Ville Laitila

Understanding and Analyzing SQL/CLI Database Usage of Java Software: Empirical Study

Master's Thesis
in Information Technology (Software Engineering)
21st March 2005

University of Jyväskylä

Department of Mathematical Information Technology

Jyväskylä

URN:NBN:fi:jyu-2005103

Author: Ville Laitila
Contact information: vimalait@cc.jyu.fi
Title: Understanding and Analyzing SQL/CLI Database Usage of Java Software: Empirical Study
Työn nimi: Tietokannan SQL/CLI-käyttötavan ymmärtäminen ja analysointi Java-sovelluksessa: empiirinen tutkimus
Project: Master's Thesis in Information Technology (Software Engineering)
Page count: 91
Abstract: This thesis focuses on SQL/CLI database usage of Java software. SQL/CLI database usage requires a relational database. A relational database and an object-oriented Java program is not a simple combination. This situation produces challenges for software maintenance. This thesis provides description about the problems of database application maintenance The empirical study of the thesis aims at finding out the most significant problems and the most essential information needs of understanding database usage in the target system.
Suomenkielinen tiivistelmä: Tässä tutkielmassa keskitytään Java ohjelmiston SQL/CLI-käyttötavan ymmärtämiseen ja analysointiin. SQL/CLI-käyttötapa edellyttää tietokannaksi relaatiotietokantaa. Relaatiotietokanta ja oliokeskeinen Java-ohjelmisto ei ole yksinkertainen yhdistelmä. Tämä tuo haasteita sovelluksen ylläpidolle. Tutkielma kuvaa tietokantasovelluksen ongelmia ylläpidon kannalta. Empiirinen osuus keskittyy selvittämään, mitkä ovat merkittävimmät ongelmat ja oleellisimmat tiedontarpeet tietokannan käytön ymmärtämisessä kohdejärjestelmässä.
Keywords: software maintenance, empirical study, program comprehension, reverse engineering, database usage, maintenance problems, maintaining database applications, SQL
Avainsanat: ohjelmistojen ylläpito, empiirinen tutkimus, ohjelman ymmärtäminen, käänteistekniikat, tietokannan käyttö, ylläpidon ongelmat, tietokantasovelluksen ylläpito, SQL

Contents

1 Introduction

Information systems have become larger through history. The big size has had an impact on understanding the source code of the system [BH92]. Need for tools that help understanding has appeared. Several tools are currently available for program comprehension purposes.

Database applications are widespread on almost every domain where software exists. A database application handles persistent data of database and hence differs substantially from other software. There are lots of dependencies between the application and the database. The dependencies are not always very clear for the developer so there are certain needs for tools supporting maintaining and understanding database applications.

Data reverse engineering aims at finding out what information is stored and how the information can be used in different contexts [MJS+00]. **Database reverse engineering** is data reverse engineering process where the target of reverse engineering is persistent data structures of an information system that uses database management system (DBMS) [MJS+00]. These research areas provide useful information about the information needs that developers or maintainers have.

Most of the current programming tools, IDEs (Integrated Development Environment) and CASE (Computer-Aided Software Engineering) tools do not exploit the reverse engineering research. When inspecting database reverse engineering research, the situation is even worse. What is most important, most reverse engineering tools available do not take into consideration either databases or database usage, which sets the database application developers and maintainers in a difficult position.

This thesis aimed at finding clues of what could be proper support for database application maintainers. The theoretical background of this minorly researched area was clarified. Conversation of more practical software development and maintenance tools was tried to initiated. Practically, the research aimed at finding common problems in the maintenance of applications that use database. Then the information needs were investigated related to the problems. The research questions were 1) what the common problems in this type of maintenance are and 2) how the problem solving process could be made easier. The hypothesis was that there are certain problems that occur e.g. in corrective and additive maintenance. Some problems are more typical in corrective maintenance and vice versa.

The research method was as follows. A query form was sent for 48 maintainers of Korppi system. Korppi is a web-based information system developed, maintained and used in the University of Jyväskylä. The system consists of Java code, JSP code and a relational

database. The maintainers were asked how problematic certain software maintenance tasks were related to database usage.

The research area is quite large covering software maintenance, object-oriented systems, relational databases and, most importantly, database usage. Database usage lacks a widely accepted theory. Also, *maintenance problems in database usage* is an untouched area where this thesis tries to open the ground.

This thesis is organized as follows. Database applications are described in chapter 2. Then relational databases and reverse engineering them are discussed in chapter 3. Reverse engineering of OO, Java software and database applications are briefly described in chapter 4. Understanding database usage is discussed in chapter 5. Software maintenance of database applications and problems of software maintenance are described in chapter 6. The conducted empirical study is described in chapter 7. The results are analyzed in chapter 8. Finally, chapter 9 represents the conclusions.

2 Database applications

This chapter aims at describing the key concepts of database applications. Background of the topic is covered in section 2.1. Section 2.2 describes Java database applications and the most important concepts of database applications. Characteristics of database applications are discussed in section 2.3

2.1 Background

With *database application* we mean here software which provides a user interface to a database, though the term is also used to mean software dedicated to managing databases. These types of systems are also called database programs, database intensive applications, data-oriented applications or more generally, information systems. The database is typically the most essential part of such application.

The amount of database applications is increasingly high. Especially web database applications have become popular, *e.g.* thousands of web markets are this type of applications. Web database application provides web-based user interface and therefore involves more technologies, *e.g.* JSP handling the HTML-based user interface. Figure 2.1 shows an example of the architecture of such application. The arrows of the figure represent usage relationships.

Relational database has been the synonym for database. It is still widely used in various application areas, though newer database technology *e.g.* object-oriented database has been predicted to take over the market. This thesis is limited to relational databases and Java systems using them.

The target system of this research is a web application using JSP technology for user interface, Java code for business logic, JDBC [EHF01] (Java Database Connectivity) for database usage and PostgreSQL for a database management system. Therefore, embedded SQL is not part of the research area.

This setting brings us specific maintenance problems that exist in the majority of maintenance tasks. This issue is more precisely discussed in the following sections. The main scope of this research is to find out the problems and the information needs related to them.

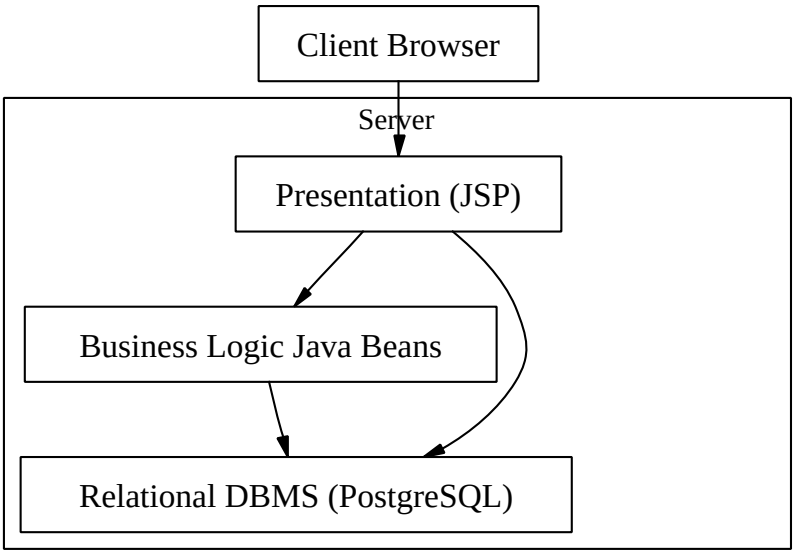

Figure 2.1: An example of web database application.

2.2 Java database applications

Relational database applications can be implemented in Java language with various ways. Subsection 2.2.1 introduces various ways how Java applications can be combined with relational databases. Then JDBC (Java Database Connectivity) is introduced in subsection 2.2.2 and a little example is given.

2.2.1 Java and relational databases

Java [GJS96] is an object-oriented programming language widespread over the world. The syntax is similar to C while some of the features are based on *e.g.* Smalltalk and C++ [Gos97].

There are several ways to build a relational database application. In Java, there exists at least three ways:

- Program uses call-level interface (*e.g.* JDBC) in order to communicate with the database. The SQL code using the database is written in the Java code as `String` objects.
- Embedded SQL (SQLJ) manages the database usage. The program is compiled in two phase because the SQLJ code is first translated into JDBC calls before the actual compilation.
- Classes are mapped to database tables with object-relational mapping (ORM) software.

2.2.2 JDBC database usage

In a Java application, database may be used by SQL and Java Database Connectivity (JDBC) [Kon98]. An SQL statement is passed as a parameter of type `String` in the JDBC API calls. There are several other APIs that are designed to manage database access, e.g. Open Database Connectivity (ODBC) [Mic95] but they are not widely used. JDBC API is object-oriented, which means practically that the API consists of classes which contain methods. With an OO API, database is always used with method invocation (call). JDBC is also vendor-neutral, dynamic SQL interface. Dynamic SQL means that SQL statements are constructed and evaluated during run-time. [IBM02, XOP95b]

What database usage is in practice? Figure 2.2 shows an example of database usage. `Connection`, `DriverManager`, `ResultSet` and `Statement` are classes and belong to the JDBC API. They are part of *java.sql*-package.

```
1  Class.forName("oracle.jdbc.driver.OracleDriver");
2  String conS = "jdbc:oracle:thin:@server:1000:STS";
3  Connection con = DriverManager.getConnection(conS,"USER","PASSWORD");
4  String query = "SELECT lastname FROM person WHERE personid ="+personID;
5  Statement statement = con.createStatement ();
6  ResultSet rs = statement.executeQuery(query);
7  while ( rs.next() ) {
8    System.out.println("Name is " + rs.getString("lastname"));
9  }
10 statement.close();
11 con.close();
```

Figure 2.2: Code example of database usage.

Line 1 loads driver for database connection. Then connection is initialized (line 3) based on the `conS` string (line 2) that has specific information about the database location and port number. In line 4, a query is formulated. The query consists of a condition that is determined on runtime because `personID` variable is a part of it. Line 5 creates a `statement` object which represents the query. The `statement` object executes the given query in line 6. Practically, the statement is sent to the RDBMS which executes it. Lines 7-9 are related to handling the results of the query and the connection is closed in lines 10 and 11.

The example is trivial but two things must be noted: First, it is important to understand call (invocation) relationships because the database is used by calls. Second, the roles of the JDBC classes and the meanings of their methods must be understood.

A sequence diagram in figure 2.3 shows the scenario of the code example. Sequence diagram shows invocation relationships between classes in a dynamic sequence. The drawback of the sequence diagram is, that it shows the calls to the database API same way as the other

calls between objects. However they have a different meaning because they define the model of the database usage.

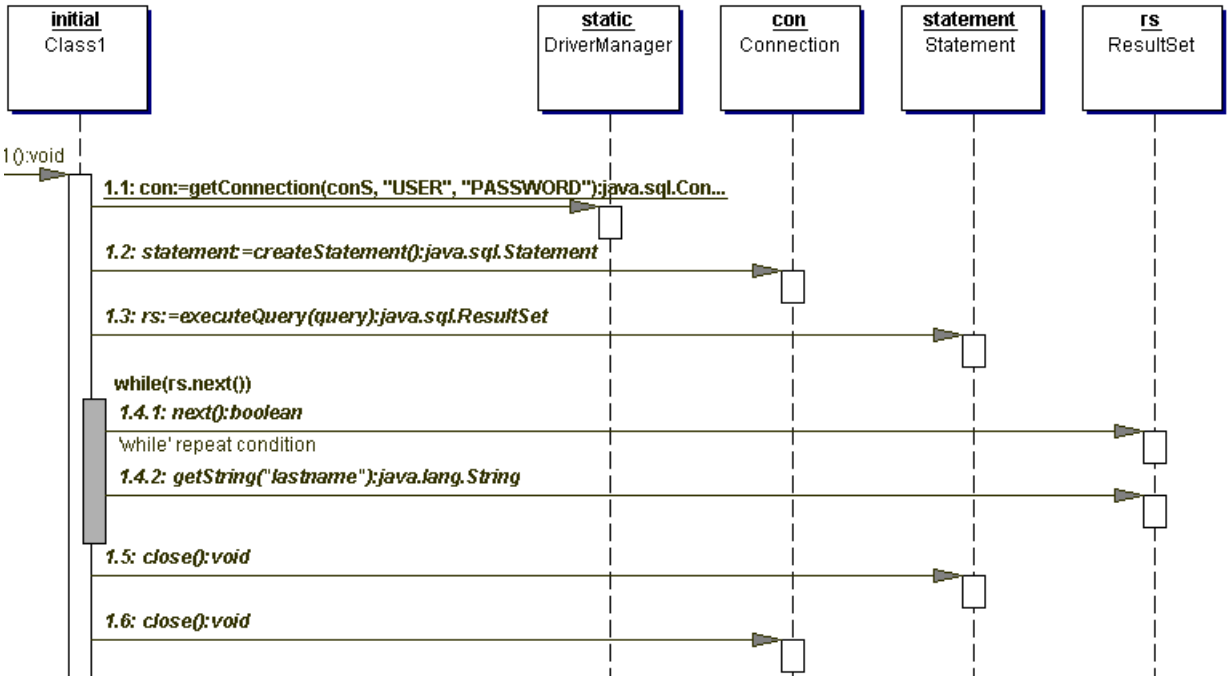

Figure 2.3: Sequence Diagram of code in figure 2.2.

The current widely used version of JDBC, version 3.0 [EHF01] includes tens of classes and interfaces. It consists of two packages: `java.sql` (core package) and `javax.sql` (optional package). The core package is a part of the Java 2 SDK.

The core package contains one interface (`Driver`) and three classes (`DriverManager`, `SQLPermission` and `DriverPropertyInfo`) related to establishing connection. There exists five interfaces for sending SQL statements to a database and one interface for retrieving the results of a query. There are many interfaces and classes for mapping types between Java and SQL, and custom mapping of user defined types (UDT). The API includes also meta-data interfaces, which provides information about the structure of the database. Error situations may be handled with various exceptions, *e.g.* `SQLException` and `SQLWarning`. [EHF01]

2.3 Characteristics of database applications

Database applications are generally more complex than applications that do not use database. Database application which uses database with SQL/CLI, may be affected by impedance mismatch. The application contains two type systems or programming models [NHR99].

The relationship between the database and the application may be described by *data dependency*. With strong data dependency, the early binding checks the dependencies between the application and the database. Any mistake in source code is detected at compile time. Also, all preparation related to DB operations are shifted to compile time, which is good for run time performance. Strong data dependency is gained using embedded SQL and precompilers. [NHR99] Weak data dependency is achieved with SQL/CLI. Compilation time is shorter and the "late" binding prevents from detecting DB schema (structure of database) related errors at compile time. [NHR99]

There exists tree major differences between database applications related to the relationship between the database and the application. They are in fact some sort of dependencies between the database and the applications. Firstly, relational database management systems provide own dialects of the SQL language. The dialects are mostly identical when considering the most widely used SQL statements. It is typical that data types and special functions differ.

Secondly, there are differences in call level interfaces. Interfaces vary in different programming languages and with different database vendors. There are, for example, JDBC [Ree97], DB2 CLI [IBM02] and ADO (ActiveX Data Objects). There are several interfaces for single language and one interface may be used by many systems with different programming languages. For example, ADO can be used from several languages, *e.g.* Visual Basic and C++. Also, a Visual Basic program can use relational database with *e.g.* ADO or RDO (Remote Data Object) [IBM02].

The third difference is the schema of the database. The names of the database tables and columns belong to database physical schema. Usually every application has its own schema, but there are also information systems where many applications use the same database and are therefore similarly dependant. Generally, business applications can not be built to be fully independent of the database schema. So, when modifying the schema, (*e.g.* changing a name of the column) source code of the application must be modified accordingly.

By analyzing these three dependencies, we can find out how strong the relationship between the database and the application is. When the relationship is strong, it is not easy to reengineer the application to use a different kind of database or CLI. Also the application with the strong relationship may be much more difficult to maintain. In that case, changes in application may require many changes in database and vice versa.

3 Relational databases

This chapter describes the key concepts of relational databases. Relational databases contain the information stored in the tables which are formulated by rows and columns. There are however more formal model behind this. A relational database may contain both data and functionality.

Section 3.1 presents basics of conceptual data modeling and describes entity-relationship (ER) diagram. Section 3.2 describes briefly the relational data model. Database schemas and data independence are discussed in section 3.3, and more specifically, relational database schema is described in section 3.4. The **primary** and **foreign keys** are also handled in the section. The role of the database and other database related issues are discussed in section 3.5. Section 3.6 describes SQL (Structured Query Language) as a way to access a relational database. Functionality stored in relational databases are described in section 3.7. Finally, section 3.8 presents relational database reverse engineering which is an important issue related to database comprehension.

3.1 Conceptual data modeling

A data model is a collection of concepts. For databases, data models are used to describe the structure of the database [EN99]. A relational database application consists of multiple techniques. Program functionality and the temporary data related to it may be handled with programming language and its type system. The persistent data is handled with database technology, relational data model and database query language.

Because of historical reasons, the database and application technologies have gone separate ways and developed their own modeling techniques and representations for the conceptual model. The UML is a popular notation for application developers who work with object-oriented systems. It does not only model data but also the structure and behavior of the system. The database society has developed the Entity-Relationship (ER) model [Che76] and its successors. There have also been approaches in order to unify the models.

Entity-relationship (ER) diagrams and UML class diagrams are both based on the analysis process of the problem domain. The output of the analysis process is called conceptual model. The ER diagram consists of *entities*, *relationships* and *attributes*. An entity represents a concept of the real-world, while a relationship among two or more entities represents interaction among the entities. An attribute is an important property of some entity. The ER

model, relational model and object-oriented model are compared later in section 5.1. [EN99]

The conceptual model of the problem domain can be achieved from the existing system using database reverse engineering tools [HHH+97, HEH+98]. They are focused on database analysis. Therefore the database reverse engineering tools may not be very helpful for a maintainer who has to do changes to the source code of the application. Database comprehension and database as a source of information for the conceptual model should not be ignored when reverse engineering database applications [HEH+98]. This issue is discussed further in section 4.6.

3.2 Relational data model

The relational data model, or shortly relational model is one of the most common database models used in information systems [EN99]. Relational databases are based on the model. There are two variants of the model, formal or informal model. The formal model represents the database as a collection of **relations**, while the informal model uses term **table** for relation. The differencies of the terminologies are described later in section 5.1.

The relational model is based on predicate logic and set theory [Cod70]. A relational database stores information in **tables** (or *relations*). A table has fixed amount of **columns** which are used to store certain information. The information of the table is stored in **rows** (also called *tuples*). [EN99]

3.3 Database schema and data independence

The description of a database is called the **database schema**. Elmasri and Navathe [EN99] specifies a database system architecture called the three-schema architecture. It divides the schema to three levels: internal (physical storage structure), conceptual (database structure described for a community of users) and external level (database described for a particular user group). [EN99]

The architecture supports data independence. There are two kinds of data independence: logical and physical. Logical data independence makes database modification tasks to affect only to the selected objects and other data remains untouched. With physical data independency, changes to the internal schema do not present changes in the other levels of the schema. [EN99]

3.4 Relational database schema and identity

A relational database schema S is a set of relations $S = \{R_1, R_2, \ ..., R_m\}$. The schema also contains a set of integrity constraints. Integrity constraint states that none of the **primary key** values can be null. Primary key is described in the next section. [EN99]

Relational database provides identity for data by primary keys. A primary key is a subset of the relation's attributes. It is such subset that no distinct tuples can have the same value. This constraint is also called the *uniqueness constraint* and the primary key is sometimes called the *superkey*. A column or a group of columns of a table may be set to form a primary key. [EN99]

The concept of foreign key needs more clarifying. The **referential integrity constraint** manages the consistency among tuples of two relations. A foreign key of R_1 referencing to relation R_2 satisfies always these two conditions [EN99]:

1. The attributes in the foreign key have the same domain(s) as the primary key attributes of R_2 .
2. A value of the foreign key always occurs as a value of the primary key or is null.

It is said that the foreign key refers to another relation, but foreign keys may be used to refer to its own relation. [EN99]

The foreign keys are used to create relationships between the relations. Types of these relationships may be 1:1, 1:M or N:M. These types are called cardinality constraints (also used in UML models). The cardinality 1:1 means that there is one-to-one row correspondence between the tables. For given tuple in R_1 it can be related to only one tuple in R_2 and conversely. With 1:M relationship between R_1 and R_2, a tuple in R_1 may relate to any number of tuples in R_2 but each tuple in R_2 must have only one tuple in R_1. The cardinality N:M defines that R_1 may contain multiple tuples for a tuple in R_2 and conversely. [EN99]

3.5 Database as a part of the system

Databases are used mostly because of data persistence. The other reasons for using separated database servers are distributing, sharing data with multiple clients or applications or data replication.

