The Generation of CORBA wrappers for Legacy Databases

Remco van de Woestijne

woestijn@bigfoot.com

Universiteit van Amsterdam

Faculteit der Natuurwetenschappen,Wiskunde en Informatica (FNWI)

Afstudeerrichting: Programmatuur

Afstudeerdocent: prof. dr. P. Klint

August 28, 2002

ACKNOWLEDGMENTS

I owe a debt of gratitude to my supervisor, Prof. Dr. P. Klint, for his guidance through the world of Software Engineering and that thing called 'Real Life'. I am greatly thankful to Tiel Chang (PinkRoccade R&D) for introducing me to the problems of supporting Legacy Systems. A big thank to all the colleagues of the 'San Francisco project', especially Irmen de Jong, for my first stint at Python Programming, Ard van der Scheer, for introducing me to Aquadiving. I would also thank the following people, Bas Toeter, my web-savvy Dive Buddy, Gerald Stap for all the discussions Gadgets and Games related, and Annelies for sharing the good and the bad times. Last but not least i like to express my thanks to my parents for their support over the years.

Contents

List of Figures

Chapter 1

Introduction

In this thesis a framework for the unlocking of databases using distributed object technology is explored. We will document observations made during the research of the transition of database access from a message based system to a distributed objects based system.

This work was first carried out during an internship within ASZ (Automatisering Sociale Zekerheid), a subsidiary of the 'Gak Groep'. ASZ develops and maintains several information systems on behalf of GAK. GAK is one of the largest Social Executive Bodies in the Netherlands, whose task it is to impose and facilitate the social security laws. Among the various information systems in use by GAK, there are several client/server applications, serving thousands of clients and using several large databases. The research for means of unlocking of legacy databases is interesting for two main reasons.

Firstly there are political factors that govern the decision making. There is a growing tendency of centralization. In the Netherlands there are three parties that are involved in social security

- Municipal Social Service (Gemeentelijke Sociale Dienst).
- Employment centres (Arbeidsbureaus).
- Social Executive Bodies (Uitvoerende Verzekerings Instantie).

The latest idea is to combine all these parties into a new body called Centrum voor Werk en Inkomen[1] (CWI). To connect all the autonomous information systems of the several parties, a combined solution is called for. But to prevent one party from imposing its software solution onto the others, a quest for an open and transparent solution begins.

Secondly, there is this technological angle. Most of the current applications in use within GAK, are based on the so-called OpenGAC, a set of services for accessing distributed database systems. This protocol for the communication between applications within the Gak Groep has been developed internally. The OpenGAC Protocol (Open Generiek Applicatie Communicatieprotocol [2]) was devised in 1994. The most important feature of OpenGAC is the Naming service. The Naming sevice shields the user from the physical location of services, and enables the user to request services by a simple alias. Most of the services that are offered are related

[1] Dutch for 'Centre for Employment and Income'.
[2] Dutch for 'Open Generic Application Communication Protocol'.

to database access. The Gak Groep maintains several nation-wide databases like BRP (Burger Registratie Personen), these are replicated over 5 locations, so the need for a Naming service is apparent. The introduction of an extra layer also enables authentication and management information for the database accesses. The

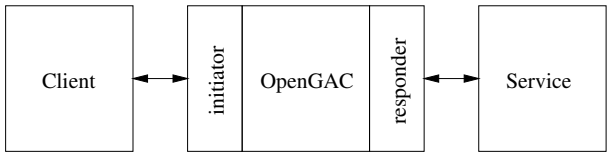


Figure 1.1: OpenGAC acts as a broker between clients and services

internally developed OpenGAC middleware solution is in turn based on Digital's DECMessageQueue product. Although this solution has shown its merit for several years, it lacks transparent database transactions. Worse yet, the continuity of the DECMessageQueue product that forms the foundation for OpenGAC, is on the brink. The vendor, Digital, will terminate the availability and support of the messaging product. Therefore a migration path has to be set out.

The research and development department of ASZ has been involved in the research of rebuilding a legacy application with distributed object orientated technologies. In the past they have used a system called Werkgever Dialoog Systeem [3] (WDS) as a pilot for these kind of studies [3]. During such a study, an interface for the database connection will have been defined in CORBA. These interfaces were written by hand, an error-prone and labor intensive process. Clearly as the number of databases grows, there is the need for automatic generation of the interfaces. Also from an engineering and consistency point of view, it would be helpful if upon change of the internal database models, the interfaces were to be regenerated automatically. We end up with database wrappers that encapsulate database records as distributed database objects, that can be accessed by our target middleware.

1.1 Impedance Mismatch

Most of the data in information systems is stored inside relational databases using a rows and columns like model. The current development of software is mostly done with object technology for reasons of better maintenance and flexibility. So we have to keep objects and relational structures synchronized. Changes made to the objects should be reflected inside the database and vice versa. Yet there is no straight mapping between objects and relational data, the so called Impedance Mismatch. So we have to come up with some sort of mapping strategy of how to represent the various rows and columns as objects with fields and methods. To accomplish these mapping techniques we can adopt reverse and forward engineering.

- **Forward engineering** takes an existing object model and constructs a relational structure out of it to store the state of the model.
- **Reverse engineering** takes an existing relational structure and creates an object orientated front-end to be used by middleware.

In the case of legacy databases we will mostly be confronted with reverse engineering. It will be very hard to alter the structure of the current databases without

[3] Employer Dialogue System.

breaking the current access programs. The business logic needs to be checked and rewritten where necessary. There is also the problem of synchronization between objects and relational structures: Changes made to the relational database should be reflected in the various distributed objects and vice versa. In the case of concurrent access to objects this leads to several concurrent conflict resolution strategies.

1.1.1 Database access in a Java environment

In this thesis we will be applying the most basic form of objectmapping strategy. Each row of data will be represented by one object with one field for each column inside the row. This means that even for the most basic wrapping technique we need to know the structure of the databases. For this purpose most databases store information about the layout and structure of the data they are storing. This metadata holds information about the number of rows and columns, the type and precision of the columns. To access the metadata one has to write native calls to vendor specific database libraries. This leads to problems if one wishes to switch platforms or change from a database engine. Luckily with the advent of Java Database Connectivity (JDBC) an initial vendor lock-in can be prevented. We rely on the metadata discovery capabilities of JDBC, and to encourage re-use we will encapsulate these metadata reflection methods inside easy to use objects. This sounds very good but the use of JDBC is not all sweet-and-honey. Currently JDBC has a minimal feature set, no support for bi-directional cursors, no support for bookmarks, and the collection that holds the database records is meant to be read-only, thereby hindering the easy synchronization between database and objects. Since the metadata is also returned in these read-only collections we had to roll our own updateable collections. The design and implementation of these objects will be examined in chapter 2

For privacy and security reasons the design and testing of metaobjects wasn't carried out on the actual GAK databases. We had to create and populate our own test databases. Doing this several times by hand soon become a bore, until the realisation was made that this work could easily be carried out by some kind of script language. The search for a Java embeddable script language led to the evaluation of several script languages. In the end we settled for JPython. JPython is easily embeddable in Java and by means of introspection we can now test Java objects in an interpreted environment. This way we should benefit from a high level scripting language and short turnaround times.

1.1.2 CORBA in a Java environment

CORBA is a form of middleware. It enables the access of programs across networks. CORBA is cross platform because every interface is first defined in Interface Definition Language (IDL) and then compiled to the platform. Many services have already been defined like the Naming Service, Transaction Service so it seems to offer almost the same features as OpenGAC. Yet OpenGAC is primarly message based with an loosely coupled publish/subscribe architecture, while CORBA is more based on distributed objects.

One of the first obstacles was how to transport the records between client and server. We could have used a simple socket connection or relied on JDBC for the transport but then we would not have had the advantages of a CORBA like platform and language independence. Also by opting for a three tier solution, we hope to have the advantages of scalability. The one big nuisance of the current CORBA

version is its lack of pass by value. Objects are not expected to travel from the server to the client and vice versa, only a reference is exchanged. This way it is not easy to transmit a whole collection of database records to the client to be cached and later to be updated at the server side. Even if you pass by value, you still have to establish some sort of conflict resolution. If two clients update the same record simultaneously which copy takes precendence?

But also on the server side there is the problem of synchronization: how are records that are updated on the backend synchronized with the server objects? Current JDBC lacks the appropriate collections and most CORBA implementations lack notification support. Since we only have to pass state and no behaviour we can overcome the lack of pass by value by the clever use of so called structs. The already developed metaobjects allow us to generate these structs for us.

1.1.3 Remote Method Invocation and Java

Now that we have taken a look at CORBA in a Java environment, we can introduce yet another middleware solution in the Java arena. RMI is brought forth as the middleware solution that runs best on only one platform, the Java platform. But since Java runs on many platforms, RMI is also said to be cross platform. The reliance on Java has its strengths and weaknesses . The strength is that in contrast with CORBA, one need not to learn a new language like CORBA IDL, since the implementation of distributed RMI objects can be done in Java. RMI objects are Java objects and therefore can be dynamically loaded, serialized as byte codes, and transported to other Java virtual machines. In this way one can circumvent CORBA's biggest shortcomming: the lack of pass by value. But all this power comes at a price, the serialization process is notably slow. Also the acceptance of RMI is still behind the acceptation of CORBA, as can be seen in the lack of firewall support. Chapter 5 examines RMI in detail.

1.2 Research question

The purpose of this thesis is to investigate the possibilities to use Java based middleware for the unlocking of legacy databases.

1.3 Chapters

This thesis is built upon the following sections. First we will introduce in chapter 2 how databases can be accessed with the use of Java Database Connectivity. Scripting inside a Java environment is discussed in chapter 3. This will enable scripting of the objects introduced in the previous chapter. Then, a description of the use of CORBA inside a Java environment is given in chapter 4, including the changes and future developments CORBA will encounter. Another closely related middleware solution called RMI is the focus for chapter 5. The experiences of the above chapter is amalgated in chapter 6 about the prototype. And finally, we give conclusions with chapter 7 and our parting remarks. Our overall approach is sketched in figure 1.2. We will explain it in the chapters to come.

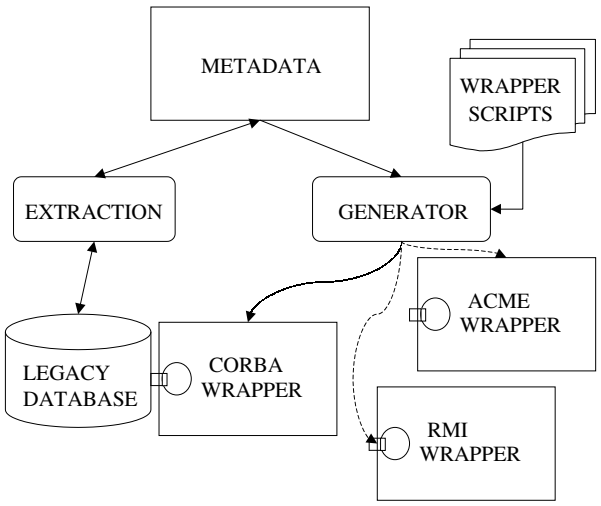


Figure 1.2: Generation Overview

Chapter 2

JDBC

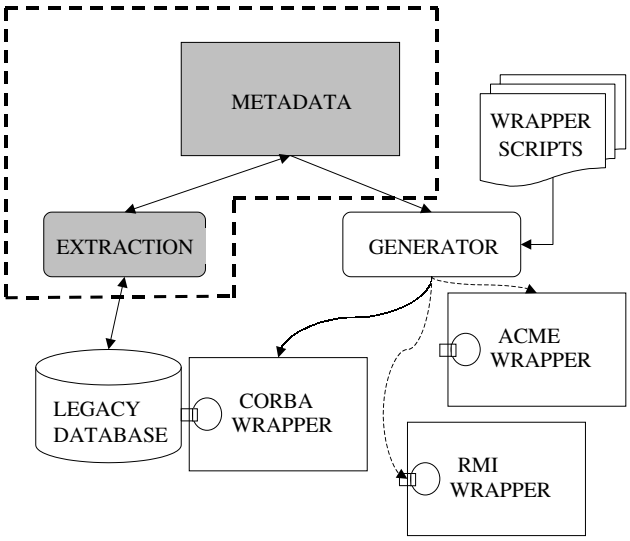


Figure 2.1: Metadata discovery with JDBC

2.1 Databases and Metadata

In this chapter we will focus on the importance of discovering metadata. Metadata describes the way data is stored inside a database. The primary goal of databases is to store and retrieve user data. But to do so the database itself has to administer all kinds of relevant data e.g. which tables are being used, which names are being used to indentify the fields, what is the size and type of data that is store inside these fields. These so called metadata can be brought to good use in end user applications where the exact need of table data is not known in advance. Examples of such applications are Report Writers, Database Administration tools and Datamining tools. Most of these metadata materializes during the creation of the database through the use of a Data Definition Language (DDL). Most database implementations have standardised on SQL DDL. But although the language of SQL DDL is standardised the way DDL is executed can differ from one vendor to another. Each database vendor supplies his own native implementation of database access. By developing with a native database access API, the developer is quickly trapped inside a vendor lock-in, making it more difficult to switch between different database implementations. The creators of Java have recognised this issue, and came up with the Java Database Connectivity API (JDBC), which should alleviate these problems.

2.2 Prerequisites

This chapter assumes that the reader is familiar with database concepts like Databases, Tables, Primary Keys, Foreign Keys, Indexes and Columns. Some basic understanding of the Data Manipulation Language (DML) and Data Definition Language as used in SQL is also assumed. DML manipulates Data inside Databases with the `SELECT`, `INSERT`, `DELETE` and `UPDATE` statements. DDL enables the creation of database objects like tables, indexes and stored procedures. For a closer look at DML and DDL SQL see [2].

Most database systems provide native proprietary library calls to make connection with the database. So besides the different set of library calls for each database, the code is also dependent on each platform of the application and database server. We see a strong shift from 2-tier client/server computing to 3-tier internet computing. The majority of database applications use relational database technology. For this reason the JDBC API is also very tailored towards relational databases and the use of the standard query language SQL. The relational model is the most widely used data model. Especially for developers of information systems, using traditional 3GL languages like C and Pascal, there is this need for a standardized interface to relational databases. This has resulted in the following standards:

- X/Open Call Level Interface (CLI)
- Microsoft Open Database Connectivity (ODBC)

2.3 Enterprise computing and Java

Mission critical systems almost always include some sort of database access. The philosophy behind Java Enterprise computing is supplying enough API support to make database connectivity and distributed computing easier. For database connectivity the Java platform supports the JDBC API and for distributed computing you can rely on either RMI (Remote Method Invocation) for use in a homogeneous

Java environment or CORBA for use in a heterogeneous environment.
So with JDBC you can use the same Java program to run against any database that supports the JDBC API. With RMI and CORBA you can write programs that use distributed object technology without facing the challenge of low-level network socket programming.

