Applications of Automated Model's Extraction in Enterprise Systems

Cristina Marinescu
LOOSE Research Group, Politehnica University Timişoara, Romania

Keywords: Enterprise Systems, Meta-models, Models, Static Analysis.

Abstract: As enterprise software systems become more and more complex, the need of automated approaches for their understanding and quality assessment increases. Usually the automated approaches make use of a meta-model according to the information is mainly extracted from the source code but when considering enterprise systems, the meta-model should contain information from two different paradigms (*e.g.*, object-oriented and relational) which are not enough to be loaded only from the source code. In this paper, based on a specific meta-model for enterprise systems, we present a set of applications (approaches) which help us to understand and assess the quality of the design, as part of the maintenance process.

1 INTRODUCTION

As object-oriented systems become more and more complex, the need of using proper reverse engineering as well as quality assurance techniques upon such systems has increased significantly. Moreover, the mentioned activities become even more difficult, when considering a category of software systems, namely *enterprise*. These systems are about the display, manipulation, and storage of large amounts of complex data and the support or automation of business processes with that data (Fowler, 2003). Consequently, almost all enterprise systems involve two programming paradigms: the object-oriented one, for implementing the entire business logic, and the relational paradigm, for ensuring the persistency of the involved data.

As stated in (Rugaber and Wills, 1996), one step toward a research infrastructure accelerating the progress of reverse engineering is the creation of an intermediate representation of software systems. Probably the best-known intermediate representation of a software system is its *model*. The model of a given software system contains specific information extracted from the source code based on a *meta-model*. The meta-model specifies the relevant entities (e.g., classes, methods) and their relevant properties and relations (e.g., inheritance, method calls) found in an object-oriented system. According to the study we present in (Marinescu and Jurca, 2006), it is not suitable to perform reverse engineering upon an enterprise system using a meta-model for representing a regular object-oriented system (*i.e.*, a system which does not involve persistency). In order to perform reverse engineering on enterprise software systems we need a specific meta-model which contains, on one hand, entities from the object-oriented part and, on the other hand, entities regarding the relational part of the enterprise system and the interactions between the two paradigms. Based on such a meta-model, in this paper we define new techniques which facilitate the maintenance of enterprise systems.

The paper is structured as follows: in Section 2 we present our approach for modeling enterprise applications. Next (Section 3), we introduce the facilities provided by our approach, and we conducted an experiment based on three case-studies. The paper concludes with a discussion on related work (Section 4) and some final remarks towards the future work (Section 5).

2 A META-MODEL FOR ENTERPRISE SYSTEMS

In this section we present each type of design information from our meta-model.

Modeling Object-oriented Entities. We model the object-oriented entities 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 el-

Marinescu, C.
Applications of Automated Model's Extraction in Enterprise Systems.
DOI: 10.5220/0007916102540261
In *Proceedings of the 14th International Conference on Software Technologies (ICSOFT 2019)*, pages 254-261
ISBN: 978-989-758-379-7

ementary properties of that design entity or links to other related data structures. For example, the structure that models the *Class* design entity has 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.

Modeling Relational Entities. According to the presentation of relational databases from (Ramakrishnan and Gehrke, 2002), 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, our meta-model for representing relational databases contains the entities *Table* and *Column* found in a relational database.

Modeling Object-relational Interactions. 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 system and the relational part there are interactions only within the data source layer. In order to capture these interactions we take into account the various usages of third-party libraries and/or frameworks that are specific for the data source layer: 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). A class containing one or more methods belonging to the data source layer is also mapped to the data source layer.

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. Within our solution, we propose to annotate the *Method* entity from the object-oriented meta-model with 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. The class which models the SQL statement 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 (classes) 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.

We store in the entity *Table* also information regarding the entities that access the table (*e.g.*, methods, classes).

Due to the fact that our meta-model for enterprise systems contains entities specific to regular object-oriented systems, we decide not to build our tool support for modeling enterprise systems from scratch – we built it on the top of the MEMORIA (Raţiu, 2004) meta-model which is part of iPLASMA (Marinescu et al., 2005). iPLASMA 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 up to high-level metrics-based analysis, or detection of code duplication. The tool platform relies only on the source code as input. All of the provided support is integrated by an uniform front-end. The DATES module (Marinescu, 2007) extracts a model of enterprise applications relying on top of the presented meta-model.

