# TRANSFORMATION OF RELATIONAL SCHEMA INTO STATIC OBJECT SCHEMA

by

Kent Kutan

Project and Report submitted to the Faculty of the

Virginia Polytechnic Institute and State University

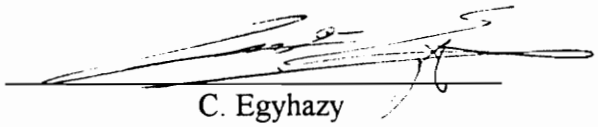
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

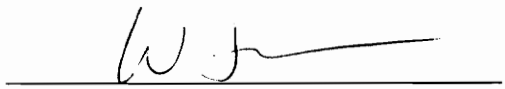
in

Computer Science

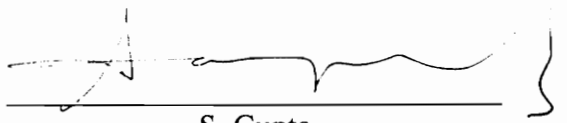
APPROVED:



C. Egyhazy



W. Frakes



S. Gupta

November, 1996

Falls Church, Virginia

Keywords: Database, Relational, Object, Schema, Model

# TRANSFORMATION OF RELATIONAL SCHEMA INTO STATIC OBJECT SCHEMA

by

Kent Kutan

Dr. Csaba Egyhazy, Chairman

Computer Science

(ABSTRACT)

The objective of this paper is to show how relational database schema can be transformed into static object-oriented database schema. First, data definition in the relational model and the object model are described. Next, the transformation rules are explained. This is followed by an illustration of an algorithm used to construct object-oriented schema out of relational schema. Finally, the algorithm is implemented through the use of C++.

## Acknowledgments

I would like to extend my thanks to my advisor, Professor Csaba Egyhazy, for giving me the opportunity to develop this paper and for his valuable reviews and suggestions. I would also like to thank my wife for her patience and encouragement during the writing of this paper.

## List of Figures

# 1 Introduction

The relational model represents data in a database as a collection of relations. A relation is a table of values. Each tuple (or row) in the table represents a collection of data values and corresponds to an instance of a table. Relationships between tables are represented by primary keys and foreign keys. In order to represent a one-to-many relationship between two tables, a foreign key needs to be embedded in a table. For many-to-many relationships, a separate table has to be created. This method of constructing relationships lacks a natural structure, preventing users from gaining meaningful insight into the application.

A good database schema should provide entities that more closely resemble their counterparts in the real world. The user should not have to struggle with computer-oriented concepts such as primary key and foreign key values. It is due to this limitation of the relational model that the user might want to transform a relational schema into an object-oriented schema. After the transformation is completed, the user can enhance the capabilities of the database by making full use of an object-oriented database technology (such as using inheritance and encapsulation) . It must be noted that this paper concentrates only on the static structure of a system, which includes attributes and relationships.

With an object-oriented schema, a database can be represented as classes and links rather than tables with primary keys and foreign keys. A class corresponds to a user-

defined data type with its own methods. With this approach, each class in the database can be more directly mapped to its real world object, making it easier for the user to build and maintain a database. One of the most important advantages of this data representation is that it provides data independence. In other words, it allows the internal representation of a class to be changed without requiring any of the applications that use the objects of that class to be rewritten. In a relational model, this type of change will not only require a modification of the application code, but also the alteration of all the tables that are related.

In addition to classes, an object-oriented schema provides links that show bi-directional associations between classes. These associations may be one-to-one, one-to-many or many-to-many relationships. This paper makes use of links only when there is a many-to-many association. For one-to-one and one-to-many associations, link attributes are folded into the class opposite the "one" side.

The difference between objects and relations is better illustrated in figure 1.1. The first part of the figure is a relational model diagram that we use as our starting point. The second half of the figure shows the same database represented by objects. There is no longer any need to use primary keys and foreign keys and all of the redundant fields are eliminated. The table "building" no longer contains the attribute "division_id", which is actually not part of a "building", but an attribute of a "division". There is also no need for a separate table just to show a many-to-many relationship. For example, the table "Employee_Proj" is needed to store a many-to-many relationship between tables

"Employee" and "Project". In the object model, this table does not exist since there is no need for it.

There have been many attempts to convert relational schema into a more object-oriented representation. Some of them concentrate on extending the relational model rather than transforming it into an object-oriented model. For example, the UNISQL/X data model developed by Sunit Gala and Won Kim [1] offers a nice tool to incorporate user-defined data types and inheritance within the framework of a relational data model. That is not the objective of this paper. This paper attempts to rebuild a relational database into an object-oriented database so that data representation is more modular and other object-oriented concepts such as methods can be applied to enhance the capabilities of the database. Premerlani and Blaha [2], Markowitz and Makowsky [3] have proposed systematic approaches to such transformations. A lot of the concepts introduced in Z. P. Yu's work [4] on transforming database schema has been used in this paper. Some commercial products that attempt to provide a relational-to-object mapping system include GemStone Systems [5] and Persistence Software [6].

The paper begins with descriptions of the relational and object data definitions. The object-modeling technique (OMT) notation [7] is used for modeling data. Then foreign key rules and the mapping process are introduced. Finally, the transformation algorithm is described and implemented in C++.

