the tables already defined in the model, or defined by the *JoinedSource*. This goes beyond the current scope of ENORM.

Figure 3.19: A sketch of meta-model with flexible data sources.



3.6.2 Qualified Associations

Qualified associations can have more than one qualifier properties. This kind of construct would need keys with *tuples* of objects, what can be quite complicated to implement using ORM tools. Qualified properties, with upper cardinality over one, are a special case, representing a map of collection elements, where each key can have more than one associated value.

3.6.3 Multiple Inheritance, Multiple Types

The profile does not include resources to deal with the persistent specialization of more than one persistent class, and the resulting mapping would be unknown. However, a class can specialize any number of other classes, as long as it only inherits persistent information from one tree branch. Single relation with multiple type attributes (ELMASRI and NAVATHE, 2003) was not included in ENORM.

3.6.4 Association Class and “n-ary”

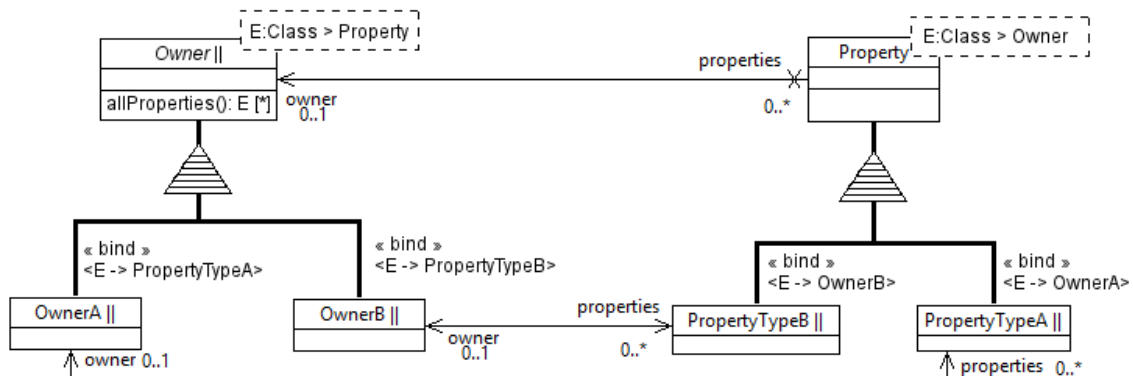
The profile does not have any specific mapping for the *Association Class* element of UML, it is as any other class. ENORM does not yet support persistent associations with more than two classes, common on ER conceptual models. These associations must be separated on binary associations.

3.6.5 Generics and Template Parameters

Mechanisms such as generics can be specified using template parameters on UML, and they are useful for strong typed languages, such as Java and C#. We did not identify any additional extension necessary to the use of template parameters in that context. However, certain combinations are difficult to model, and implement, using the studied ORM tools.

Figure 3.20 has an example of template parameter where the class *Owner* is parametrized with a parameter named *E* that identifies what kind of *Property* this *Owner* has; *Property* also has a parameter *E* that identifies what kind of *Owner* this has. Then we created two persistent classes that specializes *Owner* to own properties of persistent specializations of *Property*: *OwnerA* owns *PropertyA*, and *OwnerB* owns *PropertyB*. This is denoted by the binding of *E* at the generalization.

Figure 3.20: Template parameter example.



As a general rule, the parametrized class cannot be persistent if the parameter affects the type of its relationships. For instance, if *Owner* is persistent, and the relationship *owner-properties* is defined in function of the binding of *E* at each class, the persistence manager will not know for sure what subclasses of *Property* and *Owner* exist, in order to establish FKs between the classes.

To exemplify the above problem using Java, the *properties* of the *Owner*, and the *owner* of the *Property*, could both be implemented using generic parameters:

```
class Owner<E extends Property>...
Set<E> properties; // what E associations are persistent?
class Property<E extends Owner> ...
E owner; // what E associations are persistent?
```

At the example, *Owner* is persistent but abstract, and *Property* is not persistent. An instance of *Property* will not be persisted, only of its specializations. Nevertheless, the *owner-properties* relationship of *Property* is persistent at the subclasses, because its subclasses declare a *horizontal* specialization.

However, each pair (*A-A*, *B-B*) of subclasses overrides the *owner-properties* relationship, to explicitly tells the ORM framework what combinations are possible. For instance, *PropertyTypeB* will have a FK to *OwnerB*, and will not need a FK to *OwnerA*.

In practice, it is still difficult to use template parameters with the UML notation. For instance, it is not clear how the notation, for a relationship bound to parameter *E*, would look like, simply because template parameters are much more flexible than *Java Generics*. It is possible to specify anything as constraint, not only class types, and therefore a relationship dependent to *E* could have an indefinite destination.

3.7 ENORM Notation Reference

UML defines the notation, of the textual elements at the model, using a variation of the BNF notation. With ENORM, we extended these textual notations, describing information about the ORM mappings using the EBNF meta-language (ISO, 1996). Figure 3.21 lists the syntax definitions to describe classes, properties, associations ends, discriminators, and the mapping overrides. The EBNF here described omits the concatenation operation (commas).

Figure 3.21: EBNF specification for ENORM labels.

```

class_def =
  class_uml [PERSISTENT [datasource]];
datasource =
  table_def ["," datasource];
table_def =
  [catalog"."][schema"."]table;
property_def =
  [PK] [EMBED] property_uml [PERSISTENT property_mapping | TRANSIENT];
property_mapping =
  column_ref[":"column_def];
column_ref=
  [table_def"."]column_name;
column_def=
  [column_type][("length")]{("column_modifiers")};
length = number | number","number;
assoc_end_def =
  [PK] [EMBED] assoc_end_uml [TRANSIENT | assoc_mapping];
assoc_mapping = "{" assoc_table | join_columns "}";
assoc_table = ("joinTable=" | "Table=") table_def;
join_columns =
  "joinColumns=" (
    column_ref{"","column_ref"} |
    ext_join_column{"","ext_join_column});
ext_join_column =
  ("column_ref","column_ref");
overrides = override{"\n"override};
override =
  property_path PERSISTENT [PK] (property_mapping | join_columns);
property_path = identifier["."property_path];
column_modifiers =
  "unique" | "non-update" | "non-insert" | "read-only";
discriminator =
  property_mapping ["="discr_value];
EMBED = "«Embed»";
TRANSIENT = "⊖";
PERSISTENT = "||";
PK = "«PK»";

```

The *table*, *catalog*, *schema*, *column_name*, *column_type*, and *discr_value* non-terminals are simple *identifiers*. The non-terminals ending with “_uml” are defined by the UML specification (OMG, 2011b). The uppercase identifiers represent the terminals introduced by ENORM.

The underlined non-terminals define the notation of the non-trivial concepts represented by ENORM at the models:

- *class_def*: The notation for naming a class.
- *property_def*: The notation for naming a property. Notice that *property_uml* is itself an UML non-terminal, containing information such as *type* and *multiplicity*.
- *assoc_end_def*: The notation for naming association ends.
- *overrides*: The notation for displaying the list of overrides at a comment box. Notice that overrides usually refer to properties, and associations, defined by other classes.

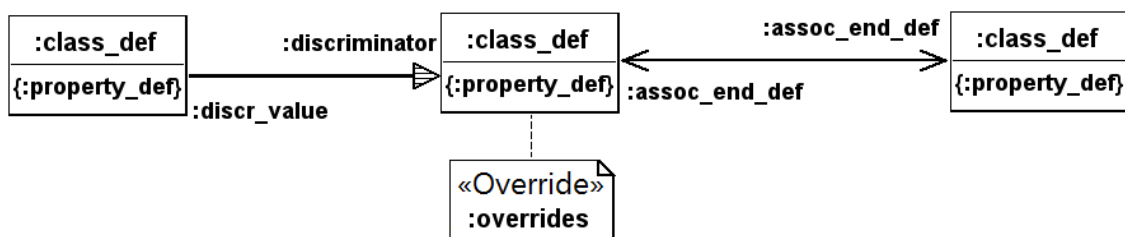
- *discriminator*: The notation to display the discriminator column, at the specialization end of the general class.

Using EBNF, the terminal symbols are specified between quotes, and the non terminal symbols are specified by rules after the equal symbol. The rules are a concatenation of elements, or definition lists separated by pipes. Parenthesis are used to group elements, brackets to delimit optional elements, and braces to delimit elements that can repeat zero or more times.

For example, the *class_def* non-terminal has a rule that concatenates the UML class name (*class_uml*), with the optional terminal *PERSISTENT* (when the class is persistent). If the class is persistent, the *class_def* can concatenate the non-terminal *datasource*. The *datasource* is defined at a recursive rule, that has at least one *table_def*.

Figure 3.22 shows the application of the non-terminal elements at the model. Notice that this example uses the alternative way to display inheritance, between the center and left classes, as described at section 3.5.3. The notational elements of UML that are not affected by ENORM, such as operations and association names, were omitted.

Figure 3.22: Visual distribution of the ENORM non-terminals.



3.8 Modeling Tool

The modeling tool was developed as a *plugin* for Eclipse, using open source components from the Eclipse Platform (ECLIPSE FOUNDATION, 2012a). The most important components were:

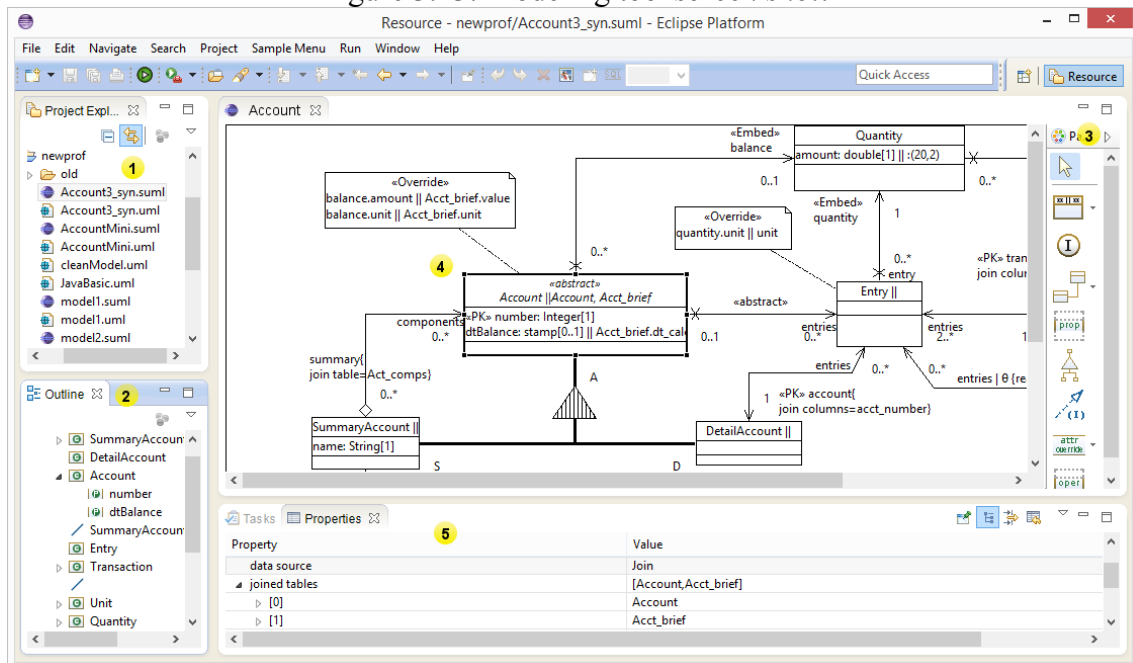
- The UML2 project, a framework intended to implement the specification of UML 2 as an Eclipse Modeling Framework (EMF) meta-model. The UML2 project provides a model to create UML and UML profiles, and store them using XMI (OMG, 2007). However, it does not include the creation of UML diagrams.
- The *Graphical Editing Framework* (GEF) of Eclipse, a framework that provides support for creating modeling tools, such as UML editors, supporting figure manipulation, figure connections, layers, and viewport/scrolling control (ECLIPSE FOUNDATION, 2012b).

The tool takes the responsibility to abstract the ENORM meta-model from the user, and draw the diagrams using the notation elements of ENORM. Figure 3.23 presents a screen capture highlighting the main windows of the tool:

1. The *package explorer* is an eclipse window that lists the model files. The *suml* files contains the visual information about the models, and the *uml* files the structural information, including the ENORM profile application.

2. The *outline* window shows the main elements of the diagram in a tree view.
3. The palette toolbar lists the elements available to the user draw the model: classes, interfaces, associations, properties, inheritance, implementation, attribute overrides, and operations. Each component in the palette has a slide, that allows the user to choose other components, such as persistent classes, embedded associations, and association overrides.
4. The *drawing canvas* is the most important window, where the user draws the model, and select elements for editing. At the above example, the *Account* class is selected.
5. The *properties* window is where the user can edit the details of the selected component in the canvas. For instance, because the *Account* class has a data source defined as *Join*, the window knows that should list the tables defined in the join. The *proxies* are therefore handled in a transparent way for the user.

Figure 3.23: Modeling tool screen shot.



The tool is distributed as *open source*, and can be downloaded from our *sourceforge* project¹.

3.8.1 Modeling Tool for the Experiments

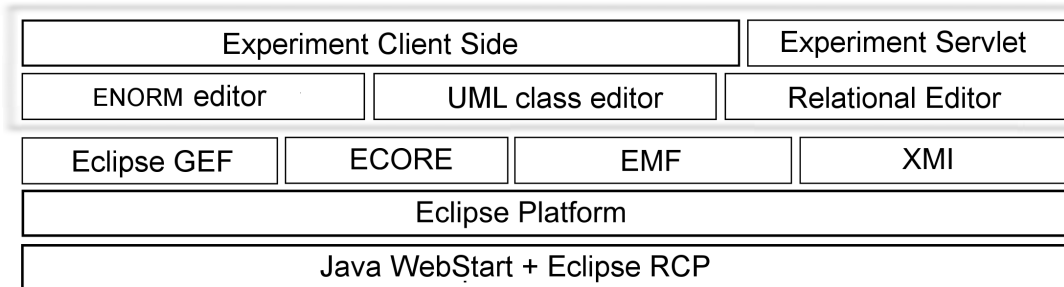
For the experiments detailed at chapter 5, a specific version of the tool was employed, capable of editing relational and UML class models. This tool is not a *plugin*, but *Rich Client Platform* (RCP) available as a Java Web Start application.

The architecture of the experimental tool is summarized at Figure 3.24. The Java Web Start allows the download and execution of Java applications, with one click on the browser, only requiring the *Java Runtime Environment* installed in the client machine

¹ <http://eorm.sourceforge.net/>

(ORACLE, 2014). The experimental modeling tool also communicates with a Java *Servlet* that manages the experiment.

Figure 3.24: Experimental tool architecture.

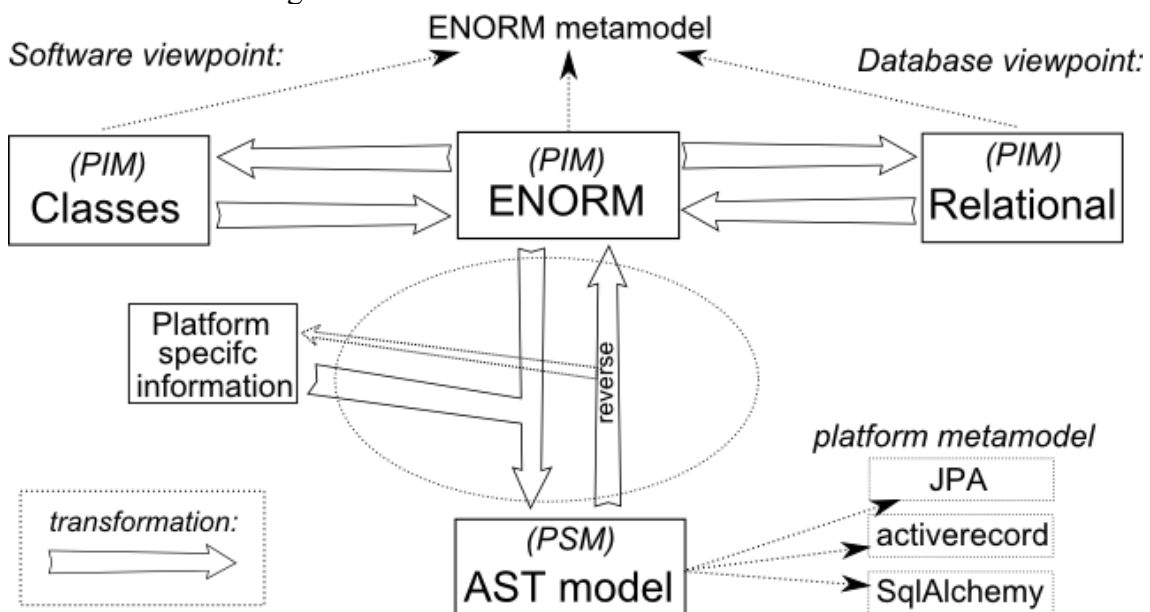


Each step of the experiment, and what modeling language should be used by the user, is managed by the tool. The models, and all data about the experiment, are uploaded to the *Servlet* after the experiment is completed.

3.8.2 Future Steps

The tool focus until now was to allow the modeling using ENORM. Transformations and model checking were left out of the tool until this moment. The ideal modeling tool would be able to implement a round-trip engineering, synchronizing code and models, respecting what is produced in one, or other artifact (Figure 3.25).

Figure 3.25: MDD scenario for ENORM models.



The idea is to create bidirectional transformations for the three studied platforms at chapter 4 (*JPA*, *RAR*, and *SA*). These transformations would use a mechanism, such as *Symmetric Lenses* or *Delta Based Transformations*, to transform models to code, and back again (DISKIN, XIONG and CZARNECKI, 2010; HOFMANN, PIERCE and WAGNER, 2011).

Because this area is a separate research topic, presenting several open research questions (BORK et al., 2008), we decided to let this for future work, instead of using established transformation languages, that did not yet cover bidirectional transformations, such as QVT and ATL (JOUAULT et al., 2006; OMG, 2011a).

The Platform Independent Models (PIM) are transformed, with the aid of Platform Specific Information (PSI), to the Platform Specific Model (PSM). The PSM is a model, but not a diagram, being the Abstract Syntax Tree (AST) of the destination code. A transformation takes PIM elements, that are visualized as diagrams, and transforms into PSM elements, that are visualized as code.

Both relational and class models are subsets of an ENORM model, and therefore function as views. All UML can be specified with ENORM, because ENORM is UML with an applied profile, but not all databases can be specified using ENORM (for instance, stored procedures cannot be expressed by ENORM). However, we assume that what cannot be represented, is part of PSI.

Therefore, what takes from the ENORM model to the PSM, including database stored procedures, are these specific information, that can be hard coded by transformations, the result of specific profiles informing the user preferences, or the PSM itself. Using the PSM, the reverse transformation can read the changes performed by the developer at the code, and use this information for the following transformations.

3.9 Other Class Models and Persistence Extensions

This section briefly describes two relevant approaches to the problem of persistence modeling and object-relational mapping. The first is a profile focusing in extending UML to draw ER/Relational database models and the second is a proposal of the Object Management Group (OMG) to standardize UML extensions to several persistence medias, including a traceability support between specific concepts and requirements.

3.9.1 A UML Profile for Data Modeling

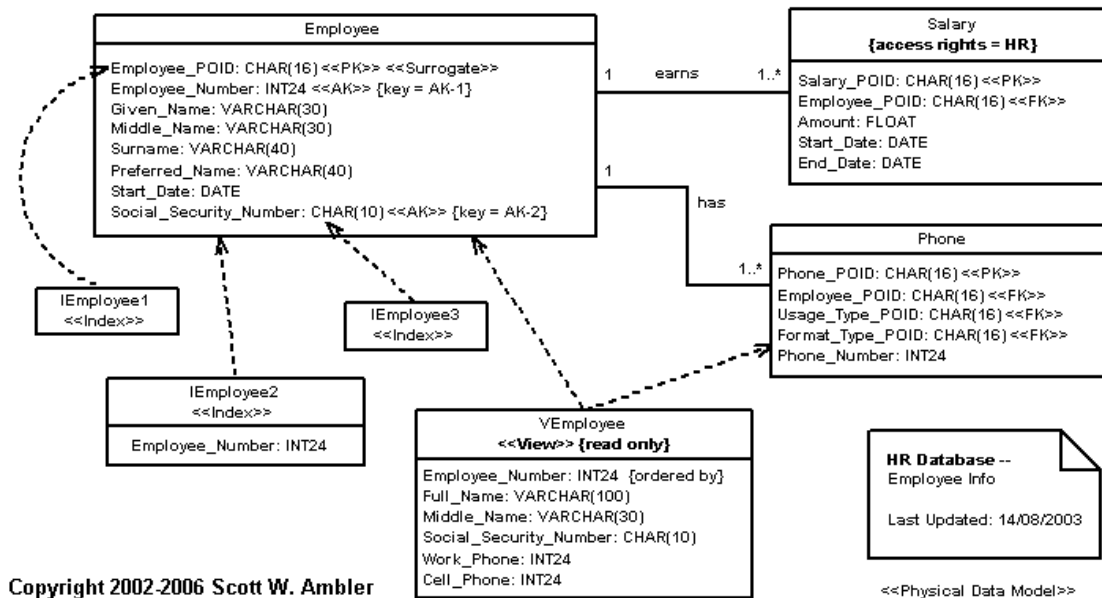
The *UML Profile for Data Modeling* was proposed to fill the absence of a data model diagram on UML. It is a profile, intended to be used by class diagrams, to emulate concepts common on database modeling tools, such as tables, PKs, foreign keys, indexes, and views. The profile also includes support to model other persistent mechanisms, such as files and XML (AMBLER, HARTFORD and RUECKERT, 2003).

The profile allows three levels of abstraction: Logical, Physical, and Conceptual data models, following the ANSI standard for database design (AMERICAN NATIONAL STANDARDS INSTITUTE, 1975). When creating a new data model, the designer will choose a model type according to these levels. The design of the software domain is done with separated class models, with no extensions.

Several stereotypes were defined, according to the abstraction level, to specify classes as tables, views, or indexes; properties as columns, foreign keys, PKs, or alternate keys; and associations as generalizations, or identifying/not-identifying, to name a few. The notation also covers triggers, access restrictions, and stored procedures, using OCL when necessary.

Figure 3.26 is an example of the profile in use, for a physical data model representing a RDB schema. The diagram describes three tables and a view of the database, with PKs, FKs, alternate keys (AKs), indexes, and database column type information. It is important to observe that the diagram does not show how the tables are related to the application classes that represent the domain.

Figure 3.26: UML Profile for data modeling example (AMBLER, HARTFORD and RUECKERT, 2003).



Models with this notation can be used as input for MDD transformations, generating SQL scripts to create/maintain a database. A tool was developed with this objective (HARTFORD, 2004).

This profile, however, does not deal with the mapping between OO classes and RDB tables. It is not intended to ORM, but as an alternative to use UML notation for database design, with a profile compatible UML design tool.

3.9.2 Information Management Meta-model (IMM)

The Information Management Meta-model (IMM) is an ongoing effort of OMG to bridge the gap between the UML, RDBs, and XML modeling. The IMM approach consists in the standardization of UML profiles, and transformations, to represent persistence using UML. It encompasses relational database design, entity-relationship models, XML Schemas, LDAP models, and a traceability model to manage the interoperability between these meta-models (OMG, 2005, 2012).

The IMM is still a request for proposal, however the modeling approach is similar to the UML profile of Ambler, by creating separated models using consolidated notations adapted to the UML meta-model. According to the proposal, when creating ER and relational models, the design decisions will be traced to requirements, and then used to generate code for the system. How exactly this traceability and code generation will work is not yet clear, but the IMM does not include an effort to model concepts *together*. Database models and class models are separated unconnected models.

By using separated models, even if all traceability is registered, the connection between concepts will be hidden from the stakeholders. ENORM follows an opposite path, by exploring the synergy of modeling the concepts together. The scope of IMM is also much more generic, because it encompasses all possible ways of representing RDBs, plus other persistent medias. The scope of ENORM is specific to relational database mapping patterns, following the domain logic described by the *Domain Model* pattern.

4 ENORM IN PRACTICE: APPLICATION EXAMPLES

This chapter discusses the implementation aspects of systems, designed with ENORM models, using three different ORM frameworks, at three distinct platforms: *JPA*, *SA*, and *RAR*. Chapter 2 surveyed the ORM tools with a pattern approach, and Chapter 3 presented our notation to represent these patterns, in the context of the surveyed tools. At this chapter, we explore the challenges and differences on mapping from ENORM models to the distinct platforms and ORM tools.

Four ENORM domain models, based on analysis patterns, are presented, and at each case, we point out the most important differences, when implementing each model on the three distinct platforms. At the end of the chapter we present a summary of guidelines to developers, highlighting the difficult points in the context of MDD.

4.1 ENORM and ORM Frameworks

The way *JPA*, *SA*, and *RAR* implements each ORM pattern is distinct. *AR* separates database definition on migration files, apart from class and mapping definitions, that are independent from the migrations. *JPA*, on the other hand, infers much of the database structure from annotations placed before each class (or XML), but does not have a central place where the database is defined. In the middle ground, *SA* allows the definition of tables, classes, and its mappings separately (*classical*) or together (*declarative*), but the table definitions are clearly separated at *run time*.

Each example implementation, presented at this chapter, is one among various possible implementations. They represent optimal implementations, for a certain version, at a certain configuration, within a specific platform. They capture limitations of each tool that are specific within these constraints.

The *JPA* examples were developed with Hibernate version 4.2.8 and Java JDK 1.7. The mapping was executed using annotations at the code, and only *JPA* annotations. We could had used specific hibernate annotations, XML instead of annotations, or other *JPA* implementations such as *EclipseLink*.

The *SA* examples were developed using the *classical* mapping, that separates the instantiation of classes, tables, and mappings. *SA* also has another way to express the mappings, by the use of the *declarative* mode, where all persistent classes extend a *Base* class. But we found the *classical* mapping more interesting to compare with the other two frameworks. The version of Python was 2.7.5, and the *SA* version was 0.8.2.

The *RAR* examples were implemented with *Ruby* version 1.9.3, *ActiveRecord* version 3.2.14, and *composite_primary_keys* version 5.0.13, this last being an

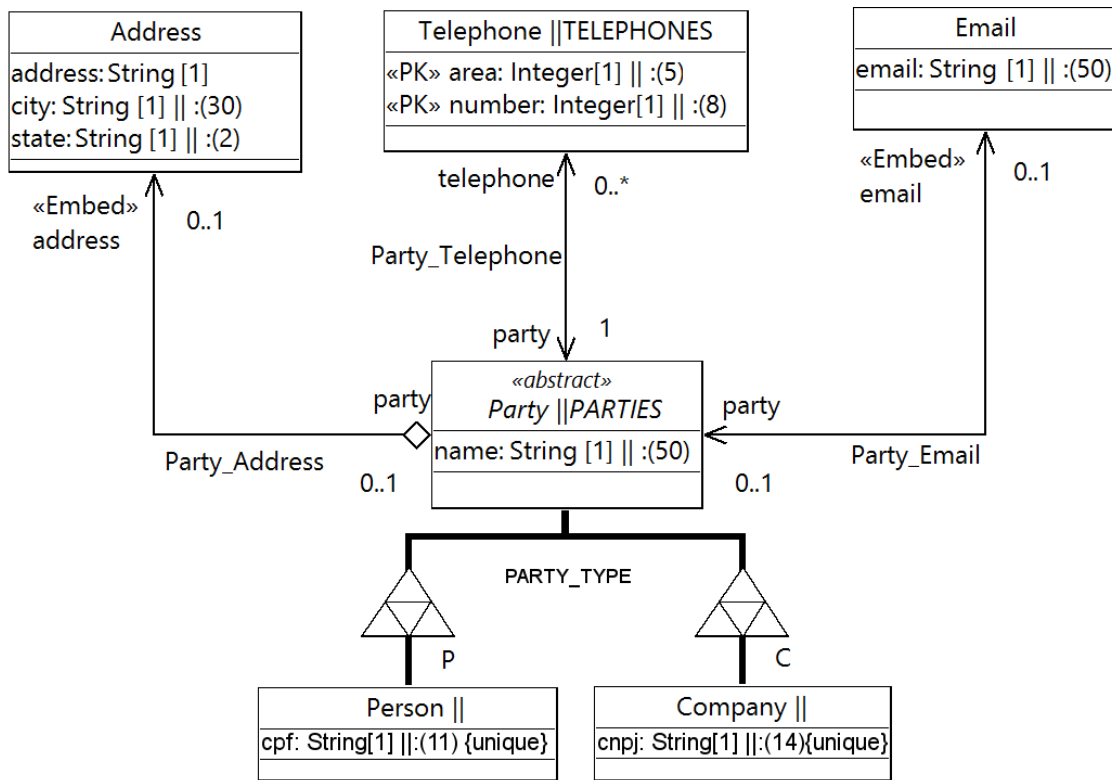
independent modification, to add support to composite keys (DR NIC WILLIAMS and CHARLIE SAVAGE, 2013). Also, migrations files were used to create the database. The projects were created using the *Ruby on Rails* framework.

The database used by the examples was the PostgreSQL version 9.2. The database creation was performed by the ORM tools, by setting options to create the database from the mappings.

4.2 Party Pattern for Accountability

The concept of accountability applies when a person, or organization, is responsible for another. This first example is a model for the abstract concept of *Party*, a pattern described by Fowler, defining a super class that abstracts the common attributes of person and organization (FOWLER, 1996).

Figure 4.1: Party pattern designed with ENORM.



The model of Figure 4.1 presents the *Party* pattern as a class with a *one-to-many* relationship to *Telephone*, and *one-to-zero/one* *Address* and *Email*. For the persistence, we decided to use the *embedded classes* solution where possible, reducing the number of tables. Classes that represents entities such as *Person*, *Company*, and *Telephone* are persistent (represented with the || sign), *Address* and *Email* are embedded. *Telephone* is persisted by table *TELEPHONES*, and *Party* by table *PARTIES*.

The *Party* hierarchy has no PK specified by the model, meaning it should not use a meaningful PK. The class that represents the organization is named *Company*. The *Telephone* class has two properties marked as PK, *area* and *number*, what configures a composite key.

The inheritance of *Party* to *Person*, and *Company*, is implemented by the *Flat* strategy, meaning that all three classes will be persisted at the same table. The `PARTY_TYPE` column is the discriminator that assumes the value P for *Person*, and C for *Company*. *Party* is abstract, therefore has no discriminator value.

Regarding the stereotypes, all persistent classes will have the *Persistent* stereotype applied. The class *Party* will have the *DiscriminatorColumn* applied, defining the column used for discrimination. Each generalization relationship aiming *Party* will have the *Flat* stereotype applied, with a *discriminatorValue* (P or C).

The properties can have complementary specification of database information, specified using the *ColumnMapping* stereotype, represented by the persistence symbol (||). For example, *city* has length of 30, *state* length of 2, and *email* length of 50; but *address* has no length specified.

The association ends *email* and *address* will have the *Embedded* stereotype. The association between *Telephone* and *Party* have no stereotype application, because associations between persistent classes are already persistent.

4.2.1 Mapping Persistent class Telephone

The following subsections present the mappings for each studied platform.

4.2.1.1 Using JPA

With *JPA*, persistent classes are those annotated with *@Entity*. Persistent FK associations are usually annotated using *@ManyToOne* at the side that owns the FK, and *@OneToMany* at the side that is referenced by the FK. Only the navigable sides have association mappings.

The *Telephone* class is annotated with *@Entity*, that has a parameter to specify the table name. All properties are mapped to columns automatically, but it is necessary to inform the length, by using the *@Column* annotation, at the instance variables that represent each property. The association end to *Party* is annotated by *@ManyToOne*.

The tricky part of the mapping is the definition of the composite PK, requiring a separated class just to hold the keys. Each property that is part of the key is annotated by *@Id*, and the *Telephone* class is annotated by *@IdClass*, pointing out the PK class. Here is the code fragment with the *Telephone* class:

```
@Entity @Table(name="TELEPHONES") @IdClass(TelephonePK.class) public class Telephone{
    @Id @Column(length=5)private int area;
    @Id @Column(length=8)private int number;
    @ManyToOne(optional=false) private Party party;
    //methods... (get/set)
}
class TelephonePK implements Serializable{
    private int area, number;
    //methods... (get/set>equals/hash)
```

4.2.1.2 Using SQLAlchemy

We have to declare the *Telephone* class, instantiate a *Table* mapping, and instantiate a *mapper* connecting the class and the table. The table is composed of column instances, informing database type, primary and foreign key constraints, and any other information

necessary to define the table. Bidirectional associations are declared at the many side of the *mapper*, but all FKs are declared at the *table* instance:

```
Telephone_table = Table('Telephones', metadata,
  Column('area', Integer(5), primary_key = True, autoincrement = False),
  Column('number', Integer(8), primary_key = True, autoincrement = False),
  Column('party_id', Integer,
    ForeignKey('Parties.id'))
)
class Telephone(object):
    #constructor/get/set methods...
TelephoneMapper = mapper(Telephone, Telephone_table)
```

4.2.1.3 Using ActiveRecord of Ruby

RAR have a separated module that deals with the definition of the database, by extending the *Migration* class. Each migration may define a *change* in database, or a pair of *up/down* sections, with instructions for the application of the migration, and its rollback. The developer can use *RAR* without using migrations, because *RAR* will not read the migrations to make the mappings, differently from *SA* that uses table objects to understand the mappings:

```
class CreateTelephone < ActiveRecord::Migration
  def up
    create_table :TELEPHONES, {:id => false} do |t|
      t.decimal :area, :precision=>5, :null => false
      t.decimal :number, :precision=>8, :null => false
      t.references :party, :null => false
    end
    execute 'ALTER TABLE "TELEPHONES" ADD PRIMARY KEY (area, number);'
  end

  def down
    drop_table :TELEPHONES
  end
end
```

The *CreateTelephone* migration creates a table named TELEPHONES, without default PK (*id=>false*). By default, *RAR* migrations would create a PK named *id*, with auto increment. We have to declare each column, followed by the precision specified at the ENORM model. The FK column is declared using the *t.references* type, with the name of the destination class. Ruby will create a column named *<class name>_id* referencing the PARTIES table, but it did not create FK constraints. Because composite keys are not supported, we have to specify an extra SQL command to create the PK constraint. This “execute” command is RDB vendor dependent.

RAR persistent classes extends the *ActiveRecord::Base* class. Most of the mapping is available by properties of the *Base* class, such as the table name (*table_name*). The problem here is that *RAR* classes also do not support composite PKs. A solution to this problem is to use the optional *composite_primary_keys* package for *RAR*, that fixes this limitation, offering the *primary_keys* list:

```
class Telephone < ActiveRecord::Base
  self.table_name="TELEPHONES" #the default in RAR is plural, but lowercase
  self.primary_keys = [:area, :number]
  belongs_to :party, :inverse_of=>:telephones
end
```

The *primary_keys* declares *area* and *number* as the PK. The *belongs_to* modifier declares *party* as a reference to the *Party* class, with inverse relationship *telephones*, thus implementing the bidirectional relationship. *RAR* associations, based on FK

pattern, are mainly mapped using *belongs_to* for *one/many-to-one* associations, and *has-many* for *one-to-many* associations.

4.2.2 Embedded classes

The following subsections present the mappings for each studied platform.

4.2.2.1 Using JPA

The embeddable classes of *JPA* must be annotated with *@Embeddable*. The attributes can have column definitions, such as the *length*, that are used when embedding the class:

```
@Embeddable public class Address {
    private String address;
    @Column(length=30) private String city;
    @Column(length=2) private String state;
    //methods... (get/set>equals/hash/...)
```

4.2.2.2 Using SqlAlchemy

The embedded classes do not need to be annotated, all they need is to implement the *composite_values* method. Any configuration is done by the *mapper* instance of the embedding class:

```
class Address(object):
    def __init__(self, address, city, state): #constructor
        self.setAddress(address)
        self.setCity(city)
        self.setState(state)
    def __composite_values__(self):
        return self.address, self.city, self.state
```

4.2.2.3 Using ActiveRecord of Ruby

Using *RAR*, the embedded classes do not need to be annotated, or modified. Any configuration is done by the persistent class that embeds the class. In the example bellow, *Address* has his three properties and a constructor. Notice that embedded instances must be immutable on *RAR*, so every attribute is declared to have only public getters by the *attr_reader* module method:

```
class Address
  attr_reader :address, :city, :state
  def initialize(address, city, state)
    @address, @city, @state = address, city, state
  end
end
```

4.2.3 Party, Person, Company, and Flat inheritance

The following subsections present the mappings for each studied platform.

4.2.3.1 Using JPA

The *Party* abstract class is defined as an *@Entity*, and the *@Inheritance* annotation defines the mapping strategy, in our case *Flat* is *Single_Table*. Following our model, we also define the *@DiscriminatorColumn* as *PARTY_TYPE*:

```
@Entity @Table(name="PARTIES") @Inheritance(strategy=SINGLE_TABLE)
@DiscriminatorColumn(name="PARTY_TYPE") public abstract class Party {
    @Id @GeneratedValue private int idParty;
    @Column(nullable = false, length=50) private String name;
    @OneToMany(mappedBy="party") private Set<Telephone> telephones;
    @Embedded private Address address;
    @Embedded private EMail email;
    //methods... (get/set/...)
```

The PK was not specified at our model, so we define a simple auto generated PK named *idParty* using *@Id* and *@GeneratedValue*. The variable *telephones* is a *one-to-many* bidirectional relationship, mapped in the inverse side by the *party* variable. The *address* and *email* are embedded instances, and since we specified the mappings at the classes, it is not needed to override this information. Property *name* has a column with *length* of 50. The specializations of *Party* are as follows:

```
@Entity @DiscriminatorValue(value="P") public class Person extends Party{
    @Column(unique=true, length=11) private String cpf;
    //methods... (get/set/...)
}
@Entity @DiscriminatorValue(value="C") public class Company extends Party{
    @Column(unique=true, length=14) private String cnpj;
    //methods... (get/set/...)
```

The *@DiscriminatorValue* annotations tell what values the discriminator column will assume for *Person* and *Company*. The properties *cpf* and *cnpj* are marked as *unique* at the *@Column* annotation, following the model. They function as alternate keys.

4.2.3.2 Using *SqlAlchemy*

We first define the *Table* instance, and its columns, including the *party_type* discriminator column, the PK, all columns needed to persist the embedded objects, and all columns that represent the properties declared at the specializations:

```
Party_table = Table('Parties', metadata,
    Column('id', Integer, primary_key=True), Column('name', String, nullable=False),
    Column('address', String(255)), Column('city', String(30)),
    Column('state', String(2)), Column('email', String(50)),
    Column('party_type', String(1), nullable=False),
    Column('cpf', String, unique = True), Column('cnpj', String, unique = True)
)
```

After that, we declare the *Party* class, and its specialization as follows, without any special ORM methods or annotations:

```
class Party(object):
    #constructor/get/set methods...
class Person(Party):
    #constructor/get/set methods...
class Company(Party):
    #constructor/get/set methods...
```

Finally we declare the instantiation of three mappers, one for each class of the inheritance tree. The mapper of the *Party* declares how to fetch the discriminator, at the *polymorphic_on* clause, by using the *party_type* column of the *Party_table* declared before. Notice that the *Table Party_table* has a property named *c*, that references all columns of this table:

```
PartyMapper = mapper(Party, Party_table, polymorphic_on=Party_table.c.party_type,
    properties={
        'address': composite(Address.Address, Party_table.c.address, Party_table.c.city,
            Party_table.c.state),
        'Email': composite(EMail.Email, Party_table.c.email),
        'telephones': relationship(Telephone, backref = "party" )})
```

The *address* property embeds an *Address*, mapping the columns at the same order of the *composite_values* method. *Email* works in a similar way. Finally, the *telephones* relationship is bidirectional and *one-to-many*, referencing *Telephone*.

SA way of mapping FK associations is to declare a *relationship* property, at the mapper that is referenced by the FK. Declaring a *backref* will add a property (*party*) to the mapping of the class that has the FK column (*Telephone*).