3 APPLICATIONS OF THE META-MODEL

The goal of this section is to present several types of specific information extracted by DATES from enterprise software systems. These are intended to help us in order to understand and assess the design quality of the analyzed systems. The size characteristics of the applications for which we present the obtained results are summarized in Table 1.

Table 1: Characteristics of the case studies.

	TRS	**Payroll**	**CentraView**
Size	590 Kb	995 Kb	14,3 Mb
Classes	54	115	1527
Methods	500	580	13369
Tables	10	12	217
LOC	13598	13025	177655

TRS is an enterprise system developed by a team of students within the software engineering project classes. *Payroll* is an industrial enterprise application whose scope is to manage information about the employees from a company. *CentraView* is an

open-source enterprise application [1] which provides a growing businesses with a centralized view of all customer and business information.

3.1 Visualizing how Tables are Accessed in the Source 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*.

3.1.1 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, we represent each table with a rectangle enriched with three metrics, as we show in Figure 1.

The width of the rectangle represents the number of statements performed within the source code upon the represented table. The height is associated with 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 1: 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 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 rectangle is near black (*i.e.*, a large value associated with the number of columns the table has), there is a high probability that the table is affected by the Multipurpose Table design flaw (Ambler and Sadalage, 2006).

[1] https://github.com/shaunscott12/CentraView

3.1.2 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 2). 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 rectangle associated with a table:

Figure 2: 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 rectangle 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 module JMONDRIAN [2] (Mihancea, 2010) for visualizing data within the iPLASMA platform.

3.1.3 Results

Visualizations in TRS. From the first overview obtained (Figure 3) we find out that:

Figure 3: Tables Accesses in TRS.

- 2 tables are unused in the source code (*e.g.*, *Flight_reservation_comp* and *PasteErrors*).

[2] https://github.com/petrufm/jMondrian.git

- there are 3 tables which are accessed in 3 classes, while the rest of the tables are accessed from only one class.
- 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.

From the second visualization (Figure 4) related to the existing types of SQL statements for manipulating data in tables we notice that:

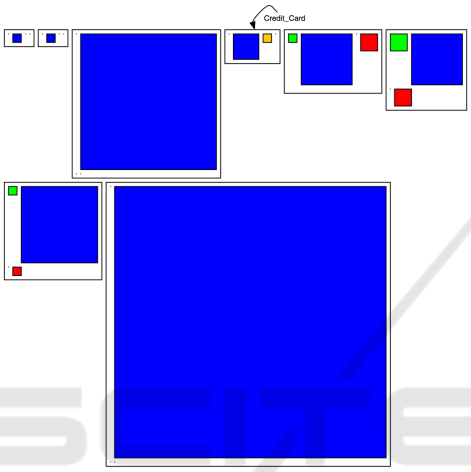

Figure 4: Distribution of Operations in TRS.

- there are 4 tables storing data which is only read within the source code (*i.e.*, only select operations are performed).
- 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 5) 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.

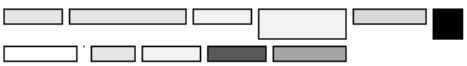

Figure 5: Tables Accesses in Payroll.

Regarding the Distribution of Operations in Payroll we notice from the Figure 6 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.

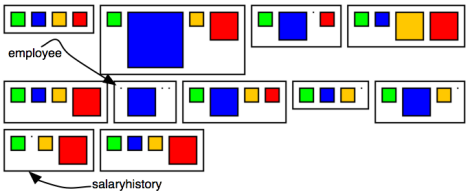

Figure 6: Distribution of Operations in Payroll.

Visualizations in CentraView. From Figure 7 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 8 we present the defined visualization obtained by analyzing CentraView. 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.

3.2 Finding Improper Usages of Exceptions in the Source Code

In this section we introduce another automatic approach that our meta-model supports related to an improper usages of exceptions in the source code. An empirical study which correlates the usage of exceptions in the source code with the defects encountered in a system is presented in (Marinescu, 2013). This study provides evidence about a positive correlation between the classes that use exceptions and the defects those classes exhibit. It shows that the classes which use exceptions are more likely to exhibit defects than the classes which do not use exceptions.