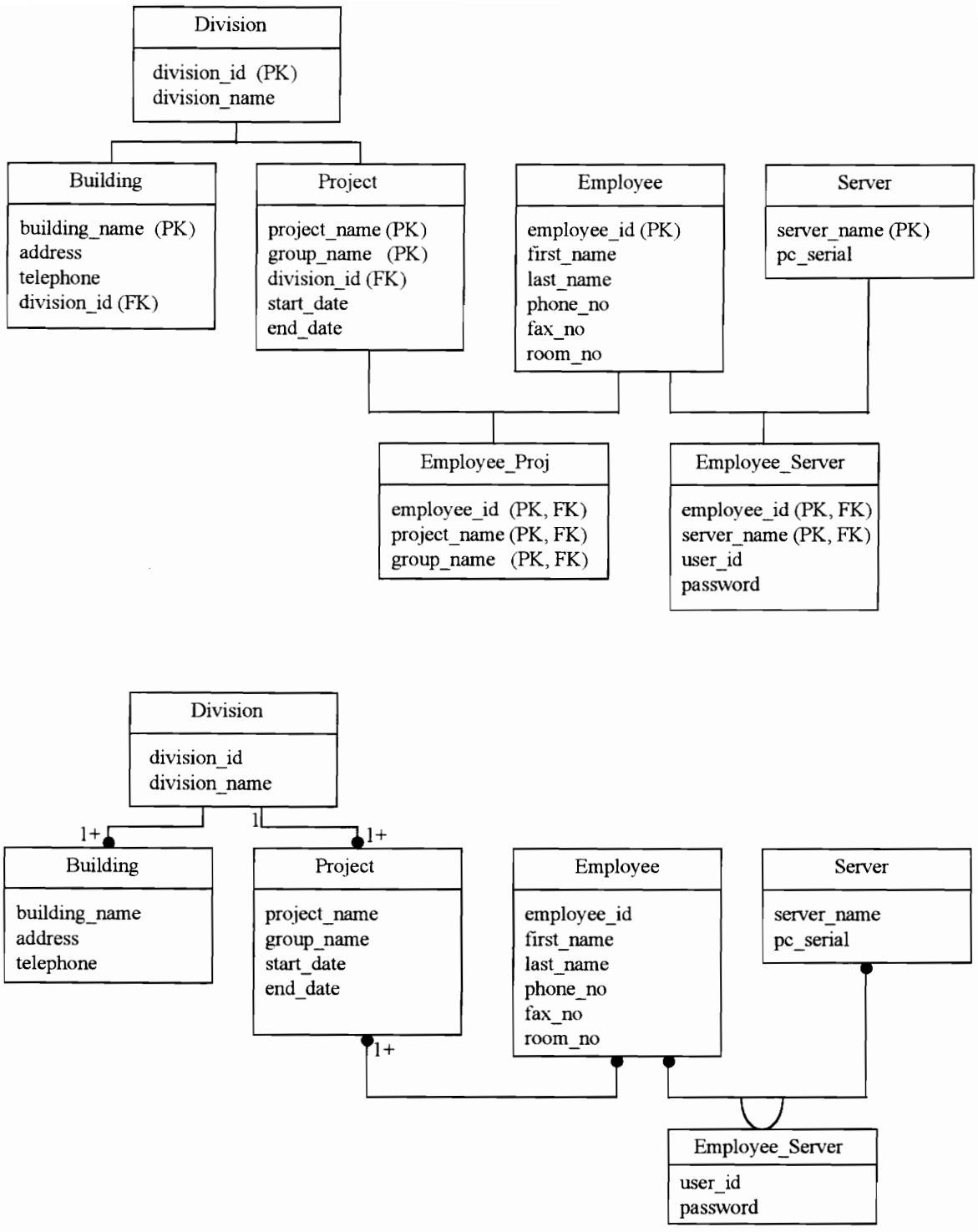



**Figure 1.1** The relational and static object database representations

# 2 Methodology

The transformation from a relational database model to an object-oriented database model will be illustrated in three steps. First, the data definition schema for each model will be introduced. The structured query language (SQL) will be used for defining data structures in a relational model. For object-oriented models, there is currently no standard data definition language. Therefore, a schema that defines the basic components of an object-oriented model will be used.

Second, the rules for mapping tables to classes and links will be described. Both many-to-many and one-to-many relationships will be covered. The foreign key rules of the relational model will provide us with the required information to determine the multiplicity and the membership type of the relationships between the tables. Finally, a detailed algorithm that explains all the stages of the transformation will be discussed.

## 2.1 Data definition

Before illustrating the transformation process, the data definition used for each model needs to be introduced since they reflect the format for the input and output files.

### 2.1.1 The relational data definition

Data in a relational model will be defined by using SQL data definition statements.

Figure 2.1 shows an example of a table definition using SQL. As you can see, the definition includes a CREATE TABLE statement for each table. Each CREATE TABLE statement specifies the name of the table to be created, the names and data types of its columns, whether the columns must have a value or not, the primary key, the foreign key(s) and the foreign key rules.

```
CREATE TABLE Employee (
        employee_id Integer NOT NULL,
        name char(30) NOT NULL,
        dept_name char(10) NOT NULL,
        address char (30) NULL,
        PRIMARY KEY(employee_id)
        FOREIGN KEY(dept_name) REFERENCES Department
                NOT NULL
                ON DELETE RESTRICT
)
```

**Figure 2.1** Relational data definition

Primary key values in a relational database represent entity identifiers whereas foreign key values represent references (unless they happen to be null). The foreign key rules at the end of the data definition statement provide important information about the nature of the relationships in the database. They answer the following two questions:

1. Can the foreign key accept nulls? In our example, can an employee exist in the database without being assigned to a department?

2. What should happen on an attempt to delete the target of a foreign key reference? In our case, what should happen if there is an attempt to delete a department

for which there exists at least one matching employee? In general, there are three possibilities:

a) RESTRICT - The delete operation can only take place if there are no employees assigned to that department that needs to be deleted.

b) CASCADE - Deleting the department cascades to the employees that work in that department, resulting in a delete operation of all employees referencing that department.

c) SET NULL - The foreign key is set to null in all the employee records containing that department and the department is then deleted. This option is only allowed if the foreign key can accept nulls.

### 2.1.2 The object data definition

The object-oriented database schema represents each entity as a class. A class by definition is an abstract data type that specifies both an object's structure and behavior. Our transformation results in a static object model that contains the representation of the class with no operations attached to it. In other words, it looks more like a user-defined data type than an abstract data type. Of course, the user can add operations and messages to make the model behave more object-oriented.

In our output, each class will be defined in the format shown in Figure 2.2.

```
CLASS Employee (
        ASSOCIATED TO ( Department ONE MANDATORY)
        ATTRIBUTES (
                employee_id Integer IDENTIFIER
                name Char(30) REQUIRED
                address Char(30) OPTIONAL)
)
```

**Figure 2.2 Object data definition**

The object data definition starts with the name of the class and lists other classes that it might be associated with. It specifies the multiplicity of the association as well as the membership type. In our example, the multiplicity is one and membership type is mandatory.

Many-to-many relationships between classes that require additional attributes to represent the relationship will be defined as links and will consist of the name of the link, the classes that are linked together, the membership type for each class and the attributes of the link.

Even though the definition of a class may look similar to the relational model in terms of the type of information it contains, it is a very different method of representing data. Each class is more directly mapped to its real-world counterpart. Any attribute that is part of an employee will exist as part of the class definition and will not have to be repeated in other classes that it has a relationship with. In the relational model, attributes have to be repeated as foreign keys in other tables, making the database a less natural representation and more difficult to understand and maintain.

## 2.2 The transformation rules

The relational model represents relationships in two ways depending on the multiplicity of the relationship. We will first study the transformation rules for one-to-one and one-to-many relationships that make use of embedded foreign keys. Next, many-to-many relationships through the use of distinct tables will be covered.