Data in the objects of the application are temporary. The data is lost when the application is terminated. *Persistent data* is data that persists over one data management session in the database [XOP95a]. Book [Ree97] describes that persistence is the act of making the state of an application stretch through the end of this process instance of the application to the next. Application is made persistent by recording its state in a data store. There are many ways to achieve data persistence. The data store may be *e.g.* a file, a relational database,

object-relational database or an object-oriented database. This thesis focuses on relational databases.

In some OO systems, relational database is mapped to the classes of the application so that the objects of the OO system are made persistent [HHH+97]. An object is persistent, if it survives the termination of program execution and can later be loaded to another program [EN99]. This mapping is called *object-relational mapping* (ORM). It is discussed more in section 5.8.

3.6 Structured Query Language

Structured Query Language [ISO03] is not a query language but a database language. It has multiple features, *e.g.* for queries, user control management and data defining. SQL is widely used to access relational databases. It consists of statements that define, retrieve and manipulate data. Practically, SQL statements of an application that uses database are used to retrieve, update, add and remove the database data. These operations manipulate the tables of a relational database.

Widely used SQL statements for these purposes are as follows:

– `SELECT` retrieves information (by running *queries*) from one or more tables.

– `UPDATE` alters information in a table.

– `INSERT` adds row(s) to a table.

– `DELETE` removes row(s) from a table.

There are many SQL standards and several RDBMS (relational database management system) vendors. Most of the vendors provide a language that is compatible with one or several SQL standards. Some dialects of SQL also support using object-relational databases. Object-relational model combines features from object model and relational model. The model is not discussed here. This thesis is focused on relational databases.

The first SQL standard is SQL-86 and after that there have been versions SQL-89, SQL-92, SQL-99 (also called SQL:1999) and SQL:2003 [ISO03]. The newer standards (SQL-92 and SQL-99) include also features like user right management and controlling transactions but they are not discussed here. SQL:2003 standard adds new features to the language (*e.g.* new sorts of columns, new data types and table functions) [EMK+04]. The existing widely used statements (*e.g.* `SELECT`, `UPDATE`) that retrieve and manipulate data remain unchanged.

SQL/CLI (Structured Query Language / Call Level Interface) is a subset of the SQL-92 standard. It has been standardized by X/Open Company Ltd in 1995. It has also become a part of the SQL-99 standard. It is often called *not-embedded SQL access method*. No

precompiler is required for SQL statement processing when compiling application because the database usage is coded in function calls. There are several functions (*e.g.* execute SQL, open connection) that define how to use the database.

SQL/CLI usage differs from embedded SQL so that SQL/CLI takes SQL statements to the variables of programming language. SQL/CLI usage allows developers to build SQL statements dynamically. The good (and bad) side of embedded SQL is that SQL statements are written to the application statically. This means that SQL can be checked before compiling the application and SQL statements are more visible in the source code. However, some embedded SQL implementations make it also possible to build SQL statements during application execution, *dynamically* [IBM02]. Dynamic SQL is more difficult to analyze than embedded static SQL but dynamic SQL allows building software with fewer lines of code (in some cases). There are also other reasons of using dynamic SQL or embedded SQL. They are not described here but *e.g.* [IBM02] gives a good overview.

3.7 Functionality in database

Most of the relational database management systems allow writing functionality into the database. Subsection 3.7.1 describes **stored procedures**, a kind of functions stored in the databases. **Triggers** which may be used to enforce additional constraints are described in subsection 3.7.2.

3.7.1 Stored procedures

Stored procedures are collections of executable SQL statements. There are several vendor-specific different extensions for writing them because the SQL standard itself is quite limited. [EN99] The purpose of stored procedures is to store the functionality which is more closely coupled to data straight into the database. They can be used in order to reduce the amount of data to be send to database server. The procedures may be called from the client application directly, or they can be used from **triggers** which are described later.

Stored procedures may be written with PL/SQL, which is a procedural language extension to SQL by Oracle. It offers features like data encapsulation, information hiding, overloading and exception handling. [EN99, chapter 10]

PL/SQL language is block-structured which means that the basic unit of the language is a block of statements. A block contains logically related declarations and statements. A PL/SQL block has the following structure [EN99]:

```
[ DECLARE
---declarations ]
```

```
BEGIN
---statements
[ EXCEPTION
---handlers]
END ;
```

The block begins with the declarations part, which is optional. The part may contain variable declarations with SQL data types or additional PL/SQL data types. The part which begins with BEGIN is an executable part and may contain procedural code like conditional, iterative and sequential control-flow statements. The exception part takes care of the error situations of the executable part. An occurring exception forces the halt of the normal execution and handling the exception. [EN99, chapter 10, section 5]

3.7.2 Triggers

A **trigger** (part of SQL2) is a condition and an action to be taken if the condition is met [EN99, chapter 23, section 1]. A trigger is also called as an active database rule. Normally triggers are used in order to manage data integrity or prevent unreasonable data being stored to database. There are several versions of trigger implementations in commercial RDBMS.

Triggers are specified with **Event-Condition-Action**, or ECA model:

- An **event** is usually an update operation that triggers the rule. There are also temporal events, *e.g.* periodic time, every day at 6:00 am and other kinds of external events.
- A **condition** determines whether the action should be executed. If no condition is specified, the action will be executed immediately after the event occurs.
- An **action** is a sequence of SQL statements or a database transaction or an external program that will be executed.

Here is an example of ECA with Oracle notation. The database schema consists of two tables: `EMPLOYEE` and `DEPARTMENT`. The columns of the tables are as follows.

```
EMPLOYEE
NAME SSN SALARY DNO SUPERVISOR_SSN
DEPARTMENT
DNAME DNO TOTAL_SAL MANAGER_SSN
```

A definition of a trigger `TOTALSAL` is given here.

```
CREATE TRIGGER TOTALSAL
AFTER INSERT ON EMPLOYEE
FOR EACH ROW
WHEN (NEW.DNO IS NOT NULL)
```

```
UPDATE DEPARTMENT
SET TOTAL_SAL =TOTAL_SAL +NEW.SALARY
WHERE DNO = NEW.DNO
```

The trigger TOTALSAL in the example is basically "Always where adding an employee (event), update the total salary of the department (action) if the department is known (condition)." [EN99]

Triggers are important from the maintainer's view point because they may add data flow relations between the tables and therefore complicate software maintenance tasks. It is essential to manage data flow relations in understanding the software and the system as a whole, making changes and testing. For example, a tester must take care of the triggers when determining what parts of the application are to be tested [HMD01].

3.8 Relational database reverse engineering

Data reverse engineering aims at finding out what information is stored and how the information can be used in different contexts [MJS+00]. Reverse engineering the persistent data structure of an information system that uses DBMS is more specifically referred to as **database reverse engineering** (DBRE).

Subsection 3.8.1 discusses the nature of database reverse engineering. Then the importance of database reverse engineering is discussed in subsection 3.8.2. Finally, subsection 3.8.3 discusses about how program comprehension and database reverse engireening (or database comprehension) are related to each other in case of database applications.

3.8.1 Nature of DBRE

Database reverse engineering is a process of recovering the schema of the database [MJS+00]. In other words, DBRE aims at supporting database comprehension. Database reverse engineering uses, as input, database declaration text and program source code that uses database. The result of the process is a conceptual model of the database.

Database reverse engineering may be useful in understanding database usage if the database documentation is not adequate. Database reverse engineering can help *e.g.*, in understanding the meanings of the tables and columns of the database, which is essential for understanding database usage. Database reverse engineering can be thought as a part of reengineering databases. Reengineering database applications is a large area and not discussed here. Reengineering relational database applications to EJB based architecture are presented in [Lu02]. Article [TP96] describes database reengineering when application using embedded SQL is translated to use a domestic SQL API called Pst/DB (similar to SQL/CLI).

Relational database reverse engineering is a specific area of database reverse engineering. It involves the analysis of SQL data definition statements or existing physical structure of the database. Reverse engineering relational databases is described by Premerlani and Blaha [PB94]. The article proposes a process where existing relational database is reengineered to an object-oriented conceptual model. Premerlani and Blaha [PB94] conclude that automated batch-oriented compilers will not succeed. The process of reverse engineering databases requires flexible and interactive tools.

A generic methodology for reverse engineering databases is presented in [HEH+98]. The methodology consists of two phases. Data structure extraction is the first phase and produces the logical schema of the database as an output. The inputs are the data of database, program source code, physical schema of the database and data definition statements. The phase consists of both program and data analysis. Program analysis is required for detecting implicit data structures. [HEH+98]

The second phase, data structure conceptualization takes the logical schema as input and produces the conceptual schema of the database. Program analysis is required here in order to achieve a complete logical schema. The conceptual schema is an interpretation of the complete logical schema. The phase consists of transforming and removing non-conceptual structures, redundancies, technical optimisations and DMS-dependent structures of the logical schema. [HEH+98]

3.8.2 Importance of DBRE

When extending an existing relational database application there may be many open questions [AEP96]:

- How to use joins to navigate among relations?
- Where particular type of data is located?
- What other relations will be affected when changing this relation?
- What application code will be affected when changing this relation?
- What portion of the data does this application use in the database?
- Where is the best place to add data for a new or changed database application?
- What existing functions are affected by a proposed change?

Without good answers to the previous questions, maintainability degrades [AEP96]. Developers cannot find the right relations from database and end up adding redundant data to the database.

3.8.3 Program comprehension and DBRE

Program comprehension and databases reverse engineering support each other in database applications [HEH+98]. Program comprehension techniques are required when reverse engineering database and also, in data-oriented applications it is essential to understand the database in order to fully understand the system.

Program comprehension is important in database reverse engineering. Finding queries from the source code may be essential for database reverse engineering, which clearly requires program comprehension. Procedural code analysis helps therefore in understanding the semantics of the data structures. Some information about integrity constaints and implicit data structures can be found from the source code of the program. Understanding the business rules and the problem domain may give essential hints to understanding the meaning of the database entities. [HEH+98]

For example, when reverse engineering a relational database into a conceptual schema, the essential part of the schema is what are the relationships between the tables. The information about the cardinalities of the relationships may be found from the source code where the queries are located. [HEH+98]

As a contradict, database reverse engineering in program comprehension is important issue. The understanding of the underlying database may ease the understanding of the functionality of the system. This issue is discussed in section 4.6.

4 Reverse engineering Java and OO systems

This chapter describes reverse engineering of Java and OO systems because code reverse engineering may be a helpful process for understanding database usage. Section 4.1 gives a definition for reverse engineering. Section 4.2 describes characteristics of OO (object-oriented) systems. Section 4.3 describes program comprehension issues of OO systems. UML (Unified Modeling Language) is shortly introduced in section 4.4 as a representation for supporting program comprehension. Understanding Java systems is handled in section 4.5. Finally, section 4.6 discusses the applicability of the current reverse engineering methodologies for database usage comprehension.

4.1 Reverse engineering

Reverse engineering aims at representing the software in a form which facilitates program comprehension. Chikofsky and Cross [CI90] defines reverse engineering as a process of analyzing a subject system to identify the system's components and their interrelationships and to create representations of the system in another form or at higher level of abstraction.

4.2 Characteristics of OO systems

OOPs (object-oriented programs) differ significantly from conventional (*i.e.* nonobject oriented) programs. In conventional programs, data is mostly used globally and it is separate from functions accessing it. The system is modularized based on its functionality. OOP is a paradigm that defines the modules of the system based on conceptual model of the problem domain [Sny93]. Code and data are encapsulated into *objects*. The code is represented as *methods* and the data as *attributes*. An object typically has state and behavior. The state is determined by its attributes and the behavior is determined by the operations (methods) of the object. Objects are generated, *instantiated* from a class that is a definition of a concept.

Many characteristics of OO systems make OO code reverse engineering different from traditional reverse engineering. The characteristics are described in [WH92] and [Tuo95]. Inheritance is a relationship based on the specialization of an existing class to define a new concept that is a special case of the existing class. The inherited class is called as *superclass* and the new class as *subclass* [Tuo95]. Dynamic binding means that the target of a

method call is not available statically (*i.e.* at compile time) and is a form of *polymorphism*. Polymorphism means that a variable of one type can hold also values of other types.

An OO system contains typically many small methods instead of a few number of large methods. This makes OO systems to resemble message-passing systems where methods send messages to methods with very little processing. Also, OO system tends to have many small modules instead of a smaller number of large modules. [WH92]

In a conventional system, there is a top-level module that calls other modules to execute certain functionalities. The top-level main module is usually a good place to start understanding system in order to achieve general understanding. In OO systems, there may not be real top-level main module where to start system understanding. [WH92]

4.3 Understanding OO systems

The characteristics described in section 4.2 lead OO system understanding to focus on finding chains of method invocations. A chain can lead over several different object classes up and down an inheritance hierarchy. Searching the chains is time-consuming and requires good understanding of inheritance dependencies and calling dependencies of the system.

Wilde and Huit [WH92] describe important dependencies of an OO system. A dependency in a software system is a directed relationship between two artifacts. In conventional programs dependencies can be classified as data dependencies, calling dependencies, functional dependencies and definitional dependencies [WH92]. In OOP, we add new entities that have dependencies: object classes, methods and messages. Variables may represent instances of an object class. Use of polymorphism and inheritance brings new kinds of dependencies into consideration. The dependencies are listed precisely in [WH92]. The dependencies exist in object-oriented languages, *e.g.* Smalltalk and C++. Wilde and Huit [WH92] describe the dependencies so that they can be applied to Smalltalk. Tuovinen [Tuo95] describes similar dependencies that exist in C++ programs. The dependencies exist similarly in Java language programs. Understanding Java is discussed later in section 4.5.

Dependencies can be classified as class-to-class, class-to-method, class-to-message, class-to-variable, method-to-variable, method-to-message and method-to-method dependencies. For example, method-to-method dependencies are “method M1 invokes method M2” and “method M1 overrides method M2”. Class-to-class dependency can be, *e.g.* “class C1 uses class C2”. It is important to note that in C++ and other statically typed OOP languages there is no concept of message. Therefore the dependencies that belong to class-to-message and method-to-message do not exist in those languages.

Calling hierarchies are a good tool for understanding conventional programs. Because of the nature of OO, calling hierarchies as such are not applicable. Inheritance hierarchy is an

important way to help understanding OO systems, but it does not show any other relationships between classes than inheritance. General system understanding of OO system could be achieved by inspecting graph of “Class Uses Class” dependencies. “Class Uses Class” dependencies help understanding the responsibilities of the classes through their relationships to the other classes.

One of the biggest problems of understanding an OO system is polymorphism and especially dynamic binding. In dynamically typed languages every method call is polymorphic. The target of a method call is necessarily not available at compile time. In statically typed languages (*e.g.* Java and C++), the information of the target is available up to certain degree at compile time. For example, if there is a variable of whose type is declared as class A, it can hold only objects of A or objects of the subclasses (direct or in-direct) of A. In languages with no static typing (*e.g.* Smalltalk), there are no type constraints attached to variables. The target can be virtually any method with same name and same number of parameters. The restriction of the target candidate can be done better in statically typed languages. It is difficult to find out exact calling relationships in OOPs if dynamic binding is used.

Wilde and Huit [WH92] suggest that using *external dependency graphs* improves the understanding of polymorphism. An external dependency graph divides methods to equivalence classes. Division is based on method names and dependencies between parameters and result type. With external dependency graphs we can restrict the number of target candidates. [WH92]

4.4 Unified Modeling Language

Unified Modeling Language (UML) is a graphical language for describing mainly object-oriented systems. It may be used for visualizing, specifying, constructing and documenting purposes [OMG03]. UML is applicable also to defining database schemas [LZ03, OMG03]. UML model can be helpful up to some degree in understanding the system. It focuses on describing objects (and classes) of the system. It includes both static and dynamic approaches.

There are various diagrams for many purposes with different approaches. The most popular diagram type, class diagram is a diagram that bases on describing the static structure of the target system. It describes the classes and the relationships between them. The UML sequence diagram (modeling dynamic aspects) describes a dynamic scenario of the system. It describes the scenario as a sequence of messages between objects (or *calls between the classes*) An example sequence diagram was presented in section 2.2.2 (figure 2.3).

UML may not be a good representation in understanding database usage because of various reasons. One reason is that UML does not describe the interaction between the application and the database with a special notation. Also, *transaction* do not belong to UML as a

separate concept.

Usefulness of UML for database applications can be raised by using *stereotypes*. There is a recommendation to use stereotype *persistent* with a class symbol in order to represent a relation [BRJ99]. A tuple of the relation (a row of the table) corresponds to an instance of the persistent class. There is also an approach [LZ03] to extend the UML metamodel with elements for modeling relational dependencies. The UML metamodel is extended with new stereotypes based on UML metaclass `Dependency` which abstract the dependencies between the tables of the database. These dependencies are inclusion dependencies, foreign key dependencies and functional dependencies. [LZ03]

4.5 Understanding Java systems

An experimental environment called Shimba supports reverse engineering of Java systems [SKM01]. Shimba environment integrates Rigi and SCED tools to analyze and visualize Java systems. Shimba collects and represents both statically available and run-time information. It uses both static and dynamic analysis. Shimba produces SCED sequence diagrams that correspond to the sequence diagrams of UML and also statechart diagrams. Sequence diagram shows the interaction between objects but statechart diagram gives information about the overall behavior of an object.

4.6 Applicability of reverse engineering methodologies for database applications

Database applications are a specific area of software where the database and its role is a significant. The current reverse engineering methodologies (*e.g.* [SKM01, WH92, HHH+97] are so general that they do not include concepts of database neither the concepts of database usage or transactions. So, they are not suitable for database applications. Database reverse engineering is fully different. It approaches the system from the viewpoint of database and often leaves the program without any attention. Some examples are [Alh03, CF03, AEP96, PB94].

While the gap between the methodologies exists, there are some useful approaches to bring them together when reverse engineering COBOL applications [HEH+98, Hen03, HHH+99, HEH+95]. These publications point out that program comprehension and database comprehension support each other, as discussed in section 3.8.3. While these publications emphasize that program comprehension is important for reverse engineering databases, there are not so many publications to declare the benefits of database comprehension for code re-

verse engineering.

One of the main purposes of database applications is to handle the database. The maintainer must be aware of the semantics of the tables and columns of the database in order to fully maintain the database handling code. Different visualizations like database usage graph may be useful when trying to comprehend the architecture of the application. [Hen03]

5 Understanding database usage

Database usage is mapping between data models. It is actions of read and write or some sort of data flow between the database and the application. Issues of understanding the database usage are described in this chapter. Database usage is far from simple, since it includes transactions, data manipulation language and mapping objects to tables.

Section 5.1 shows how different data model representations are mapped to each other. Section 5.2 discusses the most essential information related to database usage. Section 5.3 describes the different ways of SQL/CLI database usage in object-oriented systems. Logging down the dynamic SQL statements is an important maintenance method which is discussed in section 5.4. Transactions are discussed in section 5.5. Data dependencies are discussed in section 5.6 and a model of maintaining web database applications is described. Analyzing embedded SQL is discussed in section 5.7. Object relational mapping and object-relational impedance mismatch are described in sections 5.8 and 5.9. Data dictionaries are presented in section 5.10 for helping database application maintenance and development. Section 5.11 describes the essential artifacts of understanding database usage in a Java application using JDBC. Section 5.12 describes some important dependencies of database usage which are relevant to understanding the application and the database. Analyzing database usage is discussed in section 5.13 and finally in section 5.14, a way of visualizing database usage is presented .

5.1 Data model representations

Java is an object-oriented language, so it is quite natural to assume that Java software is written using object-oriented design principles, although there are some cases where Java has been used for writing almost fully procedural code. When developing such system, there are multiple terminologies and overlapping terms. Some of the terms are often mapped to each other, for example column of a table may be mapped to an attribute of a class.

Here we compare the terminology of object-oriented model, entity-relationship model and relational model. Table 5.1 describes the correspondence between the terminologies [EN94]. An entity of ER diagram can be found from an UML class diagram as a class. Also, a relationship may correspond to a class. If diagrams and classes have good names, the connection between these may be explicit. ER diagrams are used also in database design. The data of an ER diagram are transformed and normalized into a relational database schema.

