Development of a Database Abstraction Layer

Master Thesis by Eelco Heerschop, 2004
Supervised by: Drs. J. H, Jongejan & Ir. S. Achterop

RuG

Development of a Database Abstraction Layer

Master Thesis by Eelco Heerschop, 2004
Supervised by: Drs. J. H, Jongejan & Ir. S. Achterop

RuG

from *http://adodb.com:*

We are very sorry to say, we lost our entire database.

We have roughly 1000 unique and returning visitors per day and apologize to all of you for this interruption.

If you have any ideas please email me

jason@idma.com

from *http://msdn.microsoft.com:*

To add to the confusion, Microsoft introduced another data-access object model: ADO

Abstract

Databases play an important role in the current information based society and it will even grow more and more important in the near future.
This report describes an approach to create a more abstract model of databases and its connections in order to make database connections truly independent of the database management systems used.

The thesis consists of three parts. The first part of this thesis consists of the problem definition and an analysis of the most popular database access technologies used today. The second part describes the creation of an abstract model for accessing databases and the third and final part describes a practical implementation of this model in the Java programming language.

Before we proceed with the problem definition, we start with a small introduction to get more familiar with the world of databases and computer architectures.

Table of Contents

Introduction

TV, newspapers, magazines, Internet... In today's society, people tend to get overwhelmed by information. *Information*, the key property of today's society.
All those huge quantities of information must be managed in some way. Users of information sources want to be able to retrieve, store and search information. This can be achieved by databases or more correct (after all databases are basically just a place to put your stuff in): by database management systems (DBMS[1]). People use databases more and more, often without even knowing they are using them. If you visit an ATM, search the Internet or shop in your local supermarket, in all cases databases are involved. Therefore, databases are playing an increasingly important role. In earlier systems centralized architectures (see figure 1) were used, which operated on mainframe computers to provide processing for all the functions, including user applications, user interfaces, as well as all the DBMS functionality. The reason for using centralized systems was that most users accessed such systems via computer terminals that did not have any processing power and only provided display capabilities. Gradually, most users replaced their computer terminals with personal computers that had their own processing power. At first database management systems did not make use of this huge potential of processing power, but that changed with the introduction of a new architecture: the client-server architecture.

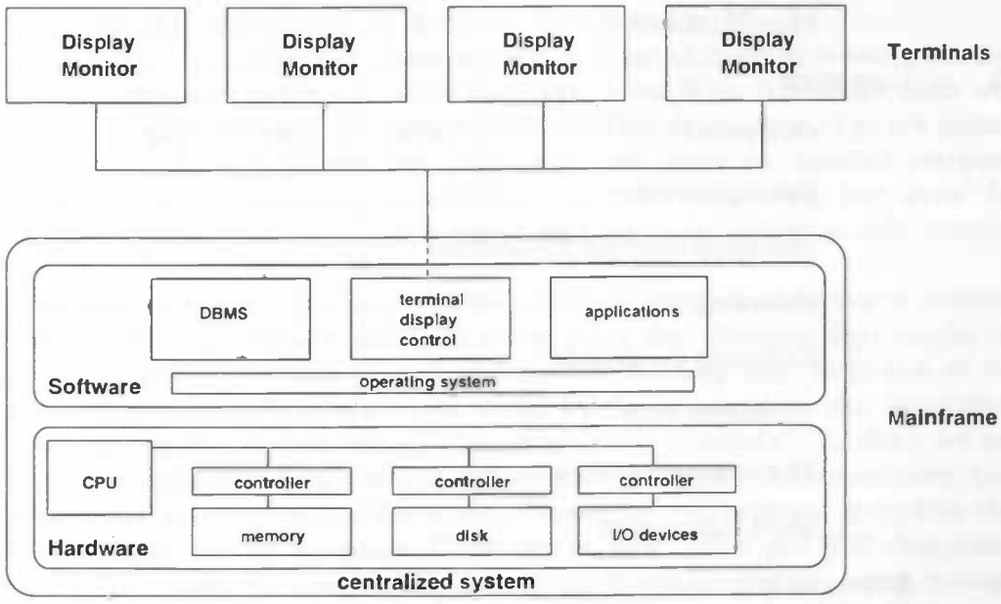


Figure 1: model of a centralized database architecture

[1] Throughout this thesis lots of abbreviations will be used. An extensive list of all abbreviations and its description can be found in the glossary.

The client-server architecture (see figure 2) is built on a framework of many computers connected via local networks and other types of computer networks. A client in this framework is a user machine that provides (graphical) user interfaces and local processing. A smaller number of computers in this framework are so called servers. A server is a machine that basically provides services to the client machines, such as printing, archiving, or database access.

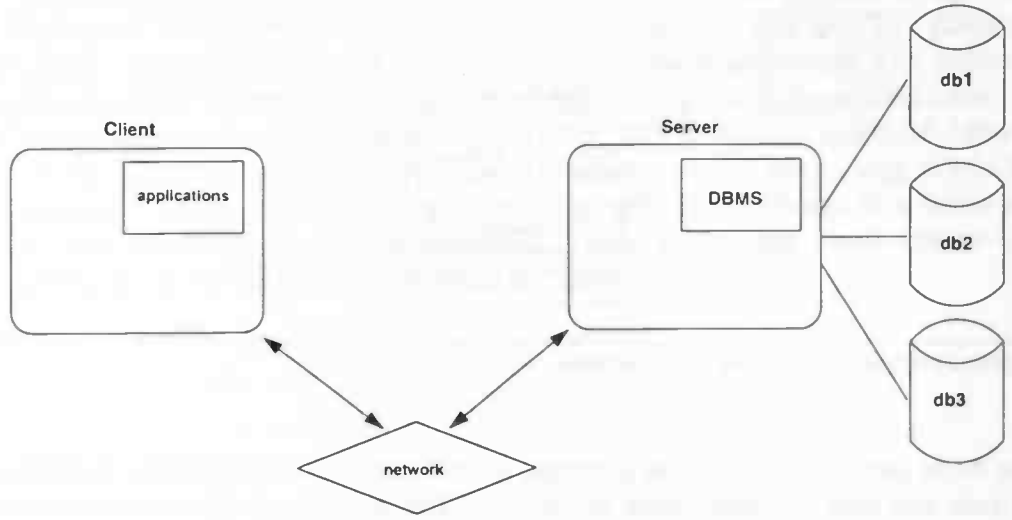


Figure 2: model of a client-server architecture

Although most systems are still based on the client-server architecture, we see another architecture rising in recent years: 3-tier architectures or multi-tier systems (see figure 3). In a 3-tier architecture functions are even more separated than in a client-server architecture. The client and server cannot connect with each other directly, but are separated by the middleware layer. Conceptually the middleware processes requests from the client and can pass them on to the (database) server. Responses from the (database) server are then interpreted again, and the filtered and formatted data is passed on to the client again. As one already might guess, the very existence of this architecture lies in the exponential growth of the Internet and the shift from static to dynamic web pages. If you look closer at a typical implementation of a 3-tier architecture, the client part of such an architecture is just a standard web browser like Mozilla or Internet Explorer, the (database) server side is some sort of DBMS and the middleware consists of a web server, usually Apache or IIS. Most web servers are capable of utilizing scripting languages like JSP, PHP and ASP that make it easy to connect with databases and render dynamic web content. One might take notice of the fact that the role of the client has become less important again. You can regard the client, which is only running a web browser, as a dumb client, just as it used to be in the first generation of database management systems.

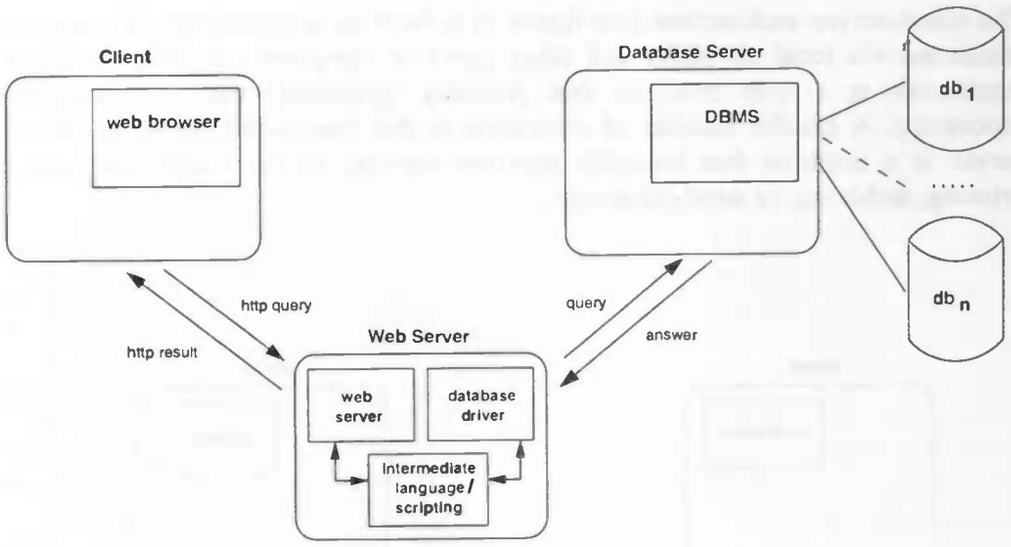


Figure 3: model of a 3-tier architecture

The 3-tier architecture arose as a merger of two technologies: database technology and network and data communication technology. The latter has made tremendous strides in terms of wired and wireless technologies including satellite and mobile communications and the standardization of protocols like Ethernet, TCP/IP and ATM.
Currently, we can see an immense increase in both the usage of dynamic content, but also in the absolute number of pages viewed on the World Wide Web. Nowadays, both web servers as well as web browsers are mature products. This development has not passed companies unnoticed. Web servers are used more and more on internal company networks, not only for providing information to their employees, but even for administration purposes. And an average employee cannot do without a web browser anymore.
Not that long ago the slogan – or vision – of the world's leading manufacturer of software products was: *"one PC at every desk"*. In recent years they changed their slogan to: *"Connect anything with everything"* and they are not the only one. Even one of the largest photo camera manufacturers in the world has been caught by the networking madness and advertises with this phrase: *"Imaging across networks"*. In short, we see that an increasing amount of data is stored and accessed using network connections. One of the more recent developments is the usage of databases by 'ordinary' programs that before did not make use of databases. For example, new office and IDE (Integrated Development Environments) suites make use of databases for storage, often without telling the end-users of those products. Databases might even be used by file system drivers in order to make fast storage, retrieval and searching of files possible on the increasingly growing capacity of hard drives.
Because current database architectures are not based on an all-in-one proprietary system, but are based on client-server or client-middleware-server architectures, methods are needed for accessing data from these database management systems. All DBMS vendors have developed their own methods for providing database connectivity including

complete proprietary languages, systems for creating user interfaces and APIs such as ODBC, JDBC, .NET and ADO/OLE DB.

One has to remark that DBMS vendors have the tendency to invent the most beautiful abbreviations and names for the simplest things. Sometimes they even give their own names to other (well known) products. For example: "*The **Oracle HTTP Server** provides a productive and high performance Web server environment for Oracle9i Application Server*", while in fact this so called 'special' web server is just the popular Apache web server. In the past, developers had to add code to their application that talks to a particular database using a proprietary language, but in recent years this has changed with arrival of non-proprietary APIs. We will discuss the benefits and drawbacks of these APIs later.

Today, we can connect our newly created application with a database in lots of different ways and every DBMS provides developers with their own way of connecting. While this is not necessarily wrong, we would like to interface with our databases in a better and easier way. We would like to use our databases in such a way that we do not have to know anything about the DBMS or even about the specific API used.

Part I, problem definition and analysis

Problem definition

We want to be able to connect to databases in an absolutely transparent manner, meaning that we are able to communicate with our databases in exactly the same way independent of the actual APIs and database management systems used. This goal can be realized by the creation of an abstract model of a database. Before we start with the creation of such a model, we first analyze a number of popular APIs for providing database connectivity followed by an analysis of more generic systems for sharing information. The second part describes the creation of an abstract model and the last part of this thesis describes a practical implementation of the created model in the Java programming language, which is suitable for both the client-server architecture as well as the 3-tier architecture.

Analysis of methods for database access

The obvious way to reach the goals described in the previous section is by inserting a new middleware layer. By middleware we mean software that connects two otherwise separate applications. Middleware serves as the glue between those applications. (See figure 3 where the web server acts as a middleware layer.) The first step in the development of our new middleware layer is taking a closer look at where we will insert it in. Middleware is nothing new: in most cases we will connect to some kind of middleware instead of the DBMS directly! Many different API's for connecting with databases have been developed, including ODBC, JDBC, .NET and ADO/OLE DB. Besides middleware that specifically has been developed to interact with databases, also more generic middleware exists. Examples are CORBA and Microsoft's Component Object Model. Both are examples of Distributed Object Systems. Distributed Object Systems are systems that allow applications to use objects on remote systems, as if the objects were local. In fact, in some systems there is absolutely no difference in programming for a local object versus a remote one. One might ask why a new middleware layer is needed. After all, there is enough to choose from. This is exactly *why!* We would like to have one generic abstraction, independent of the actual interface and version used. Vendors have the tendency to bring out new application programming interfaces or new versions of those interfaces every once in a while, making it hard for developers to keep their applications up to date. Another advantage of using one generic abstraction is the ability to bridge many different database connections over different application programming interfaces.
To get a bit more familiar with database connection technologies and Distributed Object Systems, we first take a look at a few popular application-programming interfaces, followed by a closer look at two examples of Distributed Object Systems.

ODBC

ODBC is the acronym for Open Data Base Connectivity, a Microsoft Universal Data Accessing standard that started life as the Windows implementation of the X/Open SQL Call Level Interface specification. Since its inception in 1992 it has rapidly become the industry standard interface for developing database independent applications. It is also the emerging standard interface for SQL based database engines replacing many of the first generation Embedded SQL and proprietary call level interfaces provided by database engine and database connectivity middleware vendors alike. [1-1]
In other words: ODBC is an open application-programming interface that allows developers to access a database in a predictable way. When writing code to interact with a database, you usually have to add code to your application that talks to a particular database using a proprietary language. If you want your program to talk to Access, SQL-Server and Oracle databases you have to code your program with three different database languages. However, when a developer uses ODBC, he only needs to talk the ODBC language (a combination of ODBC API function calls and the SQL language). The ODBC Manager will figure out how to connect with the database you want to use. From a programmer's point of view, databases are not directly available through the ODBC

driver, but only indirectly as so-called Data Sources that have to be configured in the ODBC Manager. Since version 2.0, the ODBC standard supports SAG SQL.
The listing below shows an example how database can be accessed using ODBC.

```
Dim conn As New rdoConnection
Dim rs AS rdoRecordset

'Attempt to create a database connection
conn.Connect = "Driver={SQL Server}; Server=MyServer; " & _
               "Database=Pubs; Uid=sa; Pwd="
conn.EstablishConnection

'Execute query
conn.Execute("SELECT * FROM Customers")

'While more rows exist, print customer id's
WHILE NOT rs.EOF
  Response.Write(rs(?CustomerID?).Value)
  rs.MoveNext()
END WHILE
```

Listing 1: code that uses ODBC in VisualBasic[1-2]

ADO DB

Another programming interface to access data in a database is ADO DB. ADO is a Microsoft technology that stands for ActiveX Data Objects and is – surprisingly - a Microsoft Active-X component. In terms of usage, ADO looks very similar to ODBC.
This comes very clear if we compare listing 1 and 2. However, ADO DB is focused more towards the creation of dynamic web content. This explains why ADO is automatically installed with Microsoft IIS.

```
Dim conn AS New ADODB.Connection()
Dim rs AS ADODB.Recordset

'Attempt to create a database connection
conn.Open("Provider=SQLOLEDB.1;UserID=sa;InitialCatalog=Northwind;
           DataSource=host;")

'Execute query
rs = conn.Execute("SELECT * FROM Customers")

'While more rows exist, print customer id's
WHILE NOT rs.EOF
  Response.Write(rs(?CustomerID?).Value)
  rs.MoveNext()
END WHILE

conn = Nothing
rs = Nothing
```

Listing 2: code that uses ADO in .NET[1-3]

JDBC

Java Database Connectivity (JDBC) provides Java developers with a standard API that is used to access databases. JDBC has capabilities to connect to a database and to retrieve the results of a query using the Java classes Connection, Statement and ResultSet respectively. JDBC tries to present a uniform interface to databases – after change of database management system your applications only need to change their driver. Unfortunately this is not entirely true, because of the different versions of the API and the different implementations of the actual JDBC drivers. Not all drivers act in the same way under certain circumstances. Typical differences between drivers lie in the methods connections with the DBMS are being made, errors are handled and result sets are accessed. Another important difference is the availability and correctness of the (meta) information about a DBMS and its databases.

JDBC driver implementations fit into one of four categories [1-4]:

1. A JDBC-ODBC bridge provides JDBC API access via one or more ODBC drivers. Note that some ODBC native code and in many cases native database client code must be loaded on each client machine that uses this type of driver. Besides the JDBC-ODBC bridge driver provided by Sun, there are also several commercial implementations.

