Towards Understanding and Quality Assessment of Enterprise Software Systems

Teză destinată obţinerii
titlului ştiinţific de doctor inginer
la
Universitatea "Politehnica" din Timişoara
în domeniul "Ştiinţa Calculatoarelor"
de către

Cristina Marinescu

Conducător ştiinţific: prof.univ.dr.ing. Ioan Jurca
Referenţi ştiinţifici: prof.univ.dr.ing. Mircea Petrescu
prof.univ.dr. Dana Petcu
prof.univ.dr.ing. Horia Ciocârlie

Ziua susţinerii tezei: 20 Martie 2009

BUPT

Seriile Teze de doctorat ale UPT sunt:

1. Automatică
2. Chimie
3. Energetică
4. Ingineria Chimică
5. Inginerie Civilă
6. Inginerie Electrică
7. Inginerie Electronică şi Telecomunicaţii
8. Inginerie Industrială
9. Inginerie Mecanică
10. Ştiinţa Calculatoarelor
11. Ştiinţa şi Ingineria Materialelor

Universitatea „Politehnica" din Timişoara a iniţiat seriile de mai sus în scopul diseminării expertizei, cunoştinţelor şi rezultatelor cercetărilor întreprinse în cadrul şcolii doctorale a universităţii. Seriile conţin, potrivit H.B.Ex.S Nr. 14 / 14.07.2006, tezele de doctorat susţinute în universitate începând cu 1 octombrie 2006.

Copyright © Editura Politehnica – Timişoara, 2009

Această publicaţie este supusă prevederilor legii dreptului de autor. Multiplicarea acestei publicaţii, în mod integral sau în parte, traducerea, tipărirea, reutilizarea ilustraţiilor, expunerea, radiodifuzarea, reproducerea pe microfilme sau în orice altă formă este permisă numai cu respectarea prevederilor Legii române a dreptului de autor în vigoare şi permisiunea pentru utilizare obţinută în scris din partea Universităţii „Politehnica" din Timişoara. Toate încălcările acestor drepturi vor fi penalizate potrivit Legii române a drepturilor de autor.

România, 300159 Timişoara, Bd. Republicii 9,
tel. 0256 403823, fax. 0256 403221
e-mail: editura@edipol.upt.ro

BUPT

Foreword

I would like to thank the advisor of my thesis, Prof. Ioan Jurca, for helping me with choosing the topic of this work. I also wish to thank him for the confidence and encouragement he gave on many occasions over all these years, and for the time he spent carefully evaluating the various aspects of my work. His support helped me to significantly improve the thesis.

I would like to thank Prof. Dana Petcu for accepting to be an official reviewer and for the material support provided as the head of the e-Austria Institute in the early years of my doctoral studies. I thank Prof. Horia Ciocârlie and Prof. Mircea Petrescu for promptly accepting to be part of the PhD committee. I also thank Prof. Octavian Proştean for accepting to chair the examination.

I would like to thank Prof. Vladimir Creţu, the head of the Computer Science and Engineering department, for his logistical support, especially through the difficult times of my last year as a PhD student.

I want to thank Marius Minea for sharing with me his huge research experience and for sacrificing his time to review one of my papers. All these provided me with lots of valuable ideas that shaped my thesis. I am grateful to Tudor Gîrba for our many stimulative and fruitful discussions (almost every time he come to Timişoara), as well as for presenting one of my papers.

I would also like to to thank Océ Software, and Mr. Marius Pentek in particular, for the 2006 grant that financially supported my research in a difficult period of time.

I would like to thank all my colleagues that have contributed to the development of iPlasma. Without their work, implementing the DATES tool, that I created for evaluating the different techniques presented in my thesis, would have been much harder: Radu Marinescu – the initiator of the platform, Daniel Raţiu and Mircea Trifu – for the contributions brought to the MEMORIA meta-model, Petru Mihancea and Andreea Ionete – for developing JMondrian, Cristian Caloghera – for creating the insider front end, Mihai Balint – for writing a lot of tests and, last, but not least Violeta Voinescu – for her effort regarding the implementation of software metrics.

I am also grateful to all those who helped me with implementation effort for the DATES tool: Diana Flacăr – for enabling DATES to extract the database model also from Microsoft Access, Tamaş-Selician Domiţian – for serializing the database model. I would also like to thank Ioana Verebi and George Ganea for their full-hearted collaboration on various research projects.

To all of my friends: Irina – I am so sad I can’t invite you anymore to my PhD defense, Adina and Roxana – for sharing your marvelous experiences with our Lord, Oana – for being such a dear and reliable friend and, last but not least, Janina, for sharing with me your experience in child raising.

BUPT

I would like to thank my grandparents and parents for the way in which I grew. I also wish to thank my sister Meli and my brother Licu for being close to me. I want to thank my other sister, Diana, for being a true friend, with whom I enjoy talk and debating almost any topic. I want to express my gratitude to Ani and Brutus for the way they supported me in many occasions.

My special thanks are for Mihăitză – for behaving so well and letting me finish the writing of the thesis, and for Radu for all the joys that shared together in the past eight years.

But above all, I would like to thank The One who gave us the greatest joy: the experience of living together with Him.

Marinescu, Cristina

Towards Understanding and Quality Assessment of Enterprise Software Systems

Teze de doctorat ale UPT, Seria 10, Nr. 16, Editura Politehnica, 2009, 139 pages, 69 figures, 17 tables.

ISSN: 1842-7707

ISBN: 978-973-625-847-3

Abstract,
The scope of this Ph.D. thesis is the research field of software engineering, while its main goal is to address the issue of high and constantly growing complexity of enterprise software systems and the issue of their permanent need for adaptation to always meet new requirements. In this context, the thesis introduces new techniques and software instruments, which are helpful in order to evaluate and enhance the quality of this type of systems.

BUPT

Contents

1 Introduction **13**

2 Enterprise Applications **17**

2.1 Characteristics of Enterprise Applications 17
2.2 The Object-Oriented Model 19
2.2.1 Design Flaws in the Object-Oriented Model 20
2.3 The Relational Model 21
2.3.1 Design Flaws in the Relational Model 22
2.4 Principles and Patterns in Enterprise Applications 23
2.4.1 Patterns for Mapping Objects to Tables 24
2.4.2 Patterns within the Data Source Layer 27

3 Techniques for Assessing the Design **29**

3.1 Understanding the Design in Regular Systems 29
3.2 Object-Oriented Design Quality Assessment 34
3.3 Assessment Methods in Enterprise Applications 38
3.4 Research Directions 41

4 Analysis Infrastructure for Enterprise Applications **44**

4.1 Modeling Enterprise Applications 44
4.1.1 Modeling Object-Oriented Entities 44
4.1.2 Modeling Relational Database Entities 45
4.1.3 Modeling Object-Relational Interactions 45
4.2 Automated Model Extraction 48
4.2.1 The iPlasma environment 48
4.2.2 The DATES module 55
4.2.3 Groups. Properties. Filters 58
4.3 Characteristics of the Case Studies 60
4.4 Evaluation of the Approach 62

BUPT

4.4.1 Storing entities from the relational database . . . 63
4.4.2 Classifying entities of the data source layer . . . 63
4.4.3 Capturing the accesses between the object-oriented and the relational paradigms . . . 65
4.5 Direct Applications of the Meta-Model . . . 66
4.5.1 Finding entities that belong to more than one layer . . . 66
4.5.2 Finding the Interactions with Databases . . . 69
4.5.3 Visualizing how Tables are Accessed in Code . . . 71

5 Relational Discrepancies Detection 79
5.1 Relational Discrepancies . . . 79
5.1.1 Missed Constraint: An Example . . . 80
5.2 Detection of Relational Discrepancies . . . 84
5.2.1 Build the Groups of Related Tables . . . 85
5.2.2 Detect "de Facto" Constraints . . . 86
5.2.3 Identify the Missed Constraints . . . 88
5.3 Evaluation of the Approach . . . 88
5.3.1 The *KITTA* Application . . . 89
5.3.2 The *TRS* Application . . . 90
5.3.3 The *Payroll* Application . . . 90
5.3.4 The *CentraView* Application . . . 91
5.3.5 Identified Relational Discrepancies according to the percentage of using the tables together in the source code . . . 93

6 Objectual Meaning of Foreign Keys 95
6.1 Objectual Meaning of Foreign Keys . . . 95
6.1.1 Relations between Tables and Classes . . . 96
6.1.2 Refined Semantics of Foreign Keys . . . 96
6.2 The Approach . . . 99
6.2.1 Discovering the N, U and S Relations . . . 99
6.2.2 Discovering the D, A and I Relations . . . 101
6.3 Evaluation of the Approach . . . 102
6.3.1 The *KITTA* Application . . . 102
6.3.2 The *Payroll* Application . . . 103
6.3.3 The *CentraView* Application . . . 104

7 Roles-Aware Detection of Design Flaws 108
7.1 Roles in the Data Source Layer . . . 108
7.1.1 Design Roles and Quality Assessment . . . 109

BUPT

7.2 Extracting Roles in the Data Source Layer 109
7.2.1 Roles in the Data Source Layer 109
7.2.2 Identifying Roles in the Data Source Layer 110
7.2.3 Design Information for Identifying Roles 112
7.3 Roles-Aware Detection of Design Flaws 112
7.3.1 Data Class versus Data Transfer Object 113
7.3.2 Enhanced Detection Rule for Feature Envy 114
7.4 Evaluation of the Approach 115
7.4.1 Identification of Roles 117
7.4.2 Roles-Aware Detection of Design Flaws 119

8 Conclusions. Future Work 122

A Visualizing Accesses to Tables 131
A.1 Tables Accesses. Implementation 131
A.2 Distribution of Operations. Implementation 133

B Roles-Aware Detection of Design Flaws 135
B.1 Table Data Gateway. Implementation 135
B.2 Data Transfer Object. Implementation 137

BUPT

List of Figures

2.1 Layers in a well-designed enterprise application. 18
2.2 Persistent classes in the domain layer. 24
2.3 Single Table Aggregation. 25
2.4 Foreign Key Aggregation. 25
2.5 Single Table Inheritance. 26
2.6 Class Table Inheritance. 26
2.7 Concrete Table Inheritance. 27

3.1 The core of the FAMIX meta-model [76]. 31
3.2 Table books. 32
3.3 Class Book. 32
3.4 Design information must be extracted both from the code and the database schema. 33
3.5 Quality assessment process for an object-oriented system. 34
3.6 System Complexity - an example. 36
3.7 Quality assessment process for an enterprise application. 37
3.8 Class Book which embeds SQL syntactical errors. 39
3.9 Research directions. 43

4.1 A simplified meta-model for object-oriented systems. 45
4.2 A Meta-Model for Relational Databases. 46
4.3 Entity Method Annotated. 47
4.4 The iPlasma analysis platform. 49
4.5 WOC Metric Implemented in Java. Numbers on the left are used for referencing the important lines of code in the text that explains it. 52
4.6 The implementation of the filter *is Public*. Numbers on the left are used for referencing the important lines of code in the text that explains it. 53
4.7 WOC metric implemented in SAIL. 54
4.8 INSIDER - a snapshot. 55

BUPT

4.9 Modeling the structure of a database. 56
4.10 AnnotationDATES – the introduced properties. 57
4.11 The group of accessed tables - implementation. 58
4.12 isUsedTable Filter - implementation. 60
4.13 The Overview Pyramid associated to KITTA. 61
4.14 Classes with design flaws in KITTA. 61
4.15 The Overview Pyramid associated to CentraView. 62
4.16 Method getButton_show of class ShowReservation. 67
4.17 Method search_person_car of class ShowReservation. 68
4.18 Tables Accesses. Representation. 71
4.19 Distribution of Operations. Representation. 72
4.20 Tables Accesses in KITTA. 72
4.21 Distribution of Operations in KITTA. 73
4.22 Tables Accesses in TRS. 73
4.23 Distribution of Operations in TRS. 74
4.24 Tables Accesses in Payroll. 75
4.25 Distribution of Operations in Payroll. 75
4.26 Tables Accesses in CentraView. 77
4.27 Distribution of Operations in CentraView. 78

5.1 Two types of relational discrepancies. 80
5.2 Library database schema, with no explicit constraints among tables. ([PK] marks the primary keys). 80
5.3 Domain Classes for Library example. 81
5.4 The BookDS data source class. 83
5.5 The PublisherDS data source class. 84
5.6 The meta-model of main entities that connect the source code with the database. 85
5.7 Building the group of related tables. 87
5.8 KITTA: The discovered related tables. 89
5.9 CentraView: Some discovered related tables. 92
5.10 Computing the percentage of having (T, R) together. 94

6.1 The objectual meaning of a foreign key, as revealed in the source code. . . . 97
6.2 Objectual meaning of foreign keys. 98
6.3 Detect a Data Keeper Class. 100
6.4 Discover the D, A and I RELATIONS. 101
6.5 Duplicated fields related to the Child table. 103
6.6 Some discovered S Relations between tables and Data Keeper classes. 105
6.7 The D Relations between tables. 106

BUPT

7.1 Relations between a design entity, roles and layers. 109
7.2 Identification of roles in the data source layer. 111
7.3 Table books. 113
7.4 Class BookDataSource. 114
7.5 Class Book - a Data Transfer Object. 115
7.6 Class BookDataSource revised. 116
7.7 Detection of Data Transfer Object. 116
7.8 Roles of methods in KITTA. 117
7.9 Roles of methods in Payroll. 118

BUPT

1

Introduction

The scope of this Ph.D. thesis is the research field of software engineering, while its main goal is to address the issue of high and constantly growing complexity of enterprise software systems and the issue of their permanent need for adaptation to always meet new requirements. Having in mind the powerful and competitive software industry, we adress also the need to create useful platforms, methodologies and tools for dealing with very complex systems [28].

In a society characterized by frequent changes, the software must evolve together with the society, more precisely together with the modeled business domain. In order to be able to evolve and adapt to new requirements it has to be prepared for changes. This means a high design and implementation quality [40]. It is well known both in the software engineering theory and in practice that large-scale, complex applications which exhibit a poor design and implementation are very dangerous because of the delayed effect of these structural problems [17]. The applications are going to run correctly a period of time, but their adaptation to new requirements is going to be unfeasible from the economical point of view. A late discovery of this problem can be dangerous because rebuilding the application can be expensive while in the case of large enterprise applications it is virtually impossible [16]. This emphasizes the need to analyze the software from multiple points of view in order to detect on time the design and implementation problems that could inhibit or make very expensive the evolution of the system.

Another important problem regarding the evolution of a software application is connected to the permanent necessity of its understanding.

In the recent years, as object-oriented systems became increasingly complex, a novel category of software systems emerged, namely enterprise applications. These systems are about the display, manipulation, and storage of large amounts of often complex data and the support or automation of business processes with that data [27].

As a reflection of our society, enterprise applications are characterized by a huge amount of heterogeneity (*e.g.*, various implementation languages used in the same project, multiple

BUPT

programming paradigms, multiple coexisting technologies). As we rely more and more on such systems, their understanding and quality assurance are crucial concerns.

From the point of view of design and implementation these systems can be regarded as object-oriented ones, usually consisting of three primary layers called presentation, domain and data source [24]. Therefore, apparently, the assessment of design quality could be based on the principles, heuristics and best practice of object-oriented design (*e.g.*, [60, 72, 29]). But is this enough?

It is clear from the state-of-the-art literature reflected in [27, 35, 56, 66, 59] that, within an enterprise application, the presentation layer, the data source layer, and also their interaction with the domain layer, are governed by a novel set of principles and patterns which state more specifically what is "good design" for an enterprise system. Furthermore, comparing these specific design rules with the ones used for the design quality assessment of "regular" object-oriented systems (object-oriented systems which do not involve persistency) we notice that sometimes they are even conflicting. For example, a *Data Transfer Object* (DTO) [27, 56] which in an enterprise application is a class carrying data between a client and a server will be always detected as a *Data Class* [26] design flaw. Thus, using a strict object-oriented perspective we are in danger of getting **incomplete** and **inaccurate** results.

Consequently, as the number and the complexity of enterprise systems is increasing, the need to maintain and evolve these systems will require more and more specific means both to understand and to assess *completely* and *accurately* the quality of their design.

In the context of software understanding, finding flaws of object-oriented design has a twofold relevance:

- flaws like *Data Class*, *Feature Envy* [26] hamper the understanding of the code as data and the functionality that uses the data are placed in different locations.
- design flaws like *Large Class* or *Long Method* [26] aside from their malign nature have also the "side effect" of helping us localize the classes and methods where a large part of a system's intelligence is placed. In other words, the location of design fragments affected by particular design flaws is also useful for an initial understanding of a legacy system.

In order to achieve the mentioned goals, the heuristics and patterns of enterprise applications design must be be taken into account. In the last ten years various valuable techniques have been defined for quality assessment in object-oriented systems [11, 21, 37, 57, 58].

Yet, regarding them from the perspective of enterprise applications, almost all of these approaches are limited in the following aspects:

- they rely exclusively on principles, heuristics and best practices of object-oriented design.
- they are based exclusively on a structural view of the source code, without integrating any additional information (*e.g.*, the database schema).

BUPT

- they do not provide features for understanding and evaluating the quality of the relationships existing within the relational part of the system (*i.e.*, the persistency layer) and of the interrelationships between the relational and the object-oriented parts.

Due to the mentioned aspects the current quality assurance techniques (*e.g.*, techniques for detecting design flaws in object-oriented systems) are necessary but not sufficient. A novel layer of dedicated quality assurance techniques is needed in order to address properly the multiple aspects of heterogeneity in design and implementation.

In this dissertation we aim to lay the foundation for a new approach of understanding and assessing the design of enterprise applications. The contributions we bring are presented below:

- different design-related analyses regarding the interactions between the object-oriented part and the relational part of an enterprise application.
- a novel approach for detecting *relational discrepancies* between data-base schemas and source code in enterprise applications.
- a novel technique for determining a refined understanding of the relations among the persistent data, by correlating the information extracted from the database schema with the way the data are used in the source code (*i.e.*, *objectual meaning of foreign keys*).
- a mechanism which help us to identify the classes and methods that fulfill *design roles* which are specific for enterprise systems, and which are described in literature in form of various patterns (*e.g.*, a class that acts as a *Data Transfer Object* [27]). In this context, we define a suite of automatic detection techniques for several *design roles*. The ability to identify such roles is a further step towards a specific understanding of an enterprise application's design. Next, we show how the detection accuracy of two well-known design flaws (*i.e.*, Data Class, Feature Envy) can be improved by taking into account these identified design roles.

In order to introduce the features we define and use specific structural analyses based on a representation (model) of a software system that contains and correlates relevant entities and relationships from both the object-oriented (*e.g.*, classes, method calls) and the relational parts (*e.g.*, tables, columns, accessed tables) of an enterprise application.

Organization. The thesis is structured as follows: in Chapter 2 we present the main characteristics of an enterprise application and several principles and patterns of good design in enterprise applications.

In Chapter 3 we present some techniques for understanding and quality assessment in object-oriented systems, as basically all enterprise applications use an object-oriented back-bone and some representative solutions that fall in the assessment for enterprise applications. We also emphasize that a new meta-model specific to enterprise applications is needed. Chapter 3 is concluded by summarizing the research directions we identified during the studies presented in Chapters 2 and the first part of Chapter 3.

BUPT

In Chapter 4 we introduce a description of the proposed meta-model that is used in order to understand and assess the design of an enterprise application. We continue the chapter by providing a detailed description of the tool support that ensures the automation of the entire approach. We end the presentation of the introduced meta-model with a suite of applications that are used as case studies in this thesis.

Chapter 5 presents a novel approach for detecting relational discrepancies between database schemas and source code in enterprise applications.
Chapter 6 proposes a mechanism for capturing the semantics of the relations between the elements of the persistency layer within an enterprise application, by correlating the information extracted from the database schema with the insight provided by the usage of the database elements within the source code. The mechanism enriches the semantics of data-base relations established by means of foreign keys by revealing its *objectual* meaning (*i.e.*, if the foreign key denotes an inheritance or an aggregation relation).

Next (Chapter 7), we introduce the notion of a *design role* attached to a design entity (*e.g.*, class, method) and a suite of techniques for automatically identifying several design roles that are related with the *data source layer* of an enterprise application. By taking into account the design roles introduced, we revise the detection method for two design flaws in order to make it more accurate for enterprise applications.

All the introduced approaches are accompanied by different experiments based on four case-studies. The conducted experiments are intended to reveal the applicability and the accuracy of the introduced features.

The thesis ends with conclusions, summarizing the brought contributions and some final remarks towards the future work (Chapter 8).

BUPT

2

Enterprise Applications

Among the very complex software systems that the industry is confronted with, in the last 5-7 years a new type of application called *enterprise* has emerged. In the first part of this chapter we present briefly the main characteristics of enterprise applications and the object-oriented, respectively the relational models, as every enterprise application relies on these models. We continue by pointing out some principles and patterns of good design in enterprise applications.

2.1 Characteristics of Enterprise Applications

An enterprise application [1] is a software product that manipulates lots of persistent data and interacts a lot with the user through a vast and complex user interface [27]. This type of software systems includes payroll, automated tellers machines, bank accounting, management applications, etc.

Usually, an enterprise application involves a lot of persistent data, its users manipulate the data concurrently, has a lot of user interface screens and need to be integrated with other enterprise applications [27].

Oftentimes, within this type of applications we need:

- to change the user interface – *e.g.*, from an interface which allows the user to interact with the software in a command line library to one based on windows, dialog boxes and menus.
- to have multiple types of interfaces, for example, a stand alone one based on a graphic library like java.awt or java.swing and another providing a subset of the operations integrated in a web brower.
- to be able to change the way data are stored – from a simple text file into a relational database or an object-oriented database.

[1] also known as an information system or data processing system [27]

BUPT

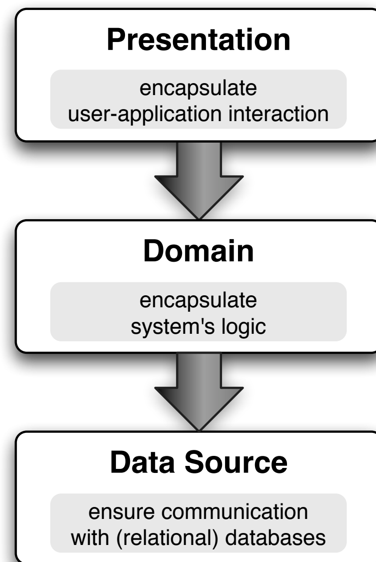

Figure 2.1: Layers in a well-designed enterprise application.

In order to satisfy these needs, it is a good practice that the user interface is totally decoupled from the part of the source code that ensure the communication with the persistency provider, this practice applying also to the logic of the application (*i.e.*, the domain). A design entity which ensures persistency, domain and is also responsible for interacting with the application's users hampers the maintenance (and, consequently, the understanding) and testing of the applications [27].

These constraints led to a multi-layered architecture (see Figure 2.1), consisting of three primary layers namely data source, domain and presentation [24]. As we can notice from the mentioned figure, in a well designed enterprise application there should be no dependency going from a lower-level layer to a higher-level one (*e.g.*, the data source and the domain layers should not depend on the presentation). The responsibility of each layer is presented next, according to [27]:

The Presentation Layer handles the interaction between the user and the application. The application might have a simple presentation which allows its run in command line mode or a graphical user interface or an interface made for working into an HTML-browser. The primary responsibilities of the presentation layer are to display information to the user and to transform commands from the user into actions upon the domain and data source [27].

The Domain Layer is the part of the application that performs the necessary work in order to satisfy the needs of its users. Most of the times it performs computations based on the existing data stored in the database and on the data received from the users and validations

BUPT

upon the data received. Due to the wide spread use of the object-oriented technology [2], there are many enterprise applications which are implemented in an object-oriented manner.

The Data Source Layer ensures the communication with the persistent data. Nowadays there are a lot of persistent mechanisms (*e.g.*, flat files, hierarchical databases, object-oriented databases, relational databases, XML databases) but, as considered in [2], many people choose to persist their data in relational databases.

During this work we refer only to those enterprise applications implemented using the object-oriented paradigm, where the persistency is provided by SQL relational databases. Due to this fact, in the next sections we briefly present the main characteristics of the object-oriented and relational models.

2.2 The Object-Oriented Model

When the object-oriented programming is used, "programs are organized as cooperative collections of *objects*, each of which represents an instance of some *class*, and whose classes are all members of a *hierarchy* of classes united via inheritance relationships" [23].

According to Booch [23], the object-oriented model encompasses the principles of abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence . Although when the object-oriented model appeared those concepts were not new; bringing all of them together for the first time had a strong positive impact on the organization of software programs.
The first four principles are considered in [23] as being major elements of the object-oriented model and that is why we are going to briefly summarize only the first four principles'characteristics.

Definition 2.2.1 An abstraction *"denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer" [23].*

In other words, abstraction provides the behavior of an object related to an interesting point of view. In this context we want to emphasize that the behavior of an object is composed by all the services (*e.g.*, operations, functions, methods which form the object's interface) provided by that object. Abstraction provides the users with information regarding *what* an object does, not *how* the object does (implements) the provided services.

Definition 2.2.2 Encapsulation *"is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation" [23].*

Encapsulation is related to the implementation of an object which has to be hidden. One of the major reasons for hiding the implementation relies on the fact that this implementation might be the subject for various changes which should not affect the clients of the object.

BUPT

Definition 2.2.3 Modularity *"is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules" [23].*

The main goal of the modularity is to reduce the complexity of an object-oriented application by letting the modules be designed and implemented independently.

Definition 2.2.4 Hierarchy *"is a ranking or ordering of abstractions" [23].*

Within the object-oriented model there are two relevant types of hierarchies: the *is a* hierarchy – which denote the class structure – and the *is part of* hierarchy – which denote the object structure.

2.2.1 Design Flaws in the Object-Oriented Model

In order to reveal a good design an object-oriented application has to follow different design:

- *heuristics* like the ones defined in [72] – *e.g.*, "All data should be hidden within its class", "Keep related data and behavior in one place", "Minimize the number of classes with which another class collaborates".
- *principles* – The Open-Closed Principle, The Interface Segregation Principle [60].
- *patterns* – Abstract Factory, Observer, Visitor, Decorator, Proxy Design Patterns [29].

If the design of an object-oriented system breaks the rules mentioned above that ensure a good design, we will encounter a lot of problems which will hamper the evolution of the system. Fowler in [26] presents some deviations – "bad smells" – from a good design and their consequences. Next, we present some of them.

Duplicated Code [26]. Having the same part of source code in many places hampers drastically the possibility of modifying the software system which contains that source code. To remove this design flaw Fowler proposes in [26] also solutions like Extract Method – when we have the same fragment of code in more places within the same class, Extract Class – when encountering the same piece of code among two unrelated classes.

Long Method [26]. A method which contains a significant number of lines of code raises many problems when it has to be understood. In this context we want to emphasize that it is difficult to say precisely which is the lower limit for the number of lines of code associated to a method which make it *long*. Different authors, based on experience [42] and statistics-based thresholds [39] tried to answer this question. Common solutions proposed for removing this flaw are Extract Method, Introduce Parameter Object, Preserve Whole Object [26].

Large Class [26]. A large class tends to play an important part within the business logic of a system. Usually they are complex classes which reveal a significant number of duplicated code. This design flaw is known in the literature as God Class. Common solutions proposed for removing this flaw are Extract Class, Split Class [26].

BUPT

Long Parameter List [26]. A method with a great number of parameters is, on one hand, difficult to understand and, on the other hand, it might be the subject of frequent changes, as more data is needed. In order to remove this flaw the most well-known solution is packing the parameters as a Parameter Object (usually a data structure). This solution is especially recommended when the same list of parameters appears in several methods throughout the system.

Shotgun Surgery [26]. A method which is affected by this design flaw is called by a lot of different other methods. Thus, if the affected method becomes the subject of a change, this change will be propagated into several places.

Feature Envy [26]. When a method is interested (*i.e.*, accesses) in the data of other classes instead of the data from its own class, then that method if affected by the Feature Envy design flaw. Usually, this is a sign that the method should belong to a different class.

Data Class [26]. When a class contains only data and does not provide functionality, this might be a sign than the data and behavior are not stored in the same place. Usually, having Data Classes into the system leads to many further design problems like:

- a lot of duplicated code, as a sign of developing new operations without knowing exactly what current operations which manipulate the contained data are available.
- methods affected by Feature Envy, as they get all the time the necessary data from this type of classes.

2.3 The Relational Model

According to [14], the relational model is often described as having three aspects, as follows:

The Structural Aspect. "A database is seen as a collection of data, typically describing the activities of one or more related organizations. The data in the database are perceived by the user as tables, and nothing but tables" [70]. Tables contain columns having different types (*e.g.*, integer, real, string, date). Every table has a schema according to which data are stored in the rows of the table.

The Integrity Aspect. "The tables satisfy certain integrity constraints. An integrity constraint is a boolean expression that is associated with some database and is required to evaluate at all times to *true*" [14]. These integrity constrains can be declared within the schema of the database in order to make sure that within tables there will not be stored data which break them.

In order to uniquely identify an entity we should assign it a *key* [2]. A *unique key* or *primary key* consists of a single column or set of columns. In a table we can not have two distinct rows with the same value stored in the columns that form the primary or the unique key.

BUPT

The *foreign key* identifies a column or a set of columns in one (referencing) table that refers to a column or set of columns in another (referenced) table. The columns in the referenced table must form a primary key or unique key. Along with the foreign key concept, the relational model includes the referential integrity rule [14]: the database must not contain any unmatched foreign key values.

The Manipulative Aspect. "The operators available to the user for manipulating those tables are operators that derive tables from tables". The most important operators according to [14] are:

- *restrict* – extracts specified rows from a table.
- *project* – extracts specified columns from a table.
- *join* – combines two tables into one on the basis of common values in a common column.

A database management system, or DBMS, "is software designed to assist in maintaining and utilizing large collections of data" [70]. A relational database management system (RDBMS) is a database management system (DBMS) that is based on the relational model. Currently there are many RDBMS providers on the market. As presented by Dates in [14], SQL (Structured Query Language) is the standard language for relational systems and most of the database vendors provides SQL both for data definition and manipulation operations. SQL has been standardized by both ANSI and ISO. Currently there are extensions to standard SQL (*e.g.*, control-flow constructs) and different data base vendors own a proprietary version of SQL (*e.g.*, MySQL, PL/SQL from Oracle, SQL PL from IBM).

2.3.1 Design Flaws in the Relational Model

Within a database, like in the case of source code, different design flaws may occur. [3] presents some common database flaws:

Multipurpose column [3] If a column stores different kind of data (*e.g.*, column *salary* stores the *salary* related to an employee or the *price* payed from a customer), an additional overhead at the source code level will occur in order to ensure a proper mechanism for a correct retrieving of the data stored. In this case the Split Column Refactoring pattern [3] would be useful.

Multipurpose table [3] This design flaw is similar to the one presented above. If a table (*EmployeesCustomers*) stores different types of entities (*e.g.*, information about employees and customers) this will lead to many unused columns (columns with NULL values) for some kind of entities.

Redundant data [3] Redundant data is a serious problem within databases because it is an opportunity of encountering inconsistencies and its handling and support, according to [4], are heavily costly. Usually eliminating the redundant data from a database implies the process of *normalization*. The most important issue regarding the normalization process is that data

BUPT

is stored in one place and one place only [2] and, as a consequence, this process produces highly cohesive and loosely coupled data schemas [2]. [14] addresses the first three normal forms as well as higher level of data normalization. Unfortunately, a database with a great degree of normalization will affect the performances regarding storing/retrieving data in/from the database, as emphasized by different authors in [34, 27, 2].

Tables with too many columns [3] Tables with lots of columns might reveal a sign of an improper cohesion within. Again, it would be helpful to normalize the tables.

Tables with too many rows [3] Tables with lots of rows drastically reduce the performance of searching entities with particular properties. In this case it is advisable to reduce the size of the table, horizontally (*e.g.*, storing some rows in another table or tables) or vertically (*e.g.*, moving some columns in different tables).

Smart columns [3] A smart column is a column where different positions within the data represent different elements (*e.g.*, storing in the *attribute* column of the table *persons* the name, age and address of a person). In this case an additional overhead of parsing will occur.

2.4 Principles and Patterns in Enterprise Applications

Fowler in [27] states that there "are different sorts of software systems, each of them having its own category of problems and complexities". Enterprise applications, being software applications which rely on two different paradigms (*e.g.*, object-oriented and relational), has to overcome the *object-relational impedance mismatch* [2]. The object-relational impedance mismatch is a set of technical and cultural difficulties which are often encountered when a relational database is used within an enterprise application whose source code is written in an object-oriented programming language or style, especially when objects and/or class definitions have to be mapped into a database schema.