Object-relational mapping is a technique which makes the bindings between the models more straightforward.

Table 5.1: Comparative terminology of data models [EN94]

Entity-relationship model	Relational model, formal	Relational model, informal	OO model
Entity type schema	Relation schema	Table description	Class description
Entity set	Relation state	Table	Collection of objects
Entity instance	Tuple	Row	Object
1:N relationship type	__ a	__ a	__ c
1:N relationship instance	__ a	__a	__ c
Attribute	Attribute	Column	Attribute
Value set	Domain	Data type	Atomic data type
Key	Candidate key	Candidate key	__ b
__ b	Primary key	Primary key	Object identifier
Multivalued attribute	__ b	__ b	Set constructor
Composite attribute	__ b	__ b	Tuple constructor

Letters a, b and c used in the table are described here.

a No corresponding concept; relationship is established by using foreign keys.

b No equivalent concept.

c No corresponding concept; relationship is established by using references.

5.2 Database usage

Database usage comprehension is a quite large area because it involves both database comprehension and program comprehension. While program comprehension is the mapping between the program and the problem domain [HEH+98], database usage comprehension is the mapping between the program and the database.

The most important issue in the database usage is the information stored in the database. The information is stored into the tables of the database. The purpose of the table is defined by *what* information is intended to be stored there. This prerequisite for understanding database usage relates to *database reverse engineering* which aims at building a conceptual model of the database.

Other important questions in database usage are as follows: What is the purpose of single database table or column? Which parts of the application use database (are dependent on database)? When (in which sequences) database is used? What part (what tables) of database

is used by the application? How (read, write) database is used? By answering to these questions we build a model of database usage.

Database usage involves also security issues. The data of database must not be visible to all users. When database is used such that queries are generated dynamically based on the input of the user, there is always a security risk that some secret data ends up to the query [CMS03]. For example, inserting a tautology (1 == 1) to a query, can result in a situation where the user gets all data entries of the associated tables instead of what the user ought to be able to see. Understanding the database queries and how they are generated is thereby important.

5.3 SQL/CLI database usage

Some important entities and interrelationships are described in figure 5.1. OOP consists of objects that are defined by classes which have methods and attributes. The classes may be involved in inheritance relationships which may result in a situation (in statically typed OO languages) where dynamic binding complicates the understanding of method invocation. In dynamically typed OO languages dynamic binding may complicate method invocation without inheritance. The methods of the classes use the database by generating SQL statements in String expressions or variables. In order to execute the statement, the method has to invoke certain methods of database API classes. The SQL statement is passed as a parameter in the invocation. The database API method passes the SQL statement for the database server and the server manipulates the database tables accordingly.

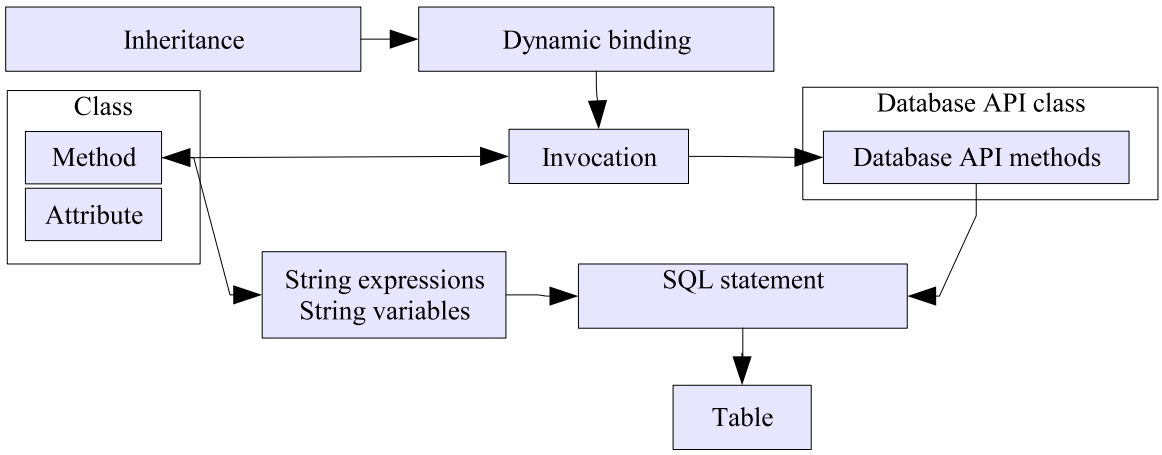

Figure 5.1: Important entities and interrelationships of SQL/CLI database usage in OOP.

Main issues in database usage are connecting to database, execution of an SQL statement and handling its results. Connection is usually handled through a specific class (often named

Connection) that belongs to the database API. It typically has operations that open and close the database connection. There can also be *a connection pool* that manages a number of open connections at one time. Thus, the database is used through a connection pool class that hides the pooling logic.

CLI Basic Control Flow [XOP95a] gives a good overview of SQL statement execution. The execution is described more practically from the programmer's perspective in [IBM02]. Executing SQL statement requires an open connection to the database. The SQL/CLI standard [XOP95b] defines that statement can first be prepared and then executed or executed directly. The preparation turns the character string form of the SQL statement into an executable form. Preparation makes it possible to execute the same statement repeatedly within a connection. The preparation allows also constructing parameterized statements. A parameterized statement is a query with *parameter markers*. Parameter marker is indicated by a question mark (?), and the place where *a host variable* (variable of programming language) is to be substituted inside an SQL statement. Executing directly means that no parameter markers are used in the SQL statement and the statement is executed directly.

Handling the results is performed through a specific data structure resembling a table. This structure is declared as a class in many OO APIs that base on SQL/CLI. For example, there is a class named `ResultSet` in JDBC API [Ree97]. `ResultSet` is one kind of a table that contains the fetched data in its columns. `ResultSet` has usually methods that allow iterating the table of results through, row by row.

The problem of understanding database usage is mapped to a problem of understanding source code. If there is information available about what classes (and more precisely: what methods) have database access responsibilities, the information can be used such that the system is inspected by searching for usages of the classes (methods).

Figure 5.2 shows a code example where several methods are involved in constructing an SQL query statement. Construction of the query is managed such that all the methods append some information to the `String`-variable that holds the SQL statement.

Line 2 in figure 5.2 creates `String`-class variable to hold the SQL statement. Line 3 has a call to a method `selectFromPerson` which formulates the beginning of the statement. Line 4 uses the beginning and appends the `WHERE`–condition to it. Method `whereConditionPersonid` creates the `SQL WHERE` condition in line 11. Then the statement is ready for execution and could be executed in line 5. This example shows that dynamic SQL can be created easily using the mechanisms of the programming language.

Parameter markers and SQL statement preparation is a feature of SQL/CLI interfaces. Figure 5.3 gives an example of using parameter markers and preparing a statement before execution. Lines 1 and 2 create a prepared statement object. It has two parameter markers that are attached on lines 3 and 4 with `setString` and `setInt` method invocations. At-

```
public void runSQL(Person person) {
  String SQLstmnt = "";
  SQLstmnt = selectFromPerson("lastname");
  SQLstmnt += whereConditionPersonid(person.getPersonID());
  // then execute it...
}
private String selectFromPerson(String field) {
  return "SELECT " + field + " FROM person ";
}
private String whereConditionPersonid(int personid) {
  return " WHERE personid = " + personid
       + " AND person.personid=questionnaire.personid";
}

```

Figure 5.2: Code example of several methods involved in constructing a query.

```
  PreparedStatement del = con.preparedStatement(
    "DELETE FROM person WHERE lastname = ? AND questionnaireid = ?");
  prepStmt.setString(1, "Norton");
  prepStmt.setInt(2, 4);
  prepStmt.execute();
  prepStmt.close();
```

Figure 5.3: Code example of using parameter markers.

tachment replaces the question marks of the statement with given parameters. The question marks are replaced in the order in which they exist in the statement and using the first parameter of the `setString` and `setInt` methods. So the order of the `setString` and `setInt` method invocation is meaningless. In this case, the final statement deletes all rows with `name` Norton and `questionnaireid` 4 from table `person` in line 5. The statement is closed in line 6. It could have been executed repeatedly using different attached values before closing.

5.4 Dealing with dynamic SQL by logging the statements

Dynamic SQL complicates database related functionality since the SQL statements are constructed at run-time. The statement may depend fully on the run-time state of the application. It is quite usual (at least in the target system) that the conditions of an SQL statement are generated dynamically depending on the user input.

In order to guarantee that the application produces valid and correct SQL statements, static checking of the statements is difficult but possible [GSD04]. However, it is more common to write debug messages to the application code which write the statements to log files before they are executed. If a statement is not valid and the DBMS gives an error message, log file contains the statement and it the error can be found.

Dealing with the effects of changes is a problem related to SQL statements. There are maintenance situations where the application source code has to be changed without affecting to the SQL statements. One method, used in the target system maintenance, is to write the SQL statement down *e.g.* to comments in the source code. Then, after the change, the statement can be checked by inspecting the log file containing the newly executed statement and comparing the old and the new statement. The problem here is, that if the statements are very dynamic, it is difficult to write down all the various forms of the statement. [Wyk03].

Instead of writing every log message to the source code where the query is constructed, a sophisticated wrapper interface for database access may provide the logging services. The target system uses classes `DB` and `PreparedDB` for executing SQL statements. These classes wrap the JDBC functionality and hence ease the use of database. Class `DB` provides SQL statement execution which corresponds to JDBC `Statement` interface. The class also includes logging mechanisms that may be swithed on and off. The class also hides the use of database driver from the other classes that use the database through it. The class hides the pooling of database connections and provides similar transaction interface that exists in the JDBC. The target system class `PreparedDB` inherits the `DB` and makes it possible to use prepared SQL statements with parameter markers. The usage of the class is similar to `PreparedStatement` interface of JDBC. Without this kind of separated database inter-

face, logging of SQL statements increases the size of the system.

Logging of SQL statements that are executed by JDBC `PreparedStatement` interface is quite problematic. A good logging entry should describe the statement exactly how it was executed after the parameter markers of the statement were replaced by the actual values. However, the interface does not provide such information, because the statement is sent to the DBMS without the actual values. In order to log down SQL statements, there are many alternatives, *e.g.* one may build the logging messages self in the code by concatenating the statement from parts, or write a method for replacing question marks with values. By doing this, duplicate code is created, and while changing the form of the statement, the logging code also has to be changed which is not meaningful. [Wyk03].

An extension [Wyk03] to the interface is presented which makes the logging less prone to error and tidies the source code. With the extension, logging features can be used quite easily. An example is presented in figure 5.4.

```
1   String sql = "select foo, bar from foobar where foo < ? and bar = ?";
2   long fooValue = 99;
3   String barValue = "christmas";
4
5   Connection conn = dataSource.getConnection();
6   PreparedStatement pstmt;
7
8   if(logEnabled) // use a switch to toggle logging.
9       pstmt = new LoggableStatement(conn,sql);
10  else
11      pstmt = conn.prepareStatement(sql);
12
13  pstmt.setLong(1,fooValue);
14  pstmt.setString(2,barValue);
15
16  if(logEnabled)
17     System.out.println("Executing query: "+
18       ((LoggableStatement)pstmt).getQueryString());
19
20  ResultSet rs = pstmt.executeQuery();
```

Figure 5.4: Code example of logging [Wyk03].

The example shows the usage of the logging features of the extension of `PreparedStatement` interface. Lines 1-3 initialize the variables holding the SQL statement and the values that are attached to it. Lines 5-6 trivially initialize the connection and declare the type of `pstmt` variable to be `PreparedStatement` which is the super class of `LoggableStatement`. A switch for turning logging off and on is used at line 8. The switch selects the right implementation of the `PreparedStament` interface. The values are attached to the statement

at lines 13-14. The statement is written to the log at lines 16-18 if the logging is switched on. Finally, the statement is executed at line 20. [Wyk03]

5.5 Transactions

A database transaction or simply transaction is a logical unit of functionality that accesses database [EN99]. A transaction includes one or more database operations which may be one of the four types: insertion, deletion, modification or retrieval. Transaction processing is needed for concurrency control in multiuser systems, *i.e.* to handle simultaneous access to tha data.

There is an analogy between transactions and sequences of UML sequence diagrams. Both describe the functionality of the system, dynamic behaviour. Elmasri and Navathe [EN99] defines that a transaction is a particular execution of a program on specific parameters. An UML sequence is a sequence of actions. A sequence diagram represents the sequence as messages between objects with a time axis [OMG03]. While the UML sequence is about interaction between the objects, a transaction is about interaction between the application and the database.

A transaction is an atomic unit of functionality that must be wholly completed or not done at all. The database management system must handle the transactions and their effects in order to provide undoing of transactions. There are several operations in order to process transactions [EN99]:

- `BEGIN_TRANSACTION` marks the beginning of the transaction execution.
- `READ` or `WRITE` specify read or write operations on the database. They are executed as part of the transaction.
- `END_TRANSACTION` marks the end of the transaction execution. This point requires checking of whether the effects of the transaction can be committed to the database or whether the transaction must be aborted.
- `COMMIT_TRANSACTION` marks a successful end of the transaction. This point indicates that the effects (updates) of the transaction can be applied permanently (committed).
- `ROLLBACK (or ABORT)` signals an unsuccessful end and forces undoing of the effects.

How transactions are supported in SQL and JDBC? A single SQL statement is always atomic since it is completed wholly or it is not executed at all and leaves the database unchanged. There is no explicit `BEGIN_TRANSACTION` operation in SQL but it is done implicitly when the statement is encountered. Either `COMMIT_TRANSACTION` or `ROLLBACK`

is required to be executed after the last SQL statement has been processed [EN99]. JDBC API does not require doing this, since it has auto-commit mode. Auto-commit means that JDBC driver does a transaction commit after each individual SQL statement when the execution of the statement is complete. The completeness is decided with following rules depending on the statement type:

- `INSERT`, `DELETE` and `UPDATE` statements are complete when they have been executed.
- `SELECT` statement is complete when any of the following conditions is true:
  - every row has been retrieved,
  - associated `Statement` object is re-executed or
  - another `Statement` object is executed on the same connection.

Transaction management may also be done at a level higher the JDBC driver by setting auto-commit mode disabled in `Connection` interface [EHF01]. In that case, each transaction must be ended by calling `commit` or `rollback` method of `Connection`.

There are also more settings that specify the transaction processing. The statement `SET TRANSACTION` of SQL2 specifies the access mode, the diagnostic area size and the isolation level. The settings are so specific that they are not described here but information about them can be found from [EN99].

5.6 Variable and data dependencies

Variable dependency graph is a weak, easy to compute version of dataflow diagram [HEH+98]. It is described as a useful representation for understanding to which variables an attribute of the database is connected. Each variable of the program is represented by a node and an arc represents a direct relation (usually dataflow) between two variables. The construction techniques of the graph are presented in [HEH+98].

The graph may be used for gaining understanding of database usage. Let us assume that certain attribute of a class contains some information that is stored in database. In order to understand how that information is used, we can read variable dependency graph for the attribute and find out the other variables that are in relation to the variable.

A model of maintaining web database applications is described in [HLCW99]. The model decomposes a web database application into hyperlink diagrams (HLD), entity-relationship diagrams (ERD) and dataflow diagrams (DFD). DFD represents how data is manipulated in the application. It comprises process, data store, external entity and the dataflow between these. The process means here a single unit of work, in web application it

could be a single web page. A data flow consists of data items that are described in ERD as attributes or represent a passing parameter between files. A data store is an entity of ERD.

DFD and ERD are used in maintenance tasks in order to identify the affected programs when database changes happen. This process is called database analysis. Database changes are deletion or addition of field, relationship or table. Also the change of the data type of an attribute is regarded as a database change. There are three types of database analyses to find out affected program source code. [HLCW99]

Attribute usage analysis searches usages of an attribute from the source code. For each process in DFD, if its input or output data flows contain the attribute, this process is identified as an affected program.

Entity usage analysis searches usages of a table from the source code. Attribute usage analysis is done for each attribute of the entity. The identified programs of entity usage analysis are the union of the programs identified by attribute usage analysis.

Relationship usage analysis searches usages of a relationship between two tables. The relationship usage analysis is based on attribute usage analysis. It is done on both attributes of a relationship. The identified programs are the intersection of the programs identified by the attribute usage analysis.

Data-flow diagrams appear to be useful in software maintenance tasks and also in regression testing of database applications [HMD01].

5.7 Analyzing embedded SQL

Analyzing embedded SQL focuses on finding ESQL (embedded SQL) statements from the program source code and extracting the information from them. The main purpose is to describe the artifacts and the dependencies between the database and the program. This section describes two approaches to analysis of embedded SQL.

Analyzing ESQL from COBOL program is described in [HEH+98]. It is a good example how code reverse engineering techniques are used to support database reverse engineering. Analysis uses an extension of program slicing technique to analyze programs with ESQL.

As a difficulty of the analysis, [HEH+98] describes that the physical schema is not explicitily declared in the program. Physical schema can be obtained by reading DDL (Data Definition Language) statements if they are available.

Suominen explains how the HyperSoft system is extended to cover the analysis of ESQL [Suo97]. The HyperSoft system is a reverse engineering tool that represents dependencies of software artifacts via hypertext. The system produces *access structures* like declaration, occurrence lists, forward call graphs, backward call graphs, forward slices and

backward slices [Suo97].

Analysis of embedded SQL is based on ESQL definition blocks that are separated from host language by `EXEC SQL` prefix [Suo97]. There are four kinds of symbols in embedded SQL statements: variables, tables, columns and cursors. The HyperSoft system searches all the occurrences of these symbols from ESQL statements.

The HyperSoft system produces occurrence lists of ESQL symbols. This is a useful feature when inspecting, *e.g.* how cursor is handled. Occurrence list for a table shows all places of the source code where the name of the table is used. This is useful in situations where the table contains erroneous or wrong data.

The HyperSoft system can perform slicing between SQL and C source code. For example, a backward slice from a variable used in ESQL statement gives all statements of the program that influence to the value of the variable. Also, a backward slice from programming language variable in C code gives slice that can contain SQL code if the variable is influenced by `SQL SELECT` statement.

Forward slice contains a subset of the program where the value of given variable influences. It can also be used between SQL and C source code. SQL UPDATE and INSERT statements belong to the forward slice if there is a relevant variable in the statement that produces data flow to the database. The system does not aim at representing the data flow between SQL statements because it could be misleading [Suo97].

Analysis of embedded SQL differs from analysis of SQL/CLI usage in many ways. There are no `EXEC SQL` prefix in the source code of SQL/CLI application. SQL is located in the values of `String`-variables. It is more difficult to find where SQL statements exist in the source code. There are no cursors or variables in the SQL statements of the SQL/CLI application unlike in embedded SQL statements.

SQL/CLI application uses SQL statements such that these kind of special slicing mechanisms as described earlier are not needed. In SQL/CLI application there may be complex SQL statement that has been constructed in source code from many different variables and in many different places. We can use backward slicing for the `String`-variable holding the complex SQL statement to find out what values the statement is constructed from.

5.8 Object-relational mapping (ORM)

There is a correspondence between the classes and the tables of the database. Attributes of the classes correspond to the fields of the tables. A row of the table may correspond to an object in the application. The mapping techniques are described in [Amb03].

A table in relational database schema may correspond to a class of OO system and in some cases, the table is handled only through the class. Understanding connections between

these areas (relational schema and application) is important in order to understand the whole application. However, in some cases the correspondence does not exist. If the classes do not represent the tables in the database directly, the application needs to be analyzed more thoroughly.

There are many frameworks for object-oriented languages (*e.g. Java Data Objects* for Java) to make the persistence of the objects "transparent". They are called object relational mapping (ORM) frameworks. These frameworks map the objects of the application to the relational database such that usage of the database (or *usage of the persistent objects*) becomes simpler for the developers. The program becomes also more independent from the database.

5.9 Object-relational impedance mismatch

Relational database and OOP is not a simple combination. These two architecture models are very different from each other. The problem of the fit between two technologies is called *object-relational impedance mismatch*. Book [Amb03, chapter 7] describes the impedance mismatch. The industry has been actively providing *object-relational model* as a solution for the problem. The majority of relational database vendors have extended their RDBMS to support some features of object-oriented model.