2. A native-API (partly Java technology-enabled) driver converts JDBC calls into calls on the client API for Oracle, Sybase, Informix, DB2, or other DBMS. Note that, like the bridge driver, this style of driver requires that some binary code be loaded on each client machine.

3. A net-protocol (fully Java technology-enabled) driver translates JDBC API calls into a DBMS-independent net-protocol which is then translated to a DBMS protocol by a server. This net server middleware is able to connect all of its Java technology-based clients to many different databases. The specific protocol used depends on the vendor. In general, this is the most flexible JDBC API alternative. It is likely that all vendors of this solution will provide products suitable for Intranet use. In order for these products to also support Internet access they must handle the additional requirements for security, access through firewalls, etc., that the Web imposes. Several vendors are adding JDBC technology-based drivers to their existing database middleware products.

4. A native-protocol (fully Java technology-enabled) driver converts JDBC technology calls into the network protocol used by DBMSs directly. This allows a direct call from the client machine to the DBMS server and is a practical solution for Intranet access. Since many of these protocols are proprietary the database vendors themselves will be the primary source for this style of driver.

One major advantage of JDBC is the availability of plenty of drivers that support many popular databases. In many cases a JDBC driver that works specifically for one DBMS is available. If not, Sun provides a driver that is compatible with ODBC, so connections to any ODBC compliant DBMS should be possible. In recent years, commercial JDBC drivers have become available for connecting with ODBC compliant database management systems directly too.

Listing 3 shows an example of making a database connection and execution a query using JDBC.

```
// Attempt to load database driver
try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
} catch (ClassNotFoundException cnfe) {
    System.err.println ("Unable to load database driver: " + cnfe);
    System.exit(0);
}

// Now attempt to create a database connection
String url = "jdbc:odbc:localhost";
Connection conn = DriverManager.getConnection(url,"db","sql");

// Create a statement to send SQL
Statement stmt = conn.createStatement();

// Execute query
ResultSet rs = stmt.executeQuery("select * from Customers");

// While more rows exist, print customer id's
while (rs.next() )
{
    System.out.println ("ID : " + rs.getInt("CustomerID"));
}
```

Listing 3: code that uses JDBC in Java [1-5]

CORBA

Specified by the Object Management Group, CORBA is the acronym for Common Object Request Broker Architecture. CORBA is an architecture for an open software bus. The central component of this architecture is the Object Request Broker (ORB), on which object components written by different vendors can interoperate across networks and operating systems without knowing where the objects they access reside or in what language the requested objects are implemented. The ORB interacts and makes requests to those objects and negotiates between request messages from objects or object servers and the affiliated data sets. [1-6]

The most important part of the OMG[1-7] specification is the Interface Definition Language (IDL)[1-8] which is used to define the interfaces to CORBA objects. CORBA objects are quite different from typical programming language objects because CORBA objects can be located anywhere on a network and can operate with objects written in other languages or on other platforms.

Since CORBA was first introduced in 1991, two new versions of the specification followed. Nowadays several implementations are available including free (open source) implementations on a variety of platforms. However, not all implementations provide the same level of functionality and robustness.

The idea of sharing objects across networks, platforms and programming languages is not a bad idea at all. Therefore it is worthwhile to look a bit deeper inside CORBA.
CORBA applications are composed of objects, individual pieces of software that combine functionality and data. Typically, there are many instances of an object of a single class, all identical in functionality but differing in that each has its own state. For each class one has to define an interface in the Interface Definition Language.
Interface definitions are independent of programming language, but they do have to be compiled to the actual programming language used. Currently there are standardized mappings from the IDL to C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python, and IDLscript. Compiling an interface generates two implementations of the defined interface. The first one is the client stub, which is the interface used by applications to perform operations on the object and the second one is the object skeleton, which serves as a framework for the actual implementation of the object.
At the moment a client wants to make use of a CORBA object, it has to use the objects' IDL interface (implemented in the client stub) to specify which operations it wants to perform. Requests from the client are routed through the ORB, which uses the same IDL interface. When requests are processed by the object implementation, the results are sent back through the ORB using the same interface again. (See figure 4)

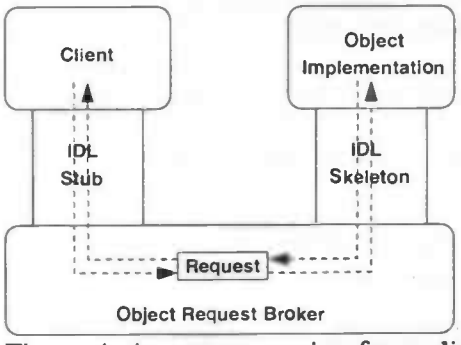


Figure 4: A request passing from client to object implementation

In CORBA, every object instance has its own unique object reference ID. Clients use the object references to direct their invocations, identifying to the ORB the exact instance they want to invoke.

An important feature of CORBA is its interoperability between ORBs. A request can be routed from one ORB to another ORB when the first ORB detects a request cannot be handled locally. This so-called 'remote invocation' is achieved by agreeing on a common protocol. Although other protocols could be used for this task, the OMG has defined a standard protocol for the ORB-to-ORB communication: IIOP.

From the client's point of view, remote object invocations are no different from normal object invocations. Because IIOP tunnels the request and results from and to the ORBs transparently, there is no need for a central server. Instead data and functionality can be distributed transparently across networks. (See figure 5)

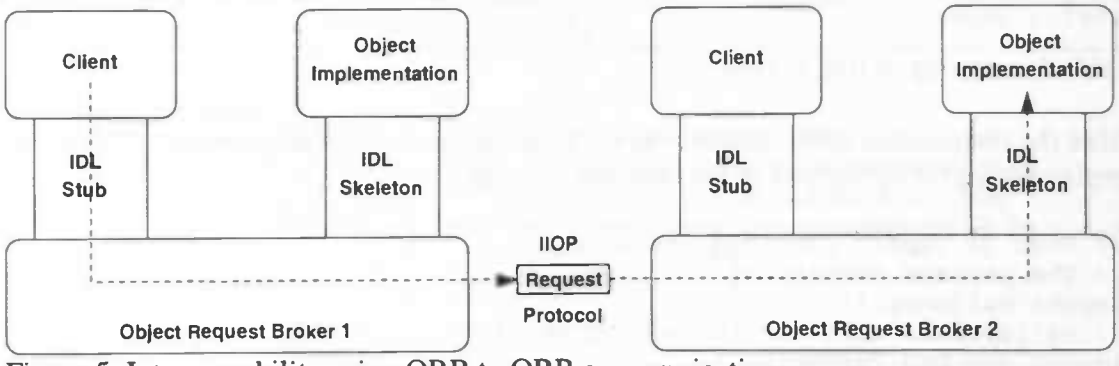


Figure 5: Interoperability using ORB to ORB communication

Developing CORBA applications is quite complicated and even the famous "HelloWorld" example requires a couple of files to be implemented.
The first step is the definition of the interface (listing 4).

```
module HelloApp
{
  interface Hello
  {
    string sayHello();
  };
};
```

Listing 4: definition of the interface for the HelloWorld example (Hello.idl) [1-9]

Next, the IDL compiler is used to generate the stubs and skeletons. As stated before, we can map the interface to different programming languages, but for simplicity we only perform a mapping to the Java language. To do this, we perform the command: idltojava Hello.idl which generates five files including a mapping of the interface to the Java programming language (listing 5).

```
package HelloApp;
public interface Hello extends org.omg.CORBA.Object {
    String sayHello();
}
```

Listing 5: the interface mapped to the Java programming language (Hello.java)

If we compare listings 4 and 5, we can easily see how the IDL statements map to the generated Java statements. (table 1)

IDL Statement	**Java Statement**
`module HelloApp`	`package HelloApp;`
`interface Hello`	`public interface Hello`
`string sayHello();`	`String sayHello();`

Table 1: mapping of IDL to java

After the compilation of the interface to the target programming language, we proceed by implementing the server part of the program (Listing 6)

```
// Step 1: import required packages
// The package containing our stubs.
import HelloApp.*;
// HelloServer will use the naming service.
import org.omg.CosNaming.*;
// The package containing special exceptions thrown by the name
service.
import org.omg.CosNaming.NamingContextPackage.*;
// All CORBA applications need these classes.
import org.omg.CORBA.*;
```

```
// Step 2: declare the server class:
public class HelloServer
{
  // Step 3: define the main() method:
  public static void main(String args[])
  {
    // Step 4: handle CORBA system exceptions
    try{

      // Step 5: create and initialize the ORB
      ORB orb = ORB.init(args, null);

      // Create the servant and register it with the ORB
      HelloServant helloRef = new HelloServant();
      orb.connect(helloRef);

      // 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("Hello", " ");
      NameComponent path[] = {nc};
      ncRef.rebind(path, helloRef);

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

    } catch(Exception e) { // Step 4, continued
        System.err.println("ERROR: " + e);
        e.printStackTrace(System.out);
      }
  }
}


//Step 6: manage the Servant object
//        note: the HelloServer needs a HelloServant.
//              The servant implements the interface generated by
//              idltojava and actually performs the work of the
//              operations on that interface. The servant will be
//              instantiated by the HelloServer
class HelloServant extends _HelloImplBase
{
  public String sayHello()
  {
    return "\nHello world !!\n";

  }
}
```

Listing 6: implementation of the server (HelloServer.java) [1-10]

