The DAME Primer

DAME - Database Access Made Easy

Dakshinamurthy Karra
Subex Systems Ltd.

No. 721, 7th Main, Mahalaxmi Layout
Bangalore
India
+91-80-3497581
+91-80-3491490
kd@subexgroup.com

The DAME Primer: *DAME* - Database Access Made Easy
by Dakshinamurthy Karra

The DAME Primer Version 1.0 for DAME 1.0 Edition
Published 2003

Dedication

> You teach best what you most need to learn
>
> —Bach, Richard

For all those who taught me by being my teachers and by being my students.

Table of Contents

Acknowledgements

Any software development is a colloborative process. We make use of any number of tools to generate the software we write (like Linux™, GCC etc.). This section is to acknowledge all the people directly or indirectly involved in making this software development a reality and easy (not the obvious ones).

- Subex Systems Ltd.(http://www.subexgroup.com/)

  First of all I would like to thank Subex for giving me permission to make this software public.

- Antlr(http://www.ANTLR.org/)

  Antlr is a parser generator. The only reason I switched to Antlr (from yacc/lex) was that Antlr supports C++ code generation. I do not think I will ever go back to using lex/yacc for developing these kind of applications.

- Extreme Programming(http://www.extremeprogramming.org/)

  Extreme Programming is an agile methodology that is very rapidly gaining ground with good reasons. This project is started in the hope that I will get more clarity on C++ refactorings and learn STL on the way. I believe I achieved both. A great thanks to all XPers who worked and made up this great methodology. "XP is common sense." This is the best compliment I could think for this methodology.

- CPPUnit(http://cppunit.sourceforge.net)

  You can't even dream of XP without a xUNIT test suite for the language you work in. Folks at CPPUnit did a great job on this one.

- Leak Tracer(http://www.andreasen.org/LeakTracer/)

  With Test First Design, I thought I do not need a leak checker. I am proven to be wrong.... Leak Tracer came to the rescue.

Chapter 1. Introduction To *DAME*

1.1. What Is *DAME*?

DAME stands for “Database Access Made Easy” and is a tool to generate C++ code around SQL statements. *DAME* can support almost any type of SQL statements - DML or DDL. This chapter explains what *DAME* is, how it relates to other database access mechanisms and why you might want to use it. We also cover how to configure and install *DAME*. At the end of the chapter, we will go through the example schema that is used for examples in this book.

1.2. History Of *DAME*

Few years back our organization is working on an application that accesses database frequently. The software is being developed using C++. The initial phase of development was smooth. However, soon we realized that our database access code is boring to write, tedious to debug and more or less horrible to look at. Most of the problems can be attributed to the following reasons:

- Most of the code is duplicated. Not much of thinking goes into writing the access code once the SQL has been decided.
- The database access libraries provides interfaces in terms of the low level language constructs like char*, int etc. There are times developers forget to put an & before a variable name!.
- Where as our wrapper classes provides the necessary abstraction to access the database, the abstraction is not high-level enough for us to enjoy writing code.
- Still the database access is in terms of columns. It is upto the developers to put all columns together and make a C++ class out of it.

1.2.1. Solution

We developed a small tool (was called SQLparser) to generate wrapper code to C++ classes. This tool reads code written in a specific format (more like PL/SQL code) and spits out C++ class code that can access the database. We found the tool to be very useful in our regular development. We reduced the LOC of database access by almost 1/10th.

It is time to revise SQLparser and make it more useful. There were problems with the original tool - the tool accepts format that looks like english and most of us are

C++/Java developers. Another problem is that SQLparser always insists that we pass an object even when we want to get a single integer from the database. The tool generated code is not generalized enough. For example, if we want to get each record from a table and do some processing, the only way to achieve was to get the result in a vector, process and delete the vector. Lastly, I am hearing about Antlr and it is time I have a look at it ;-).

The end result is *DAME* - Database Access Made Easy. This version removes almost all the problems that existed with SQLparser. I enjoyed developing it and we started to use it in one of our projects already. I hope you enjoy using it.

1.3. Why Use *DAME*?

If the previous section feels like what you are going through now you need *DAME* ;-). With *DAME* it is easy to generate database access code for any application level C++ class. Since *DAME* works at the SQL statement level, it is very flexible. *DAME* is a command line tool and can easily be integrated into any development environment. Using *DAME* consistently in a project allows the database access code to be segregated from the application code. This hopefully will make your application easier to port onto different database systems.

1.3.1. Limitations of *DAME*

DAME does not read database schema from your database to automatically generate code. I am not very sure how helpful this will be in an application with more than 10 tables.

