# TRANSIMS Database Subsystem for IOC-1

*B. W. Bush*
Energy and Environmental Analysis Group
Los Alamos National Laboratory

19 March 1997

**Abstract**

The TRANSIMS database subsystem provides low-level services for accessing and modifying TRANSIMS data. It forms a layer separating the other subsystems from the actual data files so that the other subsystems do not need to access the data files at the physical level or deal with the physical location and organization of the files. This subsystem also organizes the data and supports a variety of metadata. It uses a relational model for the storage of data.

I. Introduction ..... 3
II. Design ..... 5
A. Concepts ..... 5
1. Data Directory ..... 5
2. Data Source ..... 5
3. Data Table ..... 5
B. Classes ..... 5
1. TDbDirectory ..... 6
2. TDbDirectoryDescription ..... 7
3. TDbUserInformation ..... 9
4. TDbSource ..... 9
5. TDbSourceDescription ..... 11
6. TDbTable ..... 11
7. TDbTableDescription ..... 12
8. TDbAccessor ..... 13
9. TDbSortedAccessor ..... 14
10. TDbInserter ..... 15
11. TDbField ..... 16
12. TDbException ..... 17
III. Implementation ..... 18
A. C++ Libraries ..... 18
B. Relational Database ..... 18
IV. Usage ..... 19
A. Examples ..... 19

1. Retrieving Data ..... 19
2. Inserting Data ..... 20
B. Notes ..... 21
1. Direct Data Manipulation ..... 21
2. Object Interdependencies ..... 21
3. Description Classes ..... 21
V. Future Work ..... 21
VI. References ..... 22
VII. APPENDIX: Booch Notation Diagrams ..... 22
VIII. APPENDIX: Database Creation ..... 24
A. Oracle 7 Parameters ..... 24
1. inittransims.ora ..... 24
2. configtransims.ora ..... 25
3. tnsnames.ora ..... 25
4. listener.ora ..... 26
5. sqlnet.ora ..... 26
B. SQL Scripts for Database Creation ..... 26
1. MakeDatabase1.sql ..... 27
2. MakeDatabase2.sql ..... 27
3. Database.sql ..... 28
4. MakeRole.sql ..... 29
5. MakeUser.sql ..... 29
IX. APPENDIX: Source Code ..... 30
A. TDbAccessor Class ..... 30
1. Accessor.h ..... 30
2. Accessor.C ..... 31
B. TDbDirectory Class ..... 34
1. Directory.h ..... 34
2. Directory.C ..... 36
C. TDbDirectoryDescription Class ..... 38
1. DirectoryDescription.h ..... 38
2. DirectoryDescription.C ..... 40
D. TDbException Class ..... 41
1. Exception.h ..... 41
2. Exception.C ..... 43
E. TDbField Class ..... 44
1. Field.h ..... 44
2. Field.C ..... 45
F. TDbInserter Class ..... 46
1. Inserter.h ..... 46
2. Inserter.C ..... 47
G. TDbSortedAccessor Class ..... 50
1. SortedAccessor.h ..... 50
2. SortedAccessor.C ..... 51
H. TDbSource Class ..... 52

1. Source.h ..... 52
2. Source.C ..... 53
I. TDbSourceDescription Class ..... 57
1. SourceDescription.h ..... 57
2. SourceDescription.C ..... 58
J. TDbTable Class ..... 58
1. Table.h ..... 58
2. Table.C ..... 60
K. TDbTableDescription Class ..... 63
1. TableDescription.h ..... 63
2. TableDescription.C ..... 64
L. TDbUserInformation Class ..... 65
1. UserInformation.h ..... 65
2. UserInformation.C ..... 66
X. APPENDIX: Test Program ..... 67
A. Test.C ..... 67
B. TestAccessor.C ..... 68
C. TestCleanup.C ..... 72
D. TestDirectory.C ..... 72
E. TestDirectoryDescription.C ..... 75
F. TestField.C ..... 76
G. TestInserter.C ..... 77
H. TestSortedAccessor.C ..... 79
I. TestSource.C ..... 81
J. TestSourceDescription.C ..... 83
K. TestTable.C ..... 84
L. TestTableDescription.C ..... 86
M. TestUserInformation.C ..... 87
N. TestCleanup.sql ..... 88
XI. APPENDIX: Import Utility ..... 89
A. Import.C ..... 89

# I. Introduction

The TRANSIMS database subsystem provides low-level services for accessing and modifying TRANSIMS data. It forms a layer separating the other subsystems from the actual data files so that the other subsystems do not need to access the data files at the physical level or deal with the physical location and organization of the files. Figure 1 shows the position of the database subsystem within the TRANSIMS software architecture.