Person and *Company* mappers must refer to the supper class, or its *mapper*, by the `inherits` clause. They also declares the discriminator value with the `polymorphic_identity` clause:

```
PersonMapper = mapper(Person, inherits=PartyMapper, polymorphic_identity='P')
CompanyMapper= mapper(Company, inherits=PartyMapper, polymorphic_identity='C')
```

4.2.3.3 Using ActiveRecord of Ruby

Using *RAR*, we first define the migration that creates the `PARTIES` table, with all columns that belongs to each specialization of *Party*, and all columns that persists the embedded objects:

```
create_table :PARTIES do |t|
  t.string :name, :null => false, :limit=>50
  t.string :party_type, :null => false
  t.string :address
  t.string :city, :limit=>30
  t.string :state, :limit=>2
  t.string :email, :limit=>50
  t.string :cpf, :limit=>11
  t.string :cnpj, :limit=>14
end
```

The PK is implicit, but we have to introduce the discriminator column named `party_type`. The *Address* class maps `address`, `city`, and `state` columns, while the *Email* class maps only one column also named `email`. The properties `cpf` and `cnpj` are mapped as accepting nulls, because they are obligatory depending on the type:

```
class Party < ActiveRecord::Base # RAR does not support abstract super class for FLAT
  self.inheritance_column = "party_type" # discriminator column
  self.table_name = "PARTIES" # table name
  has_many :telephones, :inverse_of => :party
  composed_of :address, :class_name => 'Address',
    :mapping => [%w(address address), %w(city city), %w(state state)]
  composed_of :email, :class_name => 'Email', :mapping => [%w(email email)]
  def self.find_sti_class(type_name) # Customized discriminator
    case type_name
    when "P"
      Person
    when "C"
      Company
    else
      raise "unknown party type"
    end
  end
end
```

Flat inheritance using *RAR* is supported, but specifying what values the `party_type` column should use, to each class, needs some customization. By default, *RAR* will just write the class name at the type column, so we need to override the `find_sti_class` operation telling that *Person* is named “P”, and *Company* is named “C”.

```
class Person < Party
  validates :cpf, :uniqueness => true
  def self.sti_name
    :P # Person is "P"
  end
end

class Company < Party
  validates :cnpj, :uniqueness => true
  def self.sti_name
    :C # Company is "C"
  end
end
```

Party has a bidirectional *one-to-many* relationship with *Telephone*, mapped by `has_many`, with inverse variable named `party` at the *Telephone* class. It has an *Address*

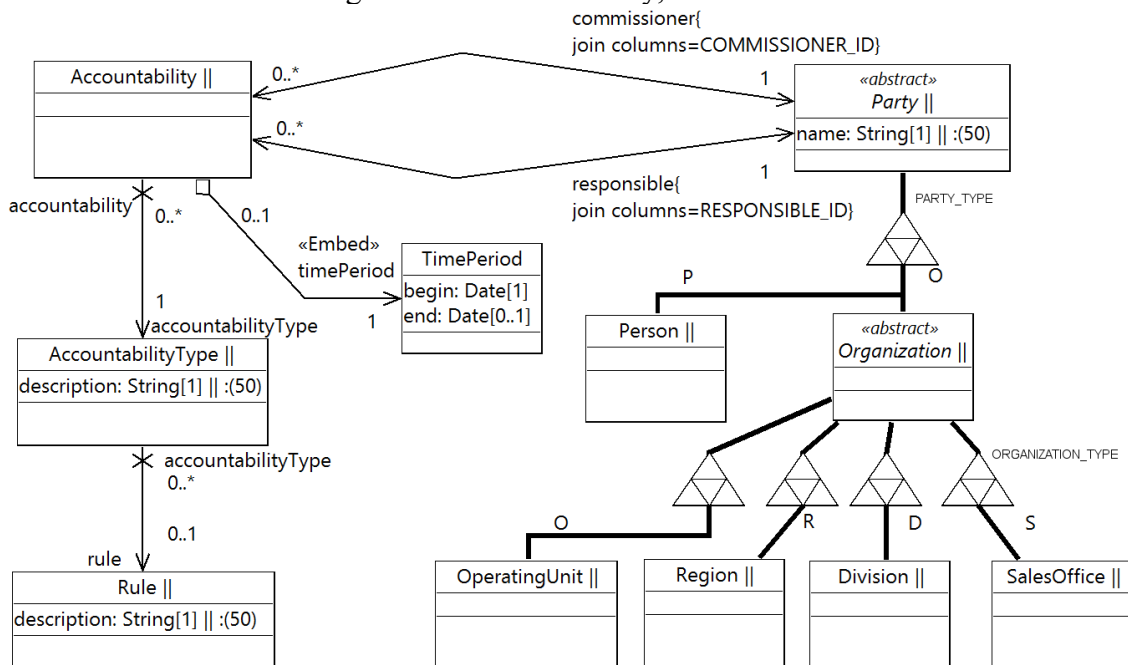
instance, embedded by the *composed_of* that maps the properties of *Address* to the columns with the same name. It also declares the embedded instance of *Email*.

The specializations of *Party* are described as follows, overriding the *sti_name* operation. This operation tells *RAR* that *Person* and *Company* are persisted with *party_type* equals to “P” and “C”.

4.3 Accountability Type Model

This second example focus in the modeling of accountabilities between various parties, following certain rules connecting types of accountabilities and parties. The model of Figure 4.2 presents our model for the *accountability* pattern (FOWLER, 1996), which describes an *Accountability* having a *commissioner* and a *responsible* parties, related to an *AccountabilityType*, at some *TimePeriod*. Each *AccountabilityType* can follow a *Rule* that will validate if the *commissioner* and *responsible* are of the adequate types for that type of accountability.

Figure 4.2: Accountability, first model.



For instance, the *AccountabilityType* “Responsible Division” should accept as *commissioner* only sales offices, and only divisions as *responsible*. It will be related to a *Rule* named “*divisionResponsibleOffice*”, that enforces by code, that the *commissioner* is of type *SalesOffice*, and *responsible* of type *Division*.

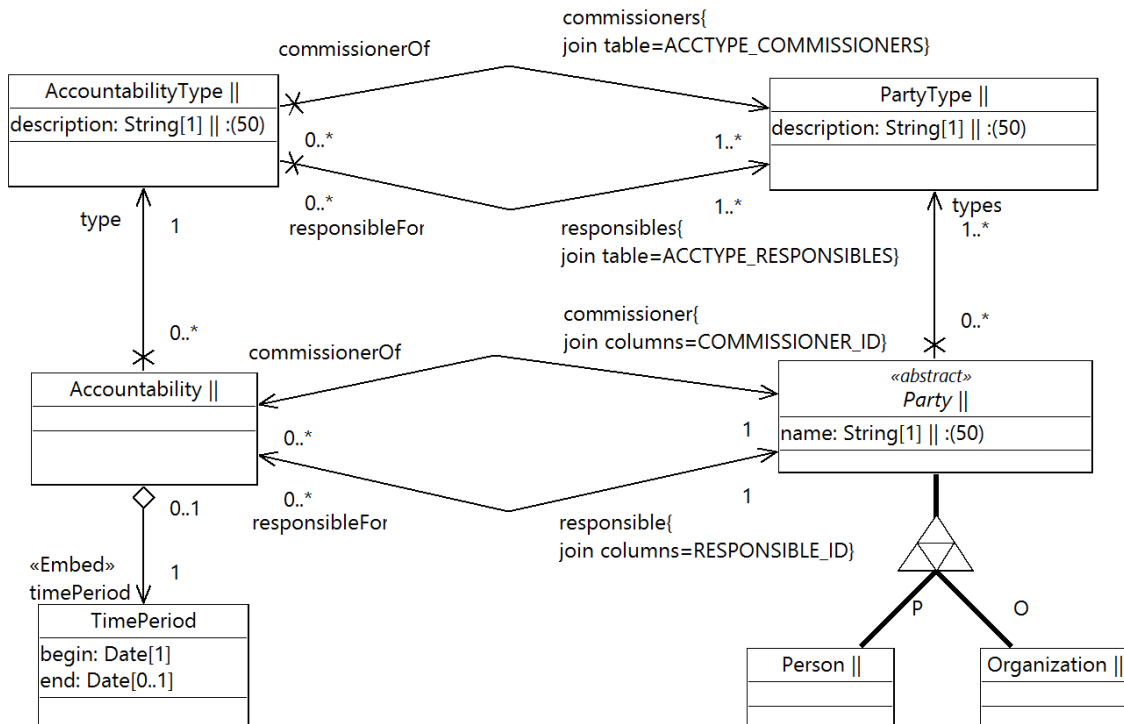
For example, an *Accountability* with type “Responsible Division” could be created between the *Division* “*Serra Gaúcha*”, and the *SalesOffice* of “*Bento Gonçalves*”. The *TimePeriod* sets the *start* of the accountability, and its *end*, or null if it is still valid.

The design of Figure 4.2 puts *TimePeriod* as an embedded relationship of *Accountability*. The specializations of *Party* are mapped using the *Flat* inheritance. The associations *commissioner* and *responsible* are mapped to use specific FKs, defined by

the join column, named COMMISSIONER_ID and RESPONSIBLE_ID. The mappings are simple, without defining specific table names, or PKs.

This model can be improved, by the application of separation of the knowledge, and operational levels, of the *Party Type* pattern, displayed by Figure 4.3. Instead of having a fixed structure, with several specializations of *Party*, we make these types dynamic by introducing the *PartyType* class, and each static subtype of organization is replaced by a dynamic instance of *PartyType*. A *Party* may now relate to one or more types. For example, the “Serra Gaúcha” *Organization* will relate to the “Division” *PartyType*.

Figure 4.3: Accountability with Party type pattern and knowledge level.



The *Rule* class can now be replaced by a dynamic solution, at which the *AccountabilityType* lists what possible types of party can assume the positions of *commissioner* and *responsible*, by the two new *many-to-many* relationships: *commissioners* and *relationships*.

The mapping proposed at Figure 4.3 introduces three new *many-to-many* relationships, mapped by association tables. The *commissioners* relationship is mapped by the join table named ACCTYPE_COMMISSIONERS, and the *responsibles* by the ACCTYPE_RESPONSIBLES table. The *many-to-many types* relationship, between *Party* and *PartyType*, will have a join table without specified name, meaning that the table name will be the default of the ORM tool.

4.3.1 Implementing the Associations

The model introduces the specification of join columns for *many-to-one* associations, and the specification of join tables to implement *many-to-many* associations. The distinction between associations that connect the same pairs of classes is what differentiates this example from the first.

4.3.1.1 Using JPA

The *Association Table* pattern is implemented by using the `@ManyToMany` annotation. The `@JoinTable` annotation can be used to specify the association table, but *JPA* will assume a table following a “default name”, if nothing is specified, as in the following example, where the *types* property represents the association between *Party* and *PartyType*:

```
@Entity @Inheritance(strategy=SINGLE_TABLE) @DiscriminatorColumn(name="PARTY_TYPE")
public abstract class Party {
    @Id @GeneratedValue private int idParty;
    @Column(nullable = false, length=50) private String name;
    @OneToMany(mappedBy="commissioner") private Set<Accountability> commissionerOf;
    @OneToMany(mappedBy="responsible") private Set<Accountability> responsibleFor;
    @ManyToMany private Set<PartyType> types;
    // methods get/set/etc...
}

@Entity public class Accountability {
    @Id @GeneratedValue private int idAccountability;
    @ManyToOne(optional=false) @JoinColumn(name="COMMISSIONER_ID")
    private Party commissioner;
    @ManyToOne(optional=false) @JoinColumn(name="RESPONSIBLE_ID")
    private Party responsible;
    @ManyToOne(optional=false) private AccountabilityType accountabilityType;
    @Embedded private TimePeriod timePeriod;
    // methods get/set/etc...
}

@Entity public class AccountabilityType {
    @Id @GeneratedValue private int accountabilityTypeId;
    @Column(nullable=false,length=50) private String description;
    @ManyToMany @JoinTable(name="ACCTYPE_COMMISSIONERS")
    private Set<PartyType> commissioners;
    @ManyToMany @JoinTable(name="ACCTYPE_RESPONSIBLES")
    private Set<PartyType> responsables;
    // methods get/set/etc...
}

@Entity public class PartyType {
    @Id @GeneratedValue private int idPartyType;
    @Column(nullable = false, length=50) private String description;
    @ManyToMany(mappedBy="commissioners") private Set<AccountabilityType> commissionerOf;
    @ManyToMany(mappedBy="responsibles") private Set<AccountabilityType> responsibleFor;
    // methods get/set/etc...
}
```

In order to specify the join columns that implement the relationships *commissioner* and *responsible*, the *Accountability* class has the `@JoinColumn` annotation, associated to the properties referencing parties. The *commissioner* uses the FK column COMMISSIONER_ID, and the *responsible* uses the FK column RESPONSIBLE_ID.

AccountabilityType has two bidirectional *many-to-many* relationships with *PartyType*, implementing the knowledge level of the model. The `@JoinTable` annotation tells the ORM framework that the association tables are named ACCTYPE_COMMISSIONERS, for the *comissioners-commissionerOf* association, and ACCTYPE_RESPONSIBLES, for the *responsible-responsibleFor* association.

4.3.1.2 Using SQLAlchemy

The first step is to declare the *Table* objects, including the columns that are FKs, and the association tables. The PARTY_PARTY_TYPE *Table* implements the *parties-partytypes* relationship, and the other association tables have the names of the model:

```
Party_table = Table('Party', metadata,...)
Party_Type_table = Table('Party_Type', metadata,...)
PARTY_PARTY_TYPE = Table('PARTY_PARTY_TYPE', metadata,...)
Accountability_table = Table('Accountability', metadata,
    Column('id_accountability', Integer, primary_key=True),
    Column('begin', DateTime, nullable=False),
```



```

Column('end', DateTime),
Column('id_acctype', Integer, ForeignKey('Accountability_Type.id_acctype',
  ondelete='CASCADE'), nullable=False),
Column('id_commissioner', Integer, ForeignKey('Party.id_party')),
Column('id_responsible', Integer, ForeignKey('Party.id_party'))
ACCTYPE_COMMISSIONERS = Table('ACCTYPE_COMMISSIONERS', metadata,
  Column('id_partytype', Integer, ForeignKey('Party_Type.id_partytype'), nullable=False),
  Column('id_acctype', Integer, ForeignKey('Accountability_Type.id_acctype'),
    nullable=False))
ACCTYPE_RESPONSIBLES = Table('ACCTYPE_RESPONSIBLES', metadata,
  Column('id_partytype', Integer, ForeignKey('Party_Type.id_partytype'), nullable=False),
  Column('id_acctype', Integer, ForeignKey('Accountability_Type.id_acctype'),
    nullable=False))
#...

```

The *many-to-many* mappings are declared using the *relationship* property, and the parameter *secondary* specifies the association table name. The join column, specified at the model, is mapped using the *primaryjoin* option of the *relationship*, that specifies the join condition:

```

PartyMapper = mapper(Party, Party_table, polymorphic_on=Party_table.c.type,
  polymorphic_identity='T', properties={
    'commissionersOf' : relationship(lambda: Accountability, backref = "commissioner",
      primaryjoin=(Accountability_table.c.id_commissioner==Party_table.c.id_party)),
    'responsiblesFor' : relationship(lambda: Accountability, backref = "responsible",
      primaryjoin=(Accountability_table.c.id_responsible==Party_table.c.id_party)),
    'types': relationship(PartyType, secondary=PARTY_PARTY_TYPE) })
AccountabilityMapper = mapper(Accountability, Accountability_table, properties={
  'timePeriod': composite(TimePeriod, Accountability_table.c.begin,
    Accountability_table.c.end),
  'accountabilityType': relationship(AccountabilityType) })
AccountabilityTypeMapper = mapper(AccountabilityType, Accountability_Type_table,
  properties={
    'commissioners': relationship(PartyType, secondary=ACCTYPE_COMMISSIONERS, backref =
      "commissionerOf"),
    'responsibles': relationship(PartyType, secondary=ACCTYPE_RESPONSIBLES, backref =
      "responsibleFor") })

```

4.3.1.3 Using ActiveRecord of Ruby

To transparently map the *Association Table* pattern, *RAR* offers the *has_and_belongs_to_many* mapping. In the next code snippet, the *party-partytypes* association of property *types* is mapped to a table named, by *RAR* convention, “*parties_party_types*”:

```

class Party < ActiveRecord::Base
  has_and_belongs_to_many :types, :class_name => 'PartyType'
  has_many :commissionersOf, :class_name => 'Accountability',
    :inverse_of=>:commissioner, :foreign_key => "commissioner_id"
  has_many :responsiblesFor, :class_name => 'Accountability',
    :inverse_of=>:responsible, :foreign_key => "responsible_id"
  #...

```

The associations from *Party* to *Accountability*, mapped by *has_many*, specify the FKs, according to the join columns of the model, by the *foreign_key* option. This is repeated at the other side of the relationship at the *Accountability* class:

```

class Accountability < ActiveRecord::Base
  belongs_to :commissioner, :class_name => 'Party', :inverse_of=>:commissionersOf,
    :foreign_key => "commissioner_id"
  belongs_to :responsible, :class_name => 'Party', :inverse_of=>:responsiblesFor,
    :foreign_key => "responsible_id"
  belongs_to :accountabilityType
  #...

```

The associations between *AccountabilityType* and *PartyType* are also mapped by the *has_and_belongs_to_many*, but with the option *join_table* that specifies the name of the table, according to what we specified at the model:

```

class AccountabilityType < ActiveRecord::Base
  has_and_belongs_to_many :commissioners, :class_name => 'PartyType',
    :join_table=>"acctype_commissioners"
  has_and_belongs_to_many :responsibles, :class_name => 'PartyType',
    :join_table=>"acctype_responsibles"
  #...

```

4.4 Account Model

The Account Model was already presented at the Figure 3.3 of Chapter 3, under the “a not so simple example” section. In this section, we will jump directly to the implementation issues of this model.

4.4.1 Entry is a dependent entity

The *Entry* class has a composite PK with two columns, and each column is a FK to another table. *Entry* is identified by the pair of referenced *Account* and *Transaction* objects.

4.4.1.1 Using JPA

The PK of *Entry* falls in the same case examined at *Telephone*, but the properties that represent the PK at *Entry* must also be annotated as `@ManyToOne` and `@JoinColumn`. The later is required to enforce that the column should not accept updates (PKs cannot be updated):

```

@Entity @IdClass(EntryPK.class) public class Entry {
  @Id @JoinColumn(updatable = false, name = "acct_number", referencedColumnName =
    "number") @ManyToOne private DetailAccount account;
  @Id @JoinColumn(updatable = false, name = "id_transaction", referencedColumnName =
    "id_transaction") @ManyToOne private Transaction transaction;
  //methods... (get/set)
class EntryPK implements Serializable {
  private int account;
  private long transaction;
  //methods... (get/set)

```

The PK properties are of the type of the destination class: *account* is a *DetailAccount*, and *transaction* is a *Transaction*. However, the properties of the PK class *EntryPK* have to be of scalar types compatible with FK types: *account* is an *int*, and *transaction* a *long*. The `@JoinColumn` also tells *JPA* that FK references a column named *number* at *Account*, but is named *acct_number* at the *Entry* table.

4.4.1.2 Using SQLAlchemy

SA easily maps FKs that are also PKs, all that is needed to do is to declare the columns as FKs and PKs when instantiating the *Table*:

```

Entry_table = Table('Entry',metadata,
  Column('acct_number', Integer, ForeignKey('DetailAccount.number'), primary_key=True,
    autoincrement=False),
  Column('id_transaction', Integer, ForeignKey('Transaction.id_transaction'),
    primary_key=True, autoincrement=False), #...

```

4.4.1.3 Using ActiveRecord of Ruby

Once again, we have to resort to the `composite_primary_keys` extension. The mapping is similar to what we have done with *Telephone* (at section 4.2.1.3), with a similar migration for the PKs. The difference is that we also declares the *many-to-one* associations using the `belongs_to` mapping:

```

class Entry < ActiveRecord::Base
  self.primary_keys = [:acct_number, :id_transaction]
  belongs_to :account, :class_name=>"DetailAccount", :foreign_key => 'acct_number',
    :inverse_of=>:entries
  belongs_to :transaction, :foreign_key => 'id_transaction', :inverse_of=>:entries

```

Differently from *JPA* and *SA*, the *acct_number* and *id_transaction* are also exposed properties of *Entry* instances. This means that a developer can assign the *account* of an *Entry* and/or the *acct_number* of the same *Entry*, what may lead to an error if they do not match the same object.

4.4.2 Account mapped by two tables

The *Account* class is mapped by the tables *Account* and *Act_brief*. Some properties are persisted at *Account*, and others at *Act_brief*.

4.4.2.1 Using JPA

The `@SecondaryTable(s)` annotation allows the specification of one or more tables that are joined with the main table, in order to persist the properties of a class. At the code of *Account*, properties mapped to *Act_brief* must be annotated by `@Column`, or `@JoinColumn`, identifying the destination *table*.

```
// omitted inheritance mappings ...
@Entity @SecondaryTable(name = "Act_Brief")
public abstract class Account {
    @Id @GeneratedValue private int number;
    @Column(name = "dt_calc", table = "Act_Brief") @Temporal(TemporalType.DATE)
    private Date dtBalance;
    // ... omitted other properties
```

The PK property *number* is persisted at the main table, with the same name of the class. But at our model, the *dtbalance* is mapped as “*Act_brief.dt_calc*”, meaning that it is persisted at the table *Act_brief*. This is implemented by the *table* option.

4.4.2.2 Using SQLAlchemy

SA has the concept of *join* object that represents a data source with multiple tables. First we declare the *Account* and *Act_brief* tables, and then we instantiate a join named *JAC*, based upon both tables:

```
Account_table = Table('Account', metadata,
    Column('number', Integer, primary_key=True), #... other columns
)
Balance_table = Table('Act_brief', metadata,
    Column('number', Integer,
        ForeignKey('Account.number'), primary_key=True, autoincrement=False),
    #... columns of Act_brief
)
JAC = join(Account_table, Balance_table)
```

The *number* column of the *Act_brief* table is the PK and a FK to *Account*. The *JAC* join is then used as the source for the mapper. The *number* property is declared using a *column_property* instance, that allows the mapping of one property to two columns. This is necessary to ensure that the *number*, at *Account* and *Act_brief* tables, is always the same for the same instance:

```
AccountMapper = mapper (Account, JAC, #...inheritance omitted
    'number': column_property(Account_table.c.number, Balance_table.c.number) #...
```

4.4.2.3 Using ActiveRecord of Ruby

There is no way to map one class to two tables, and two tables to just one class, using *RAR*. What can be done is to have two persistent classes, and make one of the two classes reference the other, dispatching the method calls using the *Bridge* pattern (GAMMA et al., 1994).

At our example, *Account* and *Act_brief* would be classes that inherit from *ActiveRecord::Base*, and have a *one-to-one* association. Each property persisted at *Act_brief*, would have an *accessor* on *Account*, dispatching the implementation at the *Act_brief* instance. The constructor of *Account* will have to create an *Act_brief*, and the association will have to cascade deletes.

An alternative of directly bridging each property is to resort to introspection and meta-programming. Our solution to the problem was to create a module that deals with the bridging automatically, by listening to the *method_missing* event, and trying to pass it out to the associated class. First we declare the classes and the mappings, as bellow:

```
class ActBrief < ActiveRecord::Base
  belongs_to :account, :class_name =>"Account", :foreign_key => 'number'
  # ... omitted other mappings
end
class Account < ActiveRecord::Base
  include MixinMod # Emulating secondary table act_brief using MIX IN
  @@mixin=MixinDesc.new(Account,"actbrief",nil) # Configure the mix in
  self.primary_key = "number"
  has_one :actbrief, :class_name =>"ActBrief", :foreign_key => 'number', :dependent =>
    :destroy, :inverse_of=>:account
  # ... omitted other mappings
end
```

The *ActBrief* has a *one-to-one* relationship with *Account*, declared by one side by *belongs_to*, and the other by *has_one* keyword. The module *MixinMod*, responsible to build the bridge, is included, and a mapping is instantiated between *Account* and *actbrief* property, and stored at the class variable *mixin*. The *MixinMod* will listen to the missing methods, and dispatch them to the *actbrief* association.

4.4.3 Vertical Inheritance of Account

The following subsections present the mappings for each studied platform.

4.4.3.1 Using JPA

When the strategy of *@Inheritance* is set to *JOINED*, each class in the inheritance tree has (at least) one table responsible for the persistence. The discriminator column is optional in that case, but following the specification of the model, it can assume *D* for *DetailAccount* or *S* for *SummaryAccount*, and is similar to the *Flat* case of *Party*.

```
@Entity @Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name = "type", discriminatorType = DiscriminatorType.STRING,
length = 1)
public abstract class Account implements { //... omitted
}
@Entity @DiscriminatorValue("D") public class DetailAccount extends Account {
//... omitted
}
@Entity @DiscriminatorValue("S") public class SummaryAccount extends Account {
//... omitted
}
```

4.4.3.2 Using SQLAlchemy

Each *Table*, of the specialization classes, is instantiated with a FK connecting the PK to the *Account* table:

```
SummaryAccount_table = Table('SummaryAccount',metadata,
Column('number', Integer, ForeignKey('Account.number', use_alter=True,
name='fk_summary_acct'), primary_key=True))
DetailAccount_table = Table('DetailAccount',metadata,
Column('number', Integer, ForeignKey('Account.number', use_alter=True,
name='fk_detail_acct'), primary_key=True))
```

The mappers are instantiated in a similar way of the *Flat* inheritance, but it specifies the tables that persist each subclass. The *inherits* option points to the general class:

```
AccountMapper = mapper (Account, JAC, polymorphic_on=Account_table.c.type #...
SummaryAccountMapper = mapper(SummaryAccount, SummaryAccount_table, inherits=Account,
  polymorphic_identity='S', #...
DetailAccountMapper = mapper (DetailAccount, DetailAccount_table, inherits=Account,
  polymorphic_identity='D', #...
```

4.4.3.3 Using ActiveRecord of Ruby

RAR does not support *Verical* inheritance. Once again, one possible implementation is to declare the classes separated, without inheritance, and resort to an association combined with the *Bridge* pattern. *RAR* offers the *polymorphic* association, that can deal with objects of distinct types at the same association:

```
class Account < ActiveRecord::Base
  #... Omitted declarations
  belongs_to :impl, :polymorphic=>true, :foreign_key=>'number', :dependent=>:destroy
```

Now the *Account* instance belongs to the specializations that implement the class, that can be of type *SummaryAccount* or *DetailAccount*. The *polymorphic* option tells the framework to use a *discriminator* column, defined at the migration, to specify what kind of *Account* is related by the *impl* association. However, *RAR* requires a column that follows a specific name, and is not flexible about the discriminator values.

```
create_table :accounts, :primary_key => 'number' do |t|
  t.string :impl_type #...
```

The column *impl_type* will assume a value equal to the name of the class, and we could not use the discriminator values entered at the model (D or S). The remaining class will be as follows:

```
class DetailAccount < ActiveRecord::Base # It does not extends Account!
  include MixinMod # Module that implements the bridge
  @@mixin=MixinDesc.new(DetailAccount,"account","number")
  self.primary_key = "number";
  has_one :account, :as => :impl, :autosave => true, :foreign_key => 'number'
  # ...omitted
class SummaryAccount < ActiveRecord::Base
  include MixinMod
  @@mixin=MixinDesc.new(SummaryAccount,"account","number")
  self.primary_key = "number"
  has_one :account, :as => :impl, :autosave => true, :foreign_key => 'number'
  # ...omitted
```

Notice that if *SummaryAccount* or *DetailAccount* inherits from *Account*, the framework will map all classes, its properties, and association ends, to the same table. Therefore, the specializations could not inherit from *Account*.

The *has_one* mapping to *account* has the “*as => :impl*” mapping telling the framework that it is the polymorphic inverse of *impl*. Both *SummaryAccount* and *DetailAccount* have a reference to *account*, and the *MixinDesc* is initialized mapping each class to the *account* association. The third attribute is the *number* key, and tells the *MixinMod* that the class must first instantiate and save the *Account*, obtain the number, and assign this to the key of the specialization. The implementation of *MixinMod* is described at the Appendix E.

4.4.4 Properties and columns with distinct names

ENORM allows the specification of a property persisted at a column with different name, by using the *ColumnMapping* applied at the property. The *Account* class maps the property *dtBalance* to a column named *dt_calc* at Figure 3.3.

4.4.4.1 Using JPA

It is trivial to specify a scalar property with a distinct name of the column that persists it. At the example of section 4.4.2.1, *dtBalance* is mapped to a column named *dt_calc* by the `@Column` annotation.

4.4.4.2 Using SQLAlchemy

Properties with a distinct name are declared at the mapper, pointing the column at the table object. The following snippet maps *dtBalance* to *dt_calc* at *Balance_table*:

```
AccountMapper = mapper (Account, #... omitted
    properties={ #...
        'dtBalance':Balance_table.c.dt_calc,#...
```

4.4.4.3 Using ActiveRecord of Ruby

RAR does not support the mapping of a column with a distinct property name. A workaround would be to declare another property named *dtBalance*, at the *Account* class, that has keeps the same value of the *dt_calc*:

```
alias_attribute :dtBalance, :dt_calc
```

The *alias_attribute* core method declares the property as a synonym property. However, using the *alias_attribute* will not hide the *dt_calc* property, and *Account* will have two properties accessors that reflect the same value, *dt_calc* and *dtBalance*.

4.4.5 Overrides and Embedded objects referencing persistent classes

The *Quantity* class is not persistent, but references a persistent class that represents its *Unit*. This section focus on the mappings of the embeddment of *Quantity*, and the possibility of overriding its mappings.

4.4.5.1 Using JPA

The *Quantity* class can be annotated with `@ManyToOne` at the *unit* property, and the ORM framework will know that should create the FK at the classes that embed *Quantity*:

```
@Embeddable public class Quantity {
    @Column(precision=20, scale=2) private BigDecimal amount;
    @ManyToOne private Unit unit;
    //...omitted
public class Entry { //...omitted
    @Embedded private Quantity quantity;
```

The *Entry* class embeds *Quantity* by the *quantity* association end. The *Account* class, at the model of Figure 3.3, also embeds *Quantity* by the *balance* association end, but has an override section specifying that the *Act_brief* is responsible for the persistence of both columns. This is implemented in *JPA* by the `@AttributeOverride(s)` and `@AssociationOverride(s)` annotations:

```
@Embedded
@AttributeOverride(name="amount", column=@Column(name="value", table="Act_Brief"))
@AssociationOverride(name="unit", joinColumns=@JoinColumn(name="unit", table="Act_brief"
)) private Quantity balance = new Quantity();
```

The *balance* property has an override at the attribute named *amount*, mapping it to the column named *value* at the secondary table *Act_brief*. It also has a mapping override at the association named *unit*, to use a column with the same name, persisted at the table *Act_brief*.

4.4.5.2 Using SQLAlchemy

The columns that persist the composite are specified at the *Table* instance that persist the owner class:

```
Balance_table = Table('Act_brief', #...
    Column('value', Numeric(20,2), nullable=True),
    Column('unit', String(15), ForeignKey('Currency.unit'), nullable=True) #...
Entry_table = Table('Entry', metadata, #...
    Column('amount', Numeric(20,2), nullable=False),
    Column('unit', String(15), ForeignKey('Currency.unit'), nullable=False) #...
AccountMapper = mapper (Account, #...
'balance': composite(Quantity, Balance_table.c.value, Balance_table.c.unit), #...
EntryMapper = mapper (Entry, #...
'quantity': composite(Quantity, Entry_table.c.amount, Entry_table.c.unit),
```

However, *SA* does not recognize the association of a composite with a persistent class transparently. It will instantiate the *Quantity* composite using the PK of *Unit*, instead of the *Unit* itself. One solution is to create a *listener* method that enforces that the value of the *unit* if indeed a *Unit*:

```
@event.listens_for(Entry.quantity, 'set')
@event.listens_for(Account.balance, 'set', propagate=True)
def loadUnit(target, value, oldvalue, initiator):
    if not (isinstance(value.unit, Unit)):
        session = Session.object_session(target) #get the session of the target
        value.unit = session.query(Unit).get(value.unit)
```

SA allows the specification of methods that are called when certain events happens. In the above example, the set event is called anytime the specified property is changed, including when the object is loaded. The method checks if the unit is a *Unit*, if not, it is the PK of the unit, and this value is used to query the real *Unit*.

4.4.5.3 Using ActiveRecord of Ruby

The *composed_of* mapping can declare the mappings between each property of the *Embedded* class, and the column that persists this property at the persistent class. But the default behavior is to deal with the PK of the *Unit*, and not the *Unit* itself. This can be overridden at the mapping, by defining the *constructor* method. This method is responsible for the instantiation of *Quantity* objects:

```
class ActBrief < ActiveRecord::Base #...omitted mappings
  composed_of :balance, :class_name => 'Quantity',
  :mapping => [ ["value", "amount"], ["unit", "unit"] ],
  :constructor => Proc.new { |amount, unit| (unit==nil) ? nil :
    Quantity.new(amount, Unit.find(unit)) }
#...omitted
class Entry < ActiveRecord::Base
  composed_of :quantity, :class_name => 'Quantity',
  :mapping => [ ["amount", "amount"], ["unit", "unit"] ],
  :constructor => Proc.new { |amount, unit| (unit==nil) ? nil :
    Quantity.new(amount, Unit.find(unit)) }
#...omitted
```

The constructor takes the *unit* PK and queries the framework for the *Unit* object, passing it to the constructor of *Quantity*. If the unit PK is nil, the resulting *Quantity* will also be nil.

4.4.6 The Account-Entry association

The *account-entry* association is abstract, but at OO languages, associations are not explicitly declared: they are implemented by association ends. Associations ends are properties, and both properties and instance variables cannot be abstract. Abstract

associations can, however, be declared as abstract accessor operations, implemented at the specialization classes. This section examines how the ORM frameworks deals with this situation.

4.4.6.1 Using JPA

It is possible to declare an abstract operation *getEntries()*, that access the *entries* association end, at the *Account* class. This operation is implemented by *DetailAccount* class, that declares the instance variable named *entries*, mapping it as the inverse of the *account* property at the *Entry* class:

```
public abstract class Account { //...omitted code
    public abstract List<Entry> getEntries(); //...
}
public class DetailAccount extends Account { //...omitted code
    @OneToMany(mappedBy="account") private List<Entry> entries;
    @Override public List<Entry> getEntries() {
        if (entries == null)
            entries = new ArrayList<Entry>();
        return entries;
    } //...
}
public class SummaryAccount extends Account { //...omitted code
    @Override public List<Entry> getEntries() {
        List<Entry> listEntry = new ArrayList<Entry>();
        for (Account acct: components) {
            listEntry.addAll(acct.getEntries());
        }
        return listEntry;
    } //...
}
```

The association *summary-entries* was redefined at the *entries* end to be *readOnly* and *Transient* (not persistent). It is also specified as a derived union of the sum of entries of all components. This is implemented at the class *SummaryAccount* by making *getEntries()* a recursive method, that calls itself at each component *Account*. Notice that *SummaryAccount* has no mapped association to *Entry*, and vice-verse.

4.4.6.2 Using SQLAlchemy

The properties were not declared ahead, because *Python* is a dynamic language. The problem is that if we declare *entries* as an abstract method, the interpreter will ask for this implementation before the instrumentation by the *SA* framework.

On the other hand, because *Python* is a dynamic language, it will not enforce if *entries* was declared at the abstract super-class. Nevertheless, we declared the *entries* operation as a placeholder at the super class, as a mapping at the *DetailAccount*, and as an operation at *SummaryAccount* class:

```
class Account(object): #other operations are omitted
    __metaclass__ = abc.ABCMeta #This is an abstract class
    def entries(self):
        pass
#... at the mappings.py
DetailAccountMapper = mapper (#...
properties={ 'entries': relationship(lambda: Entry, backref = "account",#...
#... at SummaryAccount.py
class SummaryAccount(Account): #other operations are omitted
    def __get_entries(self):
        ret = []
        for acct in self.components:
            for ent in acct.entries:
                ret.append(ent)
        return ret

    entries = property(__get_entries)
```


The redefinition of *entries* at *SummaryAccount* used the *property* object of *Python*, that maps a method as an accessor for a property. Notice that if we remove the *entries* operation at *Account*, it will make absolutely no difference, as long as *Account* is abstract.

4.4.6.3 Using ActiveRecord of Ruby

As a consequence to our workaround to implement *Vertical* inheritance, *Account* is not an abstract class. The implementation of *entries* at *Account* just forwards the call to the related specialization (*impl.entries*).

```
class Account < ActiveRecord::Base # ... omitted details
  def entries
    impl.entries
  end
class DetailAccount < ActiveRecord::Base
  has_many :entries, :foreign_key =>:acct_number, :inverse_of=>:account
  # ... omitted details
class SummaryAccount < ActiveRecord::Base
  has_and_belongs_to_many :components, :class_name=>'Account', :join_table=> "act_comps"
  # ... omitted details
  def entries
    ret = []
    components.each{|c|ret+=c.impl.entries}
    return ret
  end
```

The *DetailAccount* class implements *entries* as the association to *Entry* by the *has_many* mapping. The *SummaryAccount* implements *entries* as a read-only value calculated from the recursive concatenation of the *entries* of all *components*. The *components* is a *many-to-many* association declared by *has_and_belongs_to_many* to the *Account* class. The returning collection (*ret*), at the *entries* method, is populated by the sum of all *entries*, of every component of the *components* association.

4.5 Resource Allocation Model

The resource allocation pattern models the relationship between an *Action*, and the resources necessary to execute this action. These resources can be allocated in a general way, referencing the type of resource, or in a specific way, referencing a resource of some type.

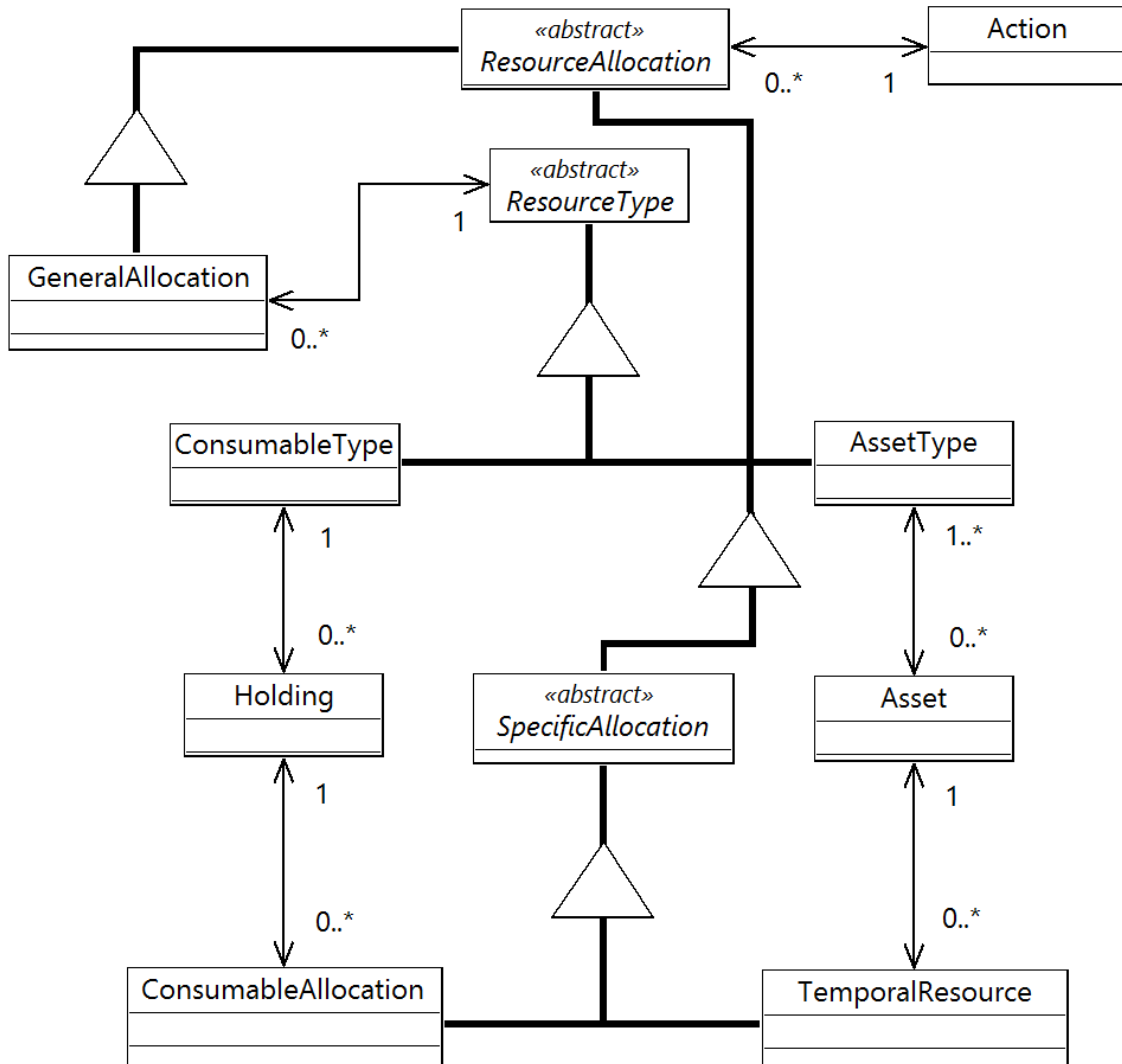
Figure 4.4 presents the conceptual model for the resource allocation pattern, based upon the book of Fowler (FOWLER, 1996). *Action* relates to zero or more *ResourceAllocation* objects. A *ResourceAllocation* can be a *GeneralAllocation* or a *SpecificAllocation*. The *GeneralAllocation* will reference a *ResourceType*. The *ResourceType* can be a *ConsumableType*, such as *gas*, or an *AssetType*, such as a *bus*.