2.4 JDBC API Overview

The Java Database Connectivity (JDBC) is an API for the connectivity of datasources with the help of SQL. Although this Structured Query Language has been adopted by the ANSI committee, this has not prevented database suppliers introducing their own SQL dialects. Although on the one hand, these extensions of standard SQL can profit from specific database properties, on the other hand, they lock the developer inside vendor specific proprietary languages. This lock-in prevents the change of database vendor without checking and rewriting the current SQL statements. To shield developers from these vendor specific extensions, this has led to adapters/bridges that translate generic database calls to specific database calls. One of the most known bridges is the Open DataBase Connectivity specification. ODBC is a C-based standard developed by the X/Open Data Management Group, and popularized by the Microsoft Windows operating system. Looking at ODBC and JDBC, there are some striking similarities, but overall JDBC has been designed on a more object orientated basis

The JDBC API is a set of interfaces for database access. The 5 most important

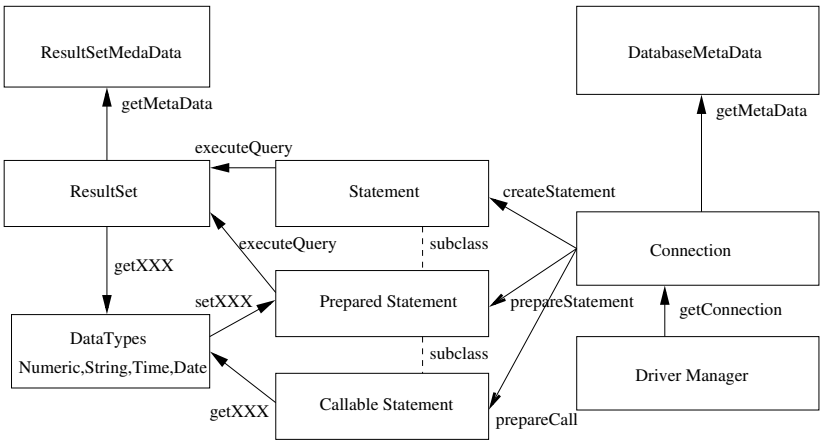


Figure 2.2: JDBC classes

interfaces are:

- **Connection**: The Connection interface enables communication with the database system and has methods to set and get various database information (eg types of transactions).

- **Statement**: The methods of the Statement interface enable the executions of various DML and DDL SQL commands.

- **ResultSet**: The ResultSet acts as a kind of Container Class to hold the results from SQL queries.
- **ResultSetMetaData**: This interface enables the the interrogation of ResultSets, so the properties of dynamic data can be discovered.
- **DatabaseMetaData**: This interface enables the querying of the database properties like tables, views and stored procedure.

In figure 2.2 we show the relationship between these classes.

2.4.1 JDBC Example

JDBC is designed to be platform-independent. The vendor-specific database calls are all encapsulated in the JDBC Driver interface. Currently there are 4 types of JDBC Drivers

- Type 1 Bridge Driver: Driver in native code that connects Java clients to non-Java database services (eg ODBC).
- Type 2 Partial Java Driver: Java wrapped native code library.
- Type 3 All-Java Network Driver: Driver that connects through database middleware.
- Type 4 All-Java Native Driver: a pure JDBC implementation to connect to databases directly.

Before we can access the database data with JDBC we have to register the appropriate driver, for this purpose we use dynamic class loading by name. For our example database we use MySQL, the type 4 database driver is called `org.gjt.mm.mysql.Driver`

```
String driver = "org.gjt.mm.mysql.Driver";
try {
   Class.forName(driver).newInstance();
} catch (Exception e) {
   System.out.println("failed loading driver "+driver+"\n"+e);
   return;
}
```

After succesful loading of the database driver, we can try to get a connection, prepare a statement, execute it, and process the returned data. The location of a database is specified through a Uniform Resource Locator (URL). This URL is of the form `<protocol>:<subprotocol>:<name>`

```
Connection con = DriverManager.getConnection("jdbc:mysql:wds","system",password");
Statement  stmt = con.createStatement();
ResultSet rset = stmt.executeQuery("select * from werkgevers");
while (rset.next()) {
   String naam = rset.getString(1);
   String plaats = rset.getString(4);
   System.out.println("Naam:" + naam + " Plaats:" + plaats);
}
```

In the example above we created a Statement object throught the use of `createStatement` method of the Connection class. Besides the Statement class there are two similar classes:

- `PreparedStatement` for repeated execution of a compiled statement.
- `CallableStatement` for stored procedures taking advantage of database system optimizations.

We can also distinguish two types of statements:

- Update statements `executeUpdate(String sql)` SQL statements that update the database, such as INSERT, UPDATE and DELETE.
- Query statements `executeQuery(String sql)` SQL statements of the form SELECT.

Query statements return a ResultSet through which database rows can be accessed. Update statements return the number of rows that were being updated.

From this small example we can see the layered approach in JDBC programming: instantiate the appropriate driver, get a connection, create a statement, execute the statement and process the result. In the example we knew in advance which fields to get and of which type the fields were. But this knowledge about the current structure of the database is not alway readily available. In those cases we need to rely on the metadata that the database offers. Specific forms of metadata can be divided in several groups

- Data Types: Strings, Numbers, Binary Large Objects
- Container Types: Physical Containers (Drives,Files), Logical Containers (Databases,Catalogs)
- Vendor specific Data: ANSI SQL compliance, Delimiter

In the case that we are more interested in names of the database fields or tables, we need to create an instance of DatabaseMetaData from the connection. Same thing holds for information about the column names of the ResultSet, in that case we need to instantiate the ResultSetMetaData class.

2.4.2 ResultSetMetaData

`ResultSetMetaData` provides extra information about `ResultSet` objects that are returned by a database query. The ResultSetMetaData class can answer the following questions:

- How many columns are in the resultset?
- What is the name of a specific column?
- What is the type of a specific column?
- From which table does a specific column originate?
- Are the column names case sensitive?

This kind of information is normally hard coded in queries. The amount of ResultSetMetaData extracted can vary, so the result information is put in yet another ResultSet.

2.4.3 DatabaseMetaData

In the previous paragraph we saw how the ResultMetaData is closely related to the ResultSet. The `DatabaseMetaData` is closely linked to the `Connection` class. With the DatabaseMetaData class you can investigate things like:

- What tables belong to which database?
- What are the primary keys for a specific table?
- To which degree of SQL compliance does the database conform?

The DatabaseMetaData supports over 130 class methods. For a closer examination see [1]

2.4.4 Turning ResultSets into Collections

Most of the methods that return complex data, do so as a ResultSet. Since databases are capable of holding thousands of records, the implementation of the ResultSet doesn't actually store the data inside the set. Instead a reference (Cursor) to the current record inside the set is obtained. The JDBC 1.0 API doesn't support scrollable cursorsets, so the cursor can only move forward. It also lacks updates to the resultset. The ResultSet isn't a real container class. To remove these shortcomings we need to store the columndata of a single row of sqldata inside a data wrapper object. This can be done dynamically by introsprection of the resultset, or statically where a resultset is inspected beforehand and source code is generated that wraps the data in the underlying resultset. The ResultSetMetaData interface is a likely candidate for the introspection of the resultset. This way we can navigate through the collection and update the record inside the object wrapper. Using this scheme we can finally turn metadata, which is also represented by ResultSets, into collections, and generate sourcecode on the basis of the discovered meta data.

2.4.5 Façade design pattern for JDBC

In the previous paragraphs we have given a overview of the 5 most important interfaces of the JDBC API: *Connection, Statement, ResultSet, ResultSetMetaData, DatabaseMetaData*. For a simple query we need to layer at least 3 of them. We can shield the layman programmer from the hairy details by introducing a façade for JDBC (Class Diagram in Figure 2.3). The Façade Design Pattern makes it easier to use the JDBC API, by hiding the complex subsystems through a simplified interface. A Design pattern is a high level design, solving a particular class of problems. It brings a general flexible solution for object oriented problems that are born out of best practices. For a broader overview of design pattern we reference to *Design Patterns* by Gamma, Helm, Johnson & Vlissides [4]. For the Software metrics of the Façade components see Appendix A.

```
rwDatabase  rwdb = new rwDatabase("org.gjt.mm.mysql.Driver");
rwdb.Open("jdbc:mysql:///","test");
rwResultSet rwrs = rwdb.execSql("select * from ducks");

while (rwrs.moreRows()) {
   System.out.println(rwrs.Column(1)+" "+rwrs.Column(2)+" "+rwrs.Column(3));
}
```

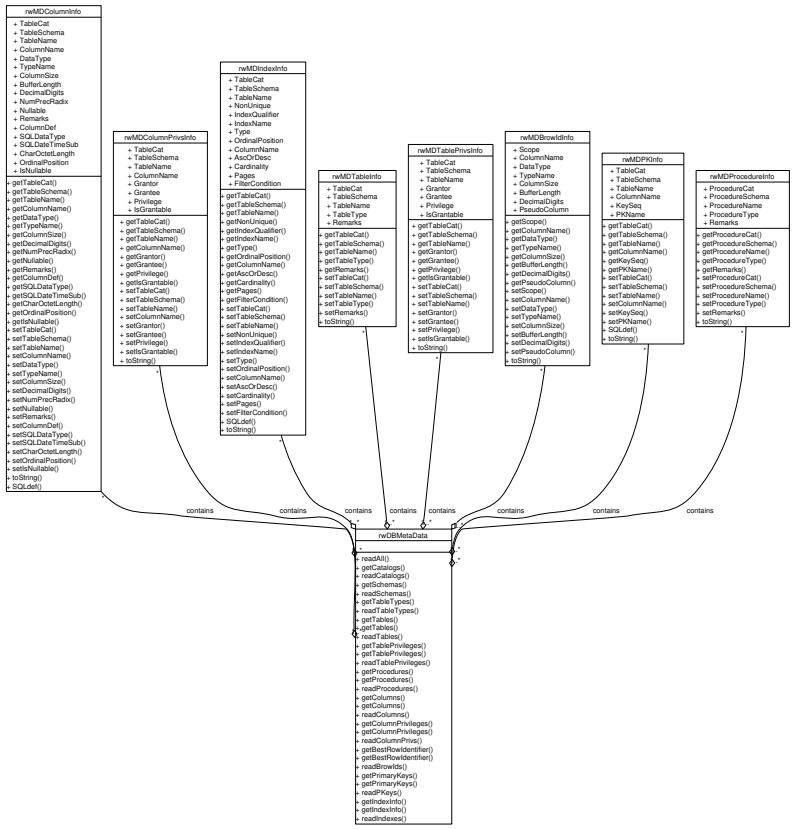


Figure 2.3: Class diagram of the Façade components (see Appendix A)

2.4.6 Object/Relational Mapping

The challenge at hand is to match software objects to database tables. Wrapping a single row of SQL data into a data wrapper is a simple form of Object/Relational mapping. One can use the template (example 2.4) as a basis of a Java Data Wrapper. The template can be generated by having all the metadata collected and printing out the collection in the appropriate language construct. However smarter schemes exist. For an overview see [7]. Most of these techniques are focused on adapting objects onto relational data, which is often the case for newly developed object oriented applications that are not burdened by legacy support. In our case we cannot simply alter the underlying database model, so we have to adapt the object interface.

We have implemented a static resultset introspection object into our database facade. We can use this introspection object to generate sourcecode for different kind of wrappers:

```
rwDatabase rwdb = new rwDatabase("org.gjt.mm.mysql.Driver");
rwdb.Open("jdbc:mysql:///","test");
rwResultSet rset = rwdb.execSql("select * from ducks");
rwResultSetMetaData rwrsmd = new rwResultSetMetaData(rset);
System.out.println(rwrsmd.columnCount);
for (int i=0;i<rwrsmd.columnCount;i++) {
    System.out.println(i+":"+rwrsmd.cmd[i]);
}
```

```
class <tablename>DataWrapper
{
      <type1>  dw<fieldname1>;
      <type2>  dw<fieldname2>;
      ...
      <typen>  dw<fieldnamen>;

  public <tablename>DataWrapper(ResultSet res)
  {
     try {
        dw<fieldname1> = res.get<type1>("fieldname1");
        dw<fieldname2> = res.get<type2>("fieldname2");
        ..
        dw<fieldnamen> = res.get<typen>("fieldnamen");
     } catch (Exception e) {
        System.err.println(e);
     }
  }
}
```

Figure 2.4: template

To wrap JDBC sources for other purposes, for example SQL, RMI, CORBA, we need to test/write different Java generator classes. A more flexible approach is to incorporate scripting, which is the topic of the next chapter.

We can see from this example, that there is no standard support for the caching of rows. In JDBC 1.0, there are no cursorsets. This ommision has been partially relieved in JDBC 2.0, yet the availability of JDBC 2.0 compliant drivers, that make the added features possible, is still low. A recent addition from Sun is the early access of JDBC CachedRowSet. The CachedRowSet can send a set of rows across a network to a thin client, and also features scrollability and updatebility of the resultset data. Another possibility is to send the data as a XML stream. To complete the picture Sun has also released the draft for JDBC 3.0, another reason for the author to consider the whole JDBC API not well thought out from the beginning.

Chapter 3

Scripting inside Java

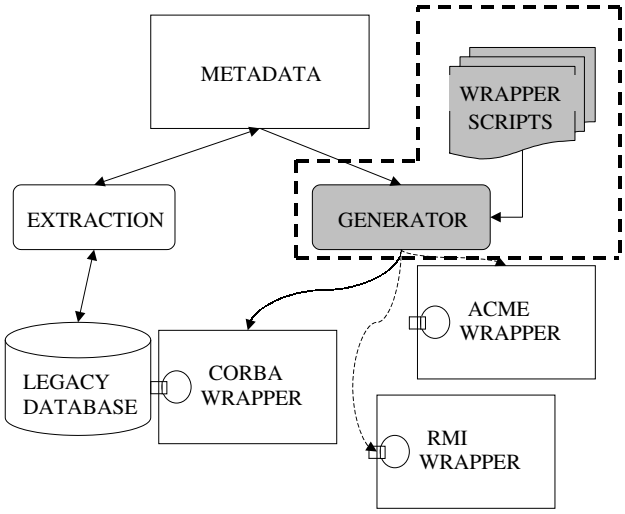


Figure 3.1: Generating Wrappers with Scripting

3.1 Introduction

Java is considered as the jack-of-all-trades language. More and more API's are being added, for example, JNDI[1], JMS[2], JTS[3] and JTAPI[4]. The various API's make it a great system programming language. But at times Java is still too low level. It lacks the ability to glue together Objects into a framework application. Other languages are better equiped for these tasks. Most of these languages are dynamic interpreted high level scripting languages. Fortunately these languages can be adapted to run on the JVM. Scripting languages still have the odour of *quick-and-dirty* programming, but with the increased use of object frameworks and component based development, scripting languages are more and more accepted as the mortar between the component bricks.