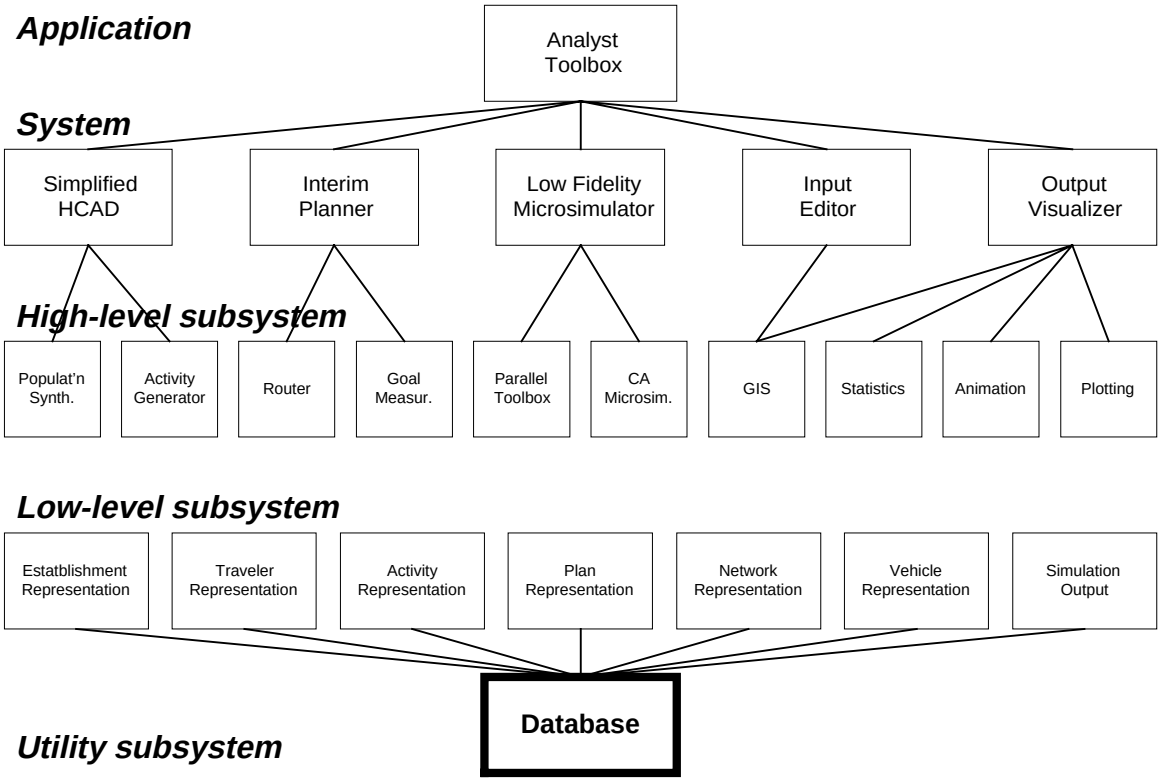


**Figure 1. Location of the database subsystem in the TRANSIMS software architecture.**

This subsystem organizes the data and supports a variety of metadata. A *data directory* contains the complete data for a project. Within the directory are *data sources* used to organize the actual data, which is stored in *data tables*. It is possible to retrieve descriptions of data directories, sources, or tables along with their interrelationships.

The database subsystem uses a relational model for the storage of data, so the data tables are structured in terms of fields and records. A variety of field types are available and the indexing of tables is also supported.

The implementation of the subsystem relies on a commercial C++ access library (DBtools.h++) to interface with a commercial relational database (Oracle 7). The details of the access library and the relational database are hidden from users, so that it will be possible in the future to change the underlying access library and/or the relational database without affecting the public interfaces exposed by the subsystem. A test program provides a means to verify correct database subsystem operation.

The body of this document outlines the design, implementation, and usage of the subsystem. The appendices contain the complete C++ source code, SQL scripts, and Oracle parameters for the subsystem.

# II. Design

## *A. Concepts*

### 1. Data Directory

All of the data associated with a particular project resides in a *data directory*. The directory merely serves to organize the data within it.

### 2. Data Source

Each data directory contains *data sources*. A source is not data—it is metadata (a description, kind, type, or class of data). Examples might be Traveler, Vehicle, Vehicle Type, Engine Type, Network Node, Network Link, etc. Sources are version-independent abstractions.