### 2.2.1 One-to-many relationships

One-to-many and one-to-one relationships are represented through primary key - foreign key pairings. The table opposite the "one" side will contain the foreign key. In our algorithm, the use of embedded foreign keys will be treated as one-to-many relationships. Let us assume that we have two tables named table A and table B. Let us say that these two tables have a one-to-many relationship with table A being on the "one" side and table B being on the "many" side as illustrated below (PK and FK stand for primary key and foreign key respectively):

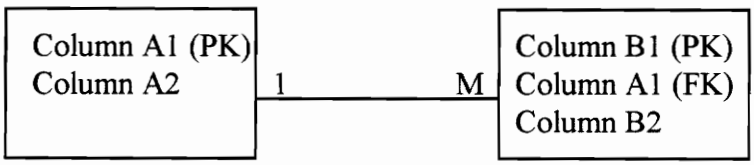
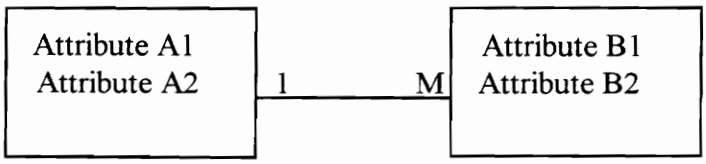



Table A Table B

In this case, the primary key of table A exists as an embedded foreign key in table B, where one record of table A can be related to many records of table B. In this case, each table will be mapped to a class containing attributes that only apply to the class itself. In

other words, table B will no longer contain the foreign key from table A. As a result, the classes will have the following structure.




Class A Class B

In order to determine membership types, the rules listed in figure 2.3 will be applied.

| **Foreign Key Rules** | **Membership type of Table A** | **Membership type of Table B** |
|---|---|---|
| NOT NULL<br>ON DELETE RESTRICT | Mandatory or Optional | Mandatory |
| NULL<br>ON DELETE SET NULL | Mandatory or Optional | Optional |
| NOT NULL<br>ON DELETE CASCADE | Mandatory or Optional | Mandatory (with existence dependency) |

**Figure 2.3 Membership type mapping rules for one-to-many relationships**

In the first case, the foreign key rule states that it can not be set to null. If a delete operation is about to be performed on a record in table A, it can only be deleted if there is no record in table B that references the primary key in that record of table A. Therefore, table B is known to have a mandatory relationship with table A. However, the membership type of table A can be either mandatory or optional. Table A can exist independently of

table B, in which case its membership type would be optional or the application might require a record in table A to be referenced by a record in table B. As it is seen, the membership type for table A cannot be determined by foreign key rules. For our transformation, we will assume table A to have a mandatory relationship.

### 2.2.2 Many-to-many relationships

The relational model represents many-to-many relationships as tables of their own. In other words, a separate table is created to store the many-to-many association between two tables. Let us say that we have two tables, table A and table B that have a many-to-many relationship. In this case, there will be a need to create a third table that includes two foreign keys corresponding to the two participants, and those foreign keys must reference the corresponding participant relations A and B. This can be illustrated as follows:

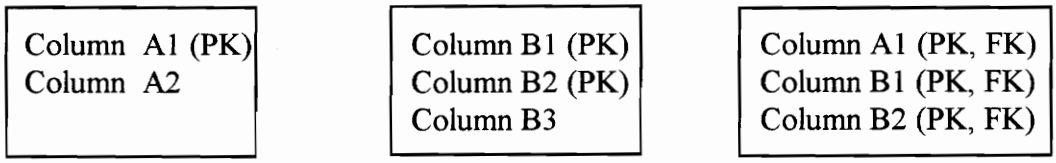



Table A Table B Table AB

Table AB, which has been created to represent the many-to-many relationship between table A and table B combines the primary keys of tables A and B and uses that combination as both its primary keys and foreign keys. In the object model, there is no need for a separate class for storing this type of relationship. There will only be two classes created for tables A and B with the schema specifying the multiplicity of the

relationship to be many-to-many. If table AB had additional columns not used in table A or B, it would become a link in the object model. The link would not contain any of the foreign keys and would consist of only the attribute that is not part of class A or class B.

The membership type for many-to-many relationships will be determined by the rules shown in figure 2.4.

| **Foreign Key Rules for Table AB** | | **Membership type of Table A** | **Membership type of Table B** |
|---|---|---|---|
| **Reference for Table A** | **Reference for Table B** | | |
| NOT NULL<br>ON DELETE RESTRICT | NOT NULL<br>ON DELETE RESTRICT | Mandatory | Mandatory |
| NOT NULL<br>ON DELETE CASCADE | NOT NULL<br>ON DELETE RESTRICT | Mandatory | Optional |
| NOT NULL<br>ON DELETE RESTRICT | NOT NULL<br>ON DELETE CASCADE | Optional | Mandatory |
| NOT NULL<br>ON DELETE CASCADE | NOT NULL<br>ON DELETE CASCADE | Optional | Optional |

**Figure 2.4 Membership type mapping rules for many-to-many relationships**

There are four cases considered for many-to-many relationships. The foreign key is assumed to be not null because it is also part of the primary key in table AB. If the deletion is restricted for both tables, this shows that there is a mandatory association on each side since neither one can be deleted if it is referred to in table AB.

The second case cascades delete operations on table A. In other words, if a record in table A is about to be deleted, its corresponding record in table AB will also be deleted. This means that a record in table B can still exist without having a record in table AB,

making the membership type of table B optional. However, if a record in table B needs to be deleted and that record has a reference in table AB, the deletion will not be allowed. This indicates that table A must keep its referenced record in table AB and table A has a mandatory relationship with table B. The third case is the exact opposite of the second case with table A having an optional and table B having a mandatory association.

Finally, the fourth case involves cascading the delete operations of a record in table A or table B to table AB. In other words, if a record in table A or table B is deleted, the records in table AB that refer to the record in table A or table B will also be deleted. This is a case where tables A and B have an optional many-to-many association with each other.

# 3 The Transformation Algorithm

The basic class used to perform the transformation (named "relational_to_object" ) contains the fields required to store both the relational and object schema for each table in the database. It has the following structure:

```
Class Relational_to_Object
{
public:
        char mapping_type [7];
        void retrieve();
        void find_association();
        void column_category();
        void set_class_attributes();
        void set_class_options();
        void output_class();
        void output_link();
        Relational_To_Object();
        ~Relational_To_Object();
private:
        char table_name[25];
        Column *column_ptr;
        char primary_key[100];
        Foreign_Key *foreign_key;
        Associated_Class *associated_class_ptr;
        int association;
        int class_attributes;
        int dependent;
        char class_dependent_on[25];
        void one_to_many(char table[25]);
        void many_to_many_link();
        void many_to_many_no_link(char table1[25], char table2[25]);
};
```

The class makes use of a structure named "Column" to represent columns in a table. Each instance of class Column stores the column name, the data type, the null rule

(whether the column can or cannot be null) and the category for the object model representation (required, optional or identifier). Following is the class definition:

```
Class Column
{
public:
        char column_name[25];
        char data_type[25];
        char null_rule[10];
        char category[11];
        Column *next;
        Column();
        ~Column();
};
```

The foreign keys also have a class of their own and store the foreign key name, the reference table, the null rule and the delete rule. The foreign key class is structured as follows:

```
Class Foreign_Key
{
public:
        char foreign_key_name[100];
        char reference[25];
        char null_rule[10];
        char delete_rule[10];
        Foreign_Key *next;
        Foreign_Key();
        ~Foreign_Key();
};
```

There is also a class named "Associated_Class" that is used to represent the associated classes in object schema. The Associated_Class has the following components:

```
Class Associated_Class
{
public:
        char class_name[25];
        char multiplicity[5];
        char membership[10];
        Associated_Class *next;
        Associated_Class();
        ~Associated_Class();
};
```

All these classes (except Relational_To_Object) serve as user-defined data types in the program and have pointers that point to their own class so that a linked list of these objects can be created. This allows their size to grow dynamically as they are needed.