The final step in our sample implementation is the creation of the client:

```
import HelloApp.*;             // The package containing our stubs.
import org.omg.CosNaming.*;    // HelloClient will use the naming srv.
import org.omg.CORBA.*;        // Required CORBA classes.


public class HelloClient
{
  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("Hello", " ");
      NameComponent path[] = {nc};
      Hello helloRef = HelloHelper.narrow(ncRef.resolve(path));

      // Call the Hello server object and print results
      String Hello = helloRef.sayHello();
      System.out.println(Hello);

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

Listing 7: implementation of the client (HelloClient.java) [1-11]

After compilation of the source files, we can finally test-drive our application by starting the client and server. The output of the client should be: "Hello World!!" (This must be the most complicated "Hello World" in the world!)

On paper CORBA looks very promising and well suited for our needs, but nevertheless there are several disadvantages:

- Complex – because of the fact that CORBA tries to be entirely platform and language independent relying on the Interface Definition Language, the interfaces and generated stubs and skeletons can become really complex. The development time is therefore rather high too.
- Slow improvement – committees generally tend to move slowly, from specification to implementation currently takes 18 months.
- Cost – although some free implementations do exist, most are not free at all. On the contrary: ORBs for various platforms are very expensive.

We can conclude that platform independence and cross platform communication is surely possible with CORBA, but at a high price: rigid structures and a long development time.

DCOM

DCOM is an extension of Microsoft's Component Object Model (COM) that allows interaction between objects executing on separate hosts in a network. The original COM model provides a framework for dynamically integrating components that interact within a single address space or between processes on a single host. The implementations of these components were packaged in Dynamic Link Libraries (DLLs). COM is essentially a binary integration scheme. It adopted the structure of C++ virtual function tables as the binary representation of an interface. COM defines an API to allow for the creation of components for use in integrating custom applications or to allow diverse components to interact. However, in order to interact, components must adhere to a binary structure specified by Microsoft. As long as components adhere to this binary structure, components written in different languages can interoperate [1-12]. A lot of Microsoft's client services such as Microsoft Transaction Server (MTS), ActiveX and OLE depend on the COM infrastructure as shown in figure 6.

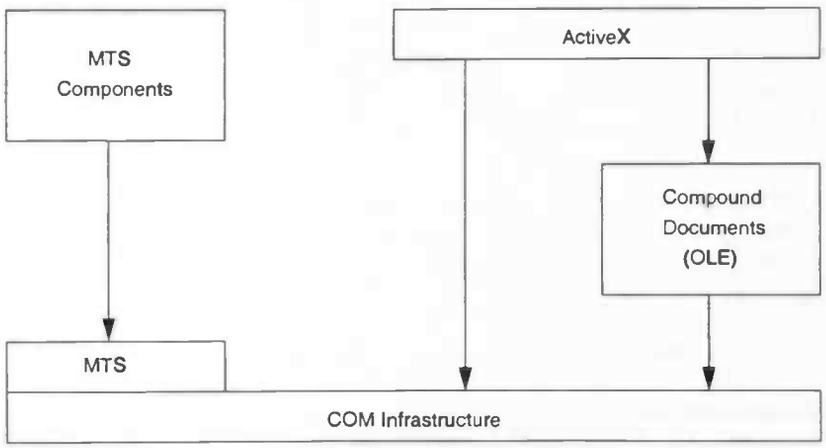


Figure 6: Component Object Model Architecture

DCOM shares a lot of similarities with CORBA. Both provide interface definition languages, but their roles are fundamentally different. While the CORBA IDL is rigorously defined and a fundamental part of CORBA, Microsoft's IDL is only a 'tool'. It is only one of several ways to define COM interfaces [1-13].
COM is a typical example of a desktop architecture and extending it to an enterprise architecture is problematic. Over the years COM has changed dramatically in its evolution from compound documents to distributed objects. The changes in design have not been graceful and really show the ad-hoc nature of Microsoft's solutions. COM is fundamentally not a well-partitioned architecture and relies on a key optimization for a single language and platform.

Part II, design

A solution by abstraction

Before we proceed with a possible solution for our problem we summarize the first chapters briefly. We have seen that there are several APIs for accessing databases. Access can be either directly or through some form of middleware (for example: through an ODBC manager). We have also seen that there are more generic ways of exchanging information. In this case we do not talk about sharing data, but about distributing components or objects that include both functionality as well as data. An example of an architecture that uses the principles of the distribution of objects is CORBA. We have also mentioned before that we want to achieve our goals by two key concepts: abstraction and middleware.
Abstraction is an important aspect, because it enables us to hide actual implementations and APIs from the outside world. The solution will be in the form of middleware, because the new software layer will be positioned right in the *middle* of user-applications and DBMS drivers, gluing everything together.

Now, let's take a closer look at the position where our new middleware fits in. If we look at figure 7 below, we can see our "*database abstraction layer*" positioned right between the application and the middleware supplied by DBMS vendors. Therefore, the developers of a database-connected application don't have to pay attention to a specific DBMS or interface anymore.

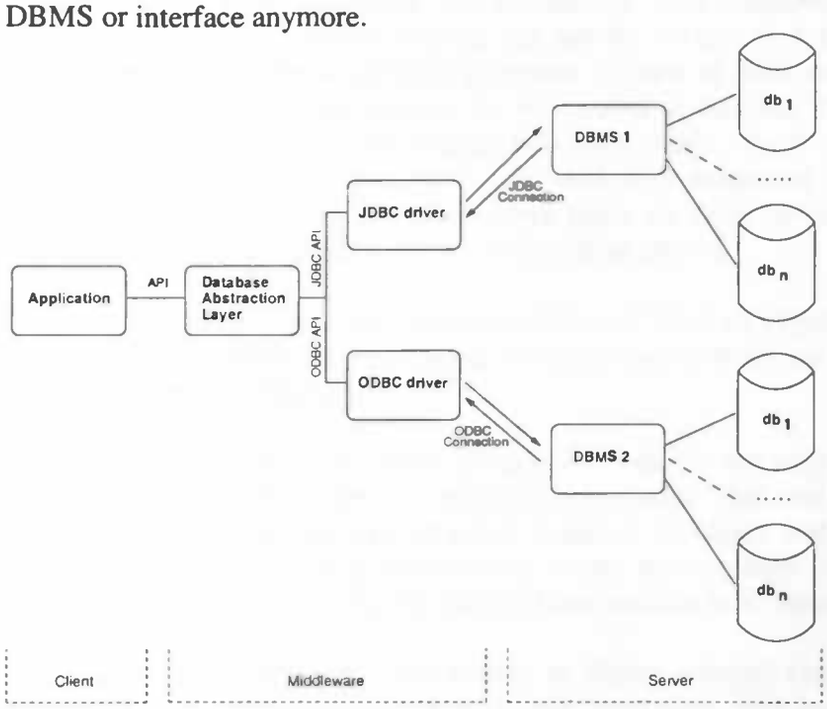


Figure 7: overview of a DAL connected application

So far, we have looked at different database connection technologies and at the place where our new middleware will be positioned. If we compare the examples above, one has to remark that the APIs all look very similar. Of course, we can only use one programming language at the time, but still, a truly independent interface for connecting with databases, regardless of the database drivers and DBMS used should be possible.

An Object Oriented Approach

Instead of just creating a new plain function-oriented application-programming interface, we are going for another approach: the object-oriented approach. Object-oriented programming has its roots in the SIMULA programming language. The version of SIMULA that was introduced in 1967 offered most of the key concepts of object-oriented programming such as objects, classes and subclasses. In SIMULA, the concept of a class groups together the internal data structures of an object as well as the implementation of its functionality. The next step in the development of object-oriented languages was SMALLTALK. This language developed at Xerox PARC in the 1970s was the first real programming language explicitly designed to be object-oriented. The introduction of C++ in the early 1980s extended C with the key concepts of SIMULA. However, the first commercial version of C++ became only available in 1995 and it took until 1998 before an ISO standard version became available [u15]. Probably the best example of a present-day object-oriented language is Java. In 1991 James Gosling worked on embedded systems software at Sun Microsystems. Because of the frustrations he had working with C++ he began developing a language called "Oak" to be a safe, object-oriented systems language. Although he was frustrated working with C++, the resulting language was still strongly influenced by it. By 1993 this language had been renamed to Java and several prototype devices using Java where available. The market however didn't seem to be interested. Around this time, the usage of the Worldwide Web grew extensively and Sun began to see uses for Java's small, safe and platform independent code in the internet community. Today many people still associate Java with the internet. The first official 1.0 release was in 1995 and Java has continually been improved and adapted at new requirements. At the time of writing, the newest version is 1.4. Java is different from other languages in the sense that programs written in Java on one hand have to be compiled and on the other have to be interpreted at runtime. This is because the Java compiler does not translate the program to machine code, which can be executed directly by the computer, but to Java byte code. Although somewhat slow, this code can be executed by a Java interpreter. Nowadays there are Java interpreters for almost every platform imaginable including PDA's and cellular phones.

It is very difficult to give an exact description of what an object-oriented programming language is. According to Webopedia, an online encyclopedia on computer related terms, object oriented programming is [2-1]:

"A type of programming in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure. In this way, the data structure becomes an object that includes both data and functions. In addition, programmers can create relationships between one object and another. For example, objects can inherit characteristics from other objects."

Thus, according to the definition above, an object-oriented language must support the programmer in such a way that both data structures and operations on those data structures can be defined. I think this is a rather limited definition, because this makes nearly every programming language a *potential* object-oriented language.

Bjarne Stroustrup, the developer of C++ states [2-2]:

"If the term "object-oriented programming language means anything it must mean a programming language that provides a mechanism that supports the object-oriented style of programming as well."

Maybe it is better to state that object-oriented programming languages are only those languages that *actively* support object oriented programming.
This does not mean that every program written in an object-oriented language is an object-oriented program. Moreover [2-3]:

- A design can be Object-oriented, even if the resulting program isn't.
- A program can be Object-oriented, even if the language it's written in isn't

But what then makes programming languages object-oriented? What characteristics must a programming language have to *actively* support object-oriented programming? Summarizing [2-4, 2-5, 2-6], the following features should be supported by an object-oriented programming language:

- Autonomous entities called objects
- Interaction by messaging, with no assumption of implementation
- Object organization, including an inheritance mechanism
- Programs as models

Objects are autonomous software entities modeled after real-world objects in the sense that they both combine state and behavior. An object's state is maintained in its variables and its behavior is implemented in its methods.

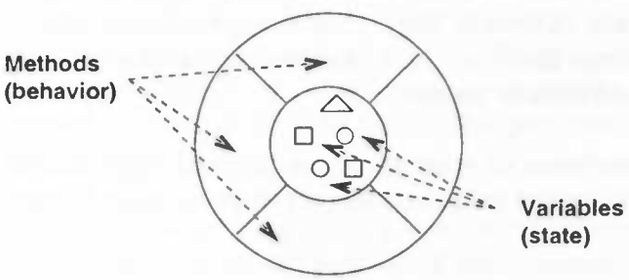


Figure 8: visual representation of a software object [2-7]

A single object does not offer much more functionality compared to a normal program. However, it gets more interesting if we let several objects communicate with each other. Objects are able to interact with each other by sending messages. Because the object's methods define its behavior, message passing supports all possible interactions between objects without direct manipulation of the variables. Message passing makes it possible for objects to communicate even if they do not reside in the same process or even on the same machine.

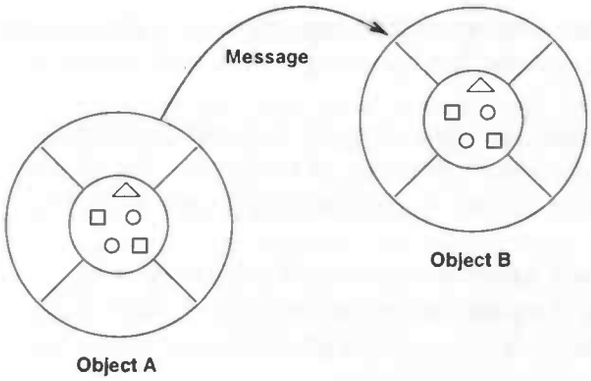


Figure 9: messaging between objects [2-8]

An object has a public interface that other objects can use to communicate with it. But the object can maintain private information and methods that can be changed at any time without affecting other objects that depend on it. Other objects do not have to know how an object is implemented in order to use it. Thus, abstraction, an important element of our *abstract database layer*, arises naturally with object-orientated programming.

Objects are organized in hierarchical structures and a proper inheritance mechanism should be made available by the programming language. Inheritance provides a powerful and natural mechanism for organizing and structuring software programs. In object-oriented programming inheritance means the ability of one object to be defined in terms of other objects. Although each descendant inherits both state and behavior of its ancestor, new state and behavior can be added easily and inherited methods can also be overridden.

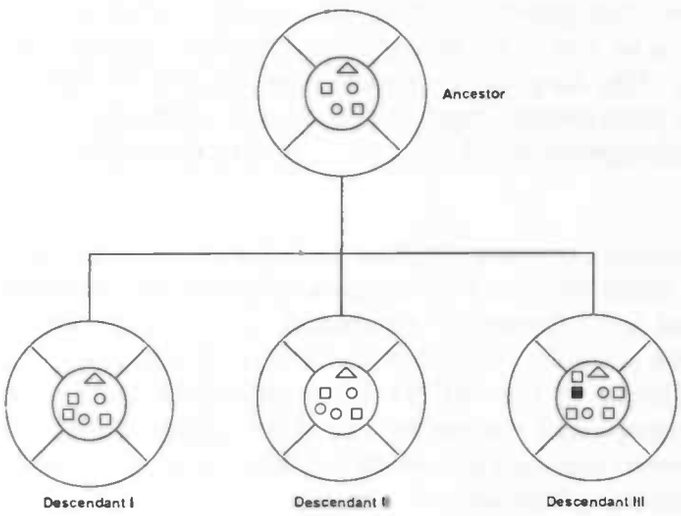


Figure 10: schematic representation of the inheritance mechanism [2-9]

Using the inheritance mechanism, programmers can create so called *abstract classes* that define generic behavior. An abstract superclass defines and partially implements the

behavior of the object. Descendants of this superclass define and implement the missing parts. Thus, inheritance takes abstraction to the next level, making it even more powerful.

Today object-oriented concepts are applied in the area of software engineering, knowledge bases, artificial intelligence and even in the area of databases. The Object-oriented approach has several advantages over the function-oriented approach including:

1. Data abstraction: The details of a class's representation are visible only to its methods; different implementations of a class can be used with no changes to the code that uses that class. (Data abstraction is not unique to object orientation but arises naturally with it)
2. Compatibility: Heuristics for constructing classes and their interfaces make it easier to combine software components.
3. Reuse: combining methods with data representations to construct classes, makes it easier to reuse existing code.
4. Extensibility: Software built using object-oriented techniques tends to be easier to extend. There are two reasons for this: Inheritance enables new classes to be built from old ones, while still participating in all the original relationships; and the classes form a loosely coupled structure that is easier to modify.
5. Maintenance: The natural modularity of the class structures make it easier to contain the effects of changes, and the use of inheritance reduces the number of disparate concepts to understand the code.

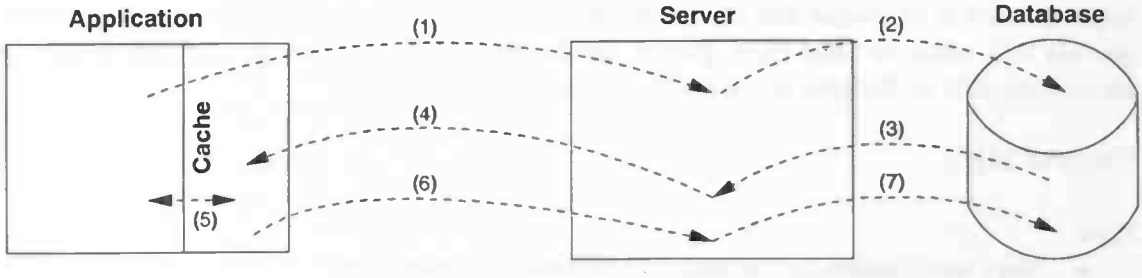
We can conclude that these basic concepts of object-oriented programming are able to provide a really powerful framework for the implementation of our database abstraction layer. Of course, choosing an object-oriented approach, also introduces a few drawbacks. For example, function-oriented methods seem to be inappropriate. This might be a serious problem, because database management systems are usually equipped with function-oriented interfaces. There is no way to get around this problem in a proper way, but fortunately, Madsen states [2-3]: "Thinking object-oriented does not have to exclude functional expressions when that is more natural. Functions, types and values are in fact needed in order to describe measurable properties of objects." ...And I cannot agree with him more!

In the past few years quite a few object-oriented database management systems have been developed. As the number of object-oriented DBMSs grew, the need for a standard language and model was recognized and a consortium of vendors and users proposed a standard called ODMG-93[2-10] which gradually evolved to the ODMG 3.0 standard [2-11, 2-12]. Unfortunately, after the completion of the ODMG 3.0 standard, the group was disbanded. Today, most database management systems are still of the relational kind. So, even if we choose for an object-oriented approach and we do just that, we have to support the function-oriented methods for accessing databases.

Designing an abstract database layer

Designing the new layer is a difficult task and we have to work very carefully making extensive use of the features object-oriented programming offers. Perhaps the most difficult part is how we perform the mapping from the (relational) database to the user application. One approach consists of extracting the relational data out of the database and mapping this data to objects. In this case the mapping should be preferably performed in such a way that the resulting objects are directly accessible and fully compatible with normal objects in the programming language used and operations on those objects should result on operations on the data in the database. Actual examples of such a higher-level abstraction are *'transparent persistent objects'* in the Java programming language [2-13, 2-14]. Whereas persistent data is information that can outlive the program that creates it, transparent persistence is the automatic storage and retrieval of persistent data. From a programmer's point of view, persistent objects are treated no differently than transient objects (or instances which only reside in memory and do not persist outside of an application). In Java, transparent persistence is available through the JDO interface. Although the specification is still under development, several preview implementations do exist. At a first glance this mechanism looks like a very good candidate for our relational/object mapping, but if we look a little bit closer at the specification we see that the emphasis is more towards mapping objects (or actually the state of objects) to databases and back again, instead of mapping existing databases to objects and back. There are layers build on top of JDO that actually do support 'reverse engineering' of existing databases. Examples of such layers are Cayenne [2-15] and KODO [2-16].

The question remains of course, whether these layers can or cannot manage huge quantities of data. For example: real life production databases can hold thousands or even millions of records and mapping all those records can be a very resource-consuming task. The makers of these layers claim the actually *can* do it using a caching mechanism (see figure 11). A somewhat extreme example of 'caching' is Prevayler [2-17]. This software layer features transparent persistence of Java objects, but instead of mapping and caching objects to RAM if needed, it keeps a copy of *all* objects in RAM at all times.



1) request from application
2) server queries database
3) results are passed to server
4) server passes result to the application cache
5) application reads/writes to the cache
6) changes are passed to the server
7) the server applies the changes to the database

Figure 11: caching of transparent persistent objects

One problem with caching is the introduction of serious concurrency issues. Another problem is the size of the RAM. Most DBMSs contain more data than fits in RAM.
The database to object mapping is performed by mapping a table from the database to a class in the object-oriented programming language. Every record in the table is mapped to an instance of the class. One might ask how records can be compared, manipulated and sorted. One way is by using the standard programming techniques, which however can be very costly. Another approach consists of making use of some special kind of database query language. In this particular case queries can be done with 'JDO Queries'.

Perhaps it is better to take a slightly different approach. In my view it is better not to model the data itself, but to model only a representation of the data. For example: if we want to create a table of all customers in a customer database, we first define an object 'table' which will contain properties like the number of columns, and the column names. We also define methods how we can retrieve records or even single items from the database. What we do not want to do is to copy records from the database to the table object.

Besides connecting to a DBMS and being able to map our data, we also have to determine how we retrieve the data we want to have. If we had chosen for transparent persistent objects we just could have used our programming language. At first sight this seems to be a very elegant approach. However, in practice, it can be a very difficult and time-consuming task to find and sort the appropriate records. However, since we use a different approach, we have to choose either for some sort of self defined language like 'JDO queries' or for the general accepted database language SQL. Nowadays, almost every DBMS does support some form of SQL. Unfortunately, there are still differences in the various SQL dialects and while some vendors have added their own functionality, others do not even support all the standard SQL functions.
Choosing for a query language to be newly defined also introduces serious problems. Besides defining a new language, we also have to parse the input and add the appropriate actions to our defined commands (probably in the form of SQL queries). The resulting queries are not the same for all DBMSs. Therefore every driver has to implement the appropriate actions. This does not mean that it is not possible, but the final result undoubtedly will be much slower and only very basic operations will be available. The major benefit is of course that the query language can be very strict and small and that all queries will result in valid SQL-queries for the target DBMS. Perhaps it is best to give a small overview of the pros and cons of both approaches:

Usage of SQL:

Pros:

- Very well known by the majority of database developers
- Short development time for the software layer

Cons

- DBMS vendors have added their own functions
- Some standard ANSI SQL functions are not implemented in all DBMSs

Usage of self-defined language:

Pros:

- small language
- strict non ambiguous definition

Cons

- limited functionality
- slower
- special DBMS specific features cannot be used
- longer development time

Eventually I decided to use SQL as the query language for the drivers. The primary reason is simple: it had to work in practice. People are familiar with SQL and they know that it works. They just wouldn't be happy working with some kind of self-defined language, how simple, straightforward and unambiguous it might be. Moreover, using another language might imply that not all functions a DBMS offers are available. While these products cost serious amounts of money, it is simply not explainable why only a 'crippled' version can be used. The question remains how we solve the differences between the various SQL dialects. The answer is very simple: ***we don't***.
What we are going to do is to give the user two different kinds of access modes: a native mode, which allows the user to talk to the DBMS directly with all features the DBMS offers and a compatibility mode which offers only a basic subset of the SQL-language.
Furthermore we can try to solve compatibility problems by supplying the 'abstract database' with extra information. This extra information or 'metadata' tells what a particular DBMS ***can*** or ***cannot*** do and ***how*** certain things should be done.

System overview

Now we know more about the object-oriented approach and the way we want to access data in the database we can elaborate our architecture depicted in figure 7. Figure 7 shows the "*database abstraction layer*" as one component positioned between the application and the database connection technologies. We divide this component into five smaller portions:

- Database Abstraction Layer – manages all connections
- Abstract Database – abstract object representing a database
- Database Driver – levels differences between the abstract database and the actual driver
- Database Table Model – returns a table model of the results of a query
- Database Tree Model – returns a tree model of the database

These components will be explained in detail later, but first we give new overview of the entire architecture (figure 12).

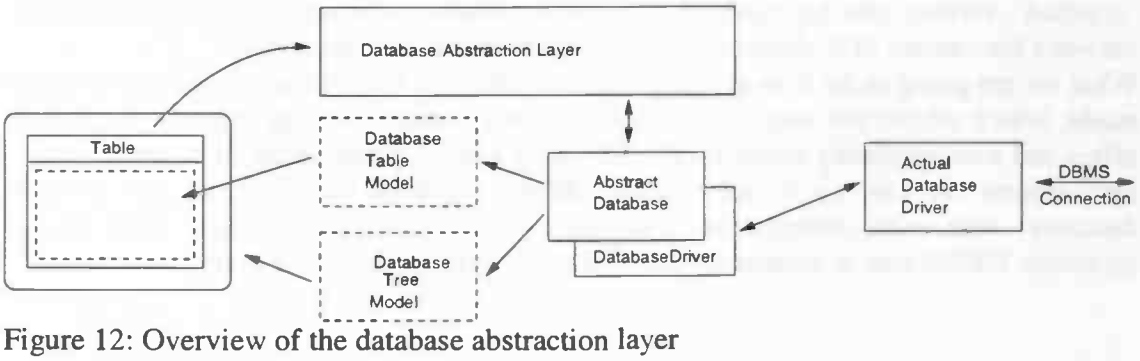


Figure 12: Overview of the database abstraction layer

The Database Abstraction Layer

The heart of the system is the database abstraction layer class. This is the class that manages all the abstract databases currently in use or stored for later usage. Through the database abstraction layer users are able to connect with databases (without specifying a specific DBMS), register new database drivers, and select the default database to use. The methods defined for this class are:

- open(), try to open a database connection with the given properties
- close(), close ***all*** database connections
- store(), store a specified connection for later usage
- select(), select the default database to use
- get(), get the current selected database
- registerDriver(), registers a new database driver

The registerDriver() method registers a new database driver. (In this context a database driver is an implementation of the abstract database class) Because all API and DBMS specific code are linked to such a database driver and we are able to load it when needed, we are able to save memory and plug-in new drivers as required.

The Abstract Database

The abstract database object is the key component of the database abstraction layer. This object will be implemented as an *abstract* class. This abstract class defines the common representation and behavior of all the database drivers. The database drivers implement the actual connection with the database connection API's which in their turn connect with the DBMS. The most important methods to be defined in the abstract database class are:

- open(), open the connection with the database
- close(), close the connection with the database
- query(), query the database
- update(), insert data in the database
- toString(), returns a human readable description of the database

Besides the methods above, we also need some methods to set and retrieve properties like the hostname of the database server, the port number and the username.

The Database Table Model

Besides the basic functions for opening and closing connection and the handling of data, we also want to be able to represent the data in the database. Therefore we create a model of the data we want to represent. Unless it is absolutely necessary, we do not want to manage the actual data, but we only create a way to *represent* the data. This means we are able to tell how a cell is defined, how many columns are used, what the column titles are and how we can retrieve a specific cell. The way we do this is tightly coupled to the used programming language and accompanying libraries and as such, a more comprehensive description of its internals can be found in the implementation part of this thesis.

The Database Tree Model

Both developers and users of databases are usually interested in what databases can offer them. With a database tree model, users of the abstraction layer have a simple but very efficient mechanism to retrieve everything they possibly want to know. The database tree model models all information including databases, tables, columns and its accompanying properties, in a tree structure that is easily accessible and searchable. One advantage of a tree structure is that it also can be presented to end-users very well.

One might ask how all this information can be retrieved. There is no driver independent approach for this. Some DBMSs support special functions like *getCatalogs()* or *getSchemas()* others support special SQL queries. Therefore the database driver, the device dependent implementation part of an abstract database, defines how we should do this. When a database tree model is created, the returned object will be compatible with the used GUI. This makes it possible to create a visual representation of the tree structure with just a single statement.

Part III, a practical implementation

Choosing a programming language and programming tools

Before I could start with the implementation of the abstraction layer I had to decide over the programming language and tools to use. I chose the Java programming language for several reasons including:

- Time – the implementation had to be realized in a reasonable amount of time.
- Database support – Java already supports the use of databases out-of-the box in the form of JDBC. Besides database drivers no extra libraries of third parties are required.
- Driver availability - for most DBMSs (free) JDBC drivers are available.[1]
- Platform independence – programs written in Java can be used on various platforms without recompilation.[2]
- Language constructs – the Java programming language supports all features we need for a truly object-oriented implementation.

1) There are different versions of JDBC. The implementation must be able to handle those differences. Besides JDBC we have to make sure ODBC drivers can be used too.
2) Some database drivers are NOT platform independent and use the Java JNI interface to load themselves. These drivers are mainly commercial ones.

Although my previous experiences with IDEs for the Java programming language, such as Jdesigner and Jdeveloper, where not so positive at all, it seemed a wise decision to make use of an IDE after all, because of the size of this project. The company where I followed my internship had used IBM Visual Age before and because of its availability I decided to use it too. Visual Age is a much more stable IDE than the ones I used before and I have not experienced any memory leaks or instability of any kind. The IDE also includes a pretty good versioning system. A free version of this IDE is available for download at the WWW [3-1]. Figure 1 shows an example of how the IDE looks like. A major advantage of this IDE is its ability to use visual composition for the user interfaces. (See figure 2) For the database abstraction layer this is not an issue at all, but for user applications this can be a big benefit. The DatabaseExplorer, a test application that is going to be build on top of the database abstraction layer, will be made using the visual composition editor of the Visual Age IDE.

Figure 1: example view of the IDE

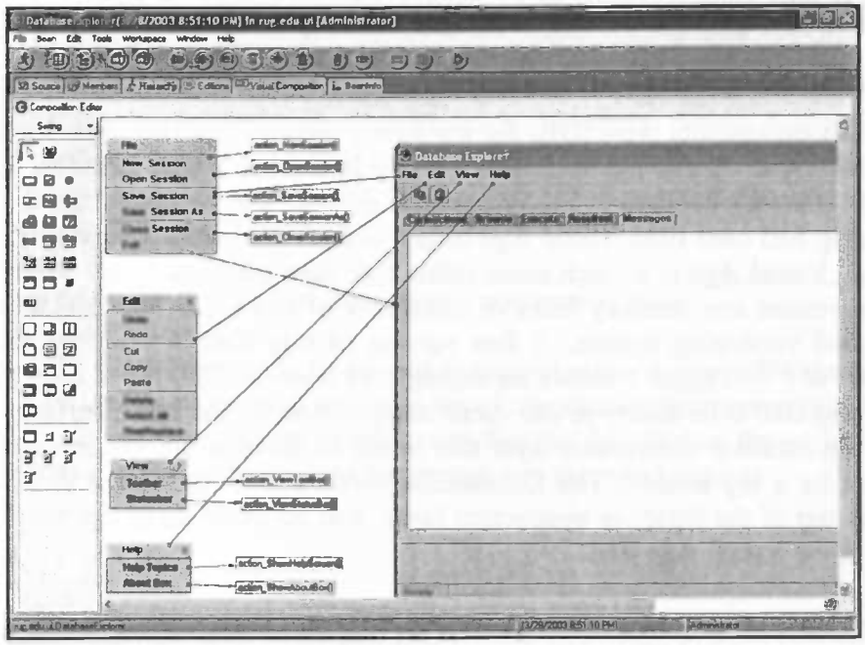


Figure 2: example view of the visual composition editor

Implementation

The Abstract Database

The first and probably most important step in our implementation is the creation of an abstract class 'AbstractDatabase'. In theory this abstract class serves just as a guideline for the implementation of the actual database drivers, but because a lot of database primitives are already available in the Java programming language it implements most methods by default. This does not mean database drivers have to make use of these 'default' implementations. The Java programming language permits the use of overloading methods. When default implementations are overloaded they will not be visible any more by default, however they scan still be called by calling the drivers super() (which is of course an instance of an AbstractDatabase). See the accompanying developers guide for an example implementation of a driver that uses such methods.