### 3. Data Table

The actual data for a source is stored in *data tables*. Each source may have several tables representing different versions of the data or different scenarios (case studies). A Network Node data source, for example, might have one table for the base case and another for the case where a new freeway has been constructed. The Traveler data source might have tables for the populations in different years. The data resides in relational tables consisting of rows (i.e., records or tuples) and columns (i.e., fields or attributes).

Each table "knows" on what other tables it depends. A Traveler table, for instance, knows with which table in the Establishment data source it is consistent. An Establishment table likewise knows with which version (table) of the Network Node data source it is associated.

## *B. Classes*

The database subsystem has classes for locating, describing, and manipulating data and metadata. Figure 2 shows the relationships between classes.

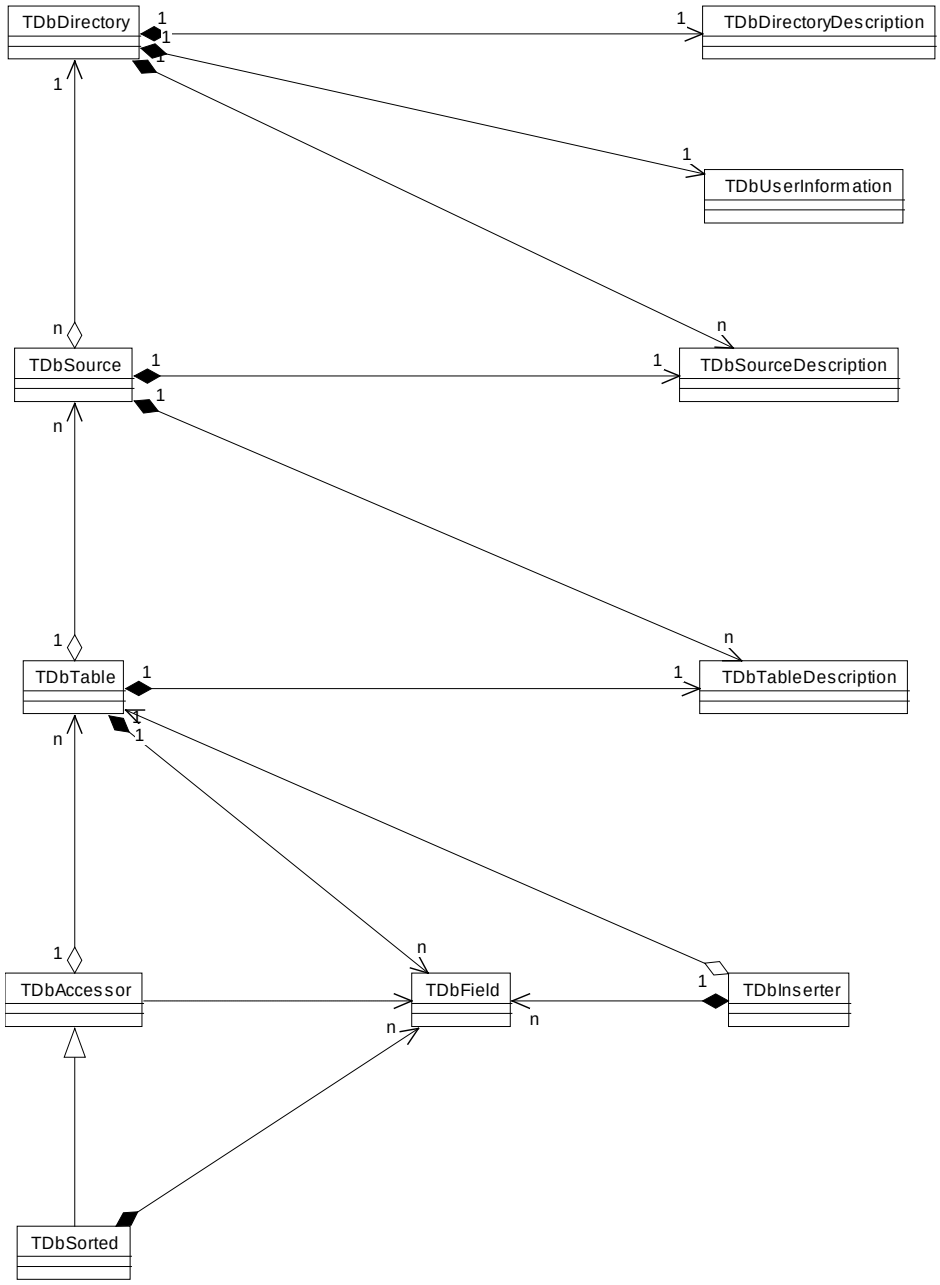


**Figure 2. Class diagram for the TRANSIMS database subsystem (unified notation).**

## 1. TDbDirectory

The data directory manages the available data sources and provides services for finding and manipulating the data sources. Each data directory has a description and several data sources, and each directory is connected to an underlying relational database via DBtools.h++ `RWDBDatabase` and `RWDBConnection` objects.

```
TDbDirectory(const TDbDirectoryDescription& description, const
           TDbUserInformation& user = TDbUserInformation())
```

Open the specified data directory. The exception `TDbDoesNotExist` is thrown if the specified data directory does not exist.