DAME does not support (at this time anyway) any kind of large objects (LOB, BLOB, CLOB). If your application makes use of LOBs then you can’t use *DAME* for that part of database access.

DAME support for datetime fields is not upto the mark (Yet). You need to get the datatime fields from the database into character strings of the format: “DD/MM/YYYY HH24:MI:SS”. The user classes are supposed to have "struct tm" variables available to process the return values or pass datetime values to database. I thought of using Boost DATE/TIME classes, but doing so adds another dependency on to the software, so I rejected that option.

DAME currently supports only C++ code generation using in-built libraries for Oracle™ and *Postgresql*. Soon *DAME* should support Java code generation using JDBC and MySQL.

DAME does not have support for NULL values. Without going into the details, I personally did not feel the need for NULL values in an application and that reflects in *DAME*

1.4. Configuring And Installing *DAME*

Installing *DAME* is not very a complicated task. You need to have Antlr and CPPUnit installed on your system before you can install *DAME*. *DAME* uses Antlr version 2.7.1 and CPPUnit 1.8.0. Any later versions of these tools might be OK for installing *DAME*.

1.4.1. Installation

1.4.1.1. Checking Dependencies

Antlr

> java antlr.Tool
>
> Should give the version number. The version should be >= 2.7.1

GCC 3.x

> g++ --version
>
> Should print the version number (with whole lot of other information). The version should be >= 3.0

CPPUnit

> Check whether the libcppunit is available for linking. You need the header files of CPPUnit accessible to the *DAME* compilation process.

Database

> *DAME* needs atleast one supported Database to be of use for development. At the time of writing this Primer *DAME* has support for Oracle™ 8i/9i and *Postgresql*. The configuration script for *DAME* looks at ORACLE_HOME environment variable for compiling Oracle™ support. Similarly *DAME* configuration script checks and executes `pg_config` script to build support for *Postgresql*.

1.4.1.2. Installation

1. ./configure
2. make
3. su
4. make install

1.5. Example Schema

This document refers to Employee and Department tables as defined in the following schema:

`postgres.sql`

```
        CREATE TABLE DEPT
                (DEPTNO NUMERIC(2) CONSTRAINT PK_DEPT PRIMARY KEY,
                 DNAME VARCHAR(14) ,
                 LOC VARCHAR(13) ) ;
        CREATE TABLE EMP
               (EMPNO NUMERIC(4) CONSTRAINT PK_EMP PRIMARY KEY,
                ENAME VARCHAR(10),
                JOB VARCHAR(9),
                MGR NUMERIC(4),
                HIREDATE DATE,
                SAL NUMERIC(7,2),
                COMM NUMERIC(7,2),
                DEPTNO NUMERIC(2) CONSTRAINT FK_DEPTNO REFERENCES DEPT);
```

The schema along with the example data loading are available in the `examples/` directory of the source distribution.

Chapter 2. Getting Started With *DAME*

In this and following chapters, I will assume that you have a database setup using the schema provided under `examples/` directory of the *DAME* source distribution. In this chapter we walk through a sample program developed using *DAME*.

> **Assumptions::** It is assumed that appropriate PATH and LD_LIBRARY_PATH variables have been setup so that *DAME* can be executed from the commandline. If you install *DAME* into the default location, there might not be any need for setting up of environment. Similarly, *DAME* Oracle™ interface library uses Oracle™'s OCI interface (library `libclntsh.so`) and you might have to setup the LD_LIBRARY_PATH to point to Oracle™'s `lib` directory.

2.1. A Simple Dame Program

Look at the following program listing (`simple.dame`).

`simple.dame`

```
class CountDB
{
    int         Count ;

    /*
     * Select the total number employees in the employee tabl
     */

    Count GetEmployeeCount (void) {
        select count(*)
        from emp
    } ;

} ;
```

This program is defining a class called `CountDB`. The `CountDB` class has only one interface, `GetEmployeeCount` that takes no parameters and returns the variable `Count`.

Warning

The function `GetEmployeeCount(void)` defines a function that does not need any parameters. In *DAME* the semantics of `GetEmployeeCount(void)` are *not* same as `GetEmployeeCount()`. The details will be discussed in next chapter.

Executing *DAME* on this file produces files: `CountDB.cpp` and `CountDB.hpp`.

```
shell> dame simple.dame
DAME - Database Access Made Easy 1.0
shell> ls Count*
CountDB.cpp  CountDB.hpp
shell>
```

2.1.1. Accessible Interfaces In CountDB

The way `CountDB` is written we have only the following interfaces generated:

```
class CountDB {
   CountDB (Dame::DatabaseIf* db) ;
   ...
   template<Iterator>
   void GetEmployeeCount(Iterator it) ;
}
```

So we can create a `CountDB` object by passing its constructor a `DatabaseIf` pointer and then use the `GetEmployeeCount` function to access the data. How do we get a `DatabaseIf` object? The next section explains that part.