Figure 7: Tables Accesses in CentraView.

Based on the mentioned findings, we consider that developers should not neglect special cases (*i.e.*, exceptions) in the source code.

We introduce a technique that automatically detects the situations in which exceptions related to information's retrieval from the database are not properly handled in the source code. Based on the information provided by our approach, the fragments from the source code which do not properly handle the exceptions are good candidates for refactoring.

3.2.1 Motivation

Most of the existing library methods which perform operations upon the persistent data reveal abnormal situations (*e.g.*, embedded SQL errors, broken communication with the database) to its callers from the data source layer by throwing exceptions. We present the problems that may arise due to an improper mechanism for handling exceptions within the data source layer with respect to an example.

Assuming that we have a table called *Books* (Figure 9) there are a lot of possibilities for retrieving the stored data within. For example, we create the *IdiomaticBookDataSource* class (Figure 10). As you notice the class is intended to have for each column in the table a method that returns its corresponding value for a given *ID*. The used library method (*e.g.*, *executeQuery*) for retrieving the requested information throws a *SQLException*. In our example the exception is not thrown to the next architectural level (domain) and, instead, a *null* value (or a particular value) is sent. This is a bad style for handling exceptions and it has a lot of drawbacks:

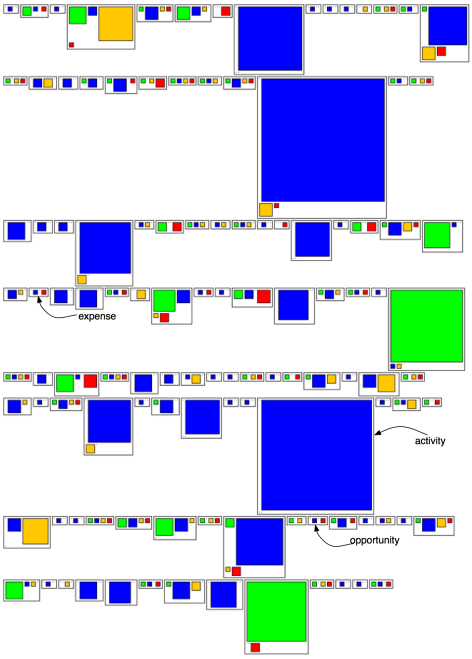

Figure 8: Distribution of Operations in CentraView.

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

Figure 9: Table books.

- the caller of the *getAuthor* method may forget to check the returned value. If the method returns *null* due to error at the database level the application will crash.
- if the caller checks the returned value, the idiomatic mechanism for handle exceptions hampers the maintenance of the source code.

According to (Bruntink et al., 2006), a system without a proper exception handling is likely to crash continuously, which renders it useless for practical purposes. In this paragraph we introduce an approach for detecting missing thrown exceptions within the data source layer of enterprise systems. It provides us

```
class IdiomaticBookDataSource {
 public String getAuthor(int id) {
  try {
   String query;
   ...
   query = "SELECT author from books " +
           "WHERE ID=" + id;
   ResultSet rs;
   rs = statement.executeQuery(query);
   return rs.getString("author"); }
  catch (SQLException e) {
   return null;  //or return ""; }}
 ... }
```

Figure 10: Class IdiomaticBookDataSource.

with those methods which make use of an idiomatic style for dealing with exceptions and, consequently, are good candidates for refactoring. In (Mortensen, 2007) the authors propose a technique for refactoring return code idiom for exceptions with aspects and present the benefits. The methods detected by our approach might be refactored according to the presented technique. In (Robillard and Murphy, 2003) is introduced a mechanism for inspecting the structure of exceptions in Java programs; based on the extracted information it allows us to detect and correct problems like unused handlers. In this context we want to emphasize that the our approach does not address problems relating to the use of exceptions in the source code; instead, it points out situations where the use of exceptions is missing from the source code.

The detection of the methods from the data source layer which use an idiomatic style for handling exceptions (*i.e.*, the missing thrown exceptions flaw is encountered) is done according to the steps presented next.