Scripting languages for Java can play different roles:

- Extension language: Use the scripting language to allow configuration and customization of the host application.
- Debugging tool: Use the scripting language as a shell to test applications interactively.
- Java API explorer: experiment with Java classes, by instantiating them on-the-fly.

3.2 Bonding System and Scripting Languages

The use of a system language combined with a scripting language may be something new on the Java platform, but it has been applied succesfully in the past on other platforms. On UNIX systems the combination of shell scripts and C programs is very common. In particular the arrival of TCL has boosted GUI programming through scripts that interface with C/C++ objects. Other control languages like Python have also contributed to a growing acceptance of scripting languages. And on the Microsoft Windows platform, Visual Basic is the preferred language to glue together COM objects written in languages as C++ or Object Pascal. The lack of shell scripts on windows platform has even led to the introduction of the Windows Scripting Host, a generic framework to manipulate COM objects with your favourite scripting language. Writing wrappers to connect C and C++ programs to high-level scripting languages can be tedious, so a tool like the Simplified Wrapper and Interface Generator (SWIG) has been developed to interface with Perl, Python, Tcl/Tk and Java.

3.3 Overview of JVM languages

During the development of the database wrapper generators, we have evaluated several scripting languages that run on top of the JVM.

- JPython: JPython is a Java implementation of the dynamic object oriented Python language. Starting from version 2.0 JPython has been renamed Jython.
- Jacl/TclBlend: Java Command Language (JACL) is a Java implementation of TCL 8.0+. TclBlend introduces TCL commands, so you can manipulate

[1] Java Naming and Directory Interface
[2] Java Message Services
[3] Java Transaction Service
[4] Java Telephony API

Java objects without writing any line of Java code. As with the other scripting languages, through the use of reflection, one can invoke methods and access the properties of arbitrary Java objects.

- Pnuts: Pnuts is a recent addition to the range of Java scripting languages. It was developed by Toyokazu Tomatsu of Sun Japan. It is an expression based scripting language and supports pseudo lisp lists and first class object functions.

Most of those scripting languages are object-oriented or object-based. They are interpreted by the JVM and are very dynamic through the use of late-bound polymorphism. Most of these languages use dynamic typing. There is an internet repository [8] that features a much larger list of programming languages for the Java virtual machine aside of Java itself, currently there are over 20 scripting languages for JVM listed.

3.4 Choosing a Scripting Language

There are two options for increasing productivity with scripting languages.

- Roll Your Own: You can develop your own domain specific language. This will allow you to have complete freedom on the syntax of the language. The Java platform now has mature compiler generator tools to help you with the lexical and syntax analysis [ANTLR,JAVACC,Coco/R], but the question remains if the time spend on developing and debugging the new language outweighs the promised productivity gain.
- Reuse: As already mentioned above there are already many implementations of scripting languages on offer.

We used the following criteria to select JPython as our preferred scripting language:

- Java Class Integration: How well is the integration of Java Classes and APIs, can Java classes be subclassed in the script language. How easy can variables be mixed.
- Familiarity: How familiar is the scripting language. How closely does it resemble a known language.
- Embeddability: How easily can the scripting language be embedded in a Java application. How well does the language integrate with Java.
- Library support: Some languages come with a wealth of libraries that offer enhanced string support and various types of Datastructures that can help generating source code.

3.5 Comparision Java and JPython scripts

For a comparison of Java and JPython we will write a script that queries our test database. Compared to the corresponding Java version (Figure 3.3) we see that the Python version (figure 3.4) is much shorter. Much of this code reduction can be attributed to implicit exception handling in Python. But how far does this code reduction affect execution speed? We will measure the time spent on JDBC initialisation (time 1) and the time used for printing the resultset (time 2). The results of 5 runs are given in figure 3.2. It looks like Java spends longer in the JDBC initialisation phase, which is hard to believe because JPython calls the same Java

JDBC methods. Printing in JPython is notably slower,but this is possible related to delayed JDBC calls inside the printing loop. The combined times of JPython are on average only a 6 percent longer than the Java version.

Java			JPython		
time 1	time 2	total	time 1	time 2	total
2259	372	2631	2019	747	2766
2245	299	2544	1993	775	2768
2217	261	2478	1929	758	2687
2234	552	2786	1912	746	2658
2231	285	2516	2029	839	2868
		12955			13747

Figure 3.2: Timings timedb.Java and timedb.py

```
import Java.sql.*;
import Java.util.Date;

public class timedb {
  public static void main(String args[]) {
    ResultSet rset;
    Date d1,d2,d3;
    d3 = new Date(); d2 = new Date(); d1 = new Date();

    try {
      Class.forName("org.gjt.mm.mysql.Driver").newInstance();
    } catch (Exception e) {
      System.out.println("failed loading driver "+e);
      return;
    }
    try {
      Connection con = DriverManager.getConnection("jdbc:mysql:///bpr","","");
      try {
        Statement st = con.createStatement();
        rset = st.executeQuery("select * from personen");
        d2 = new Date();
        while (rset.next()) {
          System.out.println(rset.getString(1)+" "+rset.getString(2)+" "+rset.getString(3));
        }
      } catch (SQLException e) {
        e.printStackTrace();
      }
      d3 = new Date();
    } catch (Exception e) {
      e.printStackTrace();
    }
    System.out.println(d2.getTime()-d1.getTime());
    System.out.println(d3.getTime()-d2.getTime());
  }
}
```

Figure 3.3: Listing timedb.java (Java)

```
from Java.sql import *
from Java.util import Date
from Java.lang import *
d1 = Date()
driver = Class.forName("org.gjt.mm.mysql.Driver")
con = DriverManager.getConnection("jdbc:mysql:///bpr","","")
stat = con.createStatement()
rset = stat.executeQuery("select * from personen")
d2 = Date()
while rset.next()>0:
   print rset.getString(1),rset.getString(2),rset.getString(3)
d3 = Date()
print d2.getTime() - d1.getTime()
print d3.getTime() - d2.getTime()
```

Figure 3.4: Listing timedb.py (Python)

Chapter 4

CORBA

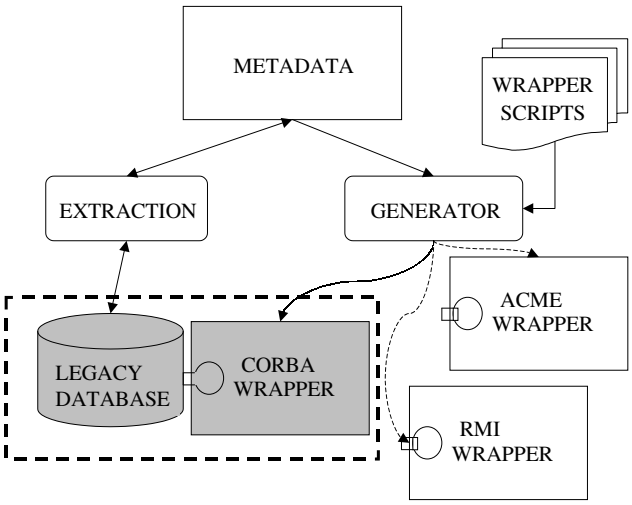


Figure 4.1: CORBA Database Wrapper

4.1 Introduction

CORBA is an acronym that stands for Common Object Request Broker Architecture. This architecture has been specified by the Object Management Group, a consortium of hardware and software producers, whose goal is to provide a component based development platform through the introduction of standardized object technology. By providing guidelines and specifications, a framework for heterogenous distributed objects is made possible. This framework- like architecture is called the Object Management Architecture. Currently the OMA is composed of several parts.

4.2 Object Management Architecture

The OMA reference model comprises the following 4 parts:

- Object Request Broker (ORB), the communication layer, also known as middleware, that realizes the communication between objects. To secure the interoperability between objects regardless of their location or development language, the interfaces of the objects are specified in the language neutral Interface Definition Language.
- Object Services, components that manage the life span of objects. Through standardized interfaces the programmer gets support for the creation, authentication, versioning of objects and other management tasks.
- Common Facilities, generic application-wide functionality like printing and mail services.
- Domain Interfaces, out of the box solutions for vertical markets.

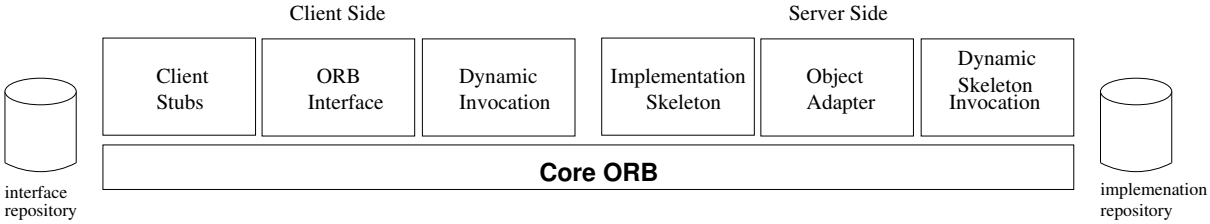


Figure 4.2: ORB Architecture

4.2.1 CORBA ORB

In this section we will primarily deal with the CORBA ORB, and we will expand on the development of CORBA in a Java setting. We will also cover the IDL to Java mapping, that is necessary to develop CORBA applications within Java.

There are many examples of specifications [ASF+SDF][PSF], yet the production of a complete specification is a far from trivial task. So, it comes as no surprise that even CORBA spots a large list of revisions. Version 1 of CORBA was first devised in 1990, augmented to version 1.2 in 1993. Then thoroughly revised to version 2.0 in 1995. The current common version stands at version 2.3. Version 3.0, with the eagerly awaited pass by value mechanism, is slated for the end of the year 2000.

The increase from version 2.x to version 3.x is more difficult than expected. To this date (june 2002) there have been no complete implementations of CORBA 3.0.

CORBA is a distributed object technology, that makes it possible to activate objects remotely, independent of the development language, the kind of operating system or computer platform.

The CORBA 1.1 specification defines the basic functionality of the Object Request Broker (ORB) and the Interface Definition Language (IDL), in which the interface that an object supports is published. The role of an Object Bus, a channel on which objects are exchanged, is played by the ORB. The ORB facilitates the retrieval of the location for objects, and the local and remote invocations of the published interfaces. The ORB is also responsible for the transmission of requests and the interception of invocations and exceptions between client and server objects. The ORB renders the programmer unaware of details like object location, chosen language of implementation and target platform. Before the management of these objects can be carried out by the ORB, one first has to register the objects with the ORB repository.

Through the use of object technology we are more able to describe and implement business processes. Also by enscapsulating methods and data, the reuse of these objects is eased. As a matter of fact the interface of an object defines the code of conduct, a programming contract of the services that the object offers. This contract is defined in the forementioned IDL. Through the means of a IDL compiler the IDL specification is translated into proxies for the client and server parts. The client proxy is better known as a stub, whilst the server proxy goes by the name of skeleton. The server functionality can be discovered by the client, either static, through the compiler generated proxy, or dynamic, by introspection of the repository through the Dynamic Invocation Interface.

4.2.2 A small CORBA example

Before we can develop a CORBA application, we need an implementation of an object request broker. Fortunately since the release of Java 2, every Java programmer now has access to a reference implementation of a light weight ORB[1]. This gives us a good opportunity to put the Java/IDL mapping to the test. For this small example, we will develop a distributed prime number tester. We can ask the prime number tester whether a number is prime or not, we can also ask it to supply a list of all the primes in a particular range. Figure 4.3 illustrates the UML object diagram we have come up with for the Primality Tester. The IDL extracted for this object diagram looks like this:

```
module PrimeTest
{
   typedef sequence<long> primeList;

   interface PrimeTester
   {
      boolean isPrime(in long number);
      boolean isProbablePrime(in long number,in long tests);
      primeList listPrimes(in long min, in long max);
   };
};
```

[1]The Java ORB is lightweight because the Corba objects are not persistent, the ORB is a transient server that doesn't support an Interface Repository.

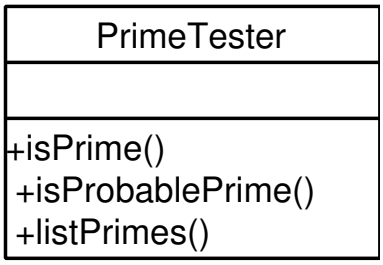


Figure 4.3: Simple UML Diagram for Primality Tester

Now we have to use an IDL compiler to translate this IDL file to the target language, in our case we will be using Java. The IDL compiler generates client stubs and server skeletons, that are to be augmented before we can use the client and server. This is achieved by:

```
idltojava PrimeTest
```

This will generate several files in the directory of the corresponding module. Although CORBA is now part of the JDK1.2 Enterprise edition, the idltojava compiler is still a seperate download. With the introduction of the JDK1.4, the Java IDL compiler is now completely written in Java, and the reference implementation of the CORBA ORB has been brought more up to date.

4.2.3 Implementing the Client (`PrimeClient.java`)

Before we can use the remote PrimeServer as a seemingly local object, we first have to import the module and do some general management tasks like asking the naming server to supply us with a reference to our target object. These lines that manage the naming context and resolve the various object references can be copied verbatim. After narrowing down and casting, we can call remote methods on the local `primeRef` object.

```
import PrimeTest.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class PrimeClient
{
   public static void main(String args[])
   {
      try {
         // create and initialize the ORB
         ORB orb = ORB.init(args, null);

         // get the root naming context
         org.omg.CORBA.Object objRef =
             orb.resolve_initial_references("NameService");
         NamingContext ncRef = NamingContextHelper.narrow(objRef);

         // resolve the Object Reference in Naming
         NameComponent nc = new NameComponent("PrimeServer", "");
         NameComponent path[] = {nc};
         Prime primeRef = PrimeHelper.narrow(ncRef.resolve(path));

         // call the Prime Tester server object and print results
```

```
         int number = 3;
         boolean isprime = primeRef.isPrime(number);
         System.out.print(number + " ");
         System.out.print( isprime ? "is" : "is not" );
         System.out.println( " a prime");

      } catch (Exception e) {
         System.out.println("ERROR : " + e) ;
         e.printStackTrace(System.out);
      }
   }
}
```

4.2.4 Implementing the Server (PrimeServer.java)

The following template handles the management of the server; the implementation of the interface is delegated to PrimeServant Object.

```
public class PrimeServer {

    public static void main(String args[])
    {
        try {
            // create and initialize the ORB
            ORB orb = ORB.init(args, null);

            // create servant and register it with the ORB
            PrimeServant primeRef = new PrimeServant();
            orb.connect(primeRef);

            // get the root naming context
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);

            // bind the Object Reference in Naming
            NameComponent nc = new NameComponent("PrimeServer", "");
            NameComponent path[] = {nc};
            ncRef.rebind(path, primeRef);

            // wait for invocations from clients
            java.lang.Object sync = new java.lang.Object();
            synchronized (sync) {
                sync.wait();
            }

        } catch (Exception e) {
            System.err.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
    }
}
```

Through the use of CORBA, a client can invoke methods on remote server objects regardless of the location of the server objects. Since the location is transparent, a client has to retrieve a reference to the interaction object. This process is called *binding*. This reference is called Interoperable Object Reference or IOR for short. The CORBA 2.1 specification finally has defined a string representation for the IOR, it consists of the prefix IOR: followed by the hexadecimal representation of the IOR in CDR format. This way we can exchange object references.