2.1.2. Creating A DatabaseIf Object

DAME tries to be generic. All the code generated by *DAME* makes use of interface objects. The gateway to the database access classes in *DAME* is `DatabaseIf`. *DAME* support libraries for C++ provides mechanisms for creating the `DatabaseIf` objects. You can create a Oracle™ `DatabaseIf` by using the following code fragment.

`example.cpp`

```
#include "dame/oradatabase.hpp"
...
```

```
            char *user = "scott" ;
            char *password = "tiger" ;
            char *database = "dame" ;
            Dame::DatabaseIf*   db = new Dame::Oracle::OraDatabase (user,
```

Similarly for *Postgresql*:

`example.cpp`

```
            #include "dame/pgdatabase.hpp"
            ...
            char *user = "scott" ;
            char *password = "tiger" ;
            char *database = "dame" ;
            Dame::DatabaseIf*   db = new Dame::Postgres::PgDatabase (user,
```

2.1.3. What is Iterator?

Now that we understand how we can create a pointer to `DatabaseIf` we are one step closer to using the `CountDB` class. Before we can make use of `GetEmployeeCount` we need to understand what the template parameter `Iterator` is.

When the function `GetEmployeeCount` is called, in turn this function calls the `Iterator` object using the operator () and passing the return value (in this case it is `Count` ie. int). You can do any thing with the value as you like and return back to the calling function `GetEmployeeCount`. Not really anything.. but most of the things.

So how do we create a Iterator object? If you are new to STL (I suggest that you get familiarized with STL anyhow, if not you are losing half the fun using C++), the following code fragment looks strange. But, it does what it needs to be done and get us the results:

`example.cpp`

```
            int
            Print (int value)
            {
                std::cout << "Number of Employees: " << value << std::endl
                return 0 ;
            }

            ...
                CountDB     cdb (db) ;
                cdb.GetEmployeeCount (std::ptr_fun (&Print)) ;
            ...
```

2.1.4. Putting It All Together

Putting all these things together here is our first program. A C++ preprocessor Macro controls whether the code is generated for Oracle™ or *Postgresql*.

```
example.cpp

#include <unistd.h>
#include <stdlib.h>

#include <exception>
#include <iostream>
#include <iomanip>
#include <vector>

#include <dame/utility.hpp>
#include "CountDB.hpp"

#ifdef ORACLE
#include "dame/oradatabase.hpp"
#else
#ifdef POSTGRES
#include "dame/pgdatabase.hpp"
#else
#error "A database need to be defined"
#endif
#endif

int
Print (int value)
{
std::cout << "Total number of employee records: " << value << std::endl ;
return 0 ;
}

int
main(int argc, char *argv[])
{
char *user = "scott", *password = "tiger", *database = "dame" ;

try {
#ifdef ORACLE
Dame::Oracle::OraDatabase db (user, password, database) ;
#else
Dame::Postgres::PgDatabase db (user, password, database) ;
#endif
CountDB                 edb (&db) ;
edb.GetEmployeeCount (std::ptr_fun(&Print)) ;
} catch (std::exception& e) {
std::cerr << e.what() << std::endl ;
exit (1) ;
}
exit (0) ;
}
```

Chapter 3. Selecting Information From Databases

Select statements are the most commonly used SQL statements. This chapter shows you how you can make use of *DAME* to access the database using select statements. First of all we revisit the earlier example and modify it to get the value into a local variable. Then we will look into an example using a C++ class and initializing the class using *DAME* by accessing the data from database. We will look under the hood of the code generated by *DAME* to understand how best we can make use its capabilities. We will look into some examples where a select returns multiple records. We will also look at some standard containers and how *DAME* can be used along with these. Finally, we round-off this discussion with constructing a `std::map` object using *DAME*

3.1. Revisiting The `GetEmployeeCount` Example

In the previous example, we used a function to use the return value from our select statement to print the number of employees from the database. However, in real-life situations we might like to get the return value in to a variable and make use of the value at some other times. For the sake of clarity the code from the previous example is reproduced:

```
template<Iterator>
GetEmployeeCount (Iterator it) ;

edb.GetEmployeeCount (std::ptr_fun(&Print)) ;
```

For saving the information into a variable we need a object that can support `operator() (int)`. The *DAME* `utlity` consists of an *adapter* named `make_single` that does exactly this.