`TDbDirectory(const TDbDirectory& directory);`
Create a copy of the specified directory.

`TDbDirectory& operator=(const TDbDirectory& directory)`
Make the directory a copy of the specified directory.

`const TDbDirectoryDescription& GetDescription() const`
Return the data directory's description.

`bool HasSource(const string& name) const`
Return whether the specified data source is available.

`const TDbSourceDescription& GetSource(const string& name) const`
Get the specified data source's description. The exception `TDbDoesNotExist` is thrown if the specified data source does not exist.

`const SourceDescriptionSet& GetSources() const`
Get the descriptions of the available data sources.

`void CreateSource(const TDbSourceDescription& description)`
Create a new data source in the data directory. The exception `TDbAlreadyExists` is thrown if the data source already exists.

`void DeleteSource(const TDbSourceDescription& source)`
Delete the specified data source from the data directory. The exception `TDbDoesNotExist` is thrown if the source does not exist.

`void Query(const string& statement)`
Perform an SQL query. The exception `TDbQueryFailed` is thrown if the query fails. Note that this exposes the relational database model.

`RWDBDatabase& GetDatabase()`
Return the database instance for the data directory. Note that this exposes the implementation.

`RWDBConnection& GetConnection()`
Return the connection instance for the data directory. Note that this exposes the implementation.

## 2. TDbDirectoryDescription

A data directory description uniquely specifies a data directory. A directory description has a name, a comment, a database name, a server name, and an access library. There is also a default server name and a default access library.

```
TDbDirectoryDescription(const string& name, const string& comment
            = "", const string& databaseName = "transims", const
            string& server = string(fgServerName), const string&
            accessLibrary = string(fgAccessLibrary))
```

Construct a directory description.

```
TDbDirectoryDescription(const TDbDirectoryDescription&
            description)
```

Construct a copy of the specified directory description.

```
TDbDirectoryDescription& operator=(const TDbDirectoryDescription&
            description)
```

Make the directory description a copy of the specified directory description.

```
const string& GetName() const
```

Return the name of the directory.

```
const string& GetComment() const
```

Return the comment for the directory.

```
const string& GetDatabaseName() const
```

Return the database name for the directory.

```
const string& GetServerName() const
```

Return the name of the server.

```
const string& GetAccessLibrary() const
```

Return the name of the access library.

```
bool operator==(const TDbDirectoryDescription& description) const
```

Return whether the directory description has the same name as the specified directory description.

```
bool operator!=(const TDbDirectoryDescription& description) const
```

Return whether the directory description has a name different from the specified directory description.

```
static const string GetDefaultServerName()
```

Return the name of the default server.

```
static void SetDefaultServerName(const string& serverName)
```

Set the name of the default server to the specified name.

```
static const string GetDefaultAccessLibrary()
```

Return the name of the default access library.

```
static void SetDefaultAccessLibrary(const string& accessLibrary)
```

Set the name of the default access library to the specified name.

## 3. TDbUserInformation

An instance of user information contains the information necessary for a user to access a database. A user information instance has a user name and a password. There is also a default user name and a default password.

```
TDbUserInformation(const string& userName = string(fgUserName),
            const string& password = string(fgPassword))
```

Construct a user information instance.

```
TDbUserInformation(const TDbUserInformation& information)
```

Construct a copy of the specified user information.

```
TDbUserInformation& operator=(const TDbUserInformation&
            information)
```

Make the user information a copy of the specified user information.

```
const string& GetUserName() const
```

Return the user name.

```
const string& GetPassword() const
```

Return the password.

```
static const string GetDefaultUserName()
```

Return the default user name.

```
static void SetDefaultUserName(const string& userName)
```

Set the default user name.

```
static const string GetDefaultPassword()
```

Return the default password.

```
static void SetDefaultPassword(const string& password)
```

Set the default password.