In order to be well designed and to overcome the object-relational impedan-ce mismatch, an enterprise application must fulfill some specific design rules and patterns. Different authors [27, 3, 56, 35, 66] have proposed in recent years such rules and patterns.
Next, we are going to present some principles and patterns of good design in enterprise applications regarding

- the data source layer because, as we have already mentioned, it is usually responsible for assuring the proper bridging between the object-oriented and relational paradigms.
- the mapping of objects into relational tables, as every enterprise application has to make persistent a part of its business object.

In the following chapters we well heavily refer to the patterns mentioned below.

BUPT

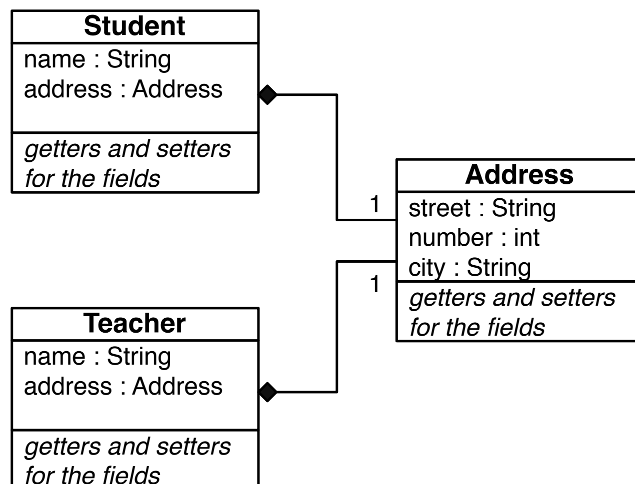

Figure 2.2: Persistent classes in the domain layer.

2.4.1 Patterns for Mapping Objects to Tables

When objects have to be mapped into tables, we have to map object-oriented characteristics like aggregation (part of), inheritance (is a), polymorphism and association to relational tables structures. Different ways in which this task can be properly accomplished have been proposed in the literature in terms of patterns [34, 27]. Next we are going to present some of them with exemplifications.

Patterns for Mapping Aggregation In [34] two patterns for mapping aggregation are presented. In the following paragraphs we are going to present them with respect to the example presented below.

Let us assume that we have to map within a relational database objects instances of the classes *Student* and *Teacher* from Figure 2.2. Keller identified in [34] two ways according to which this operation can be done.

Single Table Aggregation [34] – maps aggregation by integrating all the attributes from the involved classes into a single table. Consequently, for the previous example we have the two tables from Figure 2.3 (*e.g.*, *Students* and *Teachers*), each of them containing the fields *street*, *number* and *city* corresponding to the attributes from the *Address* class.
This pattern reveals an optimal performance – it is possible to retrieve information for a particular object in one database access without any join operation. But, unfortunately, its use hampers the maintainability of the database due to the fact that it might contain a lot of redundant data.

Foreign Key Aggregation [34] – maps aggregation into relational tables using foreign keys.

BUPT

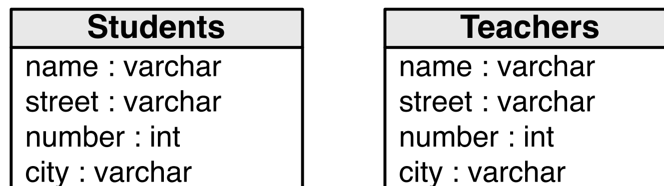

Figure 2.3: Single Table Aggregation.

Thus, for the example above we have three tables, like Figure 2.3 presents, one for each class whose instances have to be mapped in the relational database.

The use of this pattern reduces the performance associated to the retrieving of objects from the persistence part of the application, but the maintenance is affected in a positive manner.

In order to make the data from the relational tables be consistent, we have to specify within the database schema the existing foreign keys, otherwise being possible to store inconsistent data.

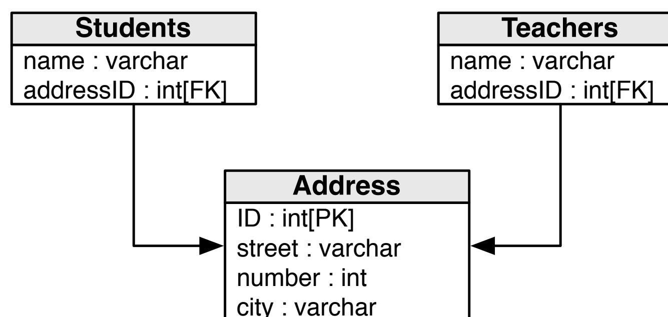

Figure 2.4: Foreign Key Aggregation.

Patterns for Mapping Inheritance Next we present some patterns for mapping inheritance into the relational model, according to [27].

Single Table Inheritance [27] – "represents an inheritance hierarchy of classes as a single table that has columns for all the fields of the various classes" [2].
Let us consider the example from Figure 2.5. In the right part of the figure is presented a table called *Students* that has four columns *Name, Type, Salary, Scholarship*. For each row in the table, column *Type* is responsible for storing the *type* of the student – Ph.D. Student or CS (Computer Science) Student. If the type of the student is Ph.D. then it has a salary, otherwise it has a scholarship. The object-oriented mapping, if Single Table Inheritance is

[2] also known as One Inheritance Tree One Table [34]

BUPT

used, is presented in the left part of the aforementioned picture. One of the major problems with this pattern is that table *Students* may end up being too large.

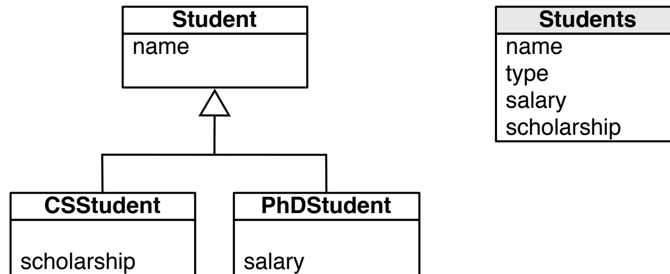

Figure 2.5: Single Table Inheritance.

Class Table Inheritance [27] – "represents an inheritance hierarchy of classes with one table for each class" [3]. We consider the same hierarchy as in the previous example that have to be mapped into one or more tables. If we use Class Table Inheritance, we will create three tables, as illustrated in Figure 2.6, one for each class that we have. The main problem with this pattern regards the loading of objects into memory, an operation which requires multiple table accesses. In this case, like presented in the pattern Foreign Key Aggregation, the relationships between the involved tables are realized in the relational model using foreign keys.

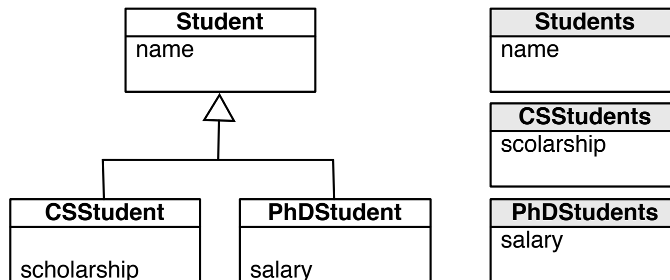

Figure 2.6: Class Table Inheritance.

Concrete Table Inheritance [27] – "represents an inheritance hierarchy of classes with one table per concrete class in the hierarchy" [4]. For the well-known hierarchy of classes from the previous examples, the tables that will be created if we use the Concrete Table Inheritance pattern, are presented in the right part of Figure 2.7. One problem regarding this pattern is that if the superclass is changed, the change is going to affect all the tables presented in the figure.

Remark. From the presentation of existing patterns used for mapping objects to tables the following conclusions are drawn:

[3] also known as One Class One Table [34]
[4] also known as One Inheritance Path One Table [34]

BUPT

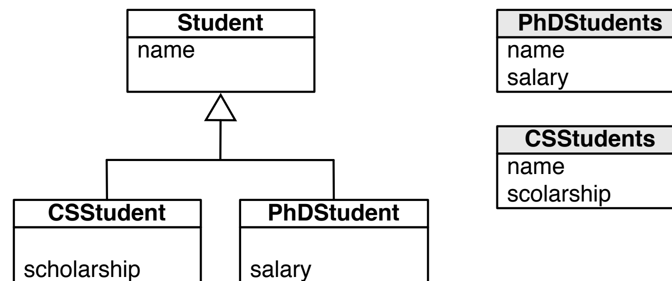

Figure 2.7: Concrete Table Inheritance.

- most of the time, increasing the performance of getting/saving an object from/in the database implies hampering the maintenance of the database. This happens mainly because increasing the time performance is done by putting the data into a minimum number of tables while the normalizing of the tables, which affects positively the maintenance, requires a more significant number of tables.

- when mapping aggregation and inheritance into relational tables it may be possible to use the same solution at the database level, using foreign keys. This causes problems when maintaining the involved tables because without taking into consideration the usage of tables in the source code it is impossible to say precisely if among the involved data is an *is a* or a *part of* relation.

2.4.2 Patterns within the Data Source Layer

Due to the fact that within an enterprise application it is wise to separate SQL accesses from the domain layer, a large number of design patterns regarding the communication between the relational paradigm (a SQL database) and the object-oriented paradigm appeared. In this section we present some of them.

Table Data Gateway (TDG) [27] – holds all the accesses that are made upon one or more database tables [5]. The tables accessed from a class is the set of distinct tables accessed from its methods. In this case in the application we usually encounter one instance for each of this type of class the application contains which handles all the rows in the accessed tables. Because the role of a TDG is to interact directly with the accessed tables and to receive/transmit the data via parameters/return statement, this type of classes are most of the time stateless. The most important achievement we gain when using such class is to decouple the operations that are performed upon the data from the database tables and from the accesses to those tables.

This pattern is used when the domain of the application is organized as a set of operations, each operation satisfying only one demand from the user (Transaction Script [27], Session Façade [56]).

[5] also known as Data Access Object [1], Domain Object Assembler [66]

BUPT

Row Data Gateway (RDG) [27] – encapsulates accesses to a single record in a data source table. For each row from the table which is processed within the application there is one instance of this type of class. Among the accessed table and the attributes of the class there is a perfect match. Like a TDG, RDG enables to decouple the business domain from the underlying data model and data access details. When using this pattern we gain the possibility of changing the structure of the database with fewer changes in the application than in the case of using TDG.

Active Record (AR) [27] – this pattern looks very much like RDG, the single difference being that it adds also domain logic into the services provided by the class. It is suitable for a domain logic which does not reveal a high degree of complexity. The major drawback of this pattern is that it couples the business domain with the database accesses for the involved table.

Data Transfer Object (DTO) [27, 56] – is an instance of a class which contains only attributes and the corresponding getters and setters for each of these attributes. This type of objects transmit data between a client (the domain layer) and a server (the data source layer which encapsulates the access to a database) in order to reduce the number of accesses upon the database tables which are affected by a poor performance regarding the necessary amount of execution time.

This pattern apparently is affected by a design flaw – it is nothing else than a *Data Class* [26]. Consequently, sometimes the design of an enterprise is governed by a set of rules conflicting with the ones from a "regular" object-oriented application. This conflicting rules have to be taken into consideration in order to perform a proper assessment of the design of such applications.

Besides the mentioned design pattern, all the classes that belongs to the *data source layer* have to be written according to the following golden rule: *Do Not Duplicate SQL* [3].

BUPT

3

Techniques for Assessing the Design

We dedicate a major part of this chapter to a briefing of several representative solutions that fall in (or are closely related with) the assessment of the design of object-oriented systems, as basically all enterprise applications use an object-oriented back-bone. The rest of this chapter presents the current techniques regarding the assessment of enterprise applications.

3.1 Understanding the Design in Regular Systems

The first part of this section emphasizes the goal of techniques for understanding the design in object-oriented systems. The section continues with the presentation of different techniques found in the literature regarding the understanding of programs and reveals that the existing techniques are not sufficient for a proper understanding of the design of enterprise applications. We conclude with some proposals that can improve the level of understanding in enterprise applications.

As it is stated in [67], from all the phases in a software life-cycle, the maintenance process is the most time-consuming and consequently, the most expensive. Due to this reality, this thesis is strongly related with the process of maintenance. According to the ANSI/IEEE Standard 729/1983

Definition 3.1.1 Software maintenance *is "the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment".*

Definition 3.1.2 Reverse Engineering *is "the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or a higher level of abstraction" [10].*

Reverse Engineering is the part of the maintenance process that helps understanding the system in order to be able to make the appropriate changes [10]. Within the reverse engineering

BUPT

process different sources of information can be used [17], two of them being the existing documentation and the source code. Unfortunately, most of the times the only source of information available to software maintainers is the source code. "Even in systems that are documented, updates to the documentation rarely keep pace with the evolution of software. In such systems, and in systems with little or no documentation, understanding the function and structure of the program is an essential prerequisite to maintaining it" [67].

As we have mentioned before, the source code is a trustworthy source of information, and in this thesis we consider the source code of an application the main source of information during a reverse engineering process.

If the source code is as small as a few tens of lines, we can perform reverse engineering on a given software system manually; but as the source code becomes larger we need tools for dealing with very complex systems [28] in order to have a powerful and competitive software industry. As it is emphasized in [36], intermediate representations are a key issue for reverse engineering tools. An intermediate representation for reverse engineering "must support different levels of abstraction – from the code-structure level up to architectural level – to be suitable for all phases of reverse engineering" [33]. [36] proposes the requirements of intermediate representations (IR) for reverse engineering tools. Next, we are going to present some of the proposed requirements [1]:

- (R2) "The semantics of the IR must be well-defined and it must exactly describe the constructs of the modeled programming languages; this is necessary for an exact analysis "[36].
- (R9) "IR should support different levels of granularity from fine-grained to coarse-grained "[36].
- (R12) "In a reverse engineering environment IR must also capture higher level abstractions "[36].
- (R13) "Not only does the IR have to have the ability to specify higher concepts, it also must provide means to express any relationships between these concepts. A part-of relationship to describe hierarchical relationships is one example, a communication relationship between two subsystems in an architectural description is another "[36].

Probably the best-known intermediate representation of software systems is based on a meta-model. The *meta-model* of a given software system specifies the existing relevant entities in the system (*e.g.*, for a regular object-oriented system – classes, methods, attributes) and their relevant properties and relations (*e.g.*, inheritance, method calls). The *model* of a given software system contains the specific information extracted from the source code based on the meta-model.

In the last years different meta-models of the source code for object-oriented applications were

[1] We omit to present the requirements which do not affect the reverse engineering of an enterprise application, compared to a "regular" object-oriented application.

BUPT

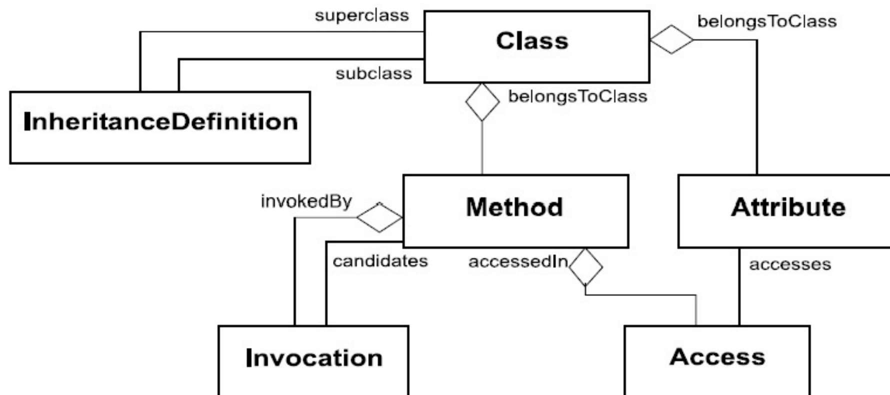

Figure 3.1: The core of the FAMIX meta-model [76].

proposed. Next, we are going to take a look at two meta-models for modeling object-oriented applications: FAMIX [76] and MEMORIA [71].

Figure 3.1 shows the core entities and relations presented in the FAMIX meta-model. All basic elements of an object-oriented languages are present (*e.g.*, class, method, attribute). Furthermore, FAMIX models dependency information, such as method invocations (which method invokes which method) and attribute accesses (which method accesses which attribute). Functions and global variable are modeled because they exist in several object-oriented languages (*i.e.*, C++, Smalltalk). This effectively makes FAMIX support hybrid object-oriented and procedural languages. FAMIX is implemented in Smalltalk.

A similar approach is found in [71]. MEMORIA is a unified meta-model for Java,C++ and, recently, for C# [65]. MEMORIA is implemented in Java and is the foundation of iPlasma – an integrated platform for quality assessment of object-oriented design [53].

In this context, the following question arises:

> *Is it sufficient to perform reverse engineering upon an enterprise application based on a meta-model for a "regular" object-oriented system?*

In order to answer the question, let us consider the example presented below: we have within an enterprise application a table called *books* with the structure presented in Figure 3.2 and class *Book* which is responsible for updating the aforementioned table – its implementation is shown in Figure 3.3.
If we extract the model of the source code from Figure 3.3 based on FAMIX or MEMORIA meta-models (*i.e.*, based on a meta-model for representing "regular" object-oriented systems), we will extract the entities: class *Book*, method *updateAuthor* and related entities strongly connected to method *updateAuthor* – library classes *String, Connection, PreparedStatement, int* and library methods *prepare, setString, setInt, execute*. Next, we are going to see if the

```
create table books (
  ID int primary key, title varchar,
  author varchar, publisher varchar, year int)
```

Figure 3.2: Table books.

requirements R2, R9, R12 and R13 for an intermediate representation for reverse engineering are fulfilled by FAMIX or MEMORIA meta-models.

```
class Book {

    public String updateAuthor(int id, String title,
                   String publisher) throws Exception {
        ...
        Connection con = ... ; //initializations
        PreparedStatement updateStatement;

        String update;
        update = "UPDATE books SET title = ?, " +
                 "publisher = ? WHERE ID = ?";

        updateStatement = con.prepare(update);
        updateStatement.setString(1, title);
        updateStatement.setString(2, publisher);
        updateStatement.setInt(3, id);

        updateStatement.execute();
    }
}
```

Figure 3.3: Class Book.

- **R2** – for an *exact analysis*, the semantics of the IR *must exactly describe the constructs of the modeled programming languages*. As we can notice, class *Book* contains, beside constructs that are present in a "regular" object-oriented system, also constructs connected to the relational part of an enterprise application: a SQL statement embedded in a string that is executed by the method *execute()*. These constructs are not captured by a meta-model specific to a "regular" object-oriented system.
- **R9** – Regarding the different levels of granularity from fine-grained to coarse-grained IR should support, a meta-model specific to a "regular" object-oriented system like FAMIX or MEMORIA is not able to support fine-grained levels of granularity. For example, it can not store information about accessed tables from the body of a method,

BUPT

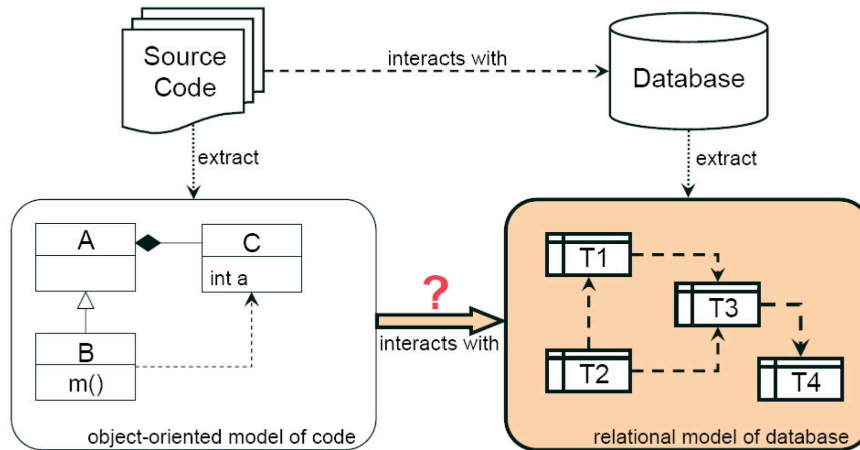

Figure 3.4: Design information must be extracted both from the code and the database schema.

due to the lack of **R2**.

- **R12** – the condition regarding higher level abstractions is also broken. For example, in a "regular" object-oriented application there is no data source layer and, consequently, it is not possible to map automatically entities from the source code into the data source layer. Due to the fact that meta-models support annotations, it is possible to annotate manually entities as belonging to the data source layer. It is obvious that for large-scale enterprise applications this is not a proper solution for reverse engineering. Moreover, manually annotating entities is an error-prone operation.

- **R13** – an enterprise application, as we presented in Section 2.1, embeds two paradigms (*i.e.*, object-oriented, relational). Thus, in such applications we have, on one hand, object-oriented concepts like classes, methods, attributes and, on the other hand, relational concepts like tables, columns, primary and foreign keys. Consequently, a proper meta-model for reverse engineering for enterprise applications should contain, beside concepts from object-oriented paradigms, concepts related to the relational paradigm. Figure 3.4 sketches the concepts that should be introduced in a suitable meta-model for enterprise applications. Moreover, as R13 states, a proper meta-model for enterprise applications must provide means to express existing relationships between these concepts (*e.g.*, accessed tables from a class, classes that access a table).

Feature [2] location in the source code is the process that identifies where a software system implements a specific feature [45]. Features can be divided in two categories: functional and nonfunctional. In this thesis, we are going to consider only nonfunctional features as relevant.

In [45] are presented some case studies which conduct to the following result: Feature location

[2]also known as concept [45], concern [73].

BUPT

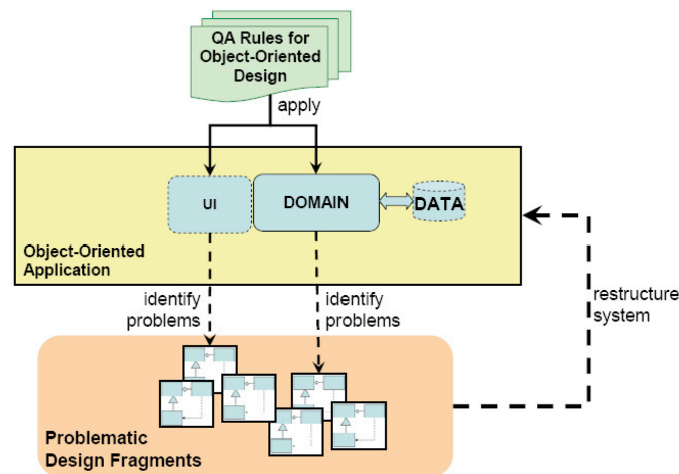

Figure 3.5: Quality assessment process for an object-oriented system.

is also needed for the maintenance of object-oriented code and object-oriented structuring does not facilitate feature location in the source code [45]. Consequently, there is a need for feature location methodologies suitable for object-oriented systems. [73] proposes a methodology for finding and describing concerns using structural program dependencies. FEAT, the tool that implements the presented methodology, uses a structural program model which consists of three type of entities: classes, fields and methods. This structural model allows locating, for example, fragments in the source code that handle Java anonymous classes. But locating fragments in the source code that, for example, insert a row in a specific table is not possible due to the lack of information from the meta-model.

The given discussion reveals that:

> *In order to perform reverse engineering on enterprise applications we need a specific meta-model which contains, on one hand, entities from a "regular" object-oriented system and, on the other hand, entities regarding the relational part of the enterprise application and the interactions between the two paradigms.*

3.2 Object-Oriented Design Quality Assessment

In order to define more precisely the problem that we intend to address in the current Ph.D. thesis, we put it in relation with the closest related approach (*i.e.*, design quality assurance for object-oriented systems). In Figure 3.5 we depict the usual process of quality assessment for object-oriented systems.

The process starts from a set of quality assurance (QA) design rules and heuristics [60, 72, 29].

BUPT

These are usually informal, and therefore they are first transformed into a set of quantifiable rules, which can be then applied on the analyzed project, more precisely on the part of the project which is designed in an object-oriented manner. The result is a set of design fragments that are affected by design problems and that need to be restructured in order to improve the quality of the system.

Software metrics are frequently used in problem detection. They quantify simple properties of design structures and can be used to identify abnormal properties of these structures. Software metrics that address the most important characteristics of good object-oriented design like cohesion, coupling and inheritance are defined in [41, 6, 39]. An important work on object-oriented software metrics is [44] where empirical threshold values which signify abnormal characteristics of design entities are also presented. They were established based on the author's experiences with some C++ and Smalltalk projects. In order to support automatic analysis based on software metrics, tool support for metrics calculation is required. Such tools are presented in [20, 53].

In [39] the authors introduce a detection strategy as a "composed logical condition, based on metrics, that identifies those design fragments that are fulfilling the condition". They also presents several detection strategies used for finding design entities (*e.g.*, classes, methods) which break the:

- Identity Harmony - "How does a design entity (class, method) define itself?" [39] (*e.g.*, detection of Feature Envy, Data Class, Brain Method)
- Collaboration Harmony - "How does a design entity interact with other design entities?" [39] (*e.g.*, detection of Intensive Coupling, Shotgun Surgery)
- Classification Harmony - "How does a design entity define itself with respect to its ancestors and descendants?" [39] (*e.g.*, detection of Refused Parent Bequest, Tradition Breaker)

Because in Chapter 7 we introduce an approach which makes use of the current detection of Data Class and Feature Envy design flaws we show next the algorithms presented in [39] which enable to find those design entities which reveal the mentioned flaws.

Thus, in order to find the *envy* methods, the algorithm takes into account if the methods:

- access directly more than a few attributes from other classes.
- make use of much more attributes from other classes than the attributes used from the class where they belong.
- the used attributes from outside belong to few classes.

Knowing the *Data Classes* from the system requires finding the classes witch offer to their users more data than functional services, do not reveal a high degree of complexity (the complexity is counted based on the McCabe's Cyclomatic Complexity [61]) and contains public data.

BUPT

A first mandatory condition that enables finding automatically the mentioned classes and methods is to have a model of the source code extracted according to a meta-model.

The problem detection phase of the reengineering process can also be addressed with visualization techniques. One well-known technique for software visualization is called polymetric view [38]. "A polymetric view is a metrics-enriched visualization of software entities and their relationships" [38]. Polymetric views help to understand the structure and detect problems of a software system in the initial phases of a reverse engineering process.

In the mentioned visualization technique software entities are represented using rectangles while the relations among them are displayed with edges. Every rectangle is enriched with up to five software metrics: the size (width and height), the color(*e.g.*, different types of gray, white and black, black being usually used for representing the highest value associated to that metric) and the position.

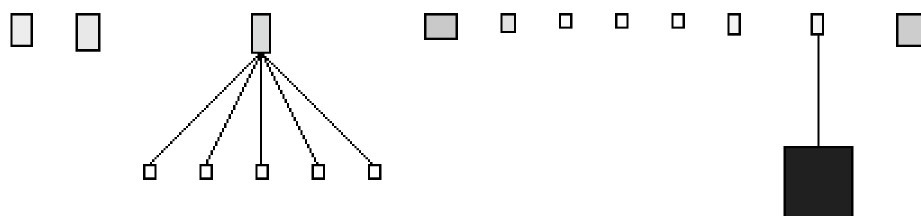

Figure 3.6: System Complexity - an example.

System Complexity View is an instance of Polymetric View visualization. In Figure 3.6 is illustrated an example of how this visualization looks like. The rectangles represent the classes from the system while the edges represent the inheritance relations among the classes. The width of the rectangles are enriched with the values of the NOA (Number of Attributes) metric, the heigh represents the NOM (Number of Methods) and the color is related to the LOC (Lines of Code) the class contain.
In [37, 19] is presented a new visualization which helps in the understanding of classes. This visualization is also metrics-based and it is called *Class blueprint*. This technique may also be applied in the context of finding anomalies among the design while it allows the identification of some of suspicious class blueprints.

Another approach for problem detection can be found in [11]. The author presents a technique for analyzing legacy code, specifying frequent design problems as PROLOG queries and locating the occurrences of these problems in a graph-based model derived from the source code.

A special problem that may appear in large applications is the duplicated code. It is one of the factors that severely complicate the maintenance and evolution of any software system. In [21] a language independent and lightweight approach to code duplication detection is

BUPT

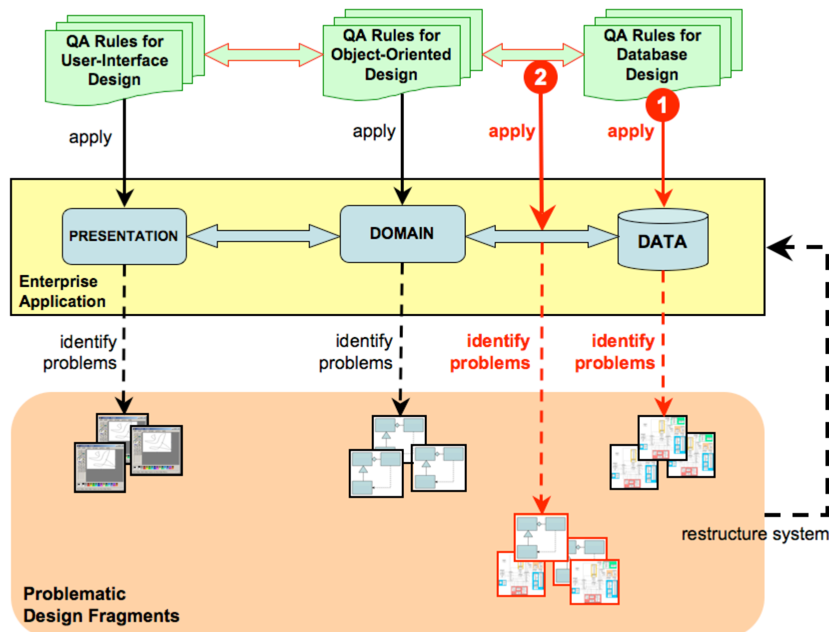

Figure 3.7: Quality assessment process for an enterprise application.

introduced.

> *Yet, regarding them from the perspective of enterprise applications, almost all of these approaches are limited in two aspects:*
>
> - *they rely exclusively on principles, heuristics and best practices of object-oriented design.*
> - *they are based exclusively on a structural view of the source code, without integrating any additional information (e.g., the database schema).*

But, as we presented in Section 2.1, enterprise applications are characterized by a huge amount of heterogeneity. They encapsulate different technologies, different paradigms and must fulfill specific design rules and heuristics depending on the concrete type of the application. All these aspects must be considered when the quality of the design and implementation are evaluated. Unfortunately, almost none of the analysis techniques presented before take into consideration this type of heterogeneity. In other words they unilaterally treat the analyzed system. Because of this reason the current quality assessment techniques strongly need improvement.

> *Yet, due to enterprise applications intrinsic heterogeneity the current quality assurance techniques (e.g. techniques for detecting design flaws in object-oriented systems) are necessary but not sufficient. A novel layer of dedicated quality assurance techniques is needed in order to address properly the multiple aspects of heterogeneity in design and implementation.*

BUPT

One of the main goals of this Ph.D. thesis is to define quality assurance techniques and an adequate tool support for detecting design problems in enterprise applications.

In Figure 3.7 we depict our vision of a generic design quality assessment approach for an enterprise software system. The approach takes into consideration the heterogeneity of such systems and the consequent need to reflect this characteristic in the techniques used for quality assessment. The first important distinction is that each of three layers of an enterprise application (*i.e.*, the presentation, the domain and the data source layer) must be addressed by a distinct set of specialized quality assurance rules. These design rules exist for both the design of user interfaces [59] and for the design of databases [27, 66, 35, 34]. Based on these rules, problematic design fragments can be identified for the presentation and the data source layer, in a similar fashion with the approach presented in Figure 3.5. Yet, none of the three layers exists in isolation; consequently there is a significant amount of the system's complexity involved in the relation between these layers. This brings us to the two issues that the current Ph.D. thesis aims to address (marked in Figure 3.7 with the bullets numbered 1 and 2):

- PROBLEM 1 – How can the rules and patterns within the data source layer of an enterprise application be made quantifiable? What is the proper tool support needed to detect design problems within the data source layer automatically?
- PROBLEM 2 – What are the proper QA design rules that specify the relation between the domain and the data source layer, especially if the data source layer is based on a relational database model? How can we overcome the paradigm shift? What is the proper tool support needed to detect design problems automatically?

3.3 Assessment Methods in Enterprise Applications

We dedicate this section to the recent techniques for assessment in enterprise applications. The scope of this presentation is to reveal the differences between existing techniques for assessment of enterprise application and our vision of a generic design quality assessment approach for an enterprise software systems presented in Figure 3.7.