```
public abstract class AbstractDatabase {
  public abstract boolean open(String connection_string);
  public abstract void close();
  public abstract boolean open(String server,  String database,
                               String user, String password);
  Finalize()
  public int getAccessMethod()
  public int getColumnCount()
  public Connection getConnection()
  public String getDatabase() // returns database name
  public int getFetchSize()
  public String[] getHeaders()
  public DatabaseMetaData getMetaData()
  public int getNumber()
  public String getPassword()
  public String getPort()
  public int getRetrieveMethod()
  public String getServer()
  public TableModel getTableModel()
  public String getType()
  public String getUser()
  public String[] nextRow()
  public void printMessage(String tp, String msg)
  public void setFetchSize(int rows)
  public void setProperties(String s, String o, String t, String d,
                            String u, String p, boolean r, int n)
  public boolean setResultSet(ResultSet r)
  public String toString()
  public boolean succesfull()
  public boolean query(String queryStr)
  public boolean update(String updateStr)
  public boolean execute(String statement)
  public int check(String statement)
}
```

Listing 8: the structure of the abstract database

Listing 1 shows the structure of the abstract database class. I decided to make this an abstract class, because it makes the design a lot cleaner. By using an abstract class, users use the same interface independent of the actual database driver used. The abstract methods open(), close() and finalize() ensures that the database drivers will implement these methods.

The source of this class can be found at the project page [3-2]. A *JavaDoc* description of this class can be found there too [3-3].

The Abstract Database Layer

The abstract database layer class is the class that manages all abstract databases and thus all database connections. It keeps track of all connections being made and also has the ability to store connections for later usage. Normally users do not use an abstract database directly, but access them through the abstraction layer.
The listing below shows the structure of the class:

```
public DatabaseAbstractionLayer() {
   public void close() // closes all connections
   protected void finalize()  // makes sure exit cleanly
   public DefaultListModel getCurrentConnections() // returns open connections
   public DefaultListModel getStoredConnections() // returns stored connections
   public AbstractDatabase getDatabase()  // returns current selected database
   public String getDefaultServerType() // returns driver to use by default
   public DefaultListModel getDrivers() // returns list of registered drivers
   public boolean open(connection parameters) // open connection [1]
   public boolean registerDriver(String driver) // registers a database driver
   public boolean selectDatabase(int index) // selects default database to use
   public boolean store(String server, String port, String type, String database,
                        String user, String password, boolean remember_password, int mode)
}
```

Listing 9: the structure of the database abstraction layer

1) Connections can be opened in several ways. A user can either specify a connection string or pass a list of connection parameters including the hostname, username and password to use. See the *JavaDoc* on the project page [3-3] for an exact specification of the parameters that can be used.

Before the abstraction layer can use a database driver, the driver has to be registered first. The RegisterDriver() method (see listing 3) tries to register the specified driver.

```
public boolean registerDriver(String driver) {
 try {
  String package_name =
            this.getClass().getName().substring(0,
                this.getClass().getName().lastIndexOf(".")+1);
  ClassLoader loader = this.getClass().getClassLoader();
  if (loader.loadClass(package_name + driver)!=null) {
    /* NOTE: method findClass not visible, using loadClass instead          */
    /* alternative method: Class.forName(package + driver).newInstance();  */
    registeredDrivers.addElement(new String(package_name + driver));
    System.err.println("[DatabaseAbstractionLayer] Registering driver: " + driver);
  }
  return true;
 } catch (Exception e) {
  System.err.println("[DatabaseAbstractionLayer] Unable to register driver: " +
                     driver + "  ("+e+")");
  return false;
 }
}
```

Listing 10: the registerDriver method

The method first retrieves the same ClassLoader used for loading the database abstraction layer. Next it uses the ClassLoader 'loader' to load the specified driver. If the specified driver can be loaded, it will be added to the list of registered drivers and the driver is ready to be used.

The Database Drivers

Database drivers handle the DBMS specific part of an abstract database. Because the abstract database has already implemented most functions for you it is generally not hard to implement a driver for a specific DBMS, especially if a newer JDBC (version 2 or higher) driver exists. The only methods we must implement are the methods that are defined ***abstract*** in the AbstractDatabase and a finalize() method. The finalize() method is also obligatory because we want to be sure that the database connection is closed properly after usage. To create a framework for the database drivers we first implement a dummy driver. This framework makes it easier to implement real drivers. There is also a second reason: the database abstraction layer and user applications might want to have a database driver object without having an actual connection with a real database.

```
public class DRV_Dummy extends AbstractDatabase {

  /**
   * DRV_Dummy constructor
   */
  public DRV_Dummy() {
    super();
  }

  /**
   * Closes the connection in a proper manner
   * Catches exception if this connection is already closed
   */
  public void close() {
    try {
      con.close();
    } catch (Exception e) {}
  }

  /**
   * Makes sure to close the connection.
   * if this object is going to be removed.
   */
  public void finalize() {
    close();
  }

  /**
   * Tries to create a connection with the database.
   */
  public boolean open(String dbURL) {
    return false;
  }

  /**
   * Tries to open the connection with the given parameters.
   */
  public boolean open(String server, String database, String user, String password) {
    return false;
  }

}
```

Listing 11: the dummy driver

The listing above shows the full source code of the Dummy driver. As you can see, the structure of this driver (actually every database driver) is very simple and straightforward. The sole purpose of the database drivers is to ***glue*** the abstract database to a real database driver. Because of its simple structure it is easy for developers to implement their own drivers making it possible to access almost every database as an *abstract* database. Currently database drivers for various DBMSs are already available including support for SQL Server, MySQL and PostgresSQL. The sources for these drivers are available at the WWW [3-4].

The Database Explorer

The Database Explorer is a graphical user application demonstrating the capabilities of the abstract database layer. Originally this application started as a small application to test the abstraction layer. This application steadily evolved in a user-friendly application embodying the powerful capabilities of the underlying software layer. One can say that it now serves both as a test case for the Database Abstraction Layer as well as an easy-to-use user application. You can take a look at the user guide for more information on how you can use this application. The interface of the Database Explorer has been created using the composition editor of Visual Age for Java.
I have tried to match the user interface with the basic functions of the database abstraction layer. The main window of the user interface has 5 tabs. The first tab allows you to connect to a DBMS, the second one lets you browse your databases, the third one lets you execute SQL statements and the fourth one shows you the results of the performed SQL queries. See the figure below

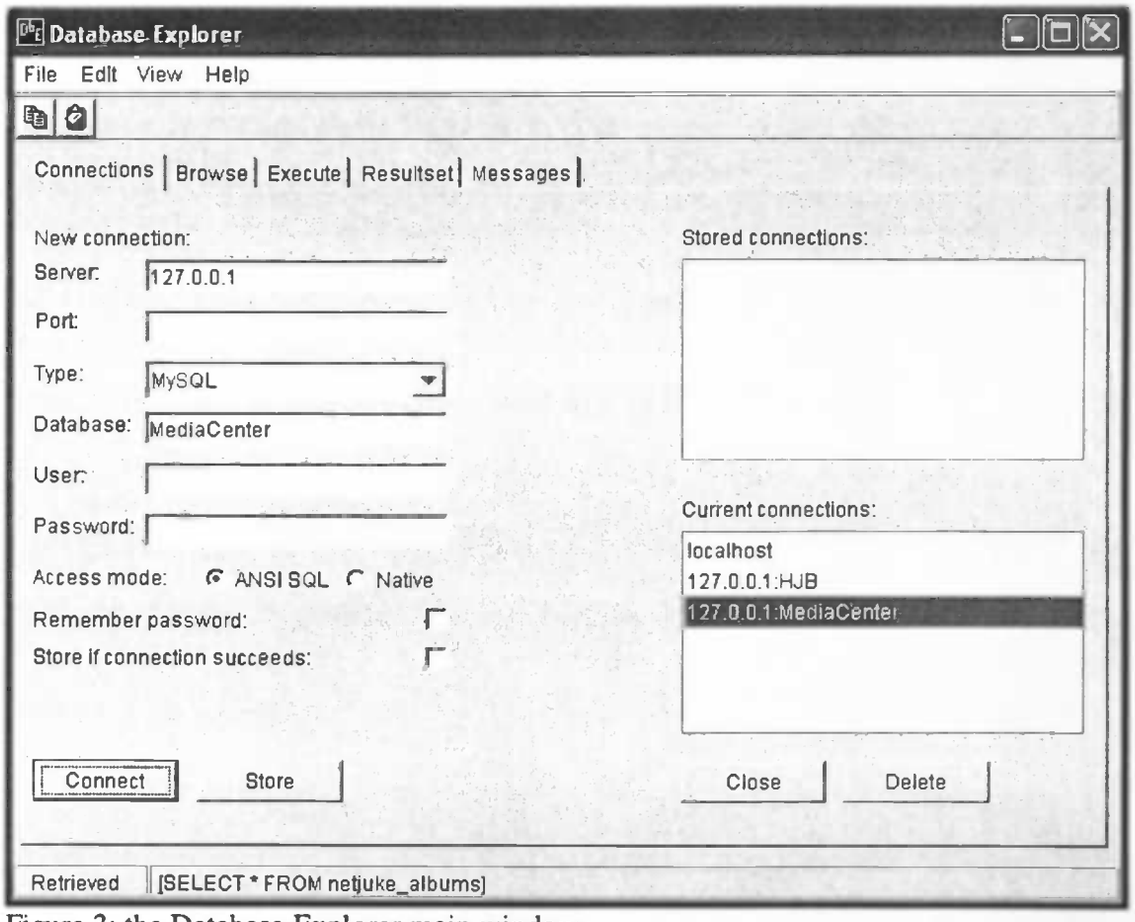


Figure 3: the Database Explorer main window

The fifth tab shows all error, warning and information messages generated by the abstraction layer or the Java subsystem. This is implemented by redirecting the *standard error* and *standard output* to the message windows. Technically this works by creating a stream to the text area of our message window (See listing 5).