`example1.cpp`

```
#include <dame/utility.hpp>
...
int count ;
edb.GetEmployeeCount (Dame::make_single (&count)) ;
```

3.2. Using *DAME* With C++ Classes

DAME will not be useful if it returns single values from select statements. In most cases we will be required to select one or more rows from the database tables. For example, we might want to get an employee record given an employee ID. The next example shows how we can achieve this.

3.2.1. The *DAME* Source

`example2.dame`

```
options {
    header_suffix = ".hpp",
    source_suffix = ".cpp",
    strip_first_char = "false",
    lower_case_file_names = "false",
    use_namespace = "Dame::Example"
}

header {
    #include "emp.hpp"
}

class SelectDB (Dame::ExampleDb::emp)
{
    int         Empno ;
    char        Name[11] ;
    char        Job[10] ;
    int         Manager ;
    double      Salary ;
    double      Commission ;
    int         Deptno ;
    int         Count ;
    date        Hiredate ;

    /*
    * Simplest form of select statement can be used as fo
    */

    Dame::ExampleDb::emp GetByEmpno (Empno) {
        select empno, ename, job, coalesce(mgr,0), sal, co
        into :Empno, :Name, :Job, :Manager, :Salary, :Comm
        from emp
        where empno = :Empno
    } ;
    ...
} ;
```

The options section in the above example is selfexplanatory. The strip_first_char option exists, so that we can name our classes like `CEmployee` and still able to get file names like `Employee`. *DAME* has support for namespaces. The `emp` is expected to be in `Dame::ExampleDb` namespace. The `SelectDB` class will be generated in the namespace `Dame::Example`. Note the *into* to collect the column values into the local variables. The into-list is not needed when we are selecting a single column value using the SQL statement as in earlier examples.

DAME expects the `Dame::ExampleDb::emp` to be defined. The definition should have setters/getters in the form of Get<variable> and Set<variable> for all the member variables that need to be accessed from the defined database access class `Dame::Example::SelectDB`.

Since the generated code now needs to access to the definition of `emp`, we include the definition using the header directive. All the text within the '`{`' and '`}`' of the header directive will be copied verbatim into the resulting Header files.

Note the declaration for the `SelectDB` class. Now we tell *DAME* to make use of the `emp` declaration while generating the code. Besides this, the only change is in the return value of the function `GetByEmpno`. We are now informing *DAME* that we need an `emp` record to be returned.

3.2.2. Accessing the `emp` record

Whenever the `GetByEmpno` is called, the function in-turns calls the Iterator object that is passed. The Iterator object will be passed a pointer to an `emp` record. When the call returns, the pointer is deleted unless and otherwise told not to do so.

We can pass a function pointer using `std::ptr_fun` if some function definition is similar to:

```
int SomeFunction (Dame::ExampleDb::emp*) ;
```

The following code stores the pointer in a variable and prints the details using that variable.

`example2.cpp`

```
Dame::Example::SelectDB                edb (&db) ;
Dame::ExampleDb::emp*                  ep ;
edb.GetByEmpno (7369, Dame::make_single(&ep)) ;
if (ep)
    ep->Print() ;
```

```
                else
                    std::cout << "Record does not exist" << std::endl ;
```

3.3. Under The Hood Of Code Generated By *DAME*

DAME uses the Iterator object whether the select statement being processed returns single or multiple records. The basic mechanism by which *DAME* passes the rows selected is through the Iterator object. *DAME* generates code that uses the Iterator object as follows:

```
template<Iterator>
GetByEmpno (int _p_Empno, Iterator it)
{
    ...
    GetByEmpno_next (it) ;
}

template<Iterator>
GetByEmpno_next (Iterator it)
{
    ...
    do {
    Dame::Example::emp* _l_forClass = new Dame::Example::emp () ;
    _l_for_Class->SetEmpno (...)
    ... More set methods to fill the local variable
    int retval = it (_l_for_Class) ;
    if ((retval & keepPtr) == 0)
        delete _l_for_Class ;
    if (retval & abortRetrieval)
        return ;
    } while (there are more records) ;
}
```

The point to be noted above is that *DAME* calls the Iterator object's `operator()` with a pointer to an object. The called function can return `DatabaseIf::keepPtr` if it wants to have the ownership of the object created within the `GetByEmpno_next`. If more records are not expected, the called function can return `DatabaseIf::abortRetrieval`, in which case the loop is terminated. The application can also call directly the `GetByEmpno_next` for retrieving more records, once the `GetByEmpno` is called.