We have mentioned previously that within this thesis we refer only to those enterprise applications implemented using the object-oriented paradigm, where the persistency is provided by SQL relational databases. A significant part of applications developed in this way embed within the source code SQL statements as strings that are sent in order to be executed upon a SQL relational databas to well-known methods as *executeQuery(String sql)* from Java. These constructions make impossible their correctness verification from the syntactical point of view at compile time. As a result, runtime errors might appear. For example, let us consider the example from Figure 3.8 where the variable *update* is a string that encapsulates a SQL statement which contains two errors (*e.g.*, two commas and WHERE clause misspelled). Due to the fact that *update* is a variable whose type is *String*, the two errors will be discovered only at runtime.
In [62] the authors summarize the main types of problems that may appear when performing accesses to relational databases as embedded strings, as follows:

BUPT

```
class Book {

    public String updateAuthor(int id, String title,
                   String publisher) throws Exception {
        ...
        Connection con = ... ; //initializations
        PreparedStatement updateStatement;

        String update;
        //syntax errors
        update = "UPDATE books SET title = ? " +
                 ",, publisher = ? " +
                 "WHEERE ID = ?";

        updateStatement = con.prepare(update);
        updateStatement.setString(1, title);
        updateStatement.setString(2, publisher);
        updateStatement.setInt(3, id);

        updateStatement.execute();
    }
}
```

Figure 3.8: Class Book which embeds SQL syntactical errors.

- strings that contain mispelled names – *e.g.*, "UPDATE boks".
- strings with syntax errors – *e.g.*, a WHEERE clause, like the one from Figure 3.8.
- SQLs which contain data type mismatch – *e.g.*, passing an int instead of a smallint.
- having the possibility of executing undesirable instructions upon the databases – *e.g.*, embedding into a value of a parameter a SQL command like DROP TABLE.

In [30] is presented a method that addresses some problems of syntactical verification of SQL incorporated statements. The authors introduced a static program technique based on an interprocedural data-flow analysis that verifies the correctness of dynamically generated query strings.

In [62] is proposed an automatic generator for classes to be used for the relational database manipulation. The generated classes based on the database schema involved in the application (*i.e.*, the object model) are used in order to construct every possible valid SQL statement. For each existing table four classes are created for dealing with each type of statement for manipulating data in the tables (select, insert, update and delete). Using the generated classes by the approach removes the shown drawbacks of using embedded SQL statements into strings.

BUPT

The same issue is addressed in [13] where are introduced Safe Query Objects, a technique for representing queries as statically typed objects while still supporting remote execution by a database server.

The advantage of the presented solutions is that in the application there will be no direct interaction with the database, but at the same time the migration to another type of database will be impossible because of the strong coupling between the domain layer and the generated classes (data source layer).

[18] shows a tool set for testing transactions in relational database applications. This includes a technique for checking complex properties of the database, namely checking if transactions are consistent with their requirements.

All the presented techniques help us to ensure the *correctness* of the communications performed within the entities of the data source layers.

> *The aim of our approach is to increase the level of understanding of the enterprise applications'design, the accuracy of object-oriented problem detection techniques and to create specific analyses applied to enterprise software systems.*

When extracting design information, specific to the relational part of the application, there is one bad habit [78], namely to rely on getting from the database schema the complete semantics of attributes, primary keys and foreign keys. This is a bad habit because this completeness is almost impossible to reach. An example of such an approach, based exclusively on the database schema, is the Referential Integrity Utility for IBM DB2 Cube Views [22], a tool which detects missing primary keys, missing foreign keys, nullable foreign keys and generates data definition language to add the necessary DB2 informational constraints.

Detecting missed constraints among database tables is part of what is called *data reverse engineering* (DRE), "a collection of methods and tools that help an organization determine the structure, function, and meaning of its data"[9]. One specific concern which needs to be addressed in the context of DRE is "to retrieve constraints which are not explicitly declared in the database schema, but are verified in the code" [31]. "Data reverse engineering is a complex and expensive task and needs to be supported by program understanding and tools [12]".

Recently, Yeh and Li proposed in [78] an approach where various procedures like field comparison, data analysis, code analysis are applied in order to determine the semantics of attributes, followed by the identifications of primary keys, foreign keys and cardinality constraints.

In [7] are presented some idiosyncracies of relational database design. Some of the presented idiosyncracies can not be detected using only the database schema and, consequently, a meta-model which embeds design entities from both existing paradigms(object-oriented and relational) is helpful for a correct identification of some design problems related to the database

BUPT

schema.

In [32] the authors propose the DB-Main approach which mainly answers the question: how a change in the database schema is propagated within the inspected application? As the authors claim, in order to use the DB-Main approach, the database has to be fully documented and some documentation must be built using reverse engineering techniques. In this case a part of the needed documentation can be extracted by looking at the usages of tables in the source code.

In [15] a tool is proposed that is specifically designed for database reengineering. The tool builds an instance of *Database*, a class containing a meta-model for representing relational databases with no dependence of the vendor. The problem with the meta-model defined by [15] is that it models solely the entities from the relational part of the application (*i.e.*, the elements that compose the database schema). In contrast, this thesis states that in order to perform reverse engineering on enterprise applications we need an *enhanced meta-model* that contains *all* the existing design entities from enterprise applications, both from its relational (*e.g.*, representation of columns and tables) and its object-oriented (*e.g.*, representation of methods and classes) part. Furthermore, it is of even greater importance that these two parts need to be properly connected, by modeling the way relational entities are used from the object-oriented code. Consequently, the meta-model proposed in [15] is incomplete to support the types of analyses that are needed in enterprise applications.

Staiger [74] proposes an approach for analyzing, programs with a graphical user interface. It detects the parts of the source code which belong to the Graphical User Interface, the widgets and hierarchies and provides us with the event handlers connected to the events generated from the widgets. In the context of this thesis the approach has proven to be insightful in providing hints on how to detect the separation of layers in an enterprise application. Although our thesis is focused on the relation between the data source layer and the domain layer, abstracting from the approach of Staiger proved to be an interesting starting point.

In enterprise applications we can find the persistency level implemented using the EJB framework. In [43] were introduced several quality attributes design primitives associated with and Enterprise Java Beans distributed system. They are architectural building blocks that target the achievement of quality attribute requirements like performance, modifiability, reliability and usability. The issue of analyzing the performance and scalability of EJB applications is also addressed in [8]. None of the previous approaches *modify* quality assurance techniques specific to "regular" object-oriented systems in order to make them suitable for enterprise applications.

3.4 Research Directions

This section is a summary of the research directions that arisen from the studies performed and presented during the previous chapters. The research directions are illustrated in Figure 3.9 and summarized below.

BUPT

As we have presented in Section 3.1, a meta-model that embeds entities regarding, on one hand, the object-oriented programming paradigm and, on the other hand, regarding the relational paradigm, is needed in order to be able to perform reverse engineering upon an enterprise application. Consequently, the first issue that is tackled in this thesis is related to the construction of *a proper meta-model for enterprise applications* which facilitates the processes of reverse engineering and data reverse engineering upon this type of applications.

The main reason we build a specific meta-model for enterprise applications is to define a suite of specific analyses that address the understanding and evaluation of design and implementation quality of enterprise applications.

We propose a set of high-level design analyses regarding the overall architecture of an enterprise application, with a special focus on the data source layer and the communication between the domain and data source layers. As a concrete example, it is well known that in an enterprise application the data source and the presentation layers must not depend directly [27]. In this context it is very interesting to detect design entities that have a *double identity* in the sense that they ensure the communication with the database and also with the user interface.

Continuing the same direction (*i.e.*, understanding and quality assurance for enterprise applications) we propose two visualizations which permit to identify different anomalies regarding the usages of tables in the source code.

Furthermore, we define a technique that detects missing foreign keys that should have been defined on various database tables. Moreover, the technique allows us to distinguish between two types of relations between tables: *part-of* and *is-a* relations. These relation types are distinguished based on the detected foreign keys and on how these tables are used in the object-oriented source code.

We also identify classes and methods that fulfill *design roles* which are specific for enterprise systems, and which are described in literature in form of various patterns (*e.g.*, a class that acts as a *Data Transfer Object* [27]). In this context, the thesis defines a suite of automatic detection techniques for several *design roles*. The ability to identify such roles is a necessary step towards a specific understanding of an enterprise application's design.

As we presented in Section 2.1, sometimes it might be the case that the rules and patterns of "good" design for enterprise applications are conflicting with the rules and patterns of "good" design for "regular" object-oriented applications. Due to this, the quality assurance techniques for "regular" object-oriented applications applied upon enterprise applications may bring inaccurate results. In this context, another research direction closely related with the aforementioned two regards the modification of quality assurance techniques for object-oriented systems in order to improve their accuracy when applied to enterprise applications. This modification is implemented by taking into account the identified design roles.

In practice for large-scale industrial enterprise applications performing the proposed analyses

BUPT

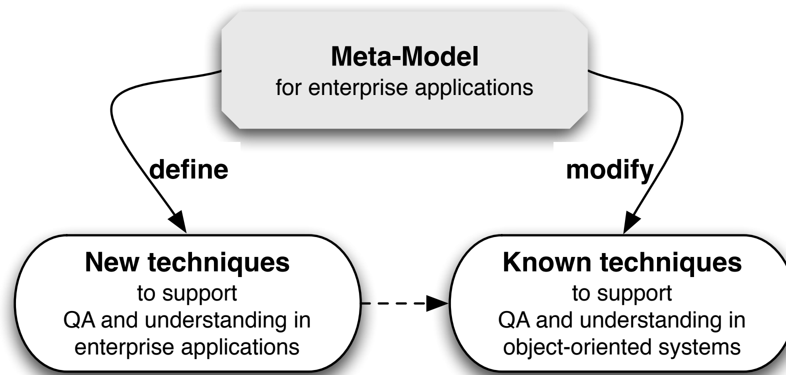

Figure 3.9: Research directions.

manually is practically impossible. Thus, we need a proper tool support which would allow us to perform all the aforementioned analyses on enterprise systems. In this context, we develop an analysis infrastructure (called DATES [3]) that permits to implement specific analyses for enterprise applications. DATES contains the meta-model that includes the mentioned specific aspects of enterprise applications. For example, we make possible for each object-oriented design entity (*e.g.*, class, method) to be mapped to one or more of the application's layers (*e.g.*, data source, domain or presentation layer) where it belongs.

[3] Design Analysis Tool for Enterprise Systems.

BUPT

4

Analysis Infrastructure for Enterprise Applications

4.1 Modeling Enterprise Applications

As we mentioned before, in order to perform reverse engineering upon an enterprise application we need a specific meta-model for this type of application. We dedicate this section to the specific meta-model for enterprise applications we have developed in order to facilitate the process of reverse engineering upon such applications. The introduced meta-model for enterprise applications contains, as Figure 3.4 presents, entities specific to regular object-oriented systems, entities representing the relational database and the interactions among them. Next, for each type of design information from the proposed meta-model we dedicate a separate section.

4.1.1 Modeling Object-Oriented Entities

Entities from object-oriented systems may be modeled using an object-oriented language. In this case, the meta-model is represented as an interconnected set of data classes, usually one for each type of design entity. The fields are either elementary properties of that design entity or links to other related data structures. For example, a structure that models the *Class* design entity is expected to have a field of type *Method* that establishes its connection to the methods the class contains. The model of the system, extracted based on the meta-model, contains also information regarding the *calls* (*i.e.*, which methods are called by a particular method) and *accesses* (*i.e.*, which variables are accessed by a particular method) from existing methods.
In Figure 4.1 we show a simplified depiction of the meta-model for representing entities from object-oriented systems. The use of an object-oriented language for implementation allows us to hide the fields that describe the entity and define operations by which an entity communicates with other entities e.g., every entity has a *getName()* method, the *Class* entity has a *getMethods()* operation that returns a collection with all the *Method* entities defined

BUPT

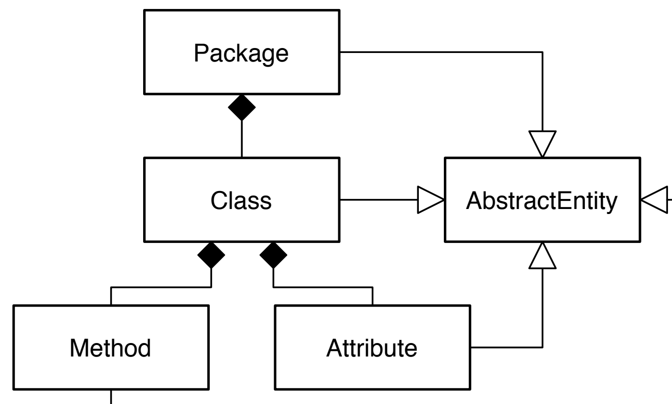

Figure 4.1: A simplified meta-model for object-oriented systems.

in the given class.

4.1.2 Modeling Relational Database Entities

In order to extract information from a database that is a part of an enterprise application, we need to establish exactly which type of information we need to extract. This brings us into the issue of *which are the design entities and the existing relations between these entities in a relational database?*

According to the presentation of relational databases from [70], a relational database consists of one or more tables where each table has its own schema. A schema of a table consists of the name of the table, the name of each field (or attribute, or column) and the type of each field from the table. Additionally, integrity constraints can be defined on the database schema. Thus, a meta-model for a relational database must contain, like in Figure 4.2, the entities *TableSchema* and *ColumnSchema* found in a relational database.

4.1.3 Modeling Object-Relational Interactions

As we mentioned before, the entities (*e.g.*, classes, methods) that ensure the communication with the relational database belong to a layer called *data source*. Consequently, between the object-oriented part of an enterprise application and the relational part there are interactions only within the data source layer. Thus, finding the interactions between the object-oriented and relational entities requires the identification of the entities (*e.g.*, classes and methods) that belong to the data source layer. But the introduced meta-model has to satisfy also the (R12) requirement from [36] and that it why we have to classify also entities from the presentation layer.

In order to do this, we take a simple approach to this issue, by taking into account the various

BUPT

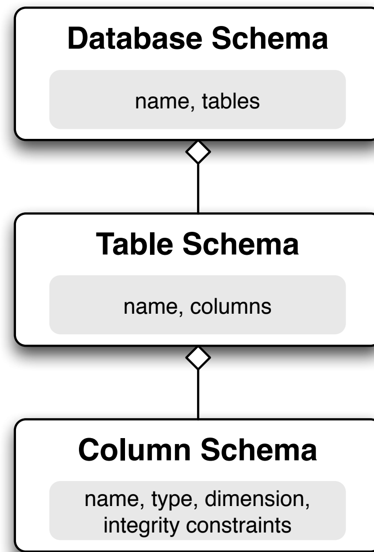

Figure 4.2: A Meta-Model for Relational Databases.

usages of third-party libraries and/or frameworks that are specific either for the presentation or for the data source layer.

Mapping of Methods to Layers. The following rules determine the mapping of methods to one of the three layers of an enterprise application:

1. A method is considered to belong to the **data source layer** if it invokes one or more methods from a specific library that provides an API for accessing and processing data stored in a data source, usually a relational database (*e.g.*, the method invokes the `executeQuery()` method from the `java.sql` package).
2. A method is mapped to the **presentation layer** if it calls one or more methods from a specific third-party library and/or framework that provides components for building the user interface (*e.g.*, calls methods from classes found in the `java.swing` package).
3. If none of the previous two rules applies for a method then the method is mapped to the **domain layer**.

Mapping of Classes to Layers. In order to map classes to layers we use the following set of rules:

1. A class containing one or more methods belonging to the **data source layer** will be mapped to the **data source layer**.
2. A class that contains one or more methods belonging to the **presentation layer** will be mapped to the **presentation layer**.

BUPT

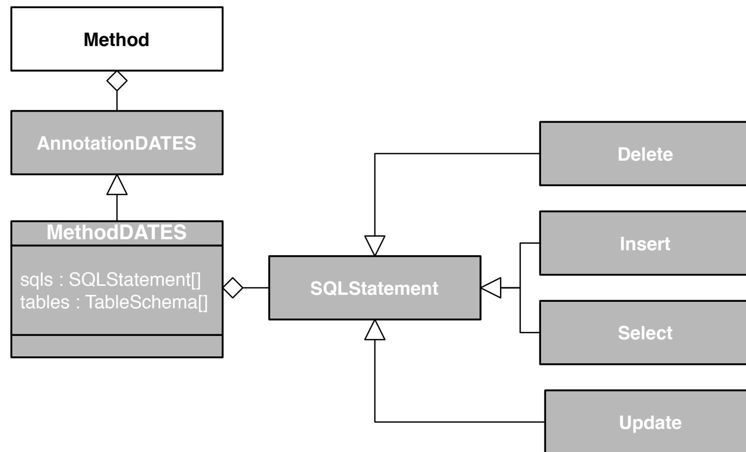

Figure 4.3: Entity Method Annotated.

3. A class that is derived directly or indirectly from a specific third-party library and/or framework that provides components for building the user interface is mapped to the **presentation layer** (*e.g.*, a class that extends `JButton`).
4. A class that contains one or more methods belonging to the **domain layer** will be mapped to the **domain layer**.
5. A class that contains methods that belong to different layers will map to each of the layers to which its methods are mapped to. For example, if a class has methods belonging to the data source layer and to the domain layer, the class will be mapped both to the data source and to the domain layer.

Remark. In conformity to these rules, methods will always belong to a single layer, while the last rule for class mapping makes it possible for a class to be assigned to several layers (usually two adjacent layers). This mapping decision might look confusing, as we would expect for each design entity to belong to a single layer. But note that we extract (reverse engineer) these mappings from the source code, where the clear initial design intentions might have become "blurred" during implementation. Thus, we expect at least some of these classes that are mapped to multiple layers to reveal signs of design flaws (*e.g.*, a class that is mapped both to the *data source* and to the *presentation* layer).

The communication between the object-oriented part of an enterprise application and the relational database is performed within the methods belonging to the *data source layer*, usually, by executing SQL commands as embedded strings from well-known methods as *executeQuery(String sql), executeUpdate(String sql)*. Thus, the methods of classes are the primary entities that ensure the communication with the databases - in this context, the *Method* entity from the object-oriented meta-model has to be enriched with information regarding the operations performed upon a relational database.

The proposed solution regarding this issue is presented in Figure 4.3. Class *MethodDATES*

BUPT

contains information regarding the operations upon the database the method performs: *e.g.*, delete, insert, select, update, each of these operations involving one or more tables. Class *SQLStatement* contains an attribute that stores the tables accessed by the operation. The information regarding the tables accessed by an entity is propagated from low-level entities (operations performed within the bodies of methods) to high-level entities according to the following rules:

- a method stores the set of the tables accessed by its body.
- a class stores the set of the tables accessed by its methods.
- a package stores the set of the tables accessed by its classes.

At this moment, finding the classes that access a particular table called *myTable* requires an iteration through the classes from the data source layer and for each class from the data source layer must be performed another iteration over its accessed tables in order to find out if the current class accesses the table *myTable*. This operation for large-scale enterprise applications might be time-consuming and for this reason we decided to store in the entity *TableSchema* also information regarding the entities that access the table (*e.g.*, functions, classes).

4.2 Automated Model Extraction

Due to the fact that the meta-model for enterprise applications contains entities specific to regular object-oriented systems, we decide not to build our tool support related to modeling enterprise applications from the scratch – we built it on the top of the MEMORIA[71] meta-model which is part of the iPLASMA [53] environment. Thus, before showing the enhancements we bring to MEMORIA we present the main characteristics of the used meta-model as well as the iPLASMA environment.

4.2.1 The iPLASMA environment

iPLASMA[1] is an integrated environment for quality analysis of object-oriented software systems that includes support for all the necessary phases of analysis: from model extraction (including scalable parsing for C++, Java and lately C#) up to high-level metrics-based analysis, or detection of code duplication. iPLASMA has three major advantages:

- extensibility of supported analyses.
- integration with further analysis tools.
- scalability – it was used in the past to analyze large-scale projects in the size of millions of code lines (*e.g.*, Eclipse and Mozilla).

Figure 4.4 presents the layered structure of iPLASMA quality assessment platform. The tool platform has the source code as input and help us during the necessary steps of performing

[1]**I**ntegrated **Pla**tform for **s**oftware **m**odelling and **a**nalysis

BUPT

static analyses, from parsing the code and creating the model of the source code up to an easy implementation of the introduced analyses. All of the provided support is integrated by an uniform front-end, namely INSIDER. Next, we are going to briefly introduce the main components of the IPLASMA quality assessment platform.

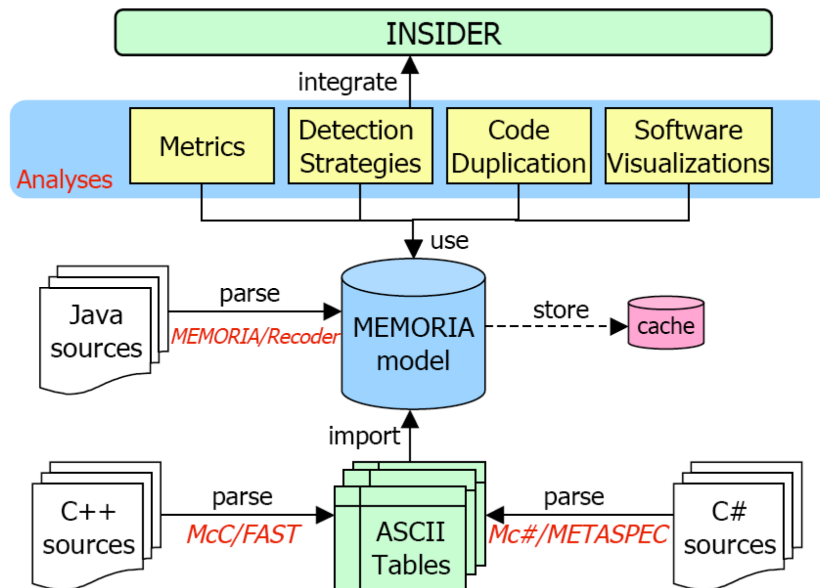

Figure 4.4: The iPlasma analysis platform.

The MEMORIA meta-model. A first step within the process of software analysis is the construction of the proper model of the system. The purpose for constructing the model is to extract from the source code the information which is relevant from the point of view of a particular goal. Thus, for analyses focused on object-oriented design it is important to know the types of the analyzed system, the functions and variables and information about their usages, the inheritance relations between classes, etc.

MEMORIA [71] is a meta-model that can represent Java, C++ and C# systems in a uniform manner, by capturing different types of design information (*e.g.*, packages, classes, variables, methods).

For Java systems the IPLASMA environment uses the open-source parsing library called Recoder [75] in order to extract all the necessary information in form of the MEMORIA object-oriented meta-model, which is implemented in Java.

McC (Model Capture for C++) [63] is a tool which extracts the necessary design information from C++ source code, based on Telelogic's FAST [25] parsing library. It receives as input a directory containing the source code and it produces a set of related ASCII tables containing

BUPT

Infos and Systems	S1(Java)	S2(Java)	S3(C++)	S4(C#)
Size	~45Mb	~77.5Mb	~80Mb	~17.5Mb
Types	10.500	10.023	24.091	3.601
Time	~3h	~5h	~28h	~2h

Table 4.1: Time Performances of Model Extractors.

the extracted design information. In iPlasma this information is loaded in terms of the MEMORIA object-oriented model.

McC# (Model Capture for C#) [65] is a tool which extracts the necessary design information from C# software systems. As McC, McC# receives as input a folder containing the source code of the analyzed system and produces as output a set of ASCII tables containing the model of the system. Due to the fact that most existing design entities are common for C++ and C# (e.g., classes, methods, attributes), it is possible to have an unified structure for the tables produced by McC# and McC. Nevertheless, some differences between the two languages exist – for example, they have different access modifiers (C# has also internal and protected internal access modifiers). At this time, McC# considers that access modifiers have, like in C++, only three types (public, protected and private). Entities declared as internal or protected internal are considered to have the default C# access (*e.g.*, private). Such minor details like this one are not extremely relevant from design analyses point of view. McC# has been implemented in Visual Studio 2003 and it is based on the Metaspec's [2] parsing library.

All the aforementioned extractors have been heavily used for analyzing different object-oriented software systems. In Table 4.1 we present the time performances (*i.e.*, the time needed to parse the source-code and build the complete model each system) related to some models extracted from different software systems, written in all the languages supported by the MEMORIA meta-model. For Java and C++ systems (S1..3) the experiments were made previously on a computer having an Intel P4 2.8GHz processor, 1024 RAM and running Windows XP. For the C# system (S4) the configuration slightly changed, the processor being an Intel Core 2 Duo.

Existing design-related analyses. Based on the extracted information from the source code in iPlasma are implemented several types of design analyses. Next, we are going to briefly present them.

Metrics. The iPlasma [53] platform contains a library of more than 80 design metrics. The metrics can be divided into the following categories:

- size metrics – measure the size of the analyzed entity (*e.g.*, Lines of Code(LOC), Number of Methods(NOM), Number of Classes(NOC), Number of Packages(NOP)).
- complexity metrics – measure the complexity of the analyzed entity (*e.g.*, Cyclomatic Complexity(CYCLO)[61]).

[2]A free trial version can be found at http://www.csharpparser.com/C#

- coupling metrics – measure the data coupling between entities (*e.g.*, Coupling Between Objects[57], Number of Called Operations (CALLS) within the body of a method, Number of Called Classes (FANOUT)[44]).
- inheritance metrics – characterize the class hierarchies (Height of Inheritance Tree (HIT)[57], Number of Direct Descendants(NDD)).
- cohesion metrics – measure the cohesion of classes (*e.g.*, Tight Class Cohesion(TCC)[6]).

All the mentioned categories of metrics are applicable at different levels of abstraction – for example, LOC is computed for the whole system, and also for a particular class or method while TCC is computed only for classes.

Some of the mentioned metrics are computed in order to obtain the *Overview Pyramid* [39] – "a metrics-based means to both describe and characterize the structure of an object-oriented system by quantifying its complexity, coupling and usage of inheritance".

In Figure 4.5 we present how the WOC metric (Weight of a Class) – the number of functional public methods divided by the total number of public members [57] – is implemented in Java within the described platform. We present this example in order to introduce some facilities provided by iPlasma when implementing different analyses.

As it is noticeable, different types of constructs (*e.g.*, FilteringRule, Group-Entity) appear within the implementation of the metric. In iPlasma every design entity is inherited from AbstractEntity and this makes possible, among others, for each design entity the following:

- finding if a particular property is true or false – this is done by applying a filter (an instance of FilteringRule) upon the entity.
- getting a group (an instance of GroupEntity) of design entities connected to the inspected entity.

In Line 1 is declared and instantiated an entity whose type is *FilteringRule* and it is used in order to remove from the group of public methods of the class the constructors. In Line 4 is declared a variable whose type is *GroupEntity* – this type is used for representing a collection of design entities and provides support for getting elements with different specified properties. Thus, in Line 5 the group of the methods the class contains is obtained and from these only the public methods are retained (except constructors). In Line 7 from the attributes of the class only those which are allowed to be used by any client of the class is retained and in Line 9 is constructed a group which keeps only the accessor methods of the class. In Lines 10..13 the value of the metric based on the previously obtained entities is computed.

We present in Figure 4.6 the implementation of the "*is public*" filter. In the constructor of the *IsPublic* class the name of the filter is defined. This is done by specifying it as a string (Line 5 from Figure 4.5) together with the type of the entity on which the filter can be applied (in this case, a method). When it is executed, it receives a parameter whose type is used in the

BUPT

```
public class WOC extends PropertyComputer {
public WOC() {
 super("WOC", "Weight of a Class", "class", "numerical");
}
public ResultEntity compute(AbstractEntity anEntity)
{
1    FilteringRule constructor = new IsConstructor();
2    FilteringRule notConstructor;
3    notConstructor = new NotComposedFilteringRule(constructor);

4    GroupEntity publicMethods;
5    publicMethods = anEntity.contains("method group").
     applyFilter("is public").applyFilter(notConstructor);
6    GroupEntity publicAttr;
7    publicAttr = anEntity.contains("attribute  group").
                 applyFilter("not encapsulated");
8    GroupEntity accessorMethods;
9    accessorMethods = publicMethods.applyFilter("is accessor");
10   double accessorM = accessorMethods.size() +
                       publicAttr.size();
11   double publicM = publicMethods.size() +
                       publicAttr.size();
12   if (publicM == 0) return new ResultEntity(0);
13   return new ResultEntity(1.00 - (accessorM / publicM));
}
}
```

Figure 4.5: WOC Metric Implemented in Java. Numbers on the left are used for referencing the important lines of code in the text that explains it.

MEMORIA meta-model for representing methods (*e.g.*, Method) and the result is based on the value stored in the model.

According to [55], when metrics are implemented, the following particular set of key mechanisms are involved at the model level:

- navigation – allows browsing through the entire model.
- selection – enables the definition of a *view of interest*, by choosing only a subset of an entity's fields.
- set arithmetic – creating a group of design entities by performing different set arithmetic operations (*e.g.*, union, intersection) between two or more groups.
- filtering – helps for finding the entities with a particular property (*e.g.*, all the public attributes from a given class).

BUPT

```
1   public class IsPublicMethod extends FilteringRule
2   {
3     public IsPublicMethod()
4   {
5     super(new Descriptor("is public", "method"));
6   }
7
8   public boolean applyFilter(AbstractEntity anEntity)
9   {
10    if (!(anEntity instanceof lrg.memoria.core.Method))
11       return false;
12
13    return ((lrg.memoria.core.Method) anEntity).isPublic();
14  }
15  }
```

Figure 4.6: The implementation of the filter *is Public*. Numbers on the left are used for referencing the important lines of code in the text that explains it.

- property aggregation – is used for computing and associating a single value for multiple entities (*e.g.*, getting the number of classes from the analyzed system).

When these mechanisms are implemented in a structure-based approach (in this case the meta-model is represented as an interconnected set of data structures, like the MEMORIA meta-model) we encounter an overhead regarding the expressions of filtering, navigation and selection. When using a repository-based approach, the meta-model being represented as a knowledge source that can be queried (oftentimes appearing physically as a relational database system in which usually one table is defined for each design entity) only the expressions of navigation and set arithmetics are obfuscated due to the constructions of the language. The mentioned obfuscations led to a new language namely SAIL [55] dedicated to the implementation of metrics and design related analyses appeared, this language part of the IPLASMA platform.

In Figure 4.7 we present the implementation of the same metric (WOC) in SAIL.

Detection Strategies. IPLASMA contains all the implementations of the detection strategies presented in [39].

Code Duplication. DUDE (Duplication Detector) [77] is a tool within IPLASMA that uses textual comparison at the level of line of codes in order to detect fragments of duplicated code.
Software Visualizations. Based on the design information extracted and imported in MEMORIA we can generate in IPLASMA using the JMONDRIAN [64] software tool different software views such as System Complexity or Class Blueprint [39].

BUPT

```
float woc(Type type)
{
 float woc;
 Operation[] operationsWAccessors;
 Operation[] publicOperations;
 Variable[] publicVariables;
 int publics;

 woc = 0;
 operationsWAccessors = select (*) from type.operations where
          ((isAccessor==false) && (scope.scopeType==type) &&
           (isConstructor==false));
 publicOperations = select (*) from type.operations where
          ((isPublic==true) && (scope.scopeType==type) &&
           (isConstructor==false));
 publicVariables = select (*) from type.variables where
          ((isPublic==true) && (scope.scopeType==type));

 publics = publicOperations.# + publicVariables.#;
 if(publics > 0) {
  woc = 100 * operationsWAccessors.# / publics ;
 }
 return woc;
}
```

Figure 4.7: WOC metric implemented in SAIL.

The INSIDER front-end. In order to achieve a complete view of a software system we need to obtain the results of different design analyses (*e.g.*, metrics, detection strategies, visualizations) computed by different tools. INSIDER is the front-end of iPLASMA which integrates in an uniform manner all the mentioned tools. It integrates all the existing tools as plugins, this way providing extensibility for further inspections.

In Figure 4.8 we present how the user interface of INSIDER looks like. It mainly consists of three parts: the group browser (top right), the group inspector (top left) and the entity browser (bottom).

The group browser allows us to navigate through the different existing types of groups of design entities (classes, methods, attributes), as well as to see different subgroups from the initial ones. For example, from all the classes of a particular system we can retain only the ones which were explicitly defined in the project (*i.e.*, excluding the library ones).