```
//redirect standard error/out output to the message tab
GuiPrintStream gos = new GuiPrintStream(new
    GuiOutputStream(getTextArea_Messages()));
System.setOut((PrintStream) gos);
System.setErr((PrintStream) gos);
```

Listing 12: redirecting the standard error/output

Generated user interfaces have the tendency to favor the ease of use at the cost of readability of the source code. I tried to keep the source as clear as possible. All action on components in the user interface are linked to methods starting with '*action*' in the source code. All components start with '*get*' followed by its name. All connections between the components and actions start with 'conn'. At the startup of this application the method 'initialize' is being executed. This method initializes all components, redirects the standard error and output as described above, creates a database abstraction layer, registers all available database drivers and makes a list of registered database drivers available to the user. Information on how to use this program can be found in the *users guide* [3-5].

Conclusions

The abstraction layer is an example of software that emerged from real life experiences. From previous projects I experienced that there was a need for a mechanism to access databases in a simple and transparent manner. By looking closely to the various database connection technologies and to object-oriented programming in general, it was possible to implement a functional abstraction layer that suits these needs.
One might ask why such an implementation is useful after all there are already lots of database connection technologies. All though that is certainly true you do not want to rely on one single connection technology, because a single connection technology is no guarantee that all DBMSs that you want to use are supported. Your DBMS might not be supported that well or even at all!
Although the current implementation is fully functional there is room still for improvement. Possible improvements are for example the implementation of more drivers (especially a CORBA client and a CORBA ORB driver, that would make the abstraction layer CORBA enabled), better SQL syntax checking and the addition of JDO support (making it possible to map tables from the database to objects and back again).

History

My first real encounter with applications that make extensible use of databases was for the course Software Engineering. During this course we had to design and implement an online reservation system. Because the system had to make database connections on various places in the implementation, we decided to make a separate class handling the database connection us. When I later encountered the same issues during my internship for Aegon Inc in Taiwan, I decided to create a more generic set of classes to handle database connections, which resulted in the first steps of the DatabaseAbstractionLayer. After my internship the abstraction layer evolved gradually until it became what it is today. More features where added and a test application was made.

Glossary

Abbreviations used in this document

ADO	-	ActiveX Data Objects, programming interface from Microsoft to access data in a database
ANSI	-	American National Standards Institute
API	-	Application Programming Interface, enables programmers to use (external) functionality in a structured manner
ASP	-	Active Server Pages, Microsoft scripting language for the creation of dynamic web pages
ATM	-	Asynchronous Transfer Mode, a network technology based on transferring data in packets. ATM differs from TCP/IP, because ATM creates a fixed channel or route whenever data transfer begins.
COM	-	Component Object Model, a software architecture developed by Microsoft to build component-based applications. COM was first released in 1993. Microsoft's infamous ActiveX is based on COM.
CORBA	-	Common Object Request Broker Architecture, OMG's open, vendor-independent specification for an architecture and infrastructure that computer applications use to work together over networks.
DB	-	Database, a collection of related data
DBMS	-	Database Management System, a collection of programs that enable users to create and maintain a database.
DCOM	-	Distributed Component Object Model, an extension of the Component Object Model (COM) that allows COM components to communicate across networks. DCOM uses the RPC mechanism to transparently send and receive information between COM components. DCOM was introduced in 1995 with the introduction of Windows NT 4
DLL	-	Dynamic Link Library, a library of functions and other information that can be accessed by a program.
IDL	-	Interface Definition Language, used to define the interfaces to CORBA objects.
IIOP	-	Internet Inter Orb Protocol, a protocol developed by the OMG to implement CORBA solutions over networks.
IIS	-	Internet Information Services, a web server from Microsoft
IP	-	Internet Protocol, specifies the format of network data packets and the addressing scheme.
ISO	-	International Standards Organization
JSP	-	Java Server Pages, server side scripting in Java used to create dynamic Web pages

JDBC	-	Java Database Connectivity, set of Java classes developed by Sun Microsystems to allow access to relational database through the execution of SQL-statements.
MDAC	-	Microsoft Data Access Components, package containing several components that provide various database technologies including JDBC ADO/OLE DB
MTS	-	Microsoft Transaction Server, a server program that manages application and database transaction requests on behalf of a client.
.NET	-	A Microsoft platform that incorporates applications, tools and services in order to erase the boundaries between applications and the Internet.
ODBC	-	Object Database Connectivity, a standard database access method. The goal of ODBC is to make it possible to access databases from applications. This goal is achieved by inserting a middleware layer, the ODBC manager, between an application and the DBMS.
ODMG	-	Object Database Management Group, a consortium of object-oriented DBMS vendors and users. Not to be confused with OMG. The ODMG stopped after the completion of its work on object data management standards in 2001.
OLE	-	Abbreviation of Object Linking and Embedding, a compound document standard developed by Microsoft Corporation. It enables you to create objects with one application and then link or embed them in a second application.
OMG	-	Object Management Group, a consortium with a membership of more than 700 companies. The organization's goal is to provide a common framework for developing applications using object-oriented programming techniques.
PHP	-	Self-referentially short for PHP: Hypertext Preprocessor, an open source, server-side, HTML embedded scripting language used to create dynamic Web pages.
ROI	-	Return on Investment, analyzes the derived benefits of an investment relative to its costs.
RPC	-	Remote Procedure Call, a type of protocol that allows a program on one computer to execute a program on a server. The client program sends a message to the server with the appropriate arguments and the server returns a message containing the result of the program executed.
SAG	-	SQL Access Group, an organization of leading hardware and software developers committed to universal database access.

SQL	-	Structured Query Language, the standard language for commercial relational DBMSs. Originally called SEQUEL and designed and implemented at IBM Research. Standardized by the joint effort of ISO and ANSI. In 1986, ANSI approved a rudimentary version of SQL as the official standard, but most versions of SQL since then have included many extensions to the ANSI standard. In 1991, ANSI updated the standard. Because the SQL Access Group developed the draft for this new standard, it is known as SAG SQL.
TCO	-	Total Cost of Ownership, reveals the total cost of a solution over time, including ongoing maintenance, integration and support costs.
TCP	-	Transmission Control Protocol, enables two hosts to establish a connection and exchange streams of data.

References

General:

[g1] Elmasri & Navate [2000],
Fundamentals of Database Systems 3rd edition,
Addison-Wesley, 2000

[g2] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin,
Helena Gilchrist, Fiona Hayes, Paul Jeremaes [1994],
Object-Oriented Development, The Fusion Method,
Prentice-Hall, 1994

[g3] Cay Horstmann, Gary Cornell,
Core Java, Volume 1 – Fundamentals,
Prentice-Hall, 1998

[g4] H.M. Deitel, P.J. Deitel ,
Java, How to program 2nd edition,
Prentice-Hall, 1998

Part I:

[1-1] ODBC information
http://www.iodbc.org

[1-2] ODBC Visual Basic example
Bretschneider, Udo translation by Wessels, Paul
Visual Basic 4.0
Databecker Gmbh & Co. KG, 1996

[1-3] ADO DB example
http://www.speech.cs.cmu.edu/~sburke/pub/tpj17.html

[1-4] Types of JDBC technology drivers
http://java.sun.com/products/jdbc/driverdesc.html

[1-5] JDBC example;
http://www.wdvl.com/Authoring/Java/Servlets/JDBC_example.html

[1-6] CORBA information
http://www.corba.org

[1-7] OMG: The Object Management Group
http://www.omg.org

[1-8] OMG IDL Specifications
http://www.omg.org/technology/documents/idl2x_spec_catalog.htm

[1-9] CORBA IDL example
http://java.sun.com/products/jdk/1.2/docs/guide/idl/tutorial/GSIDL.html

[1-10] CORBA Server example
http://java.sun.com/products/jdk/1.2/docs/guide/idl/tutorial/app/HelloClient.java

[1-11] CORBA Client example
http://java.sun.com/products/jdk/1.2/docs/guide/idl/tutorial/app/HelloClient.java

[1-12] Carnegie Mellon Software Engineering Institute
Component Object Model (COM), DCOM, and Related Capabilities
http://www.sei.cmu.edu/str/descriptions/com.html

[1-13] Tallman, Owen and Kain, J. Bradford
COM versus CORBA: A Decision Framework
Printed in Distributed Computing, September–December 1998
http://www.scit.wlv.ac.uk/~cm1924/cp3025/distrib/reading/corba/corba3/corba6.html

Part II:

[2-1]: Object oriented programming explained
http://www.webopedia.com/TERM/o/object_oriented_programming_OOP.html

[2-2]: Stroustrup, Bjarne,
What is "Object-Oriented Programming"?, 1991
http://www.research.att.com/~bs/whatis.ps

[2-3]: Ole Lehrmann Madsen,
"What object-oriented programming may be – and what it does not have to be"
Lecture Notes in Computer Science Ed: G. Goos and J. Hartmanis, 1988

[2-4]: Campione, Mary and Walrath, Kathy
Object-Oriented Programming Concepts: A Primer, 1996
http://www.ecs.umass.edu/ece/wireless/people/emmanuel/java/java/objects/
http://www.ecs.umass.edu/ece/wireless/people/emmanuel/java/java/javaOO/

[2-5]: Hostetter, Chris,
Survey of Object Oriented Programming Languages, 1998
http://www.rescomp.berkely.edu/~hossman/cs263/paper.html

[2-6]: Nguyen, Van and Hailpern, Brent
A Generalized Object Model
ACM SIGPLAN NOTICES – v21, n10, Ed: Richard Wexelblat, 1986

[2-7]: What Is an Object?
Learning the Java Language, Sun Microsystems, Inc., 2003
http://java.sun.com/docs/books/tutorial/java/concepts/object.html

[2-8]: What Is a Message?
Learning the Java Language, Sun Microsystems, Inc., 2003
http://java.sun.com/docs/books/tutorial/java/concepts/message.html

[2-9]: What Is Inheritance?
Learning the Java Language, Sun Microsystems, Inc., 2003
http://java.sun.com/docs/books/tutorial/java/concepts/inheritance.html

[2-10]: Cattell, R. G. G.
The Object Database Standard: ODMG-93
Morgan Kaufmann 1993

[2-11]: Cattell, R. G. G., Barry, Douglas K.
The Object Data Standard: ODMG 3.0
Morgan Kaufmann 2000, ISBN: 1558606475

[2-12]: Schmitt, Ingo
Der ODMG 3.0-Standard
http://www.mycosima.com/dbp2/vorlesung/odmg.pdf

[2-13]: Transparent Persistence with Java Data Objects
http://developers.sun.com/tools/javatools/articles/transparent.html

[2-14]: Transparent Persistence
http://www.service-architecture.com/object-oriented-databases/
articles/transparent_persistence.html

[2-15]: Cayenne
http://www.objectstyle.org/cayenne/

[2-16]: SolarMetric KODO JDO
http://www.solarmetric.com/Software/Documentation/latest/docs/index.html

[2-17]: Prevayler, a software prevalence layer for Java
http://www.prevayler.org

Part III:

[3-1]: IBM Visual Age for Java
http://software.ibm.com/ad/vajava

[3-2]: Project Page, Source: AbstractDatabase.java
http://www.fmf.nl/~eelco/afstuderen/
files/source/rug/edu/database/AbstractDatabase.java

[3-3]: Project Page, Javadoc
http://www.fmf.nl/~eelco/afstuderen/files/javadoc/

[3-4]: Project Page, Source: abstract database and drivers
http://www.fmf.nl/~eelco/afstuderen/files/javadoc/rug/edu/database/

[3-5]: Project Page, Documents: users guide
http://www.fmf.nl/~eelco/afstuderen/files/documents/users_guide.ps

Appendix A: Developers Guide

Overview

- Getting started
- Registering database drivers
- Creating connections
- Gathering information
- Executing statements
- Viewing results
- A programming example
- Developing database drivers
- More information

Getting started

The Database Abstraction Layer is a software layer positioned between the user application and the database making it easier to access databases from different DBMS vendors.

This guide is intended for developers who want more information on how they can make their applications 'Abstract Database'-enabled.

There are two sample applications bundled with the sources of this package. The first application is a full-blown program called 'Database Explorer' which makes extensive use of the capabilities of this software layer. The second application is a small sample application simply called: 'Example'. This example is a good starting point for integrating the Database Abstraction Layer in your application.

Before you can use this package the first time it is important to verify if you have the Java runtime environment (version 1.2.2 or higher) installed. Versions before 1.4.1 might require the setting of the correct $CLASSPATH variable. From version 1.4.1 onwards this should not be necessary.

Registering database drivers

Before we can open and use database connections we have to create an abstraction layer and register the database drivers we want to use. (Listing 1)

```
// STEP 1: create the database abstraction layer
dbl = new DatabaseAbstractionLayer();

// STEP 2: register the driver(s) you want to use
dbl.registerDriver("JDBC_MySQL");
dbl.registerDriver("More Drivers");
```

Listing 1: create an abstraction layer and register drivers

A database driver is the DBMS specific part of an 'Abstract Database'. There is no limitation in the amount of drivers that can be registered. Multiple drivers may even use the same DBMS. For example: you can create a driver accessing a SQL Server using an ODBC connection, but you can also implement a driver accessing the same SQL Server with a JDBC connection. Internally, the registerDriver() method tries to find the ***class*** specified by the given parameter in the available packages or on the local file system and loads it into memory. This enables you to add and use drivers on the fly without recompilation of all your classes.

Note: the registerDriver() method returns a Boolean which enables you to check if the specified driver was registered successfully.

Creating connections

After you have registered one or more drivers, you can try to make a connection with a database. There are three approaches that can be used to open a connection. The first approach is to open a connection without specifying the driver to use. The abstraction layer will try to figure out what driver to use by creating connections and looking for the specified database. The second method tells the abstraction layer that is **has** to use the given driver. Note: all parameters are Strings and are not required to have a value. When a parameter is empty the system will use its default value. See the user guide for a table containing all default values. The third method is driver dependent and intended for testing purposes only. It is **not** recommended to use this approach for normal use. (See listing 2)

```
// Approach A: open a connection with auto detection of the driver to use
dbl.open(server, database, user, password);

// Approach B: open a connection through the specified driver
dbl.open(server, database, user, password, driver);

// Approach C: open a connection using the supplied connection string
dbl.open("jdbc:mysql://server/database?user=XXX&password=XXX");
```

Listing 2: connecting to a database

Gathering information

You are probably curious what an established connection has to offer you. You can find out everything you want to know about your DBMS and databases by requesting a 'DatabaseTreeModel' or a 'DatabaseMetaData' object. The DatabaseTreeModel is a tree structure containing all accessible databases, tables and columns. Furthermore this structure contains a list of all metadata properties describing the capabilities of the DBMS. The DatabaseMetaData object contains all database metadata properties. It tells you, for example, whether you can or cannot write to the database, use subselect statements or use batch updates. The DatabaseMetaData object is in fact a *java.sql.DatabaseMetaData* object, already available in every version of Java. See [url1] for a comprehensive description of this class. The following example shows you how you can create and use a tree and a metadata object:

```
// Create a tree model
DatabaseTreeModel dtm = new DatabaseTreeModel(dbl);

// Create a graphical representation of the tree using a java.swing.JTree
// See [url2] for a description of the usage of Jtrees
// Note: the DatabaseTreeModel extends the DefaultTreeModel and therefore
// it can be used directly by Java Swing components
JTree aTree = new JTree();
aTree.setName("Graphic representation of the DatabaseTreeModel ");
aTree.setModel(dtm);


// Get the DatabaseMetaData object belonging to the current selected database
DatabaseMetaData dmd = dbl.getDatabase().getMetaData();

// Simple meta data test: check if DBMS supports the ODBC Minimum
// SQL grammar. See [url1] for all available methods
Boolean supportMinSQL = dmd.supportsMinimumSQLGrammar()
```

Listing 3: gathering information

NOTE: a summary of all methods and fields can be found in the generated JavaDoc files. These documents are available at the project page [url3].

Executing statements

Connecting and retrieving information is one thing, but it becomes really interesting when you can actually communicate with your databases. The Database Abstraction Layer distinguishes between two types of database access. The first type, ***read-only*** access, assures that data can only be retrieved from the database. Insertions and deletions are not possible. The corresponding method is 'query()'. The second type supports ***write*** access and can be used for deletions and insertions. The corresponding method is called 'update()'. Normally, you do not have to distinguish between these two types of access and you can just use the 'execute()' method. The 'check()' method enables you to check what kind of access you should use with a given SQL statement. The current implementation of this method performs only basic checks like searching for keywords that need write access. You are free to overload this method with a better implementation. Listing 4 gives you examples how to check and execute SQL statements.

```
// Get the selected database from the database abstraction layer
AbstractDatabase adb = dbl.getDatabase();

// Create a sample SQL query
String statement = "SELECT * FROM CustomerData";

// Check what kind of query we have
// After execution 'statement' can contain one of the following values:
// AbstractDatabase.isQuery: a valid query (read-only) statement detected
// AbstractDatabase.isUpdate: a valid update (write) statement detected
// AbstractDatabase.isError: not a valid SQL statement detected
// in this particular case 'statement' equals AbstractDatabase.isQuery
int check = adb.check(statement);

// perform query or update, based on our check
if (check==AbstractDatabase.isError) {
  System.err.println("Error in SQL Statement");
} else{
  if (check==AbstractDatabase.isQuery) {
    adb.query(statement);
  }
  if (check==AbstractDatabase.isUpdate) {
    adb.update(statement);
  }
}
```

```
// Alternative execution of our SQL statement
// Note: this is the simplest and preferred way to perform an SQL query
// The execute method decides automatically which methods to call
// Note: Returns true if execution was successful
Boolean ok = adb.execute(statement);
```

Listing 4: examples of checking and performing queries

Warning: all update statements will be executed on your database.
When used without caution, you can seriously harm the contents of your databases.

Viewing results

The results of your queries are stored in a ResultSet object. The simplest way to visualize your results is by asking the database abstraction layer to create a model of your data. The returned model can be directly inserted in a javax.swing.JTable (See listing 5) because it is compatible with a javax.swing.table.TableModel [url4]. The database abstraction layer tries to minimize the amount of network traffic between the DBMS and your application by only fetching the data rows that are actually displayed. Unfortunately, this is not possible when you are using a JDBC version 1.0 driver. If you use such a driver, the database abstraction layer will build a different, but still compatible table model, which *does* contain all your returned data. The amount of data that should be prefetched can be manually tuned with the 'getFetchSize' and 'setFetchSize' methods (See Listing 5).