Generally there are two methods to get a stringified IOR:

- Ask the CORBA Naming Service, or CORBA Trading Service for the desired reference.

- Convert an object reference to a string and store it a designated place, where a client can read it in.

The IOR is commonly shown upon startup of the Naming Service:

```
tnameserv -ORBInitialPort 1234 &

Initial Naming Context:
IOR:000000000000002849444c3a6f6f672e6f72672f436f734e616d696e672f4e616d696e6
7436f6e746578743a312e300000000000000000000000002c000100000000000006323930353
600053d00000018afabcafe0000000225b48bbf0000000800000000000000

TransientNameServer: setting port for initial object references to:1234
```

4.3 CORBA lacks Pass-By-Value

The design of CORBA is primarly client/server based. Objects reside on the server and clients only need a reference to those objects to invoke methods. That is why objects are passed *by-reference*. Only with the advent of CORBA 3.0 the partial passing *by-value* will be made possible. CORBA Vendors have just started to implement Objects-By-Value in their CORBA 2.3 products. We will not see implementations of the new features of CORBA 3.0: Pass-By-Value, CORBA Components before 2003. The current IDL simply lacks expressive power, and implementations of desired services, like Collection Service, or Persistence Service are also hard to come by. This is the main reason we have to come up with solutions for the lack of pass-by-value and the transport of collections.

Let's say we want to give access to a database and its records in a CORBA environment. There are basically two options:

- Clients only reference database record objects on the server, so for each access there is a single request. This has the benefit that the server can control concurrent access to records, and employ a simple update conflict resolution. [2]

- Clients request the database record object to be transferred to the client in order to minimize the communication with the server. Although the server will be able to handle more clients, concurrent update issues are more difficult to resolve.

To give some examples of types of record tranportation, we will introduce the following adres database with its SQL DML construct.

```
CREATE TABLE adres (
   adresnr  int(10)    NOT NULL,
   naam     varchar(25) NOT NULL,
   straat   varchar(30) NOT NULL,
   postcode varchar(8)  NOT NULL,
   PRIMARY KEY (adresnr)
);
```

The above record definition can be modeled in IDL in several manners:

- As an IDL sequence of IDL structures.

[2] There is still the issue of synchronization of the Corba database objects with the native database records, the Corba server should be notified if records are updated by the native database.

- As an IDL tuple function.
- As an collection of IDL attributes.
- As an IDL sequence of serialized Java object bytes.

We now discuss each of these approaches in more detail

4.3.1 IDL struct

A row of database data can be mapped onto a structure. A collection of records can be mapped onto an IDL sequence. This way we have a straight forward mapping that can transport records from the CORBA server to the client.

```
module AddressData
{
   struct AddressData
   {
      long iadresnr;
      string inaam;
      string istraat;
      string ipostcode;
   };
   typedef sequence <AddressData> AddressSeq;

   interface DataMediator
   {
      AddressSeq getAddresses();
      void       setAddresses(out AddressSeq);
   }
}
```

By using getAdresses we will get a list of database records from the CORBA server and by using setAdresses we can update our collection of client-side cached address records to the CORBA server.

4.3.2 IDL tuple function

In the database literature a row of database data is often characterised as an tuple function. We can map database tuples onto tuple functions, in which each data base field is represented by a function argument. This way records can be transferred by calling the tuple functions.

```
// Database -> IDL tuple function

module AdresData
{
   interface AdresData
   {
      getAdres ( out long iadresnr,
                 out string  inaam,
                 out string  istraat,
                 out string  ipostcode
               );

      setAdres ( in  long iadresnr,
                 in  string  inaam,
                 in  string  istraat,
                 in  string  ipostcode
               );
   }
}
```

4.3.3 Collection of IDL attributes

An IDL interface can have attributes. An attribute is equivalent to declaring a pair of accessor functions: one to retrieve the value of the attribute and one to set the value of the attribute. So by assigning an attribute, the corresponding database fields gets updated. Since each assignment leads to a Corba network call, this approach tends to be rather slow.

```
// Database -> IDL with attributes
module AdresData
{
   attribute long   iadresnr;
   attribute string inaam;
   attribute string istraat;
   attribute string ipostcode;
}
```

4.3.4 Serialized Java objects

Most of our CORBA experiments were carried out in a Java environment, Java has support for streaming of objects, in an all-Java environment, objects can be streamed and transported over CORBA. Although Java normally uses RMI for marshalling and transporting of Java objects over the network, it is nice to know that we can transfer arbitrary binary streams over CORBA.

```
module MiscData
{
   typedef sequence <octet> ByteSeq;

   interface DataMediator
   {
      ByteSeq getByteData();
      void    setByteData(out ByteSeq);
   }
}
```

4.3.5 Benchmarking CORBA Wrappers

In order to test the different approaches we will generate the various CORBA wrappers with JPython and our database façade. The wrappers will be compiled with the Java IDL compiler and we will use benchmarks to select our favourite mapping.

We used a python script (see listing 4.11) to generate SQL statements that insert a total of 1500 records of random address data. We used distinct postal codes in order to select multiples of 100 records. On the client side we recorded the times needed to resolve the Server (t1), the time to relay the SQL to Server (t2), the time to receive the results (t3) and the time to print to results (t4). At the server side we measured the execution time of the query (t5) and the time needed for wrapping all records onto a CORBA container (t6). The client requested the timings that were recorded on the Server. We started benchmarking with an initial request for 100 records, then we repeated the request with increments of 100 records. The final benchmarks in each testrun requested a 1000 records. Each benchmark was carried out 10 times. Most of the time was spent in initializing the CORBA ORB. The startup times of the ORB are around 370 ms, the startup times of the Java environment are much longer, so in each testrun we conducted 11 timings and ommitted the (first) slowest timing. We relay the SQL query to the Server, the server executes the SQL Query, wraps the resultset onto CORBA construct, and transports these to the client.

The SQL relaying was first carried out with a one-way invocation which on average took approximately 15 ms. Later when we discovered synchronization problems, we returned to full duplex invocations. With one-way invocation the request to change the current SQL query wasnt fully completed and we started fetching an outdated resultset. The timings measured on the client are shown on the left, and the timings from the server side are on the right.

4.3.6 The generation of IDL constructs

The current IDL compiler that is supplied by Java, uses class extension to delegate the implementation of the IDL interface. In general the Corba server is started and waits for invocations from the clients. So the source code for the server skeleton is almost invariant. The implementation of the IDL methods are handled in a Servant, the code has to be written by hand. In the case of the server needing to use `Any` types, the Servant needs to have access to the Corba factory. We dont know the amount of rows in a resultset, so we cannot allocate an array beforehand. So we use the new Java collection classes to store the SQL results and convert them later to an array. Having written such a servant as seen in listing 4.10, we can now create Python scripts to generate source code for similar servants.

In conclusion we can say that the benchmarks not only provide insight in the relative speeds of the various CORBA constructs, but also the feasibility of how to map database records on these constructs. The understanding of these mappings is important as we will need to write scripts that generate the source code for these mappings.

	Client				Server	
#records	t1	t2	t3	t4	t5	t6
100	369,50	15,50	149,50	268,90	44,20	13,70
200	368,20	15,50	192,30	315,20	45,00	28,80
300	373,10	15,50	221,20	409,20	48,00	42,90
400	370,10	15,20	245,80	344,60	52,60	58,10
500	368,70	15,30	284,00	393,30	57,30	69,80
600	369,70	15,50	319,40	483,40	57,90	76,70
700	369,60	15,30	339,70	515,30	64,80	88,80
800	370,30	15,10	383,70	627,50	68,50	106,60
900	370,00	15,40	403,30	621,10	65,80	116,70
1000	369,90	15,50	433,10	707,90	76,00	124,90

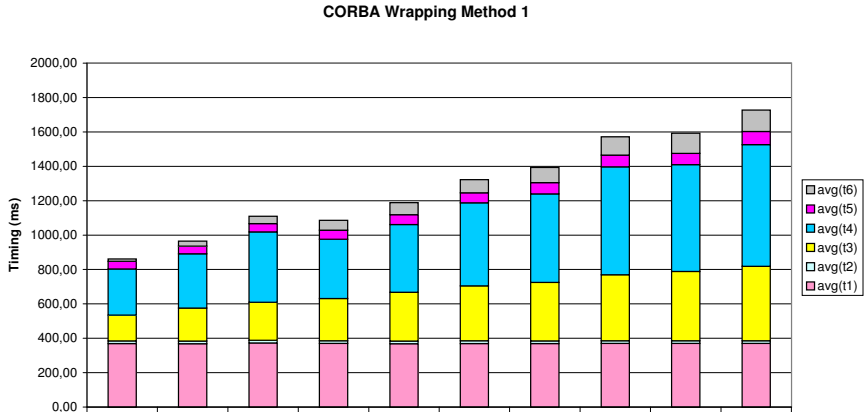
Figure 4.4: Benchmark CORBA Wrapping Method 1 (IDL Sequence of IDL Structs)

	Client				Server	
#records	t1	t2	t3	t4	t5	t6
100	386,40	59,80	403,90	187,60	40,50	402,10
200	376,90	63,70	704,40	266,80	44,10	702,90
300	370,10	66,80	1010,90	191,80	47,10	1009,20
400	372,20	68,90	1308,00	322,40	49,30	1305,90
500	370,60	72,10	1653,90	505,40	52,30	1652,20
600	373,30	76,70	1961,30	444,90	56,90	1959,80
700	371,60	82,10	2265,60	529,40	62,30	2264,00
800	372,30	89,80	2575,30	581,80	69,60	2573,30
900	371,80	86,00	2896,30	656,40	66,10	2894,60
1000	372,90	86,60	3182,30	669,40	67,00	3180,80

Figure 4.5: Benchmark CORBA Wrapping Method 2 (Tuple Functions)

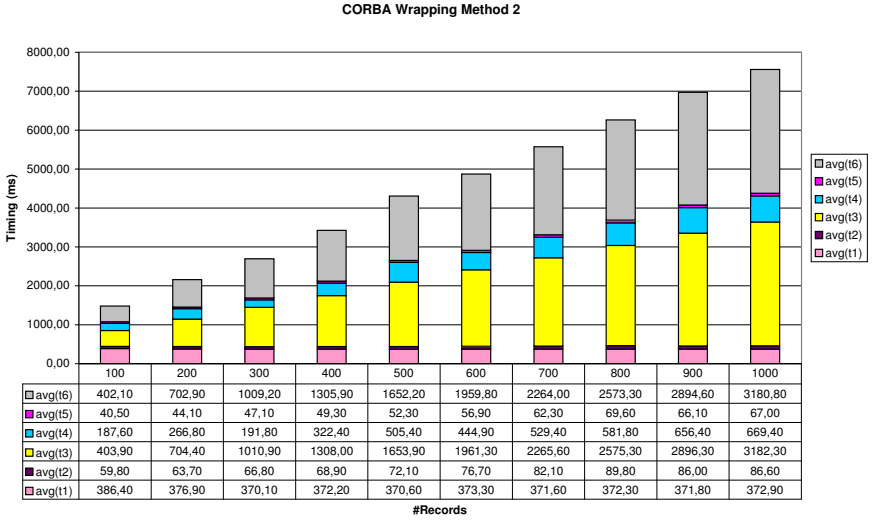
	Client				Server	
#records	t1	t2	t3	t4	t5	t6
100	380,80	16,00	393,90	363,60	41,60	35,30
200	369,20	15,90	447,00	394,30	48,10	58,70
300	370,50	16,00	499,20	540,70	48,00	81,70
400	369,00	15,50	547,80	639,00	51,70	107,50
500	368,40	15,20	597,60	625,70	55,50	138,80
600	370,40	15,30	668,40	599,10	58,60	155,90
700	370,80	15,60	710,30	510,20	62,40	185,80
800	370,40	15,30	763,40	750,70	72,90	214,30
900	372,30	15,30	808,00	819,60	66,70	234,70
1000	370,60	15,30	859,10	906,20	72,00	263,20

Figure 4.6: Benchmark CORBA Wrapping Method 3 (Serialized Java Objects)



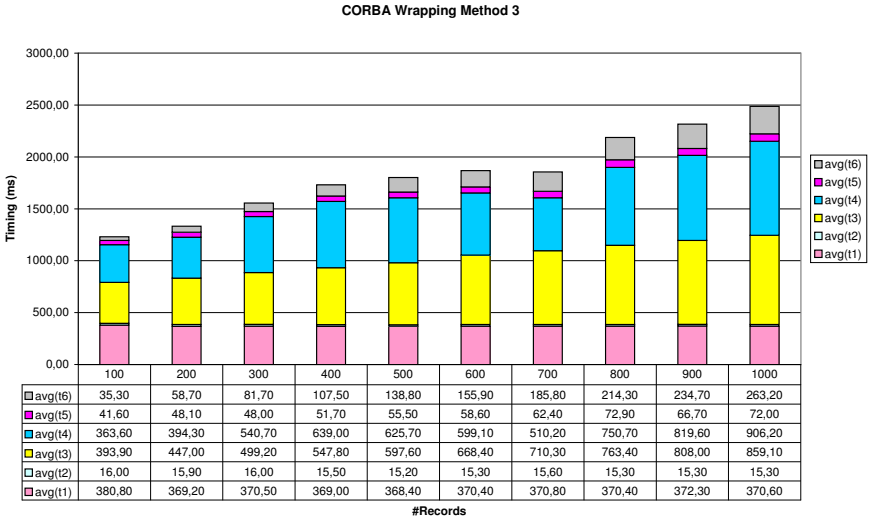
	100	200	300	400	500	600	700	800	900	1000
avg(t6)	13,70	28,80	42,90	58,10	69,80	76,70	88,80	106,60	116,70	124,90
avg(t5)	44,20	45,00	48,00	52,60	57,30	57,90	64,80	68,50	65,80	76,00
avg(t4)	268,90	315,20	409,20	344,60	393,30	483,40	515,30	627,50	621,10	707,90
avg(t3)	149,50	192,30	221,20	245,80	284,00	319,40	339,70	383,70	403,30	433,10
avg(t2)	15,50	15,50	15,50	15,20	15,30	15,50	15,30	15,10	15,40	15,50
avg(t1)	369,50	368,20	373,10	370,10	368,70	369,70	369,60	370,30	370,00	369,90