The group inspector is the part of the user interface which allows to see only those properties (*i.e.*, values of metrics) which are important in the context of different points of view. For example, if we are interested in finding how many duplications a class contains, we will select

BUPT

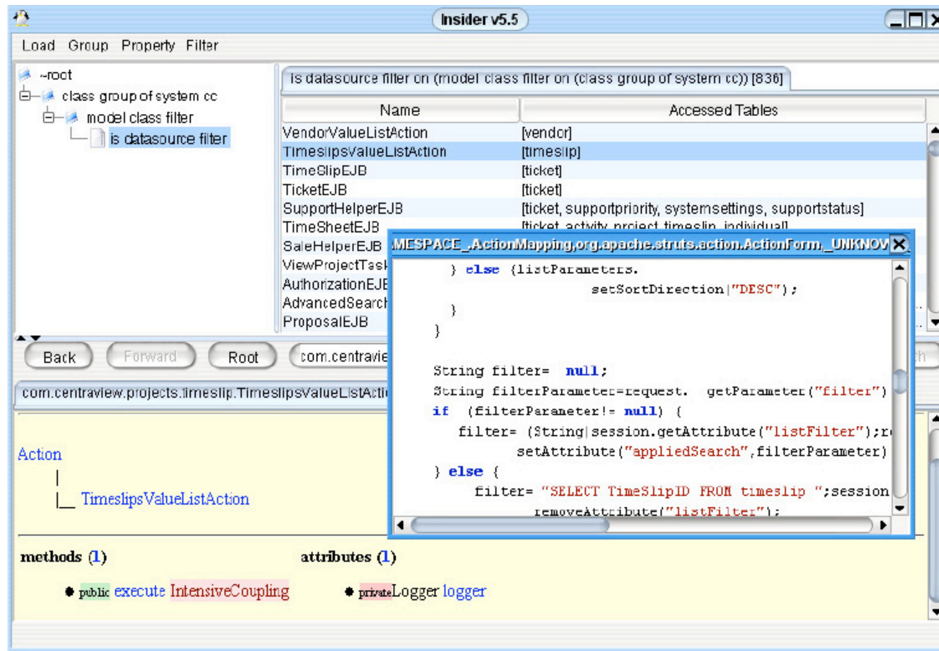

Figure 4.8: INSIDER - a snapshot.

in the group inspector only the properties related to the duplications – in this case the values of the EDUPCLS, IDUPCLS and IDUPLINES metrics [39].

The entity browser allows to see together a set of different properties for the selected design entity in the group inspector. In this context we want to emphasize that the entity browser is a plugin and, thus, configurable to show different properties of the analyzed entity. For example, the default entity browser presents for a class its ascendants, methods and attributes as well as the values of the properties of being a Data Class, a God Class or a Tradition Breaker [39].

4.2.2 The DATES module

In this section we present the tool called DATES [48] [3] which we have developed in order to model (*i.e.*, obtain an intermediate representation) enterprise applications.

First, we introduce the different necessary types for modeling relational databases, as we assume that each analyzed enterprise application contains a relational database. Consequently we create the classes *DatabaseSchema*, *TableSchema* and *ColumnSchema*. In Figure 4.9 we present the existing fields for modeling a database.

We developed also a model loader for extracting the necessary information according to the

[3] Design Analysis Tool for Enterprise Systems

BUPT

```
 public class DatabaseSchema extends AbstractEntity {
  private String databaseName;
  private HashMap<String, TableSchema> tables;
  ...
}
```

Figure 4.9: Modeling the structure of a database.

developed meta-model of the database. The information is obtained for the first time directly from the database. After the first extraction the model is saved in a repository (in this case, a simple text file). The repository is created in order to be able to load the database model without being necessarily to connect again to the database server.

At the moment we assume that the persistency is provided by SQL relational databases, in particular MySQL and Microsoft Access. Because querying the database in oder to find out its structure slightly differs in the mentioned SQL dialects, we use the Abstract Factory [29] design pattern in order to be loosely coupled with the different existing SQL dialects. Currently, when DATES is started it is configurable with a parameter whose type can be *mysql*, *access* or *file*, depending on the source from which the structure of the database is loaded. The last presented value is used when loading the model of the database from a file. This way, we ensure that DATES is able to support with minor changes the extraction of the models for different relational databases vendors (*e.g.*, DB2 from IBM).

In Figure 4.9 we present the key elements of a design entity (in this case, a class) used for modeling a database. The first key element is that this class extends the *AbstractEntity* class. This inheritance relation is needed as DATES is integrated within the iPlasma environment. More precisely, DATES extends the framework defined by iPlasma. The aforementioned inheritance relation allows to manipulate in an uniform manner the newly introduced design entity (in this case a relational database) and the existing object-oriented design entities (*e.g.*, classes, methods). Integrating DATES with iPlasma [53] provides a further benefit: it allows one not only to browse the design entities of an enterprise application, but also to assess the quality of its design. This quality assessment is performed by combining regular object-oriented assessment techniques (*e.g.*, software metrics, detection strategies) with specific techniques like the ones we defined in [46]. These techniques are described in detail in Chapters 5, 6 and 7.

The communication with the relational database for the studied enterprise applications is mainly performed by executing SQL commands as embedded strings from well-known methods as *executeQuery(String sql), executeUpdate(String sql)*. Thus, in order to find out the accesses from the object-oriented part to the relational part we have to identify first the methods which call operations from the *java.sql* package – *i.e.*, identifying the methods which belong to the data source layer. In this context we want to emphasize that our approach,

BUPT

by making use of a parameter, allows us to identify the methods from the data source layer where the communication is done by calling operations from one or more packages similar to the *java.sql* package (*e.g.*, *javax.sql*).

The *AbstractEntity* class from which every existing design entity is derived in the MEMORIA meta-model has an attribute whose type is *Hashtable* whose role is to permit for each design entity to be annotated with different properties. Within the DATES module, we annotate the methods, classes and packages with the properties shown in Figure 4.10. In the case of the methods, we add the supplementary field *ArrayList* <*SQLStatement*> *statements* in order to store the SQL statements performed within their bodies.

```
public abstract class AnnotationDATES{
 private boolean isDatasource = false;
 private boolean isPresentation = false;
 private boolean isDomain = true;

 private HashMap<String, TableSchema> accessedTables;
 ...
}
```

Figure 4.10: AnnotationDATES – the introduced properties.

After setting the layer or layers where an object-oriented design entity belongs, according to the algorithm presented in 4.1.3 we find the operations performed upon the involved database within the bodies of methods that belong to the data source layer. Finding these operations is done according to an algorithm based on string comparison.

The final step performed in order to have a complete model of an enterprise application is to set, for each existing table, the classes that access it. This operation is redundant but it is useful in order to not be obliged to iterate all the time through all the classes from the data source layer in order to find out the classes that access a particular table.
Summarizing, the extraction of design information from enterprise applications requires the following steps:

- Construct the *model of the system* according to the MEMORIA [71] meta-model. At this phase, the model of the system is not enriched with specific information regarding the interaction between the object-oriented and relational paradigms found in enterprise systems.
- Load the DATES tool in the INSIDER [53] front-end. At this moment the model of the system is enriched with specific information of enterprise application's design, both for the relational paradigm (tables, columns, primary and foreign keys – obtained from the schema of the database by an extractor we have developed) as well as for

BUPT

the object-oriented one (*e.g.*, accesses to database tables from the bodies of methods within the data source layer, obtained also by the extractor).

4.2.3 Groups. Properties. Filters

In this section we present a part of the initial enhancement introduced by DATES when it is loaded into the iPlasma environment. This enhancement provides us with different groups, properties – for getting a particular value associated to a given design entity – and filters in the INSIDER front-end and they may be used in order to implement further higher-level analyses, as we present in the next three sections.

Name	Is Applied-To	The Group Of
table group	system	the tables of the system
accessed tables	method	the tables accessed
statements group	method	SQL statements
column group	table	the columns of the table
column group	table	the columns of the table
accessed from classes	table	classes accessing table
accessed from methods	table	methods accessing table

Table 4.2: Some introduced groups.

```
public class AccessedTables extends GroupBuilder{
 public AccessedTables()
 {
1 super("accessed tables", "", "method");
 }

 public ArrayList buildGroup(AbstractEntityInterface anEntity)
 {
2 if((anEntity instanceof Method) == false)
3  return new ArrayList();

4 MethodDATES an;
5 an = (MethodDATES)anEntity.getAnnotation(Dates.DATES);
6 return new ArrayList(an.getAccessedTables());
 }
}
```

Figure 4.11: The group of accessed tables - implementation.

In Table 4.2 we present some of the new groups introduced by the DATES module. As the figure presents, the new groups are applicable for different types of design entities (*e.g.*, methods, tables). In iPlasma if an entity is contained by a different type of entity (*e.g.*, a

BUPT

method is contained by a class) then the group associated to the aggregated entity may be computed also for the container entity (*e.g.*, accessed tables from a class).

Name	Is Applied-To	Meaning
NAT	class, method	Number of accessed tables
NDbI	method	Number of SQL instructions performed
NAC	table	Number of classes which access the tables
NC	table	Number of columns the table contains
NSt	table	Number of statements performed upon
NISt	table	Number of insert statements
NSSt	table	Number of select statements
NUSt	table	Number of update statements
NDSt	table	Number of delete statements
Type	column	The type of the column
Dim	column	The dimension of the column
Def	column	The default value of the column

Table 4.3: Some introduced properties.

Name	Is Applied-To	Is True if Entity
is datasource	class, method	belongs to the datasource layer
is domain	class, method	belongs to the domain layer
is presentation	class, method	belongs to the presentation layer
isInsert	statement	is insert statement
isSelect	statement	is select statement
isUpdate	statement	is update statement
isDelete	statement	is delete statement
isUsedTable	table	is used in the source code
isUnusedTable	table	is not used in the source code
isPrimaryKey	column	is primary key
isForeignKey	column	is foreign key
isNull	column	is null

Table 4.4: Some introduced filters.

In Figure 4.11 we show how we implement the group of *accessed tables*. In Line 1 we specify the name of the group as well as the type of entity which it is computed for (in this case, a method). In Line 2 we make sure that the received parameter is indeed an instance of the *Method* class. In Line 5 we get the introduced annotation called *Dates.DATES* and we return the accessed tables from the body of method (Line 6).

In Table 4.3 we present the main available properties within DATES.

BUPT

In Table 4.4 we point out some existing filters. The presented filters are either elementary – derived directly from the schema of the database *e.g.*, isPrimaryKey, isNull – or computed by taking into account different facts *e.g.*, is datasource, isUsedTable. In Figure 4.12 we show the implementation of the *isUsedTable* filter. As we can notice, the implementation relies on the introduced group *accessed from classes*.

```
public class isUsedTable extends FilteringRule
{
 public isUsedTable()
 {
  super(new Descriptor("isUsedTable", "", "table"));
 }

 public boolean applyFilter(AbstractEntityInterface anEntity)
 {
  return anEntity.getGroup("accessed from classes").size()>0;
 }
}
```

Figure 4.12: isUsedTable Filter - implementation.

4.3 Characteristics of the Case Studies

In order to evaluate the approaches we introduce in this Ph.D. thesis we conducte different experiments on a suite of enterprise applications. The size characteristics of the systems for which we present the results obtained by applying the Tool Support described in Section 4.2 are summarized in Table 4.5.

System	Size(bytes)	Classes	Methods	Tables
KITTA	212,553	37	254	10
TRS	537,058	54	500	10
Payroll	780,871	115	580	12
CentraView	11,162,565	1527	13369	217

Table 4.5: Characteristics of the case studies.

KITTA and **TRS** are two enterprise applications were developed by two separate teams of students within the software engineering project classes. Within **KITTA** and **TRS** the persistency is provided by relational databases created in Microsoft Access.
In Figure 4.13 we present the Overview Pyramid [39] associated to **KITTA**. This analyses reveals within this application:

- a lack of inheritance relations, too many small packages and a low value related to the cyclomatic complexity (from the rectangles colored in blue).

BUPT

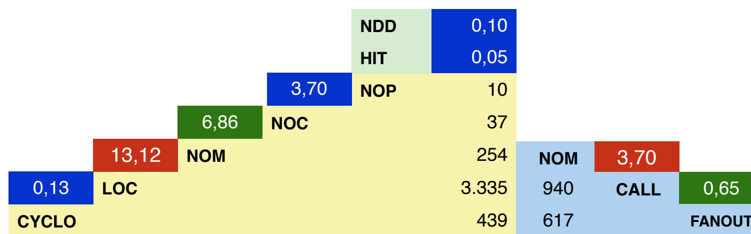

Figure 4.13: The Overview Pyramid associated to KITTA.

- the methods are quite long and they call a lot of methods from their body (from the rectangles colored in red).

- a normal value regarding the number of methods in a class and the coupling dispersion (from the rectangles colored in green).

```
ResultsFrame                 - has 1 FeatureEnvy
ParentApplicationRequest     - has 1 BrainMethod
Request                      - is DataClass
ParentApplicationForm        - has 1 BrainMethod
Kindergarten                 - has 1 BrainMethod
Search                       - has 1 IntensiveCoupling method
Child                        - is DataClass
User                         - is DataClass
DataBaseTricks               - is DataClass
```

Figure 4.14: Classes with design flaws in KITTA.

In Figure 4.14 we present the classes that are the subject of different design flaws according to the report generated by iPlasma before loading the DATES plugin. Similar reports may be generated for all the applications.

Payroll is an industrial enterprise application whose scope is to manage information about the employees from a company, the persistency being provided by a MySQL relational database.

CentraView is an open-source enterprise application obtained from *http://www.sourceforge.net/projects/centraview* which provides growing businesses with a centralized view of all customer and business information. The persistency of this application is provided by a MySQL relational database.
From the Overview Pyramid (Figure 4.15) we find that this last application reveals a lack of inheritance relations and, on average, too many methods in a class and long methods.

BUPT

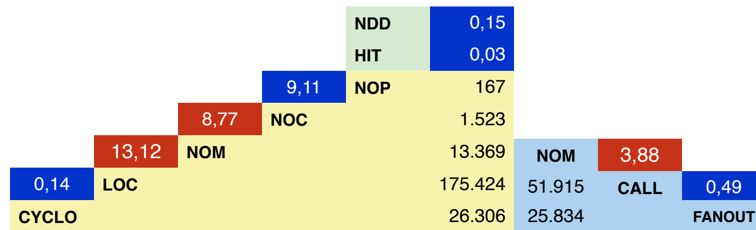

Figure 4.15: The Overview Pyramid associated to CentraView.

4.4 Evaluation of the Approach

In this section we present some experiments performed upon the enterprise applications whose main characteristics were specified in Section 4.3.

As we have already mentioned, a mandatory first step for extracting the model of an enterprise application consists of extracting the model for the object-oriented part according to the MEMORIA meta-model. In Table 4.6 we present the necessary amount of time for extracting the design entities directly from the source code according to the MEMORIA meta-model (second column), as well as the necessary amount of time for enhancing the extracted model with information specific to enterprise applications – *e.g.*, tables, accesses from methods to tables – (last column).

System	MEMORIA(seconds)	DATES(seconds)
KITTA	6	1.5
TRS	9	2
Payroll	9	2
CentraView	33	20

Table 4.6: Time Performances for models extraction.

MEMORIA has the capability of storing the information extracted from the source code within a cache file. In Table 4.7 we present for each application the size of the stored cache-file as well as the needed amount of time for loading the information from the saved file.

System	File Size(MB)	MEMORIA(seconds)
KITTA	1.1	1
TRS	4.4	3
Payroll	2.4	3
CentraView	60.5	30

Table 4.7: Time Performances for models extraction from files.

The goal of the experiments was to answer questions regarding the reliability of the extracted

BUPT

model based on the introduced meta-model:

- Does the model of the relational database store all the entities from the relational database (*e.g.*, tables, columns)?
- Are the entities of the data source layer correctly classified?
- Are the accesses from the object-oriented design entities to relational tables well-captured?
- Are the accessed-by relations regarding the tables from the relational database well-captured?

4.4.1 Storing entities from the relational database

For the small size enterprise applications, we performed a manual investigation in order to find out if the model captures all the entities from the involved relational databases within. The manual investigation consists of

- checking if the number of tables from the model (last column from Table 4.5) is equal to the number of tables from the involved database.
- checking, for each table, if the captured name and columns in the model correspond to the existing name and columns in the table from the database. We consider a column from the model corresponding to a column from a relational table if they have the same name and the same characteristics (*e.g.*, type, dimension, integrity constraints).

The performed investigation reveals that the model captures all the existing entities from the relational database within the enterprise application.

Regarding the *CentraView* application, we performed only a partial verification and the results confirm that all the checked entities from the relational database were well-captured in the obtained model.

4.4.2 Classifying entities of the data source layer

Classifying correctly the entities of the data source layer plays an important role due to the fact that only entities classified as belonging to the data source layer interact with the relational database (*i.e.*, the accesses between an object-oriented entity and a relational entity are established only if the object-oriented entity belongs to the data source layer). In this context, a first step in validating the correctness of our approach regarding the existing relations between the two involved paradigms is to check if the object-oriented entities that interact with relational entities were classified correctly as belonging to the data source layer.

Classifying correctly the entities from the data source layer in an enterprise applications means that:

BUPT

- among the entities that were classified as belonging to the data source layer there are no false positives (*e.g.*, entities erroneous identified by the approach as belonging to the data source layer).
- among the entities that were not classified as belonging to the data source layer there are no false negatives (*e.g.*, entities which were not identified by the approach as belonging to the data source layer).

For each analyzed enterprise application the number of object-oriented entities that were classified by our approach as belonging to the data source layer is presented in Table 4.8.

	KITTA	**TRS**	**Payroll**	**CentraView**
Data Source Classes	9	10	16	837
Data Source Methods	25	24	74	3349

Table 4.8: Size of the data source layer.

In order to find out whether design entities were correctly classified we performed, like in the previous case, a manual investigation for the small size enterprise applications. During the manual investigation, we browsed thought the source code of each enterprise application in order to classify every class as belonging/not belonging to the data source layer. For each class identified as belonging to the data source layer we counted the number of its methods that ensures the communication with the data source layer.

The classification obtained by performing the manual investigation of *KITTA* and *TRS* enterprise applications coincides with the one performed by our approach (*i.e.*, there are no false positives and no false negatives). But within the *Payroll* application the manual investigation classifies 18 classes as belonging to the data source layer, instead of 16 which were automatically classified. Among the 16 classes which were classified automatically as belonging to the data source layer there are no false positives and no false negatives. As well, within the classified methods of the 16 classes there are no false positives and no false negatives.

In order to find out why within the model of the *Payroll* application there are two false negatives (*i.e.*, the two classes which were manually identified as belonging to the data source layer) we checked again manually the two classes and we discovered that they contain code Java 1.5 (*e.g.*, templates) and, unfortunately, at the moment when the experiment was performed, the MEMORIA meta-model could store only design information for systems written in Java 1.4 (or earlier). This is the reason why the two classes are missing from the model and, consequently, from the data source layer too. In the meantime both the tools and the MEMORIA meta-model have been updated, yet, the parsing infrastructure still has some liabilities that do not allow yet for these entities to be correctly extracted. Yet, this encountered problem is purely a technical issue. The thorough analysis that we performed has shown that the approach is conceptually sound.

Browsing manually the entire source code of the *CentraView* enterprise application is nearly

BUPT

impossible due to the size of the source code. Thus, we performed only a partial verification which does not reveal new causes for an improper mapping of classes into the data source layer.

4.4.3 Capturing the accesses between the object-oriented and the relational paradigms

The goal of this section is to answer the last two questions formulated at the beginning of this chapter regarding the reliability of the introduced approach.

At this phase, for the first three enterprise applications, we performed a manual investigation in order to find out all the accesses from the classes belonging to the data source layer to the tables from the relational database and we compared the results with the ones provided by the introduced model of the applications. The goal of the comparison is, like in the previous case, to discover if there are false positives (*i.e.*, accesses from a class to tables erroneous identified by the approach) and false negatives (*i.e.*, accesses from a class to tables which were not identified by the approach) among the accesses from classes to tables.

Within the *TRS* application we did not find any false positive and negative regarding the accesses from classes to tables. This applies also to the accessed-by relations captured by the model and stored into the *TableSchema* class.

Within the *KITTA* application we found several false negatives and a false positive. Analyzing again manually the accesses from the classes which are on the list of false negatives we discovered that those classes access as embedded strings tables which do not exist in the relational database of the application. The introduced extractor of design information from enterprise applications retains only the accesses to existing tables and, consequently, in the model we will not find accesses to tables which are not part of the involved database. Regarding the false positive (an access to a table from a class which does not appear in the source code) we discovered that it is in the model due to the fact that the access is present in a comment from a method of the class. But this is not a serious problem because removing comments from the source code is a trivial task (for example, the approach presented in [77] removes comments before searching for duplications in the source code).

Within the *Payroll* and *CentraView* enterprise applications (an incomplete manual investigation performed for the last one) we found also several false negatives which are due to the same cause as within the *KITTA* enterprise application.

Concluding Remarks. In order to validate the reliability of our model for representing enterprise applications we performed several experiments on the mentioned four case-studies. The performed experiments:

- bring to the front one of the major problems of enterprise applications: the source code of such applications might contain SQL statements embedded as strings which access entities from the relational database which do not exist, most of the time these errors

BUPT

being discovered at runtime. With a minor enhancement, our approach would allow to discover the aforementioned type of accesses.

- reveal that in order to minimize the false positives in the model it is mandatory to remove the comments from the source code before constructing the model.

- show that it is possible to obtain the model of an enterprise application written in Java 1.5, but those entities that use Java 1.5 facilities (*e.g.*, templates) will not be captured by the model. This is a limitation of the MEMORIA meta-model for representing object-oriented systems, a limitation which is a temporary one.

4.5 Direct Applications of the Meta-Model

If the main goal of the experiments from the last section (Section 4.4) was to present information regarding the reliability of the extracted models of enterprise applications according to the DATES meta-model, the goal of this section is to present several types of specific information extracted from enterprise applications. These are intended to give a brief overview regarding the design quality of the analyzed applications. For each of the presented information we dedicate a paragraph.

4.5.1 Finding entities that belong to more than one layer

As stated in [27] by Fowler, one steady rule an enterprise system should follow is that its domain and data source layers should never depend on the presentation layer. DATES allows us to detect entities which have a double identity (belong to more than one layer), according to a lightweight approach of layers'identification presented in Section 4.1.3. This information provided by DATES allows the identification of problematic entities which should be refactored in order to increase the level of understandability and maintainability of the application.

Classes	KITTA	TRS	Payroll	CentraView
Data Source	9	10	16	837
Presentation	25	47	77	748
Data Source,Domain	9	0	0	0
Data Source,Presentation	4	10	2	659

Table 4.9: Distribution of Classes into Layers.

Findings in *KITTA* As Table 4.9 reveals, *KITTA* contains 9 classes classified by our algorithm presented in Section 4.1.3 as belonging into the data source layer, 25 into the presentation layer and a massive mixture among the data source layer and the other layers.

BUPT

```
private javax.swing.JButton getButton_show() {
 if (ivjButton_show == null) {
  try {
   ivjButton_show = new javax.swing.JButton();
   ivjButton_show.setName("Button_show");
   ivjButton_show.setOpaque(false);
   ivjButton_show.setText("Show");
   Font f = new java.awt.Font("Arial", 1, 12);
   ivjButton_show.setFont(f);
   ivjButton_show.setBounds(224, 396, 85, 23);
   // user code begin {1}
   // user code end
  } catch (java.lang.Throwable ivjExc) {
    // user code begin {2}
    // user code end
    handleException(ivjExc);
  }
 }
 return ivjButton_show;
}
```

Figure 4.16: Method getButton_show of class ShowReservation.

Findings in *TRS* As we present in Table 4.9, the *TRS* enterprise application contains 10 classes belonging to the data source layer and 47 classes belonging to the presentation layer. But according to the mentioned table, it seems that all the classes from the data source layer break the fundamental rule of enterprise application's design: *Data source layer should not depend on the presentation layer [27]*.

For example, within the data source layer we find the *ShowReservation* class, which is derived from the *JDialogClass* belonging to the *javax.swing* presentation package. This class contains several attributes whose type denote a presentation (*e.g.*, *JButton ivjButton_show*, *JLabel*, *JTable*, *JScrollPane*) and several attributes whose names embed several SQL commands (*e.g.*, *sq1*, *sq11*, *sq12*). In Figure 4.16 we show the source code of the method *getButton_show* of class *ShowReservation*, a method which belongs to the presentation layer. As we can notice, the goal of this method is to return a button with some specific characteristics. But the class *ShowReservation* contains several methods belonging to the data source layer, one of them being the method *search_person_car* from Figure 4.17, whose goal is to retrieve data from the relational database. Due to the fact that the two presented methods have a total different purpose (among them there is no cohesion), the maintenance of this class is hampered. We took a deeper look at this class and we found out that it was detected as being also a *God Class*. In this case our design analysis for finding the design entities that belong into more than one layer is helpful in order to specify concretely the causes of the fact that a class is a *God Class* (*i.e.*, it is a *God Class* because it serves both for building the user interface and for

BUPT

retrieving data).

```
public Vector search_person_car
              (String prenume,String nume,String card_nr)
{
 Vector rc, p = new Vector();
 try {
  ...
  String myDB ="jdbc:odbc:Driver...";
  Connection db = DriverManager.getConnection...;

  sql2="SELECT * FROM Car_reservation WHERE " +
       "(First_Name=? and Last_Name=? and Card_nr=?)";
  PreparedStatement pst2 = db.prepareStatement(sql2);
  pst2.setString(1,prenume);
  pst2.setString(2,nume);
  pst2.setString(3,card_nr);
  ...
  }
  ...
 } catch(SQLException ex) ...
 return Results_vector;
}
```

Figure 4.17: Method search_person_car of class ShowReservation.

Findings in *Payroll* In the *Payroll* application we found only two classes which belong to more than one layer (*i.e.*, data source and presentation). We took a closer look at those classes and found that one class is responsible for retrieving information for all the employees from the company and also for exporting the extracted information into an XML file. The goal of the other class is to test if some particular data is stored correctly in the database and the results of the test are displayed into the presentation.

Findings in *CentraView* In this application we found almost all the classes from the data source layer as belonging also to the presentation layer (659 from 837, according to Table 4.9). But, surprisingly, regarding the observation we made during the presentation of the findings from *TRS* (*i.e.*, a class which belongs into more than one layer has good chances to be identified as affected by the *God Class* design flaw), the *CentraView* contains only 25 *God Classes*. Thus, in order to find out if the mentioned observation might be false, we performed a manual investigation and discovered that almost all the classes that belong into the data source layer are highly coupled, and thus cohesive, and in most of the cases the classification of the classes from the data source layer into the presentation layer is due to the fact that they contain many *System.out.println* statements for reporting that some exceptions regarding the communication with the persistence part occur.

BUPT

4.5.2 Finding the Interactions with Databases

Finding the database interactions in methods and classes plays an important role in the maintenance and evolution of an enterprise application. This is important for the following reasons:

- from the relational point of view, it is important to know precisely if a particular table is used/not used in the source code and, if used, where exactly the operations which access it take place. We need this information in order to be able to perform several structural refactorings (*e.g.*, [2, 3]) upon different tables, refactorings which imply the modification of the source code that interacts with the tables.

- from the object-oriented point of view, knowing which classes and methods interact with the database is important, on one hand, because in that fragment of the source code database administrators can be involved in order to maximize the performance of the communication and, on the other hand, it can bring us information regarding the persistent data within the application, this information providing us with significant information about the modeled business domain.

The proposed meta-model according to the information extracted from the relational, respectively, object-oriented part of an enterprise application allows us to find different pieces of information regarding the interactions between the two existing parts. Next, we present some of them.

Information regarding the usage of tables in the source code For every existing table we can find out if it is used/not used in the source code and, if used, all the classes and methods that store/retrieve data in/from it. We can also find the type of the operations which are performed upon it (*e.g.*, select, update, insert, delete). We illustrate with respect to the mentioned four case-studies different information from the model extracted by DATES.

	KITTA	TRS	Payroll	CentraView
Tables	10	10	12	217
UnusedTables	3	2	1	102

Table 4.10: The Number of Unused Tables.

Information extracted from *KITTA* In terms of columns, the largest table has 7 columns and the smallest 3 (*i.e.*, the values of the NC metric vary between 3 and 7). As Table 4.10 presents, the source code of this application does not access all existing tables in the database, *i.e.*, it accesses only 7 tables from 10.
Regarding the *accessed by* entities from the source code associated to tables, we find one table (*Users*) accessed by a single class (*User*), and one other table (*Child*) accessed by 3 classes (*Group, Child, ParentApplicationRequest*). All other tables are accessed by 2 classes (table *Kindergarten* accessed by classes *Comandament* and *Kindergarten*; table *Intervals* accessed

BUPT

by classes *Comandament* and *TimeInterval*). This can be interpreted as follows: the table named *Child* is the hardest to change, as more classes depend on it.

Information extracted from *TRS* In this application we find 5 tables having more than 20 columns (table *Flight_reservation_comp* having NC equal to 30; table *Car_reservation* having NC equal to 23) and being aware that tables with a large number of columns reveal a design problem [3], we perform an in-depth analysis at the level of the columns. This analysis reveals that this application has a reduced degree of normalization – we find within the database, for example, 6 columns named *Address*, 7 columns named *City*, 6 columns named *First_Name*, most of these columns belonging to the same 6 tables.

Regarding the usage of tables in the source code we find, as Table 4.10 presents, 8 tables from 10 accessed in the object-oriented part of the application, 5 of them being accessed by only one class and the rest of them by 3 classes.

Information extracted from *Payroll* In this system we found that out of the 12 tables, all except one are accessed from the source code. Two of these tables are used by 2 classes, while the rest are accessed by a single class each.

The size of the involved tables in terms of columns (NC metric) in this application varies from 23 (table *employee*) to 3 (table *inflation*), most of the tables having less than 10 columns (9 tables).

Information extracted from *CentraView* This last case-study has a database consisting of 217 tables, among which only 113 are used in the source code. In terms of columns, the biggest table has 31 columns and there are 39 tables which have more than 10 columns. These results provide us with the following interpretation: first, it is surprising to find that almost half of the tables are apparently not used at all in the source code. These unused tables are a common phenomenon in systems that evolve over a long time, but they represent a significant confusion factor for someone who does not know the system and wants to understand it by looking first at the database. A second interpretation of the results concerns the tables with many columns: there is a significant number of "wide tables". In many cases this is a sign that a *normalization* of these tables is recommendable. At the same time, in many cases these tables are storing information related to essential concepts of the system.

Regarding the *accessed by* relations, we find table *individual* accessed by 15 classes from the source code and DATES, by providing us with the classes which manipulate data in/from in, helps within the process of maintaining this table. Beside this table accessed by a lot of classes, we find other 6 tables (*e.g.*, *entity*, *activity*, *module*, *opportunity*, *addressrelate*, *customfieldmultiple*) accessed by more than 5 classes. The interpretation of these findings is the following: modifications of the structure for the seven aforementioned tables should be kept to a minimum (especially table *individual*), as modifications of these tables would impact a large number of classes.

BUPT

4.5.3 Visualizing how Tables are Accessed in Code

In order to provide an overview regarding the accesses of tables in the source code we define next two visualizations, namely: *Tables Accesses* and *Distribution of Operations*.

1. Tables Accesses The main goal of this visualization is to provide an overview regarding the usage of tables in the source code in correspondence with their size. Thus, the polymetric view which represents a table is enriched with three metrics, as we show in Figure 4.18.

The width of the rectangle represents the number of statements performed within the source code upon the represented table. The height is associated to the number of classes whose methods perform operations upon the table. The fill (color) of the rectangle is a gray gradient (*i.e.*, it goes between white and black), and it is related to the number of columns in a table (*i.e.*, the more columns the darker the rectangle).

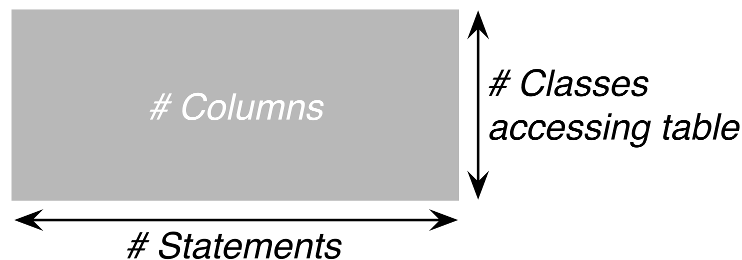

Figure 4.18: Tables Accesses. Representation.

Having in front of us the defined visualization for a given system may bring valuable information in the reengineering process. Next, we explain different situations that may appear (*i.e.*, the nature of a polymetric view representing a table) and their impact:

- a small rectangle whose size is just a pixel – the table is not used in the source code.
- a square – every class that accesses the table performs only an operation on the table.
- a rectangle with an excessively large height – the operations upon the table are spread in many classes and, consequently, the impact of changing the table would affect many entities in the source code. If in this situation the color of the polymetric view is near black (*i.e.*, a large value associated to the number of columns the table has), there is a high probability that the table is affected by the Multipurpose Table design flaw [3].

2. Distribution of Operations In order to have a detailed view about the types of operations that are performed upon the existing tables we create another type of visualization (see Figure 4.19). This visualization represents each used table as a rectangle containing 4 squares, each square having its color mapped to one of the following SQL statements: insert, select, update and delete. The side of each square is equal to the number of SQL statements whose type is represented by its color. Consequently, if the polymetric view of a table:

BUPT

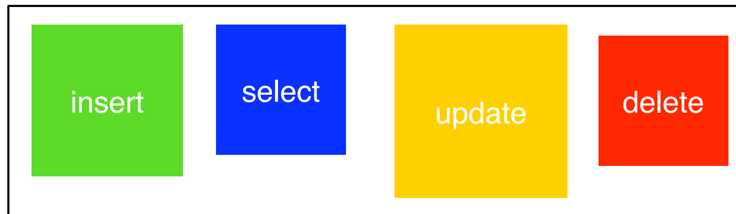

Figure 4.19: Distribution of Operations. Representation.