The problem is deep. The relational paradigm behind the relational databases and the object-oriented paradigm have many differences. Objects have both functionality and data encapsulated as a single logical unit. Relational technologies support the storage of data in tables and the manipulation of the data is performed externally via SQL or within the database via stored procedures. While relational databases aim at separating data and behavior, object-oriented programming tries to encapsulate them. Object-oriented approach considers both data and behavior when modeling the data while relational approach considers only data. This difference results in differently structured models of the problem domain. [Amb03]

Problem occurs in case of *relationships* between objects and *reference keys* between tables. A many-to-many relationship between two classes does not require third class to manage the relationship. For example, there is a many-to-many relationship between *customer* and *address*. However, there are three tables in the relational model: `Customer`, `Address` and `CustomerAddress`. [Amb03]

According to Ambler [Amb03], it is common practice not to show keys on class diagrams. However, in an application using relational database, keys are needed in every class of which objects are written to the database. Ambler calls the keys *shadow information*. He defines shadow information as "any data that objects need to maintain, above and beyond

their normal domain data, to persist themselves".

SQL is *nested* into the programming language. It is written to the values of the programming language. This affects to maintenance of the application in two ways. Firstly, if a database administrator wants to change a database query, he must understand the language of the source code in order to locate the query. To prevent this problem, some systems are designed such that there are only few specific modules that construct SQL statements and other modules use them in order to access database. Secondly, a programmer must understand SQL up to some degree if he must implement features that access the database. A programmer must understand the various symbols of the SQL statements.

In order to cope with the impedance mismatch, the object relational (OR) mapping frameworks are useful, (*e.g. Java Data Objects* for Java).

5.10 Data dictionaries help understanding database usage

Data dictionaries are used in some organizations in order to store and manage metadata about the databases, the applications and the authorizations that are used in the organization. A useful dictionary system should manage

1. descriptions of the schemas
2. detailed information on physical database design
3. description of the database users
4. high-level descriptions of the database transactions and applications and of the relationships of users to transactions
5. the relationships between database transactions and the data items (tables) referenced by them
6. usage statistics: frequencies of queries, etc.

For a maintainer of a database application, these things are very important, especially the fourth and the fifth points. By [EN99], the fourth is useful in determining which transactions are affected when the database schema is altered. In other words, it may be critical to understand the dependencies between certain application parts and database tables in order to make modifications to the database. However, Elmasri and Navathe [EN99] does not give any hint how to build the data dictionary and how to get the important information. When manually collected, reading code and database definitions, the process may be too time-exhausting. In big systems, the continuous updating of the data may become too expensive. It is clear that automated solutions become necessary when the size of the database and the application increases.

5.11 Artifacts of database usage

Artifacts related to database usage can be groupped into five categories: ER model, OOP, database API, query language and relational database schema (physical schema). Table 5.2 lists some of the artifacts of the categories and corresponding examples. Due to the large amount of artifacts of several categories, a single representation for describing database usage (showing necessary artifacts) may not be reasonable.

Table 5.2: Artifacts related to database usage.

Category	Artifact	Example
ER model	entity	Person
	relationship	Job
OOP	class	Person
	object	*e.g.* Patrick Norton
	attribute	last name
	method	changeID()
	invocation	invocation of method *changeID*
Database API	database connection	`Connection` object
	results of the query	`ResultSet` object
Query language (SQL)	statement	`INSERT` statement
	query	`SELECT` statement
Relational database	relation (table)	person
	tuple (row)	person Norton
	attribute (column)	id
	join	person <–> department

5.12 Dependencies between the artifacts

Wilde and Huitt [WH92] define object-oriented system dependencies. A dependency is a directed relationship between two things. If A is dependent on B, changing B has an impact on A. Based on the model presented in [WH92], the dependencies of database usage could be as described in table 5.3. The term *statement* refers to SQL statement in the table.

There are different distinctions between the categories. This approach to divide dependencies to the categories requires that all the SQL statements are identified from the source code. Also, having information about the database schema (*e.g.* reading data definition state-

Table 5.3: Dependencies of database usage.

Category	Dependency
Column To Table	C belongs to table T.
SQL Statement (S) To Table	S reads (`SELECT`) data from table T. S alters (`UPDATE`) data in table T. S adds (`INSERT`) data to table T. S deletes (`DELETE`) data from table T.
Method To SQL Statement	M constructs a statement S. M executes a statement S. M handles results of a `SELECT`-statement.
Method To Table	M constructs a statement that has an impact on table T. M executes a statement that has an impact on table T. M handles results of a query related to table T.

ments) will be useful in finding out the dependencies.

The first dependency, *Column To Table* can be easily found from the database schema. Database reverse engineering may also help here if the database schema information is not available. The dependency is not very useful as such but may be helpful when combined with the other dependencies.

SQL Statement (S) To Table dependencies are trivial in static SQL but may be more difficult to find when analyzing dynamic SQL. Finding the dependencies requires that source code is extracted into a form where each statement is located and identified from the source code. The identification is ambiguous. There are at least three ways to identify SQL statements in the source code.

Structure identification separates all SQL statements that differ from each other by their structure. For example, queries `SELECT * FROM person WHERE id = 2` and `SELECT * FROM person WHERE id = 3` are identified as the same query because of their identical structure.

The other way to identify SQL statements is the location identification which is based on the location of the execution place. The execution place is a line of source code, where database interface is called in order to send the SQL statement to be executed by database server. This kind of line is called later *hotspot*. If the query A is executed in place P, then it is not the same query as the query executed in place P2. In other words, if two queries share the same hotspot, then the queries are also the same. The result of this identification depends on the interpretation of database interface. Database interface is normally same as JDBC API.

However, in some cases, there may be self-made classes which create higher abstractions on top of the JDBC. Those classes should be also regarded as a part of the database interface, in order to keep this identification reasonable.

Third, value identification separates the queries that are lexically different. This identification does not serve the purpose in static SQL or dynamic SQL. In practical database applications, the queries always change in some degree.

The *Method To Table* dependencies are indirect because they are derived from *Method To SQL Statement* and *SQL Statement To Table* dependencies. They are however represented here as a separate category because it might be comfortable for a maintainer to analyze database usage at *Method To Table* level.

The dependency *Method constructs a statement* means that M is changing the value of the `String` variable that holds the statement S. For example, M gets the variable as a parameter and adds something to it and then returns the updated value. This dependency is clearly a dataflow dependency which means that in M there exists some dataflow towards the variable holding the statement S. A single SQL statement may be stored in several `String` variables in the application. In that case, all the variables would be regarded as they hold the statement S. There are plenty of these dependencies in systems that generate queries from parts dynamically.

The dependency *Method executes a statement* means that M is the method that sends the `String` variable (holding the statement) for database interface class that will send it forward for the database server. Database server then finally validates and executes it. SQL/CLI interfaces have one or two functions that send statements for the server. This type of a function is called *hotspot* in [GSD04]. These dependencies are extracted from the source code in two phases: first finding the methods that call the hotspots and then formulating the values of the `String` variable for separately to find out the SQL statements involved. In a JDBC application, every method that invokes `executeQuery` or `executeUpdate` of JDBC interface [Ree97], has such *execute* dependency. There are, in some cases, self-made database utility classes that provide methods for sending the SQL statements for JDBC interface. In those cases, it should be important to classify the use of such database utility classes as they were a part of the JDBC interface. All uses of such interfaces are classified with the execute dependency.

The dependency *Method handles results of a query* means that the method handles the data of database and usually transforms it to objects that represent the data. The results of a query are represented in a table. The method having this dependency does the mapping between relational database data types and OO language data types. This dependency can be analyzed from the source code in a JDBC application by finding the uses of `ResultSet` object (that holds the results). In order to find out the dependency, the ResultSet object must

be connected to the query. The information about *method executes statement* dependencies is useful in connecting the query to the `ResultSet` object. In some cases, it is impossible to see the connection statically. For example, there may be several queries that are all handled by some generic method and dynamic binding is used such that it cannot be found out statically.

5.13 Analyzing database usage

This section discusses briefly analysis methods of database usage. Two approaches are presented: static analysis (subsection 5.13.1) and dynamic analysis (subsection 5.13.2). This area of research is new and there are few good publications available. This discussion is based largely on code reverse engineering.

5.13.1 Static analysis

Static analysis does not require running of the system. It requires reading source code in order to find SQL statements. Static analysis of database usage should also involve the analysis of SQL statements. It is often database API specific, RDBMS and source code language specific. It is a complex area because of various technologies in databases, database interfaces and database query languages.

Use of code reverse engineering tools may help. Visualizing source code and understanding the semantics of the methods and classes is important. The majority of reverse engineering tools do not analyze database usage in any special way. The tools show the method calls to the API. Some of the tools like Java Analyzer by Cast Software and Total Access Analyzer by Fmsinc.com provide a visualization of the dependencies between database and source code entities.

An analysis technique represented in [GSD04] is useful in analyzing database usage. Dynamically generated SQL queries are checked statically. The technique makes use of a string analysis technique that is presented in [CMS03]. Using the technique represented in [GSD04] it is possible to determine statically if the SQL queries are type-safe. The technique applies the static string analysis to Java programs and uses Java bytecode and database schema as input.

The simplest way to find information in a program is using pattern (string) matching tools [HEH+98]. They can be used to find SQL statements from the source code. Pattern definition language for pattern matching is represented in [HEH+98]. Pattern matching suits well for static SQL usage, but is adequate for dynamic SQL in some cases only. Pattern matching is not useful when the application is object-oriented and involves dynamic binding. If the SQL

statements are built dynamically in several places of the program, it can be difficult to locate all uses of certain table in the program code by pattern matching.

5.13.2 Dynamic analysis

Dynamic analysis of database usage collects SQL statements runtime. Every time an SQL statement is performed, the information where it was performed (class and method) and what was the statement is saved. The structure of the statement defines what tables and columns are used.

Dynamic analysis can be done by building test cases that cover sufficient part of the application. This approach is well known from OO source code analysis [WH92]. The drawbacks of dynamic analysis are that it may be costly and time consuming.

Database logs and special logging mechanisms can be used for collecting information about the database usage. A special log collects information about which methods of which classes execute database operations.

5.14 A way of visualizing database usage

Visualizations of database usage are presented in [Hen03] and [AEP96]. Henrard [Hen03] defines *database usage graph* as a graph with vertices of two types (the programs and the collections/entity types) and the edges link the programs with the collections/entity types they use. The edges are labeled according to their usage (input, output or update).

The approach of Henrard is applicable for COBOL programs with network database model. The word *program* means here a block of statements (a method in OO language). The collections/entity would be a database table in relational model.

Henrard [Hen03] presents a combination of procedure call graph and *data usage graph* for representing general architecture of a COBOL program. Data usage graph means a graph with read and write dependencies between procedures and data entities. The distinction between database usage graph and data usage graph remains unclear.

When applying the ideas of Henrard to an OO system using a relational database with dynamic SQL statements, the result might be as follows. There would be a table usage graph that describes which methods (and classes) use which tables. There would also be method invocation graphs. In OO systems there should also be inheritance graphs. Dynamic binding clearly reduces the power of the call graphs of OO system but they are useful where dynamic binding is not used. Combined graph of database usage might have method invocation, SQL statements and database tables.

Such graph is presented in figure 5.5. The figure shows the database usage of exam-

ple code of figure 2.2. The entities and relationships of the figure are described in section 5.12. The figure describes that the method `operation1` constructs and executes a query (`SELECT` statement) that reads from table `person`. The method also handles the results of the query. The figure describes also that the columns `personid` and `lastname` are involved in the query.

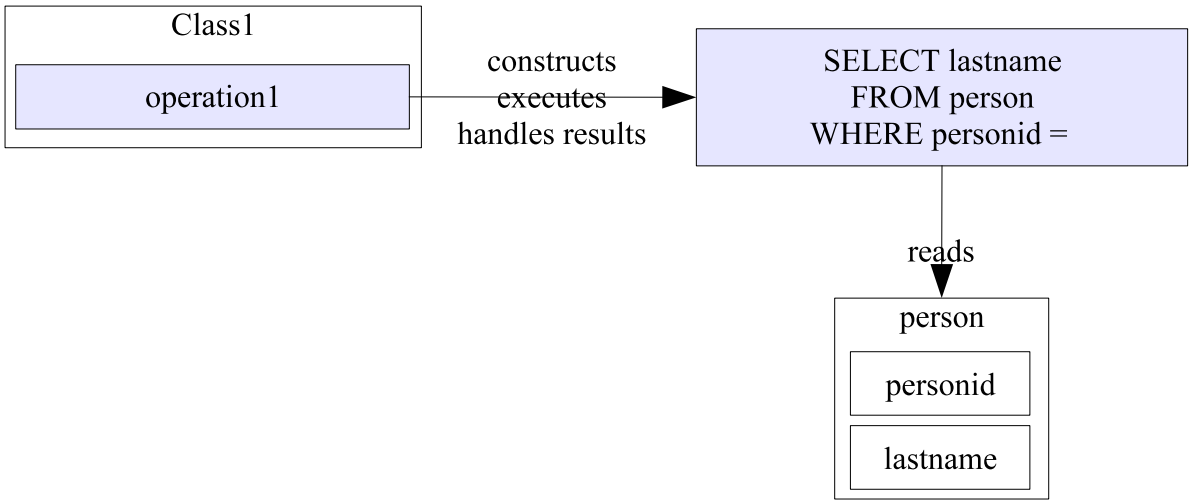

Figure 5.5: Diagram of database usage based on code in figure 2.2.

When applied to a more complex database usage scenario, the result might be like in figure 5.6. This example involves several methods that are in different roles: constructing, executing and handling the results. The arcs between methods are call relationships and the arcs between methods and the query are the same relationships described in section 5.12. The graph shows methods `method1` and `method5` are the methods that construct the query. The query is then sent for `method2` and onwards to `method3` that executes it. The `method3` then invokes `method4` in order to send the results of the query to be handled. The graph displays also that the query reads data from two tables, `table1` and `table2`.

This example is only a single database operation. A scenario with several operations involving transactions and user rights management would be interesting but too large area to be discussed here.

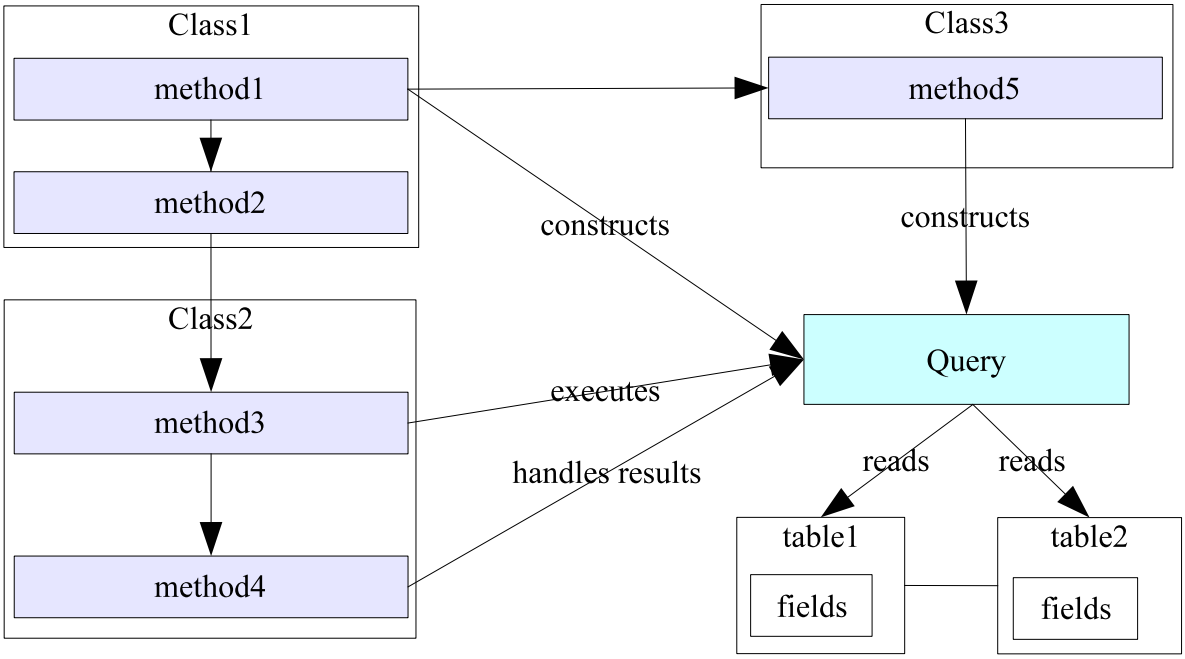

Figure 5.6: Diagram of a complex database usage scenario.

6 Software maintenance and database applications

Software maintenance is the totality of activities to provide cost-effective support to software [ABDM01]. There are several types of software maintenance. Software maintenance can be described as a process involving certain phases. The critical factor in software maintenance and also in software evolution is program comprehension [vMV95b]. Von Mayrhauser and Vans [vMV95b] describe how maintainer develops cognitive models with various ways during program comprehension.

Section 6.1 describes the types of software maintenance. Maintaining database applications is discussed in section 6.2. Maintainability of database applications is described in section 6.3 and simple metrics are presented of evaluating the maintainability of SQL statements. Section 6.4 discusses about database applications at a very general level. Section 6.5 describes different problem situations when maintaining database applications. Problematic issues of software maintenance in general are described in section 6.6. Section 6.7 represents the problems which were studied in the conducted empirical study.

6.1 Types of software maintenance

Software maintenance can be classified [Swa76] in perfective, corrective and adaptive maintenance. Sommerville [Som01] presents a software maintenance type called preventive maintenance. There are also other ways [CHK+01] to classify software maintenance tasks the classification which divides the maintenance tasks to perfective, corrective, adaptive and preventive maintenance is suitable for the purpose of this paper.

Perfective maintenance adds functionality to the system or enhances its quality attributes. Corrective maintenance aims at eliminating the failures of the software. Swanson [Swa76] classifies the failures in processing failures, performance failures and implementation failures. Adaptive maintenance adapts the software to fit in a new data environment or new processing environment. The goal of preventive maintenance is to make the software more maintainable or and to ensure the level of maintainability. It consists of re-engineering, restructuring and re-documenting.

Corrective maintenance process involves the steps: [JSKCG94]

1. Analyze the bug report to understand the nature of malfunction.
2. Develop an understanding of the software.

3. Based on information gathered in steps 1 and 2 establish association between the bug and the code.
4. Design changes and modify the software to correct the bug.
5. Test to make sure the bug is fixed.
6. Test to make sure all other functionalities are working properly.

The process of software maintenance may involve following phases: locating relevant piece of code, planning the change, implementing the change, documenting, debugging and testing. The tasks are strongly related with code comprehension. For example, study [vMV95a] shows that in **adaptive maintenance**, there are two-way task connections from program comprehension task to

- defining adaptation requirements,
- developing preliminary and detailed adaptation design,
- implementing changes,
- debugging and
- regression testing.

6.2 Maintaining database applications

Maintenance of an application that uses a relational database differs from maintenance in general. Generally, program comprehension (or *system comprehension*) becomes a larger area covering database and SQL concepts. Thus, the more complex implementation domain brings many new challenges for maintainers.

Database comprehension is an important issue in understanding the whole application and it is in a strong relationship between program comprehension. Program comprehension techniques are essential for database reverse engineering, and conversely. [HEH+98]

Database application may require changes in database schema or in program source code. The most important point is, that there is so strong relationship between the application and the database that many changes involve changes both sides and must be done accordingly. Also, understanding is in a primary role. Database comprehension is a prerequisite for DB schema changes and adding functionality which accesses existing tables.

One good example is a situation where a new column is added to a table. The most difficult job is changing all the SQL statements in the program source code handling the altered table. Making the change may be trivial but locating all the statements may take the most of the time. Testing of the altered code may also be time-consuming if the changed statements are spread over the application.

Many of the software maintenance tasks could be made easier with suitable program comprehension or reverse engineering tools. Investing in program comprehension technology is critical for the software industry [MJS+00].

6.3 Maintainability of database applications

Metrics for measuring the maintainability of database applications are presented in [PM00]. However, the restriction of the metrics is that they can be applied only for **SQL programs** (programs written only with SQL). They should be called more precisely *SQL metrics*. They are as follows [PM00]:.

NT is the number of tables referred in the statement

NN is the number of nesting `SELECT` clauses in the statement

G tells whether there exists (1==exists, 0==not) `GROUP BY` clause in the statement

Different measures are required for 4GL (*e.g.* Java) database applications [PM00]. Clearly, the maintainability of database application depends strongly on the complexity of the application. A database application has three major sources of complexity: application source code, SQL code and database schema.