The *SpecificAllocation* can be a *ConsumableAllocation*, or a *TemporalAllocation*. The *ConsumableAllocation* will refer to a *Holding* of a specific *ConsumableType*, such as *gas* from *station five*. The *TemporalAllocation* will refer to the specific allocation of an *Asset*, such as the *bus* with license “AAA 5555”.

Once understood the basic model, we now expand this example a little bit to the design presented by Figure 4.5. This design specializes assets and asset types into equipment and human resources. The *AssetType* class is specialized at *Post* and *EquipmentType*. The *Asset* is specialized at *Employee* and *Equipment*. The *many-to-many* association *assets-assetTypes* has the endings redefined by *employees-posts* and *equipments-equipmentsTypes* associations.

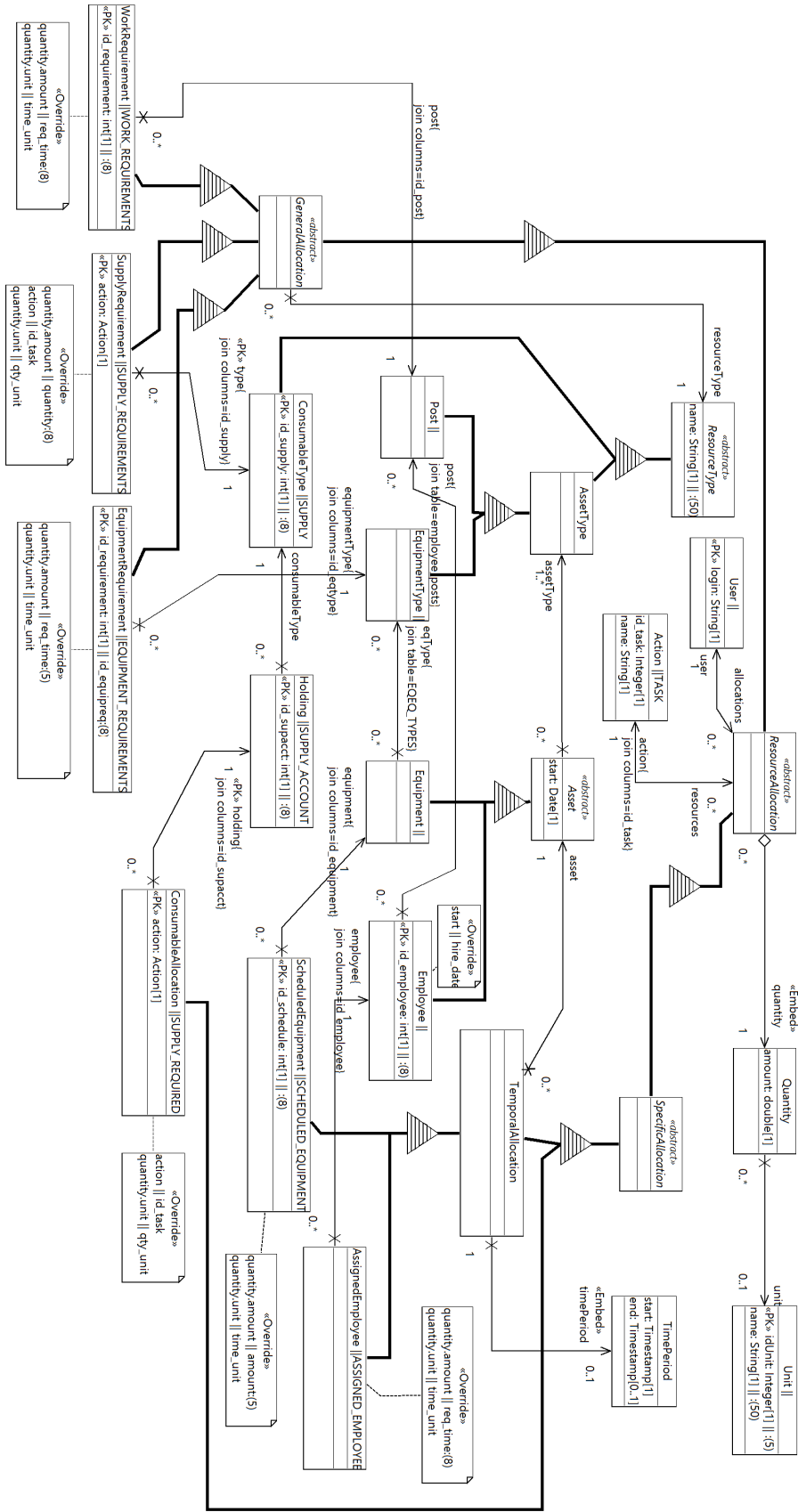
New specializations of resource allocations were created to deal with this new tree of resource types. The *GeneralAllocation* now has specializations dealing with each *ResourceType*, named *WorkRequirement* for posts, *SupplyRequirement* for consumables, and *EquipmentRequirement*. The *TemporalAllocation* is specialized into *ScheduledEquipments* and *AssignedEmployee*.

Figure 4.4: Conceptual model for the Resource Allocation Pattern.



The *ResourceAllocation* embeds a *Quantity* instance, similar to the introduced by the *Account* example, that refers to the *Unit*. The specializations later override the mappings of *Quantity*, specifying adequate names and types, according to the quantitative or time based nature of the allocation. Another examples of overrides are the *Employee* class, that overrides the inherited property *start* with a distinct column name, and the overriding of the association of *action-allocations*, to specify the FK column at the specializations of *ResourceAllocation* that deals with *Consumables*.

Figure 4.5: Resource Allocation ENORM model.



This design of this example uses the *horizontal* inheritance with abstract, non persistent, super classes. Each leaf specialization is an independent table, a design that reflects a legacy database being mapped to a framework, where the more general classes were not persistent. However, the specializations should not ignore the persistence of the inherited classes, hence the *horizontal* strategy is applied at the entire tree.

ResourceAllocation, *GeneralAllocation*, and *SpecificAllocation* are abstract classes nor marked as persistent. *TemporalAllocation* is not abstract, but it is also not persistent. All generalizations are not persistent.

4.5.1 Horizontal Inheritance at the Resource Allocation Tree

The application of *Horizontal* inheritance, with non persistent super classes, is not the typical application of the *Concrete Table* pattern.

4.5.1.1 Using JPA

When the super class is persistent, the implementation uses the `@Inheritance` annotation with the `TABLE_PER_CLASS` strategy, but if the super class is not persistent, the `@MappedSuperclass` mapping is the way to declare the mappings. The following implementation uses the annotation before the methods, instead of the annotations before the properties, used at the previous examples.

```
@MappedSuperclass public abstract class ResourceAllocation {
    @Embedded
    public Quantity getQuantity() { //..
    @ManyToOne @JoinColumn(name="id_task")
    public Action getAction() { //..

    @MappedSuperclass public abstract class GeneralAllocation extends ResourceAllocation {
        @Transient abstract public ResourceType getResourceType(); //..

    @Entity @Table(name="WORK_REQUIREMENTS") //..
    public class WorkRequirement extends GeneralAllocation {
        @Id @Column(name="id_requirement",precision=8)
        public int getRequirementId() { //..
        @ManyToOne @JoinColumn(name="id_post")
        public Post getResourceType() { //..
```

The *ResourceAllocation* embeds the allocated quantity, and maps the *many-to-one* association to the *Action*. *GeneralAllocation* declares the association to *ResourceType* as abstract. However, the *ResourceAllocation* and *GeneralAllocation* are not persistent, and not annotated with `@Entity`. Instead, they are annotated by `@MappedSuperclass`, that allows the writing of mappings, on non persistent classes intended to have persistent specializations.

The *WorkRequirement* class specializes *GeneralAllocation*, and is persistent. It first declares its PK property, named *id_requirement* at the model, and then overrides the *getResourceType* method that implements the *generalAllocation-resourceType* association to return a *Post*, using the specified join column.

The super class does not need to be abstract. The *TemporalAllocation* class is not abstract, and is also mapped as a `@MappedSuperclass`. The *AssignedEmployee* extends *TemporalAllocation*, and is persisted, as in the next implementation:

```
@MappedSuperclass public abstract class SpecificAllocation extends ResourceAllocation{
    @MappedSuperclass public class TemporalAllocation extends SpecificAllocation{ //..
    @Entity @Table(name="ASSIGNED_EMPLOYEE")
    public class AssignedEmployee extends TemporalAllocation { //..
        @Id @Column(name="ID_ASSIGNMENT") public int getIdAssignment() {
```

4.5.1.2 Using SqlAlchemy

SqlAlchemy offers the *polymorphic_union* mapping that allows the implementation of generalizations, and the *concrete* mapper option that implements the *horizontal* inheritance. Non persistent concrete classes are mapped without a table. The *polymorphic_union* lists the tables, assigning aliases to each table, that compose the union to query the super classes:

```

Work_Requirements_table = Table('Work_Requirements',metadata,
    Column('id_requirement',Integer(8),primary_key=True),
    Column('id_post',Integer(8),ForeignKey('Post.id_post'),nullable=False),
    Column('id_task',Integer(8),ForeignKey('Task.id_task'),nullable=False)#...
)# Similar tables: Supply_Requirements, Equipment_Requirements, Supply_Required,
Scheduled_Equipment, Assigned_Employee (omitted)
# Mappings:
pjoin = polymorphic_union({ # All resource allocations
    'srequirements': Supply_Requirements_table,
    'wrequirements': Work_Requirements_table,
    'erequirements': Equipment_Requirements_table,
    'consumables': Supply_Required_table,
    'sequip':Scheduled_Equipment_table,
    'aemployee':Assigned_Employee_table}, 'type', 'pjoin')

ResourceAllocationMapper = mapper(ResourceAllocation,pjoin,with_polymorphic=('**',
pjoin),polymorphic_on=pjoin.c.type,properties={
    'action':relationship(Action), 'user':relationship(User)})
genJoin = polymorphic_union({ # General allocations
    'srequirements': Supply_Requirements_table,
    'wrequirements': Work_Requirements_table,
    'erequirements': Equipment_Requirements_table}, 'type', 'genJoin')
GeneralAllocationMapper = mapper(GeneralAllocation,genJoin,concrete=True,
    inherits=ResourceAllocationMapper,with_polymorphic=('**',genJoin),
    polymorphic_on=genJoin.c.type)
WorkRequirementMapper = mapper(WorkRequirement,Work_Requirements_table,concrete=True,
    inherits=GeneralAllocationMapper,polymorphic_identity='wrequirements',properties={
    'action': relationship(Action), #...
    'post': relationship(Post) #...

```

First we declare the tables, and then the mappers for each class that participates at the *horizontal* inheritance, based on unions. The leaf classes are mapped with a *polymorphic_identity* that tells the framework what subset of the super class union is identified with the source. *WorkRequirement* is mapped to *Work_Requirements_table*, identified at the union as *wrequirements*.

4.5.1.3 Using ActiveRecord of Ruby

There is no *horizontal* inheritance on *RAR*. However, it is possible to declare a mapped super class, with its specializations persisting the attributes at its own tables, by setting the class variable *abstract_class* to true.

```

class ResourceAllocation < ActiveRecord::Base
  self.abstract_class=true
  attr_accessor :quantity
  belongs_to :action, :foreign_key => :id_task # omitted...
class GeneralAllocation < ResourceAllocation
  self.abstract_class=true
  attr_accessor :resourceType # omitted...
class WorkRequirement < GeneralAllocation
  self.table_name = "work_requirements"
  belongs_to :post, :foreign_key => 'post_id'
  composed_of :quantity, :class_name => 'Quantity',
    :mapping => [{"req_time", "amount"}, {"time_unit", "unit_id"}],

```

Despite the name, *abstract_class* does not make the class abstract, but tells the persistence framework that it should not enforce single table inheritance. It is a property inherited from the *ActiveRecord::Base* class, and not a Ruby language feature. However, trying to instantiate a class with *abstract_class=true* will raise exceptions at the framework, that will try to issue SQL commands to null tables. Concrete classes such as *TemporalAllocation* are, in practice, turned abstract by this solution.

The properties of the super-classes can be declared using the *attr_accessor* operation of *Ruby*. It is important to notice that, when there are two properties, with the same name at the general and specialization classes, *Ruby* treat both as the same. Each instance of *WorkRequirement* will have only one quantity, despite the declaration at the superclass. This is distinct from Java, where there will be two *quantity* properties, one at each context.

4.5.2 Overriding inherited properties and associations

The resource model overrides properties, and redefine associations at the specializations. Moreover, the PK are not uniform among the persistent specializations.

4.5.2.1 Using JPA

The transient *Asset* class specifies a property named *start* that is persisted by horizontal specializations such as *Equipment* and *Employee*. However, the *Employee* table persists the *start* at a column named *hire_date*, what is represented, at the model, by the override of *start*. The following code snippet implements this override with the *@AttributeOverride* mapping referencing an inherited property:

```
@MappedSuperclass public abstract class Asset {
    Date start;
    public Date getStart() {//..

@AttributeOverride(name="start", column=@Column(name="HIRE_DATE"))
@Entity @Table(name="EMPLOYEE") public class Employee extends Asset{
    //..
```

Usually, the override of associations of the super class, such as the *action-resources* by the *SupplyRequirement* class, is implemented by an *AssociationOverride*, in a similar way that associations belonging to embedded objects, such as the association *quantity-unit*. However, when using *horizontal* inheritance, we may specify a composite key for the specialization where the overwrote association participates. *SupplyRequirement* overrides *action* to make it part of the PK:

```
@Entity @Table(name="SUPPLY_REQUIREMENTS") @IdClass(SupplyRequirementPK.class)
@AttributeOverride(name="quantity.ammount", column=@Column(name="QUANTITY"))
@AssociationOverride(name="quantity.unit", joinColumns=@JoinColumn(name="QTY_UNIT"))
public class SupplyRequirement extends GeneralAllocation{
    ConsumableType supply;
    @Id @ManyToOne
    @JoinColumn(name="ID_TASK", insertable=false, updatable=false)
    public Action getAction() {
        return super.getAction();
    }
    @Id @ManyToOne
    @JoinColumn(name="ID_SUPPLY")
    public ConsumableType getResourceType() {
        return supply;
    } //.. setters and PK class omitted
```

At section 4.5.1.1, the *ResourceAllocation* class has a mapped association to *Action*, implemented by operation *getAction*. The *SupplyRequirement* class overrides the method with a new mapping, that makes this property part of the PK, what cannot be achieved by a simple *@AssociationOverride*. The tricky part here is that this override does not declares a new local variable *action*, but just passes the getter to the super class. This only works because the mappings are on the methods, and not on the properties.

4.5.2.2 Using *SqlAlchemy*

Because the table and the mappers are separated, it is easy to specify the detailed columns for each class. The relationships can be specified as general associations at the super class. For instance, the *action-resourceAllocations* association can be specified at *ResourceAllocation* and refined at its subclasses:

```
ActionMapper = mapper(Action, Task_table, properties={
    'resources':relationship(ResourceAllocation, collection_class=set,
        back_populates='action') #...
ResourceAllocationMapper = #...
    'action':relationship(Action), #...
WorkRequirementMapper = #...
    'action':relationship(Action, primaryjoin=(Task_table.c.id_task==Supply_Required_tab
        le.c.id_task), back_populates='resources') #...
```

The *SupplyRequirement* class overrides the action property to became part of its PK. *SA* will not pay much attention to the super class mappings, as long as they are abstract and, because of that, used only for queries. We just have to declare the PK at the Table of *SupplyRequirement*, and write the mappings with the correct join columns:

```
Supply_Requirements_table = Table('Supply_Requirements', metadata,
    Column('id_supply', Integer(8), ForeignKey('Supply.id_supply'), primary_key=True),
    Column('id_task', Integer(8), ForeignKey('Task.id_task'), primary_key=True)
    Column('quantity', Numeric(8,2), nullable=False),
    Column('qty_unit', Integer(5), ForeignKey('Unit.id_unit'), nullable=False), #...

SupplyRequirementMapper = #...
    'action':relationship(Action, primaryjoin=(Task_table.c.id_task==Supply_Requirements
        table.c.id_task), back_populates='resources')
    'type':relationship(ConsumableType, primaryjoin=(Supply_table.c.id_supply==
        Supply_Requirements_table.c.id_supply))
    '__quantity':Supply_Requirements_table.c.quantity,
    '__unit':relationship(Unit, primaryjoin=(Unit_table.c.id_unit==
        Supply_Requirements_table.c.qty_unit) ),
    "quantity":composite(Quantity, Supply_Requirements_table.c.quantity,
        Supply_Requirements_table.c.qty_unit), #...
```

Notice that when composite name conflicts with the composite columns, we have to map the columns with distinct names. The double underscore before the property (“__<prop name>”) tells python to obfuscate the property, since we want to access *quantity* and *unit* only by the composite *Quantity* instance, and not directly from the *SupplyRequirement*.

4.5.2.3 Using *ActiveRecord of Ruby*

The implementation can declare the properties at the super class using *attr_accessor* operation, and link renamed properties at the specializations with alias:

```
class Asset < ActiveRecord::Base
  self.abstract_class=true
  attr_accessor :assetType, :start #...
class Employee < Asset
  alias_attribute :start, :hire_date
```

```
alias_attribute :assetType, :posts
has_and_belongs_to_many :posts, :join_table=>"employee_posts" #...
```

RAR can deal independently with the property mapped to columns, and associations. The *SupplyRequirement* class inherits the association to *Action* from the *ResourceAllocation* class, but it also declares the column *id_task* as part of the PK:

```
class SupplyRequirement < GeneralAllocation
  self.table_name = "supply_requirements"
  self.primary_keys = [:id_task, :supply_id]
  belongs_to :resourceType, :class_name =>:ConsumableType, :foreign_key =>:supply_id
  composed_of :quantity, :class_name => 'Quantity',
    :mapping => [[:quantity, "amount"], [:qty_unit, "unit_id"]],
```

4.5.3 Association to general classes with horizontal specializations

The *Action* class has a bidirectional association to *ResourceAllocation*. This association, however, implies six different FKs from the six leaf specializations of *ResourceAllocation*, all pointing to *Action*.

4.5.3.1 Using JPA

MappedSuperclasses cannot participate on queries, and *JPA* does not contains a solution to a bidirectional mapping from *Action* to *ResourceAllocation*. It allows the mapping of the association from the *MappedSuperclass* to an *Entity*, but not the opposite direction.

A solution to this problem is to map each leaf association as a protected *one-to-many* association at the *Action* class, and then expose a composite collection that bridges, according to the object type, to the correct inner mapping. This can only work because these mappings are not ordered. Our example uses the *CompositeSet* implemented by the commons collections (APACHE FOUNDATION, 2008).

```
@Entity public class Action {
  @Transient private CompositeSet<ResourceAllocation> resources = null;
  @Transient public Set<ResourceAllocation> getResources() {
    return resources; // resources is initialized with all requirements
  }
  @OneToMany(mappedBy="action")
  protected Set<SupplyRequirement> getSupplyRequirements() //...
  @OneToMany(mappedBy="action")
  protected Set<ConsumableAllocation> getRequiredConsumable() //...
  @OneToMany(mappedBy="action")
  protected Set<EquipmentRequirement> getEquipmentRequirements() //...
  // the same to all scheduledEquipments, workRequirements, assignedEmployees
  protected void setSupplyRequirements(Set<SupplyRequirement> supplyRequirements) {
    addToResourceSet(this.supplyRequirements, supplyRequirements);
    this.supplyRequirements = supplyRequirements;
  } // repeat this set method for every type of resource...
```

The *Action* class exposes the composite collection *resources*, a transient property that is composed of all collections of the six leaf *ResourceAllocation* types. The *resources* variable is maintained by calling *addToResourceSet* for every collection initialized by the protected setters. An inner *mutator* class implements the behavior of transferring added objects to the right collections:

```
private class ResourceMutator<E> implements CompositeSet.SetMutator<E> {
  public boolean add(CompositeCollection<? extends E> composite,
    Collection<? extends E>[] collections, Object obj) {
    if (obj instanceof SupplyRequirement) {
      return getSupplyRequirements().add((SupplyRequirement) obj);
    } else if (obj instanceof ConsumableAllocation) {
      return getRequiredConsumable().add((ConsumableAllocation) obj);
    } // ... and so on...
```


4.5.3.2 Using *SqlAlchemy*

The *polymorphic_union* solves maps, of all resources, transparently into the property *resources* of the *Action*.

4.5.3.3 Using *ActiveRecord of Ruby*

The bidirectional mapping *action-resourceAllocation* is not supported by *RAR*. We mapped only the direction from *ResourceAllocation* to *Action*, and as a workaround, implemented a read-only access to the *resources* association end of *Action*:

```
class Action < ActiveRecord::Base
  self.table_name = "task"
  def resources
    ret = []
    ret += WorkRequirement.where(:id_task=>id)
    ret += SupplyRequirement.where(:id_task=>id)
    #repeat to every specialization (omitted...)
    ret.freeze #make it readonly
    return ret;
  end #...
```

This solution, however, is not aware of changes at the other side. A better solution would be to make *resources* observe changes at the specialization classes.

4.6 Remarks about implementing ENORM models

ENORM models graphically represent the mappings over the OO structure of class models. From the implementation examples we can create a correspondence between the concepts of ENORM, and its counterparts at each platform, presented by Table 4.1.

Table 4.1: Main correspondence of ENORM concepts.

ENORM	JPA	SA	RAR
<i>Persistent</i>	@Entity	mapper	Base
<i>DataSource</i>	@Table, @SecondaryTable, @JoinTable	Table, Join	create_table
<i>PK</i>	@Id	primary_key	primary_key
<i>Column</i>	@Column	Column	t.<column def>
<i>AssociationDef</i> , any association with persistent ends	@ManyToOne, @OneToMany, @ManyToMany, @OneToOne	relationship, secondary	belongs_to, has_many, has_and_belongs_to_many, has_one, join_table
Bidirectional assoc.	mappedBy	backref	inverse_of
<i>JoinColumn</i>	@JoinColumn(s)	ForeignKey, primaryjoin	foreign_key, t.references
<i>Embed</i>	@Embedded, @Embeddable	composite, __composite_values	composed_of
<i>Flat, Vertical, Horizontal</i>	@Inheritance, @MappedSuperclass, @DiscriminatorValue	inherits, polymorphic_identity	sti_name, polymorphic
<i>DiscriminatorColumn</i>	@DiscriminatorColumn	polymorphic_on	find_sti_class
<i>Overrides</i>	@AttributeOverride, @AssociationOverride	column_property	alias_attribute

Often this correspondence is not straightforward, due to CoC or the specific way each framework works. For example, a *Persistent* class may not have a *DataSource* specified, but by convention we have to declare a *Table* object at SA, or the

implementation will not work. A `@JoinTable` JPA annotation is both a data source specification and a *many-to-many* mapping specification, but the *secondary* attribute of SA is only a mapping, the table is specified apart.

Another issue is the non-trivial mapping of UML concepts to the platform specific concepts, summarized at Table 4.2. The first non-trivial concept is the property itself, that can be just an instance variable, or represent secondary structures, such as access operations (*getters* and *setters*).

At our implementation, we think the property as an abstraction, including the instance variable and other dependent elements. Using JAVA, we follow the Java Bean convention of private instance variable, and method based access control; for SA, the method itself makes reference to the instance variable, because the class does not declare instance variables. Moreover, persistent properties are automatically managed by SA and RAR, and are not declared within the classes.

Table 4.2: UML class mappings, non trivial cases.

UML	JPA	SA	RAR
property	“Java bean” property	Mapping and methods	Property, if not persistent
Abstract/redefined association/end	abstract access method	abstract access method	-
isOrdered x isUnique	List or Set	[...],Set , order_by	[...],Set, order
leaf element	final	<does not apply>	<does not apply>
lower card. 0 or 1	nullable=true/false	nullable=true/false	null=true/false
Interface	Interface	<does not apply>	<does not apply>
aggregation	-	-	-
composition	-	-	-

The *isOrdered* and *isUnique* combinations, however, affects the type of collection used to represent the association ends, if it is a set, a sequence (list) or a bag. For RAR and SA, persistent collections are lists when the order is specified at the mapping. Abstract associations, and redefined ends, are another issue, already discussed at the examples.

Aggregation and composition are high level abstractions that can be implemented with distinct strategies. We decided to not use these concepts as input to a particular implementation. The usual implementation consequences, such as cascading and eager fetch for composites, should be explicitly designed.

4.6.1 Guidelines for MDD

Models are abstractions of what is, or will be, implemented. Therefore, platform specific decisions should be taken by the transformation process that turns models into code. The example implementations at this chapter covered several situations where the developer had to create specific elements, ranging from variable names to classes and operations. These decisions are based upon PSI, hard coded at the transformation and/or contained at other model.

The Figure 3.25 proposed the scenario where PSM is the code, and that the transformation knows how to pick the correct decision based upon PSI, complementary to the ENORM model. This information is updated by the code itself, when the reverse operation is executed, in such a way that the following forward transformations would not override specific decisions taken by the developers. Ultimately, all PSI can be extracted by the code itself, accessible at a version controlled repository.

We identified several PSI elements, summarized by the table 4.3. The JPA, RAR, and SA columns describes at what platforms the PSI is relevant. The *Representation* column presents a list of possible information source.

Any PSI can be dealt exclusively by convention, what means that the transformation decides, and there will not have any flexibility. *Parametrized* means general parameters passed to the transformation that affects how the transformation will work, for all elements. *Marks* are fine grained parameters, specified by a platform specific complementary profile, applied at the ENORM models. *Models* are information represented by other models. Finally, *code* is the result of the aforementioned reverse engineering process.

Table 4.3: Platform specific information.

PSI	JPA	RAR	SA	Representation
PK class (name, properties, ...).	X			Code, marks.
Annotation position.	X			Code, parametrized, marks.
Method (operation implementation).	X	X	X	Code, models.
File names.	X	X	X	Parametrized, models.
Mapping names.			X	Parametrized, marks.
Default PK	X		X	Parametrized
Emulating inheritance		X		Parametrized

The implementation of the operations is behavior, and is better represented by other UML models, such as activity and sequence diagrams, and by code. UML lacks an action language (MELLOR et al., 2004), therefore it is difficult to use diagrams to represent behavior without some extension.

File names affects secondary files derived from the classes. A convention such as one file for each class may partially solve this problem, but there are specific situations, such as the migrations of RAR, or the PK classes of JPA. It is difficult to track file name changes when reverse engineering: do they represent a new concept, or an old concept at a new artifact? UML has the component model, that can be used to specify the artifacts that implement the classes.

The PK classes of JPA can have operations and other details, as any other class. They can be public, or classes visible only at the package level. To achieve a finer control of the specification, *marks* can be used. JPA also allows the placement of annotations at methods or variables.

On the other hand, SA names each mapping and table, associating with variables. There are distinct ways of organizing these mappings, within the classes, or at a separated module. The way a transformation name and organize the mappings could be parametrized, because they do not usually depends on the concept. In a similar way, the emulation of inheritance for RAR, if supported by the transformation, is a candidate for being parametrized.

Table 4.4: Design questions and response scope.

<i>Domain</i>	<i>What to observe/question during project design</i>	<i>PIM</i>	<i>PSM</i>
Model	If the framework has a data-model that controls persistence, how does it work with object-oriented domain models?		X
	Are operations assigned to classes that extend generated classes or domain classes do not support operations?		X
Classes	Are inheritance to ORM framework classes mandatory?		X
	Is there some dependency to framework classes on domain classes? Can it be decoupled?		X
	Which table(s) are mapped to that class?	X	
	Is the identity mapping defined? one or more attributes?	X	
	How the identity will be assigned? Will generation parameters for the identity be necessary? (such as table of ids, sequences, auto-columns...)	X	
Embedded	Is it necessary to distinguish a class as embeddable?		X
	Is it possible to define preferred mapping for attributes?	X	
	Is it used as identity for persistent classes?	X	
	Should the ID class follow specific rules required by the ORM framework?		X
	Relationships <i>from</i> embeddable values to other domain classes may be unsupported.		X
Attribute	Does it match a database column type or should it be an embeddable value?	X	
	If it is part of a composite Identity, does it represent an embeddable value class containing PK fields or each attribute of the identity is an individual attribute?		X
	Type parameters, such as length and precision were defined? Is the cardinality matching NULL/NOT NULL constraints?	X	
Association	Should a collection type be defined?	X	X
	How to deal with element ordering?	X	
	Can/Should the fetch configuration be specified?	X	
	Will the relationship attribute be loaded by a proxy?		X
	The collection must have a reverse attribute or collection in the opposite class that maintains a bidirectional relationship. Is it defined and documented?	X	
	Is the relation Attribute-Collection-FK clear and well documented?	X	
	Is a join table clearly defined, with FKs to the tables mapped to the related classes?	X	
	In what cases a join table must be explicitly implemented as an association class?	X	
	Maps and element collections.	X	
Inheritance	What strategy will be employed?	X	
	Is a discriminator column necessary?	X	
	Does the persistence framework support classes in an inheritance hierarchy with distinct Identities (PKs)?	X	

The Default PK is a PSI that specifies how a class, without specified PK, will be identified: what will be the name of the column, the type, and other details. RAR already defines the default PK.

Taking the questions raised at the Table 2.3, we can now check at what abstraction level each question can be better answered, using the ENORM model at Table 4.4. The questions that are described by the ENORM model are marked at the PIM column, and the questions that are platform dependent are at the PSM column.

For instance, should a class extend the framework, or there is a loose coupling mechanism? This is a PSM detail, and therefore the ENORM model should not show if the class extends *Base::ActiveRecord*. If operations should be placed at another class, this also should not be presented at the ENORM model. PSM information, such as *@Embeddable* annotations used by JPA, are also omitted, but the embedment can be represented at the association.

Regarding the collection type, it is partially specified by the model (by the *isOrdered/isUnique* combination), but can be further detailed at PSM level, to answer what implementation of *Set* should be used, as an example.

5 EMPIRICAL EVALUATION

The use of a single model can be criticized because it would violate the principle of separation of concerns, and thus decrease the quality of the resulting models (MITCHELL, 1990). However, representing the concepts of software and persistence in the same model, does not automatically mean breaking the orthogonal representation of concepts. ENORM models still separates tables and classes, and can be easily, automatically in fact, transformed to pure class or relational view models. In a similar way, aspect oriented languages, such as *AspectJ*, allows the writing of aspects and classes in the same files, representing the relationship between those concepts, and promoting a better SoC.

Moreover, one can argue that using two separated models with so many similar concepts violates the DRY principle (HUNT and THOMAS, 1999). In that scenario, using a single model would save time and avoid repeated work. However, the question remains, do the notation proposed by ENORM impair system design? That is a good question for a controlled experiment based answer.

Controlled experiments allow the evaluation of specific steps of the software process, by controlling several input variables. They require less resources than case studies in the industry, and may evaluate the notation independently from analysis and development (WOHLIN et al., 2012). This chapter presents the experiment design, execution, and the analysis of the results in order to assess that ENORM single notation does not decrease the quality of models, and may perhaps increase its quality, independently of MDD or code generation facilities.

5.1 Experimental Related Work

From what we could find out, there are few experiments comparing persistence and software artifact modeling activities, and none comparing two models with one model. In retrospect, experimental comparison between ER (CHEN, 1976) and Extended Entity-Relationship (EER) models (TEOREY, YANG and FRY, 1986), presented empirical evidence that EER models performed better than ER models, despite the inclusion of concepts such as generalization, specialization, and categories (BATRA, HOFFLER and BOSTROM, 1990). As for controlled experiments with teams, a recent systematic review did not find any paper evaluating tools in the context of a team (KO, LATOZA and BURNETT, 2013).

EER models with mandatory properties and subtype relationships, had also a superior understandability over plain ER models with optional properties (GEMINO and WAND, 2005), despite its increased grammar complexity. Another study indicated

that OO was preferred by more experienced designers, for the task of conceptual modeling databases (JAIN, GORE and SINGH, 2009).

According to our research, recent experimental studies on Software Engineering and models focus on the comprehension of models, mostly regarding the use of profiles, such as (CRUZ-LEMUS et al., 2011; JÚNIOR, PENTEADO and DE CAMARGO, 2010); or the interference of using models in some process (ALBAYRAK, 2009; STÅLHANE and SINDRE, 2008). Even when the focus includes maintenance, the participants are not asked to model, but to choose between models answering a multiple-choice questionnaire (LUCIA et al., 2010). Our experiment differ from this trend of surveying the participants about models, by requiring the participants to actually draw models, as in other experimental comparisons such as (BATRA, HOFFLER and BOSTROM, 1990; JAIN, GORE and SINGH, 2009).

5.2 Planning and Design

The Goal-Question-Metric (GQM) provides a template to the experiment definition by setting measurable goals (WOHLIN et al., 2012). The general goal is an answer to the question:

General Goal: “Do the ENORM notation affects in a negative way the modeling activities in comparison with the separated use of relational and class models?”

The object of evaluation is the modeling activity, regarding the quality of models and the time spent to complete a task. However, this general goal is too wide and ambitious to be achieved. The following constraints must be taken into account:

1. Rel. and UML are notations with a large user base. Most IT professionals have some level of experience with, at least, one of the modeling methods. ENORM is a new notation, and even with training, it could not (currently) compare to the practical experience that professionals have with UML and relational models. Not without really extensive training.
2. It is easy to find relational and class models real world use cases, but there is not yet such use cases with the new notation.
3. Without experimental validation, it is hard to convince any organization to invest its time and money on new tools and experiments.
4. The modeling activity, taken independently of development, is subjective to be evaluated. Given a problem to someone model would require someone to judge the quality for this model: does it accomplish the task? If not, how many goals were achieved? regarding the small universe of specialists of the new notation, all of them are somewhat connected to the research itself.
5. The modeling activity is not equals to model comprehension. Presenting models and multiple choices questionnaire is very different from the actual work of modeling, by hand or using a software.

The goal was constrained to fit these restrictions as follows:

To analyze the modeling activity for the purpose of comparing the use of separated relational and class models with ENORM, with respect to the quality and time, from the

viewpoint of student/novice system designers, in the context of solutions using ORM patterns or frameworks.

The following questions derive from the above goal: what method is more efficient, and what method deliver better quality? The main candidate metrics are percentage of success, errors, and time to finish each task.

Moreover, modeling can be evaluated individually or on team work. Therefore we presupposed two type of experiments, one applied to evaluate individuals and another to simulate social interaction and cooperation using distinct models, hereafter referred as *individual* and *group* experiments. The remaining of this section details the subjects, hypothesis, variables, design, and instrumentation of the experiments.

5.2.1 Subjects

The controlled experiments were executed at the *Universidade Federal do Rio Grande do Sul*. The subjects were selected among the senior undergraduate students of computer science, and graduate students attending software engineering or advanced database classes. Among the undergraduate students, only those attending the advanced software engineering class, and that were approved in the basic database and software engineering courses were selected to participate, assuring a minimal understanding of class models, relational models, and the IMP. The graduate students have a more diverse academic and professional background.

With regard to the ethics of the experiment, it is important to note that the experiments were part of a series of extra-laboratory exercises conducted within the software engineering and databases courses and students were not evaluated (graded) on their performances. Moreover, these laboratory exercises were not part of the courses, the students participated at its own free will. An informed consent was presented and signed by each participant. None of the participants had any connection to our research until the experimental sessions.

Table 5.1 shows the distribution of the participants among the experiments. The *group* experiment was executed only one time with 30 undergraduates of the software engineering class. All other subjects participated in the *individual* experiments (*I1* and *I2*). In *I1*, the subjects were evenly distributed by the type of course (undergraduates, graduates), with the same number of graduates and undergraduates at each experimental group.

Table 5.1: Subjects and experiments.

Id	Experiment	Characteristics	Year	Subject Type	Subjects
P	Pilot	Using paper, limited time, crossover	2011	Graduates	16
I1	Individual 1	Using tool, limited time, crossover	2012	Both	69
I2	Individual 2	Using tool, unlimited time	2013	Graduates	44
G	Group	Using tool, groups, auto-evaluation	2013	Undergraduates	30

All participants were trained with ORM concepts, class models, relational modeling, and the ENORM notation. The training included a small task executed in the same way

as the individual experimental tasks. The material used in the training was available to consult during the experiment². Furthermore, the participants could use any consulting material available in the internet.

All participants had the same training, before the beginning of the experiment, despite being selected to the control group of the non cross-over repetition. Despite the training focus in the new notation, the notations used by the control group were also revisited in the training, with an ORM focus. We believe that knowledge about ENORM could help the participants in the making of separated models by stressing the difficulties inherent to ORM and the IMP.

In the experiment *G*, we explored the fact that the participants were all colleagues, leaving to them the decision to select design partners. We also let them choose who had a better database background to play the part of DAs.

5.2.2 Task Design

The first decision about task design is to decide its origin. The ideal setting is the controlled experiment using a real task, in a real work place, found in a real system, but with a controlled environment. This ideal hardly can be achieved due to the following:

- An experiment with a real problem, in a real setting would take a long time to execute (days to weeks). Therefore it turns out to be difficult to control.
- A real problem adapted to a controlled setting is often reduced, for instance by using the rule of seven (MILLER, 1956). Here enter the risk of the experiment designer choosing the most convenient “seven” pieces, what may configure a threat to validity.
- A task designed solely to the experiment may also favor a treatment and presents the additional threat of being artificial.

5.2.2.1 Individual Experiments

Without a real problem with a real base solution using ENORM, we opted to employ tasks designed solely to the experiment. In order to alleviate the interaction of setting and treatment threat, we decided to employ Analysis Patterns (FOWLER, 1996) as the base theme for our *individual* experiments.

Analysis patterns are themselves an established way to divide complex problems in relatively small solutions described by OO models. Therefore we can decrease the artificiality, and avoid the rule of seven by using patterns for tasks. However, analysis patterns are not specifically oriented to designing data persistence nor ORM, such details had to be added to the cases.

We decided to evaluate the modeling activity, thus eliminating the perhaps easier to analyze way of presenting models and asking objective questions to the participants. But we had to decide to ask participants to create models, change models, or both activities. Therefore we decided that the tasks that better reflect the designing are those that *evolve* already existing models.

However, the evaluation of models is tricky, can easily become subjective, and works better with a team of independent evaluators. We could not afford to recruit such

² Training material, videos in Portuguese: <http://www.inf.ufrgs.br/~atorres/tutorial/>

team of evaluators, and train a crew would be another threat of validity. A solution was to make the tasks as most objective as possible.

In that sense, the use of Analysis Patterns offered an answer in itself, because some patterns are refinements or generalizations of others. Taking the *Account* pattern for instance, we could evolve the basic model to include the *Multi-legged Transaction* and the *Summary* patterns (FOWLER, 1996). The task would be, therefore, to present a model of *Account*, and an objective description of what changes in classes and database must be taken, to evolve the model to include the more refined patterns. The critical point was to define a step list that is objective enough, to reach only one or very few possible outputs.