## 3.1 Initialization

The algorithm assumes that the relational model will be represented using SQL, as explained in section 2.1.1 and will be available as an input file. The program creates an output file, representing the database using object schema, as described in section 2.1.

First, the input file is read and each table is loaded into an instance of the Relational_To_Object Class. All of these instances will be stored in an array of data type Relational_To_Object. The array size should be large enough to acccomodate all the tables in the database.

The public member function Relational_To_Object::retrieve() reads the table name, the column names, their datatypes, whether null is allowed or not, the primary key, the foreign key(s) and the foreign key rules.

## 3.2 Determining objects

Once all the relational tables are loaded into an array of type Relational_To_Object, the tables that have more than one foreign key, with all the foreign keys matching the primary key are identified (the "association" attribute is set to zero). For example, the "employee_proj" and "employee_server" tables in figure 1.1 are associations.

Next, the category for each column is determined based on its null rule and the table's primary key. If null is allowed, the column gets a category of "Optional". If null is not allowed, the category is "Required" and if the column happens to be a primary key, it is defined to be of category "Identifier".

The next step is to identify the class attributes. This is done by eliminating all the columns that make up the foreign keys in the table. If there are still any columns left, the remaining attributes will make up the class attributes. For example, table "building" in figure 1.1 will no longer contain its foreign key "division_id" when it is transformed into an object.

The determination of classes and links is done by checking whether the table has been marked as an association. If the table is not association, it will be defined as a class. If the table is an association and contains class attributes (after the foreign keys have been taken out), it becomes a link. For instance, table "employee_server" in figure 1.1 becomes a link after its foreign keys are eliminated. If the table is an association with no class attributes it does not need to be defined as an object. Figure 1.1 illustrates this by table "employee_proj", which is not represented in the object model.

The associated classes are determined by identifying all the tables referenced by the foreign keys in each table. The multiplicity of the relationships is defined to be "many" or "one". The tables referenced by an association and the tables containing an embedded foreign key are set to "many', while the others are set to "one". The null rule and delete rule are used to set the membership type to "Mandatory" or "Optional" by applying the transformation rules discussed in section 2.2. Finally, the resulting object schema is written to an output file.

Appendix A shows the input and output files corresponding to the database structure illustrated in figure 1.1. Appendix B contains the complete program listing written in C++.

# 4 Conclusion

To overcome the limitations of relational schema, a transformation algorithm has been developed to transform relational schema into object-oriented schema. The transformation makes use of foreign keys and foreign key rules to determine the multiplicity and membership type of relationships. This information is then used to map tables to objects that are structured closer to their real world representation. The algorithm has been coded in C++ and the results have been illustrated. There are additional transformations that can be added to the algorithm presented in this paper. For example, the concept of inheritance and subclasses can be included by searching for tables with similar primary keys. Some of these tables can be designated as subclasses of others.

## Appendix A

## Input and output files

```
CREATE TABLE Division(
  division_id Integer  NOT NULL,
  division_name Varchar(20) NOT NULL,
  PRIMARY KEY (division_id)
)

CREATE TABLE Building(
  building_name Varchar(20) NOT NULL,
  address Varchar(50) NULL,
  telephone Char(12) NULL,
  division_id Integer NULL,
  PRIMARY KEY (building_name)
  FOREIGN KEY (division_id) REFERENCES Division
    NULL
    ON DELETE SET NULL
)

CREATE TABLE Project(
  project_name Varchar(20) NOT NULL,
  group_name Varchar(20) NOT NULL,
  division_id Integer NOT NULL,
  start_date date NULL,
  end_date date NULL,
  PRIMARY KEY (project_name,group_name)
  FOREIGN KEY (division_id) REFERENCES Division
    NOT NULL
    ON DELETE CASCADE
)

CREATE TABLE Employee(
  employee_id Char(10) NOT NULL,
  first_name Varchar(15) NOT NULL,
  last_name Varchar(15) NOT NULL,
  phone_no Char(12) NULL,
  fax_no Char(12) NULL,
  room_no Char(4) NULL,
```

```
  PRIMARY KEY (employee_id)
)

CREATE TABLE Employee_Proj(
  employee_id Char(10) NOT NULL,
  project_name Varchar(20) NOT NULL,
  group_name Varchar(20) NOT NULL,
  PRIMARY KEY (employee_id,project_name,group_name)
  FOREIGN KEY (employee_id) REFERENCES Employee
    NOT NULL
    ON DELETE CASCADE
  FOREIGN KEY (project_name,group_name) REFERENCES Project
    NOT NULL
    ON DELETE RESTRICT
)

CREATE TABLE Server(
  server_name Char(10) NOT NULL,
  pc_serial Char(15) NOT NULL,
  PRIMARY KEY (server_name)
)

CREATE TABLE Employee_Server(
  employee_id Char(10) NOT NULL,
  server_name Char(10) NOT NULL,
  user_id Char(8) NOT NULL,
  password Char(8) NOT NULL,
  PRIMARY KEY (employee_id,server_name)
  FOREIGN KEY (employee_id) REFERENCES Employee
    NOT NULL
    ON DELETE CASCADE
  FOREIGN KEY (server_name) REFERENCES Server
    NOT NULL
    ON DELETE CASCADE
)
```

```
CLASS Division(
  ASSOCIATED TO (
    Building ONE MANDATORY
    Project ONE MANDATORY)
  ATTRIBUTES (
    division_id Integer IDENTIFIER
    division_name Varchar(20) REQUIRED)
)

CLASS Building(
  ASSOCIATED TO (
    Division MANY OPTIONAL)
  ATTRIBUTES (
    building_name Varchar(20) IDENTIFIER
    address Varchar(50) OPTIONAL
    telephone Char(12) OPTIONAL)
)

CLASS Project(
  ASSOCIATED TO (
    Division MANY MANDATORY
    Employee MANY OPTIONAL)
  DEPENDENT ON(Division)
  ATTRIBUTES (
    project_name Varchar(20) IDENTIFIER
    group_name Varchar(20) IDENTIFIER
    start_date date OPTIONAL
    end_date date OPTIONAL)
)

CLASS Employee(
  ASSOCIATED TO (
    Project MANY MANDATORY)
  ATTRIBUTES (
    employee_id Char(10) IDENTIFIER
    first_name Varchar(15) REQUIRED
    last_name Varchar(15) REQUIRED
    phone_no Char(12) OPTIONAL
    fax_no Char(12) OPTIONAL
    room_no Char(4) OPTIONAL)
```

```
)


CLASS Server(
  ASSOCIATED TO ()
  ATTRIBUTES (
    server_name Char(10) IDENTIFIER
    pc_serial Char(15) REQUIRED)
)

LINK Employee_Server(
  LINKED CLASSES(
    Employee MANY OPTIONAL
    Server MANY OPTIONAL)
  ATTRIBUTES (
    user_id Char(8) REQUIRED
    password Char(8) REQUIRED)
)
```