This mechanism offers the greatest flexibility. The *DAME* standard `adapter` is an example where the flexibility is used.

`dame/utility.hpp`

```
            template<typename T>
            class single_value_t : public std::unary_function<T, int>
            {
                T*      ptr ;
                bool*   isInvoked ;
            public:
                single_value_t (T* _ptr, bool* _isInvoked)
                    : ptr(_ptr), isInvoked(_isInvoked) {
                    *ptr = (T) 0 ;
                    if (isInvoked)
                        *isInvoked = false ;
                }
                int operator() (T val) {
                    if (isInvoked)
                        *isInvoked = true ;
                    *ptr = val ;
                    return DatabaseIf::keepPtr | DatabaseIf::abortRetrieva
                }
            };

            template<typename T>
            single_value_t<T>
            make_single (T* ptr, bool* isInvoked = NULL) {
                return single_value_t<T>(ptr, isInvoked) ;
            }
```

In case of single value returns, the `operator()` is returning the value `DatabaseIf::keepPtr | DatabaseIf::abortRetrieval`, because the application would like to use the pointer later. Since we need only a single record (`make_single` right?), we are passing the `DatabaseIf::abortRetrieval`.

3.4. Selecting Multiple Records

As discussed above, *DAME* uses the same syntax to access a single row or multiple rows from a table. For example, Let us see how we can we print all the employees who are in the same department as given employee.

First the *DAME* source:

`example2.dame`

```
        Dame::ExampleDb::emp GetByDeptno () {
                select empno, ename, job, coalesce(mgr,0), sal, coalesce(c
                into :Empno, :Name, :Job, :Manager, :Salary, :Commission,
                from emp
                where empno <> :Empno and
                deptno = :Deptno
        } ;
```

Note the declaration of function `GetByDeptno`. We do not want to write `GetByDeptno(void)` since we want to pass the department number and employee number through an `emp` record.

The code for performing this function is shown below:

`example3.cpp`

```
            Dame::Example::SelectDB                 edb (&db) ;
            Dame::ExampleDb::emp*                   ep ;
            edb.GetByEmpno (7369, Dame::make_single(&ep)) ;
            edb.GetByDeptno (*ep, std::mem_fun(&Dame::ExampleDb::emp::Pri
```

We made use of STLs `std::mem_fun` adapter to converter `emp::Print` to a function of type `Print(emp*)`.

3.4.1. Selecting Records Into Standard Containers

Standard STL containers comes in multiple flavors. Some of the standard containers are `vector, list, deque, set, map`. For *DAME* to be useful we should have mechanisms wherein the selected rows are converted into objects and stored in different types of containers. The standard STL defines three types of `insertors`. A front insertor that uses `push_front` to add an element into a container, a back insertor that uses `push_back` to add an element into a container and a normal insertor that uses `insert` to add an element into a container.

Corresponding to each of the insertors, *DAME* `utility.hpp` provides three adapter functions. Any standard container can be used along with these adapters.

The following program shows how we can make use of the standard containers `std::vector, std::list` to make use of *DAME*s capabilities. This program makes use of the same `GetByDeptno` defined in the ealier example.

`example3.cpp`

```
                std::vector<Dame::ExampleDb::emp*>      vep ;
                std::list<Dame::ExampleDb::emp*>        lep ;
                edb.GetByDeptno (*ep, Dame::make_back_insertor(vep)) ;
                std::cout << "Vector size = " << vep.size() << std::endl
                edb.GetByDeptno (*ep, Dame::make_front_insertor(lep)) ;
```

```
std::cout << "List size = " << vep.size() << std::endl ;
```

3.4.2. Selecting Records Into Associative Containers

Where as selecting into containers like `vector, list` is straight forward, selecting into associative containers like `std::map, std::set` requires some efforts from the users of *DAME*. Suppose we want find department wise salary totals for all employees.

First the *DAME* source:

`example2.dame`

```
Dame::ExampleDb::emp GetByDeptno () {
    select empno, ename, job, coalesce(mgr,0), sal, coales
    into :Empno, :Name, :Job, :Manager, :Salary, :Commiss
    from emp
    where empno <> :Empno and
    deptno = :Deptno
} ;
```

The Iterator class that is required to access to create the map and its usage is shown below:

`example4.cpp`

```
class Mapper
{
    std::map<std::string, double>&      m ;
public:
    Mapper (std::map<std::string, double>& _m)
        : m(_m) {
    }
    int operator() (Dame::ExampleDb::emp* e) {
        m[e->GetDeptname()] = e->GetSalary() ;
    }
} ;
...
...
Dame::Example::SelectDB                 edb (&db) ;
std::map<std::string, double>     smap ;

edb.GetSalByDept (Mapper(smap)) ;
```

> **Bad design!!!:** In a real life project using the `emp` class to retrieve the grouped by salaries might be considered as bad design choice. I did so that I do not write another sample program. Do not do this in your regular code.