- contains only a blue square (*i.e.*, upon the table only select statements are performed) it means that the table contains only constant data which is usually maintained by the database administrator.
- contains only a green square (*i.e.*, upon the table only insert statements are performed) it means that the information is only recored in the table but not manipulated directly from the source code. In this situation it would be interesting to find out who are the *real consumers* of the stored data within the table. The same is applicable if the polymetric view contains also a red square (delete statements).
- contains 2 squares whose colors are green and blue then it means that the information, after it is introduced in the table, is never modified from the source code of the application.

We implemented these visualizations within the DATES module using the existing tool JMONDRIAN within the IPLASMA platform for visualizing data. We present in Appendix A their complete implementations.

Visualizations in *KITTA* In Figures 4.20 and 4.21 we present how the existing tables are used in the source code, according to the defined visualizations.

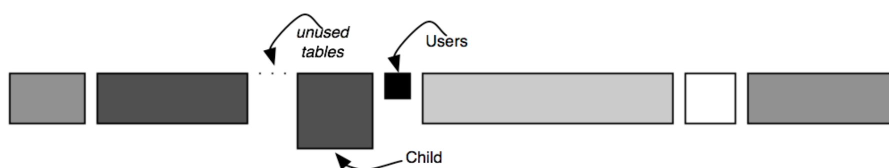

Figure 4.20: Tables Accesses in KITTA.

From the first visualization (Figure 4.20[4]) we notice that the application:

- contains three unused tables, namely *Groupp*, *MoveReq* and *EducatorReq*.
- there are 3 tables which are accessed once from different classes: *Users* from only one class, *Intervals* from 2 classes, and *Child* accessed from 3 classes. In case of *Child*, each

[4]Length, width and color correspond to the metrics specified in Figure 4.18.

BUPT

of the classes performs only a single operation on the table. Another interesting remark here is that the *Users* table has the largest number of columns (*i.e.*, the largest value for the NC metric).

- most of the tables are accessed from 2 classes.

Regarding the types of operations performed upon the tables the second visualization (Figure 4.21) shows the following:

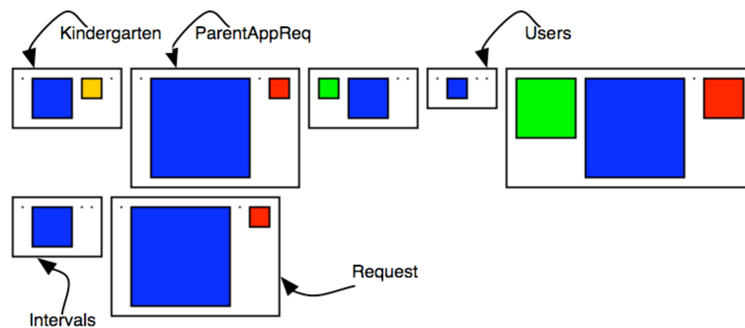

Figure 4.21: Distribution of Operations in KITTA.

- there are 2 tables whose information is only read (*i.e.*, *Intervals* and *Users*). In this situation it may be possible that only the database administrator inserts the information into tables and, consequently, manages the users that are allowed to work with the application.
- upon the table *Kindergarten* only select and update operations are performed.
- upon the tables *ParentAppReq* and *Request* only select and delete statements are performed.

Visualizations in *TRS* From the first overview obtained (Figure 4.22) we find out that:

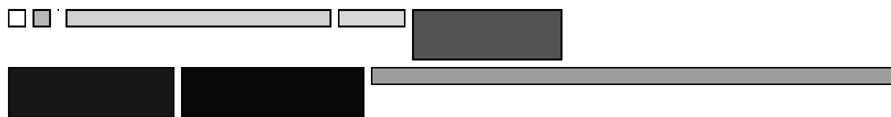

Figure 4.22: Tables Accesses in TRS.

- 2 tables are unused in the source code (*e.g.*, *Flight_reservation_comp* and *PasteErrors*).
- there are 3 tables which are accessed in 3 classes, while the rest of the tables are accessed from only one class.

BUPT

- we have a table which is accessed from a single class frequently (the rectangle with the highest width). We took a closer look at the class and found out that a lot of duplications occur, some of them being related to the *where* clauses of the SQL statements. We illustrate an example related to this .

```
String sql;
if(f=="")
 sql="SELECT * FROM Table1 Where ((City=?) and
  (#"+Check_in+"# BETWEEN Data_in AND Data_out) and
  (#"+Check_out+"# BETWEEN Data_in AND Data_out) and
  (Address Like ?) and (Level_of_service =?) and
  (Price <= ?))";
else
 sql="SELECT * FROM Table1 Where ((City=?) and
  (#"+Check_in+"# BETWEEN Data_in AND Data_out) and
  (#"+Check_out+"# BETWEEN Data_in AND Data_out) and
  (Address Like ?) and (Level_of_service =?)
  and (Price <= ?)"+f+")";
```

From the second visualization (Figure 4.23) related to the existing types of SQL statements for manipulating data in tables we notice that:

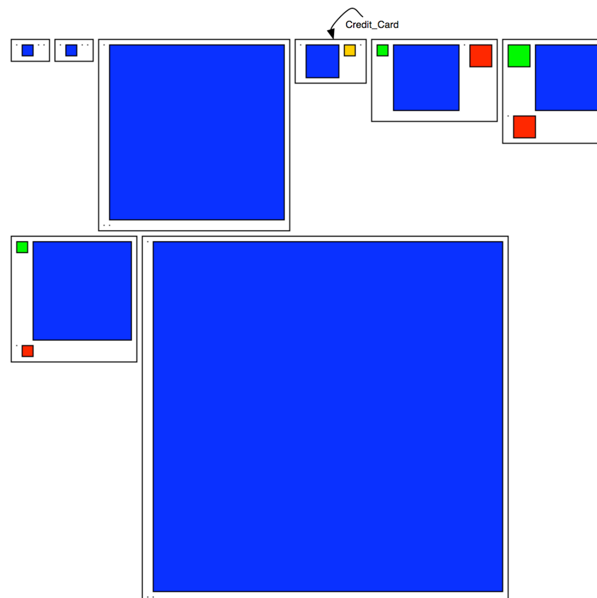

Figure 4.23: Distribution of Operations in TRS.

- there are 4 tables storing data which is only read within the source code (*i.e.*, only select operations are performed).

BUPT

- there are 3 tables whose stored information after the insertion is only selected and deleted and never modified.

- there is a table *Credit_Card* whose number of rows is never modified from the source code – only select and update operations performed upon.

Visualizations in *Payroll* From the Tables Accesses Visualization (Figure 4.24) we find out that:

- there is a table which is not accessed in the source code.

- most of the tables are accessed by only one class.

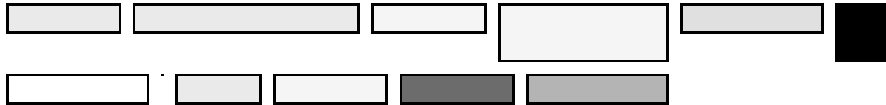

Figure 4.24: Tables Accesses in Payroll.

Regarding the Distribution of Operations in Payroll we notice from the Figure 4.26 the following:

- upon more than a half from the tables accessed in the source code (6 from 11) all the four types of SQL operations are performed.

- from table *employee* information is only read.

- the information from the table *salaryhistory* is never read, only inserted, updated and deleted.

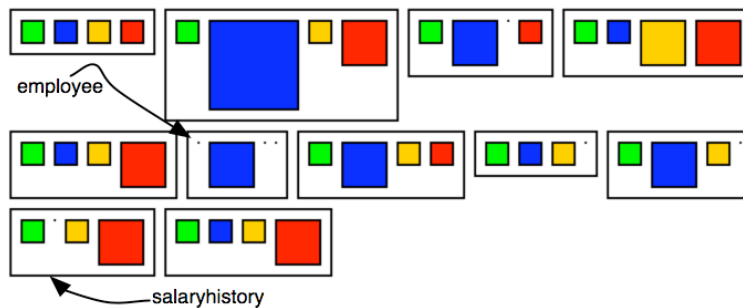

Figure 4.25: Distribution of Operations in Payroll.

BUPT

Visualizations in *CentraView* From Figure 4.26 which contains information regarding the usages of tables in the source code we discover that:

- there is a large number of unused tables.
- the most used table in the source code is table *individual* upon which 33 SQL statements are performed from 15 classes.
- there is a group of tables which are heavily used in the source code – *e.g.*, *emailmessage*(8 SQL operations), *mocrelate*(19 SQL operations), *activity*(26 SQL operations).

In Figure 4.27 we present the defined visualization obtained by analyzing the CentraView enterprise application. According to it we find out that the application contains:

- a large number of tables whose stored information is only read in the source code – *e.g.*, *activity*.
- many tables whose information is only read and modified (we find only blue and orange squares) – *e.g.*, *expense*, *opportunity*.
- tables whose rows are never updated.

BUPT

Figure 4.26: Tables Accesses in CentraView.

BUPT

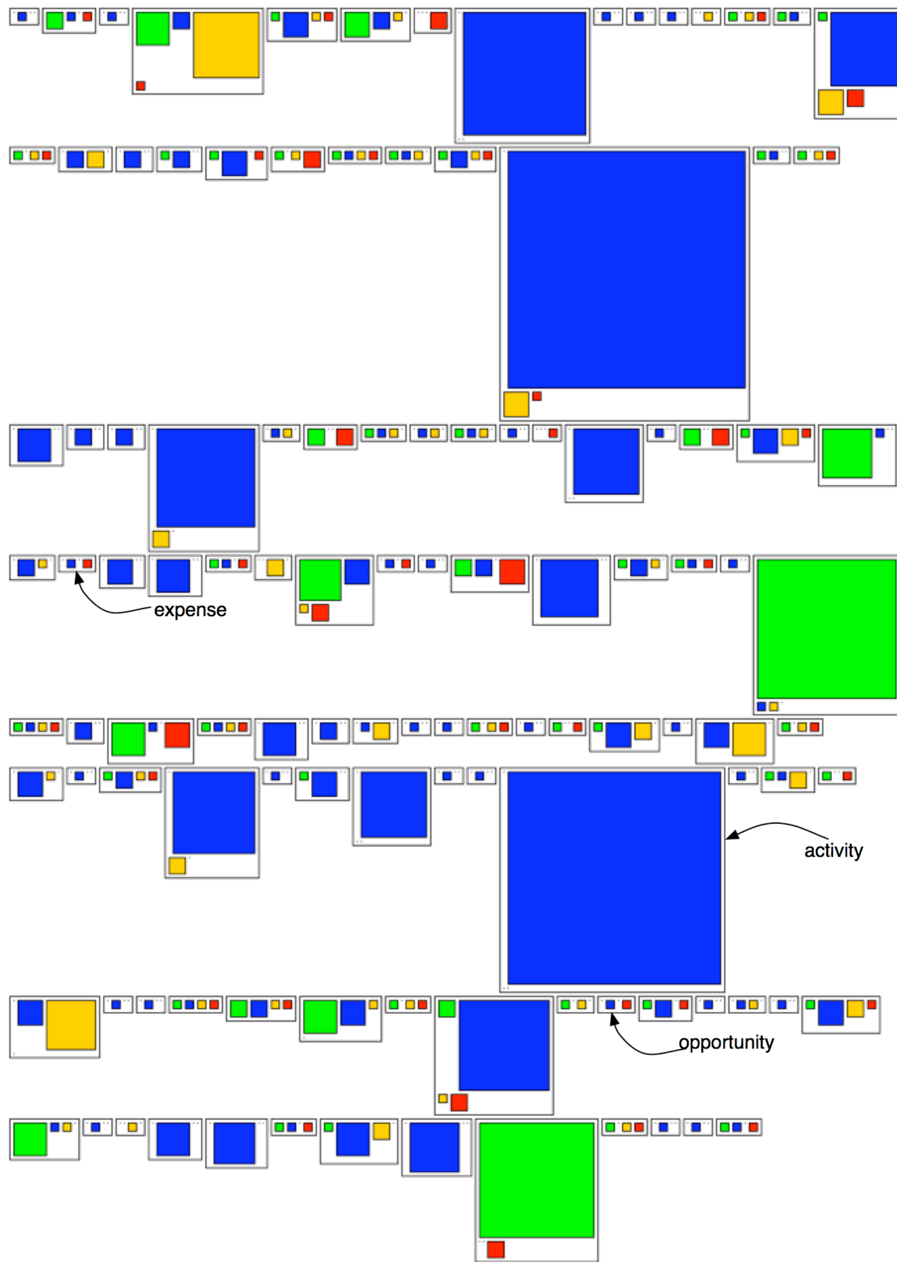

Figure 4.27: Distribution of Operations in CentraView.

BUPT

5

Relational Discrepancies Detection

One specific concern of data reverse engineering is to retrieve constraints which are not explicitly declared in the database schema but verified in the source code. In this chapter we propose a novel approach for detecting the relational discrepancies between database schemas and source code in enterprise applications, as part of the data reverse engineering process. Detecting and removing these discrepancies allows us to ensure the accuracy of the stored data as well as to increase the level of understanding of the data involved in an enterprise application.

5.1 Relational Discrepancies

A database schema models mainly tables, columns and the various relations and constraints that exist between the data stored in the database's tables. Most, if not all, of the relations are the reflection of the business logic (behavior) which is modeled in the source code. Therefore, in a normal case there should be a correlation between the constraints defined on tables in the database schema and the co-usage of those tables in the source code. But there is nothing to guarantee that the source code and the database schema stay synchronized. When this is not the case we say that a *relational discrepancy* appears between the source code and the database schema. We identified two possible cases of such relational discrepancies:

1. *Missed Constraint.* Two or more tables are constantly accessed together (jointly) in code but there is no constraint defined in the database schema to suggest that. For example, in Figure 5.1 we see that both classes B and C access always together tables T1 and T2, as if these tables would be related by a foreign key. But looking in the database schema we don't find such a constraint between the two tables. The missed constraint hampers the accuracy as well as the understanding of the stored data.
2. *Incomplete Data Usage.* The second case of relational discrepancy appears when there is an explicit constraint defined in the database schema, but it is disregarded systematically in the source code, *i.e.*, the classes don't access the tables jointly. Looking again in Figure

BUPT

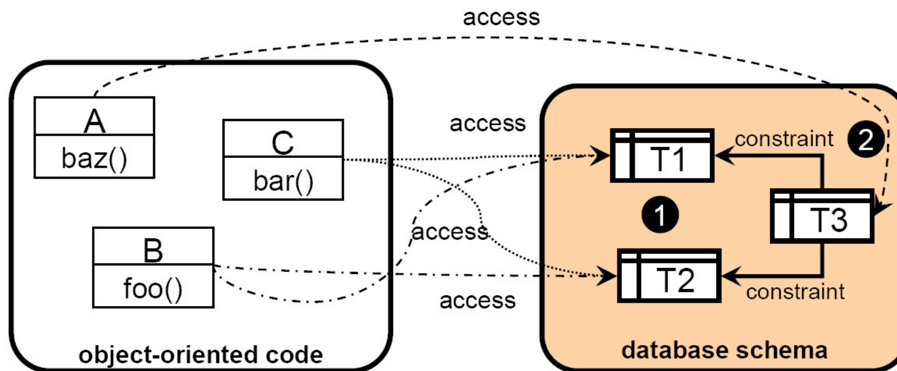

Figure 5.1: Two types of relational discrepancies.

5.1 we notice that while there is a constraint defined among T1 and T3 no class is accessing both T1 and T3 together (directly or indirectly); instead, class A which accesses table T3 does it without using the correlated data found in T1. This means that the database might contain unused data which increases the necessary effort for understanding the stored data.

While both cases of discrepancy are very interesting, we focus in this thesis on the first discrepancy case, *i.e.*, *Missed Constraint*.

5.1.1 Missed Constraint: An Example

In order to explain more clearly this case of relational discrepancy, let us consider a simple example. Considering an application that manages a library, let's assume that all the information regarding the library are stored in a relational database called `Library` which contains, among others, the following three tables, as depicted in Figure 5.2:

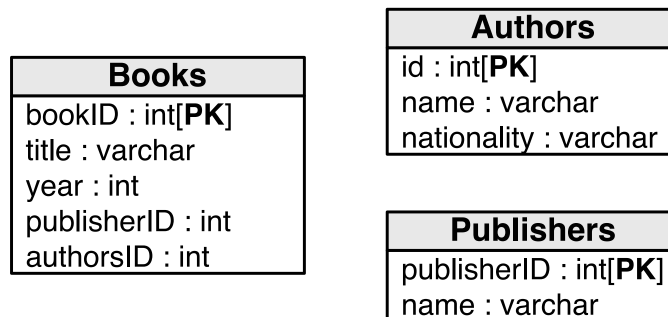

Figure 5.2: Library database schema, with no explicit constraints among tables. ([PK] marks the primary keys).

- `Books` – stores information about the existing books within the library.
- `Authors` – contains data about the authors of the existing books.

BUPT

- `Publishers` – the rows of this table contain information about all the existing publishing houses.

Every table has a primary key and no foreign key.

By looking carefully at Figure 5.2 we see that, although apparently there are no correlations among the three tables, two apparent links can be noticed in table `Books`, links that lead to the other tables – *i.e.*, columns `publisherID` and `authorsID`, meaning that every book has an author and a publisher.

Unfortunately, when the database was created, the two integrity constraints (foreign keys) that would ensure the integrity of the data stored within the table `Books` were not specified. Because these foreign keys are missing, we can't know for sure if fields `publisherID` and `authorsID` from table `Books` are correlated with fields in table `Publishers` respectively `Authors`. While this issue appears rather obvious in this example, this is mainly because:

- the example is simple and the domain of the modeled data is well-known.
- the naming of correlated fields helps us.

But in general, finding these missing correlations between tables, by looking exclusively at the database schema is hardly possible. Yet, knowing these integrity constraints is in practice an essential clue towards understanding and maintaining a large-scale system. That's why, in this thesis we aim to detect these missing constraints, by looking at the way tables are accessed in the source code.

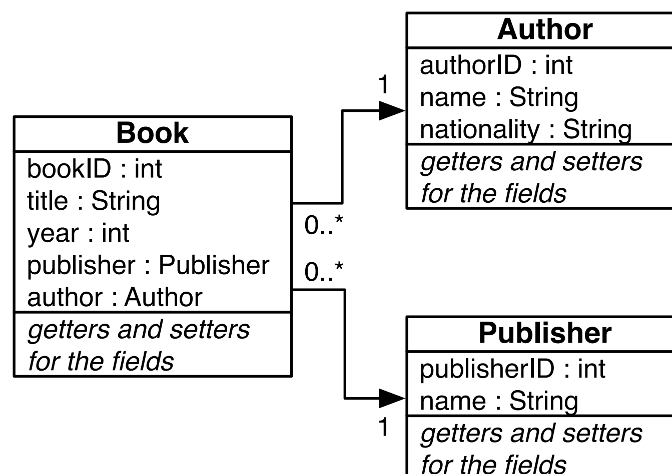

Figure 5.3: Domain Classes for Library example.

Going on with the example let's take a look at the source code, where these tables are used. In an enterprise application we find *domain objects* [27] which contain data from the

database, objects which are stored/retrieved in/from the database by means of dedicated classes belonging to the data source layer – *e.g.*, `BookDS` and `PublisherDS` (Figures 5.4 and 5.5). By applying Keller's *Foreign Key Aggregation* pattern [34] for our `Library` example we define the domain classes depicted in Figure 5.3.

We notice that a separate class was created for each table from the database, and that the `Book` class has references to the `Author` and `Publisher` classes.

Now (re)creating a `Book` object from the database implies retrieving data from the three aforementioned tables (Figure 5.2). In general, if more than two tables are involved there are two main approaches for doing this [34]:

- perform a single join between all the tables.
- access separately each table and in the end set the references between the created objects.

Each of the two approaches has its own pros and cons, but in practice both are widely used, and oftentimes we can encounter a combination of the two idioms.

For the sake of showing all the facets of the problem, in our example we use such a *combined* solution for creating a `Book` object, as seen in Figure 5.4. By looking at method `getBook` we first notice the `SELECT` statement and the join between the `Books` and `Authors` tables. This shows us that the `authorsID` column in table `Books` is *de facto* a foreign key on table `Authors`. But this is unfortunately an *implicit constraint*, *i.e.*, an information that we can't find explicitly in the database schema (Figure 5.2)! Second, we notice that a `Book` domain object also needs a reference to a `Publisher` domain object, and it obtains it by calling the `getPublisher` method from `PublisherDS` classs (see Figure 5.5); more precisely, the `Publisher` object is created separately, by accessing table `Publishers` from class `PublisherDS`, and then its reference is attached to the `Book` object. This reveals an even more hidden dependency, *i.e.*, the one between the `Books` and the `Publishers` tables. Again, the database schema didn't make this correlation explicit.

In conclusion, starting from a class/method belonging to the *data source layer* it is possible to identify the group of tables that are used jointly, – called *group of related tables* both when the tables are accessed directly and when the correlation between tables is indirect (*e.g.*, via a method call).

The ability to recover from the source code all these correlations among the database tables, would provide us with useful information regarding the integrity constraints the existing schema contains, or should contain; and, which is most important, it helps us to understand the (oftentimes implicit) dependencies between the data manipulated in the system.

Related Work. We dedicate this paragraph to a briefing of several representative solutions that fall in (or are closely related with) the discovery of hidden relationships among relational entities.

BUPT

```
class BookDS {
 public Book getBook(int id) {
  ...
  query = "SELECT * from Authors AS A,Books AS B" +
      " WHERE B.bookID=" + id +
      " AND A.id = B.authorsID";
   rs = statement.executeQuery(query);

   // create book object and set fields
   Book aBook = new Book();
   aBook.setBookID(rs.getInt("bookID"));
   aBook.setTitle(rs.getString("title"));
   aBook.setYear(rs.getInt("year"));

   // create author object and set fields
   Author theAuth = new Author();
   theAuth.setAuthorID(rs.getInt("id"));
   theAuth.setName(rs.getString("name"));
   theAuth.setNation(rs.getString("nationality"));
   // add author reference to book object
   aBook.setAuthor(theAuth);

   // get publisher object and set the
   // reference to it for the created book
   aBook.setPublisher(
      new PublishterDS().getPublisher(id));
   return aBook; }
}
```

Figure 5.4: The BookDS data source class.

A first category of approaches related to this topic is the one whose input is provided exclusively by the involved relational database. An example of such approach is *Referential Integrity Utility for IBM DB2 Cube Views* [22], a tool which detects missing primary keys, missing foreign keys, nullable foreign keys and generates data definition language to add the necessary DB2 informational constraints. The main difference between this approach and our approach resides in the fact that our approach will provide us those discrepancies depicted from the usage of data from the source code. This way, we ensure that the data from the tables are indeed used together. An instrument for finding hidden relationships based only on database schema (*e.g.*, using comparisons of names) might bring us inaccurate results.

Another category of approaches which helps in order to find out implicit data constraints among the involved database is based on program slicing, like the one proposed in [12]. This approach is applied upon enterprise applications which communicate with relational databases

BUPT

```
class PublisherDS {
 public Publisher getPublisher(int id) {
   ...
   query = "SELECT * from Publishers " +
           "WHERE publisherID=" + id;
   rs = statement.executeQuery(query);

   // create publisher object and set fields
   Publisher aPublisher = new Publisher();
   aPublisher.setID(rs.getInt("publisherID"));
   aPublisher.setName(rs.getString("name"));
   return aPublisher; }
}
```

Figure 5.5: The PublisherDS data source class.

via embedded SQL (*e.g.*, SQL code written between `EXEC SQL` and `END-EXEC`), approach which is not suitable for enterprise applications which communicate with relational databases by SQL statements embedded in a string that are executed by methods like executeQuery().

In this work we assume that persistence is provided by a relational database (e.g., MySQL) and the communication between the entities belonging to the data source layer and the relational database is performed by executing SQL commands as embedded strings. But nowadays there are also enterprise applications where the persistence layer is manipulated by frameworks like Hibernate [5]. In this context we want to emphasize that our approach can be applied upon enterprise applications which use such frameworks, but in this case the information regarding the connections between the object-oriented part and the relational part will be extracted also from existing configuring XML files (and, consequently, we need to develop and use a different model loader for extracting information from the source code).

5.2 Detection of Relational Discrepancies

Before describing the technique developed for detecting the *Missed Constraints* discrepancy, we first need to summarize the main entities and their interrelationships involved in an enterprise application, as shown in Chapter 4.

Figure 5.6 shows that on the source code side we mainly deal with *methods* that belong to *classes*. The main source code relation that we are interested in is that a method can *call* (or respectively be *called by*) other methods. On the database side, we find *tables* that have a number of *columns*. Columns have a name and can also have the special property of being *primary key* for a table, *i.e.*, the values for that column are unique within the table. Concerning the database relations, we are interested in the constraints defined by means of *foreign keys* among the columns belonging to different tables. On top of everything, the relation that we

BUPT

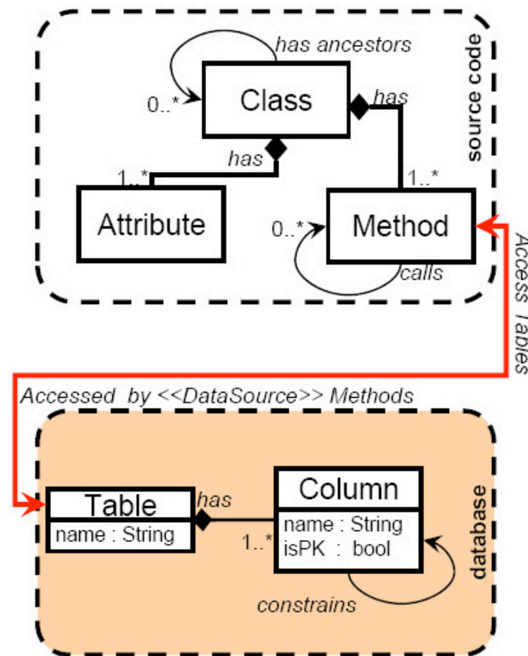

Figure 5.6: The meta-model of main entities that connect the source code with the database.

are most interested in, is the one that connects the two "worlds", *i.e.*, the source code and the database. This connection appears in methods – belonging to the *data source layer* – that access one or more tables by building and executing queries on the database (as seen in Figure 5.4).

Based on this model, we can now describe the detection process, which implies the following sequence of steps:

- *Step 1: Build groups of related tables*, based on the accessed tables information found in each method belonging to the *data source layer*.
- *Step 2: Detect* de facto *interrelationships (constraints) among the related tables*, based on a set of heuristic rules for the matching between pairs of tables.
- *Step 3: Identify missed constraints*, by comparing the *de facto* constraints identified in the the source code, with those explicitly defined in the database schema.

5.2.1 Build the Groups of Related Tables

The main goal of this first step is to find the tables which are related by the fact that they are accessed within the same usage context, at the source code level. As we have seen in the `Library` example, both the `BookDS` and the `PublisherDS` classes are directly or indirectly related with all the three tables.

BUPT

A key element of our approach is that we consider not only the direct usages of tables, but also the *indirect* ones. Consequently, for each method M belonging the the *data source layer* identified according to the algorithm presented in Section 4.1.3 the group of related tables (GRT) is built as follows:

- All tables accessed *directly* from M, by means of queries built and executed within the method are added to GRT(M).
- For each data source layer method **called from** M[1], we add to GRT(M) all the tables *directly* accessed from the called method.
- For each data source layer method which **calls method** M, we add to GRT(M) all the tables *directly* accessed from the method that calls M.

For our initial example, both GRT(`BookDS`) and GRT(`PublisherDS`) will contain all three tables, *i.e.*, `Books`, `Authors` and `Publishers`.

In Figure 5.7 we show how we implemented in the DATES tool the group of related tables associated to the method *aMethod*. In Line 2 we get all the methods that the investigated method calls and in Line 4 we get all the methods that make use of it. We create a new group which contains all the distinct methods from the previous groups (Line 6). Applying the filter *model function* to the group of operations ensures that we take into consideration only the methods defined in the analyzed application – *i.e.*, excluding the library methods (Line 7). Please remark that all the groups and filters used till Line 7 were previously created in the iPlasma environment. In Line 8 we keep from the mentioned methods only those which are part of the data source layer. We continue the implementation (Line 9) by getting all the distinct tables accessed from the group of methods created in Line 7. Finally (Line 10) we add the tables directly accessed by the inspected method. The filter *is datasource* and the group *accessed tables* are part of the DATES module.

5.2.2 Detect "de Facto" Constraints

The fact that a pair of tables are used from the same usage context does not necessarily mean that there is a relational constraint between the two. Therefore, the next step is to discover possible interrelationships among the tables found in the GRT's built for each data source method, during the previous phase.

Oftentimes there is no relational constraint between a table from a GRT group and any other table from that group; furthermore, obviously, constraints might not be bidirectional. For example, table `Publishers` is not connected to table `Books`, it is only the other way around, the relation is going only from `Books` to `Publishers`.

In order to discover the relational constraints from the source code, we use a set of heuristic rules related mainly to the *naming* of tables and columns. Aiming to discover interrelationships

[1] Of course, recursive calls are excepted.

BUPT

```
GroupEntity getRelatedTables(AbstractEntity aMethod)
{
1 GroupEntity mCalled;
2 mCalled = aMethod.getGroup("operations called");

3 GroupEntity mCallingMe;
4 mCallingMe = aMethod.getGroup("operations calling me");

5 GroupEntity related;
6 related = mCalled.union(mCallingMe).distinct();
7 related = related.applyFilter("model function");
8 related = related.applyFilter("is datasource");
9 related = related.getGroup("accessed tables").distinct();

10related.addAllDistinct(aMethod.getGroup("accessed tables"));

11return related;
}
```

Figure 5.7: Building the group of related tables.

between all the related tables, we take each *pair of tables* from each GRT that has been built in the previous step.

Considering a pair of tables ($\mathrm{T1}$, $\mathrm{T2}$) we claim to have found a *constraint from* $\mathrm{T1}$ *to* $\mathrm{T2}$ (*i.e.*, $\mathrm{T1}$ depends on $\mathrm{T2}$), if there exists at least one pair of columns ($\mathrm{CT1}$ - belonging to table $\mathrm{T1}$, $\mathrm{CT2}$ - from table $\mathrm{T2}$) having the same type that satisfies one of the following naming conditions:

- the name of $\mathrm{CT1}$ contains or is equal, – based on a case insensitive comparison – with the name of table $\mathrm{T2}$ and $\mathrm{CT2}$ is (part of) the primary key of table $\mathrm{T2}$. In our example, `authorsID` column in `Books` contains the name of the `Authors` table, `Books.authorsID` and `Authors.id` are of the same type, and `Authors.id` is the primary key of table `Authors`.

- $\mathrm{CT2}$ is (part of) the primary key of table $\mathrm{T2}$ and the name of $\mathrm{CT1}$ contains or is equal with the name of $\mathrm{CT2}$, based on a case insensitive comparison. In our example, the `publisherID` column in `Books` has the same type and the same name with the primary key of table `Publishers`.

Remark. Clearly, our approach relies heavily on naming rules. From this point of view, the heuristics above appear to be rather fragile. Yet, both our practical experience and our case studies (see Section 5.3) suggest that in most projects the database schema relies heavily on naming conventions. These naming rules might be slightly different than the ones we used,

BUPT

but this is one point where our technique can be easily adapted to comply to further such heuristics.

Still, we have to admit that unfortunately we encounter sometimes hidden dependencies among columns, that can't be captured using these naming rules (*e.g.*, two columns called `StudentID` and the other `PersonID`). For such cases we could enhance the correlation heuristics, by relying on the relations found in an ontology, – like WordNet[2] – as done in several recent reverse engineering approaches [69, 68].

5.2.3 Identify the Missed Constraints

The input of this phase is provided by the group of the pairs of tables for which in the previous step we found an interrelationship. Some of these found relations might have already been defined within the schema of the database. Consequently, in order to find out only the discrepancies between database schema and source code, we are going to inspect each possible relation from the group and if the relation has already been defined, we are going to remove it from the group.

For this last step we rely on the model of the involved database, more precisely on the constraint relations (see Figure 5.6) in order to find out if the inspected constraint has been defined in the schema. If the relation is missing then we assume that we have successfully identified a *Missed Constraints* relational discrepancy.

5.3 Evaluation of the Approach

In order to evaluate the approach we have conducted different experiments on the four case studies whose main characteristics are presented in Section 4.3.

	KITTA	**TRS**	**Payroll**	**CentraView**
Data Source Classes	9	10	16	837
Columns	52	186	89	1281
Primary Key	10	9	12	170
Foreign Keys	0	0	0	28
Missed Constraints	3	0	7	219

Table 5.1: The Identified Relational Discrepancies.