Having a starting model, and an objective step list guiding to the resulting model, would free our analysis of subjective measurements, by the cost of reducing the creative liberty of the experiment. The advantage is that the models could be analyzed by a software, with clear objectives, implemented in a programming language open for audition, and therefore better controlling the reliability of measures threat.

5.2.2.2 Group Experiment

In the group experiment the focus was in model integration. In the first stage, each subgroup designs a solution for an independent problem. In the second stage, groups have to work together a solution that uses the same database, with minimal replication and impact over the DA initial design.

We created three distinct applications with a few common requirements, one for the DA initial design, and two for the software development designs. These tasks were specifically created to the experiment, and the idea was to let the groups use their creativity. To evaluate the tasks, we decided to make each group auto-evaluate their resulting models, in order to have a distinct viewpoint of the more objective individual experiment. This auto-evaluation included objective questions requiring each group to explain how their solution would attend some practical cases. This would be complementary to the evaluation employed in the individual experiments.

5.2.3 Hypothesis Formulation, Factors and Variables

The following basic hypothesis were formulated for the experiments:

Null hypothesis (H₀): there is no difference in the rate of success, performing changes in models using separate relational and class models, or a single ENORM model. (misses(ENORM) = misses(Rel., UML)).

Alternative hypothesis 1 (H₁): The rate of success, performing changes in models using separate relational and class models, is INFERIOR than single ENORM model. (misses(ENORM) < misses(Rel., UML)).

Alternative hypothesis 2 (H₂): The rate of success, performing changes in models using separate relational and class models, is SUPERIOR than single ENORM model. (misses(ENORM) > misses(Rel., UML)).

The basic design for the experiments is the *one factor with two treatments*. The factor of interest is the set of design notations used named *method*, that identifies the treatment. The *method* is a two level independent variable assuming *A* for the use of one class and relational models, and *B* for the use of one ENORM.

The main dependent variable is the number of unreached goals, referred here for brevity as *misses*. Nevertheless, this number is actually a measure of success (KO, LATOZA and BURNETT, 2013), evaluated by previously defined goals covering certain features. If every model in a task achieves all goals, the task has a score of zero *misses*.

The following measures were taken to control the experimental environment:

- Configuration: Hardware and operational system is a concern that can affect the outcome of any experiment. Fortunately, the laboratory available for the experiment had the exact same configuration for all participants, with same hardware, operational systems, browsers, memory and so on.

- Sessions: The entire experiment was prepared to be executed locally and remotely, although only the local execution was executed. The training was designed as an individual task with fixed time frame equal to all participants on each experiment (*I1*, *I2*, and *G*). Inside this time frame (1:30 hours), the participant could study the materials, and no further explanations were given by the conductors. The experiment itself was managed by a tool, allowing each participant to start in different times. In that sense, participating on distinct sessions had no influence in the outcome.

- Experiment: each experiment was analyzed independently for the hypothesis due to the distinct mechanics.

- Tasks: for the individual experiment *I2*, the task is a four level independent variable. At *I1*, each task was analyzed independently, because they were performed at different days, and had a greater variation of subjects. In the group experiment, every group executes all tasks, and each group is a subject, therefore task is not a variable on *G*.

- Ability and experience: the subject ability in relational and class models may vary, but it is difficult to establish at this stage what abilities are relevant to solve ORM problems. The crossover of *I1* controls this factor by using the *within-subjects* technique (ROSENTHAL and ROSNOW, 2007), although the learning effect had to be later taken in calculation (KO, LATOZA and BURNETT, 2013). For the other experiments, no assignment technique was adopted. Nevertheless, the ability of each participant was surveyed, after each experiment, in terms of experience on various areas from general system development to ORM frameworks usage.

All experiments assume results as statistically relevant at $\alpha = 0.05$, meaning that the derived *p-value* must be less than 0.05 to conclude that the null hypothesis can be rejected. In the *I1* experiment, the *sequence* of the crossover is an independent variable that can assume two possible values. For the *I2* experiment, the *time* is measured as a dependent variable. The hypothesis testing of *time* is as follows:

Null hypothesis H_{t0} : there is no difference in the time necessary to perform changes, in models using separate relational and class models, or a single ENORM model. (Time(ENORM) = Time(Rel., UML)).

Alternative hypothesis 1 (H_{t1}): Time(ENORM) < Time(Rel., UML)).

Alternative hypothesis 2 (H_{t2}): Time(ENORM) > Time(Rel., UML)).

In the group experiment, the hypothesis is modified because one of the members of the group uses relational models to administrate the database. Therefore the comparison is between integrating relational and class models of developers (*A*) or ENORM models of developers (*B*) with relational models of the DA.

The main statistical analysis test for the individual experiments is the Analysis Of Variance (ANOVA), despite other analysis employed to check the data, such as the linear correlation between time and missed goals. For the group experiment, the *Mann-Whitney test* was employed due to the small number of subjects.

5.2.4 Tasks and Feature Coverage of the Individual Experiment

First we selected the Analysis Patterns which would form the starting point for each experiment. Each unit begins with a diagram representing these patterns, and a set of instructions to the application of one or more related pattern improvements over the initial model. The participants have a fixed time to perform the modifications before moving to the next case. Patterns, pattern improvements, and instructions were extracted from the literature (FOWLER, 1996), although persistence details were added. These persistence details were created to cover features of ORM, from mapping artifacts with different names to inheritance and object embedding. It is important to emphasize that although ENORM relates to *persistence design patterns*, the experiment assess the use of ORM concepts in the design of *analysis patterns*.

Table 5.2: Analysis patterns and tasks.

Task	Analysis Domain	Starting Patterns	Patterns for improvements
1	Accountability	Address book	Party
2	Accountability	Accountability	Knowledge level, party type
3	Accounting	Account, transaction, quantity	Multilegged transaction, summary account
4	Planning	Resource allocation	Resource allocation with time period

Figure 5.1: Task 1 - Address Book UML and ER models.

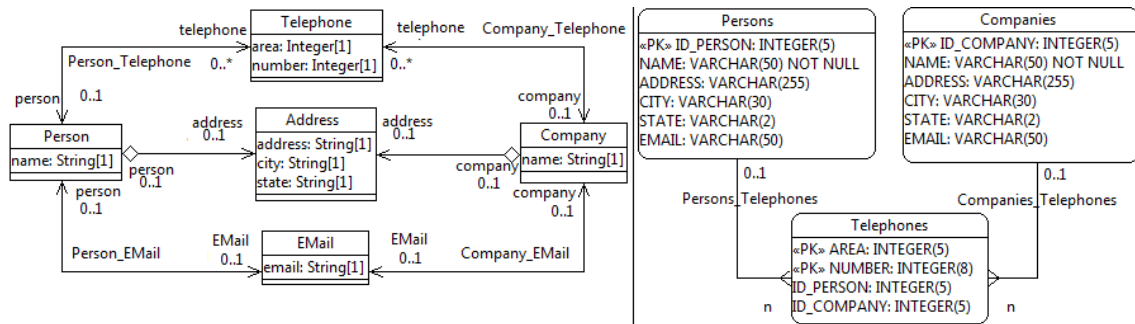
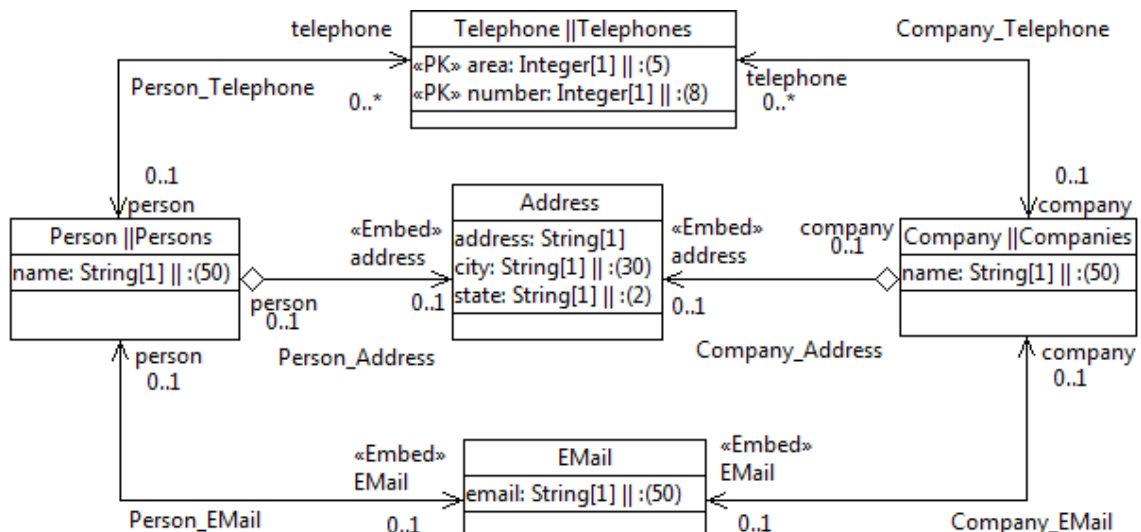


Table 5.2 shows, for each task, what is the analysis domain, the starting patterns of the initial model, and the patterns used to compose the list of steps presented to the participants in order to execute the task. Figures 5.1 and 5.2 shows models presented to

the participants performing the first task, named *Address Book*, of the control (*A*) and experimental (*B*) groups.

Figure 5.2: Task 1 - Address Book ENORM model.



The class models are similar to the models provided in the original material of Fowler, but complemented by relational models that represent one possible database that could be mapped using ORM. Looking at both models it is possible to identify the *persistence patterns* that were used, such as class embedding of *Address* and *Email* along *Person* and *Companies* tables. The following list of steps was asked to each participant with treatment A, in order to apply the *Party analysis pattern*:

- Person and Company will now specialize the new abstract class Party.
- Person and Company will now be persisted at the same table named PARTIES. The PK of PARTIES should have the same type and length of Persons's PK and should be named "ID_PARTY".
- Move the name attribute (common on Person and Company) to the Party class.
- PARTIES table should have a not null column named PARTY_TYPE with length of one, discriminating party types between Person (PARTY_TYPE="P") and Company (PARTY_TYPE="C"). Specify that this column can only assume one of these two values.
- Relationships from Person and Company to classes Address, Telephone and Email will be unified to relationships between Party and Address, Telephone and Email.
- Telephone should relate to exactly one Party, but a party may have many telephones.

The participant performing the experimental set (*B*) will be presented with only one model using the ENORM notation that represents exactly the same classes and database, but with explicit mappings, as shown by Figure 5.2. The list of steps presented to the participant is equivalent to the control group, but referring to the notation language of the single model:

- Person and Company will now specialize the new abstract class Party.
- Use the flat inheritance mapping between Party and its subclasses Person and Company. Specify the discriminator named PARTY_TYPE assuming values P for person or C for company.

- c) Move the name attribute (common on Person and Company) to the Party class.
- d) Relationships from Person and Company to classes Address, Telephone and Email will be unified to relationships between Party and Address, Telephone and Email.
- e) Telephone should relate to exactly one Party, but a party may have many telephones.

Table 5.3: Feature coverage and task goals.

Feature under evaluation	Task	Goals
Abstract association	3	Account to Entry is abstract
Abstract concept	1	Party should be an abstract class
	2	Organization not abstract
	3	Account should be abstract
Aggregation	1	Party aggregates Address
Association table	2	Correct many-to-many acctype_comissioners
	2	Correct many-to-many acctype_responsibles
Class persistence	2	Party Type should be persistent
	2	Party Type with correct table name
	4	User is persistent
Class Removal	2	Rule concept was not removed
Create class	3	Include Summary Account
	3	Include Detail Account
	4	Create User class
Create embedded class	4	Create Time Period
	4	start and end on Period
	4	Incorrect cardinality for start or end
	4	TemporalAllocation should relate to one period
	4	Allocation is unidirectional to period
	4	TimePeriod is Embeded on TemporalAllocation
Dependent entity	3	DetailAccount part of Entry's PK
Discriminators on inheritance	1	Discriminator name should be defined
	1	Discriminator values specified
	3	discriminator column type
	3	type can assume S or D
Distinction between table and class	1	PARTIES table to persist Party
Flat inheritance	1	Party should be persistent
	1	Person/Company should not have tables
	2	Remove specializations of Organization
Join column naming	3	JoinColumn named acct_number for Entry
	3	Account with joincolumn named summary
Lower cardinality	1	Party associated to one Address ? Many-to-one?
	1	Party associated to ONE EMail
Many-to-Many	2	A party can have many types
	2	Party type may have many parties
	2	AccountabilityType can ref. many Party Types
	2	Party type may reference many Accountability Types
Many-to-one	3	Zero-Many Entries should relate to one Detail Account
	4	Resource Allocation to one User
Many-to-one/(many)	2	Every party must ref. a Party Type
	2	AccountabilityType must reference a Party Type

Feature under evaluation	Task	Goals
Many-to-zero/(many)	2	Party type may have zero parties
	2	Party type may have zero accountabilities
Move association to super-class	1	Person/Company should not be associated to Telephone
	1	Party should be associated to Telephone
	1	Telephone should relate to at min. 1 Party
	1	Name column on Party
Move properties to super-class	1	Definition of Party Type
Multiple sub-typing	2	Party to Address is unidirectional
Navigability	1	PartyType is not navigable to party
	2	Account to Entry is unidirectional
	3	Address should be embedded in Party
Persistence of embeddment	1	EEmail should be embedded in Party
	1	Description property on Party Type
Properties	2	Description column with String stype
	2	Description column with correct length
	2	Schedule attribute removed
Property removal	4	Summary Account to Entries read only
Read-only feature	3	Summary Account to entries not persistent
Transient feature	3	Many Telephones for Party ? One-to-many?
Upper cardinality	1	Party should not be associated to Person
Use extension x association	1	Party should not be associated to Company
	1	Party associated to Address
Use of embeddment	1	Party associated to EMail
	1	Person should extends Party
Use of extension	1	Company should extends Party
	3	Summary Account specializes Account
	3	Detail Account specializes Account
Use of PK	1	Expected PK for Party/PARTIES
	4	Login is the PK of User
Vertical inheritance	3	Summary Account vertical inheritance
	3	Detail Account vertical inheritance
	3	Summary should be persistent
	3	Detail should be persistent
Zero/one-to-many	3	Zero-one Summary to many Account
	3	Summary Account to many entries

The expected correct answer for *Address Book* using two models or one model is the same in terms of classes and database constructs. The models are presented in the tool that managed the experiment, and the participant follows the steps changing the models. The models are evaluated by a matrix of goals based upon the steps, as listed by Table 5.3.

Each goal missed counts as one. For the control group that have two models, if the goal is missed in one of the models it counts as one, but if the same goal is missed in the two models, it still counts only as one miss (same missed goal). Some goals can only be missed in one model, such as direction of association, because they have no effect in the relational model. This approach makes the maximum number of missed goals the same, no matter the method in use.

5.2.5 Tasks of the Group Experiment

Each group had a total of 5 subjects and was assigned with the same tasks using either treatment *A* or *B*. The experiment had three stages:

1. In the first stage, each group DA worked in the persistence model (and only the persistence model) for a *room's booking* application. The remaining participants were subdivided in two subgroups of two subjects, randomly assigned to the *meeting scheduling* application, and the *appointment book* application. For the group experiment, the random assignment was performed by picking *method* and *application* from pieces of paper in a bag. Each subgroup were asked to model the applications, both classes and persistence, using the treatment assigned to his group, but the DAs should use only relational models and care only about the database. This stage had a time limit of 40 minutes.
2. In the second stage the entire group sits around one computer with all models produced to each of the three applications and the task of integrate *meeting scheduling* and *appointment book* applications using the *room's booking* data model as a legacy database. All groups were asked to find a solution with minimal tables, sharing common information, and with minimal changes to the database model produced by the DA. This stage had a time limit of 50 minutes.
3. In the third stage, executed in another session, each group receives a questionnaire containing hypothetical data, and a set of thirteen yes/no questions. The questionnaire was the same, regardless the applied treatment. Each question asks the group to check if some functionally would work under certain circumstances according their solution, and asks the group to show how the data would be fetched and updated in the database. The group must discuss and choose a consensus answer. For this last task, the groups had 1:30 hours.

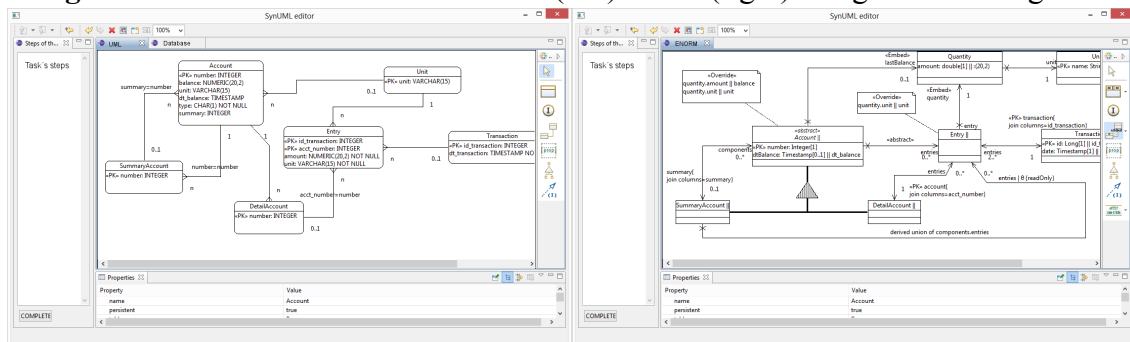
The only material subject to analysis in this case was the questions resulting from the auto-evaluation of each group, although the participants did not knew that their explanations would not count. Moreover, a minimal survey with four questions was presented in order to evaluate the difficulty level of the tasks.

5.2.6 Experimental Setting

We had to adapt our tasks and methodology to time constraints along each experiment. The experimental design for *II* employed crossover, but implied a time limit for the tasks. That was necessary due to the fact that using crossover required twice the time, because each participant had to perform the tasks using experimental and control methods.

The pilot experiment was performed at 2011, using paper and ink to create models. It served to show the limitations of time and paper to our initial design. We were already developing a design tool for ENORM, and decided to improve this tool, to support modeling with simple class and relational models (the last using the crow's foot notation). Therefore, the following experiments were executed using the modeling tool, that also managed time, the sequence of experiments, and the evenly assignment of subjects among experimental groups.

Figure 5.3: Screen shots of treatment *A* (left) and *B* (right) using the modeling tool.



The tool, based on Eclipse (ECLIPSE FOUNDATION, 2012a) and a web *servlet*, provided a balanced setting for a modeling experiment: similar interface, similar resources, same environment. Figure 5.3 shows screen shots of the tool presenting task 3 (*Account*) for treatments *A* and *B*. The feature set of the tool was, as much as possible, the same for all notations, with minimal changes in the toolbar that included specific elements for DB and ENORM. Also, eclipse allows to visualize the models side by side, one above the other, or one at a time, moving the various frames of the workspace. The participants had the opportunity to test the tool before the experiment itself, during the training class, with a simpler task.

The first set of experiments (*II*) was performed at 2012 using the tool, in two distinct sessions. Each participant executed tasks using one treatment, and then repeated the tasks with the other treatment. The first treatment to be used was selected by the order of *login* in the tool, that distributed evenly the methods applied to the students. The *login* order was dependent on the participant arrival order, which can be considered random in the scope of this experiment: the first receiving treatment *A*, followed by *B*; the second receiving treatment *B* followed by *A*; the third receiving *A*, followed by *B*; and so on.

Table 5.4: Tasks and time constraints for experiment *II*.

Task	Session	Name	Time Limit	Participants	Sequence
1	1	Address book	10 min.	69	1A, 2A, 1B, 2B or
2	1	Accountability	20 min.		1B, 2B, 1A, 2A
3	2	Account	20 min.	35	3A, 4A, 3B, 4B or
4	2	Resource allocation	23 min.		3B, 4B, 3A, 4A

Table 5.4 shows the tasks, time limits, the total number of participants for each session, and the possible sequences of experiments. At each session, each participant performed the tasks using one method and then repeated the tasks using the opposite method. About half participants of *II* could not participate on more than one session. Some did not show, but many could not participate in two sessions.

The advantage of the crossover technique is to minimize the influence of experience background, because every participant will have to solve each task with each possible

treatment. On the other hand, the analysis must take in account the learning effect. Another issue was the time limit determined for each task, that affected the measurement of the time variable. The majority of the participants used the maximum available time, rendering the time variable useless on *I1*. All participants were aware of the remaining time of each task, by a small timer presented by the tool.

The second group of experiments (*I2*) was performed at 2013, and employed the same tasks and tools, but without crossover nor time limits. Each subject experienced only one treatment, at a single session, using the same random treatment distribution. If the time reserved for the experiment was too short, the participant could choose to stop before completing the last task.

Table 5.5: Codes used in the group experiment for random assignment.

Code	Application	G1A	G2A	G3A	G1B	G2B	G3B
DA	Room's booking	100A	110A	120A	100B	110B	120B
S1	Meeting scheduling	101A	111A	121A	101B	111B	121B
S2	Appointment book	102A	112A	122A	102B	112B	122B

The group experiment was also performed at 2013 using a slightly distinct version of the same tool, capable of displaying various models at the same time. DAs were volunteers among the subjects, and the other participants decided with who they would work in pairs. Hence, each pair picked a random piece of paper in a bag telling what method (*A*, *B*) to use, the application they should design, and the number of the other pair they had to integrate (numbers ending with one and two at Table 5.5). After that, each DA picked a piece of paper from a bag determining the pairs they would work with in the integration stage, half to each method (numbers ending with **zero**).

Despite the use of a tool, the purpose of these experiments was not to evaluate a modeling tool. This was clearly stated to the participants, and the tool features were essentially the same for users of class, relational, or ENORM models.

5.3 Results and Analysis

Each of the three experiments (*I1*, *I2*, and *G*) have distinct characteristics that had to be taken in account for the data analysis and the interpretation of experimental results. In the next sections we explain the data analysis process and the results obtained of each controlled experiments. All analysis were performed with Statistical Analysis System (SAS), version 9.2, and supervision of statistics from the center for statistical advice (*Núcleo de Assessoria estatística – NAE*³) of our university.

5.3.1 Individual Experiment with Crossover (I1)

The first experiment employed a balanced crossover design, where each treatment was applied to each subject, and the starting order was balanced so that the same number of subjects started with treatment *A* and *B*. The analysis of variance (ANOVA) was employed to compare treatments *A* (UML+Rel.) and *B* (ENORM) making it possible to to verify the residual effect in the sequence of activities. In other words,

³ <http://www.mat.ufrgs.br/~nae/>

given two groups of participants *seq1* and *seq2*, with *seq1* experiencing *A* followed by *B*, and *seq2* experiencing *B* followed by *A*, test if there is a significant difference between *seq1* and *seq2* executing the same methods in distinct sequence.

Table 5.6 shows the analysis results considering *method* (treatment), *sequence* (the order starting by *A* or *B*) and *period* (two period design of crossover), all effects with the *Numerator Degrees of Freedom* equals to 1. For each experimental task, DF is the *denominator Degrees of Freedom*, f-value (F) is the Fisher statistic and P is the p-value for the comparison of the *misses* variable.

Table 5.6: Results for the ANOVA of misses considering the sequence.

Effect	Addr. Book			Accountability			Account			Resource Alloc.		
	DF	F	P	DF	F	P	DF	F	P	DF	F	P
Method	62	35.01	<.001	54	55.35	<.001	35	8.77	<0.01	33	101.79	<.001
Seq.	67	0.34	0.56	67	0.66	0.42	35	1.14	0.29	35	1.36	0.25
Period	62	4.17	0.045	54	0.00	0.97	35	6.68	0.01	33	2.96	0.1

For every task, according to the data of Table 5.6, there is a significant difference between methods *A* and *B*, with method *B* presenting a lower mean of misses (*Address Book*, $F=35.01$, $P<0.001$; *Accountability*, $F=55.35$; $P<0.001$; *Account*, $F=8.77$; $P<0.01$; *Resource allocation*, $F=101.79$; $P<0.001$). The sample evidence does not confirm the presence of residual effect, given the absence of statistical significance for the effect of *sequence* (*Address Book*, $F=0.34$; $P=0.56$; *Accountability*, $F=0.66$; $P=0.42$; *Account*, $F=1.14$; $P=0.29$; *Resource allocation*, $F=1.36$; $P=0.25$).

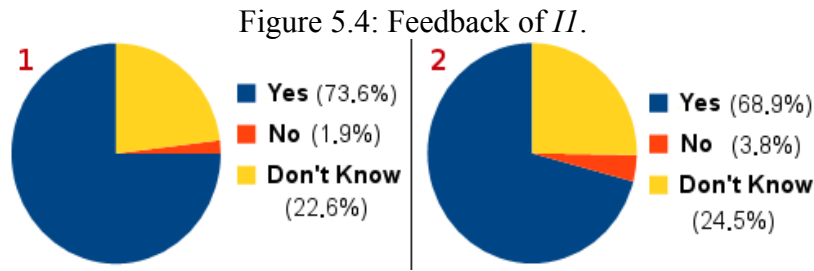
The analysis of the results of *II* rejected the null hypothesis (H_{s0}), and favored the alternative hypothesis (H_{s1}) of ENORM in all four tasks. Despite the crossover setting, the test results indicates that there is no evidence of learning from one activity to the other. Since the design was a crossover, with all participants experiencing both treatments for all tasks, we did not checked for the uniform distribution of experience levels among experimental groups. The time could not be analyzed due to the high percentage of participants using the maximum available time.

5.3.1.1 Analysis of the Feedback

After the experiment, each participant was asked to answer two questions, about his experience using the single model and the separated models. The questions were not to compare the methods, but instead capture the general feeling about the single model method, after experimenting both approaches:

1. Putting aside the notation, do you think that this new approach improves the efficiency and efficacy of software refactoring? (Yes/No/Don't Know)
2. Do you think that the proposed notation facilitates changes and refactoring of information systems? (Yes/No/Don't Know)

For the first question, 73.6 % of the participants think that a single model approach improves efficiency and efficacy. In the second question, 68.9% of the participants think that ENORM facilitates the tasks of changing models. Figure 5.4 shows the results for the questionnaire.



5.3.2 Individual Experiment with Time Measurements (I2)

The second experiment applied to the students was a design with random assignment of participants between treatments *A* (UML+Rel.) and *B* (ENORM), without time limits. In this experiment we measured non-accomplished goals (*misses*) and the amount of *time* used by each participant performing each *task* (inter-subject variable), using one *method* (between-subject variable).

Before analyzing the results, we tested the correlation between the variables *time* and *misses* in order to establish if *time* should be a covariate of *misses*. The linear correlation test of *Pearson* was applied to all data (N=157 measurements of subjects in each task), yielding a result of $p=0.23$, indicating no significant correlation between *time* and *misses*.

Based in this test, we performed independent ANOVA tests for repeated measured of *misses* and *time*. Because the data presented variance heterogeneity, it was necessary to first transform the data, with the method of minimal pondered squares, with its variance considered in the composition.

5.3.2.1 Results Regarding the Variable “Misses”

The factors *Group* and *Task* were first tested for interaction of the variable *misses*, yielding a significant interaction effect of *Group*Task*, with $p=0.0292$. Following this result, we applied the *Tukey-Kramer* multiple comparisons test to compare the means of the interaction *Group*Task*, and determine what tasks have a significant variance in the mean number of *misses* between groups.

Table 5.7: Least square means of misses at *I2*, with adjustment *Tukey-Kramer*.

Task	Group	N	Least square mean	Pr > t
Addr. Book	A	21	7.8095238	0.9992
	B	23	7.0869565	
Accountability	A	21	9.6666667	0.0109
	B	23	5.6521739	
Account	A	18	14.2563014	0.4986
	B	19	11.3597870	
Resource	A	15	8.9046605	<.0001
	B	17	3.7468807	

Table 5.7 presents the results for *Tukey-Kramer* test. Each task is detailed in terms of groups, number of subjects in this group (N), least square mean, and the *p-value*

regarding the significance in the mean number of *misses* comparing the groups per task. In tasks *Accountability* and *Resource*, we can conclude that the mean number of *misses* is significantly distinct on groups *A* and *B*, with $p=0.01$ and $p<0.001$ respectively. The participants using treatment *B* had a lesser mean number of *misses* in every task than the participants using treatment *A*. However, in tasks *Address Book* and *Account*, the mean number of *misses* is not significantly different.

5.3.2.2 Results Regarding the Variable “Time”

The factors *Group* and *Task* were again tested for interaction, but for the dependent variable *time*. The ANOVA results did not showed a significant interaction effect of *Group*Task* in relation to the response variable *time*, with $p=0.5915$. Therefore, we could not make any affirmation about the significance of *time* for combinations of *Group* and *Task*. The individual factor analysis showed that the *Group* factor did not had a significant difference in the mean time to execute the tasks, independently of task, with $p=0.482$. Table 5.8 shows the original data means and standard deviations for *time*. Regarding time, the null hypothesis (H_{t0}) was not rejected.

Table 5.8: Original data of variable *time* at *I2*.

Task	Group	N	Mean time	Std. Dev.
Addr. Book	A	21	1585.09524	1373.42899
	B	23	1660.34783	1532.10368
Accountability	A	21	1783.76190	785.01757
	B	23	1482.13043	442.67753
Account	A	18	1468.05556	702.80944
	B	19	1613.21053	750.29924
Resource	A	15	858.66667	388.13762
	B	17	650.00000	199.35019

5.3.2.3 Experience Level Influence

In this section we check if the distribution of participants among groups *A* and *B* at *I2* was fair in terms of the experience of subjects. Each participant selected an experience level between zero and three in nine categories of knowledge: General computer science, OO languages, development using ORM, SQL, SQL extensions, XML, CMs, system design, and database design and/or administration.

Table 5.9: Group statistics for experience levels.

Group	N	Mean	Std. Deviation	Std. Error Mean
A	21	14.14	7.74	1.69
B	23	14.35	6.29	1.31

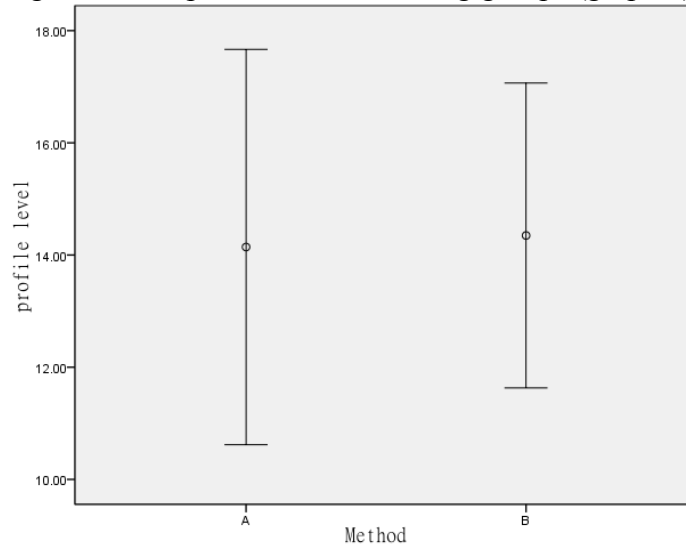
The sum of experiences (total) of each participant is our dependent variable, as summarized by Table 5.9. Thus, we checked if the total experience of the participants of groups *A* and *B* are equivalent, using a t-test.

Table 5.10: T-test for Equality of Means (Equal variances assumed).

T-test for Equality of Means	Value
t	-0.097
Sig. (2-tailed)	0.923
Mean Difference	-0.20
Std. Error Difference	2.12
95% Confidence Interval of the Difference (Lower)	-4.48
95% Confidence Interval of the Difference (Upper)	4.07

We first applied the test of *Levene* for equality of variances, that resulted in the assumption of equal variances ($F=0.75$; Sig. 0.39). Therefore, the *test-t* (Table 5.10) was applied to check if there is a variation in the mean experience level of the participants. The test showed no such variation (0.923 significance). Figure 5.5 shows a graphic of the mean experience distribution between groups using methods *A* or *B*.

Figure 5.5: Experience levels among groups (graphic).



5.3.2.4 Analysis of the Feedbacks

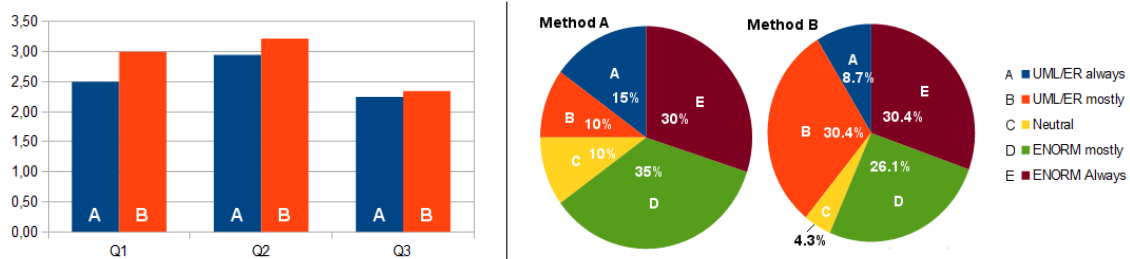
The participants of *I2* experienced only one treatment, so the questions were distinct from *II*. This time the focus was to capture difficult levels, from 1 to 5 (question 1-3), and the preferred modeling method.

1. Indicate the difficulty level for the identification of concepts, attributes, and relationships as database or software artifacts.
2. Indicate the difficulty level to accomplish the asked tasks.

3. Indicate the difficulty level to represent inheritance, many-to-many relationships and primary keys.
4. Between keeping two separated models or have only one unified model, choose the option that better reflects your preference

The results are plotted by the two graphics of Figure 5.6 The difficulty levels for identification of concepts was higher using the method **B**, but not with a big difference: the mean difficulty using relational and class models was 2.5 of 5, and using single notation was 3 of 5, a difference of 0.5. The other difficulty levels presented differences of 0.27 and 0.1 respectively to questions Q2 and Q3.

Figure 5.6: Mean difficulty levels (Q1-Q3, left) and preferred method (Q4, right).



However, applying the *Mann-Whitney U Test* over the data sets do not reject the null hypothesis, as presented by Table 5.11. This test was not corrected for ties, and the p-values could not be computed to exact values, but p is quite high. There is no significant difference between the means of difficult levels for each question at different groups.

Table 5.11: Mann-Whitney U test applied to difficult levels.

Method	N	Means			P-values		
		Q1	Q2	Q3	Q1	Q2	Q3
A	20 ⁴	2.50	2.95	2.25	0.09	0.34	0.82
B	23	3.00	3.22	2.35	0.09	0.34	0.82

The preferred method both for the subjects that experienced with ENORM (**B**) and for the subjects that were in the control group (**A**) was the single model approach. Moreover, the number of subjects that always prefer a single model is greater in the group using ENORM than in the control group. However, the percentage of participants preferring single modeling is greater in the group that used treatment **A** in comparison with the group that used treatment **B**.

5.3.3 Group Experiment (G)

The group experiment response variable was the number of *misses* in a questionnaire of auto-evaluation comprising thirteen *yes/no* questions. The total group of participants was 30, but the questionnaire, as the task itself, was answered by groups with five

⁴ One of the participants did not complete this survey.

students each. Because of that, the effective number of subjects is the number of groups (6). The data gathered was analyzed with *Mann-Whitney U Test*, and we opted for the exact test of significance (exact significance) due to the small size of the sample.

Table 5.12: Mann-Whitney ranks, means, and deviation of G .

Method	N	Mean rank	Sum ranks	Mean	Std. Deviation	Total misses
A	3	3.67	11.00	5.67	2.52	17
B	3	3.33	10.00	5.33	2.08	16
Total	6	–	–	–	–	–

Table 5.12 shows the parameters of the test, with N referring to the number of subjects (groups of five students) assigned to each treatment, along with ranks, mean, and deviation. The result of the test with $p > 0.99$ shows that there was no significant difference between the means of different methods. We could not reject the null hypothesis, and the small sample size makes difficult to generalize that there are no differences in using A nor B .

5.3.4 Analysis Summary

This section analyzed the data obtained in three experiments comparing the notation ENORM with the use of separated relational and class notations. The first two experiments compared the use of a single model against two independent models by individual subjects, the first with a *within-subjects* design. The third experiment compared the use of ENORM and relational models with class and relational models by groups of individuals simulating the integration of applications and legacy databases.

The $I1$ experiment obtained the better results for ENORM by rejecting the null hypothesis for all tasks. In $I2$, half the tasks rejected the null hypothesis favoring ENORM, and half the tasks could not reject the null hypothesis regarding the missed goals. Regarding the time variable, we could not find a correlation between the method and the measured time at $I2$. In the group experiment, we could not reject the null hypothesis regarding the missing goals.

Finally, regarding the surveys of the individual experiments, the participants had a positive opinion about the new notation. Regarding the opinion about the difficult level of the $I2$ experiment, there was no significant difference between the mean values.

A complete report about the experiments can be found at our website⁵. Appendixes A, B, C, and D presents further material describing the experimental tasks and presenting supplementary graphical view of the results. Annexes A and B includes the reports, in Portuguese, produced by the NAE.

5.4 Validity Evaluation

In this section we discuss the threats to validity that affect our study, following (WOHLIN et al., 2012).

⁵ <http://www.inf.ufrgs.br/~atorres/experiments/>

5.4.1 Internal Validity

Internal validity is concerned with the relationship between treatment and outcome. The main internal validity threats to our experiments are *maturation*, *imitation of treatments*, *compensatory rivalry*, *resentful demoralization*, and *instrumentation*.

The *maturation* threat happens due to the growing lack of interest or the learning effect. We addressed this threat by offering a training with small scale experiment and by constructing a set of cases with growing difficult level. At *I1*, we addressed the *testing* threat by not presenting the correct answers to the participants.

The *imitation of treatments* happens when subjects did not use the treatment they are expected. Because all participants had training with ENORM, we had to control this threat by limiting the available notations to the subject according to his group (*I2*, *G*), or sequence (*I1*).

Compensatory rivalry and *resentful demoralization* are opposite reactions due to the same factor, a subject that ended at an unpleasant group. This threat is stronger at *I2*, because participants experienced only one treatment. This may explain the higher preference of ENORM in the group that did not use the notation.

Finally, *instrumentation* threat is also a problem when we create models, the tools must present a similar productivity. To face this threat, we developed a tool that could be used to create all models with similar features and limitations. This, however, may introduce problems in the data collection to people used to commercial modeling tools.

5.4.2 Construct Validity

Construct validity is a matter of judging if the treatment reflects the cause construct and the outcome provides a true picture of the effect construct. The main construct validity threats are *mono-method and operation bias*, *hypothesis guessing*, and *interaction of different treatments*.

We assumed risks regarding *mono-method and operation bias*. The first due to the use of a single modeling tool, the second by having one (two, considering the time at *I2*) dependent variable.

The *hypothesis guessing* threat was alleviated by instructing the subjects about the importance of a fair dedication, despite they liking or not the new approach. It was just not possible to hide the hypothesis due to the fact that one of the treatments is new.

The *interaction of different treatments* threat is present in the cross-over experiment (*I1*), because each task is repeated using each treatment. The learning effect in this situation is inevitable, and the treatments interact. The analysis had to take in account the sequence variable, and check for the learning effect.

5.4.3 External Validity

External validity is concerned with generalization of experiment results. The main external validity threats are *interaction of setting and treatment* and *interaction of selection and treatment*.

The *interaction of setting and treatment* threat happens when we try to generalize a *toy-problem* to the real world. We addressed this by choosing problems that reflect Analysis Patterns as replacement to a real problem.

The *interaction of selection and treatment* threat is an effect of having subjects that are not representative of the population we want to generalize. By using graduate and undergraduate students, we assume a risk of using inexperienced subjects. Nevertheless, the majority of these students have work experience, and at least are in the final stage of the undergraduate course.

5.4.4 Conclusion Validity

Conclusion validity is concerned with our ability to draw the right conclusions about the relationship between the treatment and the outcome. In other words, it concerns if our data analysis is consistent with our conclusions. The main conclusion validity threats are *reliability of measures*, *random heterogeneity/homogeneity of subjects*, *random irrelevancies in experimental setting*, and *low statistical power*.

The *reliability of measures* threat is a major risk in this set of experiments. By taking models as the input of users, and setting ourselves the goals, we trade construct and external validity for less reliable measures. It is difficult to track, in an objective way, models to problems. But they reflect more closely what designers do in comparison with answering objective questions. The measures taken to check the reliability are automate and double check the achieved goals, because completely independent evaluators would not be available.