## Appendix B

## Program Listing for Schema Transformation

```
#include <fstream.h>
#include <iostream.h>
#include <string.h>

ifstream fin("relation.txt");
ofstream fout("object.txt");
int table_no = 0;

class Column
{
public:
        char column_name[25];
        char data_type[25];
        char null_rule[10];
        char category[11];
        Column *next;
        Column();
        ~Column();
};

Column::Column()
{
        strcpy(column_name, "NULL");
        strcpy(data_type, "NULL");
        strcpy(null_rule, "NULL");
        strcpy(category, "NULL");
        next = 0;
}

Column::~Column()
{
        if(next)
        {
                delete next;
```

```
                next = 0;
        }
}


class Foreign_Key
{
public:
        char foreign_key_name[100];
        char reference[25];
        char null_rule[10];
        char delete_rule[10];
        Foreign_Key *next;
        Foreign_Key();
        ~Foreign_Key();
};

Foreign_Key::Foreign_Key()
{
        strcpy(foreign_key_name, "NULL");
        strcpy(reference, "NULL");
        strcpy(null_rule, "NULL");
        strcpy(delete_rule, "NULL");
        next = 0;
}

Foreign_Key::~Foreign_Key()
{
        if(next)
        {
                delete next;
                next = 0;
        }
}

class Associated_Class
{
public:
        char class_name[25];
        char multiplicity[5];
        char membership[10];
        Associated_Class *next;
        Associated_Class();
```

```
        ~Associated_Class();
};

Associated_Class::Associated_Class()
{
        strcpy(class_name, "NULL");
        strcpy(multiplicity, "NULL");
        strcpy(membership, "NULL");
        next = 0;
}

Associated_Class::~Associated_Class()
{
        if(next)
        {
                delete next;
                next = 0;
        }
}

class Relational_To_Object
{
public:
        char mapping_type[7];
        void retrieve();   //Retrieves the relational input file.
        void find_association(); //Finds tables that reflect associations.
        void column_category();
        void set_class_attributes();
        void set_class_options();
        void output_class();
        void output_link();
        Relational_To_Object();
        ~Relational_To_Object();
private:
        char table_name[25];
        Column *column_ptr;
        char primary_key[100];
        Foreign_Key  *foreign_key_ptr;
        Associated_Class *associated_class_ptr;
        int association;
        int class_attributes;
        int dependent;
        char class_dependent_on[25];
```

```
        void one_to_many(char table[25]);
        void many_to_many_link();
        void many_to_many_no_link(char table1[25], char table2[25]);
};

Relational_To_Object::Relational_To_Object()
{
        strcpy(table_name, "NULL");
        column_ptr = 0;
        strcpy(primary_key, "NULL");
        foreign_key_ptr = 0;
        associated_class_ptr = 0;
        association = 0;
        class_attributes = 0;
        strcpy(mapping_type, "NULL");
        dependent = 0;
        strcpy(class_dependent_on, "NULL");
}

Relational_To_Object::~Relational_To_Object()
{
        if(column_ptr)
        {
                delete column_ptr;
                column_ptr = 0;
        }

        if(foreign_key_ptr)
        {
                delete foreign_key_ptr;
                foreign_key_ptr = 0;
        }

        if(associated_class_ptr)
        {
                delete associated_class_ptr;
                associated_class_ptr = 0;
        }
}

void Relational_To_Object::retrieve()
{
        char buf[25];
```

```
Column *new_column;
Column *last_column = 0;
Foreign_Key *new_foreign_key;
Foreign_Key *last_foreign_key = 0;
int table_processed = 0;

fin >> buf;
while (!fin.eof())
{
        switch (buf[0])
        {
        case ')':
                return;

        case 'C':
                //If the table is not loaded yet, continue the loop.
                if (table_processed == 1) return;
                table_processed = 1;
                fin >> buf; //skip "TABLE"
                fin.get(); //Skip space ' '.

                //Get table name which ends with the character '('.
                //Table name is assumed to be 25 characters or less.
                fin.get(buf, 25, '('); //Get Table Name
                strcpy(table_name, buf);
                fin.get(); //Skip '('
                fin >> buf;
                break;

        case 'P':
                fin.get(buf, 25, '('); //SKIP spaces and "KEY"
                fin.get();                                      //Skip '('
                fin.get(buf, 100, 41); //Read until ')'
                strcpy(primary_key, buf);
                fin.get(); //Skip ')'
                last_foreign_key = 0;
                fin >> buf;
                break;

        case 'F':
                new_foreign_key = new Foreign_Key;

                //Get foreign key name.
```

```
            fin.get(buf, 25, '('); //SKIP spaces and "KEY"
            fin.get();          //Skip '('
            fin.get(buf, 100, ')');
            strcpy(new_foreign_key->foreign_key_name, buf);

            //Get reference table name.
            fin.get();  //skip ")"
            fin >> buf;  //skip "REFERENCES"
            fin >> buf;
            strcpy(new_foreign_key->reference, buf);

            //Get null rule.
            fin >> buf;
            if (buf[0] == 'N' && buf[1] == 'O')
            {
                    strcpy(new_foreign_key->null_rule, "NOT NULL");
                    fin >> buf; //skip "NULL
            }
            else
                    strcpy(new_foreign_key->null_rule, buf);

            //Get delete rule.
            fin >> buf;  //skip "ON"
            fin >> buf; //skip "DELETE"
            fin >> buf;
            strcpy(new_foreign_key->delete_rule, buf);

            new_foreign_key->next = 0;
            if (last_foreign_key)
                    last_foreign_key->next = new_foreign_key;
            else
                    foreign_key_ptr = new_foreign_key;

            last_foreign_key = new_foreign_key;
            fin >> buf;
            break;

    default:
            new_column = new Column;

            strcpy(new_column->column_name, buf); //Copy column name.

            fin >> buf;         //Get data type.
```

```
                    strcpy(new_column->data_type, buf);

                    fin >> buf;                         //Get null rule.
                    if (buf[0] == 'N' && buf[1] == 'O')
                    {
                        strcpy(new_column->null_rule, "NOT NULL");
                        fin >> buf; //Skip "NULL".
                    }
                    else
                        strcpy(new_column->null_rule, "NULL");

                    new_column->next = 0;

                    if (last_column)
                        last_column->next = new_column;
                    else
                        column_ptr = new_column;
                    last_column = new_column;
                    fin >> buf;
                }
            }
        }