1. We detect the methods responsible for retrieving/storing the persistent data from the relational database. These methods belong to the data source layer.
2. We select from the previous group only the methods which do not throw exceptions, as we believe this rule quantifies the fact that the clients(*e.g.*, callers) of these data source methods are not informed about special situations using exceptions.

We consider that from the provided methods a special attention is requested by those whose return type is *void* because they might cause hidden defects. This is due to the fact that their callers are unable to check if the requested operation has been properly executed upon the persistent data.

3.2.2 Results

Findings in TRS. This system has 25 methods belonging to the data source layer, none of them throwing an exception. Thus, all of them use an idiomatic style for reporting exceptions. For example, we encounter a method called *cardCredit(long cardNo)* which returns the amount of money available from a particular card (see Figure 11). As we may notice, the method returns zero if:

- the amount stored by the card *cardNo* is zero.
- the card *cardNo* does not exist.
- the SQL command was not executed, due to an improper connection to the database or to a syntax error (the syntax of the embedded SQL can't be verified by the compiler).

Consequently, it is impossible for the callers of *cardCredit* to determine the exact meaning of the return value and usually they assume the first case was encountered.

```
public double cardCredit(long cardNo) {
 double credit=0;
 try {
   String sql;
   sql="SELECT Credit From Credit_card" +
       " WHERE Number=" +cardNo;
   PreparedStatement pst;
   pst = db.prepareStatement(sql);
   ResultSet rs = pst.executeQuery();
   while (rs.next()) {
    credit=rs.getDouble("Credit");
    return credit; }
   ... }
 catch(SQLException ex) {
  System.out.println("SQL Exception"); }
 return credit; }
```

Figure 11: Method cardCredit from TRS.

We encounter also a method which performs some operations upon the persistent data and returns *void*. Thus, its callers are forced to assume that no errors occur among the communication with the database, a practice which often proved to be a source of hidden defects.

Findings in Payroll. This system reveals a small number of methods (less than 1 percent) which belong to the data source layer and do not throw exceptions.

Findings in CentraView. In this system we encounter a lot of methods belonging to the data source layer which deal with exceptions in an idiomatic way. For example, we find that method *addNewSearch* inserts into table *search* a new request from the user. If the operation succeeds, it returns the newly unique id assigned to the request, otherwise *-1*. Another example we notice is method *getAccessEntityList* whose return type is a collection. This method has 146 LOC

(Lines of Code) and performs more than one SQL operation upon the involved table. Consequently, if an empty collection is returned from *getAccessEntityList*, it is impossible for its callers to establish if it is an empty collection because no data that meet the requested criteria were found or an abnormal situation occurred.

From the reported methods which throw no exception and return *void* we illustrate an interesting case in Figure 12. This method is a *private* one, being called from two other methods – *addFile* and *updateFile*. The first mentioned method collects the data which have to be inserted into the table, establishes the corresponding *id*, inserts the data (by calling *addFileLink* from Figure 12) and returns the generated *id* or zero if some error occurs regarding the assignment of the *id*. But the unexpected part is that this method (*addFile*) returns the associated *id* without checking that the data actually have been introduced into the database – the called method which inserts the data (*addFileLink*) returns void, being unable to report any error that might occur. Thus, the system's users will receive an *id* for their request, without ensuring that the request has actually been recorded into the table. In the context of this example we want to emphasize that it is also a case of an inconsistent idiomatic value returned within the application – we have previously showed an example which returns *-1* in case of encountering an error when inserting data into a table.

```
private void addFileLink(int fileId,...) {
 double credit=0;
 try {
   d1.setQuery("insert into"...);
   ...
   d1.executeUpdate(); }
 catch(Exception ex) {
  logger.error("[addFileLink] " + e); }}
```

Figure 12: Method addFileLink from CentraView.

Another example we notice is method *exportTable*. It has two parameters (*filePath*, *tabname*), the last being the name of a table whose information is intended to be saved into the file *filePath*. Because the method doesn't throw any exception and returns *void* its clients will not be able to determine when dealing with an empty file if the queried table stores no data or an error involving the database operation was encountered (*e.g.*, table *tabname* not found).