The *random heterogeneity of subjects* threat is also present, specially when using students from the same region, due to the higher homogeneity of the group. This is another threat that we have to assume at this moment, and is only alleviated by the presence of undergraduates and graduates.

The *random irrelevancies in experimental setting* threat was addressed by choosing the correct tests and performing the necessary tests to assure their validity. The population experience was also tested for the mean experience level variation at each group, checking that the participants were randomly distributed.

The *low statistical power* is a problem that mainly affects the group experiment (G). By taking each group as a subject, this experiment would require a great number of participant, or smaller groups, to perhaps reject the null hypothesis. The results strongly indicates ($p \sim 1$) no difference between the treatments.

6 CONCLUSION

This thesis presented ENORM, a single model notation to represent objects, relations, and its mappings. ENORM extends UML with a small set of new visual elements, and a meta-model comprehending the essential patterns of ORM, in the context of the most expressive ORM solutions, among popular development platforms, such as Java, Ruby, Python, and Microsoft.net.

At first we presented a survey relating the ORM patterns identified by the literature, and current practices among ORM tools. This survey addressed the relevant patterns, organizing in several criteria, and presenting examples and specific cases for JPA, RAR, SA, Entity Framework, Cayenne, and MyBatis. At the end, we summarized a set of design decisions related to the use of ORM.

At chapter 3 we presented the ENORM notation, explaining its visual elements, meta-model, special cases, limitations, and related notations. This chapter also explains the modeling tool that implements ENORM, and the future road to MDD tools employing ENORM.

At chapter 4 we focused in the *proof by code* of ENORM, by discussing the implementation of four domain models using JPA, RAR, and SA frameworks. At the end of the chapter, we summarize the concepts, identifying what are the platform specific information left out of ENORM, and what steps may be taken to implement MDD transformations.

At chapter 5 we presented the results of three controlled experiments to assess that ENORM single notation does not decrease the quality of models, independently of MDD or implementation. All experiments were focused at the modeling activity, two evaluating individuals, and one experiment evaluating groups. None of the experiments had significant disadvantage for ENORM at missed goals, or time necessary to perform the tasks. Moreover, ENORM had significant less missed goals at all tasks of the first experiment, and half the tasks on the second experiment.

Concerning the future work, in order to be considered as a complete solution, ENORM still needs a MDD tool, and there are still open questions about what is the best way to deal with code generation, round-trip engineering, and what are the MDD techniques best suited to implement such tool. Moreover, the empirical evaluation of ENORM would be stronger if a case study could be performed with real projects.

We can identify the following themes for future work:

- Empirical evaluation of ENORM with other experiments and a case study.

- Formalization of the ENORM constraints, according to the distinct frameworks constraints.
- Specification of queries and data access operations, such as finders, by models.
- Systematically identify differences between ORM frameworks, patterns, and ENORM.
- Transformations and/or code generation from/to ORM frameworks.

Despite these elements, left for future work, ENORM is a comprehensive platform independent notation for ORM design, with an original single model approach, focused at practical ORM tools, and a ready to use modeling tool for the developer community. Moreover, the feedback from the participants was positive regarding its DRY characteristics, and the maintenance of ORM systems.

REFERENCES

- ADYA, A. et al. Anatomy of the ADO.NET entity framework. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 2007. **Proceedings...** Beijing, China: ACM, 2007, p. 877–888.
- ALBAYRAK, O. An experiment to observe the impact of UML diagrams on the effectiveness of software requirements inspections. INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT, 3RD, 2007. **Proceedings...** Washington DC: IEEE Computer Society, 2009. p. 506-510
- AMBLER, S. **Agile database techniques: Effective strategies for the agile software developer**. New York: John Wiley, 2002.
- AMBLER, S.; HARTFORD, E.; RUECKERT, A. A UML Profile for Data Modeling. Available at: <<http://www.agiledata.org/essays/umlDataModelingProfile.html>>. Accessed on: apr. 2014.
- AMBLER, S. W. **Agile modeling: Effective practices for eXtreme programming and the unified process**. 1st edition. New York: J. Wiley, 2002.
- AMERICAN NATIONAL STANDARDS INSTITUTE. **ANSI/X3/SPARC Study Group on Data Base Management Systems; Interim Report. FDT (Bulletin of ACM SIGMOD)** [S.l.:s.n], 1975.
- APACHE FOUNDATION. CompositeSet (Commons Collections 3.2.1 API). Available at: <<http://commons.apache.org/proper/commons-collections/javadocs/api-3.2.1/org/apache/commons/collections/set/CompositeSet.html>>. Accessed on: jan. 2014.
- APACHE FOUNDATION. Apache Cayenne. Available at: <<http://cayenne.apache.org/>>. Accessed on: apr. 2014.
- ATKINSON, M. P.; BUNEMAN, O. P. Types and persistence in database programming languages. **ACM Computing Surveys**, v. 19, n. 2, p. 105–170. New York: ACM, 1987.
- ATZENI, P.; JENSEN, C. S.; ORSI, G.; et al. The relational model is dead, SQL is dead, and I don't feel so good myself. **SIGMOD Rec.**, v. 42, n. 1, p. 64–68. New York: ACM, 2013.
- BATRA, D.; HOFFLER, J. A.; BOSTROM, R. P. Comparing representations with relational and EER models. **Commun. ACM**, v. 33, n. 2, p. 126–139. New York: ACM, 1990.
- BAUER, C.; KING, G. **Hibernate in Action**. 2nd edition. New York: Manning Publications, 2004.

- BAYER, M. SQLAlchemy - The Database Toolkit for Python. Available at: <<http://www.sqlalchemy.org/>>. Accessed on: apr. 2014.
- BEGIN, C.; GOODIN, B.; MEADORS, L. **iBatis in Action**. 1st edition. New York: Manning Publications, 2007.
- BERLER, M.; EASTMAN, J.; JORDAN, D.; et al. **The object data standard: ODMG 3.0** (R. G. G. Cattell & D. K. Barry, Eds.) Morgan Kaufmann Publishers Inc., 2000.
- BERNSTEIN, P. A.; MELNIK, S. Model management 2.0: manipulating richer mappings. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 2007. **Proceedings...** New York: ACM, 2007. p. 1-12..
- BEYDEDA, S.; BOOK, M.; GRUHN, V. **Model-Driven Software Development**. 1st edition. New York: Springer, 2005.
- BORK, M.; GEIGER, L.; SCHNEIDER, C.; ZÜNDORF, A. Towards Roundtrip Engineering - A Template-Based Reverse Engineering Approach. In: I. Schieferdecker; A. Hartman (Eds.); **Model Driven Architecture – Foundations and Applications**, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, p. 33–47.
- BROWN, K.; WHITENACK, B. G. Crossing chasms: a pattern language for object-RDBMS integration: the static patterns. In: **Pattern languages of program design 2**. Boston: Addison-Wesley, 1996. p. 227-238.
- CARBONNELLE, PI. PYPL PopularitY of Programming Language index - pyDatalog. Available at: <<https://sites.google.com/site/pydatalog/pypl/PyPL-PopularitY-of-Programming-Language>>. Accessed on: feb. 2014.
- CHEN, N. Convention over Configuration. Available at: <<http://softwareengineering.vazexqi.com/files/pattern.html>>. Accessed on: feb. 2014.
- CHEN, P. P.-S. The entity-relationship model - toward a unified view of data. **ACM Trans. Database Syst.**, v. 1, n. 1, p. 9–36. New York: ACM, 1976.
- COPELAND, G.; MAIER, D. Making smalltalk a database system. **SIGMOD Rec.**, v. 14, n. 2, p. 316–325. New York: ACM, 1984.
- CRUZ-LEMUS, J. A.; GENERO, M.; CAIVANO, D.; et al. Assessing the influence of stereotypes on the comprehension of UML sequence diagrams: A family of experiments. **Information and Software Technology**, v. 53, n. 12, p. 1391–1403. Newton, MA, USA: Butterworth-Heinemann, 2011.
- DAYAL, U.; BERNSTEIN, P. A. On the correct translation of update operations on relational views. **ACM Trans. Database Syst.**, v. 7, n. 3, p. 381–416. New York: ACM, 1982.
- DEMICHIEL, L. JSR-000338 Java Persistence 2.1 - Final Release. Available at: <<https://jcp.org/aboutJava/communityprocess/final/jsr338/index.html>>. Accessed on: apr. 2014.
- DERSTADT, J.; VEGA, D. POCO Proxies Part 1 - ADO.NET team blog - Site Home - MSDN Blogs. Available at: <<http://blogs.msdn.com/b/adonet/archive/2009/12/22/poco-proxies-part-1.aspx>>. Accessed on: oct. 2012.

DISKIN, Z.; XIONG, Y.; CZARNECKI, K. From State- to Delta-Based Bidirectional Model Transformations. In: L. Tratt; M. Gogolla (Eds.); **Theory and Practice of Model Transformations**, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, p. 61–76.

DR NIC WILLIAMS; CHARLIE SAVAGE. Composite Primary Keys. Available at: <<http://compositekeys.rubyforge.org/>>. Accessed on: feb. 2014.

ECLIPSE FOUNDATION. Eclipse.org home. Available at: <<http://www.eclipse.org/>>. Accessed on: jan. 2014a.

ECLIPSE FOUNDATION. GEF. Available at: <<http://www.eclipse.org/gef/>>. Accessed on: apr. 2014b.

ELMASRI, R.; NAVATHE, S. B. **Fundamentals of Database Systems**. 4th edition. Boston, MA, USA: Addison Wesley, 2003.

FOWLER, M. **Analysis Patterns: Reusable Object Models** Boston, MA, USA: Addison-Wesley Professional, 1996.

FOWLER, M. Martin Fowler Bliki: POJO. Available at: <<http://www.martinfowler.com/bliki/POJO.html>>. Accessed on: apr. 2014.

FOWLER, M. **Patterns of Enterprise Application Architecture**. Boston, Massachusetts, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. M. **Design Patterns: Elements of Reusable Object-Oriented Software**. 1st edition. Boston, MA, USA: Addison-Wesley Professional, 1994.

GARCIA-MOLINA, H.; ULLMAN, J. D.; WIDOM, J. **Database Systems: The Complete Book**. 2nd edition. Upper Saddle River, New Jersey, USA: Prentice Hall, 2008.

GEMINO, A.; WAND, Y. Complexity and clarity in conceptual modeling: Comparison of mandatory and optional properties. **Data & Knowledge Engineering**, v. 55, n. 3, p. 301–326. Amsterdam, The Netherlands: Elsevier Science Publishers, 2005.

GOSLING, J.; JOY, B.; STEELE, G.; BRACHA, G. **Java(TM) Language Specification, The (3rd Edition)**. 3rd edition. USA: Addison Wesley, 2005.

HARTFORD, E. XMI2SQL. Available at: <<http://xmi2sql.sourceforge.net/>>. Accessed on: apr. 2014.

HEINEMEIER HANSSON, D. Active Record - Object-relation mapping put on rails. Available at: <<http://ar.rubyonrails.org/>>. Accessed on: apr. 2014.

HOFMANN, M.; PIERCE, B.; WAGNER, D. Symmetric lenses. In: ANNUAL ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 38th, 2011. **Proceedings...** New York: ACM, 2011. p. 371-384.

HUNT, A.; THOMAS, D. **The Pragmatic Programmer: From Journeyman to Master**. 1st edition. Boston, MA, USA: Addison-Wesley Professional, 1999.

ISO. ISO/IEC 14977:1996 - Information technology -- Syntactic metalanguage -- Extended BNF. Available at: <http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=26153>. Accessed on: feb. 2014.

JAIN, S. K.; GORE, M. M.; SINGH, G. An Experimental Study to Compare ER/EER and OO Models. **TECHNIA - International Journal of Computing Science and Communication Technologies**, p. 221–228. [S.l.:s.n].

JOUAULT, F. et al. ATL: a QVT-like transformation language. In: COMPANION ACM SIGPLAN SYMPOSIUM ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS, 21st, 2006. **Proceedings...** Portland, Oregon, ACM, 2006. p. 719-720.

JÚNIOR, J. U.; PENTEADO, R. D.; DE CAMARGO, V. V. An overview and an empirical evaluation of UML-AOF: an UML profile for aspect-oriented frameworks. In: ACM SYMPOSIUM ON APPLIED COMPUTING, 2010. **Proceedings...** New York: ACM, 2010. p. 2289-2296.

KELLER, A. M. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In: ACM SIGACT-SIGMOD SYMPOSIUM ON PRINCIPLES OF DATABASE SYSTEMS, 4th, 1985. **Proceedings...** New York: ACM, 1985. p. 154-163.

KELLER, W. Mapping objects to tables: a pattern language. In: EUROPEAN PATTERN LANGUAGES OF PROGRAMMING CONFERENCE, 1997. **Siemens Technical Report 120/SW1/FB 1997**. Irrsee, Germany: [s.n.], 1997.

KO, A. J.; LATOZA, T. D.; BURNETT, M. M. A practical guide to controlled experiments of software engineering tools with human participants. **Empirical Software Engineering**, p. 1–32. New York: Springer Science, 2013.

LUCIA, A. D.; GRAVINO, C.; OLIVETO, R.; TORTORA, G. An experimental comparison of ER and UML class diagrams for data modelling. **Empirical Software Engineering**, v. 15, n. 5, p. 455–492. New York: Springer Science, 2010.

MARTIN, J. **Managing the data-base environment**. 1st edition. Englewood Cliffs, NJ, USA: Prentice-Hall, 1983.

MELLOR, S. J.; SCOTT, K.; UHL, A.; WEISE, D. **MDA Distilled: Principles of Model-Driven Architecture**. 1st edition. Reading, Massachusetts: Addison-Wesley Professional, 2004.

MICROSOFT. ADO.NET Entity Framework. Available at: <<http://msdn.microsoft.com/en-us/library/bb399572.aspx>>. Accessed on: jan. 2014a.

MICROSOFT. EntityObject Class (System.Data.Objects.DataClasses). Available at: <<http://msdn.microsoft.com/en-us/library/system.data.objects.dataclasses.entityobject.aspx>>. Accessed on: apr. 2014b.

MILLER, G. A. The magical number seven, plus or minus two: some limits on our capacity for processing information. **Psychological Review**, v. 63, n. 2, p. 81–97. Washington, DC, USA: American Psychological Association, 1956.

MITCHELL, R. J. **Managing complexity in software engineering**. v. 17. London, UK: Peter Peregrinus Ltd., 1990.

DE MONTMOLLIN, G. The Transparent Language Popularity Index. Available at: <<http://lang-index.sourceforge.net/>>. Accessed on: feb. 2014.

O'GRADY, S. The RedMonk Programming Language Rankings: January 2013 – tecosystems. Available at: <<http://redmonk.com/sogrady/2013/02/28/language-rankings-1-13/>>. Accessed on: feb. 2014.

OMG. OMG's Model Driven Architecture. Available at: <<http://www.omg.org/cgi-bin/doc?omg/03-06-01>>. Accessed on: apr. 2014.

OMG. Request For Proposal Information Management Metamodel (IMM). Available at: <http://www.omgwiki.org/imm/lib/exe/fetch.php?id=welcome_to_imm&cache=cache&media=05-12-02.pdf>. Accessed on: apr. 2014.

OMG. XML Metadata Interchange. Available at: <<http://www.omg.org/spec/XMI/Current/>>. Accessed on: apr. 2014.

OMG. QVT 1.1. Available at: <<http://www.omg.org/spec/QVT/1.1/>>. Accessed on: apr. 2014a.

OMG. UML 2.4.1 Superstructure. Available at: <<http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/>>. Accessed on: apr. 2014b.

OMG. Welcome to IMM - Information Management Metamodel. Available at: <<http://www.omgwiki.org/imm/doku.php>>. Accessed on: apr. 2014.

ORACLE. What is Java Web Start and how is it launched? Available at: <http://www.java.com/en/download/faq/java_webstart.xml>. Accessed on: apr. 2014.

PRESSMAN, R. **Software Engineering: A Practitioner's Approach**. 5th edition. New York, NY: McGraw-Hill Science/Engineering/Math, 2001.

PRESSMAN, R. S. **Software engineering: a practitioner's approach**. New York: McGraw-Hill Higher Education, 2010.

RED HAT MIDDLEWARE. Hibernate Tools - JBoss Community. Available at: <<http://hibernate.org/tools/>>. Accessed on: apr. 2014.

ROSENTHAL, R.; ROSNOW, R. L. **Essentials of behavioral research: methods and data analysis** Boston: McGraw-Hill, 2007.

RUBYONRAILS.ORG. ActiveRecord::NestedAttributes::ClassMethods. Available at: <<http://api.rubyonrails.org/classes/ActiveRecord/NestedAttributes/ClassMethods.html>>. Accessed on: apr. 2014.

SNEED, T. What's New and Cool in Entity Framework 4.0 - DevelopMentor. Available at: <<http://www.develop.com/entityframework4>>. Accessed on: apr. 2014.

STÅLHANE, T.; SINDRE, G. Safety hazard identification by misuse cases: experimental comparison of text and diagrams. In: MODEL DRIVEN ENGINEERING LANGUAGES AND SYSTEMS, 2008. **Proceedings...** Heidelberg: Springer, 2008. p. 721-735. Lecture Notes in Computer Science, v. 5301.

TEOREY, T. J.; YANG, D.; FRY, J. P. A logical design methodology for relational databases using the extended entity-relationship model. **ACM Comput. Surv.**, v. 18, n. 2, p. 197-222. New York: ACM, 1986.

TIOBE. TIOBE Software: Tiobe Index. **Tiobe index**. Available at: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>. Accessed on: apr. 2013.

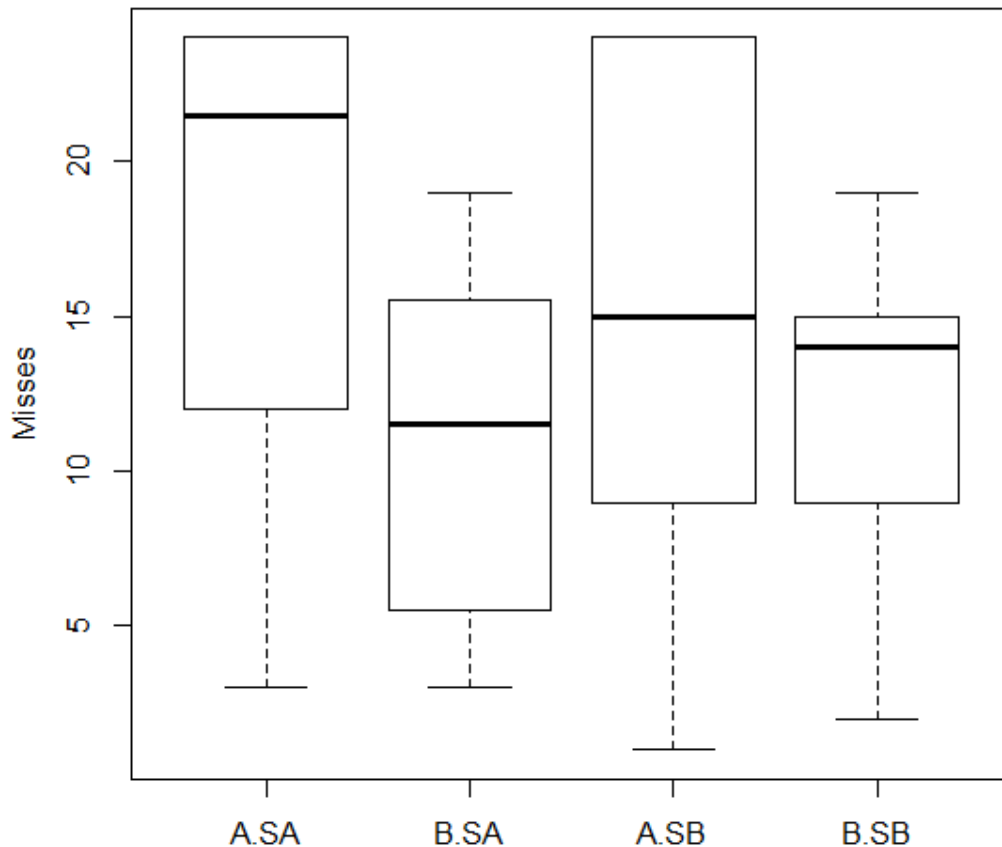
WOHLIN, C.; RUNESON, P.; HÖST, M.; et al. **Experimentation in Software Engineering**. Electronic edition. New York, NY, USA: Springer, 2012.

APPENDIX A – CROSSOVER EXPERIMENT (I1)

The graphics presented at this section were produced using R version 3.0.2. The statistical analysis was performed using Statistical Analysis System (SAS), version 9.2.

A.1 Address Book Task graphics

Figure A.1: Box-plot of misses ~method*sequence for address book at I1.



Where methods are A (control) and B (ENORM), and sequence can assume SA when the participant performs first the task A, and SB when the participant performs first the task B. A.SA are the results of method A when the subject starts with A, B.SA are the results of method B when the subject starts with A, and so on.

Figure A.2: Normal q-q plot for address book at I1.

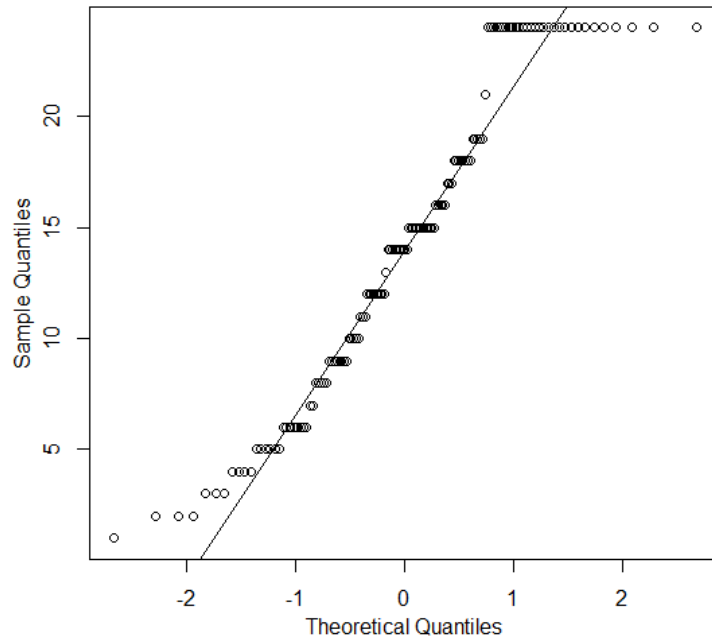
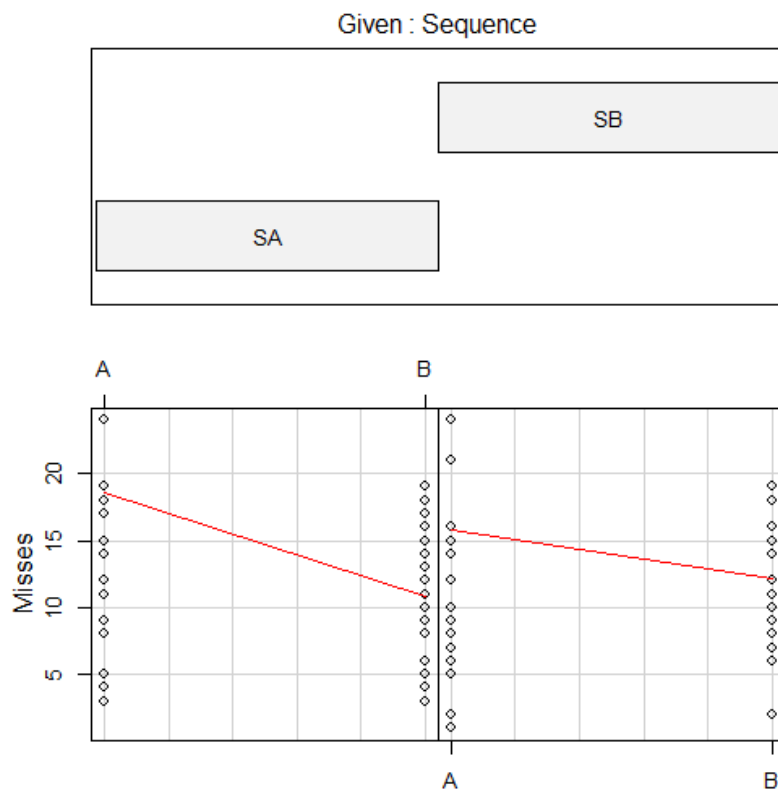
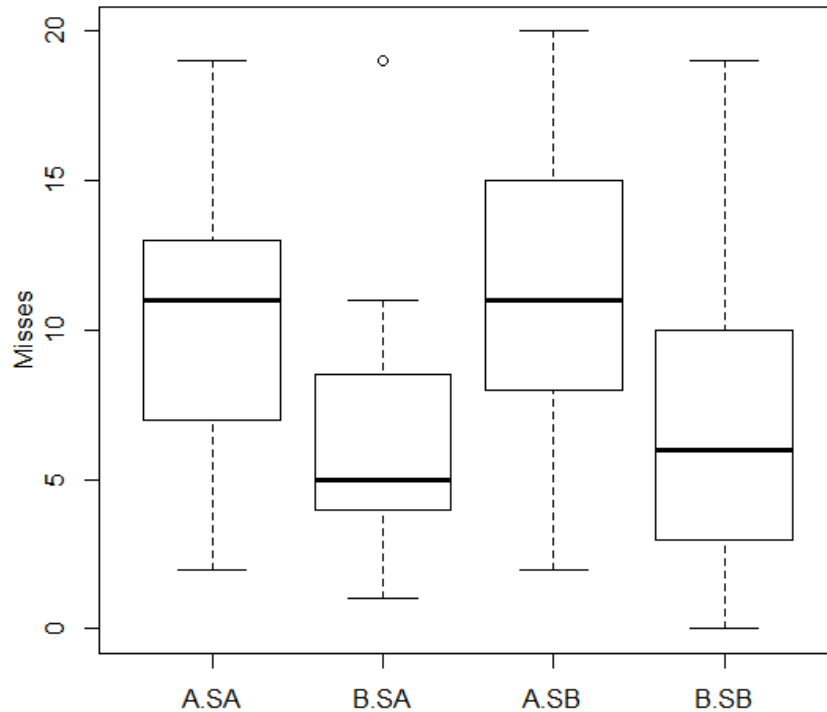


Figure A.3: Misses x method, given the sequence, for address book at I1.



A.2 Accountability Task Graphics

Figure A.4: Box-plot of misses ~ method*sequence for Accountability I1.



Where methods are A (control) and B (ENORM), and sequence can assume SA when the participant performs first the task A, and SB when the participant performs first the task B. A.SA are the results of method A when the subject starts with A, B.SA are the results of method B when the subject starts with A, and so on.

Figure A.5: Normal q-q plot for Accountability I1.

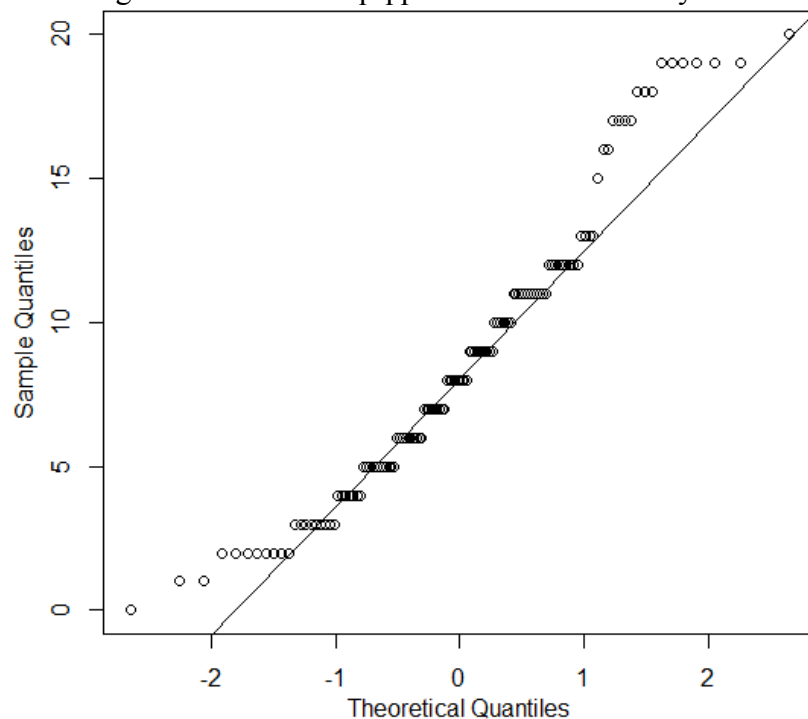
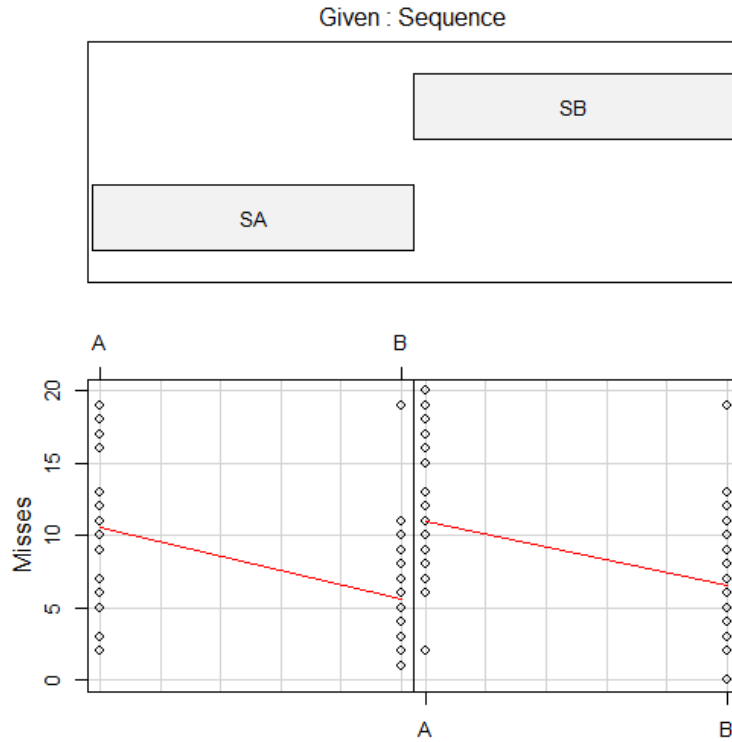
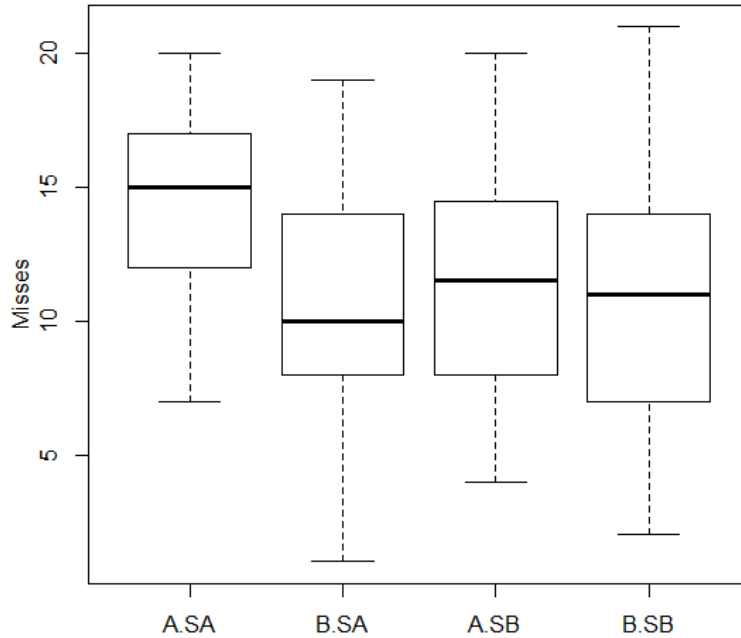


Figure A.6: Misses x method, given the sequence, for Accountability I1.



A.3 Account

Figure A.7: Box-plot of misses ~ method*sequence for Account I1.



Where methods are A (control) and B (ENORM), and sequence can assume SA when the participant performs first the task A, and SB when the participant performs first the task B. A.SA are the results of method A when the subject starts with A, B.SA are the results of method B when the subject starts with A, and so on.

Figure A.8: Normal q-q plot for Account I1.

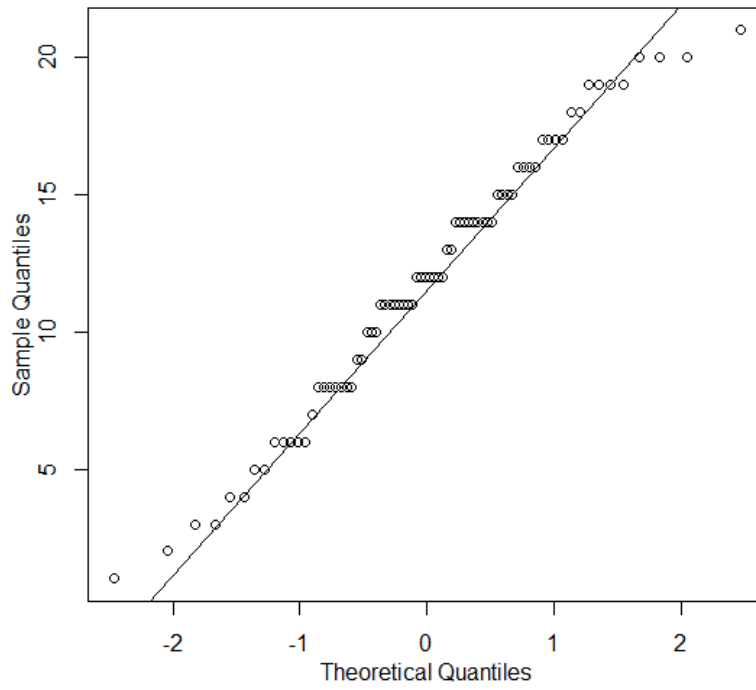
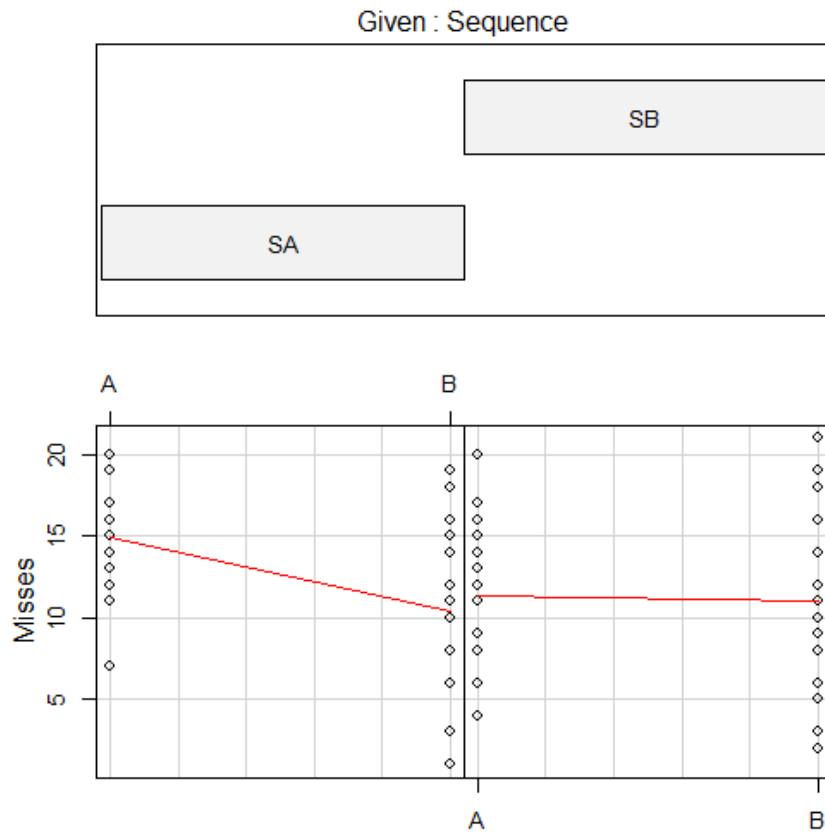
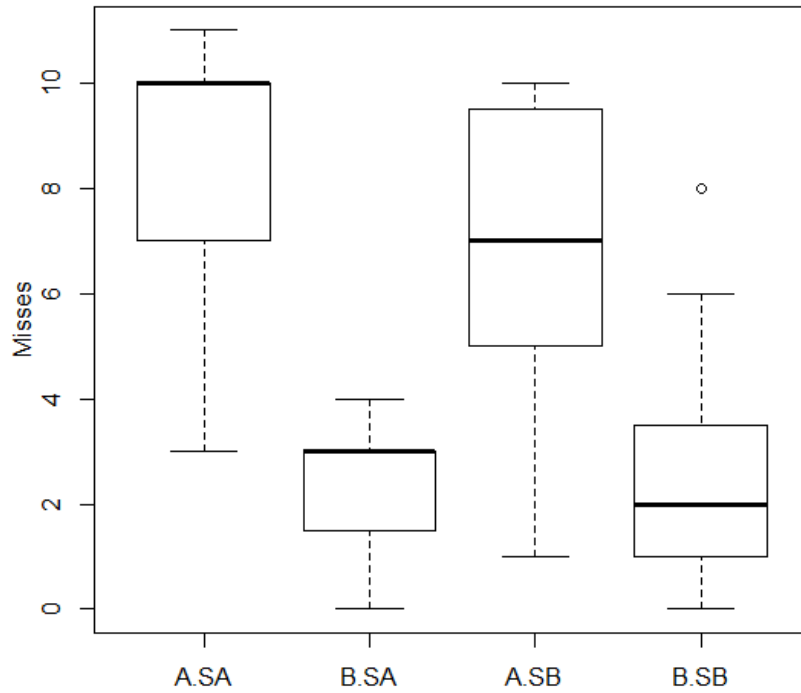


Figure A.9: Misses x method, given the sequence, for Account I1.



A.4 Resources

Figure A.10: Box-plot of misses ~ method*sequence for Resource Allocation I1.



Where methods are A (control) and B (ENORM), and sequence can assume SA when the participant performs first the task A, and SB when the participant performs first the task B. A.SA are the results of method A when the subject starts with A, B.SA are the results of method B when the subject starts with A, and so on.

Figure A.11: Normal q-q plot for Resource Allocation I1.