Figure 4.7: Graphing Benchmark CORBA Wrapping Method 1



	100	200	300	400	500	600	700	800	900	1000
avg(t6)	402,10	702,90	1009,20	1305,90	1652,20	1959,80	2264,00	2573,30	2894,60	3180,80
avg(t5)	40,50	44,10	47,10	49,30	52,30	56,90	62,30	69,60	66,10	67,00
avg(t4)	187,60	266,80	191,80	322,40	505,40	444,90	529,40	581,80	656,40	669,40
avg(t3)	403,90	704,40	1010,90	1308,00	1653,90	1961,30	2265,60	2575,30	2896,30	3182,30
avg(t2)	59,80	63,70	66,80	68,90	72,10	76,70	82,10	89,80	86,00	86,60
avg(t1)	386,40	376,90	370,10	372,20	370,60	373,30	371,60	372,30	371,80	372,90

Figure 4.8: Graphing Benchmark CORBA Wrapping Method 2



	100	200	300	400	500	600	700	800	900	1000
avg(t6)	35,30	58,70	81,70	107,50	138,80	155,90	185,80	214,30	234,70	263,20
avg(t5)	41,60	48,10	48,00	51,70	55,50	58,60	62,40	72,90	66,70	72,00
avg(t4)	363,60	394,30	540,70	639,00	625,70	599,10	510,20	750,70	819,60	906,20
avg(t3)	393,90	447,00	499,20	547,80	597,60	668,40	710,30	763,40	808,00	859,10
avg(t2)	16,00	15,90	16,00	15,50	15,20	15,30	15,60	15,30	15,30	15,30
avg(t1)	380,80	369,20	370,50	369,00	368,40	370,40	370,80	370,40	372,30	370,60

Figure 4.9: Graphing Benchmark CORBA Wrapping Method 3

```
class DbOrb1Servant extends DbOrb1._DataMediatorImplBase {
  ORB    iorb;
  rwDatabase rwdb;

  public DbOrb1Servant(ORB orb)
  {
      iorb = orb;
      rwdb = new rwDatabase("org.gjt.mm.mysql.Driver");
      rwdb.open("jdbc:mysql:///test","test");
  }

  public AdresData[] getAdresses()
  {
      AdresData adr  = null;
      rwResultSet res;
      res = rwdb.execSql("select * from adressen");

      Any returnResults= iorb.create_any();
      ArrayList ar = new ArrayList();

      while (res.more()) {
          adr = new AdresData();
          adr.iadresnr = res.getInt(1);
          adr.inaam    = res.getString(2);
          adr.istraat  = res.getString(3);
          adr.ipostcode = res.getString(4);
          ar.add(adr);
          System.out.print(".");
      }
      AdresData[] regarray = (AdresData [])ar.toArray(new AdresData[0]);
      return(regarray);
  }
}
```

Figure 4.10: Listing CORBA Database Server

```
import random
from java.io import *


def pc():
   s1 = "".join([random.choice("123456789") for i in range(4)])
   s2 = "".join([random.choice("ABCDEFGHIJKLMNOPQRSTUVWXYZ") for i in range(2)])
   return s1+" "+s2

def straat():
   s1 = "".join([random.choice(["bos","hei","zilver","goud","gras","bree","buite
n","binnen","oost","zuid","noord","west"])])
   s2 = "".join([random.choice(["weg","straat","plein","laan","baan","hof","kade
"])])
   s3 = "".join([random.choice("123456789") for i in range(random.randrange(1,4)
)])
   return s1+s2+" "+s3

def naam():
   s1 = "".join([random.choice(["P.","K.","R.","T.","A.","F.","D.","Th.","N."])
for i in range(random.randrange(1,3))])
   s2 = "".join([random.choice([" "," van "," van der "," ter "])])
   s3 = "".join([random.choice(["Oort","Berg","Linden","Zee","Velde","Wilg","Bal
k","Blok","Vaart"])])
   return s1+s2+s3

fos = FileOutputStream("filldb.sql")

print "Generating 100 adresses"
outs = ""
for i in range(100):
   outs = outs + "INSERT INTO adres VALUES (’’,’"+naam()+"’,’"+straat()+"’,’1100
 AA’);\r\n"

print "Generating 200 adresses"
for i in range(200):
   outs = outs + "INSERT INTO adres VALUES (’’,’"+naam()+"’,’"+straat()+"’,’1200
 AA’);\r\n"

print "Generating 300 adresses"
for i in range(300):
   outs = outs + "INSERT INTO adres VALUES (’’,’"+naam()+"’,’"+straat()+"’,’1300
 AA’);\r\n"

print "Generating 400 adresses"
for i in range(400):
   outs = outs + "INSERT INTO adres VALUES (’’,’"+naam()+"’,’"+straat()+"’,’1400
 AA’);\r\n"

print "Generating 500 adresses"
for i in range(500):
   outs = outs + "INSERT INTO adres VALUES (’’,’"+naam()+"’,’"+straat()+"’,’1500
 AA’);\r\n"
```

Figure 4.11: Listing filldb.py

Chapter 5

RMI

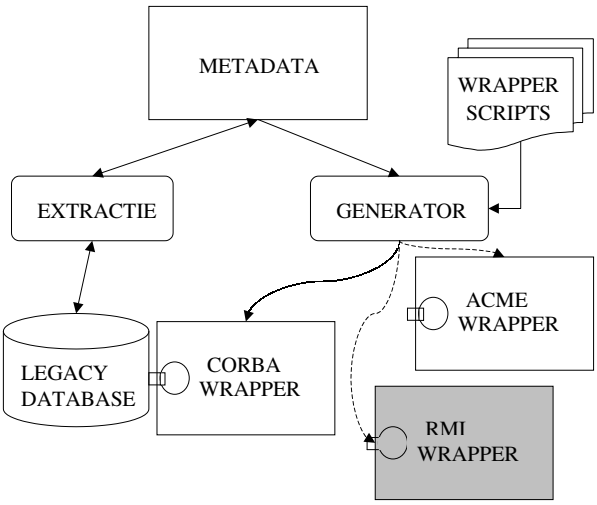


Figure 5.1: RMI Database Wrapper

5.1 Introduction

In chapter 4 we reviewed distributed object architecture using CORBA. Java Remote Method Invocation (RMI) is an alternative architecture. It is the Java native way to write distributed objects in Java that enable objects to be distributed across Java Virtual Machines (JVM) that reside, either locally or remote across networks. RMI provides the ability to execute methods on these objects in a location transparent way. RMI resembles Remote Procedure Call (RPC) technology, but instead of simple cross-process/machine networking, it adds object orientation concepts like inheritance and encapsulation.

RMI relieves the programmer from the following distributed issues:

- Marshalling
  In order to pass parameters (objects and/or primitives) over the network, the parameters need to be packaged in a way so that any participant of the heterogenous environment can use them.
- Parameter passing
  Two major conventions exist: *pass-by-value* (a copy of the data is passed) and *pass-by-reference* (changes on a remote location affect the local data).
- Distributed Garbage Collection
  Java is renound for its built-in garbage collection. On a single Virtual Machine local references to objects can easily be tracked. In a distributed environment references to objects across the network need to be taken in account.
- Activation
  Objects can be instantiated when required. Object implementation may not already be known, so object behaviour code needs to be downloaded on the fly.

5.2 Components of RMI

In RMI a *remote object* is a possible networked object whose methods can be invoked remotely from another Java VM. RMI Objects conform to the object-oriented notion of the interface separated from the implementation. The interface exposes only the methods that the client can use. This way different object behaviour can be plugged in without changing client code. In order to flag an object as remotely invokable its class must implement the *java.rmi.Remote* interface. This in turn will enable the generation of networking code for the interface. To render the illusion of local/remote transparency, RMI has to delegate local methods to actual object implementation which can reside on the network. The seemingly local object part is called a *stub* and the possible remote object part is called a *skeleton*. These parts are generated by the RMI compiler (rmic) from the class that implement the remote interface. CORBA uses a similar stub/skeleton concept to shield the programmer from networking issues.

5.2.1 Marshalling

RMI uses object serialization to marshal the method invocation parameters. Serialization is the conversion of objects hierarchies into byte streams and vice versa. In order to persist objects in Java to byte streams, these objects need to implement the *java.lang.Serializable* interface. Serialization is not a cheap operation especially with large object graphs. Much of the research regarding the use of RMI in High Performance Computing is directed to optimizing serialization. [6] [5]

5.2.2 Parameter Passing

The Java programming language uses *pass-by-reference* calling convention by default when calling normal object methods. This is in contrast with parameters invoked by RMI remote methods. Which rely on serialization and are passed by value. Nevertheless pass-by-reference parameters can be mimicked with RMI through the use of remote object references. Instead of a remote object the stub of the remote object is passed to the client, so it can invoke callbacks on the remote object. In order to pass Java RMI stubs they need to be deriven from the *java.rmi.server.UnicastRemoteObject*.

Let us consider a database broker implemented with RMI. The client can invoke SQL queries on the database broker, and database data will be returned. The database data will likely be passed by value, yet the interface to fire SQL queries will be passed by reference, as the interface methods can be considered callbacks on the database broker.

```
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

public class SQLBroker extends UnicastRemoteObject implements ISQLBroker {

   public static void main(String args[]) throws Exception {
      reg = LocateRegistry.getRegistry(9999);
      SQLBroker sqlbroker = new SQLBroker(reg);
   }

   public SQLBroker(Registry reg) throws Exception, RemoteException {
      super();
      reg.rebind("SQLBroker",this);
      Sytem.out.println("SQLBroker bound");
   }

   public void fireSQL(String sql) throws RemoteException {
      // handle SQL, possibly sending data
   }
}

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.Naming;

public class SQLBrokerClient {

   public static void main(String args[]) throws Exception {

      try {
         Remote remoteObject = Naming.lookup("rmi://machine:port/SQLBroker");
      } catch (Exception e) {
         System.err.println("Error in lookup");
         System.exit(-1);
      }

      try {
         // callback fireSQL
         sqlBroker.fireSQL("SELECT * FROM WERKNEMERS");
      } catch (RemoteException e) {
         System.err.println("Remote Error: "+e);
      }
   }
}

import java.rmi.Remote;
```

```
import java.rmi.RemoteException;
public interface ISQLBroker extends Remote {
   public void fireSQL(String sql) throws RemoteException;
}
```

5.2.3 Distributed Garbage Collection

In a world without distributed garbage collection, a lot of programming effort would be dedicated to the book keeping of remote objects. Fortunately RMI does its own housecleaning. Whenever a client gets a reference to a remote object, it will be given a *lease*. This lease will be continually renewed as long as the client keeps sending a heartbeat to the server of the remote object. In distributed environments there are many more points of failure. In the end, clients either crash or no longer reference remote objects. Their leases will expire, and the corresponding remote objects will automatically be garbage collected.

5.3 Enterprise Java Beans

Enterprise JavaBeans (EJB) is a server-side component architecture that simplifies building enterprise-wide distributed object applications in Java. By using the EJB framework you can take advantage of scalable and reliable applications without having to write your own complex distributed object framework. EJB promises rapid application development of server-side components through a prewritten distributed infrastructure. EJB is designed to support portability and reusibility across middleware services like RMI and CORBA. So instead of figuring out the way RMI and CORBA works (see the previous chapters), and designing object-relational mapping ourselves, we can rely on the EJB application server to handle these tasks for us.

5.4 CORBA and RMI convergence

The main difference between CORBA and EJB is that CORBA only has the notion of an object. It does not have the notion of a server side component that is managed by an application server. OMG is planning a CORBA components proposal[1], but it remains to been seen who will benefit from this proposal. The goal of CORBA components is to make it possible for an CORBA component to appear as an enterprise bean and vice versa. EJB meanwhile already supports CORBA. Instead of RMI through the JRMI protocol, all communication in this case is done through the RMI-IIOP protocol. RMI-IIOP enables Remote Method Invocation that uses the Internet Inter-ORB Protocol so RMI and EJB can integrate with CORBA services. This way EJB can integrate with legacy applications and also integrate non-Java languages like COBOL and C++.

5.4.1 Obstacles

As seen earlier, CORBA objects are passed around the network by reference and CORBA objects never leave their host. With RMI, objects are primarly passed by value. To overcome this semantic difference , the notion of a special *value type* is to be added to the OMG IDL specification[2]. Having this special value type, CORBA Objects-By-Value are made possible, which calls for a serialization mechanism to pack and unpack the state of a CORBA object. Besides differences regarding Distributed Garbage Collection and Narrowing, most Java programmers

[1] ftp://ftp.omg.org/pub/docs/orbos/98-12-02.pdf
[2] ftp://ftp.omg.org/pub/docs/orbos/98-01-18.pdf

only like to program in Java. To use CORBA a programmer has to learn IDL, so it comes as no suprise that OMG came up with reverse mapping: the *Java-to-IDL Mapping* specification.

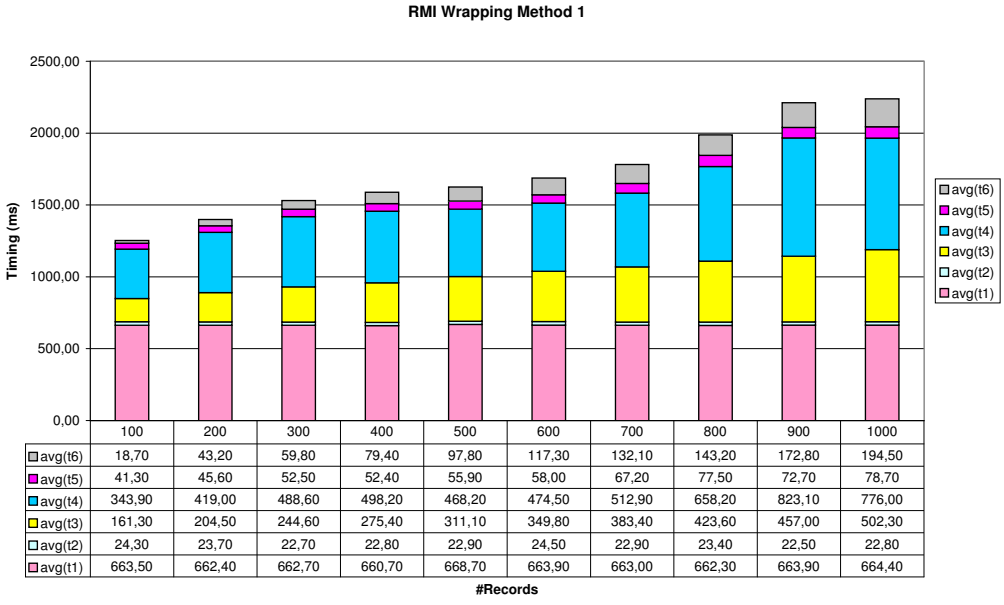
5.5 RMI Benchmark compared to CORBA

In the previous chapter we looked at the various ways we could wrap SQL results in CORBA, because CORBA is lacking pass-by-value. It is also interesting to see how RMI SQL results could be passed with RMI. Comparing listing 4.10 and listing 5.4 we see that the differences in Java code between the CORBA server and RMI server are very small. If we compare the values of the CORBA benchmark 4.4 and the RMI benchmark 5.2 we see that the time spent in startup and SQL execution (t5) are similar. For wrapping records CORBA uses an average of 1,3 ms per 100 records and RMI spends 2,0 ms per 100 records. Lookup and invocation are also slower in RMI than in CORBA.