Application source code complexity is a well researched area which we will not discuss here. SQL code complexity metrics are described in the next chapter. Database schema complexity [CPG01] is another area with metrics. There exists three simple metrics, too:

NT is the number of tables in the schema,

NA is the number of attributes of all the tables in the schema and

NFK is the number of foreign keys in the schema.

6.4 Functionality and data

The database usage becomes problematic when the software is large and there are many dependencies between the database and the application. Especially undirected, hidden dependencies hinder understanding the IS as a whole. Here we present a general level model of database usage which has four components: functionality, persistent information, database and source code. Functionality and persistent information are high-level entities of the system and they are implemented by source code and database.

Figure 6.1 describes the model components. Program functionality is implemented in source code. Source code of the program contains references to database entities, which store persistent information of the IS. When studying the database usage, there exists four

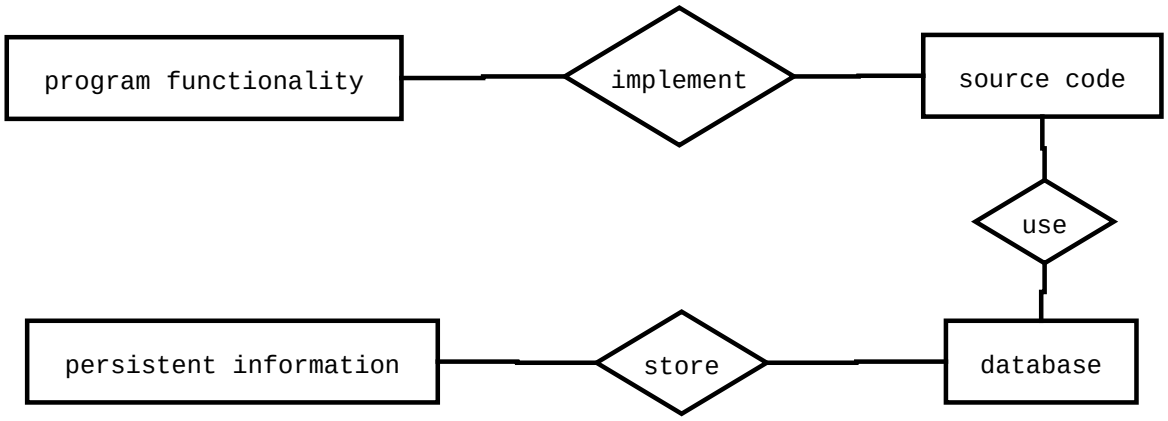

Figure 6.1: Model of IS related to database usage.

main questions related to the model.

Where is certain functionality located in the source code of the application and in the database as stored procedures or triggers? This is a question that is familiar in reverse engineering and program comprehension. But with database applications, the problem is wider because functionality may also reside in the database. The reverse version of this question is which functionality is related to this source code place, stored procedure or trigger?

Where is this information stored in the database? This may be expressed at a lower level in a form: Which table (and which column) stores this kind of information? The reversed version of the question is what information is stored in this table?

Which functionality uses or manipulates this information? In order to answer this question, we must first find out which tables are used to store that information. Then follows finding out which parts of the applications are using the tables. Finally, we must understand the places of source code in order to recognize the functionalities.

Which information (that is stored in database) is used by this functionality? First we must find out where the functionality is located and which tables are used by that source code. Finally, the answer is clear when an answer of the question **what information is stored in those tables** has been received.

As a conclusion of the above, database applications may not be easily understandable because of strong separation between functionality and information. As a contrast, object-oriented architecture tries to encapsulate the both in a single unit, *object*. Therefore, a combination of object-oriented architecture and relational databases is clearly unlogical.

6.5 Maintenance problem situations with database applications

This section discusses about problem situations with database application maintenance. Subsections 6.5.1, 6.5.2, 6.5.3 and 6.5.4 describe some problem situations in corrective, perfective, adaptive and preventive software maintenance related to database usage. Problems of regression testing database applications are discussed in subsection 6.5.5.

6.5.1 Problem situations in corrective maintenance

Corrective software maintenance processes bring some tasks and problems: locating relevant place, changing source code or DB schema and testing. In these tasks, comprehensive understanding of the system is essential.

Corrective maintenance includes work of correcting database access errors. These are typical error situations:

- A user or a developer detects erroneous data (of database).
- The system prints error message from RDBMS during execution: broken SQL statement.

Erroneous data may be created by faulty SQL code, erroneous program code (*e.g.* Java), erroneus DB functionality or user malfunction. If the situation does not give any good hints about the real reason of the error situation, the bug hunting process may be time-consuming.

There are several types of errors related to SQL code. Some of those produce a run-time error message. A faulty SQL statement may also produce erroneous data without any error message. The reasons for the falty SQL code are incorrect syntax, data type mismatch, DB schema mismatch and semantical error. They are described in table 6.1.

Table 6.1: Reasons and occurrences for faulty SQL code

Reason	Occurrence
Incorrect syntax	Run-time exception from RDBMS
Data type mismatch	Run-time exception from RDBMS
DB schema mismatch	Run-time exception from RDBMS or wrong data
Semantically incorrect	Odd behavior or missing/erroneous data

Broken SQL statement, incorrect syntax is a concrete problem in corrective maintenance. One typical instance of this problem is a missing ending parenthesis problem. This problem leads often to a run-time error message by DBMS and is therefore fairly easy to detect. The source of the problem may be complexity of constructing SQL dynamically from pieces.

Broken SQL statement, data type mismatch occurs when the data type correspondence between the DBMS and Java is not matching. This problem leads often to a run-time error message by DBMS. The source of the problem may be misunderstanding of the data types.

Broken SQL statement, DB schema mismatch is a problem where DB schema is not matching to the schema written in the SQL statement. For example, SQL query tries to read data from table `BMColl` but should be using table `Coll` instead. This problem leads often to a run-time error message by DBMS.

Broken SQL statement, semantically incorrect is a problem which is more difficult to locate than the above problems related to broken SQL. The reason is that this problem may not be detected by DBMS. Unexpected functionality or errors in processed data may occur during run-time. The source of the problem may be either complexity of SQL statement or database schema.

Corrective maintenance also tracks down bugs in software. One such database usage related bug could be *wrong data in table* which means that data in table is erroneous, missing or duplicate. A solution [GSD04] for data type mismatch checking of SQL queries has been presented. The solution is based on a string expression analysis technique presented in [CMS03].

6.5.2 Problem situations in perfective maintenance

Perfective maintenance situations may require adding functionality related to DB or altering DB schema. Here is an example of a maintenance process related to perfective maintenance. The task is to add a new column to a table named T1. Firstly, we must do some trivial things, *i.e.* determine the data type of the colum and modify the database schema. Then we must find out which parts of the application have to be modificated. Modifications must be done for two reasons: adding the functionality related to handling the new column and keeping the application working correctly.

The most important way of locating the parts to be modified is by searching the places where T1 is used in the application. This may be trivial with text-search tools like *grep* if there are no application concepts named to T1. With bigger applications and databases, text-search tools may become useless.

Planning modifications may not be easy if there are many places to change. Modifications may be targeted to Java-code handling the database queries or to SQL-statements (or parts of them) nested in Java. In normal case, both of these should be changed. Implementing modifications should be trivial.

Testing may point out to be challenging and time-consuming. Especially regression test-

ing of database applications involves many difficulties [HMD01]. In that case, there should be methods for determining which parts of the application have to be tested.

6.5.3 Problem situations in adaptive maintenance

Adaptive maintenance tasks may include altering the database schema and changing the application source code. In order to adapt the system to a new environment, it is essential to understand the system at the general level [vMV95b].

Adapting the system by changing the database schema requires locating relevant places of the application source code where to make fixes accordingly. *E.g.* a data type change of a column needs checking each SQL statement of the application source code where the column is referenced.

Changing the functionality of the system may require changes to either functionality of the application or the functionality that is stored in the database. Functionality changes require regression testing which is an important activity in adaptive maintenance [vMV95b].

Debugging is also a relevant activity in adaptive maintenance. This was already discussed in section 6.5.1

6.5.4 Problem situations in preventive maintenance

Preventive maintenance may also require changes in either functionality or database schema. For example, changing the currency of stored money values, may be achieved by a simple SQL UPDATE statement. In that case, the application source code has to be also fixed to show the relevat monetary unit in the user interface. In order to find out every place where the unit is present in the source code, following data flow dependencies is useful.

Preventive maintenance is quite near to perfective maintenance, so the cases discussed in the section 6.5.2 apply quite well here.

6.5.5 Problems in regression testing

Regression testing is important in almost all kinds of maintenance tasks [vMV95b]. Regression testing of database usage presents new problems [HMD01]: find out affected source code modules in DB schema change and affected tables in application code change. These dependencies are needed for defining what program components should be tested after the change, because testing all the components after every change is practically impossible.

6.6 General level maintenance problems

Lienz and Swanson [LS81] studied over four hundred software maintenance projects in order to find out the most common problems in software maintenance. The most frequent problem factor is user knowledge. Inadequate user knowledge is such a frequent problem in maintenance due to lack of user understanding and inadequate training. Also, user demands for new functionality and enhancements is reported often as a problem in maintenance projects. This is understandable because user demands are not controllable by maintenance personnel. Other frequent problems were *e.g.* competing demands for programmer time and documentation quality. However, the study explores the manager level personnel and is not applicable for this study.

SWEBOK [ABDM01] also stresses on the importance of the understanding in software maintenance. It lists four technical issues in software maintenance: limited understanding of software, testing, impact analysis and maintainability.

6.7 Problem classification

There are several problems of several types related to dabatase usage and software maintenance. This thesis focuses on programming level problems, though there would also be different problems, *e.g.* related to project management or the maintenance process. The problems which were studied in the empirical part of this thesis are classified by the concept areas. This section may mix the terms **task** and **problem**. Another expression for them might be **problematic task** but it is not known here if the tasks are problematic at all. The empirical study (chapter 7) was conducted for finding that out.

Subsection 6.7.1 presents the classification. Then, the categories are presented in the next four subsections. Subsection 6.7.2 describes the tasks related to the program category. Subsection 6.7.3 presents the tasks of the database category. Subsection 6.7.4 describes the category which consists of the tasks related to the relationship between the application and the database. The fourth category, SQL is described in subsection 6.7.5. Finally, subsection 6.7.6 discusses briefly about some relations between the problems.

6.7.1 Background of the classification

The problems can be classified with several ways. Some of the problems relate to comprehending application, database or relationship between database and application. There are some more general level problems that are more independent of the implementation than the others. The problem classification used is based on the concept areas, and the used concept

area defines the category for a task. A concept area is a group of concepts that are required in executing a maintenance task. There are four major concept areas in understanding database usage: program, database, application–database relationship and SQL.

The literature did not offer any classification so there was a need to create one myself. This classification was formed as follows. Several tasks were collected from the experiences with Korppi system. Because some of the tasks were database-specific and some application-specific, it felt natural to create corresponding categories. Since SQL was used for accessing database from the application and SQL related tasks were not fully program specific, a separate SQL category for SQL related tasks was created. Some of the tasks (*e.g. Finding out which DB entities are used by a specific part of the application*) did not belong to these categories because they consisted of both program and database concepts. A new category, database–application relationship for them was created. The tasks fit in the categories quite well though there surely are many other ways to classify these tasks.

6.7.2 Category *Program*

Program comprehension is a critical factor for software maintenance [vMV95b]. This category consists of tasks which belong to program comprehension activities. The category contains only the tasks that are related to program artifacts. Tasks related to database artifacts are handled later in their own category. Here are some of the tasks that belong to this category:

Understanding the program at general level is a task that requires understanding of the main components of the program. It is also relevant to know about the problem domain and what the program is used for. The task is an important part of program comprehension and is typical for perfective, corrective and adaptive software maintenance [vMV95b]. Comprehending object-oriented systems at the general level is more difficult than comprehending conventional systems because call hierarchy is not applicable for this purpose [WH92].

Understanding the meaning of a part of the program requires specific understanding about a software artifact. The artifact may be *e.g.* a class, method or a block of code. Essential is to understand what is the meaning of the artifact. The task belogs to detailed code understanding. [WH92]

Locating relevant place of code is a very frequent task in many kinds of maintenance activities. It may involve checking references of some variable, following call hierarchy, finding side-effects or finding a bug, depending on the type of the activity. One form of it is also locating specific functionality in source code which is important when maximizing the benefits of reuse. This problem may be more serious in OO systems than in

Table 6.2: Maintenance problems of database usage: Classification

Category	Task
Program	*Understanding the program at general level*
	Understanding the meaning of a part of the program
	Locating relevant place of code
	Finding out call relationships
	Understanding dynamic binding
	Determining the effects of changes
	Locating a bug
	Fixing a bug
	Dealing with the complexity of functionality
	Other program related problems
Database	*Understanding the database at general level*
	Understanding the meaning of a database entity
	Understanding the relationships between tables
	Finding out where particular data is stored
	Determining the data type of a column
	Changing the database schema
	Understanding the functionality stored in the database
	Dealing with the complexity of data
	Other database related problems
Database-application relationship	*Understanding data flow relations between the database and the application*
	Understanding data flow relations between the application modules through DB usage
	Finding out where a DB entity is used
	Finding out which DB entities are used by a specific part of the application
	Finding out the mappings between the classes and tables
	Understanding the database interface
	Understanding the transactions
	Other problems related to the relationship between the application and the database
SQL	*Understanding the meaning and the impacts of an SQL statement*
	Understanding how the SQL statement is constructed
	Understanding and managing the bindings between Java code and SQL code
	Determining how the data types should be matched between SQL and Java
	Slowness of an SQL query
	Dealing with the complexity in SQL statement
	Other SQL related problems

conventional systems because the functionality is spread over various object classes. [WH92]

Finding out call relationships is usual in program comprehension [WH92]. Following calls is more difficult when dynamic binding is in effect. It is important to identify chains of call dependencies. [WH92]

Understanding dynamic binding is a problem typical for OO systems since it complicates call relationships. Since static analysis is not able to identify all the dependencies caused by dynamic binding, other approaches are needed. [WH92]

Determining the effects of changes is a general task with any kind of software maintenance and development. It requires detecting modifications and locating the effects of changes. The task becomes more difficult when the complexity of the system increases because dependencies between the software artifacts expand. Making changes to source code requires checking all the relevant dependencies which may produce unexpected side-effects. This task is important in regression testing. [HMD01]

Locating a bug is a task of reading code or running debugger. It involves making hypotheses about the program behavior and revising them. Knowledge of the problem domain is important in corrective maintenance. [vMV97].

Fixing a bug is an action which requires good understanding of the bug. In addition, making changes to source code may cause side-effects. This task involves activity of detailed algorithms and code specifics. [vMV97]

Dealing with the complexity of functionality becomes more important when the size of the software increases. Refactoring is needed in order to manage the complexity. Some metrics, *cyclomatic complexity* can be used to detect the most complex parts of the application. [BR00]

6.7.3 Category *Database*

Database comprehension tasks relate only to database and not to the application or SQL statements. *A database entity* (or *an artifact*) is either table, view or column.

Understanding the database at general level is a common task *e.g.* when the information system is extended with new functionality and existing database structure must be utilized. Understanding becomes difficult with a database consisting of hundreds of tables [AEP96].

Understanding the meaning of a database entity is a task usual in database maintenance. This problem is also represented with a question: What information is stored in this table or column? The problem can be solved *e.g.* by locating the references of the table

(or view/column) in the application source code and finding out what information is processed there. Database documentation or data dictionary is also useful [EN99].

Understanding the relationships between tables is a task related to the previous task. The tables are related to each other with keys. This task is difficult with big databases [AEP96].

Finding out where particular data is stored a task which requires understanding tables and the database at the general level. For example, a maintainer might need to know where the addresses of students are stored. Answer might be a table named STADDR. [AEP96]

Determining the data type of a column is a task which becomes problematic only in the situation where the information about database schema is not available readily enough. The information may be fetched from multiple sources, e.g. database document, program source code or database administration application.

Changing the database schema is *e.g.* one of the following tasks: adding a column to a table, changing the data type of a column or removing a column. Dealing with the effects of the change is essential. The change may affect to the application or to other database artifacts. [AEP96]

Understanding the functionality stored in the database becomes more important when the significant part of the application functionality is stored in database. Functionality may be stored in triggers and stored procedures. They create data flow relations between tables which increases the complexity of the system. [EN99]

Dealing with the complexity of data is a proble typical for large databases. The dimensions of database complexity are *e.g.* set of relations, size and quantity of data in the database and number of associations among relations [AEP96]. The complexity can be measured by metrics [CPG01].

6.7.4 Category *Database-application relationship*

The database and the application may be totally tangled up in each other. As a result making little changes to the database schema may require huge modifications in the application source code. This duality of database and application provides us certain understanding activities to be managed:

Understanding data flow relations between the database and the application is a high-level understanding task. This kind of knowledge is important when the semantics of database entities is unknown but the maintainer is familiar with some part of the program (e.g. a user interface component) or conversely. Data flow comprehension is thereby useful in mining the missing semantics of either database or the application.

Testing is another main activity where relations must be managed [HMD01].

Understanding data flow relations between the application modules through DB usage complicates systems. When module A writes to table T and module B reads from it, there exist a data flow from A to B. These kind of dependencies make testing challenging and regression testing strategies are needed [HMD01].

Finding out where a DB entity is used is a task related to understanding the dependencies between application and database [AEP96]. This may occur in perfective or corrective maintenance. Typical situations are altering DB schema by adding a column to a table (perfective maintenance) and correcting DB usage error where erroneous data has been inserted to a table (corrective maintenance). It is essential to find out which source code modules may be affected when database changes. [AEP96]

Finding out which DB entities are used by a specific part of the application is a task that is a mirror-image of the previous task. For example, a maintainer might ask what tables are updated by `User` class. This is important task in testing, since changes in the source code may lead to erroneous data in any of those tables.

Finding out the mappings between the classes and tables is a difficult task because of the object–relational impedance mismatch [Amb03]. The differencies between the architectures make the mapping complex. Logical naming conventions might help understanding the mapping.

Understanding the database interface is a task of understanding JDBC or other kind of database interface. The database interface may be part of the application.

Understanding the transactions is a task of transaction-based systems. The target system uses transactions through `DB` class. Understanding how a transaction works is relevant, because a fault in a transaction may lead to erroneous data in the database.

6.7.5 Category *SQL*

There are two main reasons to problems related to SQL: complexity of SQL statement and complexity of how SQL is embedded within Java code. The former reason is explained in [PM00] through three simple metrics.

Understanding the meaning and the impacts of an SQL statement is a problem where the statement is too complex for comprehending fast enough. The **meaning** of a statement is what the statement should do, *e.g.* fetch all unnamed persons. The **impacts** is what the statement actually does, including any side-effects. The SQL code metrics NT, NN and G describe how understandable the SQL statement is. More higher the metrics are, the more difficult the statement is to understand. [PM00]

Understanding how the SQL statement is constructed from parts at run-time is a problem with dynamic SQL. This may create several problematical situations like difficulties in checking the validity of the SQL statement [GSD04]. Especially, if the statement is generated conditionally or under loop statements, the construction is more difficult to comprehend.

Understanding and managing the bindings between Java code and SQL code is a binding problem related to code and database schema changes [NHR99]. This is related strongly to the understanding of the meaning and the impacts of an SQL statement because if it is not known what a query fetches, it is impossible to code or change the Java statements that handle the results of the query.

Determining how the data types should be matched between SQL and Java is a problem that occurs because the mapping of data types is not very clear. For example, *REAL* (SQL) is mapped to *float* (Java) and *FLOAT* (SQL) is mapped to *double* (Java). [EHF01]

Slowness of an SQL query is a problem with both performance and complexity. SQL statements may contain difficult nested structures which use multiple tables under complex conditions. It weakens changeability of the statements and hinders the performance tuning. [EN99]

Dealing with the complexity in SQL statement is a task of SQL programming. SQL statements tend to be complex in large systems. Several statements in the target system consist of over one hundred lines of code. Complexity metrics are useful in detecting the threats in the application [PM00].

6.7.6 Problem relations

Problems are strictly related to each other. Figure 6.2 illustrates some of the relationships between the practical maintenance problems. The purpose of figure 6.2 is not to list all relationships between problems but to demonstrate how the problems could be connected together. The problems of the SQL category are not involved in the figure. The relationships are derived from the following definition. If problem B might occur when solving problem A, there is a directed relationship from A to B.

For example, the meaning of a table might be found out by searching the locations of table usage in the Java code and studying *e.g.* the SQL code using that table. Therefore there is a relationship in figure 6.2 from *Understanding the meaning of a database entity* to *Where an entity of DB is used is used*.