## 4. TDbSource

A data source organizes the different data tables (versions) that may exist for a type of data. Each data source is associated with a data directory and has a description and several tables.

```
TDbSource(TDbDirectory& directory, const TDbSourceDescription&
            description)
```

Open the specified data source. The exception `TDbDoesNotExist` is thrown if the specified source does not exist.

```
TDbSource(const TDbSource& source)
```

Construct a copy of the specified source.

`TDbSource& operator=(const TDbSource& source)`

Make the source a copy of the specified source.

`const TDbSourceDescription& GetDescription() const`

Return the data source's description.

`bool HasTable(const string& name) const`

Return whether the specified data table (version) of the data source exists.

`const TDbTableDescription& GetTable(const string& name) const`

Return the description for the specified data table (version) of the data source. The exception `TDbDoesNotExist` is thrown if the specified source does not exist.

`const TableDescriptionSet& GetTables() const`

Return the set of descriptions of the available data tables (versions) of the data source.

```
void CreateTable(const TDbTableDescription& description, const
        string& sql)
```

Create a new data table (version) of the data source with the specified description and using the specified SQL statement. The exception `TDbAlreadyExists` is thrown if the specified table already exists. The exception `TDbCreationFailed` is thrown if the table cannot be created. Note that this exposes the relational database model.

```
void CreateTable(const TDbTableDescription& description, const
        FieldCollection& fields, const FieldCollection& index)
```

Create a new data table (version) of the data source with the specified description and with the specified fields and primary index. The exception `TDbAlreadyExists` is thrown if the specified table already exists. The exception `TDbCreationFailed` is thrown if the table cannot be created.

`void DeleteTable(const TDbTableDescription& description)`

Delete the specified data table from the data source. The exception `TDbDoesNotExist` is thrown if the table does not exist.

`TDbDirectory& GetDirectory()`

Return the directory for the source.

`RWDBDatabase& GetDatabase()`

Return the database instance for the data source. Note that this exposes the implementation.

`RWDBConnection& GetConnection()`

Return the connection instance for the data source. Note that this exposes the implementation.

## 5. TDbSourceDescription

A data source description uniquely specifies a data source within a data directory. A source description has a name and a comment.

```
TDbSourceDescription(const string& name, const string& comment =
            "")
```

Construct a source description.

```
TDbSourceDescription(const TDbSourceDescription& description)
```

Construct a copy of the specified source description.

```
TDbSourceDescription& operator=(const TDbSourceDescription&
            description)
```

Make the source description a copy of the specified source description.

```
const string& GetName() const
```

Return the name of the source.

```
const string& GetComment() const
```

Return the comment for the source.

```
bool operator==(const TDbSourceDescription& description) const
```

Return whether the source description has the same name as the specified source description.

```
bool operator!=(const TDbSourceDescription& description) const
```

Return whether the source description has a name different from the specified source description.

## 6. TDbTable

A data table contains fields (i.e., columns or attributes) and records (i.e., rows or tuples). Each data table also is associated with a data source, and has a description, possibly some dependent tables, and several fields.

```
TDbTable(TDbSource& source, const TDbTableDescription&
            description)
```

Open the specified table. The exception `TDbDoesNotExist` is thrown if the specified table does not exist.

```
TDbTable(const TDbTable& table)
```

Construct a copy of the specified table.

```
TDbTable& operator=(const TDbTable& table)
```

Make the table a copy of the specified table.

```
const TDbTableDescription& GetDescription() const
```

Return the data table's description.

`const TableDescriptionSet& GetDependentTables() const`

Return the set of data tables on which the data table depends.

`void AddDependentTable(const TDbTableDescription& description)`

Add a the specified table to the dependents. The exception `TDbAlreadyExists` is thrown if the specified table is already a dependent.

`void RemoveDependentTable(const TDbTableDescription& description)`

Remove the specified table from the dependents. The exception `TDbAlreadyExists` is thrown if the specified table is already a dependent.

`bool HasField(const string& name) const`

Return whether the specified field exists in the data table.

`const TDbField& GetField(const string& name) const`

Return the specified field of the data table. The exception `TDbDoesNotExist` is thrown if the specified field does not exist.

`const FieldSet& GetFields() const`

Return the set of available fields in the data table.

`TDbSource& GetSource()`

Return the source for the table.

`TDbDirectory& GetDirectory()`

Return the directory for the table.

`RWDBDatabase& GetDatabase()`

Return the database instance for the data table. Note that this exposes the implementation.

`RWDBConnection& GetConnection()`

Return the connection instance for the data table. Note that this exposes the implementation.