void Relational_To_Object::find_association()
{
        Foreign_Key *foreign_key_point;
        char foreign_keys[100];

        if (foreign_key_ptr == 0)
                association = 0;
        else if (foreign_key_ptr->next == 0)
                association = 0;
        else
        {
                strcpy(foreign_keys, foreign_key_ptr->foreign_key_name);
                strcat(foreign_keys, ",");
                foreign_key_point = foreign_key_ptr->next;
                while(foreign_key_point)
                {
                        strcat(foreign_keys, foreign_key_point->foreign_key_name);
                        foreign_key_point = foreign_key_point->next;
                }
                if(strcmp(foreign_keys, primary_key) == 0)
```

```
                        association = 1;
                else
                        association = 0;
        }
}

void Relational_To_Object::column_category()
{
 Column *column_point;
 int i;

 //Loop through all the columns.
 column_point = column_ptr;
 while(column_point)
 {
                if (strcmp(column_point->null_rule, "NULL") == 0)
                        strcpy(column_point->category, "OPTIONAL");
                else
                        strcpy(column_point->category, "REQUIRED");
                column_point = column_point->next;
 }

 //Set primary key columns to "IDENTIFIER".
 column_point = column_ptr;
 strcpy(column_point->category, "IDENTIFIER");
 column_point = column_point->next;
 i = 0;
 while(primary_key[i] != 0)
 {
        if(primary_key[i] == ',')
        {
                strcpy(column_point->category, "IDENTIFIER");
                column_point = column_point->next;
        }
        i++;
 }
}

void Relational_To_Object::set_class_attributes()
{
        Column *column_point;
        Column *previous_column;
        Column *next_column;
```

```
        Foreign_Key *foreign_key_point;
        char foreign_key[100];
        char field_name[25];
        int column_deleted;
        int i;
        int b;
        int end_loop;
        char buf[25];

        foreign_key_point = foreign_key_ptr;
        //Loop through all columns until the foreign key column is found.
        while(foreign_key_point)
        {
                strcpy(foreign_key, foreign_key_point->foreign_key_name);
                b = 0;
                i = 0;
                end_loop = 0;
                while(!end_loop)
                {
                        if(foreign_key[i] == ',' || foreign_key[i] == 0)
                        {
                                buf[b] = 0;

                                column_point = column_ptr;
                                previous_column = 0;
                                column_deleted = 0;
                                //Loop through all columns until foreign key column is
found.
                                while(!(column_deleted))
                                {
                                        strcpy(field_name, column_point->column_name);
                                        if(strcmp(buf, field_name) == 0)
                                        {
                                                next_column = column_point->next;
                                                if(previous_column)
                                                        previous_column->next =
next_column;
                                                else
                                                        column_ptr = next_column;

                                                column_point->next = 0;
                                                column_deleted = 1;
                                        }
```

```
                                else
                                {
                                        previous_column = column_point;
                                        column_point = column_point->next;
                                }
                        }
                        if(foreign_key[i] == 0)
                                end_loop = 1;
                        b = 0;
                        i++;
                }
                else
                {
                        buf[b] = foreign_key[i];
                        b++;
                        i++;
                }
        }

        foreign_key_point = foreign_key_point->next;
    }

    if (column_ptr == 0)
        class_attributes = 0;
    else
        class_attributes = 1;
}

void Relational_To_Object::set_class_options()
{
    char table[25];
    char table1[25];
    char table2[25];
    Foreign_Key *foreign_key_point;

    if(association == 0)
    {
        strcpy(mapping_type, "CLASS");
        foreign_key_point = foreign_key_ptr;
        while(foreign_key_point)
        {
            strcpy(table, foreign_key_point->reference);
            one_to_many(table);
```

```
                        foreign_key_point = foreign_key_point->next;
                }
        }
        else if(association == 1)
        {
                if(class_attributes == 1)
                {
                        strcpy(mapping_type, "LINK");
                        many_to_many_link();
                }
                else
                {
                        strcpy(mapping_type, "NONE");
                        foreign_key_point = foreign_key_ptr;
                        strcpy(table1, foreign_key_point->reference);
                        foreign_key_point = foreign_key_point->next;
                        strcpy(table2, foreign_key_point->reference);
                        many_to_many_no_link(table1, table2);
                }
        }
}

Relational_To_Object relational_to_object[25];

void Relational_To_Object::one_to_many(char table[25])
{
        int ref;
        int end_of_linked_list;
        Associated_Class *assoc_class_point;
        Associated_Class *ref_assoc_class_point;

        //Loop until the end of the associated class linked list for the
        //table with the embedded foreign key is reached.
        if(associated_class_ptr)
        {
                assoc_class_point = associated_class_ptr;
                end_of_linked_list = 0;
                while(!end_of_linked_list)
                {
                        if(assoc_class_point->next == 0)
                                end_of_linked_list = 1;
                        else
                                assoc_class_point = assoc_class_point->next;
```

```
        }
        assoc_class_point->next = new Associated_Class;
        assoc_class_point = assoc_class_point->next;
}
else
{
        associated_class_ptr = new Associated_Class;
        assoc_class_point = associated_class_ptr;
}


//Loop through all tables to find the referenced table.
for (ref = 0; ref < table_no; ref++)
{
        if(strcmp(relational_to_object[ref].table_name, table) == 0)
                break;
}

//Loop until the end of the associated class linked list for
//the referenced table is reached.
if(relational_to_object[ref].associated_class_ptr)
{
        ref_assoc_class_point =