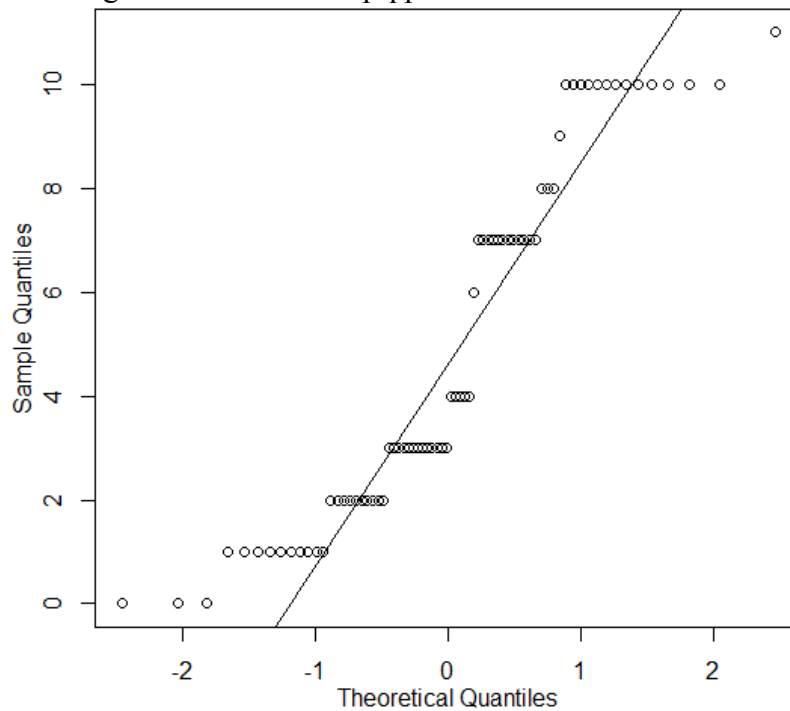
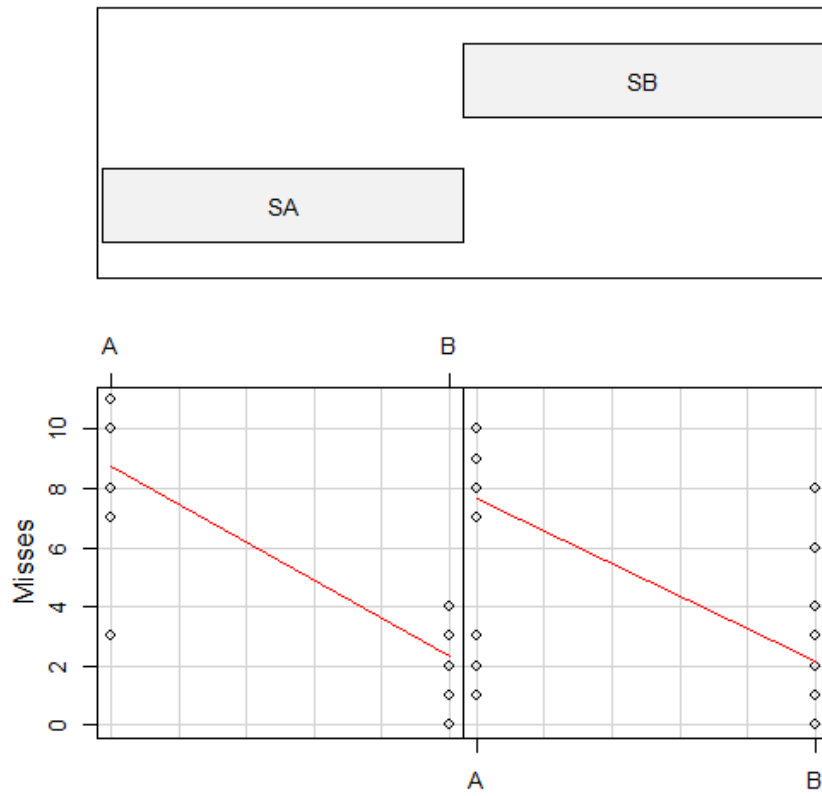


Figure A.12: Misses x method, given the sequence, for Resource Allocation II.
Given : Sequence



APPENDIX B – NON-CROSSOVER EXPERIMENT (I2)

The graphics presented at this section were produced using R version 3.0.2. The statistical analysis was performed using Statistical Analysis System (SAS), version 9.2.

Figure B.1: Box-plots of mixes by group, for each task of I2.

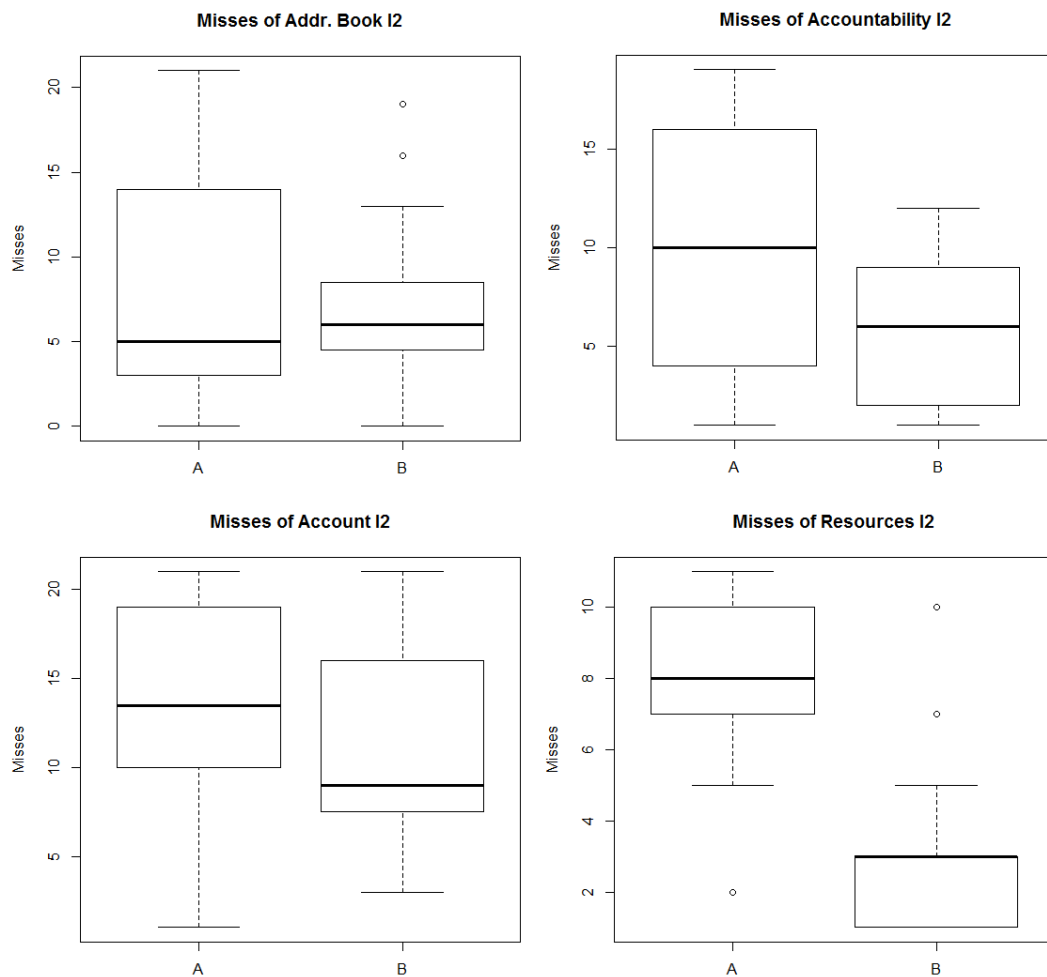


Figure B.2: Box-plots of time (in seconds) by group, for each task of I2.

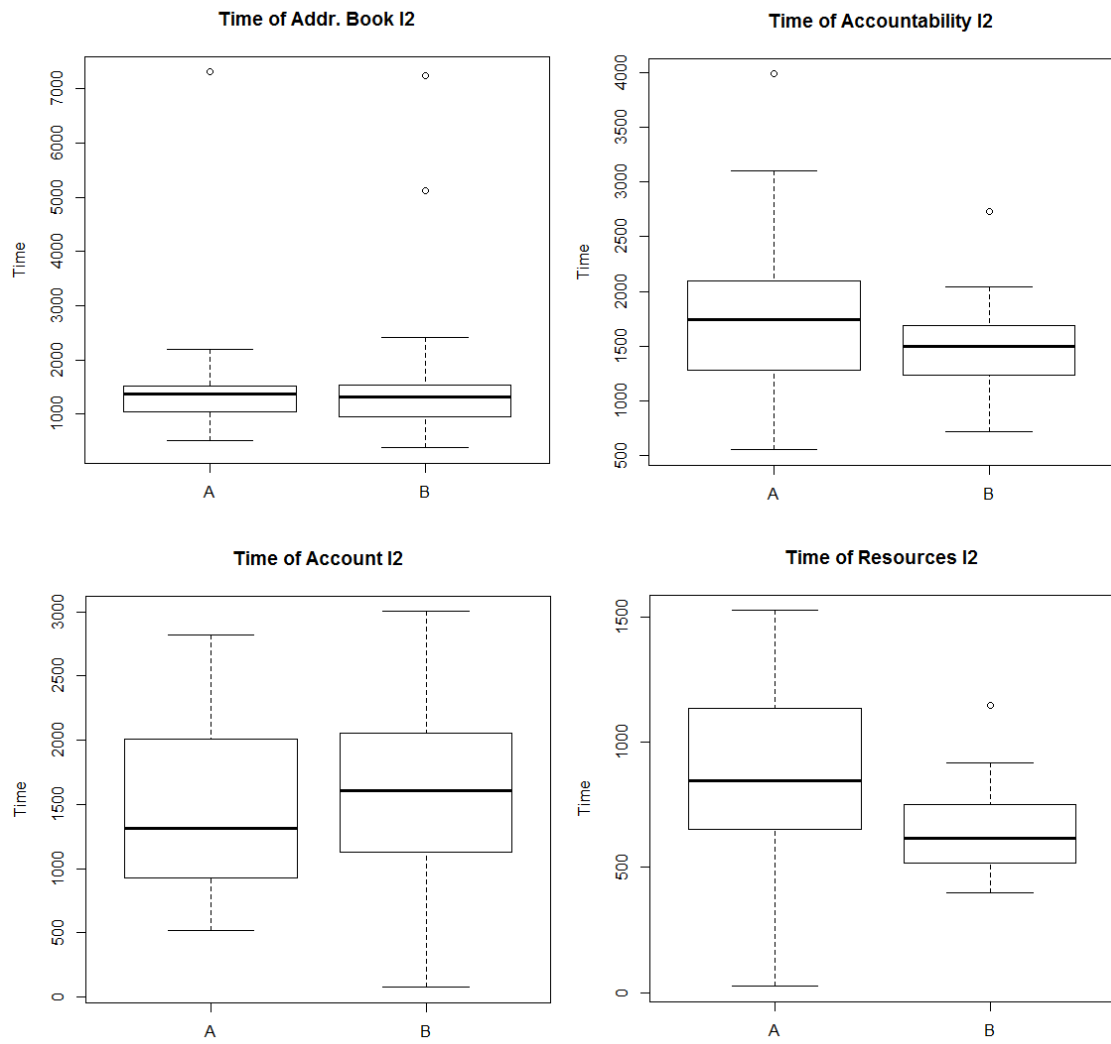


Figure B.3: Quantiles of misses, for each task of I2.

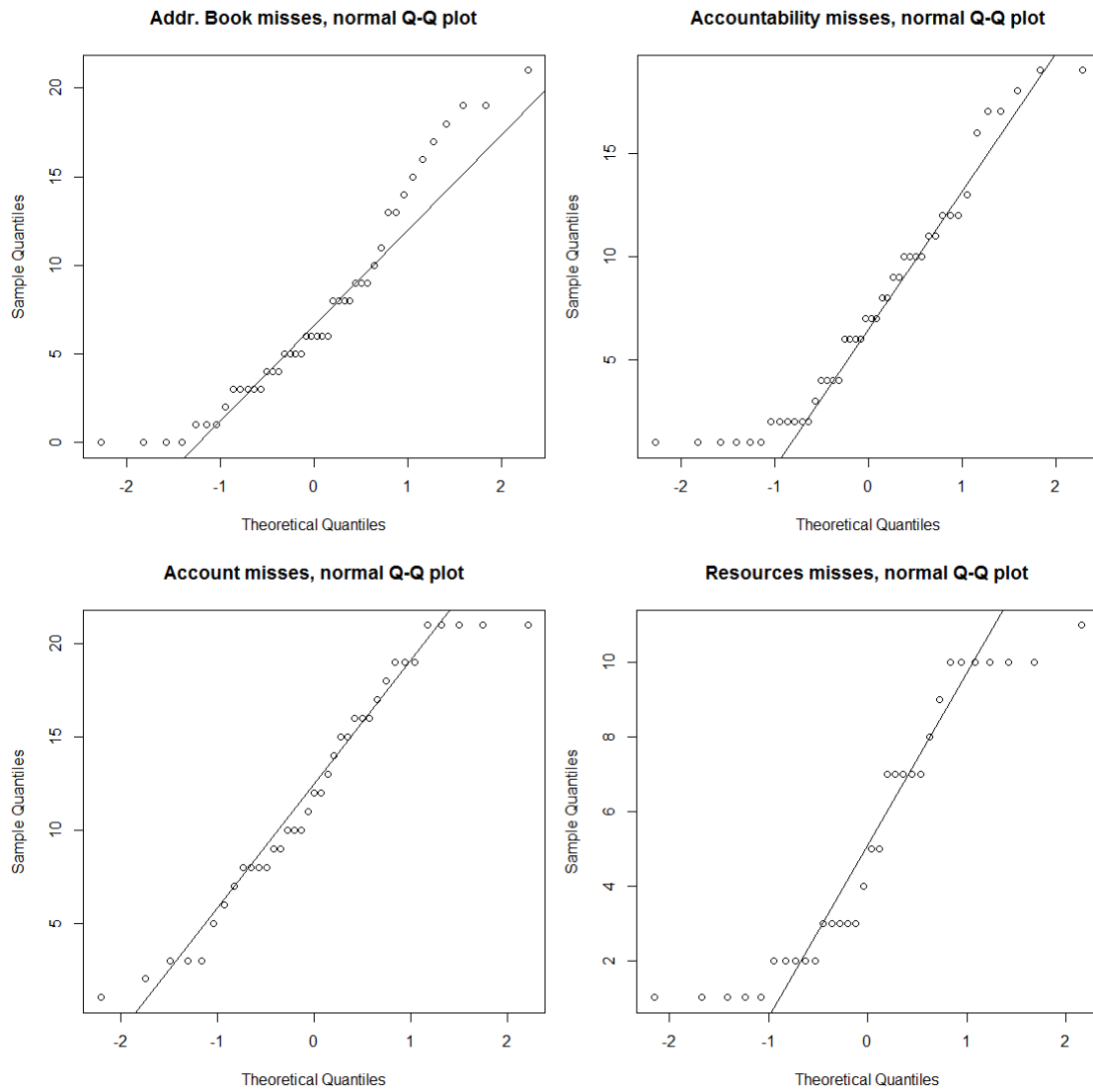
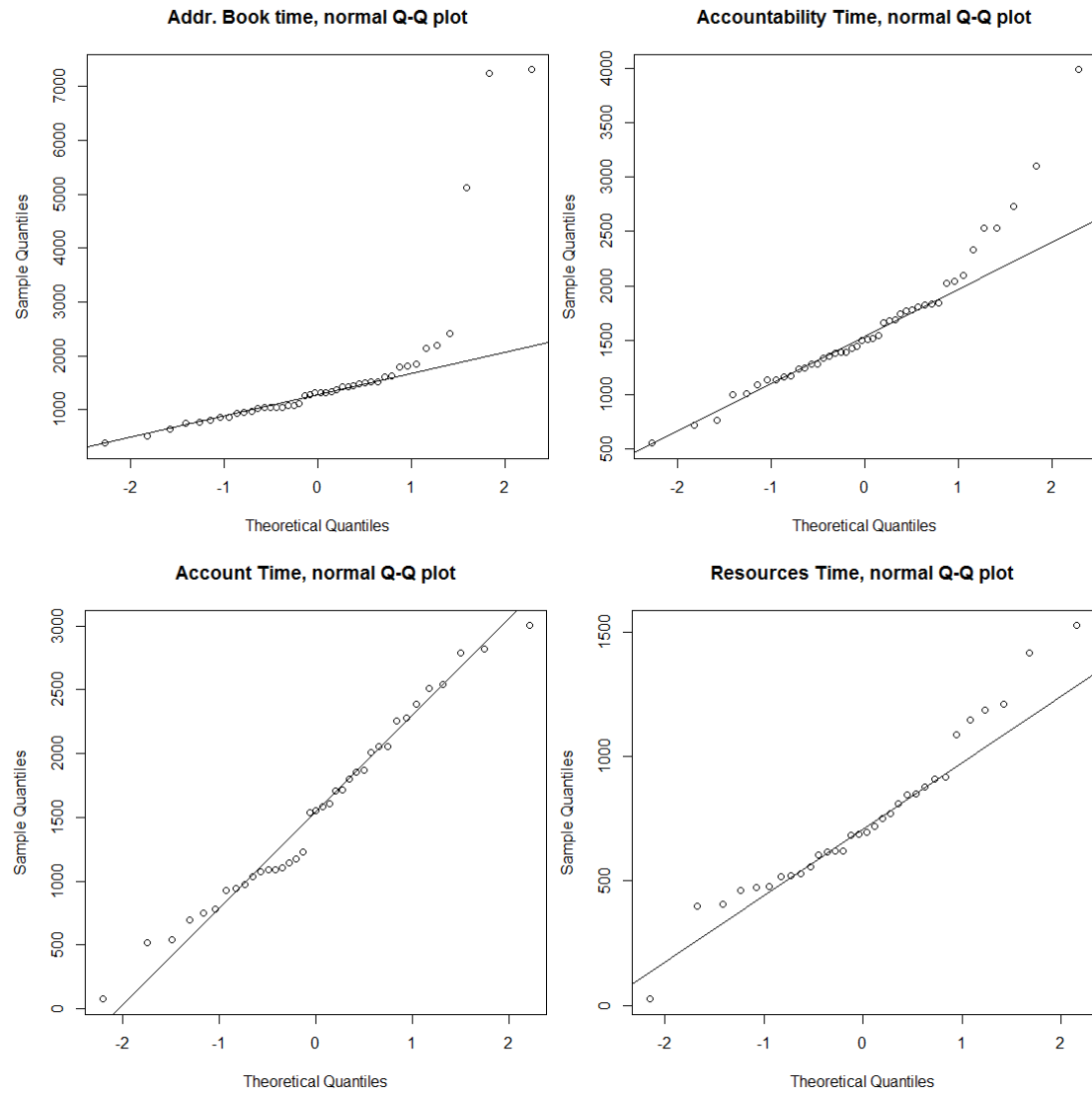


Figure B.4: Quantiles of time, for each task of I2.



APPENDIX C – TASKS (INDIVIDUAL EXPERIMENTS)

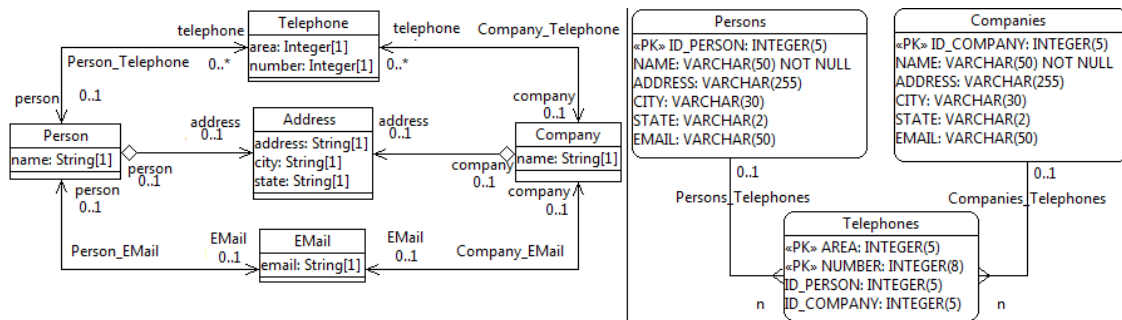
All instructions were translated from the original in Portuguese. The tasks were originally presented by the modeling tool to the participants.

C.1 Address Book

Instructions for treatments A and B.

C.1.1 Instructions - Treatment A

Figure C.1: Initial Address Book UML and ER models.

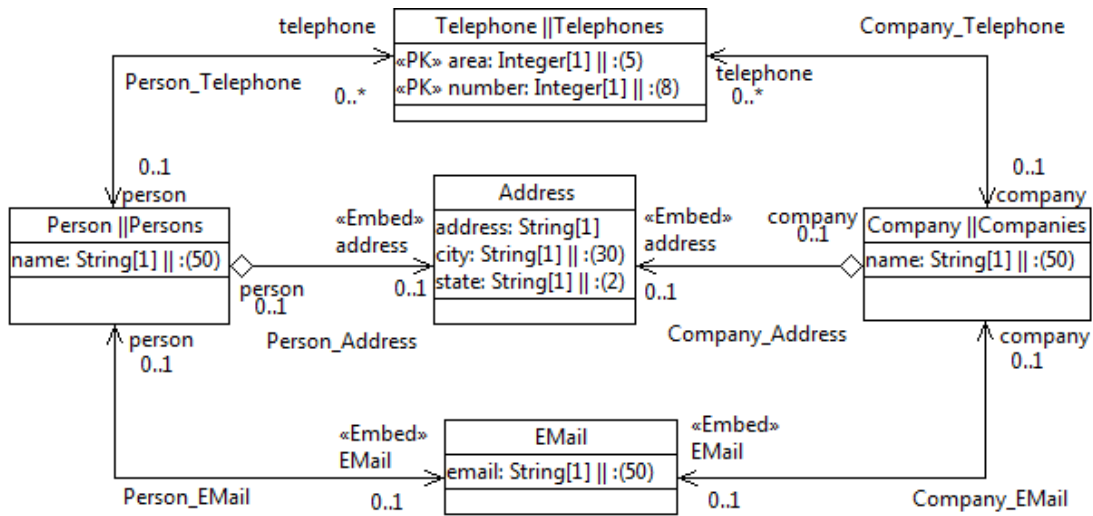


- Person and Company will now specialize the new abstract class Party.
- Person and Company will now be persisted at the same table named PARTIES. The PK of PARTIES should have the same type and length of Persons's PK and should be named "ID_PARTY".
- Move the name attribute (common on Person and Company) to the Party class.
- PARTIES table should have a not null column named PARTY_TYPE with length of one, discriminating party types between Person (PARTY_TYPE="P") and Company (PARTY_TYPE="C"). Specify that this column can only assume one of these two values.
- Relationships from Person and Company to classes Address, Telephone and Email will be unified to relationships between Party and Address, Telephone and Email.
- Telephone should relate to exactly one Party, but a party may have many telephones.

C.1.2 Instructions - Treatment B

- Person and Company will now specialize the new abstract class Party.
- Use the flat inheritance mapping between Party and its subclasses Person and Company. Specify the discriminator named PARTY_TYPE assuming values P for person or C for company.

Figure C.2: Initial Address Book ENORM model.



- c) Move the name attribute (common on Person and Company) to the Party class.
- d) Relationships from Person and Company to classes Address, Telephone and Email will be unified to relationships between Party and Address, Telephone and Email.
- e) Telephone should relate to exactly one Party, but a party may have many telephones.

C.1.3 Expected Results

Figure C.3: Expected models (UML and database).

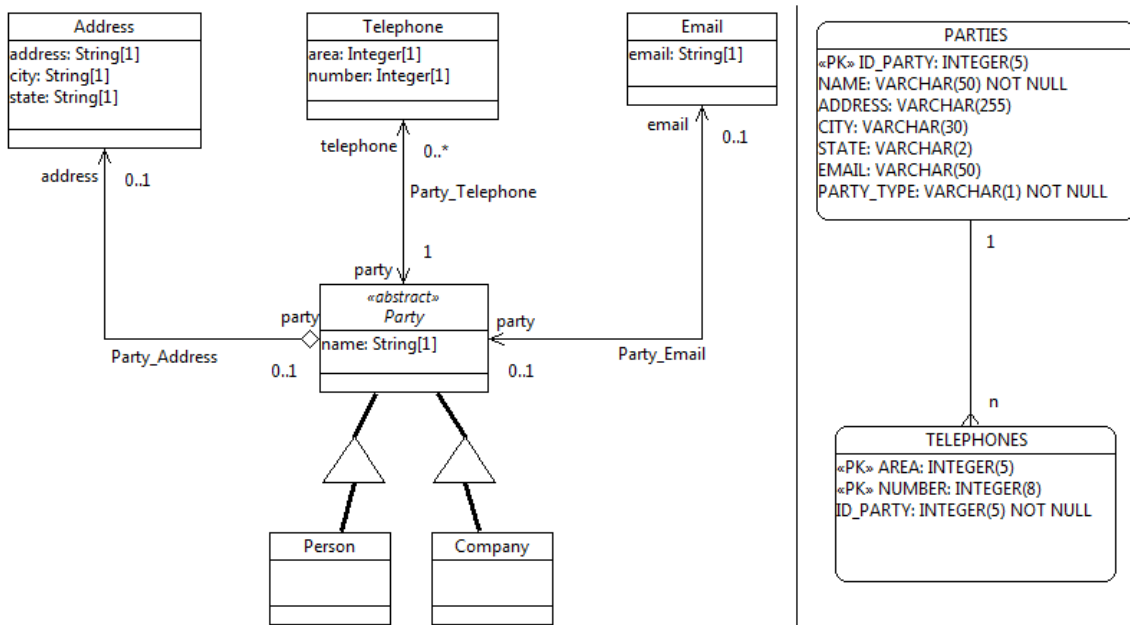
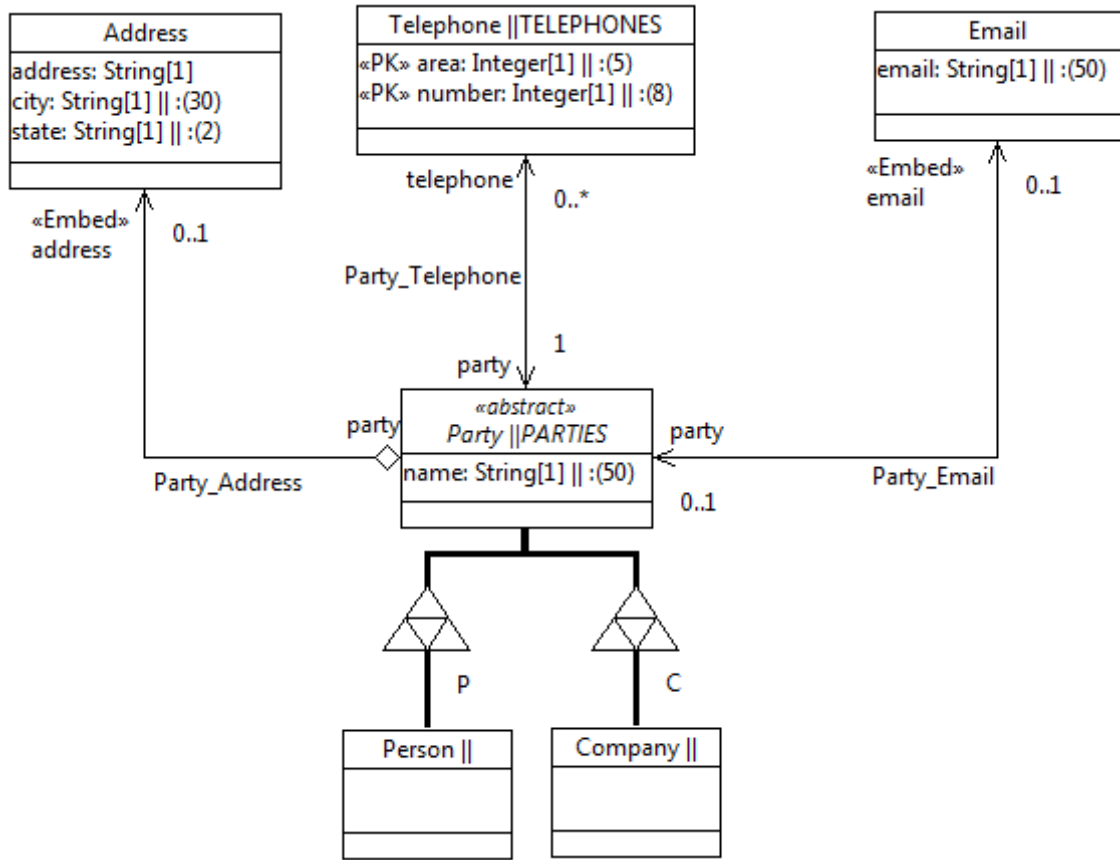


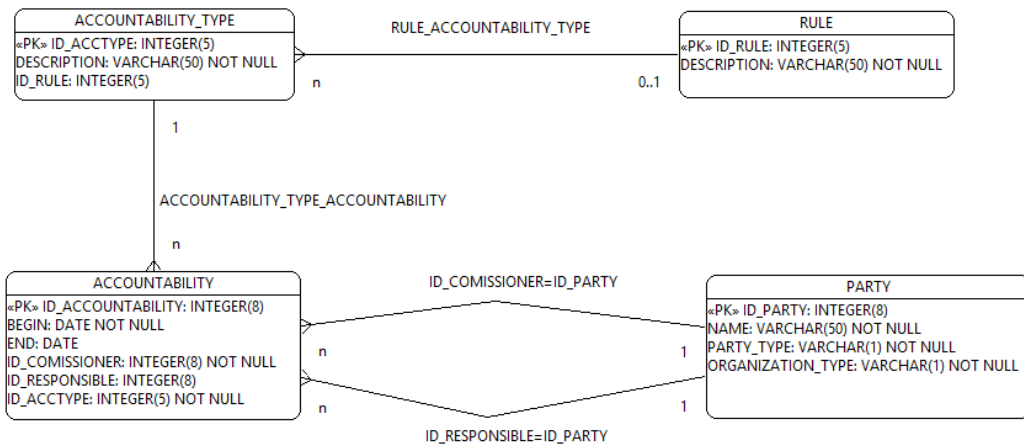
Figure C.4: Expected ENORM model.



C.2 Accountability

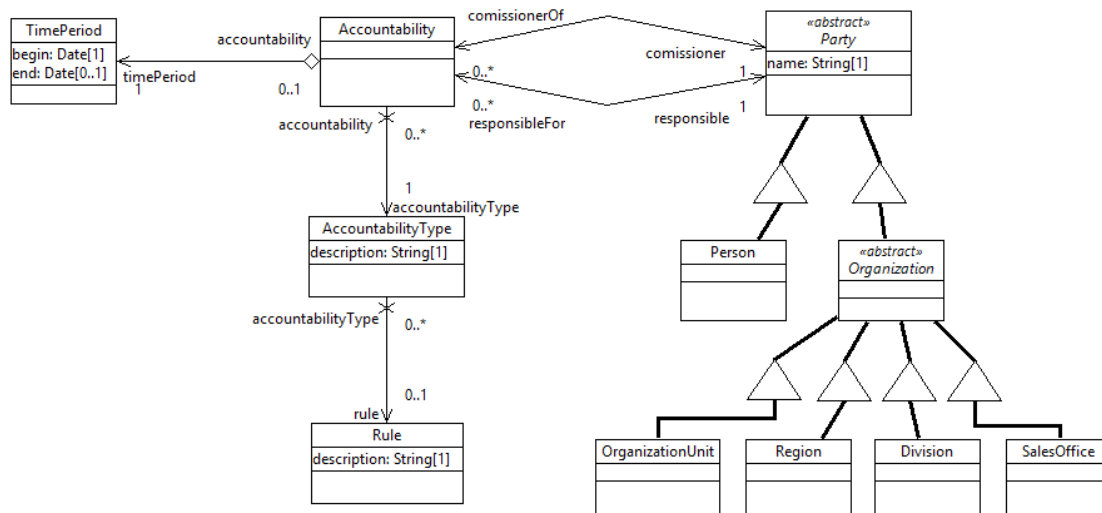
C.2.1 Instructions - Treatment A

Figure C.5: Initial Accountability persistence model.



Purpose: To represent all responsibility relationships between two "parties" such as who is the chief of *John Doe*, who manages the division of *Minas Gerais* or offices that are under the supervision of *Serra Gaucha* division.

Figure C.6: Initial Accountability UML class model.



Description of the concepts represented in the model:

Party = Legal party, general concept that encompasses entities such as persons and organizations.

Accountability = Responsibility between two parties. The “**Responsible**” association identifies who is the responsible for the entity represented by the “**Commissioner**” association.

AccountabilityType = A type of responsibility registered. For instance: the accountability of a division over a sales office; the accountability of a person as manager of divisions, sales offices, or regions.

Rule = Rule that determines what type of **Party** can assume the positions of **Commissioner** and **Responsible**. The rules are previously implemented and identified by this relation. For example, an instance of **Rule** named “SalesOfficeToDivision” restricts the relation for the accountability of one Division relating to one commissioned sales office.

Task:

The current model has a limited number of organization classifications, demanding the creation of new classes to represent each type of “**Party**”. The goal is to refactor the model, allowing the dynamic registration of the existing types of parties. Another goal is to register how the types of Party affect the accountability types (**AccountabilityType**). Create a new model with mapped classes that includes all of the following changes:

1. Remove the subtypes of **Organization**. **Organization** is no longer abstract.
2. Create a persistent class named **PartyType** that represents the types of **Party**. Add a mandatory property named **description**, with String type and maximum length of 50.
3. Create a unidirectional association between **Party** and **PartyType**. One **Party** will have one-to-many “**PartyType**”. One **PartyType** will have zero/one-to-many “**Party**”.
4. Remove the **Rule** class.
5. Create an association between **AccountabilityType** and **PartyType** classes, named **commissioners**. This relation should be of zero/one-to-many **AccountabilityType** to

one-to-many **PartyType**. To persist this relation, define a join table with the name **ACCTYPE_COMMISSIONERS**.

6. Create an association between classes **AccountabilityType** and **PartyType** named **responsibles**. This relation should be of zero/one-to-many **AccountabilityType** to one-to-many **PartyType**. To persist this relation, define a join table named **ACCTYPE_RESPONSIBLES**.

7. Name the ends so that associations of steps 5 and 6 could be distinguished.

8. Perform the necessary changes on the database mappings reflecting the required changes, with the following guidelines:

a) Specify the name of the join column attribute when there is two or more one-to-many relationships between two classes.

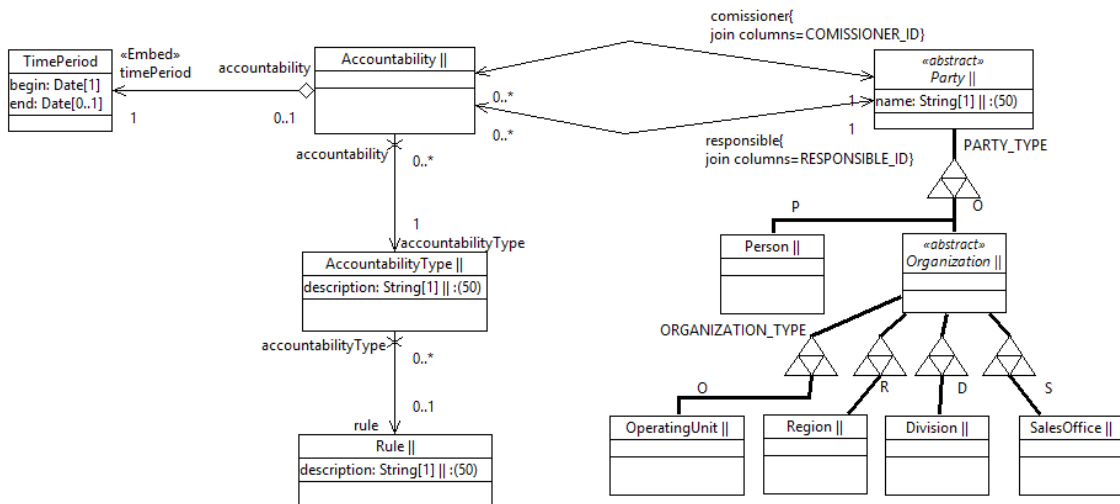
b) Specify a name for the join tables when there is two or more relationships many-to-many between two classes. By standard, two instances (a1,b1) can only relate at most one time, but may relate to any other instances as well, for example (a1,b2; a5,b1).

Guideline:

- The models should contain the minimal necessary changes (and nothing more) to meet the requirements.

C.2.2 Instructions - Treatment B

Figure C.7: Initial Accountability ENORM model.



Purpose: To represent all responsibility relationships between two "parties" such as who is the chief of *John Doe*, who manages the division of *Minas Gerais* or offices that are under the supervision of *Serra Gaucha* division.

Description of the concepts represented in the models:

Party = Legal party, general concept that encompasses entities such as persons and organizations.

Accountability = Responsibility between two parties. The **“Responsible”** association identifies who is the responsible for the entity represented by the **“Commissioner”** association.

AccountabilityType = A type of responsibility registered. For instance: the accountability of a division over a sales office; the accountability of a person as manager of divisions, sales offices, or regions.

Rule = Rule that determines what type of **Party** can assume the positions of **Commissioner** and **Responsible**. The rules are previously implemented and identified by this relation. For example, an instance of **Rule** named “SalesOfficeToDivision” restricts the relation for the accountability of one Division relating to one commissioned sales office.

Task:

The current models have a limited number of organization classifications, demanding the creation of new classes to represent each type of “**Party**”. The goal is to refactor the models, allowing the dynamic registration of the existing types of parties. Another goal is to register how the types of Party affect the accountability types (**AccountabilityType**). Create new class and database models that includes all of the following changes:

1. Remove the subtypes of **Organization**. **Organization** is no longer abstract.
2. Create a class named **PartyType** that represents the types of **Party**. It will be persisted at a table named **PARTYTYPE**. Add a mandatory property named **description**, with String type and maximum length of 50.
3. Create a unidirectional association between **Party** and **PartyType**. One **Party** will have one-to-many “**PartyType**”. One **PartyType** will have zero/one-to-many “**Party**”. This association should be persisted at the database.
4. Remove the **Rule** class.
5. Create an association between **AccountabilityType** and **PartyType** classes, named **commissioners**. This relation should be of zero/one-to-many **AccountabilityType** to one-to-many **PartyType**. To persist this relation, define a table with the name **ACCTYPE_COMMISSIONERS**.
6. Create an association between classes **AccountabilityType** and **PartyType** named **responsibles**. This relation should be of zero/one-to-many **AccountabilityType** to one-to-many **PartyType**. To persist this relation, define a table named **ACCTYPE_RESPONSIBLES**.
7. Name the ends so that associations of steps 5 and 6 could be distinguished.
8. Perform the necessary changes on the database model reflecting the required changes at class model, with the following guidelines:
 - a) Remove unused tables.
 - b) Create necessary new tables for new classes of relationships.
 - c) Create, alter, or remove columns and primary keys (PKs) as a consequence of the changes.
 - d) Create, alter, or remove columns with integrity references (Fks).

Guideline:

- The primary keys of the original tables should not be changed. New tables should follow the pattern **ID_#TABLE_NAME#**.

- Join tables follows the standard name as a concatenation of the related tables. Their PKs are composites of the FKs with name **FK_ID_#FOREIGN_KEY#**. By standard, two instances (a1,b1) can only relate at most one time, but may relate to any other instances as well, for example (a1,b2; a5,b1).

- The models should contain the minimal necessary changes (and nothing more) to meet the requirements.

C.2.3 Expected Results

Figure C.8: Expected response for Accountability persistence model.

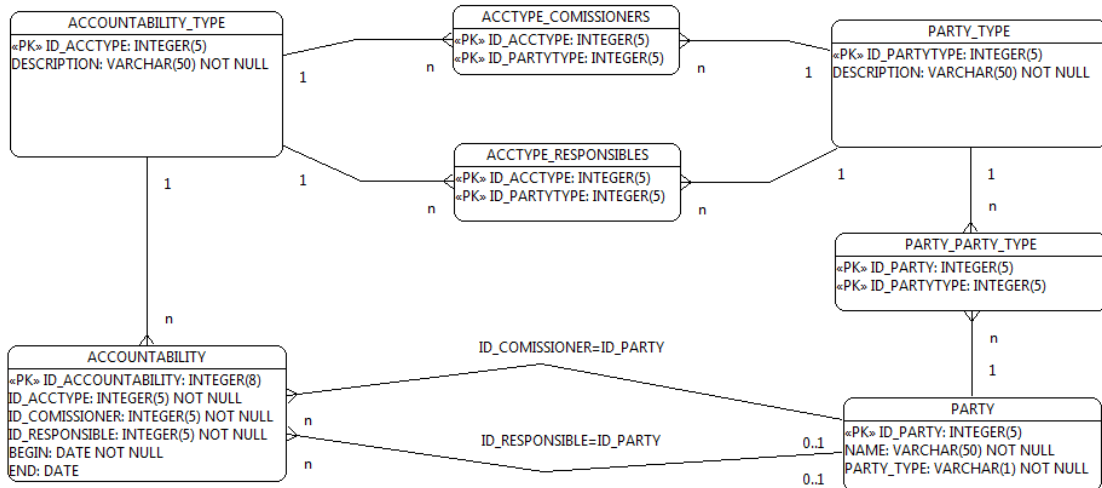


Figure C.9: Expected response for Accountability UML model.