3.4.3. Selecting Records In Chunks

DAME provided adapter `make_XXX_insertor` accepts a second parameter. The parameter controls the number of records that are fetched each time. You can use the `_next` functions to retrieve the next set of records as shown below:

```
std::vector<Dame::ExampleDb::emp*>      vep ;
GetByDeptno (ep, Dame::make_back_insertor(vep, 5)) ;
..... Fetched 5 records and processed
GetByDeptno (ep, Dame::make_back_insertor(vep, 5)) ;
..... Fetch next 5 records.
```

DAME does not provide any indication at the end of the record set. You need to check the size of the container or use some other mechanism to perform this check.

Chapter 4. Updating Database Tables

A whole lot of other SQL statement types are also supported by *DAME*. You can generate code to insert, update and delete rows from databases. You can also use *DAME* for DDL statements like *Create Table*. However, I strongly suggest against using *DAME* for DDL as it might be much more easier to provide a set of SQL scripts to perform the same tasks.

In this chapter, we will have a look at general syntactic considerations for *DAME*s support for non-select SQL statements.

4.1. Example of *DAME* Source

```
nonselect.dame

options {
header_suffix = ".hpp",
source_suffix = ".cpp",
strip_first_char = "false",
lower_case_file_names = "false",
use_namespace = "Dame::Example"
}

header {
#include "emp.hpp"
}

class NonSelectDB (Dame::ExampleDb::emp)
{
int         Empno ;
char        Name[11] ;
char        Job[10] ;
int         Manager ;
double      Salary ;
double      Commission ;
int         Deptno ;
int         Count ;
char        Deptname[16] ;
date        Hiredate ;

/*
 * Inserting an emp record
 */

void Insert () {
```

```
insert into emp (empno, ename, job, mgr, sal, comm, deptno, hiredate)
values (:Empno, :Name, :Job, :Manager, :Salary, :Commission, :Deptno, to_d
} ;

void UpdateByEmpno () {
update emp
set ename = :Name, job = :Job, mgr = :Manager, sal = :Salary, comm = :Comm
hiredate = to_date (:Hiredate, ’dd/mm/yyyy hh24:mi:ss’)
where empno = :Empno
} ;
} ;
```

As you can observe, the usage of *DAME* for insert/update and delete statements is very similar to that of select statements. One major difference is that all of these functions returns `void`. The parameters can be passed either through `emp` object or through the function declarations.

Chapter 5. Using Transactions

DAME has simple support for transactions through the `DatabaseIf` interface. However, the transaction semantics differs from one database system to another database system. For example, In Oracle™ a database connection can start a transaction and still commit chosen SQL statements, where as *Postgresql* does not support such flexibility. *DAME*s support for transactions is rudimentary in nature, but still useful and enough for a lot of applications.

5.1. `DatabaseIf` Support For Transactions

Every database connection supports a *autocommit*. By default, the autocommit for a database connection is set to true. Every SQL statement executed using this database connection is committed to the database and immediately visible to all other connections.

You need to turn the autocommit off using `SetAutoCommit` before you can make use of transaction capabilities. You need to use `StartTransaction` to start a transaction. It is expected that a transaction is always committed or rollbacked using functions `Commit` and `Rollback`.

5.1.1. Transaction Semantics For Oracle™

DAME library for Oracle™ uses the default transaction created by Oracle™ OCI call to open a database connection. If the autocommit flag is turned on after a transaction is started all the database updates from then on are *Committed* to the database till the flag is turned off. One still need to commit or rollback the transaction to commit/rollback changes that were made while the autocommit flag is turned off.

5.1.2. Transaction Semantics For *Postgresql*

DAME library for *Postgresql* uses the "BEGIN WORK" SQL statement to start a transaction. The autocommit flag does not have any bearing on the database updates that take place during the transactions.

5.2. Suggested Use Of Transactions

Use the KISS (Keep It Simple, ...) principle for using transactions. You can follow the following guidelines, if you want to use transactions in your application.

1. Be consistent in usage of transactions. If you decide to use transactions in your application, do so in all the modules.
2. Turn off the AutoCommit flag as soon as the database connection is created.
3. Do not forget to use `Rollback` and `Commit` functions.
4. Do not use your understanding of Database specific transaction semantics.. like Oracle™s ability to commit selective SQL statements even when the transaction is on. This will surely hurt you in future, if you need to port your application to another database.