                relational_to_object[ref].associated_class_ptr;
        end_of_linked_list = 0;
        while(!end_of_linked_list)
        {
                if(ref_assoc_class_point->next == 0)
                        end_of_linked_list = 1;
                else
                        ref_assoc_class_point = ref_assoc_class_point->next;
        }
        ref_assoc_class_point->next = new Associated_Class;
        ref_assoc_class_point = ref_assoc_class_point->next;
}
else
{
        relational_to_object[ref].associated_class_ptr =
                new Associated_Class;
        ref_assoc_class_point =
                relational_to_object[ref].associated_class_ptr;
}
```

```
        //Set the class_name of the associated classes.
        strcpy(assoc_class_point->class_name, table);
        strcpy(ref_assoc_class_point->class_name, table_name);

        //Set class options by applying the one-to-many relationship
        //transformation rules.
        strcpy(assoc_class_point->multiplicity, "MANY");
        strcpy(ref_assoc_class_point->multiplicity, "ONE");

        if ((strcmp(foreign_key_ptr->null_rule, "NOT NULL") == 0) &&
                (strcmp(foreign_key_ptr->delete_rule, "RESTRICT") == 0))
        {
                strcpy(assoc_class_point->membership, "MANDATORY");
                strcpy(ref_assoc_class_point->membership, "MANDATORY");
        }

        else if((strcmp(foreign_key_ptr->null_rule, "NULL") == 0) &&
                        (strcmp(foreign_key_ptr->delete_rule, "SET NULL") == 0))
        {
                strcpy(assoc_class_point->membership, "OPTIONAL");
                strcpy(ref_assoc_class_point->membership, "MANDATORY");
        }
        else if((strcmp(foreign_key_ptr->null_rule, "NOT NULL") == 0) &&
                        (strcmp(foreign_key_ptr->delete_rule, "CASCADE") == 0))
        {
                strcpy(assoc_class_point->membership, "MANDATORY");
                strcpy(ref_assoc_class_point->membership, "MANDATORY");
                dependent = 1;
                strcpy(class_dependent_on, table);
        }
}

void Relational_To_Object::many_to_many_link()
{
        char null_rule1[10];
        char null_rule2[10];
        char delete_rule1[10];
        char delete_rule2[10];
        Foreign_Key *foreign_key_point;
        Associated_Class *assoc_class_point1;
        Associated_Class *assoc_class_point2;

        associated_class_ptr = new Associated_Class;
```

```
assoc_class_point1 = associated_class_ptr;

assoc_class_point2 = new Associated_Class;
assoc_class_point1->next = assoc_class_point2;

foreign_key_point = foreign_key_ptr;

//Set the class names of the associated class.
//Also get null and delete rules.
strcpy(assoc_class_point1->class_name,
                foreign_key_point->reference);
strcpy(null_rule1, foreign_key_point->null_rule);
strcpy(delete_rule1, foreign_key_point->delete_rule);

foreign_key_point = foreign_key_point->next;
strcpy(assoc_class_point2->class_name,
                foreign_key_point->reference);
strcpy(null_rule2, foreign_key_point->null_rule);
strcpy(delete_rule2, foreign_key_point->delete_rule);

//Set multiplicity.
strcpy(assoc_class_point1->multiplicity, "MANY");
strcpy(assoc_class_point2->multiplicity, "MANY");

//Set membership.
if ((strcmp(null_rule1, "NOT NULL") == 0) &&
        (strcmp(delete_rule1, "RESTRICT") == 0) &&
        (strcmp(null_rule2, "NOT NULL") == 0) &&
        (strcmp(delete_rule2, "RESTRICT") == 0))
{
        strcpy(assoc_class_point1->membership, "MANDATORY");
        strcpy(assoc_class_point2->membership, "MANDATORY");
}

else if ((strcmp(null_rule1, "NOT NULL") == 0) &&
                        (strcmp(delete_rule1, "CASCADE") == 0) &&
                        (strcmp(null_rule2, "NOT NULL") == 0) &&
                        (strcmp(delete_rule2, "RESTRICT") == 0))
{
        strcpy(assoc_class_point1->membership, "MANDATORY");
        strcpy(assoc_class_point2->membership, "OPTIONAL");
}
```

```
        else if ((strcmp(null_rule1, "NOT NULL") == 0) &&
                                (strcmp(delete_rule1, "RESTRICT") == 0) &&
                                (strcmp(null_rule2, "NOT NULL") == 0) &&
                                (strcmp(delete_rule2, "CASCADE") == 0))
        {
                strcpy(assoc_class_point1->membership, "OPTIONAL");
                strcpy(assoc_class_point2->membership, "MANDATORY");
        }

        else if ((strcmp(null_rule1, "NOT NULL") == 0) &&
                                (strcmp(delete_rule1, "CASCADE") == 0) &&
                                (strcmp(null_rule2, "NOT NULL") == 0) &&
                                (strcmp(delete_rule2, "CASCADE") == 0))
        {
                strcpy(assoc_class_point1->membership, "OPTIONAL");
                strcpy(assoc_class_point2->membership, "OPTIONAL");
        }
}
void Relational_To_Object::many_to_many_no_link(char table1[25],

                                char table2[25])
{
        int ref1;
        int ref2;
        int end_of_linked_list;
        char null_rule1[10];
        char null_rule2[10];
        char delete_rule1[10];
        char delete_rule2[10];
        Associated_Class *ref1_assoc_class_point;
        Associated_Class *ref2_assoc_class_point;
        Foreign_Key *foreign_key_point;

        //Loop through all tables to find the first referenced table.
        for(ref1 = 0; ref1 < table_no; ref1++)
        {
                if(strcmp(relational_to_object[ref1].table_name, table1) == 0)
                        break;
        }

        //Loop until the end of the associated classs linked list for the
        //first referenced table.
        if(relational_to_object[ref1].associated_class_ptr)
```

```
{
        ref1_assoc_class_point =
                relational_to_object[ref1].associated_class_ptr;
        end_of_linked_list = 0;
        while(!end_of_linked_list)
        {
                if(ref1_assoc_class_point->next == 0)
                        end_of_linked_list = 1;
                else
                        ref1_assoc_class_point = ref1_assoc_class_point->next;
        }
        ref1_assoc_class_point->next = new Associated_Class;
        ref1_assoc_class_point = ref1_assoc_class_point->next;
}
else
{
        relational_to_object[ref1].associated_class_ptr =
                new Associated_Class;
        ref1_assoc_class_point =
                relational_to_object[ref1].associated_class_ptr;
}

//Loop through all the tables to find the second referenced table.
for (ref2 = 0; ref2 < table_no; ref2++)
{
        if(strcmp(relational_to_object[ref2].table_name, table2) == 0)
                break;
}

//Loop until the end of the associated class linked list for
//the second table is reached.
if(relational_to_object[ref2].associated_class_ptr)
{
        ref2_assoc_class_point =
                relational_to_object[ref2].associated_class_ptr;
        end_of_linked_list = 0;
        while(!end_of_linked_list)
        {
                if(ref2_assoc_class_point->next == 0)
                        end_of_linked_list = 1;
                else