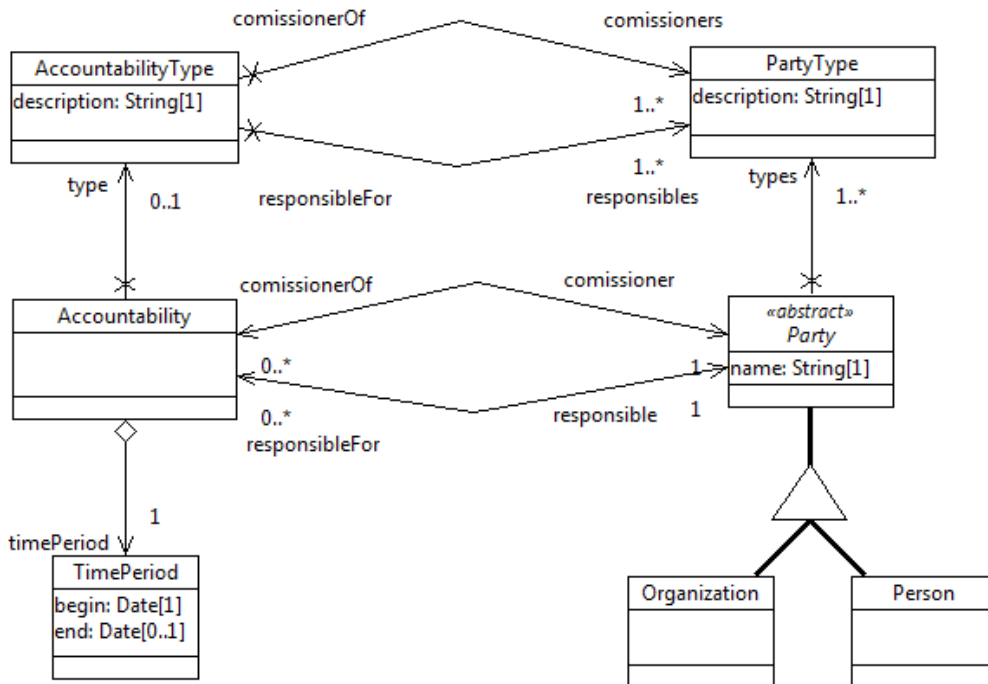
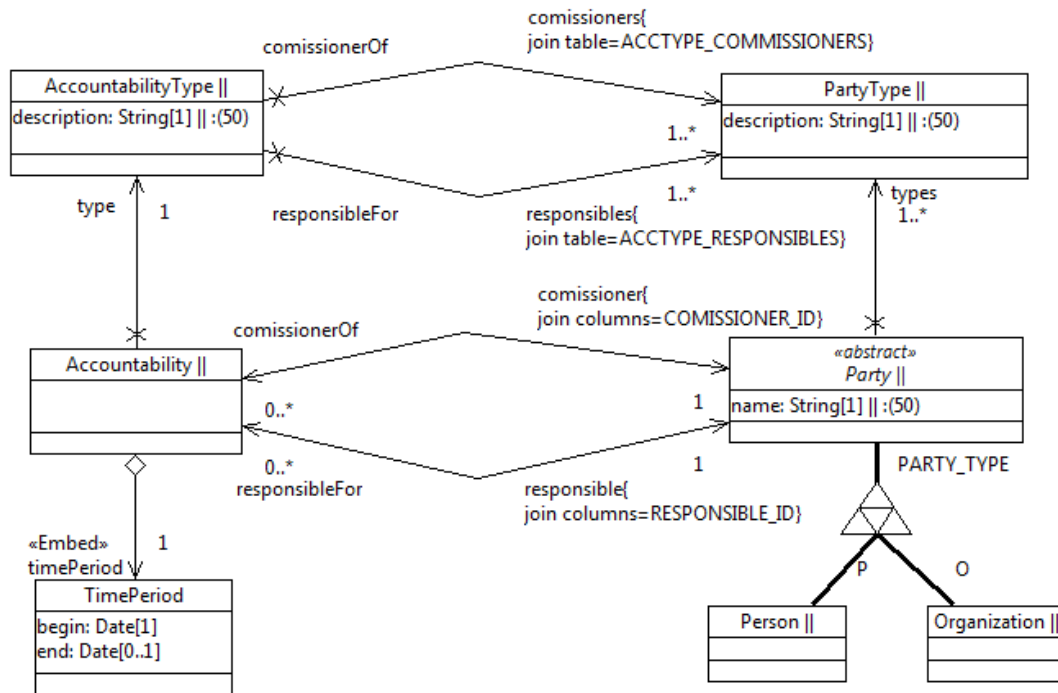


Figure C.10: Expected response for Accountability ENORM model.



C.3 Account Task

C.3.1 Instructions - Treatment A

Figure C.11: Initial Account persistence model.

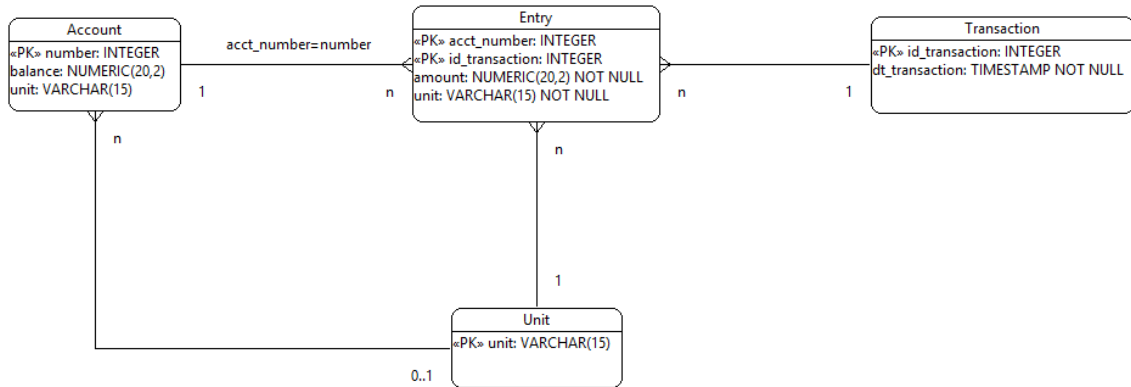
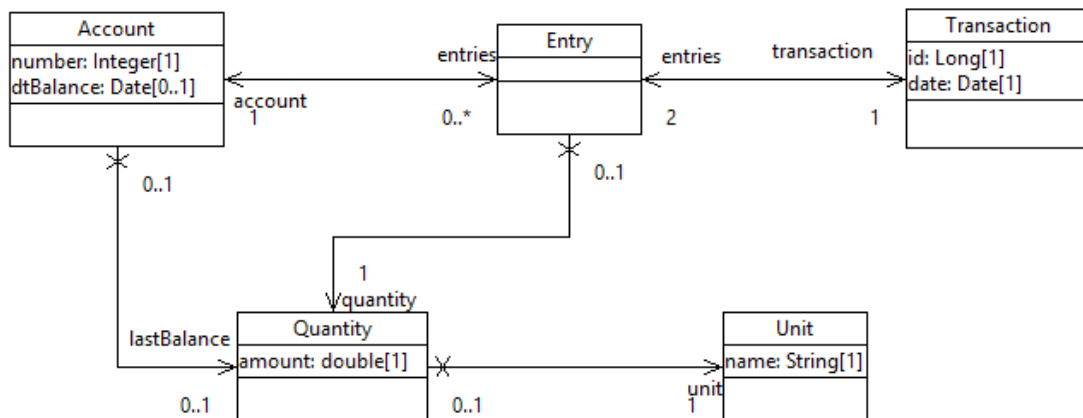


Figure C.12: Initial Account UML class model.



Accounts and transactions

The **Account** pattern models one *account* as a set of value entries (**Entry**).

Transaction) between two accounts register entries on the origin account, with positive value, and destination account, with negative value. For example, when transferring 50 Reais from account A to account B, it is created a transaction with two entries, one related to account A with value of R\$ -50.00, and another to account B with value of R\$ 50.00.

At the presented model, accounts and transactions can represent different types of things, with the type specified by the **Unit** associated to the registered **Quantity**. Continuing the example, the *entry* of A is represented with a **Quantity** instance with **amount** of -50 and **unit** R\$. The relation **quantity** of **Account** is the value of the last *balance*, IE, is calculated by the sum of all entries related to that account. The balance is calculated by demand, and stored by the account with the balance date.

Task:

1. Implement multiple leg transactions. For inventory accounts, it is common the scenery where the transaction involves several accounts. For example, when transferring to the central depot 100 sacks of tobacco, we can registry that 70 sacks are from the *Santa Cruz* unit account, and 30 from the *Venâncio Aires* unit account. This transaction relates to three entries, one positive related to the central deposit account, and two negatives related with *Santa Cruz* and *Venâncio Aires*. The sum of entries at one transaction must always be zero, and the minimal number of entries is 2.

2. Implement summary accounts.

One **SummaryAccount** is a virtual account that represents a set of other accounts. The “psychical” accounts are the detail accounts (**DetailAccount**) that relates with the entries of distinct transactions. The balance of summary account is the sum of balances of all their **component** accounts. The summary account can contain detail and other summary accounts.

Changes in the class model.

To implement the summary accounts, create two specializations of **Account**: **SummaryAccount** and **DetailAccount**. The relation **entries** between **Account** and **Entry** should now be *abstract* and *unidirectional* from **Account** to **Entry**. **DetailAccount** now implements **entries**, with a bidirectional relation between one **DetailAccount** and zero or more **entries**.

Zero/One **SummaryAccount** relates to zero or more elements of the general Account, by the bidirectional association *components*. **SummaryAccount** should implement the relation **entries**, but as a derivative union of *components.entries* (just draw the relation and mark the **entries** end as *read-only*). Finally, the **Account** class should now be abstract, and continues to relate to its quantity.

Changes in the mappings.

SummaryAccount and **DetailAccount** should use the vertical inheritance strategy, at which each class is stored in its own table with the same name. Define a discriminator

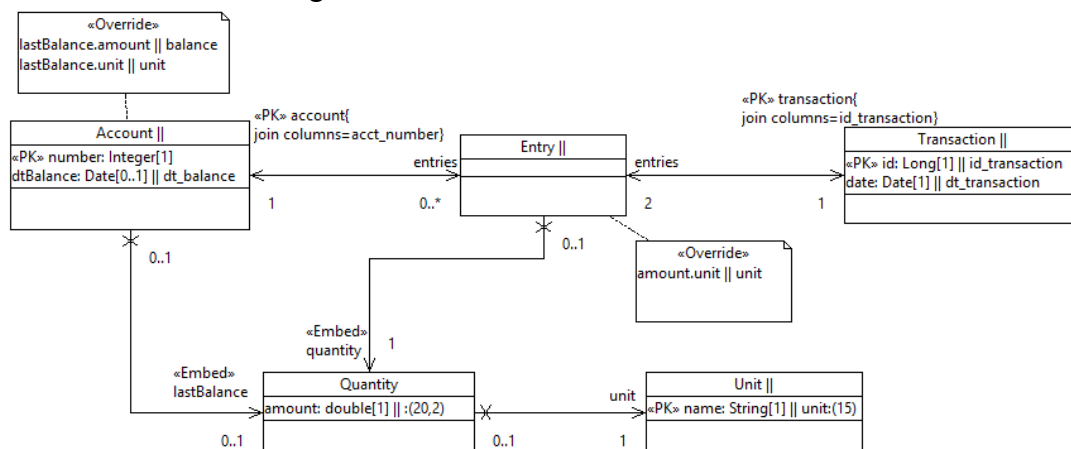
column at the specialization with name *type* and discriminator values S (**SummaryAccount**) and D (**DetailAccount**).

The persistent relation between **Entry** and **Account** should now be between **Entry** and **DetailAccount**. This relation is part of the primary key of **Entry**. The join column name (*acct_number*) should be moved from the relation **Entry** and **Account** to the new relation. The old relation is now abstract, and should not have any mappings.

Between the **SummaryAccount** and **Entry**, the **entries** relation must be marked as *not persistent* and *read-only*. The relation **components** between **Account** and **SummaryAccount** should be mapped with one join columns named **summary** at the **SummaryAccount** end.

C.3.2 Instructions - Treatment B

Figure C.13: Initial Account ENORM model.



Accounts and transactions

The **Account** pattern models one *account* as a set of value entries (**Entry**).

Transaction) between two accounts register entries on the origin account, with positive value, and destination account, with negative value. For example, when transferring 50 Reais from account A to account B, it is created a transaction with two entries, one related to account A with value of R\$ -50.00, and another to account B with value of R\$ 50.00.

At the presented model, accounts and transactions can represent different types of things, with the type specified by the **Unit** associated to the registered **Quantity**. Continuing the example, the *entry* of A is represented with a **Quantity** instance with **amount** of -50 and **unit** R\$. The relation **quantity** of **Account** is the value of the last *balance*, IE, is calculated by the sum of all entries related to that account. The balance is calculated by demand, and stored by the account with the balance date.

Task:

1. Implement multiple leg transactions. For inventory accounts, it is common the scenery where the transaction involves several accounts. For example, when transferring to the central depot 100 sacks of tobacco, we can registry that 70 sacks are from the *Santa Cruz* unit account, and 30 from the *Venâncio Aires* unit account. This transaction relates to three entries, one positive related to the central deposit account, and two

negatives related with *Santa Cruz* and *Venâncio Aires*. The sum of entries at one transaction must always be zero, and the minimal number of entries is 2.

2. Implement summary accounts.

One **SummaryAccount** is a virtual account that represents a set of other accounts. The “psychical” accounts are the detail accounts (**DetailAccount**) that relates with the entries of distinct transactions. The balance of summary account is the sum of balances of all their **component** accounts. The summary account can contain detail and other summary accounts.

Changes in the class model.

To implement the summary accounts, create two specializations of **Account**: **SummaryAccount** and **DetailAccount**. The relation **entries** between **Account** and **Entry** should now be *abstract* and *unidirectional* from **Account** to **Entry**. **DetailAccount** now implements **entries**, with a bidirectional relation between one **DetailAccount** and zero or more **entries**.

Zero/One **SummaryAccount** relates to zero or more elements of the general **Account**, by the bidirectional association *components*. **SummaryAccount** should implement the relation **entries**, but as a derivative union of *components.entries* (just draw the relation and mark the **entries** end as *read-only*). Finally, the **Account** class should now be abstract, and continues to relate to its quantity.

Changes in the database.

SummaryAccount and **DetailAccount** are persisted at its own tables, with a many(summary/detail)-to-zero/one **Account**. The primary key will be the account number, overriding the relationship to zero/one-to-one. Create a discriminator attribute at **Account**, named **type**, with char (1) type, and that can assume the values S or D.

The relationship **components** should be implemented as a many-to-zero/one relationship between **Account** and **SummaryAccount**, with a foreign key named **summary**.

The relationship **entries** should be moved to be between the table **DetailAccount** and **Entry**, and continues to define the primary key of **Entry**.

Do not forget to alter the class and database models.

C.3.3 Expected Results

Figure C.14: Expected response for Account persistence model.

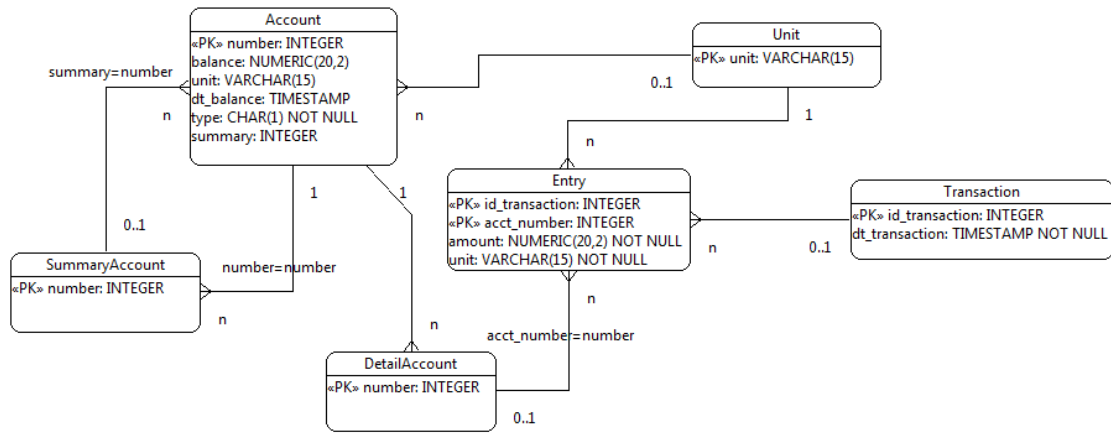


Figure C.15: Expected response for Account UML class model.

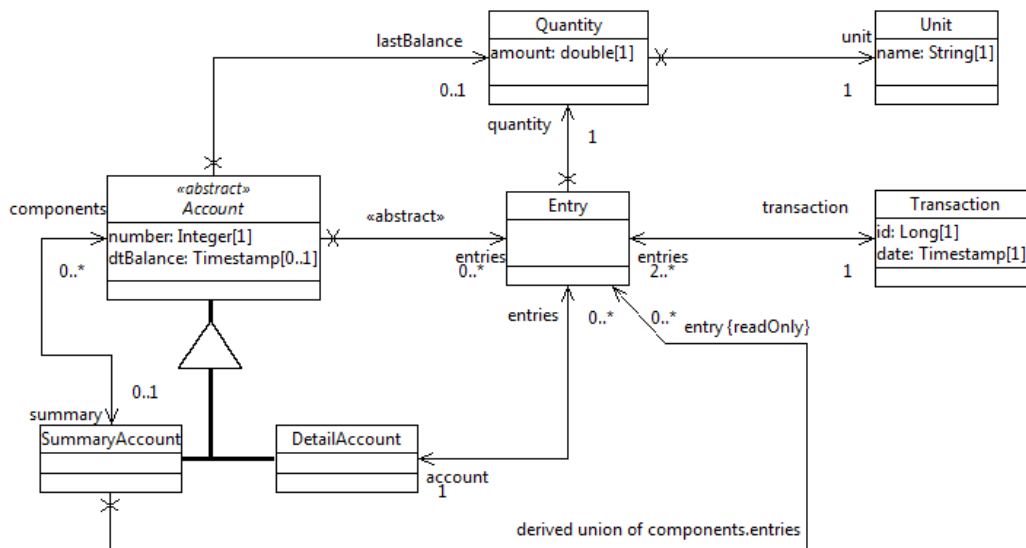
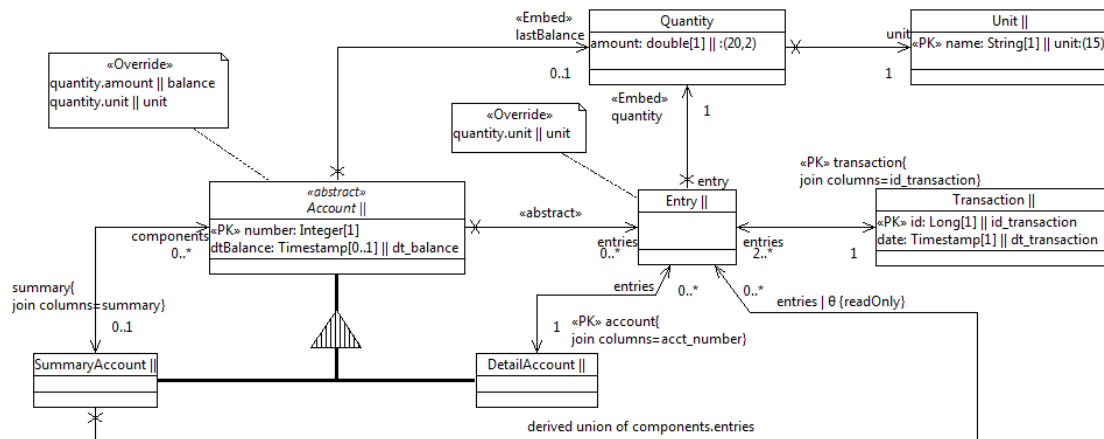


Figure C.16: Expected response for Account ENORM model.



C.4 Resource Allocation

C.4.1 Instructions - Treatment A

Resource allocation for action (tasks)

A legacy resource allocation system was integrated with another, object oriented framework based system, specializing the classes of the basic pattern to reflect the database structure at use. The goal of the system is to specify the necessary resources to perform **tasks** (actions), before and/or after its execution. The resources are allocated in a **generic** way, by referencing its **type**, or at a **specific** way, referencing the *consumed/allocated* elements.

Classes of the resource allocation pattern:

Action: Action or task. Lists a set of resource allocations necessary for the execution.

ResourceAllocation: a general resource allocation related to one **Action** and one **Quantity**.

Quantity: represents a pair (quantity, unit) specified for each resource allocation. All resource allocations have quantity/unit of resources.

Unit: Measure units for supplies and time.

GeneralAllocation: a *generic* resource allocation that specified only the *resource type* at use.

ResourceType: resource type, can be allocated in generic way.

AssetType: a *non consumable* resource type, such as *teacher, car* or *computer*.

ConsumableType: a *non consumable* resource type, such as *ink, paper*, or *gas*.

SpecificAllocation: a *specific* resource allocation: what teacher, what car, or what and where came from the ink and paper.

TemporalAllocation: *specific* allocation of *asset* resources, allowing to register the time of utilization (past or future).

Asset: a *non consumable* resource, reusable and measured by time of use. Ex. *Professor Gary Booch*.

ConsumableAllocation: allocation of *consumable resources*, specifying its *source*.

Holding: Identified the source for a specific *consumable resource type* (**ConsumableType**).

Figure C.17: Initial Resource allocation persistence model.

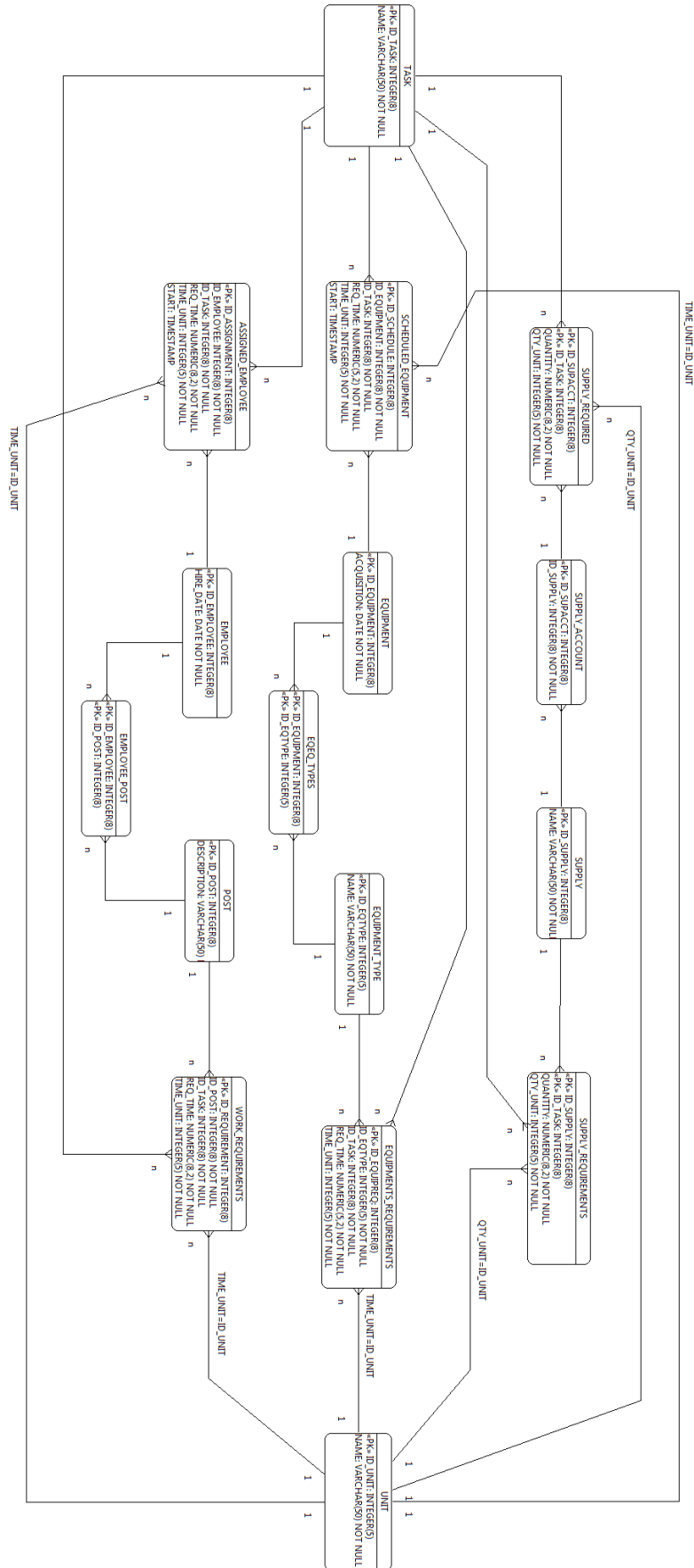
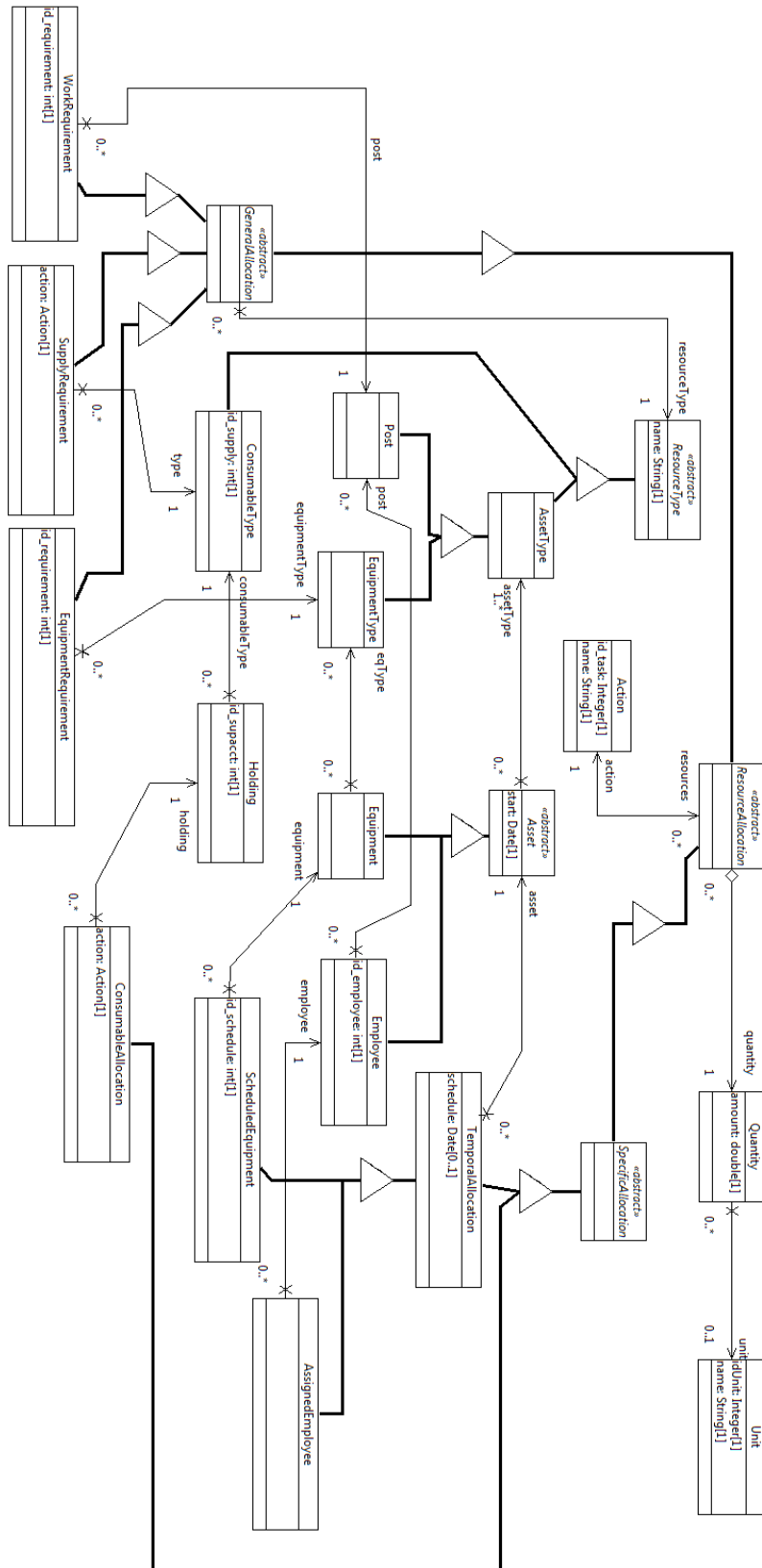


Figure C.18: Initial Resource allocation UML class model.



The following classes distinguishes non consumable resources that are human and psychical.

EquipmentType: types of equipments.

Equipment: available equipments. An equipment can be of more than one type of equipment.

Post: Posts or functions that can be exercised by employees to accomplish an action.

Employee: Employee or consultant, able to exercise one or more functions.

SupplyRequirement: generic allocation for supplies, specifying only the type of supply at use.

EquipmentRequirement: generic allocation of an equipment type.

WorkRequirement: generic allocation of one type of function requiring an employee.

ScheduledEquipment: allocation for one specific equipment, at an action.

AssignedEmployee: allocation for one specific employee, at an action.

Task

1. Change the specification of the **TemporalAllocation** class, in such way that it stores the resource's period of use, in the context of the task. For this, create a new class named **TimePeriod** with two attributes of the *timestamp* type (date+hour): **start** [1] and **end** [0..1]. **TemporalAllocation** will uni-directionally relate with one, and only one, **Time Period**.

TimePeriod should not be persisted at its own table, but have its attributes embedded at the persistence of **TemporalAllocation**. The **schedule** attribute should be erased.

2. Create a persistent class named **User**, with **login** as the primary key. Relate **ResourceAllocation** and **User** in such a way that each resource allocation always identify a responsible user (Horizontal inheritance implies at all relationships of the transient super class with persistent classes being implicit persisted).

C.4.2 Instructions - Treatment B

Resource allocation for action (tasks)

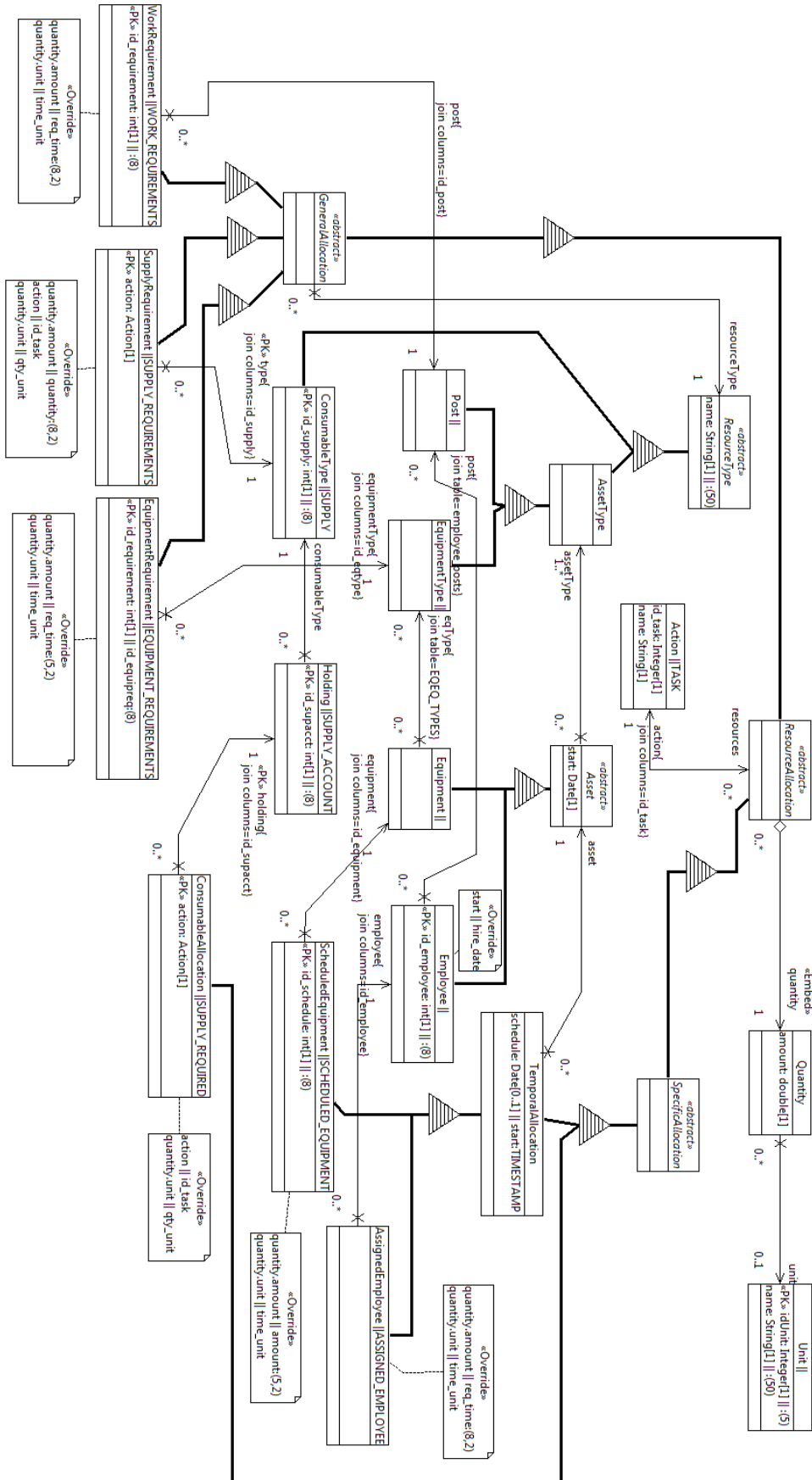
A legacy resource allocation system was integrated with another, object oriented framework based system, specializing the classes of the basic pattern to reflect the database structure at use. The goal of the system is to specify the necessary resources to perform **tasks** (actions), before and/or after its execution. The resources are allocated in a **generic** way, by referencing its **type**, or at a **specific** way, referencing the *consumed/allocated* elements.

Classes of the resource allocation pattern:

Action: Action or task. Lists a set of resource allocations necessary for the execution.

ResourceAllocation: a general resource allocation related to one **Action** and one **Quantity**.

Figure C.19: Initial Resource allocation ENORM model.



Quantity: represents a pair (quantity, unit) specified for each resource allocation. All resource allocations have quantity/unit of resources.

Unit: Measure units for supplies and time.

GeneralAllocation: a *generic* resource allocation that specified only the *resource type* at use.

ResourceType: resource type, can be allocated in generic way.

AssetType: a *non consumable* resource type, such as *teacher, car* or *computer*.

ConsumableType: a *non consumable* resource type, such as *ink, paper, or gas*.

SpecificAllocation: a *specific* resource allocation: what teacher, what car, or what and where came from the ink and paper.

TemporalAllocation: *specific* allocation of *asset* resources, allowing to register the time of utilization (past or future).

Asset: a *non consumable* resource, reusable and measured by time of use. Ex. *Professor Gary Booch*.

ConsumableAllocation: allocation of *consumable resources*, specifying its *source*.

Holding: Identified the source for a specific *consumable resource type* (**ConsumableType**).

[The following classes distinguishes non consumable resources that are human and psychical.](#)

EquipmentType: types of equipments.

Equipment: available equipments. An equipment can be of more than one type of equipment.

Post: Posts or functions that can be exercised by employees to accomplish an action.

Employee: Employee or consultant, able to exercise one or more functions.

SupplyRequirement: generic allocation for supplies, specifying only the type of supply at use.

EquipmentRequirement: generic allocation of an equipment type.

WorkRequirement: generic allocation of one type of function requiring an employee.

ScheduledEquipment: allocation for one specific equipment, at an action.

AssignedEmployee: allocation for one specific employee, at an action.

[The following tables were defined for persistence](#)

TASK: Register an action or task. Each action relates with various necessary resources to its execution.

SUPPLY_REQUIRED: Register the quantity of one supply relating to its source.

SUPPLY_REQUIREMENTS: Register the quantity of one supply without informing its source.

SUPPLY: Register the supply types (ink, gas, coffee ...)

SUPPLY_ACCOUNT: Identify a supply source (consumable).

EQUIPMENT_REQUIREMENTS: Register the equipment type, and requested time, for a task.

SCHEDULED_EQUIPMENT: Register an equipment allocated to one task, informing the equipment, how much time of use and an optional start date/time (START).

EQUIPMENT_TYPE: Types of available equipments, such as a car and a computer.

EQUIPMENT: The available equipments. One equipment is related to various types of equipments by the EQUIP_EQUIP_TYPES table.

ASSIGNED_EMPLOYEE: Register one employee allocated, indicating the dedication time, and an optional date/time (START).

WORK_REQUIREMENTS: Register one type of employee (labor) necessary to the task, and the required time.

POST: Post or function, exercised by employees to execute a task.

EMPLOYEE: Each employee is related to one of more functions, listed by the EMPLOYEE_POSTS table.

UNIT: Measure units for supplies and time. Examples: liters, hours, packages, days...

[Relationship between tables and classes:](#)

Action = TASK.

Unit = UNIT.

Holding = SUPPLY_ACCOUNT.

ConsumableType = SUPPLY.

ConsumableAllocation = SUPPLY_REQUIRED.

EquipmentType = EQUIPMENT_TYPE.

Equipment = EQUIPMENT.

Post = POST.

Employee = EMPLOYEE.

SupplyRequirement = SUPPLY_REQUIREMENTS.

EquipmentRequirement = EQUIPMENT_REQUIREMENTS.

WorkRequirement = WORK_REQUIREMENTS.

ScheduledEquipment = SCHEDULED_EQUIPMENT.

AssignedEmployee = ASSIGNED_EMPLOYEE.

Task

1. Change the specification of the **TemporalAllocation** class, in such way that it stores the resource's period of use, in the context of the task. For this, create a new class named **TimePeriod** with two attributes of the *timestamp* type (date+hour): **start** [1] and **end** [0..1]. **TemporalAllocation** will uni-directionally relate with one, and only one, **Time Period**.

TimePeriod should not be persisted at its own table, but have its attributes embedded at the persistence of **TemporalAllocation**. The **schedule** attribute should be erased.

Execute all necessary changes at the affected tables.

2. Create a persistent class named **User**, with **login** as the primary key. Relate **ResourceAllocation** and **User** in such a way that each resource allocation always identify a responsible user.

Execute the necessary changes in the database model in a way that all the distinct resource allocations always identify one User.

C.4.3 Expected Results

Figure C.20: Expected changes on the response for Account persistence model (only affected tables are depicted).

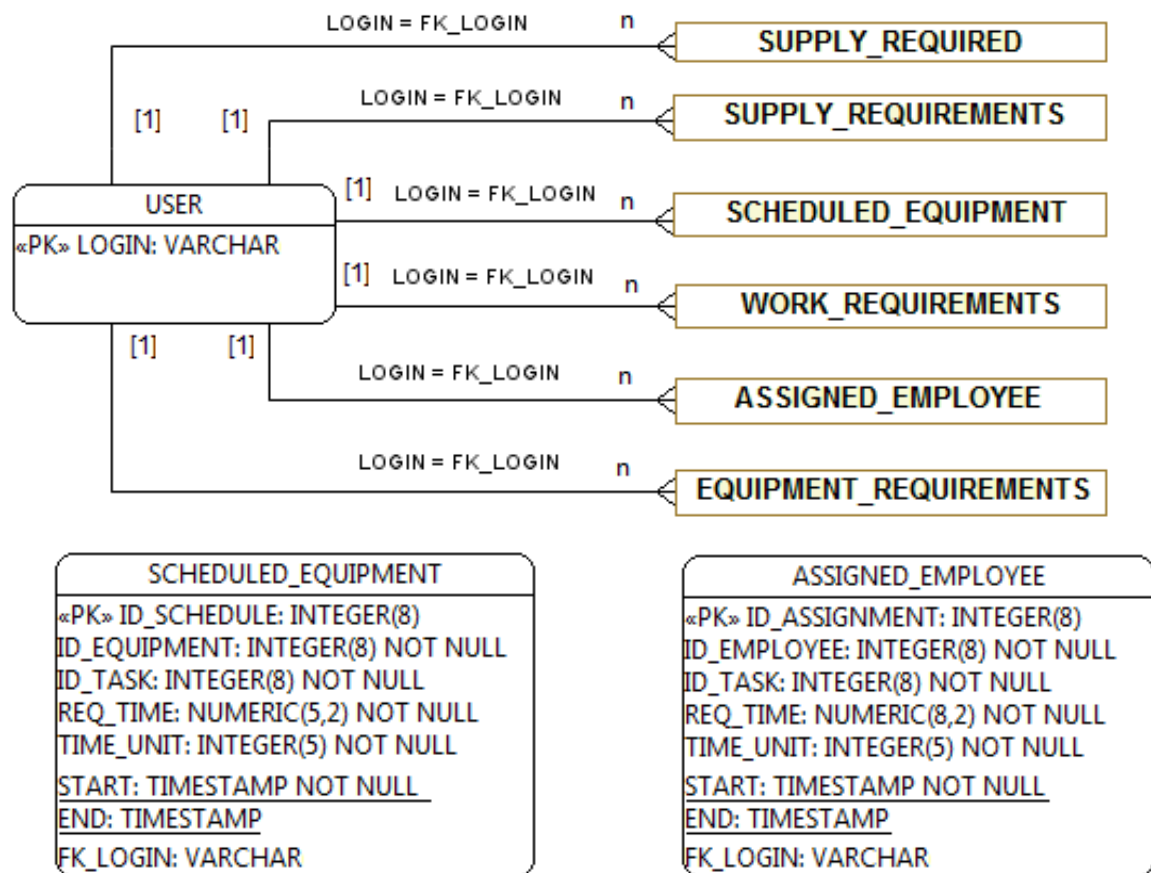


Figure C.21: Resource allocation expected UML response model.