```
String query = "SELECT filename FROM test.files";

// STEP 4: perform the query
dbl.getDatabase().execute(query);

// Use specified fetch size of 10 rows at a time
dbl.getDatabase().setFetchSize(10);

// STEP 5: display the results
getTable().setModel(dbl.getDatabase().getTableModel());
```

Listing 5: visualizing query results

A programming example

We can summarize the code examples above to create a simple, but good example showing the capabilities of the Database Abstraction Layer. The following listing contains the full source code of an example program that is able to connect to a database, retrieve an entire table and view the contents of the retrieved table.

```
import java.awt.*;
import javax.swing.*;
import rug.edu.database.*;

/**
 * Example application for
 * the Database Abstraction Layer
 *
 */
public class Example extends JFrame {
        private JTable ivjTable = null;
        private JScrollPane ivjScrollPane = null;
        private rug.edu.database.DatabaseAbstractionLayer dbl = null;

  /**
   * Example constructor comment.
   */
  public Example() {
        super();
        initialize();
  }
```

```
/**
 * Tries to:
 * - creates a database connection
 * - perform a query
 * - display the results
 * <br>
 * Creation date: (2/2/2004 1:59:07 PM)
 */
public void doQuery() {
        // Configuration
        String driver = "JDBC_MySQL";
        String server = "";
        String user = "";
        String password = "";
        String database = "";
        String query = "SELECT filename FROM test.files";

        // STEP 1: create the database abstraction layer
        dbl = new DatabaseAbstractionLayer();

        // STEP 2: register the driver(s) you want to use
        dbl.registerDriver("JDBC_MySQL");

        // STEP 3: open the connection
        dbl.open(server, database, user, password);

        // STEP 4: perform the query
        dbl.getDatabase().execute(query);

        // STEP 5: display the results
        getTable().setModel(dbl.getDatabase().getTableModel());
}

/**
 * Creates the scrollable pane
 * Returns the ScrollPane property value.
 * @return javax.swing.JScrollPane
 */
private javax.swing.JScrollPane getScrollPane() {
        if (ivjScrollPane == null) {
                try {
                        ivjScrollPane = new javax.swing.JScrollPane();
                        ivjScrollPane.setName("ScrollPane");
                        ivjScrollPane.setVerticalScrollBarPolicy
                            (JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
                        ivjScrollPane.setHorizontalScrollBarPolicy
                            (JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
                        getScrollPane().setViewportView(getTable());
                } catch (java.lang.Throwable ivjExc) {
                        handleException(ivjExc);
                }
        }
        return ivjScrollPane;
}
```

```
/**
 * Creates the table in the scrollpane
 * Returns the Table property value.
 * @return javax.swing.JTable
 */
private javax.swing.JTable getTable() {
	if (ivjTable == null) {
		try {
			ivjTable = new javax.swing.JTable();
			ivjTable.setName("Table");
			getScrollPane().setColumnHeaderView(ivjTable.getTableHeader());
			getScrollPane().getViewport().setBackingStoreEnabled(true);
			ivjTable.setBounds(0, 0, 200, 200);
		} catch (java.lang.Throwable ivjExc) {
			handleException(ivjExc);
		}
	}
	return ivjTable;
}

/**
 * Called whenever the part throws an exception.
 * @param exception java.lang.Throwable
 */
private void handleException(java.lang.Throwable exception) {

	/* Uncomment the following lines to print uncaught exceptions to stdout */
	// System.out.println("--------- UNCAUGHT EXCEPTION ---------");
	// exception.printStackTrace(System.out);
}

/**
 * Initialize the class.
 */
private void initialize() {
	try {
	 setName("Example");
	 setDefaultCloseOperation(javax.swing.WindowConstants.DISPOSE_ON_CLOSE);
		setSize(600, 400);
		setTitle("Database Abstraction Layer - Example");
		setContentPane(getScrollPane());
	} catch (java.lang.Throwable ivjExc) {
		handleException(ivjExc);
	}
	// make the connection, perform the query and show the results
	doQuery();
}
```

```
/**
 * Starts the application.
 * @param args an array of command-line arguments
 */
public static void main(java.lang.String[] args) {
	try {
		/* Set native look and feel */
		UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
		/* Create the frame */
		Example aExample = new Example();

		/* Ask java.awt.Toolkit for the current screen size */
		Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();

		/* Pack frame on the screen */
		aExample.pack();

		/* Center frame on the screen */
		Dimension frameSize = aExample.getSize();
		if (frameSize.height > screenSize.height)
			frameSize.height = screenSize.height;
		if (frameSize.width > screenSize.width)
			frameSize.width = screenSize.width;
		aExample.setLocation((screenSize.width - frameSize.width) / 2,
				(screenSize.height - frameSize.height) / 2);

		/* Add a windowListener for the windowClosedEvent */
		aExample.addWindowListener(new java.awt.event.WindowAdapter() {
			public void windowClosed(java.awt.event.WindowEvent e) {
				System.exit(0);
			};
		});
		aExample.setVisible(true);
	} catch (Throwable exception) {
		System.err.println("Exception occurred in main() of Example");
		exception.printStackTrace(System.out);
	}
  }

}
```

Listing 6: a complete example.

If you take a closer look at this example, you will see that there is only one method performing all database related actions. This method, doQuery(), creates the database abstraction layer, registers the database driver you want to use, opens the connection with the database, performs the query and finally displays the results. The other methods in this example are just to initialize and glue the GUI-components together.

In order to run this example, make sure to edit the variables 'server', 'user', 'password' 'driver', 'database' and 'query' in the 'doQuery()'-method.
When we run this example, the resulting table looks like the figure below:

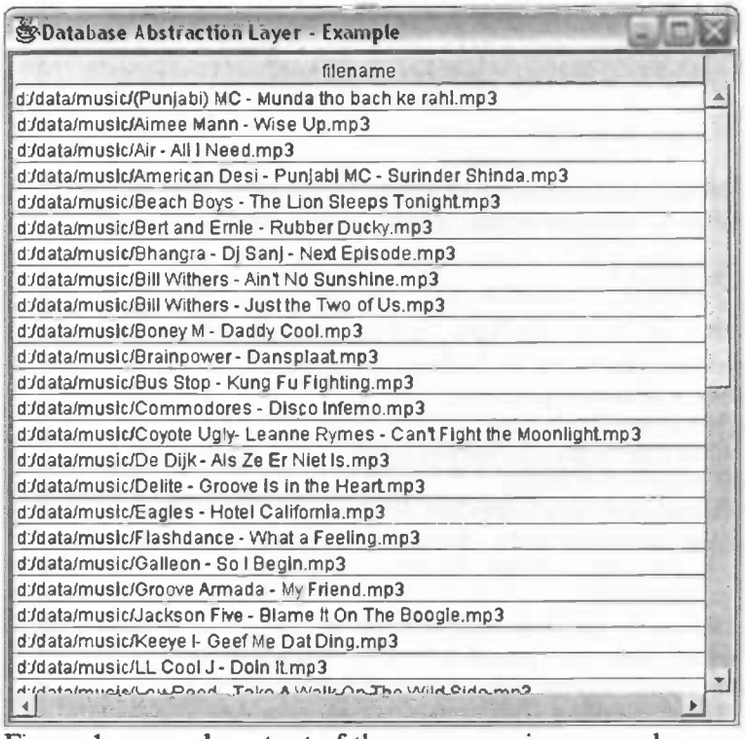


Figure 1: example output of the programming example.

Developing database drivers

Although the source package contains drivers for various DBMSs it is possible that your DBMS is not supported yet. If this is the case, you will have to add a database driver yourself. First you will have to find out what kind of connections your DBMS supports. Some DBMSs allow 'native' access directly from Java; others require the use of the JDBC or ODBC API. It is best to use the following guideline in case you have more than one option:

- First try to find a JDBC driver for your DBMS, preferably a version supporting JDBC 2.0 or higher. Implement the database driver using this JDBC driver. (See JDBC_Dummy for a generic implementation of such a driver.) Usually this results in a robust and fast combination.
- Use a 'native' Java implementation for accessing your DBMS. This can result in a very fast driver, but it might take some time to develop the driver.
- Use a 'native' ODBC driver. If you have an ODBC compliant DBMS and you can find a native ODBC driver for that particular DBMS, you can use that too.
  Note: in most cases these drivers are commercial products.
- Use the supplied data bridge. The supplied JDBC_ODBC driver, which implements an AbstractDatabase using Sun's ODBC bridge, enables you to use your ODBC compliant DBMS directly without the need for a new driver. Because of the many translation steps between the database and the application this is not the recommended thing to do for production systems.

In short, you can say JDBC is preferred over ODBC. The reason for this is twofold: we use Java for our programming language and JDBC has evolved to an integral part of that language and the second reason is that there are a lot of good and free JDBC drivers available. There is only one exception: Microsoft products generally work better with ODBC than with JDBC. There is a JDBC implementation for Microsoft's SQL-server, however you will not be able to use the JDBC 2.0 features.

Besides creating a totally new database driver, you can also implement an alternate database driver. For example: you already have a database driver, but you would like to be able to log and analyze all queries and the returned data. Now you have two options: create and insert a dummy 'pass through' driver or make a copy of an original driver and alter it.

During the following example we will create a generic 'pass through' driver logging all actions to a database.

We start with creating a copy of the Dummy driver and rename it to 'DRV_PassThrough'. Now, when we open the driver in an editor we see that we **must** implement the following methods:

- DRV_PassThrough(), the constructor
- open(), defines how to open the connection to the database[1]
- close(), defines how to close the connection to the database
- finalize(), makes sure to close the connection to the database when a instance of this driver is removed.

[1] There are two versions of the open() method to implement; the first version has only one parameter representing a 'connection string'. This string defines exactly how and where we should connect to. Unfortunately, this string is not exactly defined in the JDBC standard and different drivers use different schemes. The second version has four parameters representing the server, database, user and password to use. The primary goal of this version of the open() method is to level out the driver dependent differences mentioned above.

The primary task of the constructor is to initialize its superclass the 'AbstractDatabase'. Next, it initializes the driver used for logging and real database driver (see listing 7)

```
/**
 * The DRV_PassThrough constructor.
 * Initializes the superclass AbstractDatabase
 */
public DRV_PassThrough() {
  super();
  dal = new DatabaseAbstractionLayer(); // new hidden abstraction layer
  dal.registerDriver(logDriver);        // driver used for logging
  if (!adbDriver.equals(logDriver)) {
    dal.registerDriver(adbDriver);      // real driver to use
  }
  if (initializeLog()) {
    printMessage(DRV + NOTE, "Log initialized");
  } else {
    printMessage(DRV + NOTE, "Log is unavailable");
  }
}
```

Listing 7: the DRV_PassThrough constructor

The implementation of the open() method is somewhat special in this particular case. We do not open the database connection ourselves, because we do not exactly know which driver we should use. Instead we just ask the (hidden) database abstraction layer to do the job for us. If we have successfully opened the connection, we point the connection object and retrieve method of our database driver to the actual object. This ensures us that the user will see the real driver instead of our 'pass through' driver. (See listing 8)

```
/**
 * open the database connection
 *  ....but we are just a fake pass through driver ?!!!!
 *
 * Q: how do we know where we should connect to and
 *      what driver should be used ?
 *
 * A: just let the database abstraction layer do all the work!
 */
public boolean open(String server, String database, String user, String password) {

            boolean success = dal.open(server, database, user, password);

            if (success) {
                  con = dal.getDatabase().getConnection();
                  retrieve_method = dal.getDatabase().getRetrieveMethod();
            }

            printMessage(DRV + NOTE, "PassThrough driver initialized\n");
            return success;
 }
```

Listing 8: opening the database connection

The next method we must implement is the close() method. Normally we only have to close one connection, but because we log our actions to a database we have to close two connections. (See listing 9)

```
/**
 * Close the connection in a proper manner
 * Catch exception if this connection is already closed
 *
 * Closing the connection is important, because most
 * DBMS's can handle only a limited amount of connections.
 */
public void close() {
   // close connection to the database
   try {
      con.close();
   } catch (Exception d) { }
   // close connection to the database we are logging to
   try {
         log.close();
   } catch (Exception l) { }
}
```

Listing 9: the close() method

The last method that we are required to implement is the finalize() method. This method makes sure all connections are closed before this instance is removed. (See listing 10)

```
/**
 * Makes sure to close the connection
 * if this object is going to be removed.
 */
public void finalize() {
   this.close();
}
```

Listing 10: the finalize() method

Now that we have implemented all methods that are required for every driver, we can adapt our driver by adding extra methods and overloading some default methods of the AbstractDatabase class. An important fact in the Java programming is that overloaded methods still can be called by making its call to its super class. This saves us a lot of work and eases the implementation of the other methods specific for our example pass through driver (See listing 11)

```
/**
 * Execute the given statement.
 * Creation date: (3/2/2004 1:52:07 PM)
 * @return boolean
 * @param statement java.lang.String
 */
public boolean execute(String statement) {
	printMessage(DRV + NOTE, "Execution caught");
	boolean res = super.execute(statement);
	return res;
}

/**
 * Initialize our log.
 * Creation date: (4/2/2004 11:31:38 PM)
 * @return boolean
 */
public boolean initializeLog() {
	log = dal.open("","log","","");
	if (log) {
		Date dt = new Date();
		logID = dt.getTime();
		String dbCreate = "CREATE TABLE `log_" + logID +
		"` (`id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
		`action` VARCHAR(255) DEFAULT '' NOT NULL, `success`
		INT NOT NULL, `starttime` VARCHAR(255) NOT NULL,
		`duration` VARCHAR(255) NOT NULL);";
		log = dal.getDatabase().update(dbCreate);
	}
	return log;
}

/**
 * Perform query.
 * Creation date: (3/2/2004 2:06:25 PM)
 * @return boolean
 * @param queryStr java.lang.String
 */
public boolean query(String queryStr) {

	long sTime = new Date().getTime();
	boolean res = super.query(queryStr);
	long eTime = new Date().getTime();

	writeLog(queryStr, res, sTime, eTime-sTime);
```

```
        if (res) {
                printMessage(DRV + "Query caught:",
                "query was successful; (executed in " + (eTime-sTime) + " ms)");
        } else {
                printMessage(DRV + "Query caught:",
                "query was unsuccesful; (returned in " + (eTime-sTime) + " ms)");
        }
        return res;
}

/**
 * Return a description of this driver.
 * Creation date: (4/2/2004 10:37:41 PM)
 * @return java.lang.String
 */
public String toString() {
        return "pass through -> " + super.toString();
}

/**
 * Perform update
 * Creation date: (3/2/2004 2:06:03 PM)
 * @return boolean
 * @param updateStr java.lang.String
 */
public boolean update(String updateStr) {

        long sTime = new Date().getTime();
        boolean res = super.update(updateStr);
        long eTime = new Date().getTime();

        writeLog(updateStr, res, sTime, eTime-sTime);

        if (res) {
                printMessage(DRV + "Update caught:", "update was successful;
                (executed in " + (eTime-sTime) + " ms)");
        } else {
                printMessage(DRV + "Update caught:", "update was unsuccesful;
                (returned in " + (eTime-sTime) + " ms)");
        }
        return res;
}
```