## 7. TDbTableDescription

A data table description uniquely specifies a data table. A table description has a name, a comment, and a relational database (implementation) table name.

```
TDbTableDescription(const string& name, const string& comment =
            "", const string& tableName = "")
```

Construct a table description.

`TDbTableDescription(const TDbTableDescription& description)`

Construct a copy of the specified table description.

`TDbTableDescription& operator=(const TDbTableDescription& description)`

Make the table description a copy of the specified table description.

`const string& GetName() const`

Return the name of the source.

`const string& GetComment() const`

Return the comment for the source.

`const string& GetTableName() const`

Return the table name for the source.

`bool operator==(const TDbTableDescription& description) const`

Return whether the table description has the same name as the specified table description.

`bool operator!=(const TDbTableDescription& description) const`

Return whether the table description has a name different from the specified table description.

## 8. TDbAccessor

A table accessor provides facilities for navigating a data table's records. Each table accessor is associated with a table, and has DBtools.h++ `RWDBTable` and `RWDBReader` objects. The accessor may or may not be at a record (see Figure 3).

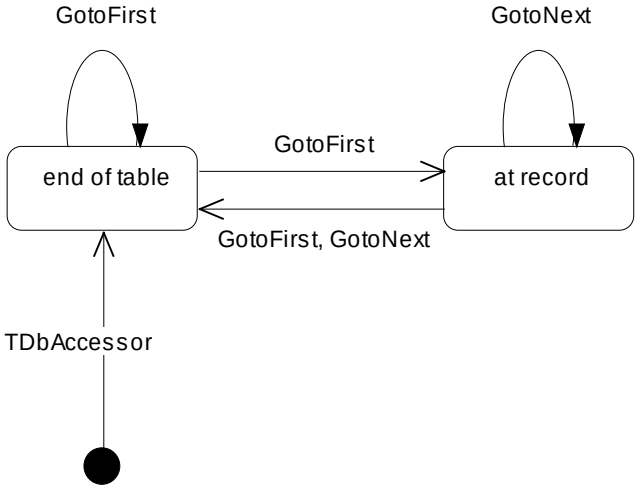


**Figure 3. An accessor is either at a record or at the end of the table. The operations `GotoFirst()` and `GotoNext()` are used to navigate and the operation `IsAtRecord()` reveals the current state.**

`TDbAccessor(TDbTable& table)`

Open an accessor for the specified table.

`TDbAccessor(const TDbAccessor& accessor)`

Create a copy of the specified accessor.

`TDbAccessor& operator=(const TDbAccessor& accessor)`

Make the accessor a copy of the specified accessor.

`TDbTable& GetTable()`

Return the accessor's data table.

`size_t GetRecordCount() const`

Return the number of records in the accessor's table.

`bool IsAtRecord() const`

Return whether the accessor is at a record.

`void GotoNext()`

Advance to the next record. The exception `TDbCannotRead` is thrown if the record cannot be read.

`virtual void GotoFirst()`

Position the accessor on the first record. The exception `TDbCannotRead` is thrown if the record cannot be read.

```
void GetField(const TDbField& field, char& value) const
void GetField(const TDbField& field, unsigned char& value) const
void GetField(const TDbField& field, short& value) const
void GetField(const TDbField& field, unsigned short& value) const
void GetField(const TDbField& field, int& value) const
void GetField(const TDbField& field, unsigned int& value) const
void GetField(const TDbField& field, long& value) const
void GetField(const TDbField& field, unsigned long& value) const
void GetField(const TDbField& field, float& value) const
void GetField(const TDbField& field, double& value) const
void GetField(const TDbField& field, string& value) const
```

Get the value for the specified field. The exception `TDbCannotRead` is thrown if the record cannot be read; the exception `TDbDoesNotExist` is thrown if the specified field does not exist.

`RWDBReader& GetReader()`

Return the reader instance for the accessor. Note that this exposes the implementation.

## 9. TDbSortedAccessor

A sorted table accessor provides facilities for navigating a data table's records sequentially. Each sorted table accessor has the fields sorted in a particular order.

```
TDbSortedAccessor(TDbTable& table, const
            TDbSource::FieldCollection& fields)
```

Open an accessor for the specified table, sorted by the specified fields.

`TDbSortedAccessor(const TDbSortedAccessor& accessor)`
Create a copy of the specified accessor.

`TDbSortedAccessor& operator=(const TDbSortedAccessor& accessor)`
Make the accessor a copy of the specified accessor.

`void GotoFirst()`
Position the accessor on the first record. The exception `TDbCannotRead` is thrown if the record cannot be read. Note that calling this function is an expensive operation.