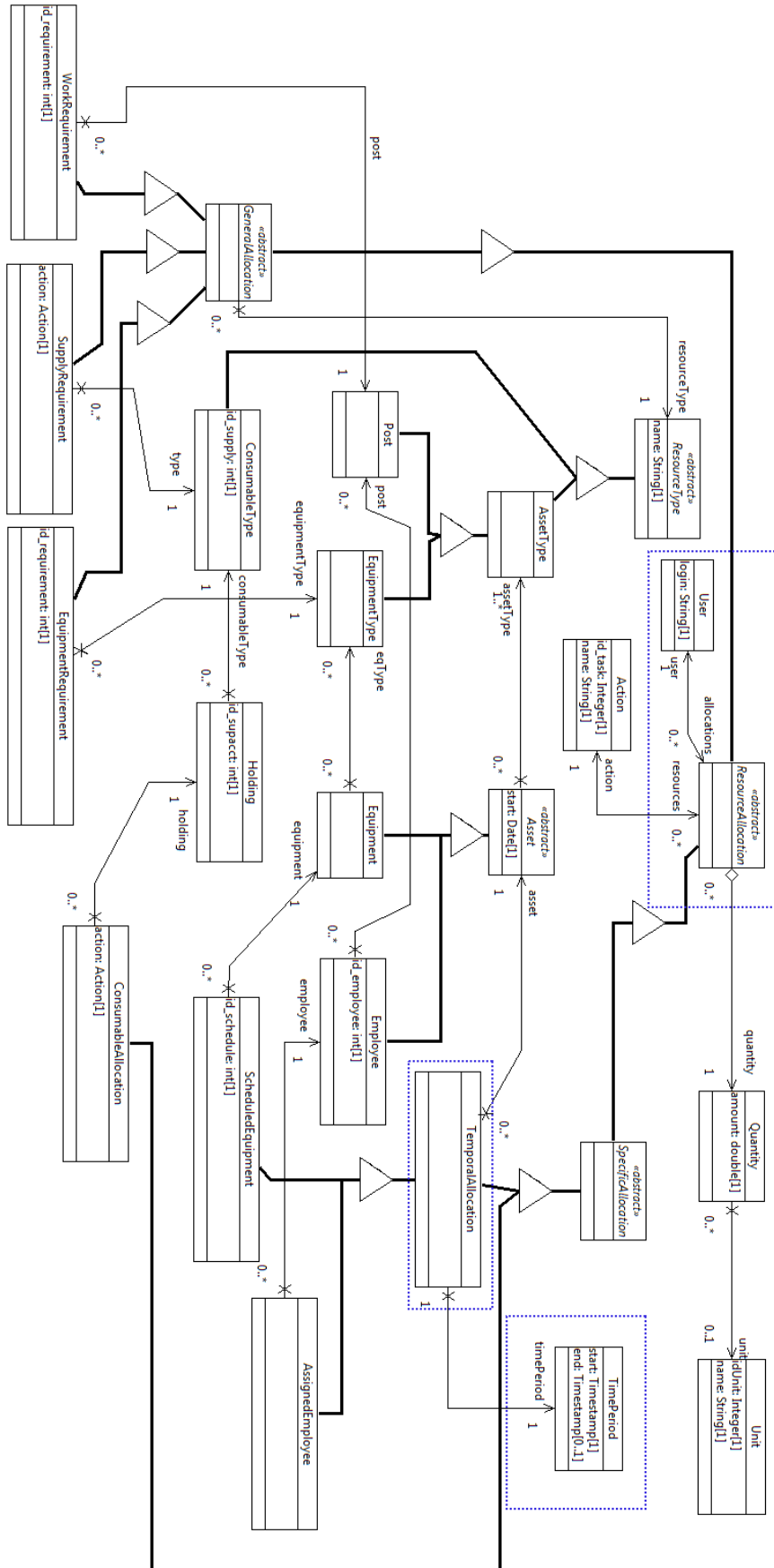
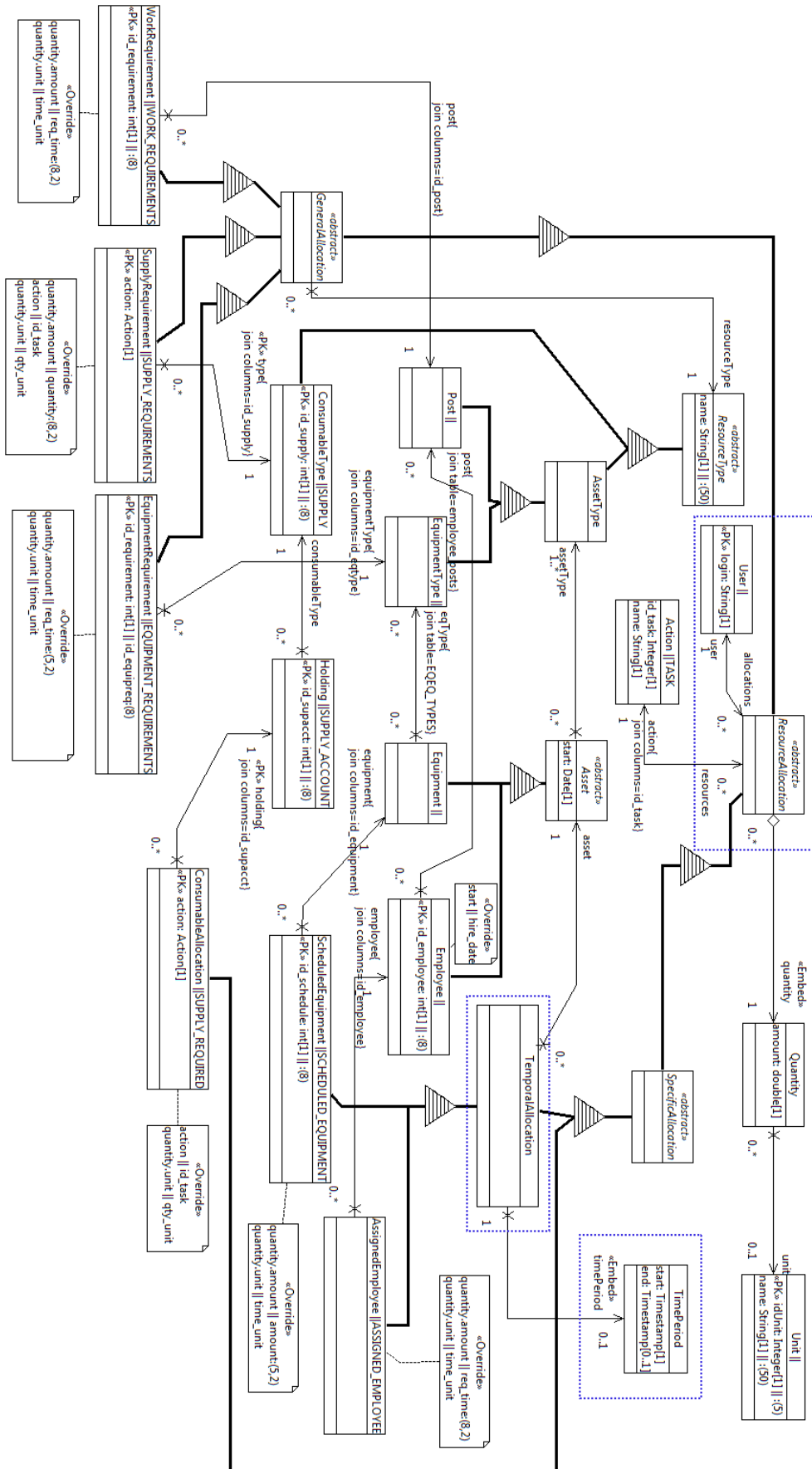


Figure C.22: Resource allocation expected ENORM response model.



APPENDIX D – TASKS (GROUP EXPERIMENT)

The task descriptions were translated from the original Portuguese.

D.1 Meeting Scheduling System – description

Using the *<models>*, design the following system:

- a) One meeting occurs at one place, at a date and time, with a maximum duration, participants (some participants have mandatory presence for the success of the meeting, while others may have an optional presence) and can have a defined agenda (optional).
- b) The participant that creates the meeting informs at what times he is available within the time period when the meeting can occur.
- c) He/She lists the participants, informing their emails.
- d) Each participants informs what times he/she is availble, within the time period when the meeting can occur.
- e) The creator participant can, from the responses of the other participants, close the meeting, choosing a time where all have availability, and choosing a place for the meeting.
- f) The place can be chosen from an internal list, or may be an external place informed by the creator participant.
- g) The place, if internal, must be free in the time period designated for the meeting.
- h) Each internal place has the following information: name, location, number of seats, if have projector, if have air-conditioning, if have sound system, if have computer, if have teleconference system, and how many computers are available. The only necessary information about external places are a name and an optional description (examples: pub, parking lot A, campus porch).
- i) For each appointed meeting, it should be possible to query the system about the following:
 1. If all participants entered their timetable.
 2. What are the common time at which all, or the majority, agree – There may be zero or more times.
 3. What are the common times at which all obligatory participant can meet.
 4. What paces are available at some specific time, with parameter such as: it has air-conditioning? projector? Etc...

D.2 Appointment Book System – description

Using the <models>, design the following system:

- a) The appointment book allows the user to register appointments. An appointment has a name, an optional description, start, end (if not punctual) and may repeat at some frequency.
- b) Regarding the duration, with precision of minutes, there are the following types of appointment:
 - 1) Punctual appointments last just an instant and are like reminders. Example: turn off the oven.
 - 2) Normal appointments last some time period no longer than a day.
 - 3) Full day appointments last for one or more days. However, they can start at any time in one day, and end at any time in another day. Examples: vacations, conferences, academic week.
- c) It is possible to create appointments that contain another appointments, as long as the it is inside the time period of the parent appointment. Example: courses in the academic week.
- d) It is possible to create appointments that repeat. The frequency can be daily, weekly, monthly or annual. Examples: birthdays, regular meetings, and classes.
- e) The cancellation of an appointment that repeats should not erase its history, only the future repetitions.
- f) An appointment can be created as public or private. If it is private, the user can inform for what groups of contacts the appointment is visible. It is the user himself that creates these groups, informing the group name and the member's emails.
- g) It should be possible to query the system about:
 - 1) What are the appointments at a given time period?
 - 2) If some user can see the appointment.
 - 3) Given an appointment, what other appointments occur inside it.
 - 4) When, and how many times, a canceled appointment happened.

D.3 Room's Booking Database – description

- 1) Create the database model necessary to control the following booking of rooms:
 - a) The goal of the model is to store the booking of rooms for users. Each user, identified by his name and email, chooses a room from a list and enters the start and end times for a reservation.
 - b) Each reservation must have a name and can have an optional description.
 - c) It must be possible to query, by the reservations, what rooms were/will be occupied at some time period.
 - d) The rooms are stored at the system, maintained by an administrator.
 - e) Each room has the following information: name, capacity, and description.

2) Describe in the following space, using your language or SQL, the following queries for your model:

1. What rooms are occupied at some time period.
2. What rooms with some capacity are occupied at some time period.
3. If there are conflicting reservations.

D.4 Database Integration Stage Instructions

The model create by the DA as a simple booking system will be discarded and replaced by the new systems. Based upon the models that you created for the meetings and appointments systems, the DA and other participants must reach a common database model that meets the following prioritized list:

1. Allows both systems to attend their requirements.
2. Cause the least possible impact at the starting model for the DA (Consider impact as exclusion, modification, or creation of new tables, attributes, and relationships).

The DA will serve as mediator of the integration model, responsible to keep the model simple to migrate the old data. The other participants should guarantee that the integration model can be adapted to his own system, demanding changes in the data model when necessary to fulfill the requirements.

The participants (responsible for the meeting and appointments systems) can (and should) make the necessary modifications at their application models (using assigned the notation), adapting the system to the new integrated database system.

D.5 Auto-evaluation survey

Answer the following questionnaire trying to find a consensus at your group. Answer the questions at following THE ORDER, in a cumulative way (it should take in account all that happened before).

Considering the following initial data for the meeting and appointment systems:

João, Maria, José, Rafael and Ana are system users.

The meeting places are:

Sala verde, with capacity for six people, air-conditioning, one computer, projector, and teleconference.

Sala azul, with capacity for four people, air-conditioning, and projector.

Auditório, with capacity for eighty people, air-conditioning, projector, and sound system.

Laboratório, with capacity for twenty five people, air-conditioning, twenty computers, and projector.

Shifts defined at this survey: Morning 8-12h / Afternoon: 13-17h / Night 18-22h

João is a teacher, and teaches (every) *Monday, Wednesday, and Friday* in the *morning* and at *night*, creating those appointments as repetitive at the system.

Maria is a teacher and will teach this week only on *Monday*, during all day. Between *Tuesday* and *Friday* of this week, she will be involved with the organization of a conference at his university (this is an appointment with various days).

José is the director of a TI company, working between 9:00 and 19:00 every business day of the week.

Rafael is a student, and employee of *José*, working between 9:00 and 18:00 hours.

Ana is a visiting researcher, participates in the conference (between *Tuesday* and *Friday*), and is busy at *Monday*.

Considering that both systems work in an independent way, only sharing the data by the database, but not necessarily interfering with the data of the other system. In other words, one meeting confirmed does not *necessarily* creates an appointment, and the times in the appointment does not *necessarily* are used for sugesting times for meetings. It is up to you to decide if (and how) the system were integrated, and stick to this decision to the entire survey.

 1. Is it possible to represent the above agendas at the appointments system? Describe how these objects would be instantiated by the system, and stored in the database, to justify your answer.

Yes No

2. *João* wants to invite *José* and *Ana* to a lecture at his night class, between 19:00 and 22:00 hours, using the meeting system. The lecture will last *one hour*. Each participant informs the times, and *Ana* chooses *Wednesday* and *Friday* nights (despite her conference appointment, it would fit as a child appointment).

a) Describe how these objects would be instantiated by the system, and stored in the database. It is possible to select a time (by query) compatible for the three, by the system?

Yes No

b) Check if the meeting times stored at the database interfere in the query of the times stored at the appointment system in such a way that it would return incorrect results.

Don't interfere Interfere

c) When querying for places with capacity for more than forty people, only the *Auditório* is selected?

Yes No

3. *Ana* wants to create a child appointment at her agenda for the lecture at the *João*'s class. Describe how these objects would be instantiated by the system, and stored in the database.

a) Check if these new objects/rows are correctly stored.

Yes No

b) Considering all the data so far, it is possible to correctly query the appointments of *Ana*? Describe how it would be.

Yes No

4. *João* creates a *punctual appointment* (child) inside his *Wednesday's class* for remembering to take his students to the correct lecturing room. Describe this appointment as objects in the system and rows in the database. This appointment can be properly created?

Yes No

5. *Rafael* will present a paper *Friday morning*, and receives clearance from his boss. *Rafael* try to reserve a half hour meeting between 8:00 e 12:00 hours of Friday, inviting *Maria* (obligatory, his adviser), and optionally *João*, *Ana* and *José*. *José* accepts with the condition of being a *teleconference*. *João* is not available Friday morning. *Ana* has appointments between 10:00 and 12:00 hours, and a meeting reserving the green room at this period. *Maria* has an appointment between 8:00 and 10:00 horas. Describe how these objects would be instantiated by the system, and stored in the database. Each participant will inform the times according to his appointments.

a) Check if these new objects/rows are correctly represented.

Yes No

b) When querying for the possible meeting times, check if the system responds correctly: for all participants, no time is selected. Excluding *João*, neither time is possible. Excluding *Maria*, it would be possible between 8 e 10 at the *Sala Verde* (the only one with teleconference), but her presence is obligatory, this is not an option. Between 10:00 and 12:00, all obligatory participants can attend, but not at the *Sala Verde*, which has teleconference (the system will probably not know that *José* cannot be present in a room without teleconference, but this does not configures an error in the system because was not in the requirements).

Yes No

6. *Maria* wants to create a meeting between Tuesday and Friday to talk with *João* about the conference. The two enter their possible times, and *Maria* chooses as an **external place** a coffee shop in the neighborhood. Describe the objects, rows, and check if they are correctly represented by the models.

Yes No

7. *Rafael* creates a private appointment that repeats every *Wednesday night*, football with friends. Only a few friends and colleagues, such as *José*, can see the appointment. Describe the objects and database rows, and check if *Jose* can see the description of the appointment, but *Maria* can't.

Yes No

8. At the end of semester, *João* changes his classes, and no longer minister classes at *Wednesdays*. When canceling his repeating appointment of *Wednesday*, describe how would be the objecys and rows that store the history of appointments.

a) It is still possible to know that *João* had classes *Wednesday*?

Yes No

b) It is still possible to know that *João* had a *punctual appointment* in that Wednesday with a lecture?

() Yes () No

D.6 Results of the Group Experiment (G)

Table 1: Missed goals according to the questionnaire.

Group	1	2.a	2.b	2.c	3.a	3.b	4.a	5.a	5.b	6	7	8.a	8.b	Sum
A	1	2	1	0	1	2	1	2	2	0	1	2	2	17
B	1	1	0	0	1	2	1	3	1	2	2	1	1	16

APPENDIX E – EXTENSIONS TO ACTIVE RECORD

The following Ruby class emulates the inheritance and joined sources by capturing the method missing listener and forwarding to a related active record.

```
# MixinMod is a workaround to both the absence of JOINED inheritance
and secondary tables.
# This workaround includes another ActiveRecord related with
belongs_to or has_one as a Mixin.
# This mixin will allow access to the properties of the related
activerecord, simulating inheritance
# To make use of this module do as follows:
# 1-declare the relationship to the referenced ActiveRecord (the
parent class)
# 2-include MixinMod, BEFORE the relationship
# 3-create a MixInDesc descriptor:
# MixInDesc.new(<name of your class>,<relationship
variable>,<primary_key or NIL>)
# If the primary key is informed, the MIXIN will try to save the
related object when creating the object,
# and use its primary key to initialize the descendent object. For
instance, if Worker descends from Person
# and you inform pk as "person_id", when you create a Worker, the
MixIn will create a Person, save, and
# copy the person_id value from Person to Worker
module MixinMod
  class MixInDesc
    @@mixins = {}
    attr_reader :attr, :key
    def initialize(clazz, attr, key)
      @attr, @key = attr, key
      @@mixins[clazz] = self
    end
    # getter
    def self.mixins
      @@mixins
    end
  end
  def method_missing(sym, *args, &block)
    eval(MixinDesc.mixins[self.class].attr).__send__(sym, *args,
&block)
  end
  def initialize (attr = nil, options = {})
    super( attr, options )
    mixin = MixInDesc.mixins[self.class]
    if (eval(mixin.attr)==nil)
      eval("build_"+mixin.attr) #build_<attributename>
      if (mixin.key!=nil)
        eval(mixin.attr).save
      end
    end
  end
end
```

```

        eval("self."+mixin.key+" = "+mixin.attr+"."+mixin.key)
      end
    end
  end
end

```

The next code exemplifies the use within the *Accounting* example. The *Account* class is the join of *ActBrief* and *Account* tables by instantiating *MixinDesc* and including the *MixinMod*. The specializations of *Account* also declare their relationship to *Account* with the same module, but informing the PK field.

```

class ActBrief < ActiveRecord::Base
  belongs_to :account, :class_name =>"Account", :foreign_key =>
'number'#, :inverse_of=>:actbrief
  composed_of :balance, :class_name => 'Quantity',
    :mapping => [ ["value", "amount"], ["unit", "unit"] ],
    :constructor => Proc.new { |amount, unit| (unit==nil) ? nil :
Quantity.new(amount,Unit.find(unit)) }
end
# There is no vertical inheritance on active record. Emulating
inheritance with a one-to-one relationship and implementing as Adapter
class Account < ActiveRecord::Base
  include MixinMod # Emulating secondary table act_brief
  @@mixin=MixinDesc.new(Account,"actbrief",nil)
  self.primary_key = "number"
  alias_attribute :dtBalance, :dt_calc
  belongs_to :impl, :polymorphic => true, :foreign_key => 'number',
:dependent => :destroy
  has_one :actbrief, :class_name =>"ActBrief", :foreign_key =>
'number', :dependent => :destroy, :inverse_of=>:account
  belongs_to :summary, :class_name =>"SummaryAccount", :foreign_key =>
'summary', :inverse_of=>:components
  # Operations -----
  def calc_balance
    impl.calc_balance
  end
  def entries
    impl.entries
  end
end
class DetailAccount < ActiveRecord::Base
  include MixinMod # Mixin that implements an adapter
  @@mixin=MixinDesc.new(DetailAccount,"account","number")
  self.primary_key = "number";
  has_one :account, :as => :impl, :autosave => true,:foreign_key =>
'number'
  has_many :entries, :foreign_key =>:acct_number,
:inverse_of=>:account

  # Operations -----
  def calc_balance
    if (balance==nil || balance.unit==nil)
      firstEntry = entries.find(:first)
      if (firstEntry==nil || firstEntry.quantity==nil)
        return nil
      end
      u = firstEntry.quantity.unit
      balance=Quantity.new(0.0,u)
    end
    account.dt_calc = DateTime.now
    am = entries.sum(:amount)
  end
end

```

```

        account.balance = Quantity.new(am,balance.unit)
        return self.balance
    end
end
class SummaryAccount < ActiveRecord::Base
  include MixinMod # Mixin that implements an adapter
  @@mixin=MixinDesc.new(SummaryAccount,"account","number")
  self.primary_key = "number"
  has_one :account, :as => :impl, :autosave => true, :foreign_key =>
'number'
  has_and_belongs_to_many :components, :class_name =>
'Account', :join_table=>"act_comps"
  # Operations -----
  def entries
    ret = []
    components.each{|c|ret+=c.impl.entries}
    return ret
  end
  def calc_balance
    qty = nil
    sum = 0
    for acct in components
      q = acct.calc_balance
      if (qty==nil)
        qty = q
      end
      if (q!=nil)
        sum = sum + q.amount
      end
    end
    if (qty!=nil)
      qty = Quantity.new(sum,qty.unit)
      balance = qty
      dt_calc = DateTime.now
    end
    return qty
  end
end
end

```

APPENDIX F – RELATED PUBLICATIONS

Papers covering Chapter 2, surveying patterns and ORM frameworks:

TORRES, A.; GALANTE, R.; PIMENTA, M. A synergistic model-driven approach for persistence modeling with UML. **Journal of Systems and Software**, v. 84, n. 6, p. 942–957. New York: Elsevier Science Inc. , 2011.

TORRES, A.; GALANTE, R.; PIMENTA, M; MARTINS, A. Relations are from Mars, objects are from Venus: a survey on O-R mapping patterns and frameworks. **Computing Surveys**, submitted, partially accepted at April 2012, awaiting final decision.

Papers covering Chapter 3, presenting the ENORM notation:

TORRES, A.; GALANTE, R.; PIMENTA, M. ENORM: An Essential Notation for Object-Relational Mapping. **SIGMOD Record**, accepted with changes at December 2013, awaiting final decision.

Papers covering Chapter 5, focused at the empirical evaluation:

TORRES, A.; GALANTE, R.; PIMENTA, M. Comparing ENORM, an Object-Relational Mapping notation, with separated relational and class modeling: an experimental study. **Empirical Software Engineering**, submitted by January, 2014.

ANNEX A – EXPERIMENTAL REPORT – EXPERIMENT I1

Author: NAE/UFRGS – *Núcleo de Acessoria Estatística.*

SOFTWARE

Todas as análises foram executadas utilizando-se o software SAS Enterprise Guide 4.2.

METODOLOGIA

Para a comparação entre os métodos A e B foi utilizada a análise de variância (ANOVA) para dados coletados num delineamento crossover, que possibilita, adicionalmente, verificar o efeito residual na seqüência (ou ordem) das atividades.

RESULTADOS

ACCOUNTABILITY

Type 3 Tests of Fixed Effects				
Effect	Num DF	Den DF	F Value	Pr > F
Método	1	54	55.35	<.0001
Sequencia	1	67	0.66	0.4201
Período	1	54	0.00	0.9706

Least Squares Means						
Effect	Método	Estimate	Standard Error	DF	t Value	Pr > t
Método	A	11.1839	0.5617	54	19.91	<.0001
Método	B	6.4634	0.5424	54	11.92	<.0001

Verifica-se diferença significativa entre os métodos A e B ($F=55,35$; $p<0,0001$), apresentando o método B um menor número médio de erros. As evidências amostrais não comprovam a presença de efeito residual, dada a não significância estatística para o efeito de seqüência ($F=0,66$, $p=0,4201$), indicando que não há evidência de aprendizado de uma atividade para a outra.

ACCOUNT

Type 3 Tests of Fixed Effects				
Effect	Num DF	Den DF	F Value	Pr > F
Método	1	35	8.77	0.0055
Período	1	35	6.68	0.0141
Sequencia	1	35	1.14	0.2923

Least Squares Means						
Effect	Método	Estimate	Standard Error	DF	t Value	Pr > t
Método	A	13.0868	0.7732	35	16.93	<.0001
Método	B	10.7309	0.7732	35	13.88	<.0001

Verifica-se diferença significativa entre os métodos A e B ($F=8,77$; $p=0,0055$), apresentando o método B um menor número médio de erros. As evidências amostrais não comprovam a presença de efeito residual, dada a não significância estatística para o efeito de seqüência ($F=1,14$, $p=0,2923$), indicando que não há evidência de aprendizado de uma atividade para a outra.

ADDRBOOK

Type 3 Tests of Fixed Effects				
Effect	Num DF	Den DF	F Value	Pr > F
Método	1	62	35.01	<.0001
Sequencia	1	67	0.34	0.5636
Período	1	62	4.17	0.0455

Least Squares Means						
Effect	Método	Estimate	Standard Error	DF	t Value	Pr > t
Método	A	16.6384	0.7775	62	21.40	<.0001
Método	B	11.6054	0.7911	62	14.67	<.0001

Verifica-se diferença significativa entre os métodos A e B ($F=35,01$; $p<0,0001$), apresentando o método B um menor número médio de erros. As evidências amostrais não comprovam a presença de efeito residual, dada a não significância estatística para o efeito de seqüência ($F=0,34$; $p=0,5636$), indicando que não há evidência de aprendizado de uma atividade para a outra.

RESOURCE

Type 3 Tests of Fixed Effects				
Effect	Num DF	Den DF	F Value	Pr > F
Método	1	33	101.79	<.0001
Sequencia	1	35	1.36	0.2512
Período	1	33	2.96	0.0945

Least Squares Means						
Effect	Método	Estimate	Standard Error	DF	t Value	Pr > t
Método	A	7.4587	0.3871	33	19.27	<.0001
Método	B	2.4159	0.3888	33	6.21	<.0001

Verifica-se diferença significativa entre os métodos A e B ($F=101,79$; $p<0,0001$), apresentando o método B um menor número médio de erros. As evidências amostrais não comprovam a presença de efeito residual, dada a não significância estatística para o efeito de seqüência ($F=1,36$; $p=0,2512$), indicando que não há evidência de aprendizado de uma atividade para a outra.

ANNEX B – EXPERIMENTAL REPORT – EXPERIMENT I2

Universidade Federal do Rio Grande do Sul
Instituto de Matemática
Departamento de Estatística



Relatório de Assessoria Estatística

“Análise de Variância para Medidas Repetidas das Variáveis Respostas “Erros”
e “Tempo”, com “Grupo” e “Tarefa” como Fatores.

Pesquisador(a):

Alexandre Torres

Equipe de Assessoria:

Profa. Jandyra Fachel e Gilberto P. Mesquita

Porto Alegre, 12 de novembro de 2013.

1) **Análise Estatística: Análise de Variância para Medidas Repetidas da Variável Resposta "Erros".** Como os dados apresentaram heterogeneidade de variâncias foi necessário transformar os dados. A transformação utilizada foi pelo Método dos Mínimos Quadrados Ponderados, cuja variância dos Tratamentos entrou na composição da ponderação.

Anova MR para Erros - Dados Ponderados

The GLM Procedure

Class Level Information		
Class	Levels	Values
Grupo	2	A B
Rep	23	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
Tarefa	4	Account Accountability AddrBook Resources

Number of Observations Read	176
Number of Observations Used	157

The GLM Procedure

Dependent Variable: Erros Erros

Weight: ivar_erro ivar_erro

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	49	171.8137472	3.5064030	7.11	<.0001
Error	107	52.7425019	0.4929206		
Corrected Total	156	224.5562491			

R-Square	Coeff Var	Root MSE	Erros Mean
0.765126	10.97749	0.702083	6.395662

Source	DF	Type III SS	Mean Square	F Value	Pr > F
Grupo	1	13.68312835	13.68312835	27.76	<.0001
Rep(Grupo)	42	96.25749814	2.29184519		
Tarefa	3	32.61088567	10.87029522	22.05	<.0001
Grupo*Tarefa	3	4.60819414	1.53606471	3.12	0.0292

Tests of Hypotheses Using the Type III MS for Rep(Grupo) as an Error Term					
Source	DF	Type III SS	Mean Square	F Value	Pr > F
Grupo	1	13.68312835	13.68312835	5.97	0.0188

Através da Análise de Variância acima, podemos concluir que o efeito da Interação Grupo*Tarefa é significativo, $p = 0,0292$, em relação à Variável Resposta Erros.

2) Teste de Comparações Múltiplas de Tukey-Kramer para Comparar Médias da Interação Grupo*Tarefa.

Anova MR para Erros - Dados Ponderados

The GLM Procedure

Least Squares Means

Adjustment for Multiple Comparisons: Tukey-Kramer

Grupo	Tarefa	Erros LSMEAN	LSMEAN Number
A	Account	14.2563014	1
A	Accountability	9.6666667	2
A	AddrBook	7.8095238	3

Grupo	Tarefa	Erros LSMEAN	LSMEAN Number
A	Resources	8.9046605	4
B	Account	11.3597870	5
B	Accountability	5.6521739	6
B	AddrBook	7.0869565	7
B	Resources	3.7468807	8

Least Squares Means for effect Grupo*Tarefa Pr > t for H0: LSMean(i)=LSMean(j) Dependent Variable: Erros								
i/j	1	2	3	4	5	6	7	8
1		0.0466	0.0014	0.0007	0.4986	<.0001	<.0001	<.0001
2	0.0466		0.9010	0.9980	0.9155	0.0109	0.3769	<.0001
3	0.0014	0.9010		0.9877	0.2136	0.6232	0.9992	0.0163
4	0.0007	0.9980	0.9877		0.4062	0.0058	0.5458	<.0001
5	0.4986	0.9155	0.2136	0.4062		<.0001	0.0097	<.0001
6	<.0001	0.0109	0.6232	0.0058	<.0001		0.7322	0.1626
7	<.0001	0.3769	0.9992	0.5458	0.0097	0.7322		0.0030
8	<.0001	<.0001	0.0163	<.0001	<.0001	0.1626	0.0030	

Através do Teste de Comparações Múltiplas de Tukey-Kramer, podemos concluir, por exemplo, que dentro da Tarefa “Accountability”, o número médio de erros do Grupo A difere significativamente do número médio de erros do Grupo B, $p= 0,0109$. Por outro lado, podemos concluir, por exemplo, que dentro do Grupo A, o número médio de erros da Tarefa “Account” difere significativamente do número médio de erros das demais Tarefas (“Accountability”, “AddrBook” e “Resources”), $p= 0,0466$, $p= 0,0014$ e $p= 0,0007$, respectivamente.

3) Estatísticas Descritivas: Média e Desvio Padrão da Variável Resposta “Erros”. Dados Ponderados.

Anova MR para Erros - Dados Ponderados

The GLM Procedure

Level of Grupo	N	Sum of Weights	Erros	
			Mean	Std Dev
A	75	3.2428303131	8.74054697	1.05523665
B	82	6.1630818149	5.16185376	1.19123774

Level of Tarefa	N	Sum of Weights	Erros	
			Mean	Std Dev
Account	37	0.9997840367	11.9633064	1.00132736
Accountability	44	2.1517496097	6.6332952	1.06097622
AddrBook	44	1.4748550267	7.2992554	0.99018169
Resources	32	4.779523455	4.8452071	1.32092096

Level of Grupo	Level of Tarefa	N	Sum of Weights	Erros	
				Mean	Std Dev
A	Account	18	0.4306826179	13.1666667	1.00000000
A	Accountability	21	0.5258764608	9.6666667	1.00000000
A	AddrBook	21	0.433330058	7.8095238	1.00000000
A	Resources	15	1.8529411765	7.6666667	1.00000000

Level of Grupo	Level Tarefa of	N	Sum of Weights	Erros	
				Mean	Std Dev
B	Account	19	0.5691014188	11.0526316	1.00000000
B	Accountability	23	1.6258731489	5.6521739	1.00000000
B	AddrBook	23	1.0415249687	7.0869565	1.00000000
B	Resources	17	2.9265822785	3.0588235	1.00000000

4) Estatísticas Descritivas: Média e Desvio Padrão da Variável Resposta “Erros”. Dados Originais.

Anova MR para Erros - Dados Originais

The GLM Procedure

Level of Grupo	N	Erros	
		Mean	Std Dev
A	75	9.58666667	6.30763873
B	82	6.76829268	5.09732145

Level Tarefa of	N	Erros	
		Mean	Std Dev
Account	37	12.0810811	6.12997362
Accountability	44	7.5681818	5.47041824
AddrBook	44	7.4318182	5.82854933
Resources	32	5.2187500	3.48021482

Level of Grupo	Level Tarefa of	N	Erros	
			Mean	Std Dev
A	Account	18	13.1666667	6.46483702
A	Accountability	21	9.6666667	6.31928266
A	AddrBook	21	7.8095238	6.96145852
A	Resources	15	7.6666667	2.84521319
B	Account	19	11.0526316	5.77805892
B	Accountability	23	5.6521739	3.76114943
B	AddrBook	23	7.0869565	4.69925568
B	Resources	17	3.0588235	2.41015011

1) **Análise Estatística: Análise de Variância para Medidas Repetidas da Variável Resposta "Tempo"**. Como os dados apresentaram heterogeneidade de variâncias foi necessário transformar os dados. A transformação utilizada foi pelo Método dos Mínimos Quadrados Ponderados, cuja variância dos Tratamentos entrou na composição da ponderação.

Anova MR para Tempo - Dados Ponderados

The GLM Procedure

Class Level Information		
Class	Levels	Values
Grupo	2	A B
Rep	23	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
Tarefa	4	Account Accountability AddrBook Resources

Number of Observations Read	176
Number of Observations Used	157

The GLM Procedure

Dependent Variable: Tempo Tempo

Weight: ivar_tempo ivar_tempo

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	49	185.8142314	3.7921272	4.34	<.0001
Error	107	93.4004656	0.8729015		
Corrected Total	156	279.2146970			

R-Square	Coeff Var	Root MSE	Tempo Mean
0.665489	0.097186	0.934292	961.3448

Source	DF	Type III SS	Mean Square	F Value	Pr > F
Grupo	1	0.43455598	0.43455598	0.50	0.4820
Rep(Grupo)	42	55.59953439	1.32379844		
Tarefa	3	69.01792245	23.00597415	26.36	<.0001
Grupo*Tarefa	3	1.67360985	0.55786995	0.64	0.5915

Tests of Hypotheses Using the Type III MS for Rep(Grupo) as an Error Term					
Source	DF	Type III SS	Mean Square	F Value	Pr > F
Grupo	1	0.43455598	0.43455598	0.33	0.5697

Através da Análise de Variância acima, podemos concluir que o efeito do Fator Tarefa é significativo, $p < 0,0001$, em relação à Variável Resposta Tempo.

2) Teste de Comparações Múltiplas de Tukey-Kramer para Comparar Médias de Tempo.

Anova MR para Tempo - Dados Ponderados

Least Squares Means

Adjustment for Multiple Comparisons: Tukey-Kramer

Tarefa	Tempo LSMEAN	LSMEAN Number
Account	1561.19145	1
Accountability	1632.94617	2
AddrBook	1622.72153	3
Resources	791.78629	4

Least Squares Means for effect Tarefa Pr > t for H0: LSMean(i)=LSMean(j) Dependent Variable: Tempo				
i/j	1	2	3	4
1		0.9623	0.9937	<.0001
2	0.9623		1.0000	<.0001
3	0.9937	1.0000		0.0011
4	<.0001	<.0001	0.0011	

Através do Teste de Comparações Múltiplas de Tukey-Kramer, podemos concluir, por exemplo, que o tempo médio da Tarefa “Resources” difere significativamente do tempo médio das demais Tarefas (“Account”, “Accountability” e “AddrBook”), $p < 0,0001$, $p < 0,0001$ e $p = 0,0011$, respectivamente.

3) Estatísticas Descritivas: Média e Desvio Padrão da Variável Resposta “Tempo”. Dados Ponderados.

Anova MR para Tempo - Dados Ponderados

The GLM Procedure

Level of Grupo	N	Sum of Weights	Tempo	
			Mean	Std Dev
A	75	0.0001812194	1199.79319	1.15362311

Level of Grupo	N	Sum of Weights	Tempo	
			Mean	Std Dev
B	82	0.0005886932	887.94240	1.43697444

Level Tarefa	of	N	Sum of Weights	Tempo	
				Mean	Std Dev
Account		37	0.0000701924	1537.85086	0.99120010
Accountability		44	0.0001514458	1550.00077	1.01618021
AddrBook		44	0.0000209312	1620.32253	0.98865086
Resources		32	0.0005273432	689.39849	1.03980154

Level of Grupo	Level Tarefa	of	N	Sum of Weights	Tempo	
					Mean	Std Dev
A	Account		18	0.0000364416	1468.05556	1.00000000
A	Accountability		21	0.0000340769	1783.76190	1.00000000
A	AddrBook		21	0.0000111329	1585.09524	1.00000000
A	Resources		15	0.000099568	858.66667	1.00000000
B	Account		19	0.0000337508	1613.21053	1.00000000
B	Accountability		23	0.0001173689	1482.13043	1.00000000
B	AddrBook		23	9.7983179E-6	1660.34783	1.00000000
B	Resources		17	0.0004277752	650.00000	1.00000000

4) Estatísticas Descritivas: Média e Desvio Padrão da Variável Resposta "Tempo". Dados Originais.

Anova MR para Tempo - Dados Originais

The GLM Procedure

Level of Grupo	N	Tempo	
		Mean	Std Dev
A	75	1467.34667	962.178939
B	82	1389.97561	986.662437

Level of Tarefa	N	Tempo	
		Mean	Std Dev
Account	37	1542.59459	721.20337
Accountability	44	1626.09091	640.40277
AddrBook	44	1624.43182	1442.13847
Resources	32	747.81250	315.81604

Level of Grupo	Level of Tarefa	N	Tempo	
			Mean	Std Dev
A	Account	18	1468.05556	702.80944
A	Accountability	21	1783.76190	785.01757
A	AddrBook	21	1585.09524	1373.42899
A	Resources	15	858.66667	388.13762
B	Account	19	1613.21053	750.29924
B	Accountability	23	1482.13043	442.67753
B	AddrBook	23	1660.34783	1532.10368

Level of Grupo	Level of Tarefa	N	Tempo	
			Mean	Std Dev
B	Resources	17	650.00000	199.35019

OBS: Os programas estatísticos utilizados nas análises estatísticas foram o “Statistical Analysis System” (SAS), versão 9.2 e o “Statistical Package For The Social Sciences” (SPSS / PASWSTAT), versão 18.