```
/**
 * Write actions to our log.
 * Creation date: (5/2/2004 12:04:57 AM)
*/
public void writeLog(String action, boolean success, long starttime, long duration) {
 if (log) {
        String successStr = "0";
        if (success) {
                successStr = "1";
        }
        String actionStr = action; // NOTE: should be escaped??
        String logEntry = "INSERT INTO `log_" + logID +
        "` (`id`, `action`, `success`, `starttime`, `duration`)
        VALUES (", '" + actionStr + "', '" + successStr + "',
        '" + starttime + "', '" + duration + "');";
        dal.selectDatabase(0);        // make sure we select the log database
        boolean res = dal.getDatabase().update(logEntry);
        if (res) {
                printMessage(DRV + "Logged:",
                actionStr);
        } else {
                printMessage(DRV + "Failed to log the following statement:",
                actionStr);
        }
   }
 }
```

Listing 11: other implemented methods

The full source of this example can be found on the WWW [url5].
The example driver gives you an idea of a possible driver implementation.
Another possible driver implementation is for example an analyzer driver, analyzing errors by contacting a validation web service. [url6]

More information

More information can be found on the World Wide Web at:
http://www.fmf.nl/~eelco/afstuderen/

You can contact me by e-mail at:
eelco@fmf.nl

The source code for the Database Abstraction Layer:
http://www.fmf.nl/~eelco/afstuderen/files/sources.zip

Note:
The downloadable package includes programming examples and all the Java sources.

Links on the Word Wide Web:
[url1]: http://java.sun.com/j2se/1.4.1/docs/api/java/sql/DatabaseMetaData.html
[url2]: http://java.sun.com/j2se/1.4.1/docs/api/javax/swing/JTree.html
[url3]: http://www.fmf.nl/~eelco/afstuderen/files/javadoc/index.html
[url4]: http://java.sun.com/j2se/1.4.1/docs/api/javax/swing/table/TableModel.html
[url5]: http://www.fmf.nl/~eelco/afstuderen/files/source/rug/edu/database/
[url6]: http://developer.mimer.se/validator/

Appendix B: User Guide

Overview

- Getting started
- Managing database connections
- Browsing
- Executing commands
- Viewing results
- The message window
- More information

Getting started

The Database Explorer enables you to explore databases on various database management systems.

Originally this application started as a small application to test the Database Abstraction Layer. The Database Abstraction Layer is a software layer positioned between the user application and the database making it easier to access databases from different DBMS vendors. The application steadily evolved in a user-friendly application embodying the powerful capabilities of the underlying software layer. One can say that it now serves both as a test case for the Database Abstraction Layer as well as an easy-to-use user application.

Before you run this application the first time it is important to verify wheter you have the Java runtime environment (version 1.2.2 or higher) installed. Versions before 1.4.1 might require the setting of the correct $CLASS_PATH variable. From version 1.4.1 onwards this should not be necessary. You can start the application by running start.bat or start.sh depending on your platform. While the application initializes, a splash screen will be shown and finally the main window will come up.
The main window consists of various tabbed pages that can be brought up by clicking the corresponding tab. By default the application will start with the 'connection' page, ready to make a connection.

Managing database connections

To open a new connection, click on the 'connections'-tab to bring up the 'connection'-page. In the 'connection'-page you will find the various connection properties at the left side of the page. Here you can fill in the properties required for your connection. When a property is left blank, the application will substitute the blank property with the default value for that property (see table 1).

Property	Description	Default
Server	hostname or IP-address of the server	localhost or 127.0.0.1
Port	port number of the database server	default port number, specified by the driver
Type	the server/DBMS type to connect to specifies which driver should be used	AutoDetect, tries to auto detect the correct driver
Database	selects the initial database	none, connect to the DBMS, but no to a specific database
User	the username used for authentification	empty string [1]
Password	the password used for verification	empty string [1]
Access mode	access the database in compatibility mode or with full access control [2]	'Entry Level SQL-92'
Remember password	remember the password if you store the connection properties	do not remember password
Store if connection succeeds	stores connection for later usage	do not store connection

Table 1: overview of the connection properties.

[1] Some older versions of Java (1.2.2) and JDBC database drivers permit the use of empty usernames and passwords.

[2] You can select 'Entry Level SQL-92' to access the database in compatibility mode or 'Full Native Access' for full access control. Note: in compatibility mode, statements will be checked for keywords before execution.

If you choose to select the 'store if connection succeeds' property, the connection properties will be saved for later usage in this session and can be made available again by clicking on the corresponding entry in the 'stored connections'-list at the right side of the 'connection'-page. It is also possible to save your stored connections to a file, so you can use it again the next time you use the application. You can do so by saving your session with the 'save session' command in the file menu.

When have set all the necessary properties, you can now connect with your database by clicking the 'connect'-button. (Note: you can also choose to store your connection properties without making the connection by clicking the 'store'-button.)
If all goes well, the status message at the bottom of the screen will be updated to 'connected' and the new connection will be added to the 'current connections'-list.
If the status message says 'error', the application failed to connect to your database. You can find out what went wrong by clicking on the 'messages'-tab. The 'message'-page shows all error- and warning-messages generated by the underlying software layers.

If you are finished using a certain database you can close it by selecting the database you want to close in the 'current connections'-list, followed by clicking the 'close'-button on the bottom of the tabbed page.

It is also possible to remove stored connections. You can delete stored connections by selecting the connections you want to remove in the 'stored connections'-list followed by clicking the 'delete'-button on the bottom of the 'connection'-page.

Note: the easiest way to close all open connections and to remove all stored connections is by starting a new session. You can do this by issuing the 'new session' command, which can be found in the file menu. Of course it is also possible to reopen saved sessions. You can reopen saved sessions with the 'load session'-command, which too can be found in the file menu.

Browsing

You are probably curious what the current connections have to offer you. You can find out by selecting the 'browse'-page in the main window. This page shows a tree structure similar to the figure below:

Figure 1: browsing through your connections

In the root of this tree you will find all available connections. Each connection has two branches: one branch called 'properties' and another one called 'databases'. In the properties branch you will find all relevant properties for this specific connection.

Properties include basic settings like read only access and the used username, but also descriptions of the DBMS dependent features. Besides telling you what you can do, it also tells you what you cannot do! This is important because not all DBMSs support all standard SQL-92 functions. For example in the figure above, you can see that MySQL does not support UNIONS or sub queries. The databases branch shows all databases that can be accessed through the connection. When you click on a database, the next branch containing all the tables will be shown. Now, when you click on a table, the tree is expanded even further and the individual columns become visible. (See figure 1)

A pop-up menu is brought up if you double click a table or column. This pop-up menu allows you to perform the basic commands 'view' and 'drop' on the selected table or column. The 'view' command retrieves all records from the selected table or column and shows them in the 'result set'-page. The 'drop' command allows you to delete the selected column or table.

Note: some database drivers let you show and browse all the databases that are managed by the DBMS you are connected to, even if you made a connection with a specific database. Others show only the database where you made a connection to in the first place. A third type of database driver does show all available databases, but only permits you to browse the selected database. The exact behavior varies on the combination of drivers, DBMSs and configuration used. For example: a database administrator can choose to allow a specific user only read-only access to a certain database while hiding all other databases contained in the DBMS.

Executing commands

Besides browsing through your database, you can also build and execute SQL queries.
You can type your queries or updates in the text editor of the 'execute'-page.
The statements can be executed by clicking the 'execute'-button

Warning: all statements typed in the editor will be executed on your database. When used without caution, you can seriously harm the contents of your database.

Viewing results

The 'result set'-page shows the results of the queries you executed from the 'execute'- or 'browse'-page. After you perform a query, the application will try to retrieve the query results as efficiently as possible by first creating a model of the data to represent. When possible, records are only sent from database server to the application when they are actually needed, thus saving network bandwidth. The created model also determines how your results are formatted. Although the application tries to format the results in the best possible way, it is possible that the formatting does not suit your needs. You can change the formatting easily. For example: if you want to make a certain column wider, go to one of the side edges of the column header with your mouse, click on it and move your mouse towards the neighboring column. You can also easily exchange columns by clicking on the header of the column you want to move and dragging the column to the place you want to position it. (See figure 2)

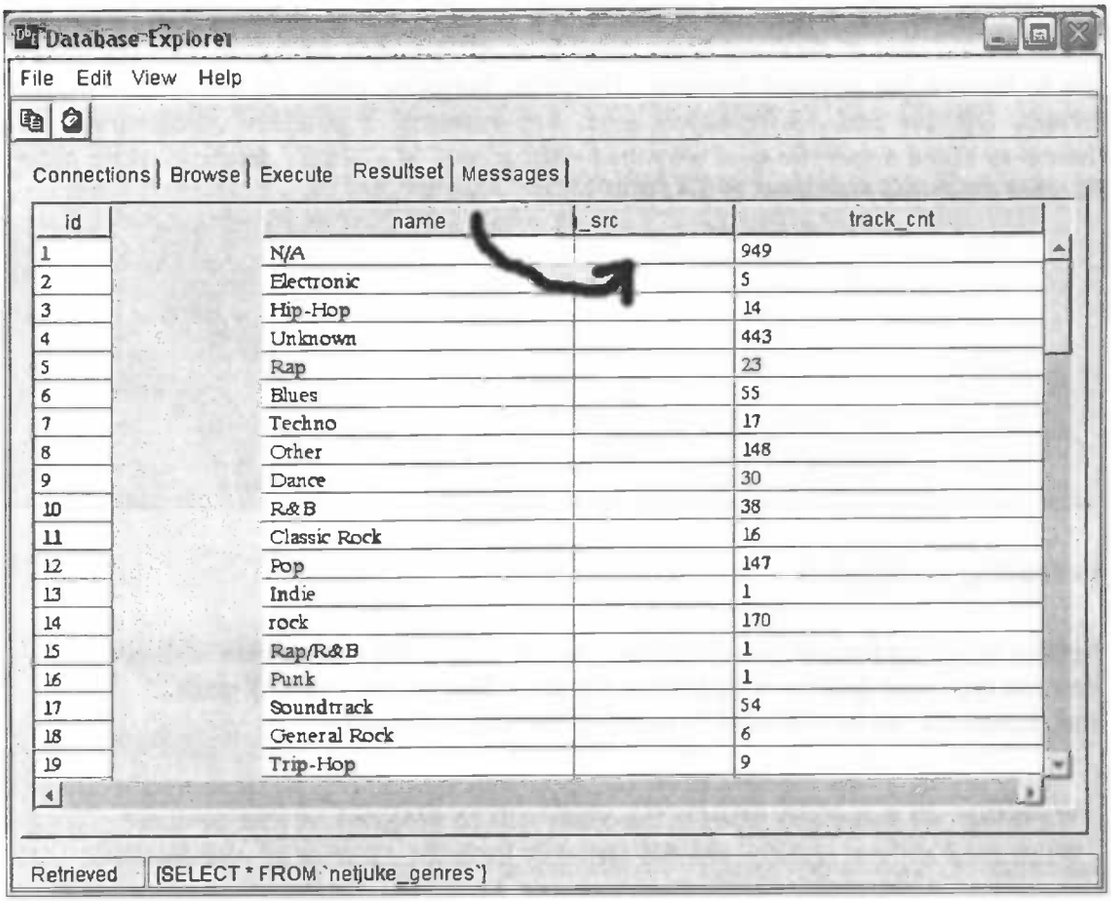


Figure 2: swapping columns

The message window

The message window shows all errors, warnings and other information generated by the underlying software components. This is important because it enables you to keep track of all the operations performed 'behind the scenes'. If anything goes wrong you can trace back the operations performed and where it went wrong. Each line in the message window represents exactly one message with the following syntax (See figure 3):
[software component] [action | type of message]: [message]

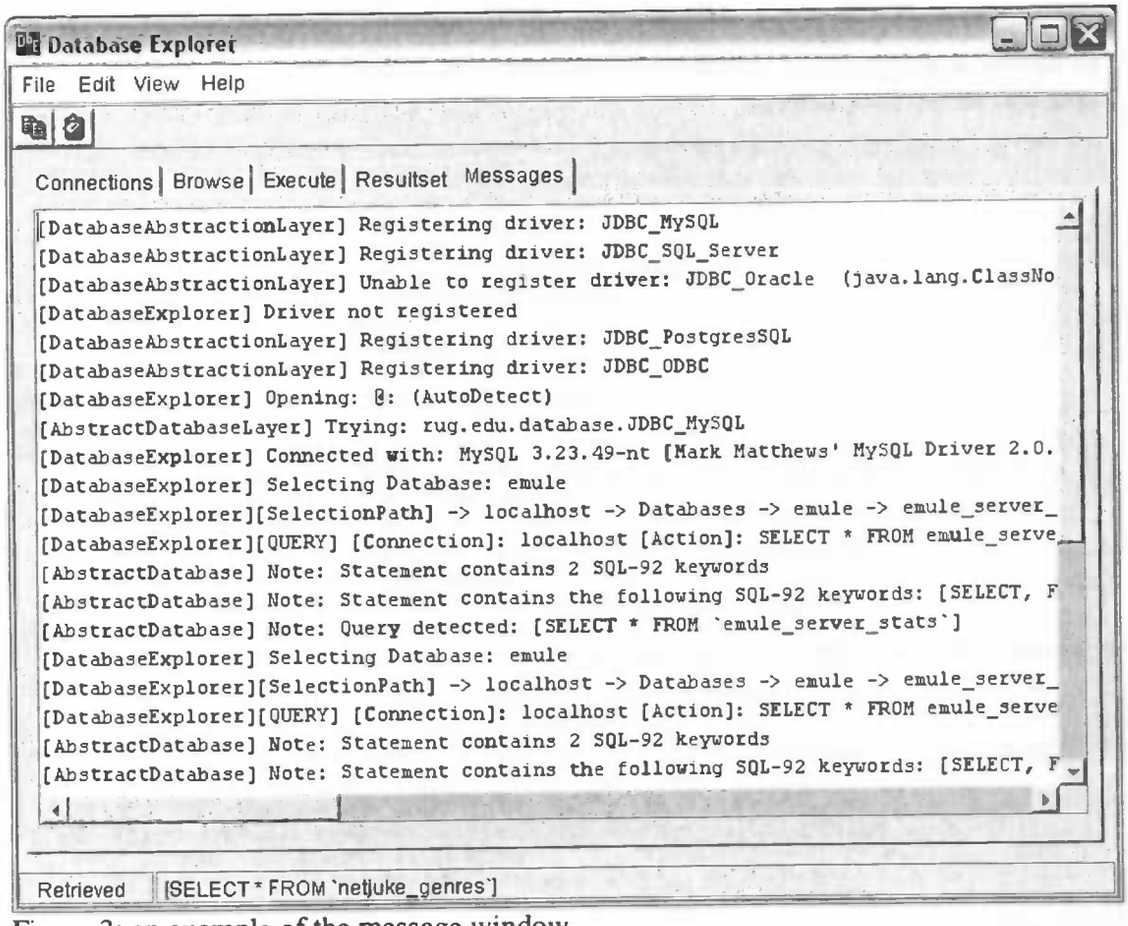


Figure 3: an example of the message window

More information

More information can be found on the World Wide Web at:
http://www.fmf.nl/~eelco/afstuderen/

You can contact me by e-mail at:
eelco@fmf.nl

You can download the Database Explorer application at:
http://www.fmf.nl/~eelco/afstuderen/files/sources.zip

Note:
The downloadable package includes the Database Abstraction Layer and all the Java sources.

Appendix C: Source Code

This appendix shows an overview of the most important Java classes that are implemented. A full description of these classes can be found in the generated Java documentation (JavaDoc) [1]. The full source code can be downloaded from the project page on the WWW [2,3].

AbstractDatabase.java

```
package rug.edu.database;

import java.util.*;
import javax.swing.table.*;
import java.sql.*;

/**
 * Implements the abstract database
 *
 *    - handles the connection with the database
 *    - handles queries and transactions
 *    - buffers resultsets
 *    - stores its properties
*/
public abstract class AbstractDatabase {

      /* final static variables                                          */
      /** Auto-detect mode, tries different retrieve methods until a
          correct one has been found                                      */
      final static int byTrial = 0;
      /** Retrieve table data using catalog information                   */
      final static int byCatalog = 1;
      /** Retrieve table data using schema information                    */
      final static int bySchema = 2;
      /** Retrieve table data by comparing catalog and database names     */
      final static int byComparison = 3;
      /** Compatability mode: only use entry level SQL-92 grammar         */
      final public static int useEntrySQL92 = 0;
      /** Full access mode: allow users full SQL access to the DBMS       */
      final public static int useFullAccess = 0;
      /** SQL statement is erroneous                                      */
      final public static int isError = -1;
      /** SQL statement is a query                                        */
      final public static int isQuery = 0;
      /** SQL statement is an update                                      */
      final public static int isUpdate = 1;
      /** How error messages should be announced                          */
      final static String ERROR = "Error:";
      /** How warnings should be announced                                */
      final static String WARNING = "Warning:";
      /** How notes should be announced                                   */
      final static String NOTE = "Note:";

      /* connection variables                                            */
      /** how to retrieve/search the databases; default: autodetect       */
      protected int retrieve_method = byTrial;
      /** how to access the DBMS; default: compatibility mode             */
      protected int access_method = useEntrySQL92;
```

```
/** The connection to database<br><br>@see java.sql.Connection  */
protected Connection con = null;
/** The resultset returned from the last query                  */
private ResultSet rset = null;
/** The number of columns in the current resultset              */
private int columns = 0;
/** Set to true when last query was successful                  */
private boolean succes = false;

/* configuration variables                                      */
/** The address of the database server. This can either be a
    hostname or an IP-address                                   */
private String server = "";
/** The port number of database server                          */
private String port = "";
/** The initial selected database                               */
 private String database = "";
/** The username used to identify to the DBMS                   */
private String user = "";
/** The password used for authorization                         */
private String password = "";
/** type of database server                                     */
private String type = "";
/** follow number, if more connections with the same properties
    coexist                                                     */
private int number = 0;
/** store password in configuration file ? default=false        */
private boolean remember_password = false;

/**
 * AbstractDatabase constructor.
 */
public AbstractDatabase() {
            super();
}

/**
 * This is the default implementation for SQL-statement checking
 * This method can be overridden by the actual driver because not
 * all DBMSs' provide the same level of functionality.
 * NOTE: very basic checking. Can be implemented much better!
 * NOTE: the most important task is to gather information for the
 *       user in case it goes wrong.
 * @return {isQuery,isUpdate,isError}
 * @param String (the SQL statement to be checked)
 */
public int check(String statement) {
      …
}

/**
 * Closes the connection with the database
 */
public abstract void close();

/**
 * Execute a SQL statement.
 * This is the default implementation for
 * executing a SQL statement.
 * The check() method checks if the given
 * string is a valid SQL statement compliant
 * with the used access method.
 * @return boolean
```

```
 * @param statement java.lang.String
 */
public boolean execute(String statement) {
      int statementType = this.check(statement);
      if (statementType == isQuery) {
            return query(statement);
      }
      if (statementType == isUpdate) {
            return update(statement);
      }
      // not an update or query statement
      return false;
}