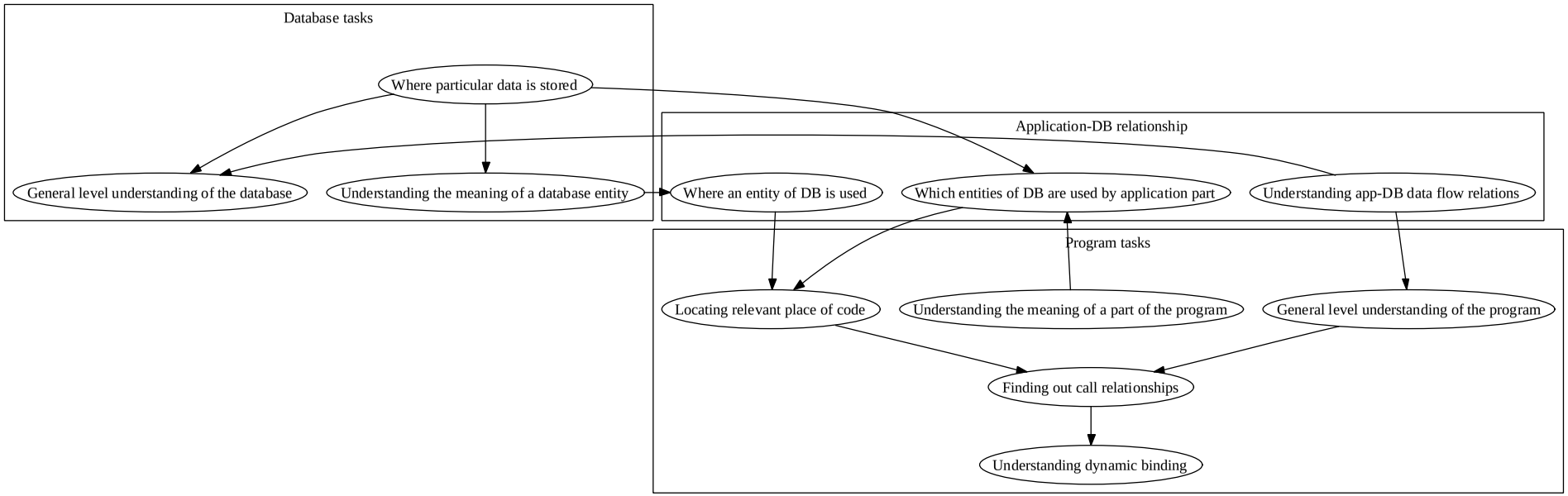

Figure 6.2: Some relationships between the presented problems.

7 Empirical study

The empirical part is described in this chapter. The target system maintainers were sent a questionnaire asking how problematic and important certain tasks were in practical programming or code understanding tasks related to database usage.

Section 7.1 describes the attributes of the target system. Hypotheses are discussed in section 7.2. The questionnaire is described in section 7.3. Limitations of the study are described in section 7.4, and the results of the questionnaire are described in section 7.5.

7.1 The target system

The empirical study was performed in order to recognize most significant problems related to database usage in target system called Korppi. The system has a long history, since its development has been started as early as in 1998 as a student project coordinated by the department of applied mathematics in the University of Jyväskylä.

The development of the current version was started in 2000. Since then, there have been nine four-month software projects and individual developers developing the system further [Kor05]. There has been more than 50 persons involved in maintaining or developing the system. Some of the persons have been developing the system as techical advisors or employed maintainers. If the project supervisors and representatives are also counted, the number is near to one hundred.

The involved persons are mainly students with little experience of software development. Many of the developers had not received training on Java when they began working. The mandatory main programming courses were in C++ yet in the spring of 2002. So, the skills and the experience must have not been the best possible.

The system consists of more than 255 000 lines of source code. The database of the system has over one hundred tables. The system resembles a legacy system because it is a demanding challenge for the maintainers. There are 28 394 (at 18.1.2004) registered users. There are at least 250 simultaneous users.

The developers used various tools like JBuilder and NetBeans. Database schema changes were executed with the WWW-based UI of postgres.

7.2 The hypotheses

Like stated in chapter 2, database applications are very different from other applications. This leads to major differencies in software development and maintenance of database applications. One of the biggest difference is increased complexity. The complexity hinders program comprehension and makes maintaining slower.

Due to practical experiences in maintaining the target system, some hypotheses occurred. One of the most important experiences was that the changing of code or database was both difficult due to complex dependencies between the Java classes and database artifacts. That raised many questions with architecture, persistence frameworks and object-relational mapping. Using some persistence frameworks or even doing the mapping by hand could have been very helpful. However, the situation was bad and the parts of the software could not be refactored easily to reach a better architecture.

The base hypothesis is that a database application like the target system has many significant problems in software maintenance. What are the problems and to what are they related? This research aims at finding out the most important problems and information needs related to database usage. This was achieved by asking the maintainers and developers of the target system the problemacy of tasks related to database usage.

Another hypothesis is that finding table usages is important for database applications [HMD01]. It is also interesting to see how significant is the role of understanding database in the maintenance of the target system.

7.3 The questionnaire

The questionnaire (see Appendix A) was an online form. The URL of the questionnaire was sent to 50 e-mail addresses of persons who had been involved in the development or maintenance of the target system. Since e-mail was used so there was a little risk of old or invalid e-mail addresses. In fact, two of the addresses were reported old by the mail system, so the real count was 48.

The questionnaire consisted of five main parts. The first part handled the background information of the participant: age and work experience. Also they were asked how long time had elapsed from the last work experience with a database application (*e.g.* the target system). Then parts 2-4 dealt with the problematic nature of some maintenance and development tasks. There were also asked the task importancy. The tasks are identical to the tasks represented in the earlier chapter in table 6.2. The final part had a question about whether the participant had answered to the earlier questions based on possessed experience with Korppi or on other experience or the both.

The answering time of the questionnaire was initially two weeks. After 17 answers were received in that time period, another notification of the questionnaire was sent and the answering time was extended to three weeks. Finally, 22 answers were gotten. This covered 46% of the total 48 persons to whom the questionnaire was meant.

7.4 The limitations of the study

It is important to note that the whole population does not consist of all database application developers but only of the developers and maintainers of the target system. The target system differs significantly from other system because it has been developed in small projects during several years. This development process has had an influence to the current architecture of the system.

Some of the e-mail addressess, where the notification was sent, were expired or not used anymore. Especially those who had graduated and employed years ago, had certainly switched the e-mail address. They were typically developing first modules of the system and the nature of their tasks differs therefore from the tasks of the latest developers. So the persons who answered to the questionnaire do not represent very well the whole group of the maintainers and developers. but describe the latter situation of the Korppi.

7.5 The results

This section represents the answers of the questionnaire. Subsection 7.5.1 describes the background properties of the subjects. Subsection 7.5.2 represents a table of the task containing the problemacy value. Subsection 7.5.3 represents a similar table about the task importance. Other tasks which were received as literal answers are described in subsection 7.5.4.

7.5.1 Subject background

The background of the subjects (the persons who answered to the questionnaire) is described here. The subjects do not represent very well the whole group of the maintainers of the target system, as discussed in section 7.4.

The subjects were typically young, 24-27 year old (half of them). Most of the subjects (72%) had been involved in developing database application inside a year. The IT-related work experience was quite high because the highest class (over 3 years of experience) was 40% while the smallest class (below year) was covering only 23% of the subjects. The subjects were quite experienced because 55% of them had over 2 years of work experience

There were some subjects who had been as a project member, technical advisor and

other maintainer or developer. Almost every one of the subjects had been involved in the development of the target system as a project member. About 27% of the subjects had been technical advisors and 41% had been also developing or maintaining the system within a different relationship. Generally, the subjects had deep experience about the target system.

The types of the maintenance tasks were also asked. The asked types were corrective, perfective, adaptive and preventive. Some subjects had been doing all kinds of maintenance, but the most frequent type was perfective, 59%. Corrective was also quite frequent, 46% while preventive and adaptive were quite rare, 18% and 23%.

They were also asked how much the answers about task problemacies and importancies were based on the experience with the target system. There were five alternatives: completely based on the target system, mainly based on the target system but partly to other systems, both the target system and other systems, mainly other systems and partly the target system and completely other systems. This question was asked in the end of the questionnaire because that point was the best to determine the answer to this question. Most of the subjects had experience with other systems also (see table řefBaseOfAnswersTable).

Table 7.1: Experience types of which the answers are based on

	Frequency	Percent (%)
Completely the target system	2	9.1
Mainly the target system but partly other	7	31.8
Both the target system and other	8	40.9
Mainly other and partly the target system	4	18.2
Completely other	0	0

7.5.2 Problematic tasks

The subjects were asked the task problemacy especially related to database usage. It is important that the subjects were told to answer to the problemacy and importancy questions using their experience from both the target system and other experience. The alternatives to the question about task problemacy were 0 (not problematic at all), 1, 2, 3, 4 and 5 (very problematic). Importance of the tasks related to the totality of the working were also in a similar scale: 0 (not important at all), 1, 2, 3, 4 and 5 (very important). Means of the task problemacies were calculated.

Table 7.2 describes the tasks and their problemacies. The column **N** shows the number of subjects that answered to the question. Problemacy is described by mean value. Categories (column **Cat** in the table) are P (Program), D (Database), R (Relationship between

the application and the database) and S (SQL). The categories are based on the classification presented in section 6.7. The column **STDEV** presents the standard deviation of the problemacy.

The two most problematic tasks were *Determining the effects of changes* and *Dealing with the complexity of functionality* from the **program** category. The third was *Finding out which DB entities are used by a specific part of the application* from the **Database–Application relationship** category.

Table 7.2: Task problemacy

Task	Cat	N	Mean	STDEV
Determining the effects of changes	P	22	3.82	1.181
Dealing with the complexity of functionality	P	22	3.00	1.195
Finding out which DB entities are used by a specific part of the application	R	22	2.95	1.253
Dealing with the complexity in SQL statement	S	21	2.86	1.276
Locating a bug	P	22	2.77	1.232
Dealing with the complexity of data	D	22	2.77	1.510
Slowness of an SQL query	S	22	2.64	1.293
Understanding the functionality stored in the database	D	22	2.64	1.093
Finding out where a DB entity is used	R	22	2.50	1.263
Finding out the mappings between the classes and tables	R	22	2.27	1.486
Understanding data flow relations between the application modules through DB usage	R	22	2.27	0.703
Finding out call relationships	R	22	2.24	1.136
Fixing a bug	P	21	2.18	1.435
Understanding how the SQL statement is constructed	S	22	2.14	1.246
Understanding and managing the bindings between Java code and SQL code	R	21	2.10	1.179
Understanding the relationships between tables	D	22	2.09	1.444
Locating relevant place of code	P	21	2.05	1.117
Understanding data flow relations between the database and the application	R	22	2.00	0.816
Understanding dynamic binding	P	21	1.95	1.322
Understanding the transactions	R	22	1.82	1.140
Understanding the meaning of a part of the program	P	22	1.77	0.973
Understanding the program at general level	P	22	1.73	1.120
Understanding the database interface	R	22	1.64	1.049
Finding out where particular data is stored	D	22	1.59	1.141
Understanding the database at general level	D	22	1.55	1.438
Changing the database schema	D	22	1.50	1.263
Understanding the meaning of a database entity	D	22	1.41	1.098
Determining how the data types should be matched between SQL and Java	S	22	1.36	1.093
Understanding the meaning and the impacts of an SQL statement	S	22	1.27	1.077
Determining the data type of a column	D	22	0.64	0.727

7.5.3 Important tasks

The task importancy was asked in order to find out whether the most problematic tasks are important for the work. The tasks may not be similarly important or relevant if they are compared to the success of executing maintenance tasks.

What is most important, is that if the tasks are executed well enough, maintenance in the future becomes easier. For example, if the maintainer is changing the schema by renaming some columns, and he does not check accurately all the places of source code, where the columns are referenced, he may seem to get the system working. But in the future, the system may fail because of invalid references in some SQL clause inside the application source code.

The subjects were asked how important these tasks are when considering the performance of the whole maintenance work. The scale was 0–5 where zero means *not important at all* and five means *very important*. Table 7.3 shows the tasks ordered by importance (described as mean in the table). Column Cat shows the category of the task, column N presents the number of answerers to the question, and column STDEV shows the standard deviation of the answers

The three most important tasks were related to database and SQL categories: *Understanding the relationships between tables* (database category), *Understanding the meaning and the impacts of an SQL statement* (SQL category) and *Understanding the database at general level* (DB category).

7.5.4 Other tasks

Literal answers were received which contained information about problematic tasks and feedback about the reasons, why there were some problems in the maintenance. The individual answers were:

- Ensuring expandability was challenging. Optimizing the code too early lead into poor extensibility.
- Communicating and concurrent changes were quite problematic (3 at scale 0-5).
- When changing the database schema, removing a column or changing the data type of a column is very difficult (5) but adding a column is not difficult at all.
- The SQL statements were spread all over the application source code.
- The tools used were slow and unstable. Much time was spent with configuring the tools to work properly.

Table 7.3: Task importance

Task	Cat	N	Mean	STDEV
Understanding the relationships between tables	D	22	4.18	0.644
Understanding the meaning and the impacts of an SQL statement	S	22	4.09	0.921
Understanding the database at general level	D	22	4.09	1.019
Understanding the program at general level	P	22	4.00	1.345
Determining the effects of changes	P	22	3.82	1.332
Understanding the functionality stored in the database	D	22	3.82	1.097
Dealing with the complexity of functionality	P	22	3.82	1.097
Slowness of an SQL query	S	22	3.77	1.110
Dealing with the complexity of data	D	22	3.77	0.869
Locating a bug	P	22	3.73	1.241
Understanding the meaning of a part of the program	P	22	3.68	0.995
Understanding the meaning of a database entity	D	22	3.64	1.136
Dealing with the complexity in SQL statement	S	22	3.59	0.959
Fixing a bug	P	22	3.59	1.403
Understanding data flow relations between the database and the application	R	22	3.45	0.912
Understanding and managing the bindings between Java code and SQL code	S	21	3.38	1.161
Understanding how the SQL statement is constructed	S	22	3.36	1.177
Understanding data flow relations between the application modules through DB usage	R	22	3.23	1.020
Finding out where particular data is stored	D	22	3.23	1.193
Locating relevant place of code	P	22	3.14	1.521
Finding out call relationships	P	21	3.10	1.136
Understanding dynamic binding	P	21	3.05	1.499
Finding out which DB entities are used by a specific part of the application	R	22	2.95	1.463
Understanding the transactions	R	22	2.91	1.269
Finding out the mappings between the classes and tables	R	22	2.91	1.306
Understanding the database interface	R	22	2.82	1.332
Changing the database schema	D	22	2.68	1.211
Finding out where a DB entity is used	R	22	2.59	1.368
Determining how the data types should be matched between SQL and Java	S	22	2.41	1.260
Determining the data type of a column	D	22	2.18	1.259

8 Analysis

The results of the questionnaire are analyzed here. Section 8.1 discusses about how the categories are different. Section 8.2 describes the task significance by calculating the mean of problemacy and importance. Section 8.3 discusses about the most significant problems. Information needs related to these problems are then discussed in section 8.4.

8.1 Categories differ

It is notable that the two database related tasks were at the highest positions (first and third) by importance, but were not regarded problematic. Similar but reversed phenomenon exists with database–application relationship related task *Finding out which DB entities are used by a specific part of the application.* It is third in the problemacy list but 22nd in the importance list. Task *Finding out where a DB entity is used*behaves also similarly. Generally, database–application relationship is not regarded as a very important category but the tasks of the category are regarded quite problematic.

8.2 Task significance

By taking average of the two averages (importance and problemacy), we get table 8.1 which describes the significance of the problems in the maintenance situations. Table column **Cat** means the task category. The table headers **S**, **P** and **I** mean significance (mean of **P** and **I**), problemacy and importance.

Minimum significance was as low as 1.41 while the maximum was 3.82. The mean of the significance numbers was 2.74 and median was also 2.74.

The five most significant tasks are related to change effects, software complexity and corrective maintenance (*Locating a bug*). This is well aligned with the fact that about half of the subjects had been involved in corrective maintenance tasks related to the target system. When finding and fixing bugs, it is essential to manage the complexity if the size of the software increases. If the sofware is very complex, dealing with the effects of changes becomes difficult and important because in a complex systems there are more unknown dependencies that may lead to unexpected side effects.

Tasks may be observed by the categories specified earlier, but also with another way.

Table 8.1: Task significance

Task	Cat	S	P	I
Determining the effects of changes	P	3.82	3.82	3.82
Dealing with the complexity of functionality	P	3.41	3.00	3.82
Dealing with the complexity of data	D	3.27	2.77	3.77
Locating a bug	P	3.25	2.77	3.73
Dealing with the complexity in SQL statement	S	3.24	2.86	3.59
Understanding the functionality stored in the database	D	3.23	2.64	3.82
Slowness of an SQL query	S	3.20	2.64	3.77
Understanding the relationships between tables	D	3.14	2.09	4.18
Finding out which DB entities are used by a specific part of the application	R	2.95	2.95	2.95
Fixing a bug	P	2.89	2.18	3.59
Understanding the program at general level	P	2.86	1.73	4.00
Understanding the database at general level	D	2.82	1.55	4.09
Understanding how the SQL statement is constructed	S	2.75	2.14	3.36
Understanding data flow relations between the application modules through DB usage	R	2.75	2.27	3.23
Understanding and managing the bindings between Java code and SQL code	S	2.74	2.10	3.38
Understanding data flow relations between the database and the application	R	2.73	2.00	3.45
Understanding the meaning of a part of the program	P	2.73	1.77	3.68
Understanding the meaning and the impacts of an SQL statement	S	2.68	1.27	4.09
Finding out call relationships	P	2.67	2.24	3.10
Locating relevant place of code	P	2.62	2.05	3.14
Finding out the mappings between the classes and tables	R	2.59	2.27	2.91
Finding out where a DB entity is used	R	2.55	2.50	2.59
Understanding the meaning of a database entity	D	2.52	1.41	3.64
Understanding dynamic binding	P	2.50	1.95	3.05
Finding out where particular data is stored	D	2.41	1.59	3.23
Understanding the transactions	R	2.36	1.82	2.91
Understanding the database interface	R	2.23	1.64	2.82
Changing the database schema	D	2.09	1.50	2.68
Determining how the data types should be matched between SQL and Java	S	1.89	1.36	2.41
Determining the data type of a column	D	1.41	0.64	2.18

Some of the tasks require general level understading and do not require going to the specifics, *e.g. Understanding the program at general level, Understanding the database at general level* and *Understanding data flow relations between the database and the application.* These tasks were relatively significant (2.73 – 2.86) and the most significant of them was *Understanding the program at general level.*

The specific level understanding tasks like *Understanding the meaning of a part of the program, Understanding the meaning of a database entity* and *Understanding the meaning and the impacts of an SQL statement* are located in the latter half of the list. The significance values 2.73, 2.52 and 2.68 are slightly below the mean 2.74. The subjects clearly do not consider them of being problematic.

Data type related tasks *Determining the data type of a column* and *Determining how the data types should be matched between SQL and Java* are the least significant tasks. Data types are not a very challenging issue for maintainers though there exists some oddities with SQL and Java data type mapping.

Surprisingly, some of the tasks which are usually regarded as serious issues in program comprehension were not very high on the list. For example, the task *Finding out call relationships* was below than the average, with significance 2.67. Also, *Understanding dynamic binding*, 2.50 was the seventh from the bottom. This could mean that database applications would be remarkably different than the other applications.

8.3 The most significant problems

This section discusses about the most significant (problematic and important) problems. Based on the results, the most significant tasks are as follows.

1. *Determining the effects of changes*
2. *Dealing with the complexity of functionality*
3. *Dealing with the complexity of data*
4. *Locating a bug*
5. *Dealing with the complexity in SQL statement*
6. *Understanding the functionality stored in the database*
7. *Slowness of an SQL query*
8. *Understanding the relationships between tables*
9. *Finding out which DB entities are used by a specific part of the application*
10. *Fixing a bug*

These problems are presented in figure 8.1 with boxplots. The problems are in the same order in the figure as in the previous list. The figure describes the mean and the range of the answers.