If we take a look at listing 4.10 and listing 5.4, we again see the striking similarities between the CORBA database servant and the RMI database servant.

	Client				Server	
#records	t1	t2	t3	t4	t5	t6
100	663,50	24,30	161,30	343,90	41,30	18,70
200	662,40	23,70	204,50	419,00	45,60	43,20
300	662,70	22,70	244,60	488,60	52,50	59,80
400	660,70	22,80	275,40	498,20	52,40	79,40
500	668,70	22,90	311,10	468,20	55,90	97,80
600	663,90	24,50	349,80	474,50	58,00	117,30
700	663,00	22,90	383,40	512,90	67,20	132,10
800	662,30	23,40	423,60	658,20	77,50	143,20
900	663,90	22,50	457,00	823,10	72,70	172,80
1000	664,40	22,80	502,30	776,00	78,70	194,50

Figure 5.2: Benchmark RMI Wrapping Method



	100	200	300	400	500	600	700	800	900	1000
avg(t6)	18,70	43,20	59,80	79,40	97,80	117,30	132,10	143,20	172,80	194,50
avg(t5)	41,30	45,60	52,50	52,40	55,90	58,00	67,20	77,50	72,70	78,70
avg(t4)	343,90	419,00	488,60	498,20	468,20	474,50	512,90	658,20	823,10	776,00
avg(t3)	161,30	204,50	244,60	275,40	311,10	349,80	383,40	423,60	457,00	502,30
avg(t2)	24,30	23,70	22,70	22,80	22,90	24,50	22,90	23,40	22,50	22,80
avg(t1)	663,50	662,40	662,70	660,70	668,70	663,90	663,00	662,30	663,90	664,40

Figure 5.3: Benchmark RMI Wrapping Method

```
public class DbRMIServer extends java.rmi.server.UnicastRemoteObject
                         implements DataMediatorServer {

   public DbRMIServer () throws java.rmi.RemoteException {
      super();
   }

   public void DBinit()
   {
      rwdb = new rwDatabase("org.gjt.mm.mysql.Driver");
      rwdb.open("jdbc:mysql:///test","test");
   }

   public void setSQL(String Sql) throws java.rmi.RemoteException
   {
      iSQL = Sql;
   }

   public ArrayList getAdresses() throws java.rmi.RemoteException
   {
       System.out.println("getAdresses");
       AdresData adr  = null;
       rwResultSet res;

       res = rwdb.execSql(iSQL);
       ArrayList ar = new ArrayList();

       while (res.more()) {
           adr = new AdresData();
           adr.iadresnr = res.getInt(1);
           adr.inaam    = res.getString(2);
           adr.istraat  = res.getString(3);
           adr.ipostcode = res.getString(4);
           ar.add(adr);
       }

       return (ar);
   }
}
```

Figure 5.4: Listing RMI Database Server

Chapter 6

Implementation of the Wrapper Generator

6.1 Introduction

In this chapter we will combine the technologies of the previous chapters, and present a framework consisting of a Graphical User Interface, that supports Database browsing, reusing the Database façade objects of chapter 2, an Editor for editing the source generating scripts, and a Scripting Engine that ties it all together.

We will now discus the design of the R2D2 [1]framework. This framework consists of several components, such as a Database metadata viewer, and several scripts to generate source code for the generation of database wrappers for several environments. The framework application is written in Java, and the GUI consists of Swing components. Not all parts of the userinterface were written in Java, for example the contents of the database browser are filled by Python generated Java objects, showing how easy it is to integrate Python Objects and Java Objects.

6.2 R2D2 framework

For the generation of source code that wraps database connectivity in various distributed environments, we supply the R2D2 framework. This framework has 3 views, each on a separate tab, that allows browsing of database data, editing of source generating scripts, and a viewer for generated source code. Figure 6.1 shows a typical view when R2D2 is first started.

6.2.1 Database Browser

The database browser consists of two panels, the left panel shows the current selected database tree, this tree shows the major components of the database like tables, fields, indexes and privileges. More information about these components is given on the right panel in a table view. The table view uses the Java Swing user interface. Swing uses the Model-View-Controller Pattern for decoupling the presentation from the underlying data. This way several views can be shown from the same data, and changes in the data will be automatically passed to each view. Each view for example Tree View, Table View needs a Model, so for R2D2 a TableDBModel and a default TreeModel were written in Java. The population of these

[1]Although R2D2 sounds like a good name for a droid, in this case it is the acronym for 'Relational Data to Distributed Data'

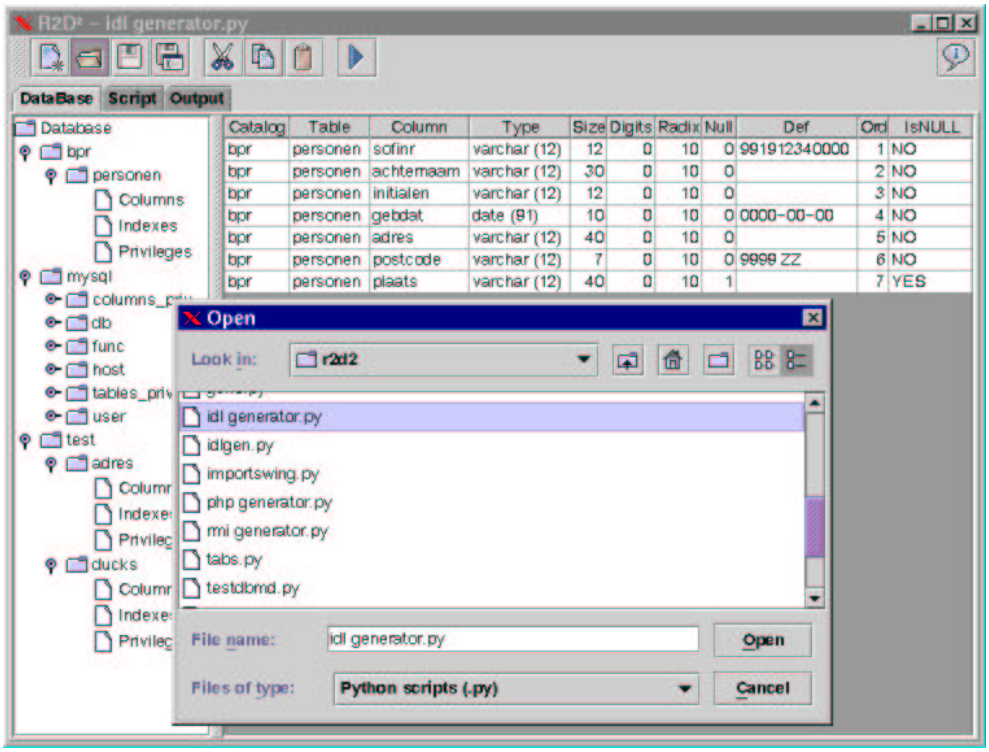


Figure 6.1: R2D2 : browsing database metadata

models and the coordination of the interaction between the Tree View and Table View is carried out by JPython. JPython is also responsible for large portions of the user interface. The creation of all the user interface components are done in the *GUI.py* script, and thus allows easy customization of the R2D2 user interface. The following snippet from GUI.py shows how action buttons can be customized.

```
bp = JToolBar()

new = JButton(actionCommand="new", borderPainted=1)
new.icon=ImageIcon("rsrcs/New24.gif")
new.toolTipText="New (Ctrl-N)"

go = JButton(actionCommand="go", borderPainted=1)
go.icon=ImageIcon("rsrcs/Play24.gif")
go.toolTipText="Run the code (F4)"

bp.add(new)
bp.add(Box.createHorizontalStrut(10))
bp.add(go)
```

6.2.2 Script Editor

The script editor also consists of two panels, the upper panel features a basic editor, and the lower panel is for messages from the scripting engine. The script editor is used to write the source generating scripts. Currently the scripting engine only support Python scripts. We can use Python as an extension language and also as a Java API explorer. The script editor has basic editor features like copy, cut, paste, and management functions to support the creation, saving, loading of files. The editor also checks whether file changes are not yet committed. The contents of the

editor are passed to the scripting engine when the run button is clicked. Should an exception occur during the script execution, then the exception and traceback is output in the lower message panel. The traceback can be used to pinpoint the linenumber of the offending script construct.

```
import javax.swing as swing

class myframe:
   def exitProg(self,event):
      self.win.dispose()

   def __init__(self):
      self.win = swing.JFrame("Welcome")
      self.close = swing.JButton("Close Me",actionPerformed=self.exitProg)
      self.win.contentPane.add(self.close)
      self.win.pack()
      self.win.show()

f = myframe()
```

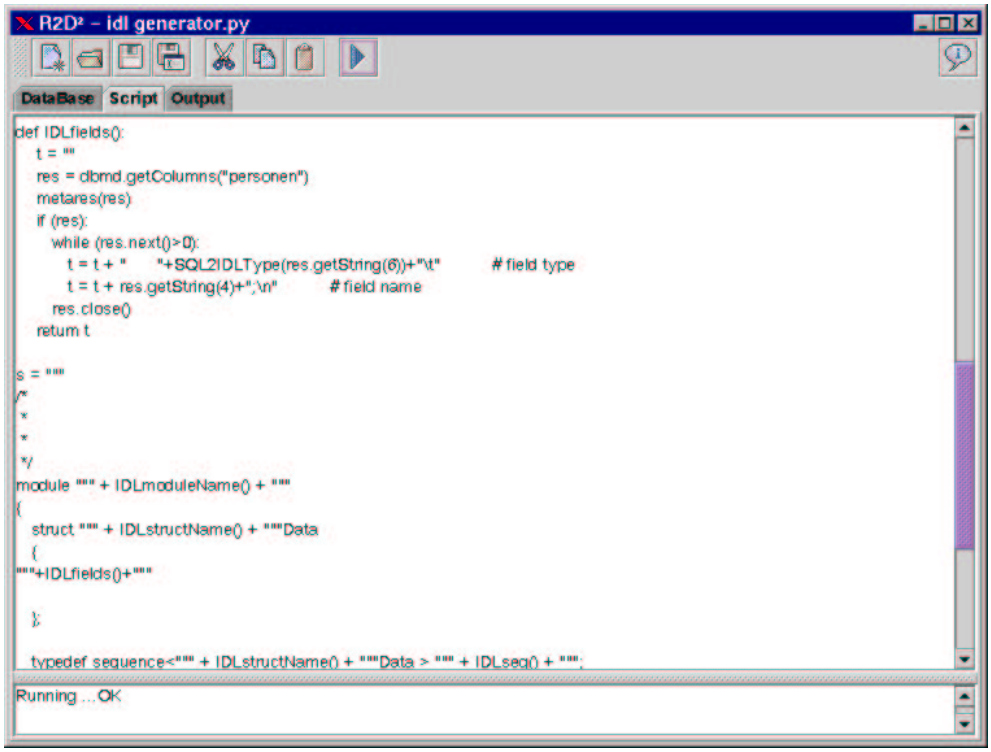


Figure 6.2: R2D2 : source code generating script

6.2.3 Generating Output

In our framework we have a python base class that forms the basis for a simple source generator. This base class can print itself, and includes an header with the name of the generator and its version number, a timestamp when the source code was generated, and the contents of the source body. This way scripts can be added that extend from the BaseGenerator.

```
from time import time, gmtime

class BaseGenerator:
   version = "1.00"
   name   = "BaseGenerator"
   body    = ""

   def timenow(self):
       gmt = gmtime(time())
       return '%02d:%02d:%02d GMT' % (gmt[3], gmt[4],gmt[5])

   def header(self):
       return """
/*
 *  generator: """+self.name+" "+self.version+"""
 *  generated at:""" +self.timenow() +"""
 */"""

   def body(self):
       return ""

   def __repr__(self):
       return self.header()+self.body()
```

In chapter 2 we saw that quite often we had to write database wrappers by hand. We even proposed a template (figure 2.4) to generate these wrappers. We will now use Python and our database façade objects to write these templates for us. We will use the triple quote mechanism of Python to preserve the preformatted template body and use user defined functions to fill in the template fields.

```
class datawrapgen(BaseGenerator):

    name = "datawrapgen"
    version = "1.00"

    def body(self):
        s = """
class """ + self.tableName() + """Datawrapper
{
""" + self.tabledatafields() + """

   public """ + self.tableName() + """Datawrapper(ResultSet res)
   {
     try {
     """ + self.fieldsTable() + """
     } catch (Exception e) {

     }
   };
};
"""
        return s
```

Running this script will generate the following output as depicted in figure 6.3

6.2.4 Generated output

By adding a scripting engine to the Framework, end-users can drive and extend the R2R2 application. Therefore we have incorporated a Java-based scripting language interpreter that runs in the same JVM as the application. Besides the benefits there are some drawbacks as well. The scripting engine has direct reference to all the application Java classes, so we have to introduce an API or obfuscate classes in order to safeguard the scriping from using all the classes. Bare access to all the