/**
 * Returns the database access method.
 * Currently available access methods are:
 * {useEntrySQL92,useFullAccess}
 * @return int access_method
 */
public int getAccessMethod() {
      return access_method;
}

/**
 * Delivers the number of columns in the current table.
 * @return int columns
 */
public int getColumnCount() {
  return columns;
}

/**
 * Return the current connection object of this database
 * See: java.sql.Connection
 * @return Connection (The connection with the database)
 */

public Connection getConnection() {
  return con;
}

/**
 * Returns the database name.
 * @return String database
 */
public String getDatabase() {
  return database;
}

/**
 * Retrieves the fetch size in rows for the current ResultSet
 * object. Returns -1 if the fetch size cannot be retrieved.
 * @return int the current fetch size
 */
public int getFetchSize() {
      try {
            return rset.getFetchSize();
      } catch (Exception e) {
            return -1;
      }
}
```

```
/**
 * Delivers all the headers from the current table.
 * @return an array of strings with headers
 *
 */
public String[] getHeaders() {
  int i;
  if (columns == 0) return null;
  String[] result = new String[columns];
  try {
      for (i = 0; i < columns; i++) {
          result[i] = rset.getMetaData().getColumnLabel(i + 1);
      }
  } catch (SQLException e) { //do nothing }
  return result;
}

/**
 * Returns the DatabaseMetaData object belonging to the current
 * connection. The DatabaseMetaData object contains comprehensive
 * information about the database as a whole.
 * For more information see Interface java.sql.DatabaseMetaData
 * @return java.sql.DatabaseMetaData The MetaDataObject belonging
 * to the current connection
 */
public DatabaseMetaData getMetaData() {
  try {
      return this.getConnection().getMetaData();
  } catch (Exception e) {
      return null;
  }
}

/**
 * Returns the followup number
 * @return int_number
 */
public int getNumber() {
  return number;
}

/**
 * Returns the password used for connecting with this database
 * This method will return the password ONLY if
 * remember_password is set to TRUE (using setProperties())
 * @return String password
 */
public String getPassword() {
  if (remember_password==true) {
       return password;
  }
  return "";
}

/**
 * Returns the connection port number.
 * @return String port number.
 */
public String getPort() {
  return port;
}
```

```
/**
 * Returns the retrieval method used<br>
 * @return [byCatalog|byComparison|bySchema|byTrial]
 */
public int getRetrieveMethod() {
  return retrieve_method;
}

/**
 * Returns the server on which this database is located.
 * This can either be a hostname or an IP address.
 * @return server_string
 */
public String getServer() {
  return server;
}

/*
 * Returns a tablemodel of the currently available ResultSet.
 * @see rug.edu.database.DatabaseTableModel
 * @return rug.edu.database.DatabaseTableModel
 */
public TableModel getTableModel() {
  try {
      return new DatabaseTableModel(rset);
  } catch (Exception e) {
      // Creation of table model failed. Possible causes: old
      // JDBC 1.0 or buggy driver. Try creating a failsafe
      // table model based on the default table model
      ...
  }
  return null;
}

/**
 * Returns the type of DBMS used.
 * @return type_string
 */
public String getType() {
    return type;
}

/**
 * Returns the username used for the current connection.
 * @return String user
 */
public String getUser() {
    return user;
}

/**
 * Returns a new row from the ResultSet in an array of Strings.
 * If there are no more results available 'null' is returned.
 * @return An array with all the values from a single row or
 * null if empty.
 */
public String[] nextRow() {
  int i;
  String[] result = null;

  try {
      if (rset.next()) {
          result = new String[columns];
          for (i = 0; i < columns; i++) {
```

```
                result[i] = rset.getString(i + 1);
            }
        }
    } catch (Exception e) { return null; }
    return result;
}

/**
 * Open the connection with the database
 */
public abstract boolean open(String connection_string);
public abstract boolean open(String server, String database,
                             String user, String password);

/**
 * Prints a message to the standard output
 * @param String tp; type of message.
 * Predefined message types are: [ERROR|WARNING|NOTE]
 */
public void printMessage(String tp, String msg) {
  System.out.println("[AbstractDatabase] " + tp + " " + msg);
}

/**
 * Performs a query via the given connection and stores
 * the ResultSet and the number of columns.
 * This only works for SQL "select" statements.
 * Update, delete and insert statements have to
 * be made with update().
 * @param queryStr The query to be executed.<br>
 * @return True if query was executed successfully.
 */
public boolean query(String queryStr) {
      ...
}

/**
 * Tries to give the database driver a hint how many rows should
 * be fetched from the database when more rows are needed for this
 * ResultSet object. If the fetch size specified is zero, the
 * database driver ignores the value and is free to make its own
 * best guess as to what the fetch size should be.
 */
public void setFetchSize(int rows) {
      ...
}

/**
 * Sets the basic properties for this AbstractDatabase.
 */
public void setProperties(String s, String o, String t, String d,
                          String u, String p, boolean r, int n) {
  server = s;
  port = o;
  type = t;
  database = d;
  user = u;
  password = p;
  remember_password = r;
  number = n;
}
```

```
    /**
     * Sets the ResultSet for this AbstractDatabase
     * @return Boolean. TRUE if successful.
     */
    public boolean setResultSet(ResultSet r) {
      ...
    }

    /**
     * Checks if the last query was successful.
     * @return TRUE if the last query was successful.
     */
    public boolean succesfull() {
      return succes;
    }

    /**
     * Returns a human readable description of the database
     * connection.
     */
    public String toString() {
      ...
    }

    /**
     * Performs an update.
     * @param the SQL statement to be executed
     * @return true if statement was executed successfully
     */
    public boolean update(String updateStr) {
      ...
    }

} // End Of AbstractDatabase.java
```

DatabaseAbstractionLayer.java

```
package rug.edu.database;

import javax.swing.DefaultListModel;

public class DatabaseAbstractionLayer {

    /** current selected database                          */
    private AbstractDatabase adb = null;
    /** list of current connections                        */
    private DefaultListModel currentConnections = null;
    /** list of stored connections                         */
    private DefaultListModel storedConnections = null;
    /** list of registered database drivers                */
    private DefaultListModel registeredDrivers = null;
    /** default server type                                */
    private String defaultServerType = "AutoDetect";
```

```
/**
 * AbstractDatabase constructor
 * initializes the database abstraction layer,
 * creates the lists with the connections and drivers
 */
public DatabaseAbstractionLayer() {
  super();
  currentConnections = new DefaultListModel();
  storedConnections = new DefaultListModel();
  registeredDrivers = new DefaultListModel();
}

/**
 * Closes ALL open connections
 * This is important because most DBMSs' can handle
 * only a limited amount of open connections.
 */
public void close() {
  ...
}

/**
 * Code to perform when this object is garbage collected.
 * Any exception thrown by a finalize method causes the
 * finalization to halt. But otherwise, it is ignored.
 */
protected void finalize() throws Throwable {
  ...
}

/**
 * Returns the list of all database connections currently
 * available.
 */
public DefaultListModel getCurrentConnections() {
  return currentConnections;
}

/**
 * Returns the selected 'default' database. i.e.: the database
 * on which operations will be performed by default.
 */
public AbstractDatabase getDatabase() {
  return adb;
}

/**
 * Returns the default server type
 * The 'default server type' specifies the driver to be used by
 * default.
 */
public String getDefaultServerType() {
  return defaultServerType;
}

/**
 * Returns a list with the currently registered database drivers
 */
public DefaultListModel getDrivers() {
  return registeredDrivers;
}
```

```
/**
 * Returns the list of stored connections.
 */
public DefaultListModel getStoredConnections() {
  return storedConnections;
}

/**
 * Opens a connections with a database
 * The connection string specifies the server, database, username
 * and password to be used. The connection string is driver
 * dependent, see the documentation of the driver you want to use.
 * NOTE: normally users would use: open(String server,
 * String database, String user, String password)which
 * is an driver independent implementation of the open() method.
 * @return boolean, true if a connection can be established
 */
public boolean open(String connection_string) {
  ...
}

/**
 * Opens a connection with a database.
 * Users may supply the server, database, username and password to
 * use. When these parameters are not supplied the defaults will
 * be used.
 * NOTE: java >= 1.4.1 requires the use of a username/password
 * combination.
 */
public boolean open(String server, String database, String user,
                                                    String password) {
  return this.open(server, database, user,
                          password, defaultServerType);
}

/**
 * Opens a connections with a database
 * Users may supply the server, database, username, password and
 * driver to use.
 * When these parameters are not supplied the defaults will be
 * used. NOTE: normally users would use: open(String server,
 * String database, String user, String password) which is an
 * driver independent implementation of the open() method.
 */
public boolean open(String server, String database, String user,
                    String password, String driver) {
  ...
}
```

```
    /**
     * Try to find and preload a driver (or any other class)<br>
     * with a given classname.
     * @return boolean false on failure, true if succesful
     * @param String driver (classname)
     */
    public boolean registerDriver(String driver) {
      try {
        String package_name = this.getClass().getName().substring(0,
                        this.getClass().getName().lastIndexOf(".")+1);
        ClassLoader loader = this.getClass().getClassLoader();
        if (loader.loadClass(package_name + driver)!=null) {
          /* NOTE: method findClass not visible, using loadClass */
          /* instead. Alternative method:                        */
          /* Class.forName(package + driver).newInstance();      */
          registeredDrivers.addElement(new String(pkg_name + driver));
          System.err.println("Registering driver: " + driver);
        }
        return true;
      } catch (Exception e) {
        System.err.println("Unable to register driver: " + driver);
        return false;
      }
    }

    /**
     * Selects the default database to use.
     */
    public boolean selectDatabase(int index) {
      ...
    }

    /**
     * make a database object and store its properties for later
     * usage.
     * @param  properties; see AbstractDatabase.setProperties()
     * @param  int mode: defines the mode for storage
     *          0: only add when a similar entry does not exist
     *          1: add a new entry, even when a similar entry exists
     *          2: replace entry, when a similar entry exists
     * @return boolean false if entry could not be added or updated.
     */
    public boolean store(String server, String port, String type,
                String database, String user, String password,
                boolean remember_password, int mode) {
      ...
    }

} // End Of DatabaseAbstractionLayer.java
```

DatabaseTableModel.java

```
package rug.edu.database;

import java.io.*;
import java.sql.*;
import java.util.*;
import javax.swing.table.*;
import javax.swing.event.*;
```

```
/**
 * This class takes a ResultSet object and implements the TableModel
 * interface in terms of it so that a Swing JTable component can display
 * the contents of the ResultSet.
 * Note 1: This class requires a scrollable JDBC 2.0 ResultSet.
 * In case such a ResultSet is not available, the Database-
 * AbstractionLayer will automatically create a compatible table model.
 * Note 2: This class only provides read-only access to the created
 * table model.
 */
public class DatabaseTableModel implements TableModel {

      /** The ResultSet to interpret                          */
      ResultSet results;
      /** Additional information about the results            */
      ResultSetMetaData metadata;
      /** How many rows and columns in the table              */
      int numcols, numrows;

      public final static int CELL_MODE_PLAIN = 1;
      public final static int CELL_MODE_MAILING = 3;
      public final static int CELL_MODE_CUSTOMERS = 2;
      private int cell_mode = CELL_MODE_PLAIN;

      /**
       * This constructor creates a TableModel from a ResultSet.
       */
      DatabaseTableModel(ResultSet results) throws SQLException {
            this.results = results;                 // Save the results
            metadata = results.getMetaData();       // Get metadata on them
            numcols = metadata.getColumnCount();    // How many columns?
            results.last();                         // Move to last row
            numrows = results.getRow();             // How many rows?
      }

      public void addTableModelListener(TableModelListener l) {}

      /**
       * Call this when done with the table model.
       * It closes the ResultSet and the Statement object used
       * to create it.
       */
      public void close() {
        try {
            results.getStatement().close();
        } catch( SQLException e ) {};
      }

      /**
       * Automatically close when we are garbage collecting
       */
      protected void finalize() {
        close();
      }

      /**
       * return current cell mode
       */
      public int getCellMode() {
        return cell_mode;
      }
```

```
/**
 * This TableModel method specifies the data type for each column.
 * We could map SQL types to Java types, but there's no need for
 * that. All the returned data is converted to strings.
 */
public Class getColumnClass(int column) { return String.class; }

/**
 * These two TableModel methods return the size of the table
 */
public int getColumnCount() { return numcols; }

/**
 * This TableModel method returns columns names from the
 * ResultSetMetaData
 */
public String getColumnName(int column) {
  try {
    return metadata.getColumnLabel(column+1);
  } catch (SQLException e) { return e.toString(); }
}

/**
 * Returns the number of rows
 */
public int getRowCount() { return numrows; }

/**
 * This is the key method of TableModel: it returns the value at
 * each cell of the table. We use strings in this case. If
 * anything goes wrong, we return the exception as a string, so it
 * will be displayed in the table. Note that SQL row and column
 * numbers start at 1, but TableModel column numbers start at 0.
 * getValueAt can process and return results/cells in different
 * ways using setCellMode(). This allows getValue to return nice
 * String representations of the data.
 */
public Object getValueAt(int row, int column) {
  try {
      results.absolute(row+1);       // Go to the specified row
      Object o = results.getObject(column+1);
      if (o == null) {               // No data
        return null;
    } else {
          // Convert data to a nice string representation
        if (cell_mode == CELL_MODE_PLAIN) {
          // Default cell mode, simple data conversion
          return o.toString().trim();
        } else { ... // other cell modes }
  } catch (SQLException e) {
      System.err.println(e);
      return e.toString();
  }
}

/**
 * Our table is not editable
 * Returns false
 */
public boolean isCellEditable(int row, int column) {
  return false;
}
```

```
        public void removeTableModelListener(TableModelListener l) {}

        /**
         * set the cell mode for this table model
         * see: getValueAt()
         */
        public void setCellMode(int value) {
          cell_mode = value;
        }

        /**
         * Sets the contents of the specified cell.
         * Since its not editable, we don't need to implement this method
         */
        public void setValueAt(Object value, int row, int column) {}

} // End Of DatabaseTableModel.java
```

DatabaseTreeModel.java

```
package rug.edu.database;

import java.sql.*;
import java.util.*;
import java.awt.*;
import java.io.*;
import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.*;

/**
 * Creates and manages a tree model.
 * The DatabaseTreeModel enables a user to see a representation
 * of all available connections including its properties, databases,
 * tables and column names.
 */
public class DatabaseTreeModel extends javax.swing.tree.DefaultTreeModel
{
        // description of root
        public final static String ROOT_DESCR = "Connections";
        // description of information node
        public final static String INFO_DESCR = "Properties";
        // description of table node
        public final static String SCHEMA_DESCR = "Databases";

        private DatabaseAbstractionLayer dbl = null;
        private DefaultMutableTreeNode root = null;

        /**
         * DatabaseTreeModel constructor
         */
        public DatabaseTreeModel(javax.swing.tree.TreeNode root_node) {
          super(root_node);
          root = (DefaultMutableTreeNode)this.getRoot();
          this.buildTree();
        }
```

```
/**
 * DatabaseTreeModel constructor
 */
public DatabaseTreeModel(javax.swing.tree.TreeNode root_node,
                                        boolean asksAllowsChildren) {
  super(root_node, asksAllowsChildren);
  root = (DefaultMutableTreeNode)this.getRoot();
  this.buildTree();
}

/**
 * DatabaseTreeModel constructor
 * <br>
 */
public DatabaseTreeModel(DatabaseAbstractionLayer d) {
  super(new DefaultMutableTreeNode(ROOT_DESCR));
  root = (DefaultMutableTreeNode)this.getRoot();
  dbl = d;
  this.buildTree();
}

/**
 * Adds a single database connection to the tree
 */
public void addConnection(int index) {
  ...
}

/**
 * Builds an entire tree based on all current connections
 */
public void buildTree() {
  if (dbl!=null) {
    for (int i=0; i<dbl.getCurrentConnections().getSize(); i++) {
       addConnection(i);
    }
  }
}

/**
 * Cleans an entire tree by removing all its children
 */
public void cleanTree() {
  root.removeAllChildren();
  this.reload();
}

/**
 * Create a new custom tree renderer for displaying
 * alternative icons. Use: tree.setCellRenderer
 * to set the cell renderer.
 * @return javax.swing.tree.DefaultTreeCellRenderer
 */
public DefaultTreeCellRenderer getCellRenderer() {
  DefaultTreeCellRenderer renderer = new
                              DefaultTreeCellRenderer();
  renderer.setOpenIcon(null);
  renderer.setClosedIcon(null);
  renderer.setLeafIcon(null);
  return renderer;
}
```

```
    /**
     * Removes a single database connection from the tree<br>
     *
     */
    public void removeConnection(int index) {
        root.remove(index);
        this.reload();
    }

} // End Of DatabaseTreeModel.java
```

References

[1] The generated Java documentation (JavaDoc):
http://www.fmf.nl/~eelco/afstuderen/files/javadoc/index.html

[2] The project page:
http://www.fmf.nl/~eelco/afstuderen/

[3] The source code can be downloaded directly from:
http://www.fmf.nl/~eelco/afstuderen/files/source.zip