The results presented in this section are obtained automatically with the usage of DATES (introduced in Section 4.2) upon these case studies. In this context we want to emphasize that all the 3 steps of the algorithm for detecting missed constraints were implemented within DATES. Moreover, we brought an improvement to the visualization Tables Accesses presented in Section 4.5.3 – we introduced edges which represent the found missed constraints.

[2] http://wordnet.princeton.edu

BUPT

Besides the possibility of visualizing the found missed constraints, DATES allows us to save them in a text file, from where they can be manually applied on the schema of the database within the analyzed enterprise application.

As a secondary effect we provide for each table the group of *related tables* computed based on the first step of the presented algorithm. This group allows us to find for each table the tables which are used in the same usage context.

In Table 5.1 we present the number of the identified missed constraints between database schema and source code. Next, for each system we are going to discuss the obtained results.

5.3.1 The *KITTA* Application

As we can see in Table 5.1, the *KITTA* application contains 10 primary keys and none foreign key. By applying our introduced technique we find 3 missed relations between the existing tables, as Figure 5.8 reveals.

```
1. T:Priorities         C:PREQ_ID_PK
   T:ParentAppReq       C:PREQ_ID_PK

2. T:ParentAppReq       C:REQ_ID_PK
   T:Request            C:REQ_ID_PK

3. T:Priorities         C:PREQ_ID_PK
   T:Request            C:REQ_ID_PK
```

Figure 5.8: KITTA: The discovered related tables.

In order to find out whether the discrepancies were correctly identified we performed a manual investigation. Identifying correctly the relations between the entities from the existing relational database in an enterprise application means that:

- among the entities that were classified as being related by foreign keys there are no false positives (*e.g.*, entities erroneously identified by the approach as being related).
- among the entities that were not classified as being related by foreign keys there are no false negatives (*e.g.*, entities which are related and were not identified by the approach as being related).

We performed an investigation through all the 25 methods which were identified by our approach as belonging to the data source layer in order to accomplish the mentioned tasks. We found out that for most of the methods (19) the group of related tables is composed by none or just one class and, consequently, only the remaining 6 groups were further inspected.

Three groups contain two tables – *e.g.*, (*ParentAppReq*, *Priorities*), (*Priorities*, *Request*) and

BUPT

the first relation is contained into two groups. From these groups of related tables we found out that each request has a priority. Each of the remaining three groups contain the same four tables: *ParentAppReq*, *Priorities*, *Request* and *Child*. From this group we depicted the second missed constraint from Figure 5.8. Based on our analysis all the found constraints appear to be real (*i.e.*, we did not detect any false positives).

Regarding the false negatives, at first sight it is surprising that between the *Child* table and the rest no relationship was found, at least among the *ParentAppReq* table. Consequently, we took a look at the columns of the tables and we discovered that the *Child* table contains the *age*, *FNAME* and *LNAME* columns while within the *ParentAppReq* instead of having a reference into the *Child* table, we have the duplicated columns *CHILD_FNAME* and *CHILD_LNAME*. Anyway, our approach reports that the tables are used together but is not able to detect a missed constrained due to the mentioned redundancy of data.

5.3.2 The *TRS* Application

In this application our approach did not identify any missed constraints. Like in the previous case, we performed an in-depth investigation in order to ascertain if the findings are correct.

The investigation reveals that the application contains 24 methods belonging to the data source layer. For most of the methods (23) the group of related tables is composed by only one table. We discovered only one method whose group of related tables contains 3 tables: *Car_reservation*, *Flight_reservation* and *Hotel_reservation*. As suggested by their names, the tables contain information about different types of reservations a person may apply for and among them there is no missed constraint.

5.3.3 The *Payroll* Application

As we can see in Table 5.1, the *Payroll* application contains 16 classes identified as belonging to the data source layer, the involved data is stored in 12 tables and there are 12 primary keys (one primary key for each table) and no foreign key. By applying our introduced technique we obtain 7 discrepancies between database schema and source code.

During the manual investigation performed, we browsed through each of the 12 existing tables in order to find out which interrelationships exist among them. We have discovered 8 interrelationships, 7 of them being discovered by the approach. For example, the approach reports that

- every stored history regarding salaries is attached to an employee (table *salaryhistory* should have a foreign key *employee* that refers column *Id* from the table *employee*).
- every stored evaluation belongs to an existing employee (there is a table called *evaluation* which should have a foreign key *employee* that refers column *Id* from the table *employee*).
- every employee has a stored history regarding his position (table *positionhistory* should have a foreign key *employee* that refers column *Id* from the table *employee*).

BUPT

Regarding the missed constraint which was not discovered by our approach, we found out that the false negative is due to the fact that one of the tables involved in the unreported constraint was *not used in the source code*. Probably, it would have been good to apply in this case the *Drop Table* refactoring pattern presented in [3].

5.3.4 The *CentraView* Application

This application, as already mentioned, is a large-scale enterprise application which has 1527 classes, 836 belonging to the data source layer and the involved data is stored among 217 tables. By applying our approach, the tool found 219 missed constraints between database schema and source code. Like in the previous case, in order to find out whether the identified constraints were correctly identified we performed a manual investigation.
Due to the fact that *CentraView* is significantly larger than *Payroll*, a manual investigation of the 217 tables is not feasible. Thus, we performed a "smart" manual investigation based on a subset of the interconnected tables, chosen using the following strategy: as displayed in Table 5.1, *CentraView* contains 28 foreign keys, *i.e.*, 28 relations involving 20 tables. We manually removed from the database schema the 28 relations aiming to see if by applying our approach, we will find these initially defined interrelationships among the 20 tables.

By applying our technique, we found only half (14) of the initial set of 28 relational constraints initially defined on the database schema. Thus, 14 relations were missing from the reported list and, therefore, we performed an in-depth analysis in order to find out the causes.

We discovered that 11 out of the 14 initial relations that our approach did not detect, are missing from the reported list due to the fact that 8 of the tables involved in these foreign key were simply not used in the source code! Probably the *Drop Table* pattern [3] is again recommendable in this case.

Regarding the remaining 3 undetected relations, we found out that they were not found because our matching heuristics did not apply for the involved columns. Yet it is worth mentioning that the 3 foreign keys were all defined between the same two tables, *i.e.*, the *search* resp. the *individual* table. The three undetected relations are foreign keys on columns *OwnerBy*, *CreatedBy*, *ModifiedBy*, all three referring to column *IndividualID*. By analyzing these column names, we are confident that our approach can be enhanced by enriching the matching heuristics with additional relations extracted from an ontology, like we suggested at the end of Section 5.2.2.

Apart from that "smart" manual investigation we took a closer look on the 219 reported constraints that should be applied upon the schema of the database and we present next some of them.

In Figure 5.9 we present some found discrepancies. Due to a high degree of understanding provided by the names of tables and columns, we can quickly find out the meaning of most discovered missed constraints. Thus, we expect more tasks to belong to a project (1), to

BUPT

```
1. T:task            C:ProjectID
   T:project         C:ProjectID
2. T:invoice         C:billaddress
   T:address         C:AddresID
3. T:ticket          C:individualid
   T:individual      C:IndividualID
4. T:attachment      C:MessageID
   T:emailmessage    C:MessageID
5. T:attendee        C:IndividualID
   T:individual      C:IndividualID
6. T:attendee        C:ActivityID
   T:activity        C:ActivityID
7. T:attendee        C:ActivityID
   T:applicationform C:ActivityID
8. T:uidlist         C:accountID
   T:emailaccount    C:AccountID
```

Figure 5.9: CentraView: Some discovered related tables.

encounter more invoices at the same address (2), an individual to have more tickets (3), an emailmessage to have none or more attachments (4) and an individual to attend more activities (5).

Regarding the three relations 5 - 7, we noticed that all of them should be applied upon the table *attendee* (*i.e.*, it is possible to discover more discrepancies that should be applied on the same table - this explains why it is possible to have 219 missed constraints which should be applied, due to the fact that at least 8 tables are unused, on less that 217 tables).

It is quite easy to see that relation 5 has been correctly identified but at a first sight relations 6 and 7 are ambiguous because they should be applied on the same column *ActivityID* which should refer two different tables (*activity* and *applicationform*). In this case we took a closer look on the two last tables and we discovered that table *activity* contains information regarding an activity (*e.g.*, type, priority, status, owner, DueDate, CompletedDate) and table *applicationform* contains also related information regarding an activity (*e.g.*, salaryactual, assessmentneg) and even the primary key for table *applicationform* is called *ActivityID* – this case is nothing but a common practice for a foreign key to refer more than one table. Moreover, searching in the report provided by the introduced approach, we discovered that when a foreign key should refer column *ActivityID* from table *applicationform*, it always should refer also column *ActivityID* from table *activity* – we encountered this situation 5 times.

The last relation from Figure 5.9 is an example of a missed constraint whose meaning cannot be easily discovered but within the provided report there are only a few relations of this type (10).

BUPT

5.3.5 Identified Relational Discrepancies according to the percentage of using the tables together in the source code

The previous experiments shows the applicability of the introduced approach. Despite the fact that the results are satisfactory enough, they may be affected by a weakness: when we find missed constraints among large enterprise applications it is hard to determine if between the pair of tables (T, R) there is a missed constraint relation (detected according to the algorithm presented in Section 5.2) if table T is heavily used in the source code and it appears only once together with table R in the groups of related tables build from each method belonging to the data source layer.

We consider advisable to consider that among the tables (T, R) there might be a missed constraint if and only if from all the groups of related tables where T appears table R appears too in a percent greater than a threshold provided to the approach by the database maintainers.

	KITTA	**Payroll**	**CentraView**
10%	3	7	219
20%	3	7	219
30%	3	7	201
40%	3	7	185
50%	3	7	169
60%	2	7	155
70%	2	7	131
80%	2	7	107
90%	2	7	70
100%	2	7	63

Table 5.2: Relational Discrepancies according to percentages.

Consequently, we enhance our approach with a parameter whose goal is to narrow the discovered missed constraints. The value of the parameter represents the percentage of having every table T, from all the groups of related tables where table T appears, together with table R. In Figure 5.10 we present how we compute for each pair of tables (T, R) the mentioned percentage.

The value contained by the *relatedTables* parameter embeds all the groups of related tables build in the first part of the algorithm presented in Section 5.2 (Line 1). In Line 4 we iterate through all the groups of related tables. If the current group contains table T we increase the value of the *is* variable (Line 7) and we check also if the group contains also table R. If it does, we increase also the value of having the two tables together in the current group (Line 9). Finally, we return the percentage obtained by dividing the value of the computed variables *is* and *isTogether*.

BUPT

```
1  double getPercent(TableSchema T, TableSchema R,
                   Collection relatedTables) {
2  Iterator it = relatedTables.iterator();
3  int is=0, isTogether=0;
4  while(it.hasNext()) {
5  GroupEntity g = (GroupEntity)it.next();
6  if(g.getElements().contains(T)) {
7   is++;
8   if(g.getElements().contains(R))
9    isTogether++;
   }
  }
10 return ((double)isTogether)/is*100;}
```

Figure 5.10: Computing the percentage of having (T, R) together.

We decide that among (T, R) there is a missed constraint identified by the last steps of the introduced algorithm if and only if the percentage computed according to the implementation from Figure 5.10 is greater or equal with the value of the parameter received by the approach.

In Table 5.2 we present the number of missed constraints found by the enhanced approach. For each application where there were initially detected missed constraints we perform more experiments, varying the introduced threshold every time with 10 percents.

In the *KITTA* application from the 3 missed constraints initially found only 2 of them remain if we set the threshold to 100%. In the *Payroll* application all the missed constraints reported previously are present regardless of the introduced threshold. In the *CentraView* system more than a half of the initially found missed constraints are obtained using a threshold lower that 70%.

Conclusions. The main features of the introduced approach are:

- it allows us to identify automatically the related tables from which data are retrieved in the source-code.
- based on the related tables, it identifies possible interrelationships between relational tables (*e.g.*, Missed Constraints).

BUPT

6

Objectual Meaning of Foreign Keys

While foreign keys are an important means for indicating relations within the persistent data, oftentimes, by looking solely at the database schema, it is impossible to determine the exact nature of these relations. In this chapter we propose a novel approach for determining a refined understanding of the relations among the persistent data, by correlating the information about foreign keys extracted from the database schema with the way the data are used in the source code. We applied our approach on several case-studies and the encouraging findings indicate that the approach provides relevant information towards recovering an accurate data model of an enterprise application.

6.1 Objectual Meaning of Foreign Keys

In an enterprise application the two most reliable sources of information for knowledge recovery are the database schema and from the source code. For each of these there is a good and a bad part:

- In a RDBMS, while the **database schema** usually captures well the key model elements – as this is mainly what is kept persistent – the ability to express *relationships* (*e.g.*, using foreign keys) is rather weak [34], being almost impossible to express precisely the type of that relation (to differentiate between a part-whole and a hierarchical specialization relation).

- Concerning the **source code** the good part is that the semantics of relationships that can be expressed is much richer than in the case of database schemas (*e.g.*, the explicit inheritance relation, or the containment relation expressed using instance variables [17]). Yet, at the source code level we have another problem: the elements of the business model are "cluttered" with implementation details, and this makes it hard to distinguish between the source code elements (*e.g.*, classes and methods) that correspond to the business model and those that are strictly used to support the concrete implementation.

BUPT

Consequently, in order to recover the relationships among business model elements, and thus recover the understanding of the modeled domain, we need to combine information both from the database schema and the source code. In this context, we propose a two step approach to recover these relations:

1. **Step 1**: We use the information (*i.e.*, tables, columns, foreign keys) from the database schema to focus (*i.e.*, reveal) the classes that store the data from the database. We calls these classes *Data Keepers*.
2. **Step 2**: We extract from the source code the relationships among *Data Keepers* and use them to refine the semantics of the foreign key constraints defined among various tables.

6.1.1 Relations between Tables and Classes

Considering a table `T`, there can be several ways it can be linked with the source code:

- is not accessed in the source code, probably it is useless to understand its structure and function. We consider that between `T` and source code there is a N (**N**ot Used) RELATION and usually we should apply the Drop Table Refactoring Pattern [3].
- is used in the source code. We say that between `T` and source code there is a U (**U**sed) RELATION and consequently it is worth to take a closer look at `T`.
- is used in the source code and there are one or more classes whose instances store data from `T`. We say that those classes are Data Keeper Classes [1] related to `T` and between `T` and the object-oriented part there is a S (**S**toring Data) RELATION. Knowing for `T` the names of its related Data Keeper classes might provide us with additional information about the meaning of the data within table `T` and the methods of the classes might provide us with information regarding the function of the same data.

Finding the previous relations is not possible unless we take into consideration the interactions between source code and existing tables. When we extract relations associated to entities of the relational paradigm (*e.g.*, tables, columns, constraints) by taking into account information from the object-oriented source code, we consider that the retrieved relation has an *objectual* meaning. Obviously, all the N, U and S RELATIONS have an *objectual meaning*.

6.1.2 Refined Semantics of Foreign Keys

One important relation that connects the elements (*e.g.*, tables) of a relational database schema are the foreign keys constraints. But, as we mentioned before, the semantics of this relation is very vague as it can denote anything from a simple dependency to an aggregation relation between the concepts represented by the two tables.

Assuming that we have two tables `TFrom` and `TTo` connected by a foreign key which relates data from `TFrom` to data from `TTo`, and two Data Keeper classes – `CTo` storing data from table `TTo` and `CFrom` storing data from table `TFrom` – the data from the two tables connected

[1] we do not use the Data Transfer Object [27, 56] term because a Data Keeper might also have functionality and might also access the database. Every DTO is a Data Keeper.

BUPT

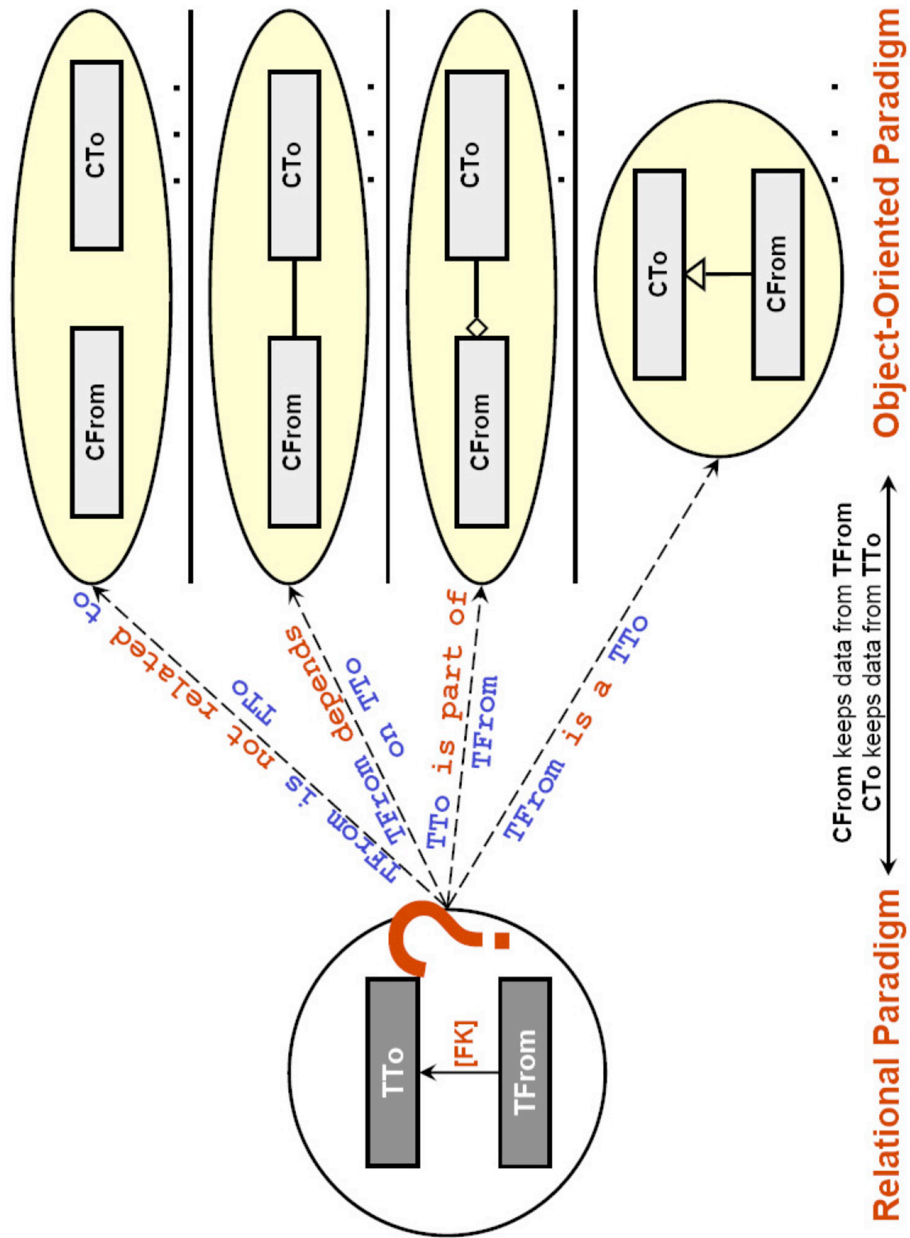

Figure 6.1: The objectual meaning of a foreign key, as revealed in the source code.

BUPT

by the foreign key may be used in the source code in different ways, as Figure 6.1 presents. If we know how the data from the tables is used in the source code, we would infer different objectual meanings of the foreign key, as follows:

- if between `CFrom` and `CTo` is no interaction, we consider that this situation reflects also the relation among the tables, being possible that table `TFrom` contains unused data.
- if methods from `CFrom` call methods from `CTo`, we consider that, as well as among the Data Keeper classes, between the two tables related by the foreign key there is a D (**D**ependency) RELATION, the objectual meaning of the foreign key being that data from `TFrom` *is used together* with data from `TTo`.
- if between `CFrom` and `CTo` there is a relation of aggregation, we consider that the A (**A**ggregation) RELATION among the Data Keeper classes is available also for the tables related by the foreign key, having the objectual meaning that the data stored in `TTo` *is part of* the data stored in `TFrom`.
- if `CFrom` is derived from `CTo`, we objectual interpret that the data stored within `TFrom` *is a* specialized data of `TTo` and, consequently, an I (**I**nheritance) RELATION occurs.

In this chapter we propose a mechanism which provides us with the objectual meaning of foreign keys (the D, A and I RELATIONS) [49], as we consider that those relations can provide us with significant information which increases the level of fulfilment of the goals of DRE. The approach specifies, as well, the cause/causes (*e.g.*, classes from the object-oriented part) which establish the provided relations. The objectual meaning of foreign keys is provided if and only if the tables connected by the constraints have corresponding Data Keeper classes in the source code and, consequently, we need first to find out the relations between tables and source code (*e.g.*, the N, U and S RELATIONS mentioned above).

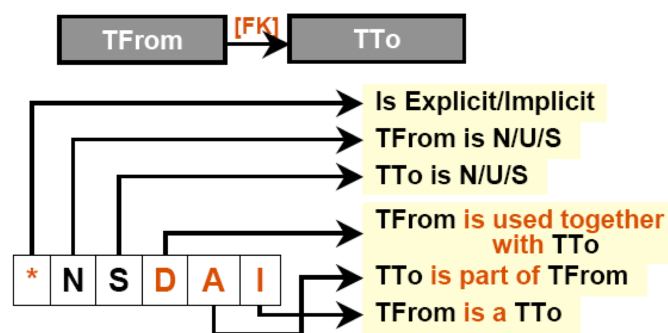

Figure 6.2: Objectual meaning of foreign keys.

Concretely, our approach provides us for each foreign key (defined explicitly in the schema of the database or implicitly, inferred from the usage of tables in the source code, as presented in Chapter 5) a description of its objectual meaning in 6 characters, as depicted with Figure 6.2, as well as the cause/causes that establish the description. The first character reveals the

BUPT

type of the characterized relation (explicit or implicit). The next two characters specify the relation between the two tables and the object-oriented part (the N, U or S RELATIONS). The last three characters appears in the characterization, obviously, if and only if the relation among the tables and the object-oriented part has the S type (for each table, we have at least a Data Keeper in the system). A foreign key might have none or all of the three objectual meanings.

Remark. Often we encounter situations where the database schema relies heavily on naming conventions and in this context, if there it is a foreign key between a table called *Person* and a table called *Student*, we expect an I objectual meaning and, thus, we can consider useless our approach. But it may happen that this assumption is false, the data from the tables being totaly unrelated by an I RELATION, and our approach can confirm/infirm every assumption of this type we make.

6.2 The Approach

Based on the model we introduced in Section 4, we present below how we discover *objectual meaning* of foreign keys in enterprise applications.

6.2.1 Discovering the N, U and S Relations

As we previously mentioned, knowing the N, U and S RELATIONS is a mandatory step for fulfilling the goal of our approach. Based on the model described above, discovering the N and U RELATIONS between tables and source code requires counting the number of methods belonging to the data source layer which access each inspected table. Consequently, if the number is zero, it means that we encounter a N RELATION, otherwise we have a U RELATION.

In order to find out if between a table `T` and the object-oriented part there is a S RELATION, as well as the causes that ensure the relation (*e.g.*, the Data Keeper classes corresponding to `T`), we perform the following three operations:

- we detect the Data Keeper classes from the inspected enterprise application.
- for each Data Keeper we find the related tables (tables from which the Data Keeper stores/retrieves data)
- based on the previous two phases, we discover for each table the set of corresponding Data Keepers.

Detect the `Data Keeper` Classes. The detection of a `Data Keeper` is done according to the algorithm presented in Figure 6.3. Because Data Keepers store the data transported from/to the database their detection is done by visiting all the methods belonging the data source layer and considering all the classes that are used in these methods as:

BUPT

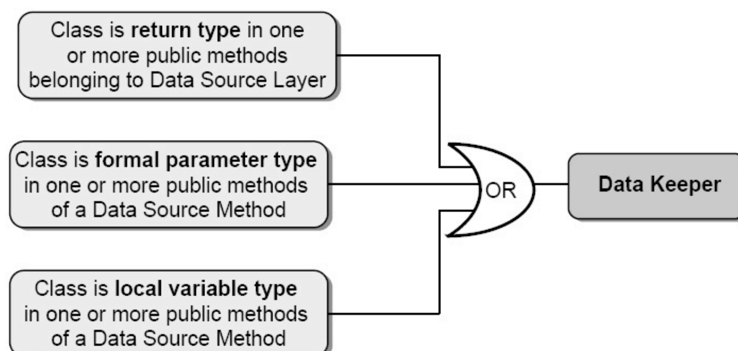

Figure 6.3: Detect a Data Keeper Class.

- return types – for the case where data extracted from the database must be passed to the other layers of the system.
- parameter types – for the case where data is passed to the data source layer in order to be stored in the database.
- types of local variables – for the case where multiple data having the same type are extracted from the database and passed to the other layers embedded in a predefined collection.

Find the `related tables` for each keeper. We consider for each `Data Keeper` the following tables as being related:

- the tables accessed from each method belonging to the data source layer that *calls* methods from the current Data Keeper.
- the tables accessed from each method belonging to the data source layer *being called by* methods from the current Data Keeper.
- the tables *directly accessed* by the methods from the Data Keeper.

The first two operations ensure that we find the tables to/from which data belonging to the current Data Keeper are stored/retrieved. The last operation might look confusing, as we would expect for each Data Keeper Class to belong only to the domain layer and not also to be responsible for storing/retrieving data in/from tables. But note that we reverse engineer legacy enterprise applications where clear initial design intentions might have become "blurred" during implementation. Thus, we expect at least some of these Data Keepers that access also the database tables to reveal signs of design flaws.

Detect the corresponding `Data Keeper` classes for each table. Finding for each table the set of related `Data Keepers` is nothing else than processing the results obtained from the

BUPT

previous two steps. Consequently, for each table `T`, we have to iterate through the existing Data Keeper classes and for each current keeper (CDK) we have to see if it is related to `T`. If so, we add CDK into the set of `Data Keepers` classes corresponding to table `T`. If the size of this set is greater than zero, it means that between the table and the object-oriented part we encounter a S RELATION and its cause it provided by the computed set itself.

6.2.2 Discovering the D, A and I Relations

Discovering the D, A and I RELATIONS among tables which are related by a foreign key relies on two categories of input data, the first containing the foreign keys and the second being provided, for each table, by the set of related `Data Keepers` created within the previous phase.

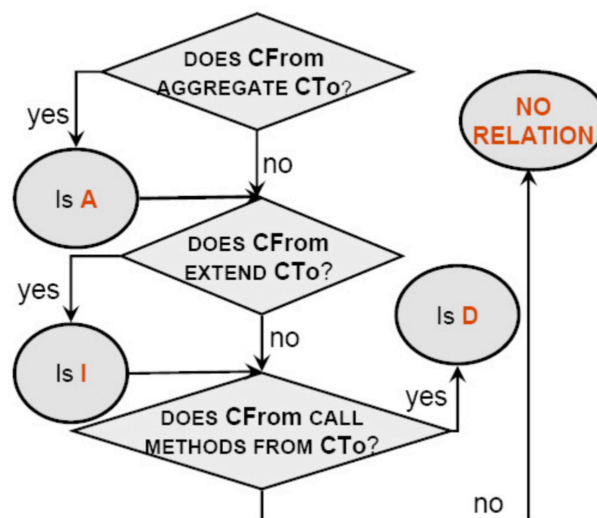

Figure 6.4: Discover the D, A and I RELATIONS.

In Figure 6.4 we present the defined algorithm for discovering the D, A and I RELATIONS, algorithm which has to be applied on each possible pair of classes (`CFrom`, `CTo`), `CFrom` belonging to the group of Data Keepers related to `TFrom` and `CTo` belonging to the group of Data Keepers related to `TTo`.

If `CFrom` has an attribute whose type is `CTo` it means that the data stored in table `TTo` *is part of* the data stored by `TFrom` and, consequently, between the two involved tables there is an A RELATION. The algorithm continues by checking if `CFrom` extends `CTo`, regardless of the fulfilment of the previous condition (*i.e.*, it is possible for class `CFrom` to aggregate class `CTo`, as well as to be descended from `CTo`). If the second condition is reached we encounter an I RELATION. Finally, we check if there are calls from methods belonging to `CFrom` to methods belonging to `CTo` – this condition establishes a D RELATION between the involved

BUPT

tables (TFrom, TTo) related by a foreign key.

If none of the previous conditions are satisfied, it means that the object-oriented part of the enterprise application contains data provided by tables TFrom and TTo related by a foreign key, but this relation is not visible in the source code. It might be possible that only a part of the data contained by table TFrom is used.

As we can notice, we may encounter only an A RELATION between TFrom and TTo. At first sight this is not a problem, but in a well-designed enterprise application we expect a class to aggregate another class in order to make use of the services provided by the aggregated class. Thus, if between tho tables there is only an A relation this probably means that among the keepers that generate the relation we might encounter a violation of the encapsulation principle. Yet, correlating design flaws with the types of relations between tables in the involved database is also beyond the scope of this thesis.

6.3 Evaluation of the Approach

In order to evaluate the approach we have conducted different experiments on the same four case studies whose main characteristics are presented in Section 4.3. Next, for each system which contains explicitly or implicitly tables related by foreign keys we are going to present the obtained objectual meanings of foreign keys. As it is noticeable, the *TRS* application is not part of this experiment because it does not satisfy the mentioned condition regarding the foreign keys. The objectual meaning of foreign keys have been detected automatically by using the DATES module which contains the implementations of the algorithms for their detection.

6.3.1 The *KITTA* Application

Among all the 3 pairs of tables connected by foreign keys detected in this application we find DIA RELATIONS. By looking at the causes that generate them we notice many Data Keeper classes connected to each table and most of the keepers are connected to all of the three tables involved in the relations.

The fact that most of the Data Keepers store data from all the tables bring in front of us the possibility of having design problems into the schema of the database like duplicated information (in particular, we discover this during the experiments performed for the validation of the discrepancies 'detection) or multipurpose columns.

We inspected in-depth the connected Data Keeper classes for all the tables in order to find out information about the business model. We found out that more than a half of the existing tables have 5 or 6 associated Data Keepers. This situation is responsible for different undesirable facts as follows:

- the same primary data is revealed by more than one object.

BUPT

- the operations involving the same data are spread into multiple classes.

For example, the *Child* tables has 5 corresponding Data Keepers: *Child*, *DataBaseTricks*, *Group*, *Kindergarten* and *ParentApplicationRequest*. As Figure 6.5 presents, instead of encapsulating a field having the *Child* type into the *ParentApplicationRequest* class, the developers include within the *ParentApplicationRequest* class most of the existing attributes from the *Child* class.

```
class Child
{
 private Connection connection;
 private String firstName;
 private String lastName;
 private int age;
 private int id;
 private int parentId;
}

class ParentApplicationRequest
{
 protected TimeInterval interval;
 protected Vector kindergartens;
 protected int childAge;
 protected String childLastName;
 protected String childFirstName;
 private int parentRequestId;
 private Statement statement;
}
```

Figure 6.5: Duplicated fields related to the Child table.

6.3.2 The *Payroll* Application

As we can see in Table 5.1, the *Payroll* application contains 112 methods identified as belonging to the data source layer, the involved data is stored in 12 tables and there are 12 primary keys (one primary key for each table) and no foreign key. By applying the technique we introduced in [50] we obtain 7 missed constraints (*i.e.*, 7 foreign keys) between database schema and source code.

Our approach reports that among the 7 constraints there are only D RELATIONS, which mean:

- each table involved in the classified constraint is used in the source code and, moreover, there is at least a Data Keeper class associated to each table.

BUPT

- the data stored in each involved pair of tables is used together (*e.g.*, when we increase the value stored in the *MonthlySalary* column belonging to table *salaryhistory* we expect this operation to be performed for a particular *employee*).

Performing an in-depth investigation regarding the tables used in the generated report, we find out that table *employee* is used together with other 6 tables (*e.g.*, *salaryhistory*, *evaluation*, *training*, *training_employee*, *presence* and *positionhistory*) and table *training_employee* is used together with table *training*, the last relation being indirectly related to table *employee*. Due to the fact that every stored data is related to table *employee*, we can say that this table tends to centralize most of data in the database belonging to the Payroll system. In general this centralization phenomenon can be either a database design flaw, or an inherently central table, which is very important for the understanding of the entire database.

In order to find out whether the relations were correctly identified we performed a manual investigation. Identifying correctly the relations between the entities from the existing relational database in an enterprise application means that:

- among the entities that were classified as being related – by D RELATIONS – there are no false positives (*e.g.*, entities erroneously identified by the approach as being related).
- among the entities that were not classified as being related there are no false negatives (*e.g.*, entities which are related – by A or I RELATIONS – and were not identified by the approach as being related).

Within the *Payroll* we did not find any false positive or negative regarding the classification reported by our approach.