4 RELATED WORK

A tool specifically designed for database reengineering is proposed in (de Guzman et al., 2005). The main difference between this approach and our approach resides in the fact that our approach provides a meta-model for representing relational databases and also the interactions between an object-oriented programming language (at this moment, Java) and the databases.

A recent approach that quantifies the quality of the database schema and detects design flaws related to the database usages is described in (Delplanque et al., 2017). The approach is accompanied by a tool named DBCritics. Another recent approach accompanied by a tool named SQLInspect (Nagy and Cleve, 2018) for detecting SQL anomalies in the source code is introduced in (Nagy and Cleve, 2017). Our approach complements the existing analyses by providing new dimensions: visualizing how tables are accessed in the source code and finding improper usages of exceptions.

Currently there are also enterprise systems where the persistence layer is manipulated by frameworks like Hibernate (Bauer and King, 2007). In this context we want to emphasize that our approach can be applied upon enterprise systems 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 (and, consequently, we need to develop and use a different model loader) also from existing configuring XML files.

5 CONCLUSIONS AND FUTURE WORK

In this paper we decribe a specific meta-model for representing enterprise software systems in order to facilitate the process of reverse engineering upon these systems. As applications of the described meta-model (i) we introduce two visualizations regarding the accesses of tables in the source code, and (ii) we capture some improper usages of exceptions in the source code.

In the future, we intend to (i) extend the tool support in order to be able to use it upon enterprise systems written using other technologies (.NET, different persistency providers, different communication techniques), (ii) continue the evaluation of the introduced approaches against other enterprise systems, and (iii) 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.

REFERENCES

Ambler, S. and Sadalage, P. (2006). *Refactoring Databases: Evolutionary Database Design.* Addison-Wesley.

Bauer, C. and King, G. (2007). *Java Persistance with Hibernate.* Manning Publications.

Bruntink, M., van Deursen, A., and Tourwé, T. (2006). Discovering faults in idiom-based exception handling. In *Proc. International Conference on Software Engineering (ICSE)*. ACM Press.

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

Delplanque, J., Etien, A., Auverlot, O., Mens, T., Anquetil, N., and Ducasse, S. (2017). Codecritics applied to database schema: Challenges and first results. In *Proc. International Conference on Software Analysis, Evolution and Reengineering (SANER)*.

Fowler, M. (2003). *Patterns of Enterprise Application Architecture.* Addison-Wesley.

Marinescu, C. (2007). DATES: Design analysis tool for enterprise systems. In *Proc. IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*.

Marinescu, C. (2013). Should we beware the exceptions? an empirical study on the eclipse project. In *Proc. International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*.

Marinescu, C. and Jurca, I. (2006). A meta-model for enterprise applications. In *Proc. International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE Computer Society Press.

Marinescu, C., Marinescu, R., Mihancea, P., Raţiu, D., and Wettel, R. (2005). 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.

Mihancea, P. (2010). A Novel Client-Driven Perspective on Class Hierarchy Understanding and Quality Assessment. In *Ph.D. Thesis*.

Mortensen, M. (2007). Refactoring idiomatic exception handling in C++: Throwing and catching exceptions with aspects. In *Proc. International Conference on Aspect-Oriented Software Development*. ACM Press.

Nagy, C. and Cleve, A. (2017). A static code smell detector for sql queries embedded in java code. In *Proc. International Working Conference on Source Code Analysis and Manipulation (SCAM)*.

Nagy, C. and Cleve, A. (2018). Sqlinspect: a static analyzer to inspect database usage in java applications. In *Proc. International Conference on Software Engineering: Companion Proceeedings (ICSE)*.

Raţiu, D. (2004). *Memoria: A Unified Meta-Model for Java and C++.* Master Thesis, "Politehnica" University of Timişoara.

Ramakrishnan, R. and Gehrke, J. (2002). *Database Management Systems.* McGraw Hill, second edition.

Robillard, M. P. and Murphy, G. C. (2003). Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology*, 12.

Rugaber, S. and Wills, L. (1996). Creating a research infrastructure for reengineering. In *Proc. IEEE Working Conference on Reverse Engineering*.