Appendix A. Understanding *DAME*

This chapter gives more details about how *DAME* is implemented. We use the *DAME* syntax to walk through the functionalities of the application. This is *not* a formal syntax for *DAME* language. It is better to go through the sources and Antlr file `src/dame/dame.g` to understand the syntax. In this chapter we go through the *DAME*s major syntactical elements to understand the semantics better. I hope that whatever information I missed out in the earlier parts of this Primer I will be able to capture here.

A.1. *DAME* Syntax

A.1.1. The *DAME* Script File

script_file : (option_list)? (header)? (class_definition)*

The dame source file consists a set of options, a header that needs to be output verbatim into the header files generated and zero or more class definitions. According to this syntax an empty file is also a valid *DAME* source!. However, the source files are generated for each of the class definitions, so even if you have a proper *DAME* source file without class definitions, *DAME* can not do anything with it.

> **Future Additions:** Currently with *DAME* you need to reproduce the options in each of the source files even though they are same. This will soon change and *DAME* will be able to pick default options from multiple sources (`/etc/dameoptions`, `$HOME/.dameoptions` and `$CURRENTDIR/.dameoptions`).

The options are a comma seperated list of the form variable = value. The value is always enclosed in "". The boolean options should be provided with values "true" or "false". *DAME* does not process anything in the raw_header. It just strips the initial '{' and '}' and writes it verbatim into the each header files it generates. The following options are supported by *DAME*

header_suffix and source_suffix

Suffixes to be used while generating C++ source code.

strip_first_char

> Some coding standards insist that we prefix a ’C’ before every class name. This option when set to true strips the first character from the class names and uses it for file name generation.

lower_case_file_names

> Converts all characters in the class name to lower case and uses it as the filename.

use_namespace

> This string is a standard C++ namespace string. The classes generated will be put into this namespace. Highly recommended to avoid namespace pollution.

A.1.2. Class Definitions

class_definition: CLASS className (LPAREN for_class RPAREN)? LCURLY declaration_list function_list RCURLY SEMI

There are two types of classes that can be generated using *DAME*. One that can be used stand alone and one for which we will be making use of another associated class (for_class in the grammer). When for_class is omitted, you are expected to pass parameters to each of the functions you create - a void is also a valid parameter. When for_class was given a function declaration with no parameters assumes that the parameters are passed through a const for_class pointer.

The for_class is expected to be a regular C++ class. The class should implement accessors (settors and gettors) for each of the variables that are used in the *DAME* generated class. All getters start with Get and setters with Set.

The declaration list is a semi-colon seperated list of variable declarations. Unlike in C/C++ you can not declare more than a single variable in each statement. *DAME* supports the following data types:

int

> Any integer type that is stored in the database can be used for this type. It includes `NUMERIC`, `NUMBER` with zero precision along with `INTEGER`, `SMALLINT`. This corresponds to C++ datatype `int`.

double

> Any floating type that is stored in the database can be used for this type. It includes `NUMERIC`, `NUMBER`. This corresponds to C++ datatype `double`.

char

> As with any database schema, the size of the character buffer is required when declaring a string. This can be any character string representation supported by the database like `VARCHAR2`, `VARCHAR`, `char` . This corresponds to C++ STL's `std::string`.

date

> This data type represents the datetime field in the database. This corresponds to Posix time structure `struct tm` . *DAME* does not support reading date and time columns from the database. The date and time fields are converted into character strings before used by *DAME*. Hence any access to the date/time fields should convert the fields to/from character strings using the database provided functions like `to_char` and `to_date`.

The declaration list is followed by one or more function definitions. The function definitions are the topic of next section.

A.1.3. Function Definitions

function_def: (for_class | VOID) IDENTIFIER LPAREN (param=id_commalist)? RPAREN LCURLY sql_statement RCURLY SEMI

A function definition always has a return type. The return type is either one of the variables that existed in the declaration list or the associated class defined for the *DAME* class (for_class). *DAME* will throw up an error, in case the return value is not a valid variable. The return value can also be void. If the sql_statement is a select statement and the return type is void, then *DAME* will assume that return type is for_class. Do not get confused by the return type of a function in the *DAME* source. The generated code always create functions that return void. All the errors are propogated using exceptions.

The parameters passed (id_commalist) is either a single word "void" or a comma seperated list of variables that exist in the declaration section of the class definition. If the parameter passed is void, the sql_statement can't make use of any variables from the declaration list. If the parameter list is empty, *DAME* assumes that the parameters are passed using the associated class (for_class).