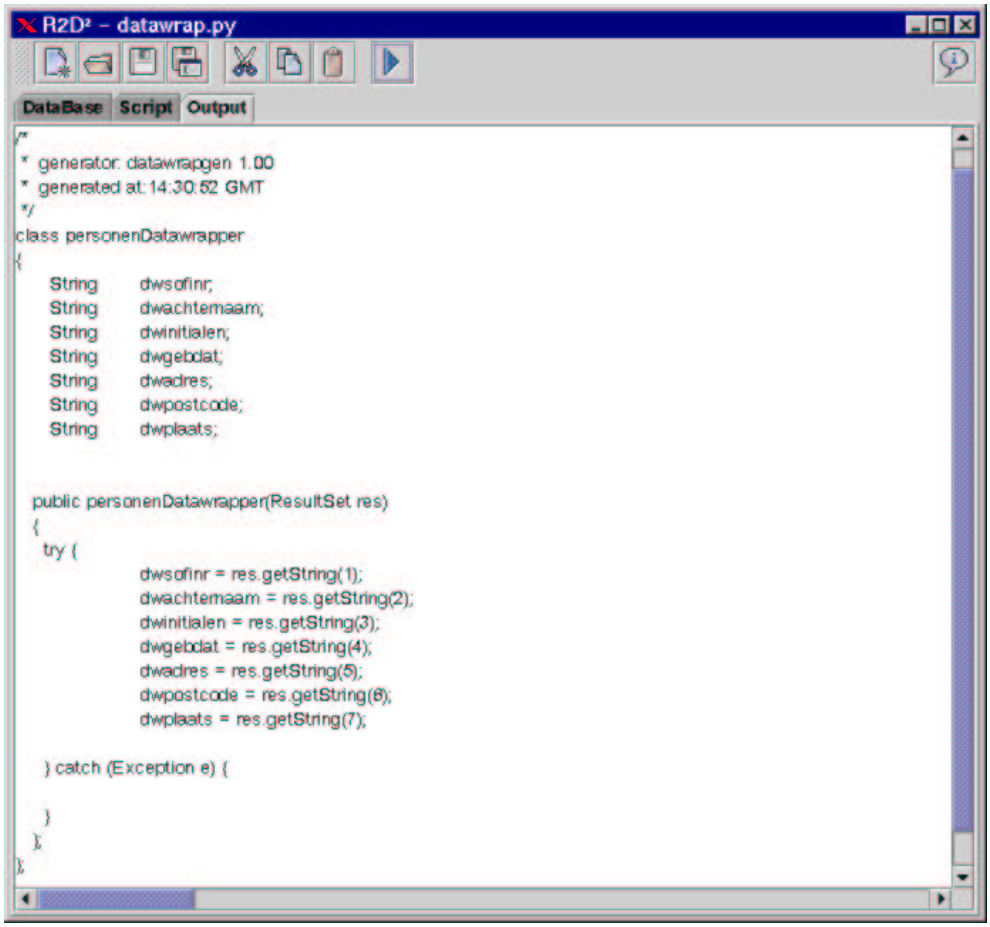


Figure 6.3: R2D2 : output of datawrap.py

API's can lead to scripts that need complex API's for each subsystem and how to use them. Here one can re-use the façade interface of 2.4.5

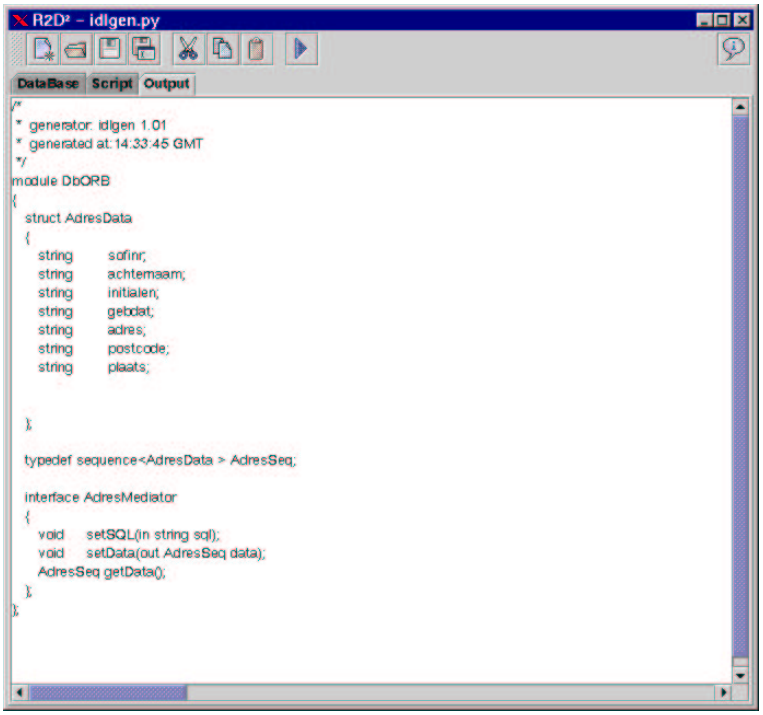


Figure 6.4: R2D2 : source code generated by idlgen.py

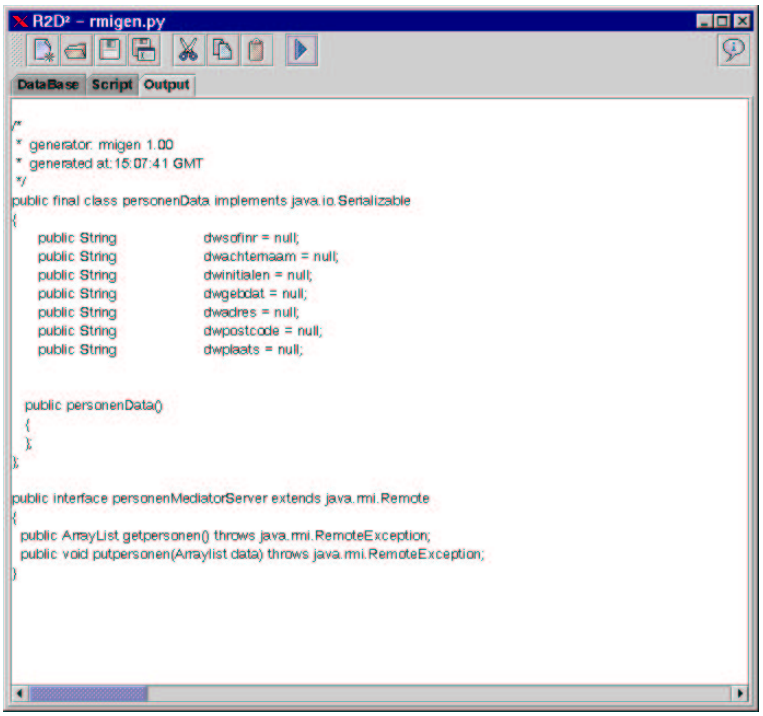


Figure 6.5: R2D2 : source code generated by rmigen.py

Chapter 7

Conclusions

7.1 Results

Our research was aimed primarily at answering the central question: *How to use Java based middleware like CORBA,RMI to unlock legacy databases*

Upon tackling this rather broad research question, we found out that we needed

- some way to extract metadata from the legacy databases
- programs that help generate source code that wrap database data for use in distributed environments like CORBA and RMI.

In Chapter 2 we discussed the Databases and metadata, and what Java offers for database connectivity. We saw that the initial offering was lacking features to have automatic object/relational mapping. We wrote our database façade components to make database access much more manageable, and simplified the discovery and management of various database metadata. We hinted at some template engine to ease the generation of source code. In order to actually generate source code we examined some scripting engines that could be run on top of Java in chapter 3. We took a closer look at what distributed environments entail on the Java platform. We discovered clear differences between parameter passing conventions between CORBA and RMI. But these could be overcome by writing different scripts for each distributed environment. In Chapter 6 we delivered the R2D2 framework, that reuses the the façade objects. Combined with the scripting engine of Chapter 3 we can traverse the discovered metadata and generate source code for database access for various distributed environments. Through scripting we can generate sourcecode for future environments and also customize and extend the current framework

7.2 Future Research

- The current version of R2D2 can generate source code from discovered database metadata. For use in distributed environments like CORBA and RMI an additional compiling/pre-processing step is needed. These precompilers are not well integrated into the current framework. Further research on the behaviour of these compilers could lead to a tighter integration of these middleware compilers.
- The R2D2 framework uses JPython as a scripting engine to customize the Java userinterface and to create source code. A deeper study to upcoming embeddable JVM languages could lead to scripting engines that are more pluggable, so end users can swap in their favourite scripting language of choice.

- The current source generating scripts use a basic strategy to convert database data into data classes. Further examination of persistence frameworks will lead to more insight in the object/relational mapping and hence to better generated source code
- Database metadata discovery is currently only available to Java, some export mechanism would enable other tools to use the metadata on different platforms.

7.3 Parting thoughts

The use of middleware hasn't diminished over the years. Instead several forms of middleware have come to the limelight. CORBA has been a longtime contender, but with the curent wave of Java technologies, RMI is gaining more acceptance. Instead of writing a framework from the ground up, more and more application servers are being offered. Application Servers built with Java are mostly based on the Enterprise Java Bean (EJB) specification. The EJB specification doesnt impose a specific middleware like CORBA or RMI, so it is the vendors decision to choose an platform. RMI promises easy transportation of deep object hierarchies at expense of speed. CORBA enables easy integration with non Java technologies. In hindsight the in-house design of robust object mapping techniques, synchronization techniques soon lead to proprietary solutions that are hard to maintain over time. This very same problem that has plagued the OpenGAC solution.The dependency on one technology or manufacturer makes it harder to switch parties or technology. But even promissing cures like EJB and CORBA can suffer from underspecified specifications and thereby limiting transparant interchange of various middleware components. But at least you can hold the vendor of the EJB responsible. There is no single solution for object wrapping, although reverse engineering has less possibilities because of the constraints imposed by the existing datamodel, it can be conducted easier than forward engineering but with limited model expression. The synchronization of objects and underlying datastore is still a work in progress, it is hard to make a sound design, so it is better to be developed by companies that do so as a core business.

Appendix A

DBMeta sources

A.1 Some metrics of the software produced

We can distinguish 4 types of software we have produced for this thesis.

- The various Benchmark classes for different types of CORBA Wrapping

Client	Server	Specification
BmOrb1Client.java 1 Class (67 LOC)	DbOrb1Server.java 2 Classes (56 LOC)(23 LOC)	DbOrtb1Test.idl (18 LOC)
BmOrb2Client.java 1 Class (73 LOC)	DbOrb2Server.java 2 Classes (66 LOC)(23 LOC)	DbOrtb2Test.idl (18 LOC)
BmOrb3Client.java 1 Class (77 LOC)	DbOrb3Server.java 2 Classes (67 LOC)(23 LOC)	DbOrtb3Test.idl (20 LOC)
BmRMIClient.java 1 Class (77 LOC)	DbiRMIServer.java 1 Class (89 LOC)	DataMediatorServer.java RMI Interface (6 LOC)

- The java core sources of the R2D2 interface

r2d2.java	16 Methods	226 LOC
GUILoader.java	5 Methods	29 LOC
JTextAreaWriter.java	8 Methods	41 LOC
ExampleFileFilter.java	12 Methods	87 LOC
TableDBModel.java	7 Methods	59 LOC
ColumnDBModel.java	7 Methods	87 LOC
IndexDBModel.java	7 Methods	79 LOC

- The python source generating scripts used by R2D2

GUI.py	7 Methods	195 LOC
importswing.py	0 Methods	23 LOC
clearwindows.py	0 Methods	3 LOC
datawrap.py	6 Methods	72 LOC
dbtree.py	4 Methods	66 LOC
basegen.py	4 Methods	26 LOC
idlgen.py	9 Methods	76 LOC
rmigen.py	7 Methods	81 LOC
xmlgen.py	7 Methods	67 LOC

- The rwDBMetaData classes which act as a façade for the JDBC classes and allow for easy discovery of metadata from JDBC compliant databases.

The sources for rwDBMetaData and R2D2 are available from http://www.ramco.nl

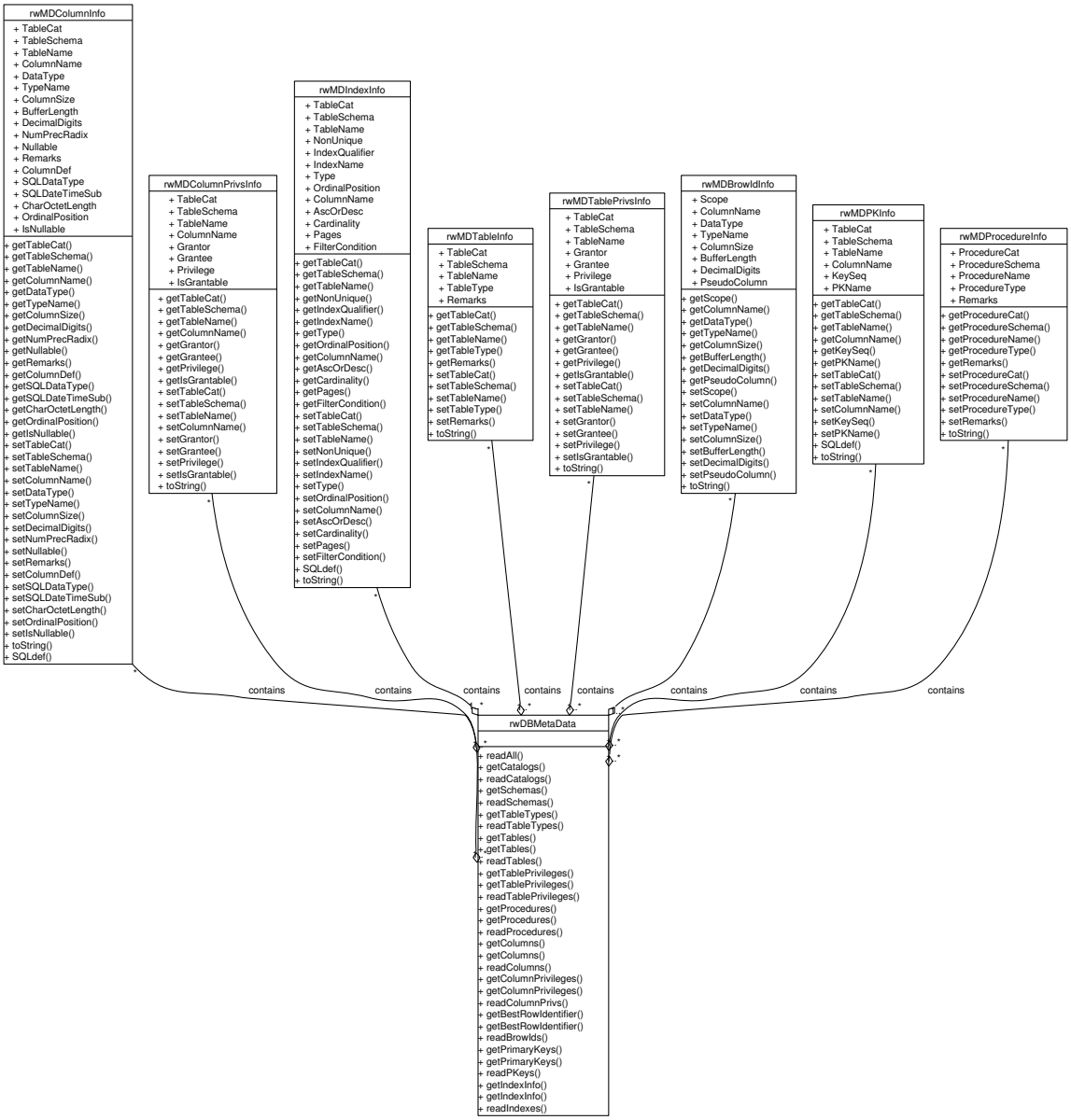


Figure A.1: Class diagram of the Façade components

Java Source: rwDBMetaData (379 LOC)		
Number of Classes: 1	Number of Methods: 32	rwDBMetaData (377 LOC)
rwDBMetaData	rwDBMetaData	(17 LOC)
rwDBMetaData	readAll	(13 LOC)
rwDBMetaData	getCatalogs	(7 LOC)
rwDBMetaData	readCatalogs	(9 LOC)
rwDBMetaData	getSchemas	(7 LOC)
rwDBMetaData	readSchemas	(9 LOC)
rwDBMetaData	getTableTypes	(7 LOC)
rwDBMetaData	readTableTypes	(9 LOC)
rwDBMetaData	getTables	(5 LOC)
rwDBMetaData	getTables	(7 LOC)
rwDBMetaData	readTables	(18 LOC)
rwDBMetaData	getTablePrivileges	(5 LOC)
rwDBMetaData	getTablePrivileges	(7 LOC)
rwDBMetaData	readTablePrivileges	(20 LOC)
rwDBMetaData	getProcedures	(5 LOC)
rwDBMetaData	getProcedures	(7 LOC)
rwDBMetaData	readProcedures	(18 LOC)
rwDBMetaData	getColumns	(5 LOC)
rwDBMetaData	getColumns	(7 LOC)
rwDBMetaData	readColumns	(34 LOC)
rwDBMetaData	getColumnPrivileges	(5 LOC)
rwDBMetaData	getColumnPrivileges	(7 LOC)
rwDBMetaData	readColumnPrivs	(21 LOC)
rwDBMetaData	getBestRowIdentifier	(5 LOC)
rwDBMetaData	getBestRowIdentifier	(7 LOC)
rwDBMetaData	readBrowIds	(23 LOC)
rwDBMetaData	getPrimaryKeys	(5 LOC)
rwDBMetaData	getPrimaryKeys	(7 LOC)
rwDBMetaData	readPKeys	(21 LOC)
rwDBMetaData	getIndexInfo	(5 LOC)
rwDBMetaData	getIndexInfo	(7 LOC)
rwDBMetaData	readIndexes	(31 LOC)