6.3.3 The *CentraView* Application

This application is a large-scale enterprise application which has 1527 classes, 3349 methods belonging to the data source layer and the involved data is stored among 217 tables. Like in the previous evaluation, before discovering the objectual meaning of foreign keys, we applied the technique introduced in [50] in order to depict the missed constraints. This step is not mandatory, but necessarily in order to find out more information about the analyzed enterprise application. We found 219 missed constraints between database schema and source code. Next, we are going to describe different found objectual meaning of foreign keys, as well as their consequences.

Among the existing 217 tables, 115 are used in the source code, 76 of them being connected to the source code by one or more Data Keeper classes. In Figure 6.6 we present some associations provided by the approach between tables and their Data Keeper classes. From the provided information, for example, we find out that:

- the data from the *literaturerequestlink* table is related to an activity, and we expect that the system contains different types of activities (class *ActivityVO* has a method *getType()*).

BUPT

```
1. Table:      literaturerequestlink
   Data Keeper:ActivityVO

2. Table:      eventregister
   Data Keeper:EventAttendeeVO

3. Table:      ticket
   Data Keeper:TimeSlipDBVO
```

Figure 6.6: Some discovered S Relations between tables and Data Keeper classes.

- *eventregister* contains registrations for a particular attendee – class *EventAttendeeVO* contains information regarding an event as well as an attendee.

- the information contained into the *ticket* table is related to a particular project (class *TimeSlipDBVO* has a method which provides the associated *projectID*, as well as information retrieved from the *ticket* table).

Among the 247 foreign keys (28 extracted from the database schema and 219 implicit) whose objectual meaning was retrieved by the introduced approach, we find 41 type A RELATIONS. This indicates that data stored in the *vendor* table *is part of* the data stored in the *payment* table, the data stored in the *project* table *is part of* the data stored in the *cvorder* table as well as the data stored in the *invoice* table, the data stored in the table *addressrelate is part of* the data stored in the *invoice* table and a *cvfile is part of* the data stored in the *literaturetable*.

We encountered 4 type I RELATIONS among tables. Thus, we expect data stored within tables *projectlink*, *customfieldscalar* and *customfieldmultiple* to be different types of *syncnote*. As we can notice, there are only a few relations of this type and we think this reflects the fact that among the existing Data Keeper classes there are only a few inheritance relations.

We encounter 2 relations having the DAI type, one of these relations being among table *user* related by a foreign key to table *individual*. In this case, a user *is an* individual containing a lot of information related to other individuals.

We find 7 relations having the D type. In Figure 6.7 we present the tables whose stored data is used together in the source code.

We notice that class *ProposalListForm* is a reported cause for more than one of the relations from Figure 6.7, collaborating with various data which belongs to different unrelated tables. Consequently, we expect this class to implement more than one functionality. The performed investigation indicated that, indeed, the class provides a lot of functionalities, being affected by the *God Class* [26] design flaw. In this case, our approach is useful, beside classifying the relations among tables, to encounter design flaws related to an improper use of the data

BUPT

```
1. cvfilefolder cvfolder

2. cvfilefolder cvfile

3. emailmessagefolder cvfolder

4. cvfilelink cvfile

5. proposallink proposal

6. emailrecipient emailmessage

7. emailrule account
```

Figure 6.7: The D Relations between tables.

stored in the database but, as previously mentioned, at this moment we consider this aspect beyond the scope of our investigation.

We find 66 plain SS RELATIONS related to the 247 foreign keys. In this case two possible situations exist:

- the `CFrom` Data Keeper related to table `TFrom` does not make use of the data stored in the referenced table `TTo` (*i.e.*, an incomplete data usage).
- the two Data Keeper classes are used together in order to process all data from the two existing tables.

The remaining constraints have not been reported as having one of the D, A or I objectual meaning because the involved tables do not have associated Data Keepers in the object-oriented part – for example, we have 39 SU RELATIONS, 30 US and UU RELATIONS among the involved tables related by foreign keys.

***CentraView* – an overall classification.** Performing an in-depth inspection of the provided relations, we found the *God Tables* within the application. For example, the most coupled table we find is table *individual*, being part of 17 relations, 3 of them being A RELATIONS and 2 being DAI RELATIONS. We think a first step in comprehending the structure of the involved tables is to take a look at the most coupled relations and our approach, by providing the nature of the relations defined as foreign keys, helps us in order to accomplish this aim.

Conclusions. The work presented in this chapter is a step forward in enhancing the understanding of persistent data, as part of the maintenance of enterprise applications. The proposed approach provides the objectual meaning of foreign keys, as revealed from the interactions between the source code and database schema *i.e.*, the **D**EPENDENCY, **A**GRREGATION

BUPT

and INHERITANCE relations. The aforementioned relations exist if and only if the involved tables have corresponding Data Keeper classes in the object-oriented source code.

The introduced features require the use of the specific meta-model for enterprise applications introduced in Chapter 4 which captures in an unitary manner the entities from the object-oriented and the relational paradigms and, most important, the relations between the two. The approach presented in this paper has shown that the strength of such a meta-model that bridges the gap between the object-oriented and the relational "worlds" facilitates a simple expression of analyses that uses information from both sides.

We have conducted an experiment in which we automatically detected the objectual semantics of foreign keys in enterprise applications. In Table 6.1 we summarize the found objectual meaning of foreign keys. The experiment reveals that by using the approach presented in this paper a significant number of foreign key constraints can be semantically refined; thus, the approach contributes to a better understanding of the database schema.

	Constraints	**Relations**
KITTA	3	3 DIA
Payroll	7	7 D
CentraView	247	7 D, 41 A, 4 I, 2 DAI

Table 6.1: Discovered Objectual Meaning of Foreign Keys.

BUPT

7

Roles-Aware Detection of Design Flaws

According to the section dedicated to the presentation of principles and patterns in enterprise applications (Section 2.4), sometimes the design of such applications is governed by a set of rules conflicting with the ones from a "regular" object-oriented application. Consequently, these conflicting rules have to be taken into consideration when we want to perform an accurate assessment of the design of such type of software systems. In this chapter we present an approach based on roles which improves the accuracy of the detection of the two well-known design flaws: Data Class and Feature Envy [26].

7.1 Roles in the Data Source Layer

In an enterprise system each design entity (*e.g.*, class, method) belongs to a layer. Furthermore, Fowler in [27] identifies and describes in the form of patterns different types of classes and methods that have (or should have) responsibilities with respect to the structure of an enterprise application or, more specific, with respect to the design of a particular layer. In this context we defined in [46] the notion of **design role** as follows:

Definition 7.1.1 (Design Role) *A* design role *is a specific responsibility that a design entity (*e.g.*, a class or a method) might have with respect to the design of a specific layer or to the collaboration between two layers within an enterprise application. A* design role *is reflected in the design entity by a suite of constraints related to its structure and/or the functionality that it provides.*

If a class or method has a particular *design role* then knowing it provides the engineer who maintains or evolves the system with additional semantical information about its place within the layer where it belongs. In Figure 7.1 we present the relations between a design entity and its (potential) design role, respectively its layer. As we can see in the figure, not every entity will have "enterprise-specific" role. The figure makes also clear that if we know the role of an entity, we also know the layer it belongs to.

BUPT

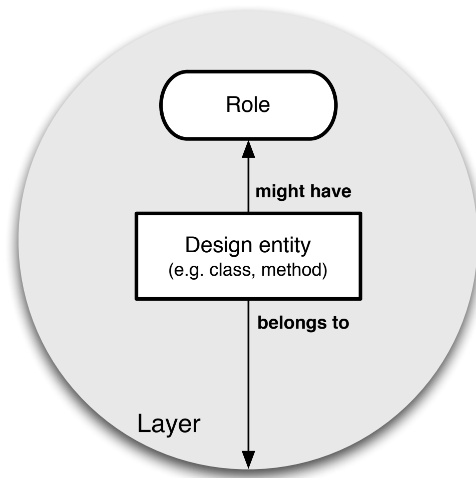

Figure 7.1: Relations between a design entity, roles and layers.

7.1.1 Design Roles and Quality Assessment

The knowledge about layers and design roles is very important both for understanding and for assessing the design of an enterprise application, at least for the following two reasons:

- Patterns of desirable and avoidable interactions in an EA are defined in terms of an entity's *affiliation* to a specific layer and/or to a specific role. For example, a class should not contain methods that belong to the presentation layer and other methods belonging to the data source layer. Or, a class that is a *Table Data Gateway* [27, 1] (that is its role) should not contain methods that belong to the domain layer.

- They help the reverse engineer of the system by providing him with *orientation points* about the interactions in the system in terms of layers and roles.

7.2 Extracting Roles in the Data Source Layer

As we presented the importance of layers and roles in the context of quality assessment of enterprise applications, an essential question arises: When these design roles are described in an informal manner, how can they be identified automatically in a given enterprise application? In this section we are going to present a suite of *role-mining techniques* for several design roles that characterize the data source layer.

7.2.1 Roles in the Data Source Layer

Before presenting the detection techniques we are first going to present the main specific roles that methods and classes may have within the *data source* layer of an enterprise application.

BUPT

Why do we focus on the *data source* layer? The reason resides in the fact that this layer is usually responsible for assuring the proper bridging of two different paradigms *i.e.*, the relational database and the object-oriented model; and it is well known that this issue raises many understanding and design quality concerns [35].

Design Roles of Methods. Design roles are revealed especially by particular patterns of provided functionality and therefore methods are the lower level design entities that can have design roles. Thus, within the data source layer, a method might have one of the following four roles: to *retrieve* (R), *insert* (I), *modify* (M) or *delete* (D) information from tables of the relational database that ensures the persistency within the enterprise application. Knowing the role of methods (*i.e.*, R, I, M, D) has multiple uses:

- it helps understanding the specific purpose of each method that interacts with the database.
- it helps localizing within the system all design fragments that have the same specific role (*e.g.*, all methods that *modify* information from the database).
- knowing that a method tends to cumulate more than a single role (*e.g.*, a method that both *retrieves* and *modifies* database information) is often a sign of an ill-designed method.

Design Roles of Classes. According to the way a class is structured, different roles associated to a class within the data source layer have been identified. We have presented in Section 2.4.2 several design roles of classes. Next, we will focus on the following roles: *Table Data Gateway (TDG)* [27], *Row Data Gateway (RDG)* [27] and *Active Record (AR)* [27].

7.2.2 Identifying Roles in the Data Source Layer

In order to browse through the classification of the methods from the data source layer according to the R, I, M, D design roles, we have developed a similar visualization to the one from Section 4.5.3 regarding the Distribution of Operations. In this new visualization a polymetric view represents a method which performs one or more of the SQL well-known instructions. Inside the rectangle representing a method one or up to four squares may appear, one for each type of SQL operation performed. Every square has a color, according to its meaning (green, blue, orange and red – i.e. the same meaning as in Section 4.5.3).

Starting from the definitions of the three roles of classes, we identified a set of specific features based on which they can be automatically identified in the source code. *A first mandatory condition for our identification rules is that each table is accessed by a single class*. Thus, if a class C accesses a table that is also accessed by another class, then no role can be assigned to C. Apart from this precondition, the rest of the roles' identification process for a class is captured by the flowchart depicted in Figure 7.2.

Thus, if class C accesses one or more tables, is stateless (*i.e.*, it contains only static and/or final attributes) and all its public methods ensure the communication with the database (belong to the data source layer) then class C is a *Table Data Gateway*.

BUPT

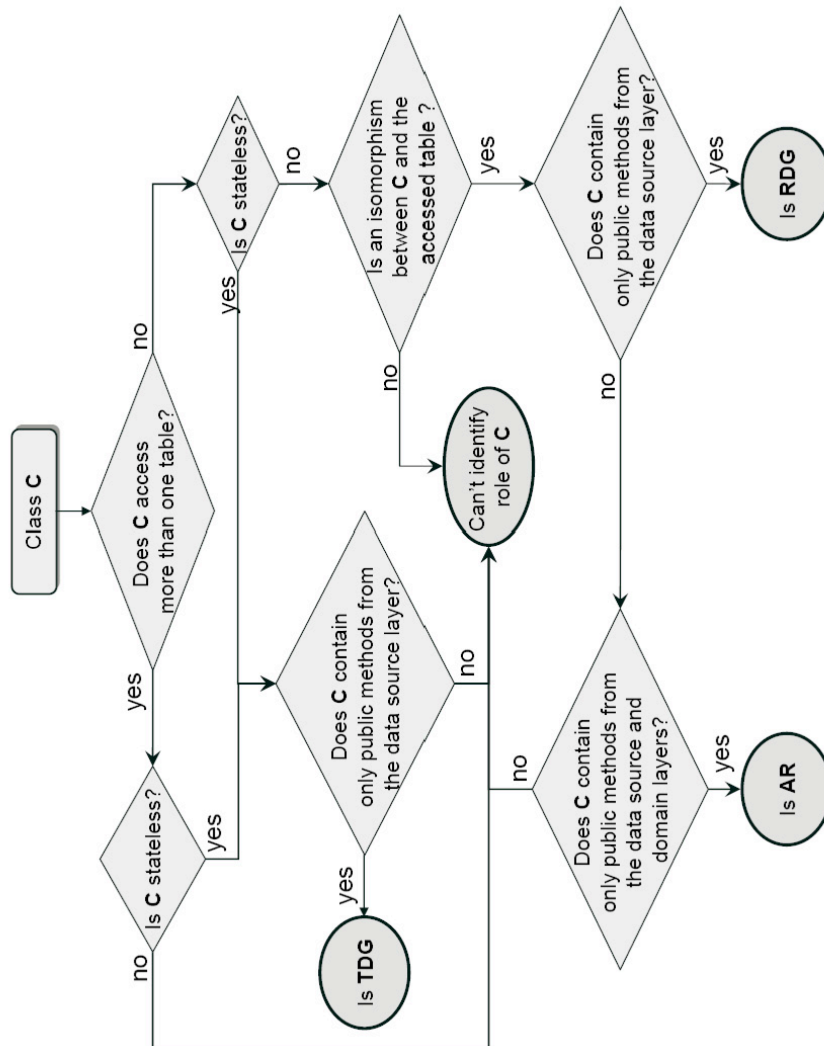

Figure 7.2: Identification of roles in the data source layer.

BUPT

In a similar fashion we say that a class C is a *Row Data Gateway* if it accesses one single table, all its public methods belong to the data source layer, and for each column in the accessed table class C defines an attribute.

Eventually, if class C accesses one table, for each column in the accessed table defines an attribute and, beside public methods from the data source layer, it contains also public methods belonging to the domain layer, then class C is considered to be an *Active Record*.

Related Work. Increasing the level of understanding in software systems by identifying features (roles) in the source code is not a new technique. For example, in [73] is introduced a concern graph for finding and describing concerns (subsets of a program source code activated when exercising a functionality) using structural program dependencies in object-oriented applications. Our approach, by extracting from the source code the dependencies between object-oriented design entities (e.g. classes, methods) and relational entities (e.g., tables, columns, primary and foreign keys), makes possible the identification of a complementary set of features specific for enterprise applications.

7.2.3 Design Information for Identifying Roles

Based on the extraction algorithm presented in Figure 7.2, we conclude that for the identification of the three aforementioned roles from the data source layer the following information is needed:

- the *attributes* of a given class.
- the *columns* of a given database table.
- the *specific functionality* of a given method (*i.e.*, communication with the database, domain logic).
- the *tables* from the database accessed by a given class.

The information needed in order to extract automatically the roles of classes from the *data source layer* is obtained from the source code according to the meta-model introduced in Chapter 4.

7.3 Roles-Aware Detection of Design Flaws

Being aware of the various design roles that classes and methods can have in an enterprise system, does not only improve the understanding of a system's design, but it can also help to improve the accuracy of detecting design problems, by taking into account the specific design rules of enterprise applications. In this section we present a roles-aware enhancement for the detection of two well-known related design flaws *i.e.*, *Data Class* and *Feature Envy* [26, 72].

BUPT

7.3.1 Data Class versus Data Transfer Object

Data Classes are dumb data holders that provide almost no functionality. The lack of functional methods may indicate that related data and behavior are not kept in one place; this makes it a strong signal for an improper data abstraction. But is a *Data Class* always a design flaw? As we are going to see next, in an enterprise application, the answer is not obvious and it depends on layers and roles.

In order to explain, let us consider the following example. Considering an enterprise application that manages a library, let's assume that all books are stored in a table called *books* (see Figure 7.3), and we need to find out the information about a specific book, identified based on its ID.

```
create table books (
  ID int primary key,
  title varchar,
  author varchar,
  publisher varchar,
  year int)
```

Figure 7.3: Table books.

In order to retrieve the information, we create the *BookDataSource* class. As you notice in Figure 7.4 the class is designed to fulfill the *Table Data Gateway* role. For each column in the table a method that returns its corresponding value was created (*e.g.*, method *getAuthor* returns the value stored in the column named *author* for the given *ID*). Consequently, a class from the domain layer that needs all the information about a book would need to call 4 methods. It is clear that retrieving in this manner large amounts of data, by performing numerous fine-grained calls to the server will be simply a performance killer [56].

So, how should methods from a *Table Data Gateway* class retrieve information to its clients? A solution that is very often encountered in enterprise systems is the use of a *Data Transfer Object* (DTO) [1] [27, 56]. A DTO is the instance of a class with little more than a bunch of fields and corresponding getters and setters for each of these fields. Such an object carries data between a client (the domain layer) and a server (the data source layer which encapsulates access to a database) in order to reduce the number of fine-grained method calls. In Figure 7.6 we see how the use of a DTO (*i.e.*, class *Book* from Figure 7.5) changes the *BookDataSource* example.

In an enterprise application a *Data Transfer Object* will always be detected as a design flaw *i.e.*, as a *Data Class*. But reporting a class that fulfills the *Data Transfer Object* design role as a design flaw, is a *false positive*; in other words an undesirable "detection noise". In order

[1] a.k.a. Value Object [1]

BUPT

```
class BookDataSource {
    public String getAuthor(int id)
               throws Exception {
        ...
        String query;
        query = "SELECT author from books " +
                "WHERE ID=" + id;
        ResultSet rs = statement.executeQuery(query);
        return rs.getString("author");
    }

    public String getTitle(int id)...
    public String getPublisher(int id)...
    public String getYear(int id)...
}
```

Figure 7.4: Class BookDataSource.

to improve the detection accuracy of *Data Classes* [57] for enterprise applications, classes that have a DTO design role must be removed from the list of suspects.

Based on the previous considerations, we can define the following detection rule for a *Data Transfer Object* role: these are classes that fulfill the following conditions (see Figure 7.7):

1. the class is a *Data Class* in conformity with the detection rule described in [39].
2. the class appears as a **return type** in at least one public method of a class that has a *Table Data Gateway* design role or,
3. the class appears as a **formal parameter type** or as a local variable type in at least one public method of a *Table Data Gateway* class.

The first mentioned condition ensures that the analyzed class does not contain functionality and, obviously, contains only attributes and accessor methods (*i.e.*, setters and getters) while the other two reflect the usage of the class from an entity whose role is *Table Data Gateway*. Thus, the second condition reflects the situation when data are read from a method belonging to a TDG class and passed, probably, to methods from the domain layer. The last condition is fulfilled by methods which are responsible for storing data within tables and the data which has to be made persistent is received usually from the domain layer embedded into a parameter.

7.3.2 Enhanced Detection Rule for Feature Envy

Feature Envy [26] is another frequent design flaw that refers to methods that appear to be more interested in the data (*i.e.*, the attributes) of another class than those of its own class. Oftentimes, this design flaw appears in methods which collaborate with *Data Classes*. In other words, *Feature Envy* usually is the sign of abnormality that appears on the clients side in case of an improper data encapsulation [57].

BUPT

```
class Book {

    private String author, title, publisher;
    private int year;

    public void setAuthor(String author) {
        this.author = author;
    }

    public String getAuthor() {
        return author;
    }
    //other getters and setters methods
}
```

Figure 7.5: Class Book - a Data Transfer Object.

In [58] Marinescu proposes a metrics-based technique for automatically detecting methods affected by Feature Envy. The detection technique takes into account how many attributes are accessed by methods from other classes – either directly or by using accessor methods. This "foreign" usage is then compared with how many attributes are accessed from the definition class. Based on the aforementioned detection rule, the `getBook` method listed in Figure 7.6 is apparently affected by *Feature Envy*, due to the fact that it accesses all the fields of class `Book` (via setter methods). But as discussed in this section, class `Book` is designed by intention as a data carrier *i.e.*, as a *Data Transfer Object*. Thus the apparent *Feature Envy* we identified in *getBook* is harmless and should not be refactored.
Based on the design roles discussed so far in this chapter we can now propose a more accurate detection rule for *Feature Envy*, in the context of enterprise applications. The new detection rule for Feature Envy will count usages of attributes from another class *only if the provider class has not the design role of a* Data Transfer Object. In Section 7.4 we will see that the impact of eliminating DTO classes from the list of data providers increases the accuracy of detecting *Feature Envy*.

7.4 Evaluation of the Approach

In order to evaluate the approach, we have conducted different experiments on the suite of enterprise applications whose main characteristics are summarized in Table 4.3. Most of these experiments are made automatically and are based on the implementations we present in the Appendix B. These implementations were introduced into the DATES module, part of the iPlasma environment.

In iPlasma the detection techniques for identifying design flaws are implemented as Java

BUPT

```
class BookDataSource {

    public Book getBook(int id) throws Exception
    {
        ...
        query = "SELECT * from books " +
                "WHERE ID=" + id;
        rs = statement.executeQuery(query);
        Book b = new Book();
        b.setAuthor(rs.getString("author"));
        b.setTitle(rs.getString("title"));
        b.setPublisher(rs.getString("publisher"));
        b.setYear(rs.getInt("year"));
        return b;
    }
}
```

Figure 7.6: Class BookDataSource revised.

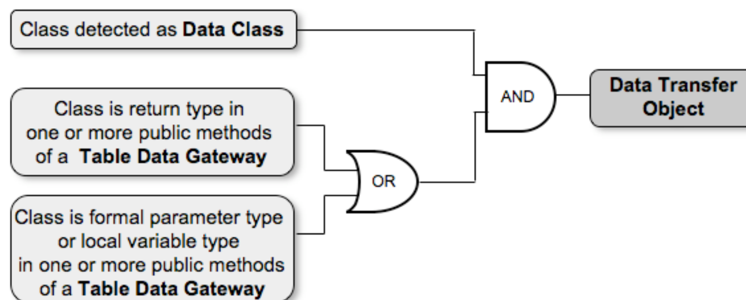

Figure 7.7: Detection of Data Transfer Object.

classes. In order to find the entities affected by a particular design flaw we apply the corresponding detection technique on the model of the analyzed system. When the DATES plugin is loaded into the INSIDER front-end it will override the existing implementations of detections for finding entities affected by Data Class and Feature Envy design flaws in "regular" object-oriented applications with the ones suitable for enterprise applications presented in Section 7.3.

The result of our approach described in this chapter adds into the DATES module the filters described by Table 7.1. The last two columns of the table are only new implementations of the existing filters, by taking into account the design roles of classes.

BUPT

Name	Is Applied-To	Is True if Entity
isTDG	class	is Table Data Gateway
isRDG	class	is Row Data Gateway
isAR	class	is Active Record
isDTO	class	is Data Transfer Object
Data Class	class	is Data Class
Feature Envy	method	is Feature Envy

Table 7.1: The introduced filters.

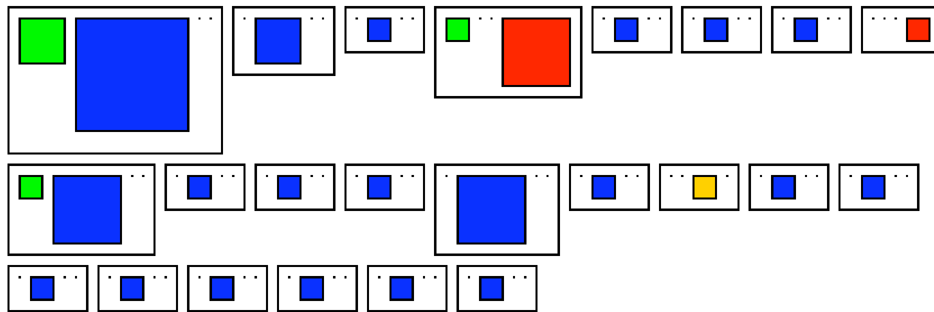

Figure 7.8: Roles of methods in KITTA.

7.4.1 Identification of Roles

In Figure 7.8 we present the visualization of the roles of methods from the *KITTA* application. As we can notice, there are some methods which have more than a role. For example, the fourth method from the figure which has 2 roles (*e.g.*, insert and delete, according to the *Distribution of Operations* introduced in Section 4.5.3) accesses 4 tables from the source code and performs an insert into table *child* and three delete operations from *priorities*, *parentappreq* and *request*.

In contrast, in the *Payroll* application, as revealed by Figure 7.9 each method has a single role and accesses a single table.

In the rest of this section we present the results of applying the algorithm presented in Figure 7.2 for the identification of roles within the analyzed applications. In order to give a view about the size of the data source layer, we present in the first line of Table 7.2 the number of classes that were mapped to the data source layer according to the rules defined in Section 4.1.3.

As a result, we identified a total of 45 classes that have precise design roles. 11 of these classes come from the **Payroll** case-study and are identified as being a *Table Data Gateway* (TDG). One further class from the **TRS** system is also mapped to the TDG role.

BUPT

	KITTA	TRS	Payroll	CentraView
Data Source Classes	9	10	16	837
Table Data Gateway	0	1	11	33
Row Data Gateway	0	0	0	0
Active Record	0 (2)	0	0	0

Table 7.2: Identified roles.

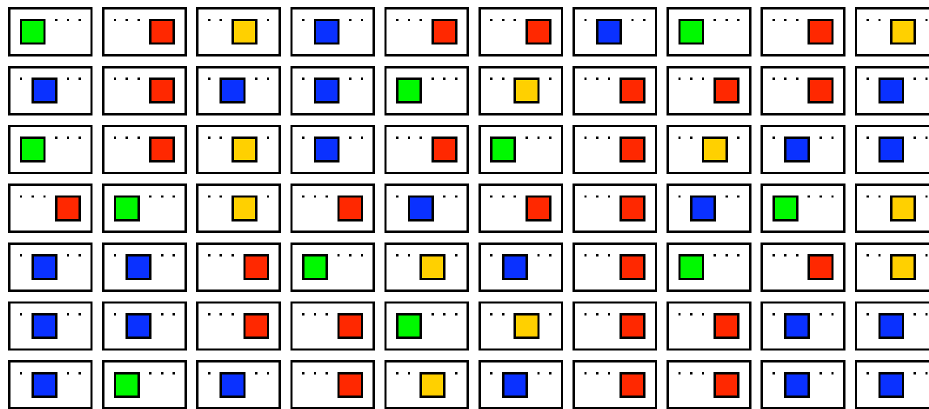

Figure 7.9: Roles of methods in Payroll.

At first sight the surprising thing is that our approach did not identify any class as being a *Row Data Gateway* or *Active Record*. In order to check if this is due to the rather small size of the case-studies or if it is a conceptual problem, we performed a in-depth analysis, partially manually and partially supported by the iPlasma [53] analysis environment. We analyzed in particular classes from the data source layer with a special focus on those for which no mapping to a role was made. The findings and particularities of each of the four systems are discussed next.

The *KITTA* application. In this system 1 class from the 9 belonging to the data source belongs also to the domain and presentation layer, so it breaks the steady rule about the dependencies between the main layers. 5 classes from the data source belong both to the data source and domain layer and access two or more tables, so definitely they do not have one of the presented roles. The rest of the remaining classes (3) look very much like *Active Records* but they were not identified because they break the condition regarding the isomorphic schema between the attributes of the class and the columns in the accessed table. At this point we decided to relax the condition about the isomorphic schema and consider a class as being an *Active Record* if between its attributes and the columns in the accessed table there is a match greater than 80%. Using this more loose condition, 2 out of the 3 classes from the application were identified as *Active Records*.

BUPT

The *TRS* application. Here, the application of the aforementioned relaxation of the rule for *Active Record* did not improve the number of classes mapped to roles. Apart from that, we found 2 classes that belong to all the three main layers and the other remaining 7 belonging both to the domain and presentation layer. Again, some of the classes were accessing more than one table, while among those that access only one table the matching level between attributes and table columns was under 80%.

The *Payroll* application. In this application only 4 classes from the data source layer were not classified as having a role. Here we discovered that the precondition about the access of a table by a single class (see Section 7.2.2) is the reason why 2 further classes were not identified as being a *Table Data Gateway*.

The *CentraView* application. Due to the great number of classes which were classified as belonging to the data source layer a complete manual inspection is almost impossible. Thus, we performed a reduced inspection which is presented next.

According to the mapping of classes into layers from the Section 4.5.1 in *CentraView* from the 837 classes classified as belonging into the data source layer, 659 belong also into the presentation layer and, consequently, they can not be classified as having a *Row Data Gateway* or *Active Record* design roles. Thus, we continue our inspection by regarding only to the remaining 178 classes which were mapped only into the data source layer. Moreover, in order to find RDG or AR classes we filter them according to the number of tables accessed – *i.e.*, we are only interested in the mentioned cases only to those classes that access a single table. From the remaining classes which were not identified as having a role we find out only 10 which access a single table.

We performed an investigation among the 10 classes and none of them have the role of being a *Row Data Gateway* or *Active Record*. For example, we found class *SaveDraftEJB* as having 2 attributes while the accessed table *emailmessage* contains 23 columns. Such discrepancy is reflected in most of the 10 classes (7), while in the case of the remaining 3 we have only a match among the number of attributes from the class and the number of columns from the accessed table. Anyway, one attribute has a user-defined type and, obviously, its type cannot have the same type as the column.

7.4.2 Roles-Aware Detection of Design Flaws

In Section 7.3 we proposed an enhanced detection technique for two well known design flaws (*i.e.*, Data Class and Feature Envy) so that they deliver more accurate results on enterprise applications. Next, we are going to discuss the results obtained by detecting the two design flaws using first the initial detection rules found in [58] and then the enhanced version described in Section 7.3. The first line of Table 7.3 presents the number of Data Classes from the applications obtained by considering each analyzed enterprise application as being a "regular" object-oriented one. By putting in correspondence the identified roles in each application with the first line of Table 7.3, the results do not surprise us because when we encounter

BUPT

	KITTA	**TRS**	**Payroll**	**CentraView**
Data Class	4	8	15	259
Data Transfer Object	0	2	11	11
Revised Data Class	4	6	4	248

Table 7.3: Detection of Data Class.

a class whose role is a *Table Data Gateway* we are aware that it is possible to have *Data Classes* with a special role. The data from the second line of Table 7.3 are obtained by applying the *Data Transfer Object* identification presented in Section 7.3. Using the proposed enhanced detection technique for identifying classes affected by Data Class design flaw in enterprise applications, *Data Transfer Objec*t classes will not be detected as being affected by the design flaw.

	KITTA	**TRS**	**Payroll**	**CentraView**
Feature Envy	1	7	48	513
Revised Feature Envy	1	5	9	433

Table 7.4: Detection of Feature Envy.

In Table 7.4 we present the number of methods from each analyzed system that are affected by the *Feature Envy* design flaw. Similar to the previous table, the first line contains the results obtained by disregarding the fact that the system is an enterprise one. It was no surprise that *Payroll* and *CentraView* were the system where most *Feature Envy* methods were detected, when the "classical" detection technique was applied, due to the fact that they have 11 *Data Transfer Objects* (which is also a consequence of the large number of *Table Data Gateway* classes identified before).

When we applied the revised *Feature Envy* detection technique (see Section 7.3.2) the accesses to the attributes of *Data Transfer Objects* classes were not counted anymore. Consequently, a large number of the methods that were apparently affected by *Feature Envy* disappeared from the reported suspects because they were either used for retrieving and storing data from the database (methods from the data source layer) or they were manipulating the data from the database (methods from the domain layer).

Concluding Remarks.

- The first conclusion is that, living in a less than perfect world, the identification rules for detecting design roles must be less strict in order to also capture the cases where the particularities of a specific role are slightly altered.
- Most of the applications showed us that in some cases the basic rule for designing enterprise applications *i.e.*, the separation between layers is brutally broken. Thus, the

BUPT

identification rules work fine only if the application is conforming (at least intentionally) to the general design rules and practice defined for enterprise applications.

- The experiments indicate that roles have a big impact on eliminating “noise" (*i.e.*, false positives) from “classical" detection rules.
- In order to obtain accurate results when we apply “regular” object-oriented detection techniques on enterprise applications we need to analyze the impact of roles upon the detection techniques for other design flaws.
- The results encourage us to extend the number of identifiable roles, as this will be definitely necessary for improving the accuracy of the detection for further design flaws (*e.g.*, ShotgunSurgery [26]).

BUPT

8

Conclusions. Future Work

The contribution of this Ph.D. thesis is in the field of understanding and quality assurance for enterprise applications.