`const TDbSource::FieldCollection& GetSortFields() const`
Return the sort fields for the table.

`RWDBReader& GetReader()`
Return the reader instance for the accessor. Note that this exposes the implementation. The reader is not necessarily sorted.

## 10. TDbInserter

This class is used for inserting data into a table. Each table inserter is associated with a table and has DBtools.h++ `RWDBTable`, `RWDBConnection`, and `RWDBPhraseBook` objects along with a field map holding the current values of the fields.

`TDbInserter(TDbTable& table)`
Open an inserter for the specified table.

`TDbInserter(const TDbInserter& inserter)`
Make a copy of the given inserter.

`TDbInserter& operator=(const TDbInserter& inserter)`
Make the inserter a copy of the given inserter.

`TDbTable& GetTable()`
Return the inserter's data table.

`void Insert()`
Insert the current record into the table. Note that the data may not be immediately written to disk. The exception `TDbCannotWrite` is thrown if the data cannot be written.

`void Flush()`
Flush any changes in the table not yet written to disk.

```
void SetField(const TDbField& field, char value)
void SetField(const TDbField& field, unsigned char value)
void SetField(const TDbField& field, short value)
void SetField(const TDbField& field, unsigned short value)
void SetField(const TDbField& field, int value)
void SetField(const TDbField& field, unsigned int value)
void SetField(const TDbField& field, long value)
void SetField(const TDbField& field, unsigned long value)
void SetField(const TDbField& field, float value)
void SetField(const TDbField& field, double value)
void SetField(const TDbField& field, const string& value)
void SetField(const TDbField& field)
```

Set the value of the specified field. The exception `TDbDoesNotExist` is thrown if the field is not in the table.

## 11. TDbField

A field is a column specification in a data table. A field has a name, a type, and may have a size (only if it is a string).

```
enum EType {kUnknown, kChar, kUnsignedChar, kShort,
            kUnsignedShort, kInt, kUnsignedInt, kLong,
            kUnsignedLong, kFloat, kDouble, kString}
```

Field types.

```
TDbField(const string& name, EType type = kUnknown, size_t size =
            0)
```

Create a field with the specified name and type.

```
TDbField(const TDbField& field)
```

Construct a copy of the specified field.

```
TDbField& operator=(const TDbField& field)
```

Make the field a copy of the specified field.

```
const string& GetName() const
```

Return the name of the field.

```
EType GetType() const
```

Return the type of the field.

```
size_t GetSize() const
```

Return the size of the field, if any.

```
bool operator==(const TDbField& field) const
```

Return whether the field has the same name as the specified field.

```
bool operator!=(const TDbField& field) const
```

Return whether the field has a name different from the specified field.

## 12. TDbException

A database exception signals the failure of a database subsystem function. Each exception has a message. Figure 4 shows the hierarchy of exception classes.