The next section describes the sql_statement. The semi-colons at the end of function definitions are noise - but are required.

A.1.4. The SQL Statement

sql_statement: (.*) (INTO target_comma_list)? FROM (.*)

DAME does not parse the SQL statement. Any syntactical errors or semantic errors (like table does not exist) were returned to the application at run time using exceptions. *DAME* removes the part from "into" till "from" from the sql_statement before it is passed to the database. *DAME* parses this part of the statement for a list of variables (prefixed by ’:’). As a special case, a select statement that retrieves a single column need not have a INTO part, if the return type is specified in the function definition.

> **Future:** Future versions of *DAME* will provide for replacements of regular expressions in an sql_statement while generating code. This rewrite engine might help us to write sql_statement for one specific database and use for multiple other databases using a map database. For example, Oracle™ 9i supports `COALESCE` where as Oracle™ 8i does not. In this case the map database entry can map `coalesce` into an equivalent using `nvl`.

Appendix B. *DAME* Database Interfaces

DAME can be used with multiple databases. This is possible because *DAME* generates code that uses a set of interfaces. The *DAME* library for Oracle™ and *Postgresql* implements these interfaces. Even though only the `DatabaseIf` is visible to user of *DAME* application there are two interfaces that are available through *DAME* - `DatabaseIf` and `RecordSetIf` This chapter shows you how you can make use of these interfaces.

B.1. The `DatabaseIf` Interface

The `DatabaseIf` is the primary interface available to a particular database. All the associated objects are created using the `DatabaseIf`. Let us look at the `DatabaseIf` definition.

`src/include/dame/databaseif.hpp`

```
class DatabaseIf {
...
    virtual RecordSetIf* CreateRecordSet (const std::string& s
    bool SetAutoCommit (bool bCommit) ;
    bool GetAutoCommit() const ;
    virtual void Commit (void) = 0 ;
    virtual void Rollback (void) = 0 ;
    virtual void StartTransaction (void) = 0 ;
...
```

A `DatabaseIf` object models a connection to the database in real world. The construction of this object will be different for different databases as we have seen in the examples earlier. A user can execute multiple SQL statements using the same database connection. For each such SQL statement the user requests the `DatabaseIf` object to create a record set object. A constructed of `DatabaseIf` object should have a open database connection associated with it.

B.2. The `RecordSetIf` Interface

The record set models a particular SQL statement that can be executed on a database. The `RecordSetIf` objects are created using the `CreateRecordSet` provided by the `DatabaseIf` interface. There can't be any `RecordSetIf`s that are not associated with an open database connection obtained through `DatabaseIf`.

`src/include/dame/recordsetif.hpp`

```
            class RecordSetIf {
            ...
        virtual void Execute (void) = 0 ;
        virtual void SetColumnBuffer (int index, Util::SQLString buffer) =
        virtual void SetColumnBuffer (int index, int *buffer) = 0 ;
        virtual void SetColumnBuffer (int index, double *buffer) = 0 ;
        virtual void SetParameterBuffer (const char* name, const char *bu
        virtual void SetParameterBuffer (const char* name, const int *buf
        virtual void SetParameterBuffer (const char* name, const double *
        virtual bool End() = 0 ;
        virtual void Next() = 0 ;
```

The `RecordSetIf` interface provides functionality to execute a SQL statement onto the database. The user is expected to use `SetColumnBuffer` and `SetParameterBuffer` to attach buffers. The column buffers are used by the `RecordSetIf` to return back values that are retrived from the database. The parameter buffers are used to pass values to the SQL statement.

Note that there are no interfaces defined to retrieve the datetime fields. The date/time column types are handled by *DAME* using the helper class `Dame::Util::Date`.

Note also that there are no transaction semantics that are enforced by the `RecordSetIf` class. The transactions are associated with each connection of a database and so are handled by `DatabaseIf`

Appendix C. Extending *DAME*

DAME is an open source project. There are multiple ways in which *DAME* can be extended. Few of the future directions I can think of, I will list here.

- Support for more databases. Currently *DAME* supports Oracle™, *Postgresql*. *DAME* support for MySQL is being worked upon. There are other databases out there and *DAME* can be extended to support those databases.

  Support for other databases is easy. We need to implement the interfaces `DatabaseIf` and `RecordSetIf` and *DAME* can also be used for that database.

- Support for more languages. Java support on the way. It is possible to extend *DAME* to support other languages as well C, C# etc.
- Once the SQL rewrite engine (see earlier chapter) is ready, the usefullness of *DAME* will increase. Then the mapping databases becomes critical and focal point for *DAME*