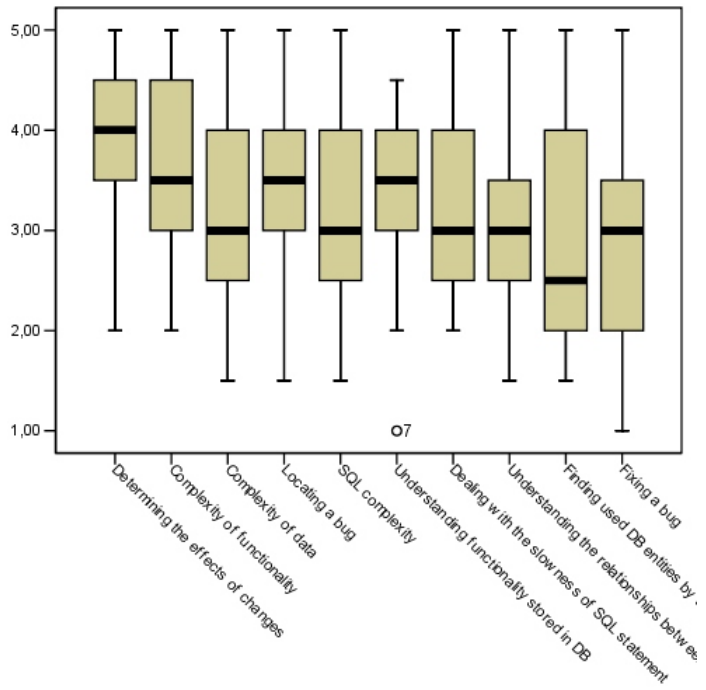

Figure 8.1: The boxplots of ten most significant problems ordered by significance.

Dealing with change effects with a big software is difficult and important. These results can be explained with following facts. The target system is very large, containing over 255,000 lines of code. The SQL statements inside the Java source code are also related to this problem, as stated in section 5.4. Because the SQL statements of the target system are spread all over the source code, this problem is emphasized.

Dealing with the complexity of the application and database is another very demanding issue for the maintainers. The complexity of the target system has increased continuosly, because the system has been developed in separate projects where the project members have not had earlier experience of the system. Every project has been assigned to add certain functionality to the system and some projects have also partly maintained the existing functionality. The increasing complexity has been tried to manage by training the members and explaining the architecture of the system for them. However, due to limited project time and limited experience of developing Java/JSP relational database applications, every project has developed their own kind of architecture. The project members have not been too aware of the existing functionality of the system and have end up adding redundant code to the system. As an example of the mixed architecture (or lack of architecture), one column is referenced

in over 30 Java or JSP files.

Finding and fixing bugs is the heart of corrective maintenance. The debugging features of the development tools are in important role in these tasks. The target system is based on multiple servers (Tomcat for JSP and PostgreSQL for database management system) which complicated the debugging sessions. For example, launching a local Tomcat server for debugging may require tens of seconds. The system was also debugged by writing debugging messages, running the system, and inspecting the log files, which is sometimes slow. The developers also complained about that the development tools were not reliable enough.

Understanding functionality stored in the database being one of the most significant problems is quite a surprising result. This issue has not been discussed in the theory. The article [HMD01] about regression testing emphasizes this because the functionality of DB may create data flow dependencies between the tables which are not always visible for the maintainer.

SQL queries tend to be slow if they are written without being aware of the performance issues. The slowness also depends on the specialities of the database management systems. Some structures may be optimized by the server while others may not.

A very interesting result is, that finding out which DB entities (tables, views or columns) are used in the application was also a significant problem. This result confirms the hypothesis that the relationship between the application and the database is in an important role in the maintenance of the target system. This problem is quite more significant than its inverse version *Finding out where a DB entity is used*. Based on my own experiences of the maintenance of the target system, I expected that this would have be less significant than the inverse version. This may depend on the type of the maintenance task. For example, a situation where a table contains erroneous data is related to the problem *Finding out where a DB entity is used* while a situation with unawareness of how the database used from an application module is related to the *Finding out which DB entities are used by a specific part of the application*. Figure 8.2 describes the differecies between the answerers who had been involved in corrective and perfective maintenance tasks (some of the were involved in both) and their opininons about these tasks. Preventive and adaptive maintenance are not discussed here because so few of the subjects had been involved in those types of software maintenance.

The means, which are present in figure 8.2 are calculated from the significancy of the task (mean of problemacy and importancy). Finding used DB entitites is much more significant for the subjects who had been in corrective maintenance than who had not. In the perfective maintenance, there is no such difference. The task *Finding out where a DB entity is used* behaves similarly, it is regarded as much more significant with the users who have been doing either corrective, perfective or both types of maintenance.

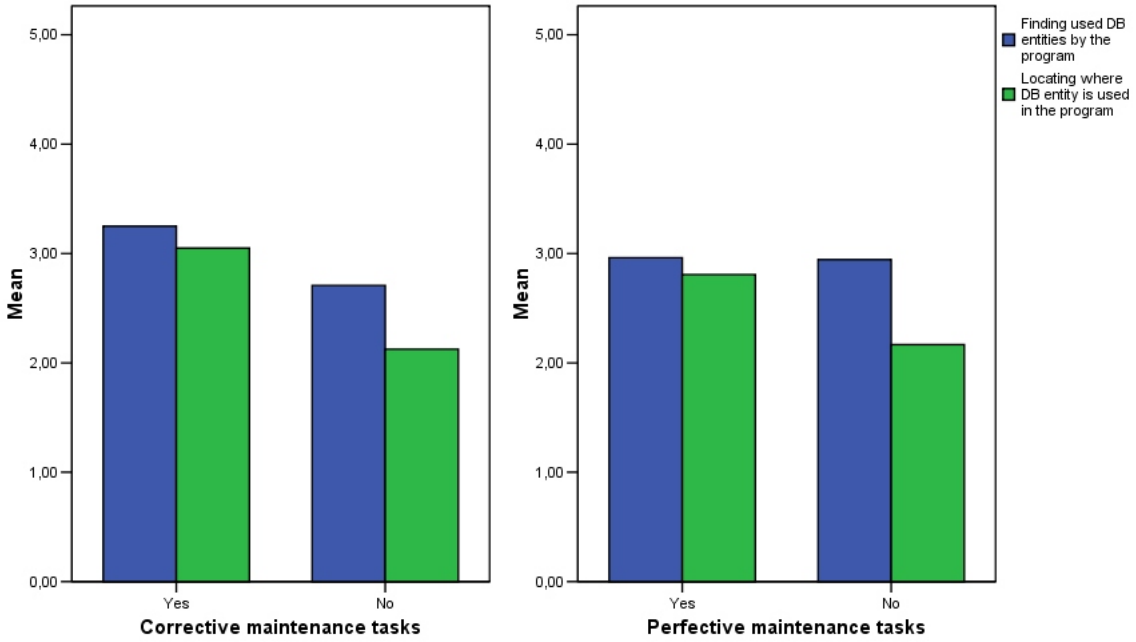

Figure 8.2: Corrective and perfective maintenance tasks related to the two problems of understanding the dependencies between the database and the application.

Figure 8.3 gives more information about the ten most significant tasks. The tasks are numbered as 1 to 10 and plotted in the scatterplot with the axis CY and CN. The axis CY means the significancy of the task in the group of answers who had been involved in corrective maintenance. The axis CN is similar but with the users who had not been involved in it. As shown in the figure, some tasks are at the diagonal, which means that they are tasks that are not dependant on this issue at all. On the other hand, some tasks belong to the corner where CN is high but CY is low, which means that the tasks are not so significant in corrective maintenance.

Figure 8.4 gives similar information about the properties of the tasks in perfective maintenance. The tasks are the same tasks as above. The axis PY means the significancy of the task in the group of answers who had been involved in perfective maintenance. The axis PN is similar but with the users who had not been involved in it.

As a result of searching correlation between the task significance and the maintenance type (corrective and perfective), one statistically significant (at the 0.05 level) correlation was found. Complexity of data is correlated with perfective maintenance, as shown in the figure 8.5. The Spearman correlation coefficient was 0.481 and two-tailed significancy 0.023. The correlation predicts that if a person been involved in perfective maintenance, he does not keep task *Dealing with the complexity of data* so significant (problematic and important for the whole task).

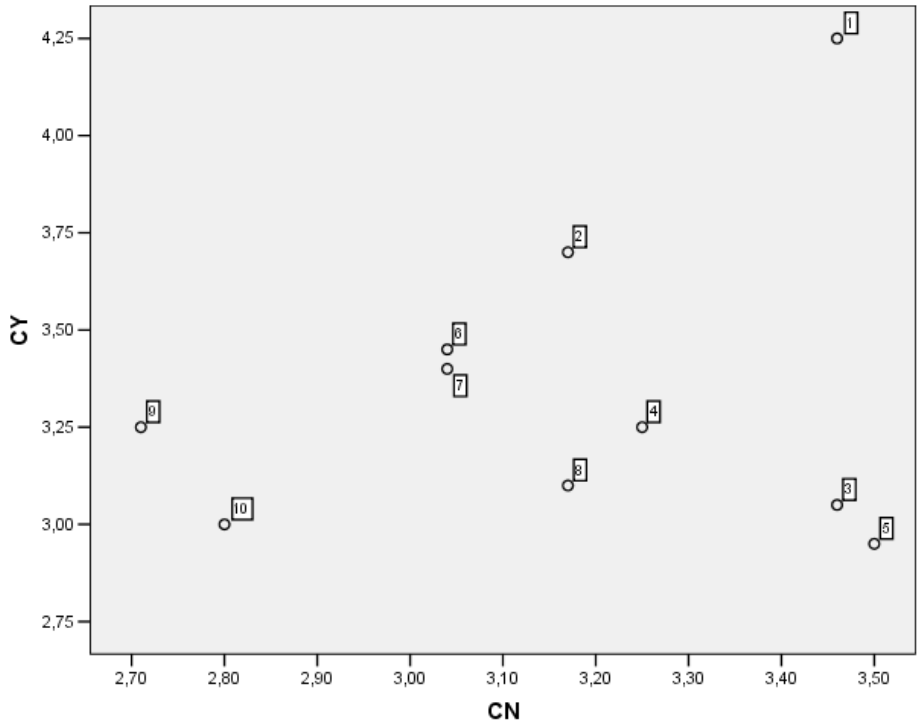

Figure 8.3: The ten most significant tasks and their significancies in corrective maintenance.

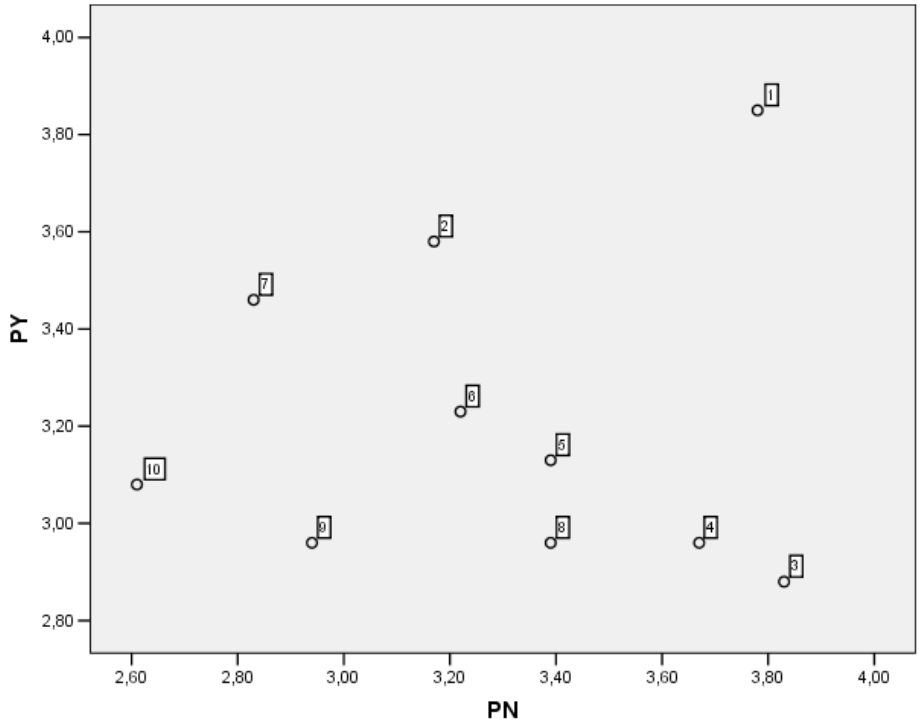

Figure 8.4: The ten most significant tasks and their significancies in perfective maintenance.

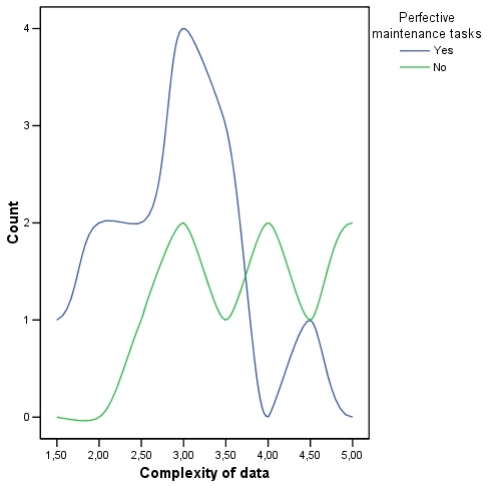

Figure 8.5: Perfective maintenance tasks related to the data complexity problem.

Some other correlations between the task significancies also exist, *e.g.*

- *Understanding the relationships between tables* and *Dealing with the complexity of data*
- *Understanding the relationships between tables* and *Dealing with the complexity in SQL statement*
- *Understanding the functionality stored in the database* and *Dealing with the complexity of data*
- *Dealing with the complexity of functionality* and *Determining the effects of changes*

The first and second correlations are fully logical because complex relationships are one factor of the data and SQL complexity. The third correlation is also clear. More complex the data goes, the more difficult is to understand the functionality which uses the complex data. The fourth describes that if the application is very complex, the effects of changes can be difficult to determine. Inspecting the correlations between the task significancies is not the goal of the study, but it is inspiring to see that there are clear and logical connections between the variables.

8.4 Essential information needs

Essential information needs related to the problems presented above are discussed here. Very essential is to understand the whole IS (application, database and data flow) at a general level in order to maintain the application successfully. The general level understanding was however mentioned to be quite trivial. The change effects occur when there are lots of dependencies in software. If the developer is not aware of all the relevant dependencies he

may cause side effects while changing the code. It is essential to manage the dependencies.

The problems about managing the complexity of the software requires some discussion. Complexity can be managed with abstractions. A complex system can be reverse engineered to a simpler model which is also easier to understand and manage. The complexity reside in the program functionality, in the database or in the SQL statements. The complexity is often due to bad design. Thereby, reengineering database applications to a new design is a relevant issue.

Corrective maintenance is a process that needs support. With poor debugging tools and the target system with lots of artifacts and dependencies, there are problems. Locating a bug is done by *e.g.* inspecting logs, reading code, trying to understand a piece of code, running debugger or just simulating the code in mind. The essential thing for bug hunting is a good understanding of the system, both database and the application.

Understanding the functionality stored in the database proves to be problematic. Stored procedures and triggers may cause multiple dependencies (mainly data flow) between the application and the database, between the application modules and between the database tables. Managing these dependencies is important, especially in testing [HMD01]. A stored procedure can be understood by inspecting the source code places where it is referenced.

Optimizing SQL is a task that needs knowledge about the database server implementation issues and relation algebra [EN99]. Complex queries are difficult to understand and thereby hard to optimize.

Larger database applications include hundreds of database tables. Relationships between tables are complex and access paths difficult to find. Database documentation might be useful in understanding the relationships. There are also several approaches to extract a conceptual model from a relational database, *e.g.* [Alh03]. Using the model, relationships may become understandable. In addition to the documentation, a source of information for understanding relationship is the application source code. The SQL statements which use multiple tables with `JOIN` clause contain information about the relationships between the tables [Alh03]. Thereby, this problem requires undestanding where those SQL statements are located in the source code which belongs to the task *Finding out which DB entities are used by a specific part of the application*

Finding out which DB entities (*e.g.* tables) are used in the application is a challenging task. It is not usually documented, and the traditional reverse engineering tools are not capable of mining the information. What makes it difficult to reverse engineer, is the mix of the technologies: Java (general-purpose language), JSP (user interface and a sort of language), SQL (database language) and PostgreSQL (database management system that presents some requirements for the used SQL syntax). They all define the requirements for the reverse engineering process and make the simple thing complex.

Essential dependencies which cover these information needs are as follows.

- A class or module uses a table/view
- A class or module uses a column
- An SQL statement in the application uses a relationship
- A table relates (with 1:1, 1:M or N:M type relationship) to a table.
- A class or method uses a stored procedure

9 Conclusion

Maintenance of relational database applications differs essentially from maintaing other kinds of applications. Mix of program and database technologies increases complexity. Relational databases involve concepts like table and column. Database usage is mapping between the database and the application. The maintainers need to observe this in order to successfully manage changes to the application. The maintainers need various types of comprehension: program comprehension and database comprehension.

Both database comprehension and program comprehension are vital for maintaining database applications. With large applications, *e.g.* over 100 tables and 20,000 LOC, it is also important to understand and manage the dependencies between the application and the database.

An empirical study was conducted to find out what are the most significant problems and related information needs when maintaining a database application called Korppi. Based on the answers of 22 maintainers or developers, the most significant problems were *e.g.* dealing with the effects of changes, managing the complexity (of functionality and data), understanding table relationships and finding out which database tables, columns or views are used in the application.

In database application maintenance, there are such information needs that are peculiar to database applications. The information needs are basically related to managing all sorts of software dependencies, including the database dependencies and the dependencies between the application and the database. For example, *Class uses table* is a dependency which was regarded more important than call dependency.

The limitations of the study are due to the properties of the target system. Because the answers based so heavily on the experience with the target system, it is sure that some of detected problems may be unique for the system.

10 References

[ABDM01] Alain Abran, Pierre Bourque, Robert Dupuis, and James W. Moore. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, 2001.

[AEP96] Jacqueline M. Antis, Stephen G. Eick, and John D. Pyrce. Visualizing the Structure of Large Relational Databases. *IEEE Softw.*, 13(1):72–80, 1996.

[Alh03] Reda Alhajj. Extracting the Extended Entity-Relationship Model from a Legacy Relational Database. *Information Systems*, 28(6):597–618, 2003.

[Amb03] Scott W. Ambler. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. John Wiley & Sons, 2003.

[BH92] Erich Buss and John Henshaw. Experiences in Program Understanding. In *Proceedings of the 1992 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 157–189. IBM Press, 1992.

[BR00] K. H. Bennet and V. T. Rajlich. Software Maintenance and Evolution: A Roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 73–87. ACM Press, 2000.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Longman Publishing Co., Inc., 1999.

[CF03] Yossi Cohen and Yishai A. Feldman. Automatic High-Quality Reengineering of Database Programs by Abstraction, Transformation and Reimplementation. *ACM Transactions on Software Engineering and Methodology*, 12(3):285–316, 2003.

[Che76] Peter Pin-Shan Chen. The Entity-Relationship Model–Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[CHK+01] Ned Chapin, Joanne E. Hale, Khaled Md. Kham, Juan F. Ramil, and Wui-Gee Tan. Types of Software Evolution and Software Maintenance. *Journal of Software Maintenance*, 13(1):3–30, 2001.

[CI90] Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Softw.*, 7(1):13–17, 1990.

[CMS03] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise Analysis of String Expressions. In *Proceedings of the 10th International Static Analysis Symposium, SAS'03*, volume 2694 of *LNCS (Lecture Notes in Computer Science)*, pages 1–18. Springer-Verlag, June 2003.

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[CPG01] Coral Calero, Mario Piattini, and Marcela Genero. A Case Study with Relational Database Metrics. In *ACS / IEEE International Conference on Computer Systems and Applications (AICCSA 2001)*, pages 485–487. IEEE Computer Soc., 2001.

[EHF01] Jon Ellis, Linda Ho, and Maydene Fischer. *JDBC API Specification 3.0 Final Release*. Sun Microsystems, Inc., 2001.

[EMK+04] Andrew Eisenberg, Jim Melton, Krishna Kulkarni, Jan-Eike Michels, and Fred Zemke. SQL:2003 Has Been Published. *ACM SIGMOD Record*, 33(1):119–126, 2004.

[EN94] Ramez A. Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, 2nd Edition.* Benjamin/Cummings, 1994.

[EN99] Ramez A. Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, 3rd edition.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[GJS96] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[Gos97] James Gosling. The Feel of Java. *Computer*, 30(6):53–57, 1997.