                        ref2_assoc_class_point = ref2_assoc_class_point->next;
        }
```

```
        ref2_assoc_class_point->next = new Associated_Class;
        ref2_assoc_class_point = ref2_assoc_class_point->next;
}
else
{
        relational_to_object[ref2].associated_class_ptr =
                new Associated_Class;
        ref1_assoc_class_point =
                relational_to_object[ref2].associated_class_ptr;
}

//Set the class name of the associated classes.
strcpy(ref1_assoc_class_point->class_name, table2);
strcpy(ref2_assoc_class_point->class_name, table1);

//Set multiplicity.
strcpy(ref1_assoc_class_point->multiplicity, "MANY");
strcpy(ref2_assoc_class_point->multiplicity, "MANY");

//Get the null rule and the delete rule for both tables.
foreign_key_point = foreign_key_ptr;
strcpy(null_rule1, foreign_key_point->null_rule);
strcpy(delete_rule1, foreign_key_point->delete_rule);

foreign_key_point = foreign_key_point->next;
strcpy(null_rule2, foreign_key_point->null_rule);
strcpy(delete_rule2, foreign_key_point->delete_rule);

if ((strcmp(null_rule1, "NOT NULL") == 0) &&
        (strcmp(delete_rule1, "RESTRICT") == 0) &&
        (strcmp(null_rule2, "NOT NULL") == 0) &&
        (strcmp(delete_rule2, "RESTRICT") == 0))
{
        strcpy(ref1_assoc_class_point->membership, "MANDATORY");
        strcpy(ref2_assoc_class_point->membership, "MANDATORY");
}

else if ((strcmp(null_rule1, "NOT NULL") == 0) &&
                        (strcmp(delete_rule1, "CASCADE") == 0) &&
                        (strcmp(null_rule2, "NOT NULL") == 0) &&
                        (strcmp(delete_rule2, "RESTRICT") == 0))
{
        strcpy(ref1_assoc_class_point->membership, "MANDATORY");
```

```
                strcpy(ref2_assoc_class_point->membership, "OPTIONAL");
        }

        else if ((strcmp(null_rule1, "NOT NULL") == 0) &&
                                (strcmp(delete_rule1, "RESTRICT") == 0) &&
                                (strcmp(null_rule2, "NOT NULL") == 0) &&
                                (strcmp(delete_rule2, "CASCADE") == 0))
        {
                strcpy(ref1_assoc_class_point->membership, "OPTIONAL");
                strcpy(ref2_assoc_class_point->membership, "MANDATORY");
        }

        else if ((strcmp(null_rule1, "NOT NULL") == 0) &&
                                (strcmp(delete_rule1, "CASCADE") == 0) &&
                                (strcmp(null_rule2, "NOT NULL") == 0) &&
                                (strcmp(delete_rule2, "CASCADE") == 0))
        {
                strcpy(ref1_assoc_class_point->membership, "OPTIONAL");
                strcpy(ref2_assoc_class_point->membership, "OPTIONAL");
        }
}

void Relational_To_Object::output_class()
{
        Associated_Class *assoc_class_point;
        Column *column_point;

        fout << "CLASS ";
        fout << table_name << "(" << "\n";
        assoc_class_point = associated_class_ptr;
        fout << "   ASSOCIATED TO (";
        while(assoc_class_point)
        {
                fout << "\n";
                fout << "      " << assoc_class_point->class_name << " ";
                fout << assoc_class_point->multiplicity << " ";
                fout << assoc_class_point->membership;
                assoc_class_point = assoc_class_point->next;
        }
        fout << ")" << "\n";
        if(dependent)
        {
                fout << "   DEPENDENT ON(" << class_dependent_on << ")";
```

```
                fout << "\n";
        }

        column_point = column_ptr;
        fout << "   ATTRIBUTES (";
        while(column_point)
        {
                fout << "\n";
                fout << "      " << column_point->column_name << " ";
                fout << column_point->data_type << " ";
                fout << column_point->category;
                column_point = column_point->next;
        }
        fout << ")" << "\n"; //end of Attributes
        fout << ")" << "\n"; //end of Class
        fout << "\n";
}


void Relational_To_Object::output_link()
{
        Associated_Class *assoc_class_point;
        Column *column_point;

        fout << "LINK ";
        fout << table_name << "(" << "\n";
        assoc_class_point = associated_class_ptr;
        fout << "   LINKED CLASSES(";
        while(assoc_class_point)
        {
                fout << "\n";
                fout << "      " << assoc_class_point->class_name << " ";
                fout << assoc_class_point->multiplicity << " ";
                fout << assoc_class_point->membership;
                assoc_class_point = assoc_class_point->next;
        }
        fout << ")" << "\n";

        column_point = column_ptr;
        fout << "   ATTRIBUTES (";
        while(column_point)
        {
                fout << "\n";
```

```
            fout << "      " << column_point->column_name << " ";
            fout << column_point->data_type << " ";
            fout << column_point->category;
            column_point = column_point->next;
        }
        fout << ")" << "\n"; //end of Attributes
        fout << ")" << "\n"; //end of Link
        fout << "\n";
}
```

```
void main()

        //Start importing relational tables into an array of class
        //Relation_To_Object.
        //Open the file relation.txt

        while (!fin.eof())
        {
                relational_to_object[table_no].retrieve();
                table_no ++;
        }

        for (int i = 0; i < table_no; i++)
        {
                relational_to_object[i].find_association();
                relational_to_object[i].column_category();
        }

        for (i = 0; i < table_no; i++)
                relational_to_object[i].set_class_attributes();

        for (i = 0; i < table_no; i++)
                relational_to_object[i].set_class_options();

        for (i = 0; i < table_no; i++)
        {
                if(strcmp(relational_to_object[i].mapping_type, "CLASS") == 0)
                        relational_to_object[i].output_class();
                else if(strcmp(relational_to_object[i].mapping_type, "LINK") == 0)
                        relational_to_object[i].output_link();
        }
}
```

## References

[1] S. Gala and W. Kim, *Database Design Methodology: Converting a Relational Schema into an Object-Relational Schema*, DB Design Methodology, 1994.

[2] W. Premerlani, M. Blaha, *An Approach for Reverse Engineering of Relational Databases,* Communications of the ACM, Volume 37 Number 5, May 1994.

[3] V. M. Markowitz, J.A. Makowsky, *Identifying Extended Entity-Relationship Object Structures In Relational Schemas,* IEEE Trans. Software Eng., 16 , 8 August 1990.

[4] Z. P. Yu, *Transformations Between Object-Oriented Model and Relational Model,* Technical Report Virginia Tech, 1994.

[5] F. Hayes, *Software Mapping Layers Ease Object-Relational Translations,* Computerworld, Volume 30 Number 24, June 10, 1996.

[6] F. Hayes, *Persistence Software Slims Object-to-Relational Translator,* Computerworld, Volume 29 Number 47, November 20, 1995.

[7] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.