The first part of the thesis presents in a brief manner the main characteristics of enterprise applications and the state of the art regarding the actual techniques of understanding and quality assessment of object-oriented and enterprise applications. The presentation reveals that:

- in order to perform reverse engineering on enterprise applications a meta-model specific to this kind of applications is an absolute necessity.
- quality assurance for enterprise applications requires specific techniques and tools that have to take into account their particularities.
- the existing techniques for quality assurance for "regular" object-oriented system might bring inaccurate results when applied to enterprise software systems.
- when performing the process of data reverse engineering it is advisable to take into account besides the information extracted from the schema and the database also information extracted from the usage of tables in the source code.

The second part of the thesis is a step forward in designing, understanding and quality assessing of enterprise applications by taking into account their particularities that distinguish them from "regular" object-oriented systems.

We introduced a specific meta-model for representing enterprise applications in order to facilitate the process of reverse engineering upon this type of applications. We create a model extractor which loads from the source code as well as from the schema of the database the information specified in the meta-model. We evaluated the accuracy of the introduced meta-model by performing different experiments on a suite of enterprise applications.

As primary applications of the introduced meta-model we created a suite of design related

quality analyses which enable to find out the accesses of tables in the source code, the affiliation of design entities to the three well-known existing layers (data source, domain and presentation) as well as the entities which belong to multiple layers. We also defined some visualizations regarding the accesses of tables in the source code.

Regarding the data reverse engineering, as part of understanding enterprise applications the introduced approach:

- allows us to identify automatically the related tables from which data are retrieved in the source code.
- based on the related tables, it identifies possible interrelationships between relational tables (*e.g.*, Missed Constraints).
- provides us with the objectual meaning of foreign keys, as revealed from the interactions between source code and database schema (the D, A, I RELATIONS).

The main features regarding quality assessment of enterprise applications:

- allow us to identify automatically roles which design entities (classes and methods) might have within the application. The identification helps understanding the specific purpose of each entity and, at the same time, helps localizing within the system all design fragments that have the same specific role.
- increase the accuracy of the detection of two well-known design flaws (*Data Class and Feature Envy*) by making them take into account the identified design roles that entities might have. This way, design related analyses for "regular" object-oriented became suitable when applied to enterprise applications.

We conducted different experiments in which we automatically:

- extracted the models of further inspected enterprise applications.
- identified interrelationships between tables in enterprise applications, interrelationships which were not explicitly defined in the schema of the involved relational table.
- extracted objectual meaning of foreign keys in enterprise applications, some of the classified foreign keys being explicitly defined in the schema of the involved relational database and others being automatically inferred by our technique.
- identified roles in enterprise applications, by using the extraction algorithm described in Chapter 7. The experiment that we performed has revealed a further contribution: based on the identified roles, the accuracy of traditional techniques for detecting design flaws has been improved, by removing a number of false positives for the *Data Class* and *Feature Envy* flaws. This has been possible because traditional object-oriented quality assessment techniques do not take into account the special roles that a class or method might have in an enterprise application.

BUPT

Contributions of the thesis The problems addressed within this dissertation were presented at [47] within the Ph.D. Symposium in 2006, as part of the Working Conference on Reverse Engineering.

We summarize the contributions of this Ph.D. thesis as follows:

- The DATES meta-model which allows us to find out the connections between the two programming paradigms involved in enterprise applications. The introduced meta-model was published in [51].
- Different static analyses regarding the communication between the object-oriented and relational parts of enterprise applications.
- The approach which allows us to detect the missed constraints within tables, as part of the data reverse engineering process. The introduced detection is based on the usages of tables in the source code. We present this work also in [50].
- The refined meaning of foreign keys (*i.e.*, the objectual meaning term we introduced in [49]) which allows us to differentiate between an *is a* and a *part of* relations among tables related by foreign keys.
- Finding different roles among the methods and the classes (*e.g.*, Table Data Gateway, Row Data Gateway and Active Record) from the data source layer.
- Creating new static analyses which identify design flaws in enterprise applications by taking into account the various roles of classes. The main subject of the paper [46] are the last two mentioned contributions.
- The tool support called DATES that accompanies all the introduced approaches. We presented DATES at an international conference and a part of its facilities are presented in [48]. Regarding the infrastructure built and used we published the papers [55, 54, 53, 52].

Future Work We will focus our future work on the next fronts:

- We intend to extend the tool support in order to be able to use it upon enterprise applications written using other technologies (.NET, different persistency providers, different communication techniques). In this direction, we have already constructed a tool called Mc# which extracts design facts from C# source code [65].
- We intend to continue the evaluation of the introduced approaches against other enterprise applications.
- We intend to experiment the extraction of Row Data Gateway and Active Record roles using different thresholds regarding the isomorphic schema between the class and the accessed table within the class.

BUPT

- Due to the fact that roles proved to have a big impact on eliminating false positives from "classical" detection rules, we are going to extend the number of identifiable roles in order to analyze their impact on other detections of design flaws (*e.g.*, Shotgun Surgery [26]).
- We intend to integrate our tool support into an IDE like Eclipse in order to facilitate its use, as the integration would allow to analyze the system in real-time instead of using a separate tool to construct its model.

BUPT

Bibliography

[1] D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns: Best Practicies and Design Strategies.* Prentice Hall, 2001.

[2] S. Ambler. *Agile Database Techniques.* John Wiley & Sons, 2003.

[3] S.W. Ambler and P.J. Sadalage. *Refactoring Databases: Evolutionary Database Design.* Addison-Wesley, 2006.

[4] C. Batini and M. Scannapieca. *Data Quality: Concepts, Methodologies and Techniques.* Springer Verlag, 2006.

[5] C. Bauer and G. King. *Java Persistance with Hibernate.* Manning Publications, 2007.

[6] J.M. Bieman and B.K. Kang. Cohesion and Reuse in an Object-Oriented System. *Proc. ACM Symposium on Software Reusability*, 1995.

[7] M. Blaha and W. Premerlani. Observed idiosyncracies of relational database designs. In *Proc. Working Conference on Reverse Engineering*, 1995.

[8] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of ejb applications. In *Proc. OOPSLA*, 2002.

[9] E. Chikofsky. *The Necessity of Data Reverse Engineering - Preface to Data Reverse Engineering: Slaying the Legacy Dragon.* McGraw-Hill, 1996.

[10] E.J. Chikofsky and J.H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[11] O. Ciupke. Automatic Detection of Design Problems in Object-Oriented Reengineering. In *Technology of Object-Oriented Languages and Systems*, 1999.

[12] A. Cleve, J. Henrard, and J.L. Hainaut. Data reverse engineering using system dependency graphs. In *Proc. Working Conference on Reverse Engineering*, 2006.

[13] W. Cook and S. Rai. Safe query objects: Statically typed objects as remotely executable queries. In *Proc. International Conference on Software Engineering*, 2005.

[14] C.J. Date. *An Introduction to Database Systems.* Addison Wesley, 2004.

[15] I. de Guzman, M. Polo, and M. Piattini. An integrated environment for reengineering. In *Proc. IEEE International Conference on Software Maintenance*, 2005.

BUPT

[16] Tom DeMarco. *Controlling Software Projects; Management, Measurement and Estimation.* Yourdan Press, New Jersey, 1982.

[17] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns.* Morgan Kaufmann, 2003.

[18] Y. Deng and D. Chays. Testing database transactions with agenda. In *Proc. International Conference on Software Engineering*, 2005.

[19] S. Ducasse and M. Lanza. The class blueprint: Visually supporting the understanding of classes. In *IEEE Transactions on Software Engineering*, 2005.

[20] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proc. International Symposium on Constructing Software Engineering Tools*, 2000.

[21] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. IEEE International Conference on Software Maintenance*, 1999.

[22] C. Baragoin et al. *DB2 Cube Views: A Primer.* IBM International Technical Support Organization, 2003.

[23] G. Booch et al. *Object-Oriented Analysis and Design with Applications.* Addison Wesley, 2007.

[24] K. Brown et al. *Enterprise Java Programming with IBM Websphere.* Addison-Wesley, 2001.

[25] SEMA FAST Parser. Internal Programmer's Manual 2001.

[26] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[27] M. Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, 2003.

[28] European Union. Community Research & Development Information Service CORDIS FP. Homepage of information society technologies. In *http://www.cordis.lu/ist/*, 2005.

[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[30] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proc. International Conference on Software Engineering*, 2004.

[31] J.L. Hainaut, J. Henrard, J.M. Hick, D. Roland, and V. Englebert. The nature of data reverse engineering. In *Proc. Data Reverse Engineering Workshop*, 2000.

[32] J.M. Hick and J.L. Hainaut. Strategy for database application evolution: The DB-MAIN approach. In *Proc. Conceptual Modeling - ER*, 2003.

[33] R. Kazman, S.G. Woods, and S.J. Carriere. Requirements for integrating software architecture and reengineering models: Corum ii. In *Proc. IEEE Working Conference on Reverse Engineering*, 1998.

BUPT

[34] W. Keller. Mapping objects to tables: A pattern language. In *Proc. European Conference on Pattern Languages of Programs*, 1997.

[35] W. Keller. Object/relational access layers: a roadmap, missing links and more patterns. In *Proc. European Conference on Pattern Languages of Programming and Computing*, 1998.

[36] R. Koschke, J.F. Girard, and M. Wurthner. An intermediate representation for reverse engineering analyses. In *Proc. IEEE Working Conference on Reverse Engineering*, 1998.

[37] M. Lanza and S. Ducasse. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. In *OOPSLA 2001 proceedings*, 2001.

[38] M. Lanza and S. Ducasse. Polymetric views - a lightweight visual approach to reverse engineering. In *IEEE Transactions on Software Engineering*, 2003.

[39] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice.* Springer Verlag, 2006.

[40] Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.

[41] W. Li and S. Henry. Maintenance Metrics for the Object Oriented Paradigm. *IEEE Proc. First International Software Metrics Symp.*, pages 52–60, may 1993.

[42] M. Lippert and S. Roock. *Refactoring in Large Software Projects.* John Wiley & Sons, 2006.

[43] A. Liu, L. Bass, and M. Klein. *Analyzing Enterprise JavaBeans Systems Using Quality Attribute Design Primitives.* Technical Note CMU/SEI-2001-TN-025., 2001.

[44] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice-Hall, 1994.

[45] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev. Static techniques for concept location in object-oriented code. In *Proc. International Workshop on Program Comprehension*, 2005.

[46] C. Marinescu. **Identification of Design Roles for the Assessment of Design Quality in Enterprise Applications.** In *Proc. IEEE International Conference on Program Comprehension (ICPC), Athens, Greece*. IEEE Computer Society Press, 2006.

[47] C. Marinescu. **Quality Assessment of Enterprise Software Systems.** In *Proc. Working Conference on Reverse Engineering (WCRE Doctoral Symposium), Benevento, Italy*. IEEE Computer Society Press, 2006.

[48] C. Marinescu. **DATES: Design Analysis Tool for Enterprise Systems.** In *Proc. IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), Paris, France*. IEEE Computer Society Press, 2007.

[49] C. Marinescu. **Discovering the Objectual Meaning of Foreign Key Constraints in Enterprise Applications.** In *Proc. Working Conference on Reverse Engineering (WCRE), Vancouver, Canada*. IEEE Computer Society Press, 2007.

BUPT

[50] C. Marinescu. **Identification of Relational Discrepancies between Database Schemas and Source-Code in Enterprise Applications.** In *Proc. International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE Computer Society Press, 2007.

[51] C. Marinescu and I. Jurca. **A Meta-Model for Enterprise Applications.** In *Proc. International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE Computer Society Press, 2006.

[52] C. Marinescu, R. Marinescu, P.F. Mihancea, D. Raţiu, and R. Wettel. **Analysis Infrastructure for Quality Assesment of Object-Oriented Design.** In *Proc. International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. Editura Mirton, Timişoara, 2005.

[53] C. Marinescu, R. Marinescu, P.F. Mihancea, D. Raţiu, and R. Wettel. iPlasma: An integrated platform for quality assessment of object-oriented design. In *Proc. IEEE International Conference on Software Maintenance (ICSM Industrial and Tool Volume), Budapest, Hungary*. IEEE Computer Society Press, 2005.

[54] C. Marinescu, R. Marinescu, and T. Gîrba. **A Dedicated Language for Object-Oriented Design Analyses.** In *CAVIS Workshop*. eAustria Research Institute from Timişoara, 2004.

[55] C. Marinescu, R. Marinescu, and T. Gîrba. **Towards a Simplified Implementation of Object-Oriented Design Metrics.** In *Proc. IEEE International Software Metrics Symposium, Como, Italy*. IEEE Computer Society Press, 2005.

[56] F. Marinescu. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms.* John Wiley & Sons, 2002.

[57] R. Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, "Politehnica" University of Timişoara, 2002.

[58] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proc. IEEE International Conference on Software Maintenance*, 2004.

[59] R. Martin, D. Riehle, and F. Buschmann. *Pattern Languages of Program Design 3.* Addison-Wesley, 1998.

[60] R.C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2002.

[61] T.J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, dec 1976.

[62] R. McClure and I. Kruger. Sql dom: Compile time checking of dynamic sql statements. In *Proc. International Conference on Software Engineering*, 2005.

[63] Petru Florin Mihancea. The extraction of detailed design information from C++ software systems. Master's thesis, Faculty of Automatics and Computer Science, "Politehnica" University of Timişoara, 2004.

BUPT

[64] Petru Florin Mihancea. *Type Highlighting. A Reverse Engineering Visual Approach to Characterize the Clients of Class Hierarchies.* Ph.D. Report 3, "Politehnica" University of Timișoara., 2007.

[65] P.F. Mihancea, G. Ganea, I. Verebi, C. Marinescu, and R. Marinescu. **McC and Mc#: Unified C++ and C# Design Facts Extractors Tools.** In *Proc. International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE Computer Society Press, 2007.

[66] C. Nock. *Data Access Patterns: Database Interactions in Object-Oriented Applications.* Addison-Wesley, 2003.

[67] S. Paul, A. Prakash, E. Buss, and J. Henshaw. Theories and techniques of program understanding. In *Proc. Conference of the Centre for Advanced Studies on Collaborative research*, 1991.

[68] D. Rațiu and F. Deissenboeck. How programs represent reality (and how they don't). In *Proc. Working Conference on Reverse Engineering*, 2006.

[69] D. Rațiu and F. Deissenboeck. Programs are knowledge bases. In *Proc. IEEE International Conference on Program Comprehension*, 2006.

[70] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* McGraw Hill, second edition, 2002.

[71] D. Ratiu. *Memoria: A Unified Meta-Model for Java and C++.* Master Thesis, "Politehnica" University of Timișoara, 2004.

[72] A.J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[73] M.P. Robillard and G.C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proc. International Conference on Software Engineering*, 2002.

[74] S. Staiger. Static analysis of programs with graphical user interface. In *Proc. European Conference on Software Maintenance and Reengineering*, 2007.

[75] COMPOST Team. *Recoder Project, http://recoder.sourceforge.net/*. University of Karlsruhe.

[76] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, Institute of Informatics and Applied Mathematics, University of Bern, 2001.

[77] R. Wettel and R. Marinescu. Archeology of code duplication: Recovering duplication chains from small duplication fragments. In *Proc. International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC05)*, 2005.

[78] D. Yeh and Y. Li. Extracting entity relationship diagram from a table-based legacy database. In *Proc. European Conference on Software Maintenance and Reengineering*, 2005.

BUPT

Appendix A

Visualizing Accesses to Tables

In this appendix we provide the implementation details regarding the introduced visualizations from Section 4.5.3.

A.1 Tables Accesses. Implementation

In order to obtain automatically the *Tables Accesses* visualization we implemented it within the iPLASMA environment which contains the jMONDRIAN module which enables performing visualizations.

In Line 1 we include the new analysis into a package containing the *plugins* name in order to be loaded when the DATES module is instantiated within the used environment. In Lines 2..11 we include different classes from more packages, most of them being part of iPLASMA.

```
1  package lrg.dates.plugins.visualizations;

2  import lrg.common.abstractions.plugins.visualization.*;
3  import lrg.common.abstractions.entities.*;
4  import lrg.jMondrian.view.ViewRenderer;
5  import lrg.jMondrian.layouts.FlowLayout;
6  import lrg.jMondrian.figures.Figure;
7  import lrg.jMondrian.painters.RectangleNodePainter;
8  import lrg.jMondrian.util.LinearNormalizerColor;
9  import lrg.jMondrian.commands.AbstractNumericalCommand;
10 import lrg.insider.util.Visualization;
11 import java.util.ArrayList;
```

In Line 12 we create a class which extends *AbstractVisualization*, this relation being mandatory in order to have a new plugin which displays the defined entities. In Line 14 we establish the name of the visualization and we defined explicitly the entity for which the defined analysis is applied (in this case, the whole system).

BUPT

```
12public class AccessedTables extends AbstractVisualization{
13 public AccessedTables(){
14  super("Tables Accesses", "", "system");
15 }

16 public void view(AbstractEntityInterface entity) {

17  ArrayList set = entity.getGroup("table group").getElements();
18  Figure f = new Figure();

19  f.nodesUsing(set, new RectangleNodePainter(true).
20    width(new AbstractNumericalCommand(){
21           public double execute(){
22            return
23             ((Double)(((AbstractEntity)receiver).
24              getProperty("NSt").
25              getValue()))doubleValue() * 10;
26    }}).
27    height(new AbstractNumericalCommand(){
28           public double execute(){
29            return
30             ((Double)(((AbstractEntity)receiver).
31              getProperty("NAC").
32              getValue()))doubleValue() * 10;
33    }}).
34    color(new LinearNormalizerColor(set,
35           Visualization.metricCommand("NC"))).
36           name(Visualization.stringCommand("Name")));

37    f.layout(new FlowLayout(5,5,300));
38    ViewRenderer r;
39    r = new ViewRenderer("Tables Accesses");
40    f.renderOn(r);
41    r.open();
42    }
43}
```

The parameter received by the method *view*(Line 16) is nothing else than the system upon the visualization is performed. We get from the analyzed system all the tables(Line 17) end for each existing table we create a node inside the figure instantiated in 18. In 20 we specify the width of the rectangle representing the table (in this case, the value of the *NSt – Number of Statements* metric), in 27 its height is enriched with the value of *NAC – Number of Classes which access the tables* while in 34 we associate a color representing the Number of Columns the table contains.

The last group of lines specify the parameters of the created visualization (*i.e.*, title, dimensions, layout).

BUPT

A.2 Distribution of Operations. Implementation

In this section we present how we implemented the visualization regarding the *Distribution of Operations*. Like in the previous case, it is applicable to a system (Line 5).

The figure displayed is created within the body of emthod *createFigure* from Line 12. In Line 14 we get from the system all the existing table and in 15 we filter them in order to present only those which are used in the source code, by applying the *isUsedTable* filter.

```
1  package lrg.dates.plugins.visualizations;
2  //import statements ...
3  public class OperationsDistribution extends
                                 AbstractVisualization{
4  public OperationsDistribution(){
5   super("Distribution of Operations", "", "system");
6  }

7  public void view(AbstractEntityInterface theSystem) {
8  Figure f = createFigure(theSystem);
9  ViewRenderer r = new ViewRenderer("Distribution of Operations");
10 f.renderOn(r); r.open();
11 }

12 public Figure createFigure(AbstractEntityInterface theSystem) {
13   GroupEntity allTables;
14   allTables = theSystem.getGroup("table group");
15   allTables = allTables.applyFilter("isUsedTable");
```

We represent each used table in the source code with a rectangle which contains four rectangles inside. From Lines 25 to 30 we get for each table the corresponding set of select, insert, update and delete SQL statements. In 31 we create a collection of four elements, each element being the group of one of the four existing types of SQL operations.

```
16   ArrayList<AbstractEntityInterface> alls;
17   alls = allTables.getElements();
18   Figure f = new Figure();

19   f.nodesUsingForEach(alls, new
20     RectangleNodePainter(true).
21     name(Visualization.stringCommand("Name")),
22      new AbstractFigureDescriptionCommand(){
23       public Figure describe(){
24        Figure fig = new Figure();

25        GroupEntity select = new
26         GroupEntity("select",((AbstractEntity)receiver).
27          getGroup("select operations").getElements());

28        GroupEntity insert = ... //insert operations
```

BUPT

```
29        GroupEntity update = ... //update operations
30        GroupEntity delete = ... //delete operations

31        ArrayList statements = new ArrayList();
32        statements.add(insert); statements.add(select);
33        statements.add(update); statements.add(delete);
```

The width and the height representing each group of instructions are proportional with the cardinality of the group and they are set by the instructions from Lines 37 and 41. Regarding the color, it is set according to the type of the SQL instructions – for example, for the *insert* SQL statements the color is green (Line 48).

```
34        fig.nodesUsing(statements, new RectangleNodePainter(true)
35           .width(new AbstractNumericalCommand(){
36                  public double execute(){
37                   return ((GroupEntity)receiver).size() * 10;
38                  }})
39           .height(new AbstractNumericalCommand(){
40                   public double execute(){
41                    return ((GroupEntity)receiver).size() * 10;
42                   }})
43           .color(new AbstractNumericalCommand(){
44                  public double execute(){
45                   GroupEntity group = (GroupEntity)receiver;
46
47                   if(group.getName().equals("insert"))
48                      return Color.GREEN.getRGB();

49                   if(group.getName().equals("select"))
50                      return Color.BLUE.getRGB();

51                   if(group.getName().equals("update"))
52                      return Color.ORANGE.getRGB();

53                   if(group.getName().equals("delete"))
54                      return Color.RED.getRGB();

55                   return Color.WHITE.getRGB();}}));
56        fig.layout(new FlowLayout(5,5,90));
57        return fig;}});

58        f.layout(new FlowLayout(5,5,500));
59        return f;
60 }}
```

BUPT

Appendix B

Roles-Aware Detection of Design Flaws

This appendix presents the implementation of two roles (*e.g.*, Table Data Gateway and Data Transfer Object) that classes may have within enterprise applications according to the algorithms presented in Section 7.1, respectively Section 7.3.

B.1 Table Data Gateway. Implementation

The implementation of the identification of the Table Data Gateway role within the DATES module implies extending the class *FilteringRule* (Line 10), class which was imported by the instruction from Line 8. In Line 13 we specify that this filter is applied to a class.

```
1  package lrg.dates.plugins.filters.classes;

2  import lrg.common.abstractions.plugins.Descriptor;
3  import lrg.common.abstractions.entities.*;
4  import lrg.dates.core.AnnotationDATES;
5  import lrg.dates.core.schema.sql.TableSchema;
6  import lrg.dates.privates.filters.IsStaticOrFinal;
7  import lrg.dates.privates.filters.
            HasOnlyPublicDatasourceMethods;
8  import lrg.common.abstractions.plugins.filters.
            FilteringRule;
9  import java.util.Collection;
10 public class IsTDG extends FilteringRule
11 {
12  public IsTDG() {
13   super(new Descriptor("isTDG", "", "class"));
14 }
```

Starting with Line 15 we override the *applyFilter* method from the base class in order to let it return the proper value regarding the TDG role of a class.

BUPT

In Line 19 we get from the current class the annotation we introduced when the DATES module was loaded within the iPLASMA plaftorm. In order to find out if the class contain only public datasource methods we apply the *HasOnlyPublicDatasourceMethods* filter upon it (Line 23).

In Line 27 we get the group of attributes the class contains and we apply the *IsStaticOrFinal* filter upon them. If the number of the attributes of the class is equal with the number of attributes which are static or final then it means that the condition regarding the stateless of the class was fulfilled (Line 30).

```
15 public boolean applyFilter(AbstractEntityInterface anEntity) {
16  if (anEntity instanceof lrg.memoria.core.Class == false)
17   return false;
18 AnnotationDATES an;
19 an = (AnnotationDATES)anEntity.getAnnotation("dat");

20  //check if the class has only public datasource methods
21  HasOnlyPublicDatasourceMethods pds;
22  pds = new HasOnlyPublicDatasourceMethods();
23  if (pds.applyFilter(anEntity) == false)
24   return false;

25  //check if the class has only static final attributes
26  GroupEntity attributes;
27  attributes = anEntity.getGroup("attribute group");
28  GroupEntity attrStFi;
29  attrStFi = attributes.applyFilter(new IsStaticOrFinal());
30  if(attributes.size() != attrStFi.size())
31   return false;
```

The final checked condition is related to the number of tables the class accesses. We get the collection of accessed tables by the class from the value assigned to *an* variable from Line 19. If the class accesses at least one tables then it means that we find a class which has the *Table Data Gateway* role.

```
32  //check if the class accesses one or more table
33  Collection<TableSchema> accessedTables;
34  accessedTables = an.getAccessedTables();
35  if (accessedTables.size()==0)
36   return false;

37 return true;
38 }
39}
```

In the following lines we provide the implementation of the *applyFilter* method from the class *HasOnlyPublicDatasourceMethods*. In Line 2 we obtaine the group of methods the class contains. In Line 3 we retain inside the group only those methods which are public. Next (Line 4), we create a new group which contains only the methods which are public and belongs

BUPT

to the data source layer. Finally, we compare the size of the two groups and we return the corresponding result.

```
1 public boolean applyFilter(AbstractEntityInterface anEntity) {
2  GroupEntity mt = anEntity.getGroup("method group");
3  mt = mt.applyFilter(new IsPublicMethod());

4  GroupEntity ds = mt.applyFilter(new isDatasource());
5  return (mt.size()==ds.size());
6 }
```

B.2 Data Transfer Object. Implementation

In order to establish if a class is a *Data Transfer Object* we create a new class which, like in the previous case, extends *FilteringRule*. In Line 12 we instantiate the filter for identifying Data Classes inside "regular" object-oriented systems and a mandatory condition which has to be fulfilled in order to be a DTO is that the class has to be identified as being a regular Data Class (Line 14).

```
1  package lrg.dates.plugins.filters.classes;

2  import lrg.common.abstractions.plugins.filters.FilteringRule;
3  import lrg.common.abstractions.plugins.Descriptor;
4  import lrg.common.abstractions.entities.*;
5  import lrg.dates.core.Dates;
6  import lrg.dates.plugins.filters.isDatasource;

7  public class IsDTO extends FilteringRule {
8   public IsDTO() {
9    super(new Descriptor("isDTO", "", "class"));
10 }

11 public boolean applyFilter(AbstractEntityInterface anEntity) {
12 lrg.insider.plugins.filters.memoria.classes.DataClass dc =
     new lrg.insider.plugins.filters.memoria.classes.DataClass();

13 boolean isDataClass = dc.applyFilter(anEntity);
```

In Line 16 we get from the analyzed system all the classes and in 17 we retain only those which are defined in the project (*i.e.*, we exclude the library classes). From the existing classes we get in 20 only those which were identified as having the TDG role.

In the following lines we get all the types which appear into the identified TDG as return types, parameter types and local variables types. We create in 24 a new group which contains the distinct entities of the aforementioned types.

In Line 27 we filter the created group in order to let it contain only the defined classes within

BUPT

the project. If the created group contains the inspected entity (referenced by the *anEntity* variable) then it means that we identified a class whose role is *Data Transfer Object*.

```
14 if(isDataClass) {
15  lrg.memoria.core.System system = Dates.getSystem();

16 GroupEntity classes = system.getGroup("class group");
17 classes = classes.applyFilter("model class");

18 IsTDG isTDG = new IsTDG();

19 GroupEntity ds = classes.applyFilter(new isDatasource());
20 ds = ds.applyFilter(isTDG);

21 GroupEntity rt = ds.getGroup("return types").distinct();
22 GroupEntity pt = ds.getGroup("parameter types");
23 GroupEntity vt = ds.getGroup("local variable types");

24 GroupEntity types = new GroupEntity("g", rt.getElements());
25 types.addAllDistinct(pt);
26 types.addAllDistinct(vt);

27 types = types.applyFilter("model class");

28 if (types.isInGroup((AbstractEntity)anEntity))
29  return true;
30 }
31 return false;
32 }
33}
```

BUPT

List of Publications

Scientific articles (papers) published in the proceedings (volumes) of international scientific conferences organized abroad, with ISI ranking

[1] **Cristina Marinescu**, **Discovering the Objectual Meaning of Foreign Key Constraints in Enterprise Applications**, Proceedings of the 14th Working Conference on Reverse Engineering (WCRE 2007), Vancouver, Canada, IEEE Computer Society Press. ISBN 0-7695-3034-6.

[2] **Cristina Marinescu**, **DATES: Design Analysis Tool for Enterprise Systems**, Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007), Paris, France, Tool Demonstrations Section, IEEE Computer Society Press. ISBN 0-7695-2880-5/07.

[3] **Cristina Marinescu**, **Identification of Design Roles for the Assessment of Design Quality in Enterprise Applications**, Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC 2006), IEEE Computer Society Press. ISBN 0-7695-2601-2/06.

[4] **Cristina Marinescu**, **Quality Assessment of Enterprise Software Systems**, Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006), Doctoral Symposium Section, Benevento, Italy, IEEE Computer Society Press. ISBN 0-7695-2719-1/06.

[5] **Cristina Marinescu**, Radu Marinescu, Tudor Girba, **Towards a Simplified Implementation of Object-Oriented Design Metrics**, Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS 2005), IEEE Computer Society Press. ISBN 0-7695-2371-4.

Scientific articles (papers) published in the proceedings (volumes) of international scientific conferences organized in Romania, with ISI ranking

[6] **Cristina Marinescu**, Ioan Jurca, **A Meta-Model for Enterprise Applications**, Proceedings of the 8th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2006), Timisoara, Romania, IEEE Computer Society Press. ISBN 0-7695-2740-X/06.

BUPT

Scientific articles (papers) published in the proceedings (volumes) of international scientific conferences, indexed in internationally recognized publication databases

[7] **Cristina Marinescu**, **Identification of Relational Discrepancies between Database Schemas and Source-Code in Enterprise Applications**, Proceedings of the 9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2007), Timisoara, Romania, IEEE Computer Society Press. ISBN 0-7695-3078-8/08. [DBLP, IEEE Explore]

[8] Petru F. Mihancea, George Ganea, Ioana Verebi, **Cristina Marinescu**, Radu Marinescu, **McC and Mc#: Unified C++ and C# Design Facts Extractors Tools**, Proceedings of the 9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2007), Timisoara, Romania, IEEE Computer Society Press. ISBN 0-7695-3078-8/08. [DBLP, IEEE Explore]

[9] **Cristina Marinescu**, Radu Marinescu, Petru F. Mihancea, Daniel Ratiu, Richard Wettel, **iPlasma:An Integrated Platform for Quality Assessment of Object-Oriented Design**, Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005), Tool Demonstration Track, 2005. [DBLP, IEEE Explore]

Scientific articles (papers) published in the proceedings (volumes) of other scientific conferences organized in Romania

[10] **Cristina Marinescu**, Radu Marinescu, Petru F. Mihancea, Daniel Ratiu, Richard Wettel, **Analysis Infrastructure for Quality Assesement of Object-Oriented Design**, Proceedings of the 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2004), Timisoara, 2004

[11] **Cristina Marinescu**, Radu Marinescu, Tudor Girba, **A Dedicated Language for Object-Oriented Design Analyses**, CAVIS Workshop, eAustria Research Institute from Timisoara, 2004.

BUPT

List of Research Grants

International Research Grants (as Team Member)

[1] Swiss National Science Foundation IB7320-110997/2005 - *Network of reengineering expertise*.

[2] BMBWK GZ45.527/1-VI/B/7a/2002-2005 - *Verificarea sistemelor*. Grant of the Austrian Government.

National Research Grants (as Director)

[3] PN-II TD 94/17.09.2008 - *Asigurarea Calitatii in Sistemele Software Distribuite*.

[4] CNCSIS TD GR76/23.05.2007 - *Evaluarea Calitatii in Sistemele Software distribuite*

[5] CNCSIS TD 58GR/18.09.2006 - *Evaluarea Calitatii in Sistemele Software distribuite*

National Research Grants (as Team Member)

[6] PN-II 357/1.10.2007 - *Metode si instrumente pentru asigurarea continua a calitatii in sisteme software complexe*.

[7] CNCSIS 98GR/11.06.2008 - *Tehnologii bazate pe inteligenta artificiala pentru software-ul si infrastructura de retea specifice retelelor radio de urmatoarea generatie.*

[8] CEEX Modul II 5880/18.09.2006 - *Mediu distribuit pentru controlul si optimizarea evolutiei sistemelor software.*

[9] CEEX Modul II 3147/01.10.2005 - *Asigurarea calitatii proiectarii in sisteme software industriale*

[10] CNCSIS 46GR/11.05.2007 - *Tehnologii bazate pe inteligenta artificiala pentru software-ul si infrastructura de retea specifice retelelor radio de urmatoarea generatie*.

[11] CNCSIS A1/GR181/19.05.2006 - *Mediu integrat evolutiv pentru asigurarea calitatii software-ului*