[GSD04] Carl Gould, Zhendong Su, and Premkumar Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Soc., 2004.

[HEH+95] Jean-Luc Hainaut, Vincent Englebert, Jean Henrard, Jean-Marc Hick, and Didier Roland. Requirements for Information System Reverse Engineering Support. In *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*, page 136. IEEE Computer Soc., 1995.

[HEH+98] Jean Henrard, Vincent Englebert, Jan-Marc Hick, Didier Roland, and Jean-Luc Hainaut. Program Understanding in Databases Reverse Engineering. In *Proceedings of the 9th International Conference on Database and Expert Systems Applications,* pages 70–79. Springer-Verlag, 1998.

[Hen03] Jean Henrard. *Program Understanding in Database Reverse Engineering*. PhD thesis, University of Namur, Belgium, jhe@info.fundp.ac.be, 2003.

[HHH+97] Jean Henrard, Jean-Luc Hainaut, Jan-Marc Hick, Didier Roland, and Vincent Englebert. Contribution to the Reverse Engineering of OO Applications - Methodology and Case Study. In *Proceedings of the IFIP 2.6 Working Conference on Database Semantics*. Chapman-Hall, 1997.

[HHH+99] Jean Henrard, Jean-Luc Hainaut, Jan-Marc Hick, Didier Roland, and Vincent Englebert. Data Structure Extraction in Database Reverse Engineering. In *Proceedings of the Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling*, pages 149–160. Springer-Verlag, 1999.

[HLCW99] Chia-Lin Hsu, Hsien-Chou Liao, Jiun-Liang Chen, and Feng-Jian Wang. A Web Database Application Model for Software Maintenance. In *Proceedings of the Fourth International Symposium on Autonomous Decentralized Systems*, pages 334–338. IEEE Computer Soc., March 1999.

[HMD01] Ramzi A. Haraty, Nash'at Mansour, and Bassel Daou. Regression Testing of Database Applications. In *Proceedings of the 2001 ACM Symposium on Applied Computing*, pages 285–289. ACM Press, 2001.

[IBM02] *Application Development Guide: Programming Client Applications, Version 8*. IBM, 2002.

[ISO03] ISO/IEC. *Information Processing Systems - Database Language SQL*. 2003.

[JSKCG94] K. Jambor-Sadeghi, M. A. Ketabchi, J. Chue, and M. Ghiassi. A Systematic Approach to Corrective Maintenance. *The Computer Journal*, 37(9):764–778, 1994.

[Kon98] Manu Konchady. An Introduction to JDBC. *Linux J.*, 1998(55es):2, 1998.

[Kor05] Korppiadmin. Historia - korppi, 2005. Available at <https://korppi.it.jyu.fi/kotka/help/faq/history.jsp>, checked 19.1.2005.

[LS81] Bennet P. Lientz and E. Burton Swanson. Problems in application software maintenance. *Communications of the ACM*, 24(11):763–769, 1981.

[Lu02] Jianguo Lu. Reengineering of Database Applications to EJB Based Architecture. In *Proceedings of the 14th International Conference on Advanced Information Systems Engineering*, pages 361–376. Springer-Verlag, 2002.

[LZ03] Liwu Li and Xin Zhao. UML Specification of Relational Database. *Journal of Object Technology*, 2(5):87–100, 2003.

[Mic95] Microsoft Corporation. *Open Database Connectivity*, 1995. Available at <http://www.microsoft.com/data/odbc/default.htm>, checked 19.3.2005.

[MJS+00] Hausi A. Müller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, and Kenny Wong. Reverse Engineering: A Roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 47–60. ACM Press, 2000.

[NHR99] Udo Nink, Theo Härder, and Norbert Ritter. Generating Call-Level Interfaces for Advanced Database Application Programming. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 575–586. Morgan Kaufmann Publishers Inc., 1999.

[OMG03] OMG. Unified Modeling Language (UML) Specification (2003). Version 1.5, March 2003, UML Specification, revised by the OMG on the World Wide Web: http:// www.omg.org, 2003.

[PB94] William J. Premerlani and Michael R. Blaha. An Approach for Reverse Engineering of Relational Databases. *Communications of the ACM*, 37(5):42–49, 1994.

[PM00] Mario Piattini and Antonio Martínez. Measuring for Database Programs Maintainability. In Mohamed T. Ibrahim, Josef Küng, and Norman Revell, editors, *DEXA*, volume 1873 of *Lecture Notes in Computer Science*, pages 65–78. Springer, 2000.

[Ree97] George Reese. *Database Programming with JDBC and Java*. O'Reilly, 1997.

[SKM01] Tarja Systä, Kai Koskimies, and Hausi Müller. Shimba – An Environment for Reverse Engineering Java Software Systems. *Software – Practice & Experience*, 31(4):371–394, 2001.

[Sny93] Alan Snyder. The Essence of Objects: Concepts and Terms. *IEEE Software*, 10(1):31–42, 1993.

[Som01] Ian Sommerville. *Software Engineering (6th Ed.)*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[Suo97] Timo Suominen. Upotetun SQL:n analysointi Hypersoft-järjestelmässä. Master's thesis, University of Helsinki, Finland, 1997.

[Swa76] E. Burton Swanson. The Dimensions of Maintenance. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 492–497. IEEE Computer Soc., 1976.

[TP96] Antti-Pekka Tuovinen and Jukka Paakki. Translating SQL for Database Reengineering. *ACM SIGPLAN Notices*, 31(2):21–26, 1996.

[Tuo95] Antti-Pekka Tuovinen. Analyzing, Understanding and Maintaining Object-Oriented Programs. Technical report, HyperSoft Project, 1995. Dept. of Computer Science, University of Helsinki.

[vMV95a] Anneliese von Mayrhauser and A. Marie Vans. Industrial Experience with an Integrated Code Comprehension Model. *Software Engineering Journal*, 10(5):171–182, 1995.

[vMV95b] Anneliese von Mayrhauser and A. Marie Vans. Program Comprehension During Software Maintenance and Evolution. *Computer*, 28(8):44–55, 1995.

[vMV97] Anneliese von Mayrhauser and A. Marie Vans. Program Understanding Behavior During Debugging of Large Scale Software. In *ESP '97: Papers Presented at the Seventh Workshop on Empirical Studies of Programmers*, pages 157–179. ACM Press, 1997.

[WH92] Norman Wilde and Ross Huitt. Maintenance Support for Object-Oriented Programs. *IEEE Transactions on Software Engineering*, 18(12):1038–1044, 1992.

[Wyk03] Jens Wyke. JDBC Query Logging Made Easy, 2003. Available at <http://www-128.ibm.com/developerworks/java/library/j-loggable.html>, checked 1.3.2005.

[XOP95a] *X/Open Guide: Data Management: Reference Model*. X/Open Company Limited, 1995.

[XOP95b] *X/Open Technical Standard: Data Management: SQL Call Level Interface (CLI)*. X/Open Company Limited, 1995.

A Appendix. Query form

Kyselyyn vastaaminen

Korppi-järjestelmän ylläpitäjien/kehittäjien ongelmat liittyen tietokannan käyttöön

Taustatiedot

Oletko ollut mukana Korppi-järjestelmää kehittämässä...

sovellusprojektissa toteutusryhmän jäsenenä?	sovellusprojektin teknisenä ohjaajana?	muulla tavalla ylläpitäjänä/kehittäjänä?
☐	☐	☐

Jos tehtäviisi on kuulunut ylläpitoa, niin minkätyyppisissä ylläpitotehtävissä olet ollut mukana?

Korjaava (korjataan bugeja)	Täydentävä (lisätään uutta toiminnallisuutta)	Muokkaava (sovitetaan ohjelmistoa uuteen tai muuttuneeseen ympäristöön)	Ehkäisevä (parannetaan ylläpidettävyyttä)
☐	☐	☐	☐

Mikä on ikäsi?

- ○ Alle 20 vuotta
- ○ 20-23 vuotta
- ○ 24-27 vuotta
- ○ Yli 27 vuotta

Kuinka paljon sinulla on työkokemusta tietotekniikan alalta?

- ○ Alle vuosi
- ○ 1-2 vuotta
- ○ 2-3 vuotta
- ○ yli 3 vuotta

Kuinka pitkä aika on siitä, kun viimeksi olet työskennellyt tietokantasovelluksen (esim. Korppi) parissa?

- ○ Alle vuosi
- ○ 1-2 vuotta
- ○ 2-3 vuotta
- ○ yli 3 vuotta

Sovellukseen liittyvät ongelmat

Seuraavaksi tässä ja kolmessa seuraavassa osiossa kysytään erilaisten tehtävien ongelmallisuutta ja merkittävyyttä. Tehtävien konteksti on tietokantasovelluksen ylläpito ja kehittäminen. Tarkoituksena on siis keskittyä Korppi-järjestelmään ja tietokantasovelluksiin yleensä, sekä ohjelmiin, jotka käyttävät tietokantaa. Tärkeintä on että vastaukset perustuvat omiin kokemuksiin ja näkemykseen.

1. Kuinka ongelmalliseksi koette seuraavan tehtävän kokemuksiinne perustuen? (0 = Ei lainkaan ongelmallinen, 5 = Erittäin ongelmallinen)

	0	1	2	3	4	5
1. Sovelluksen ymmärtäminen yleisellä tasolla	○	○	○	○	○	○

2. Ohjelman osan merkityksen ymmärtäminen	○	○	○	○	○	○
3. Relevantin koodin paikan löytäminen	○	○	○	○	○	○
4. Kutsusuhteiden selvittäminen	○	○	○	○	○	○
5. Dynaamisen sidonnan ymmärtäminen	○	○	○	○	○	○
6. Muutoksen vaikutusten määrittäminen	○	○	○	○	○	○
7. Vian paikantaminen	○	○	○	○	○	○
8. Vian korjaus	○	○	○	○	○	○
9. Toiminnallisuuden monimutkaisuuden hallinta	○	○	○	○	○	○

2. Muut ongelmalliset tehtävät sovellukseen liittyen

Jos jotkut muut tehtävät sovellukseen liittyen ovat mielestäsi ongelmallisia, syötä ne tähän kenttään ja arvioi myös kunkin ongelmallisuus yo. asteikon (0-5) mukaisesti:

3. Kuinka merkittäväksi koette seuraavan tehtävän työn suorituksen kokonaisuuden kannalta kokemuksiinne perustuen? (0 = Ei lainkaan merkittävä, 5 = Erittäin merkittävä)

	0	1	2	3	4	5
1. Sovelluksen ymmärtäminen yleisellä tasolla	○	○	○	○	○	○
2. Ohjelman osan merkityksen ymmärtäminen	○	○	○	○	○	○
3. Relevantin koodin paikan löytäminen	○	○	○	○	○	○
4. Kutsusuhteiden selvittäminen	○	○	○	○	○	○
5. Dynaamisen sidonnan ymmärtäminen	○	○	○	○	○	○
6. Muutoksen vaikutusten määrittäminen	○	○	○	○	○	○
7. Vian paikantaminen	○	○	○	○	○	○
8. Vian korjaus	○	○	○	○	○	○
9. Toiminnallisuuden monimutkaisuuden hallinta	○	○	○	○	○	○

4. Muiden ongelmallisten tehtävien merkittävyys tehtävän työn suorituksen kokonaisuuden kannalta.

Jos ylempänä kysymyksessä 2 syötit muita tehtäviä, syötä tähän samat tehtävät ja arvioi kunkin merkittävyys yo. asteikon (0-5) mukaan:

Tietokantaan liittyvät ongelmat

1. Kuinka ongelmalliseksi koette seuraavan tehtävän kokemuksiinne perustuen? (0 = Ei lainkaan ongelmallinen, 5 = Erittäin ongelmallinen)

	0	1	2	3	4	5
1. Tietokannan ymmärtäminen yleisellä tasolla	○	○	○	○	○	○
2. Tietokannan osien tai alkioiden (taulu/sarake/näkymä) merkitysten ymmärtäminen	○	○	○	○	○	○
3. Taulujen välisten suhteiden ymmärtäminen	○	○	○	○	○	○
4. Tiedon tallennuspaikan selvittäminen (esim. mihin tauluun ja sarakkeeseen tallentuu opiskelijan etunimi)	○	○	○	○	○	○
5. Sarakkeen tietotyypin selvittäminen	○	○	○	○	○	○
6. Tietokantarakenteen muuttaminen (esim. sarakkeen lisäys tauluun)	○	○	○	○	○	○
7. Tietokantaan tallennetun toiminnallisuuden ymmärtäminen	○	○	○	○	○	○
8. Tiedon monimutkaisuuden hallinta	○	○	○	○	○	○

2. Muut ongelmalliset tehtävät tietokantaan liittyen

Jos jotkut muut tehtävät tietokantaan liittyen ovat mielestäsi ongelmallisia, syötä ne tähän kenttään ja arvioi myös kunkin ongelmallisuus yo. asteikon (0-5) mukaisesti:

3. Kuinka merkittäväksi koette seuraavan tehtävän työn suorituksen kokonaisuuden kannalta kokemuksiinne perustuen? (0 = Ei lainkaan merkittävä, 5 = Erittäin merkittävä)

	0	1	2	3	4	5
1. Tietokannan ymmärtäminen yleisellä tasolla	○	○	○	○	○	○
2. Tietokannan osien tai alkioiden (taulu/sarake/näkymä) merkitysten ymmärtäminen	○	○	○	○	○	○
3. Taulujen välisten suhteiden ymmärtäminen	○	○	○	○	○	○
4. Tiedon tallennuspaikan selvittäminen (esim. mihin tauluun ja sarakkeeseen tallentuu opiskelijan etunimi)	○	○	○	○	○	○
5. Sarakkeen tietotyypin selvittäminen	○	○	○	○	○	○
6. Tietokantarakenteen muuttaminen (esim. sarakkeen lisäys tauluun)	○	○	○	○	○	○
7. Tietokantaan tallennetun toiminnallisuuden ymmärtäminen	○	○	○	○	○	○
8. Tiedon monimutkaisuuden hallinta	○	○	○	○	○	○

4. Muiden ongelmallisten tehtävien merkittävyys tehtävän työn suorituksen kokonaisuuden kannalta.

Jos ylempänä kysymyksessä 2 syötit muita tehtäviä, syötä tähän samat tehtävät ja arvioi kunkin merkittävyys yo. asteikon (0-5) mukaan:

Sovelluksen ja tietokannan suhteesen liittyvät ongelmat

1. Kuinka ongelmalliseksi koette seuraavan tehtävän kokemuksiinne perustuen? (0 = Ei lainkaan ongelmallinen, 5 = Erittäin ongelmallinen)

	0	1	2	3	4	5
1. Tietokannan ja sovelluksen välisten tietovirtojen ymmärtäminen	○	○	○	○	○	○
2. Sovelluksen moduulien/osien välisten tietokannan käytöstä aiheutuvien tietovirtojen ymmärtäminen	○	○	○	○	○	○
3. Tietokannan alkion (taulun/sarakkeen/näkymän) käyttöpaikan tai käyttöpaikkojen selvittäminen sovelluksen lähdekoodeista	○	○	○	○	○	○
4. Selvittäminen, mitä kaikkia tietokannan osia/alkioita tietty sovelluksen osa käyttää	○	○	○	○	○	○
5. Taulujen ja luokkien vastaavuuden ymmärtäminen (mikä taulu sisältää luokan sisältämää tai käsittelemää tietoa)	○	○	○	○	○	○
6. Tietokantarajapinnan ymmärtäminen	○	○	○	○	○	○
7. Transaktioiden ymmärtäminen	○	○	○	○	○	○

2. Muut ongelmalliset tehtävät sovelluksen ja tietokannan suhteeseen (esim. tietokannan käyttöön) liittyen

Jos jotkut muut tehtävät sovelluksen ja tietokannan suhteeseen liittyen ovat mielestäsi ongelmallisia, syötä ne tähän kenttään ja arvioi myös kunkin ongelmallisuus yo. asteikon (0-5) mukaisesti:

3. Kuinka merkittäväksi koette seuraavan tehtävän työn suorituksen kokonaisuuden kannalta kokemuksiinne perustuen? (0 = Ei lainkaan merkittävä, 5 = Erittäin merkittävä)

	0	1	2	3	4	5
1. Tietokannan ja sovelluksen välisten tietovirtojen ymmärtäminen	○	○	○	○	○	○
2. Sovelluksen moduulien/osien välisten tietokannan käytöstä aiheutuvien tietovirtojen ymmärtäminen	○	○	○	○	○	○
3. Tietokannan alkion (taulun/sarakkeen/näkymän) käyttöpaikan tai käyttöpaikkojen selvittäminen sovelluksen lähdekoodeista	○	○	○	○	○	○
4. Selvittäminen, mitä kaikkia tietokannan osia/alkioita tietty sovelluksen osa käyttää	○	○	○	○	○	○
5. Taulujen ja luokkien vastaavuuden ymmärtäminen (mikä taulu sisältää luokan sisältämää tai käsittelemää tietoa)	○	○	○	○	○	○
6. Tietokantarajapinnan ymmärtäminen	○	○	○	○	○	○
7. Transaktioiden ymmärtäminen	○	○	○	○	○	○

4. Muiden ongelmallisten tehtävien merkittävyys tehtävän työn suorituksen kokonaisuuden kannalta.

Jos ylempänä kysymyksessä 2 syötit muita tehtäviä, syötä tähän samat tehtävät ja arvioi kunkin merkittävyys yo. asteikon (0-5) mukaan:

SQL-lauseisiin liittyvät ongelmat

1. Kuinka ongelmalliseksi koette seuraavan tehtävän kokemuksiinne perustuen? (0 = Ei lainkaan ongelmallinen, 5 = Erittäin ongelmallinen)

	0	1	2	3	4	5
1. SQL-lauseen merkityksen ja vaikutuksen ymmärtäminen	○	○	○	○	○	○
2. SQL-lauseen koostamisen ymmärtäminen (lause muodostetaan Java-koodissa useasta osasta ajonaikaisesti)	○	○	○	○	○	○
3. SQL-lauseen osien ja Java-koodin osien välille syntyvien Sidosten ymmärtäminen ja hallinta	○	○	○	○	○	○
4. Tietotyyppien vastaavuuksien määrittäminen Java-kielen ja SQL-kielen välillä	○	○	○	○	○	○
5. SQL-lauseen hitauden optimointi	○	○	○	○	○	○
6. SQL-lauseihin kuuluvan monimutkaisuuden hallinta	○	○	○	○	○	○

2. Muut ongelmalliset tehtävät SQL-lauseisiin liittyen

Jos jotkut muut tehtävät SQL-lauseisiin liittyen ovat mielestäsi ongelmallisia, syötä ne tähän kenttään ja arvioi myös kunkin ongelmallisuus yo. asteikon (0-5) mukaisesti:

3. Kuinka merkittäväksi koette seuraavan tehtävän työn suorituksen kokonaisuuden kannalta kokemuksiinne perustuen? (0 = Ei lainkaan merkittävä, 5 = Erittäin merkittävä)

	0	1	2	3	4	5
1. SQL-lauseen merkityksen ja vaikutuksen ymmärtäminen	○	○	○	○	○	○
2. SQL-lauseen koostamisen ymmärtäminen (lause muodostetaan Java-koodissa useasta osasta ajonaikaisesti)	○	○	○	○	○	○
3. SQL-lauseen osien ja Java-koodin osien välille syntyvien Sidosten ymmärtäminen ja hallinta	○	○	○	○	○	○
4. Tietotyyppien vastaavuuksien määrittäminen Java-kielen ja SQL-kielen välillä	○	○	○	○	○	○
5. SQL-lauseen hitauden optimointi	○	○	○	○	○	○
6. SQL-lauseihin kuuluvan monimutkaisuuden hallinta	○	○	○	○	○	○

4. Muiden ongelmallisten tehtävien merkittävyys tehtävän työn suorituksen kokonaisuuden kannalta.

Jos ylempänä kysymyksessä 2 syötit muita tehtäviä, syötä tähän samat tehtävät ja arvioi kunkin merkittävyys yo. asteikon (0-5) mukaan:

Vastauksien perusta

Mihin kokemuksiisi vastauksesi perustuu

Täysin Korppi-kokemuksiin	Pääosin Korppi-kokemuksiin ja hieman muihin kokemuksiin	Sekä Korppi-kokemuksiin että muuhun	Pääosin muihin kokemuksiin ja hieman Korppi-kokemuksiin	Täysin muihin kokem
○	○	○	○	○

Lähetä