Java Source: rwMDColumnInfo (82 LOC)		
Number of Classes: 1	Number of Methods: 37	rwMDColumnInfo (82 LOC)
rwMDColumnInfo	getTableCat	(1 LOC)
rwMDColumnInfo	getTableSchema	(1 LOC)
rwMDColumnInfo	getTableName	(1 LOC)
rwMDColumnInfo	getColumnName	(1 LOC)
rwMDColumnInfo	getDataType	(1 LOC)
rwMDColumnInfo	getTypeName	(1 LOC)
rwMDColumnInfo	getColumnSize	(1 LOC)
rwMDColumnInfo	getDecimalDigits	(1 LOC)
rwMDColumnInfo	getNumPrecRadix	(1 LOC)
rwMDColumnInfo	getNullable	(1 LOC)
rwMDColumnInfo	getRemarks	(1 LOC)
rwMDColumnInfo	getColumnDef	(1 LOC)
rwMDColumnInfo	getSQLDataType	(1 LOC)
rwMDColumnInfo	getSQLDateTimeSub	(1 LOC)
rwMDColumnInfo	getCharOctetLength	(1 LOC)
rwMDColumnInfo	getOrdinalPosition	(1 LOC)
rwMDColumnInfo	getIsNullable	(1 LOC)
rwMDColumnInfo	setTableCat	(1 LOC)
rwMDColumnInfo	setTableSchema	(1 LOC)
rwMDColumnInfo	setTableName	(1 LOC)
rwMDColumnInfo	setColumnName	(1 LOC)
rwMDColumnInfo	setDataType	(1 LOC)
rwMDColumnInfo	setTypeName	(1 LOC)
rwMDColumnInfo	setColumnSize	(1 LOC)
rwMDColumnInfo	setDecimalDigits	(1 LOC)
rwMDColumnInfo	setNumPrecRadix	(1 LOC)
rwMDColumnInfo	setNullable	(1 LOC)
rwMDColumnInfo	setRemarks	(1 LOC)
rwMDColumnInfo	setColumnDef	(1 LOC)
rwMDColumnInfo	setSQLDataType	(1 LOC)
rwMDColumnInfo	setSQLDateTimeSub	(1 LOC)
rwMDColumnInfo	setCharOctetLength	(1 LOC)
rwMDColumnInfo	setOrdinalPosition	(1 LOC)
rwMDColumnInfo	setIsNullable	(1 LOC)
rwMDColumnInfo	rwMDColumnInfo	(2 LOC)
rwMDColumnInfo	toString	(11 LOC)
rwMDColumnInfo	SQLdef	(15 LOC)

Java Source: rwMDColumnPrivsInfo (34 LOC)		
Number of Classes: 1	Number of Methods: 18	rwMDColumnPrivsInfo (34 LOC)
rwMDColumnPrivsInfo	getTableCat	(1 LOC)
rwMDColumnPrivsInfo	getTableSchema	(1 LOC)
rwMDColumnPrivsInfo	getTableName	(1 LOC)
rwMDColumnPrivsInfo	getColumnName	(1 LOC)
rwMDColumnPrivsInfo	getGrantor	(1 LOC)
rwMDColumnPrivsInfo	getGrantee	(1 LOC)
rwMDColumnPrivsInfo	getPrivilege	(1 LOC)
rwMDColumnPrivsInfo	getIsGrantable	(1 LOC)
rwMDColumnPrivsInfo	setTableCat	(1 LOC)
rwMDColumnPrivsInfo	setTableSchema	(1 LOC)
rwMDColumnPrivsInfo	setTableName	(1 LOC)
rwMDColumnPrivsInfo	setColumnName	(1 LOC)
rwMDColumnPrivsInfo	setGrantor	(1 LOC)
rwMDColumnPrivsInfo	setGrantee	(1 LOC)
rwMDColumnPrivsInfo	setPrivilege	(1 LOC)
rwMDColumnPrivsInfo	setIsGrantable	(1 LOC)
rwMDColumnPrivsInfo	rwMDColumnPrivsInfo	(2 LOC)
rwMDColumnPrivsInfo	toString	(6 LOC)

Java Source: rwMDIndexInfo (55 LOC)		
Number of Classes: 1	Number of Methods: 29	rwMDIndexInfo (55 LOC)
rwMDIndexInfo	getTableCat	(1 LOC)
rwMDIndexInfo	getTableSchema	(1 LOC)
rwMDIndexInfo	getTableName	(1 LOC)
rwMDIndexInfo	getNonUnique	(1 LOC)
rwMDIndexInfo	getIndexQualifier	(1 LOC)
rwMDIndexInfo	getIndexName	(1 LOC)
rwMDIndexInfo	getType	(1 LOC)
rwMDIndexInfo	getOrdinalPosition	(1 LOC)
rwMDIndexInfo	getColumnName	(1 LOC)
rwMDIndexInfo	getAscOrDesc	(1 LOC)
rwMDIndexInfo	getCardinality	(1 LOC)
rwMDIndexInfo	getPages	(1 LOC)
rwMDIndexInfo	getFilterCondition	(1 LOC)
rwMDIndexInfo	setTableCat	(1 LOC)
rwMDIndexInfo	setTableSchema	(1 LOC)
rwMDIndexInfo	setTableName	(1 LOC)
rwMDIndexInfo	setNonUnique	(1 LOC)
rwMDIndexInfo	setIndexQualifier	(1 LOC)
rwMDIndexInfo	setIndexName	(1 LOC)
rwMDIndexInfo	setType	(1 LOC)
rwMDIndexInfo	setOrdinalPosition	(1 LOC)
rwMDIndexInfo	setColumnName	(1 LOC)
rwMDIndexInfo	setAscOrDesc	(1 LOC)
rwMDIndexInfo	setCardinality	(1 LOC)
rwMDIndexInfo	setPages	(1 LOC)
rwMDIndexInfo	setFilterCondition	(1 LOC)
rwMDIndexInfo	rwMDIndexInfo	(2 LOC)
rwMDIndexInfo	SQLdef	(3 LOC)
rwMDIndexInfo	toString	(9 LOC)

Java Source: rwMDTableInfo (24 LOC)		
Number of Classes: 1	Number of Methods: 12	rwMDTableInfo (24 LOC)
rwMDTableInfo	getTableCat	(1 LOC)
rwMDTableInfo	getTableSchema	(1 LOC)
rwMDTableInfo	getTableName	(1 LOC)
rwMDTableInfo	getTableType	(1 LOC)
rwMDTableInfo	getRemarks	(1 LOC)
rwMDTableInfo	setTableCat	(1 LOC)
rwMDTableInfo	setTableSchema	(1 LOC)
rwMDTableInfo	setTableName	(1 LOC)
rwMDTableInfo	setTableType	(1 LOC)
rwMDTableInfo	setRemarks	(1 LOC)
rwMDTableInfo	rwMDTableInfo	(2 LOC)
rwMDTableInfo	toString	(5 LOC)

Java Source: rwMDTablePrivsInfo (31 LOC)		
Number of Classes: 1	Number of Methods: 16	rwMDTablePrivsInfo (31 LOC)
rwMDTablePrivsInfo	getTableCat	(1 LOC)
rwMDTablePrivsInfo	getTableSchema	(1 LOC)
rwMDTablePrivsInfo	getTableName	(1 LOC)
rwMDTablePrivsInfo	getGrantor	(1 LOC)
rwMDTablePrivsInfo	getGrantee	(1 LOC)
rwMDTablePrivsInfo	getPrivilege	(1 LOC)
rwMDTablePrivsInfo	getIsGrantable	(1 LOC)
rwMDTablePrivsInfo	setTableCat	(1 LOC)
rwMDTablePrivsInfo	setTableSchema	(1 LOC)
rwMDTablePrivsInfo	setTableName	(1 LOC)
rwMDTablePrivsInfo	setGrantor	(1 LOC)
rwMDTablePrivsInfo	setGrantee	(1 LOC)
rwMDTablePrivsInfo	setPrivilege	(1 LOC)
rwMDTablePrivsInfo	setIsGrantable	(1 LOC)
rwMDTablePrivsInfo	rwMDTablePrivsInfo	(2 LOC)
rwMDTablePrivsInfo	toString	(6 LOC)

Java Source: rwMDBrowIdInfo (34 LOC)		
Number of Classes: 1	Number of Methods: 18	rwMDBrowIdInfo (34 LOC)
rwMDBrowIdInfo	getScope	(1 LOC)
rwMDBrowIdInfo	getColumnName	(1 LOC)
rwMDBrowIdInfo	getDataType	(1 LOC)
rwMDBrowIdInfo	getTypeName	(1 LOC)
rwMDBrowIdInfo	getColumnSize	(1 LOC)
rwMDBrowIdInfo	getBufferLength	(1 LOC)
rwMDBrowIdInfo	getDecimalDigits	(1 LOC)
rwMDBrowIdInfo	getPseudoColumn	(1 LOC)
rwMDBrowIdInfo	setScope	(1 LOC)
rwMDBrowIdInfo	setColumnName	(1 LOC)
rwMDBrowIdInfo	setDataType	(1 LOC)
rwMDBrowIdInfo	setTypeName	(1 LOC)
rwMDBrowIdInfo	setColumnSize	(1 LOC)
rwMDBrowIdInfo	setBufferLength	(1 LOC)
rwMDBrowIdInfo	setDecimalDigits	(1 LOC)
rwMDBrowIdInfo	setPseudoColumn	(1 LOC)
rwMDBrowIdInfo	rwMDBrowIdInfo	(2 LOC)
rwMDBrowIdInfo	toString	(6 LOC)

Java Source: rwMDPKInfo (33 LOC)		
Number of Classes: 1	Number of Methods: 15	rwMDPKInfo (33 LOC)
rwMDPKInfo	getTableCat	(1 LOC)
rwMDPKInfo	getTableSchema	(1 LOC)
rwMDPKInfo	getTableName	(1 LOC)
rwMDPKInfo	getColumnName	(1 LOC)
rwMDPKInfo	getKeySeq	(1 LOC)
rwMDPKInfo	getPKName	(1 LOC)
rwMDPKInfo	setTableCat	(1 LOC)
rwMDPKInfo	setTableSchema	(1 LOC)
rwMDPKInfo	setTableName	(1 LOC)
rwMDPKInfo	setColumnName	(1 LOC)
rwMDPKInfo	setKeySeq	(1 LOC)
rwMDPKInfo	setPKName	(1 LOC)
rwMDPKInfo	rwMDPKInfo	(2 LOC)
rwMDPKInfo	SQLdef	(3 LOC)
rwMDPKInfo	toString	(8 LOC)

Java Source: rwMDProcedureInfo (24 LOC)		
Number of Classes: 1	Number of Methods: 12	rwMDProcedureInfo (24 LOC)
rwMDProcedureInfo	getProcedureCat	(1 LOC)
rwMDProcedureInfo	getProcedureSchema	(1 LOC)
rwMDProcedureInfo	getProcedureName	(1 LOC)
rwMDProcedureInfo	getProcedureType	(1 LOC)
rwMDProcedureInfo	getRemarks	(1 LOC)
rwMDProcedureInfo	setProcedureCat	(1 LOC)
rwMDProcedureInfo	setProcedureSchema	(1 LOC)
rwMDProcedureInfo	setProcedureName	(1 LOC)
rwMDProcedureInfo	setProcedureType	(1 LOC)
rwMDProcedureInfo	setRemarks	(1 LOC)
rwMDProcedureInfo	rwMDProcedureInfo	(2 LOC)
rwMDProcedureInfo	toString	(5 LOC)

Bibliography

[1] R. Cattell, G. Hamilton and S. White *JDBC(TM) API Tutorial and Reference: Universal Data Access for the Java(TM) 2 Platform (2nd Edition).* Addison-Wesley, Reading, 1999

[2] C.J. Date. *An introduction to Database Systems VOL 1.* Addison-Wesley, Reading, 1988.

[3] M.H. Derksen. Re-engineering van het WDS. Technical report, Hogeschool Den Haag / ASZ BV R&D, 1997.

[4] R. Johnson E. Gamma, R. Helm and J. Vlissides. *Design Patterns.* Addison-Wesley, Reading, 1995.

[5] R. Veldema J. Maassen, R. van Nieuwpoort and E. Bal. Efficient Java RMI for parallel programming. Technical report, Vrije Universiteit,Amsterdam, 1999.

[6] C. Nester, M. Philippsen, and B. Haumache. A more efficient RMI for Java. Technical report, University of Karlsruhe, Germany, 1998.

[7] S.W. Ambler. *Building Object Applications That Work.* Cambridge University Press/SIGS Books 1998

[8] Tolk. *http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html.* JVM Languages Web Repository, 1999.