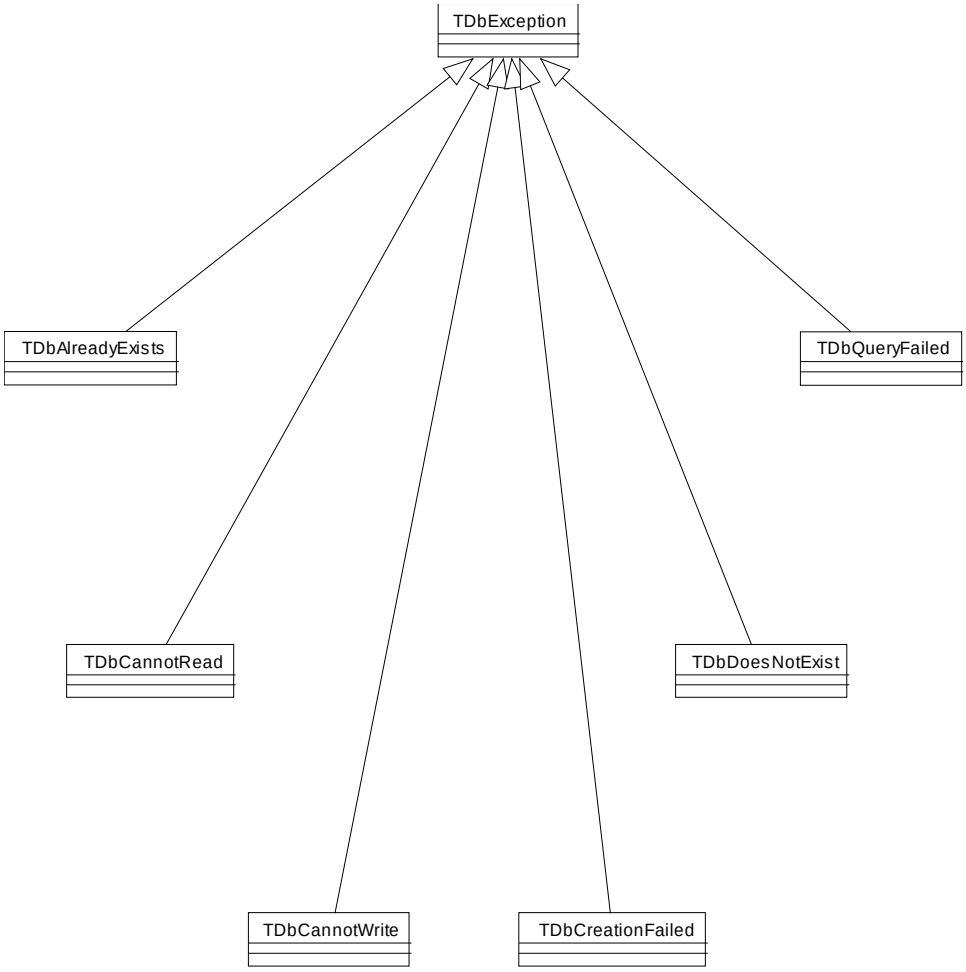


**Figure 4. Exception hierarchy for the TRANSIMS database subsystem (unified notation).**

`TDbException(const string& message = "Database error.")`
Construct an exception with the specified message text.

`TDbException(const TDbException& exception)`
Construct a copy of the specified exception.

`TDbException& operator=(const TDbException& exception)`
Make the exception a copy of the specified exception.

`const string& GetMessage() const`
Return the message text for the exception.

`class TDbAlreadyExists`

This exception is thrown when an attempt is made to create something that already exists.

`class TDbCannotRead`

This exception is thrown when an attempt to read data fails.

`class TDbCannotWrite`

This exception is thrown when an attempt to write data fails.

`class TDbDoesNotExist`

This exception is thrown when an attempt is made to access something that does not exist.

`class TDbQueryFailed`

This exception is thrown when a query fails.

`class TDbCreationFailed`

This exception is thrown when table creation fails.

# III. Implementation

## *A. C++ Libraries*

The current implementation of the database subsystem uses the C++ database access library DBtools.h++ from Rogue Wave Software [Ga 95a; SL 95; Su 95] to access an Oracle 7 relational database [KL 95; Lo 94]. This library isolates the subsystem from all of the details of the Oracle Call Interface (OCI) library. The TRANSIMS database subsystem can access relational databases from other vendors such as Sybase without modification—one only needs to modify the connection parameters in the `TDbDirectoryDescription` instance appropriately.

The Booch Components [RW 94] provide C++ container classes that the database subsystem uses extensively. The subsystem also uses the Tools.h++ library [Ke 94] (via the DBtools.h++ library dependence on Tools.h++), the standard C++ library [Pl 95], the standard C library [Pl 92], and the POSIX library [Ga 95b]. All of these libraries compile on a wide variety of platforms (UNIX and otherwise).

## *B. Relational Database*

Each data directory in the database subsystem corresponds to a database and user name combination in a relational database. Each `TDbDirectoryDescription` instance contains the name of the database, the name of the server on which it resides, and the name of the DBtools.h++ access library encapsulating the vendor-specific interface. Each `TDbUserInformation` instance contains the SQL user name and password within the relational database. The `TDbDirectory` constructor requires these two objects in order to connect to the correct data directory.

Each data source corresponds to an entry in the metadata table SOURCE_INDEX:

```
--  Source index schema.
CREATE TABLE SOURCE_INDEX (
    NAME VARCHAR(50) NOT NULL,
    COMMENT_TEXT VARCHAR(250),
    PRIMARY KEY (NAME)
);
```

The NAME and COMMENT_TEXT fields map directly to the fName and fComment data members of TDbSourceDescription instances, respectively.

Each data table corresponds to a relational database table *plus* an entry in the metadata table TABLE_INDEX:

```
--  Table index schema.
CREATE TABLE TABLE_INDEX (
    TABLE_NAME VARCHAR(50) NOT NULL,
    SOURCE VARCHAR(50) NOT NULL,
    NAME VARCHAR(50) NOT NULL,
    COMMENT_TEXT VARCHAR(250),
    PRIMARY KEY (TABLE_NAME),
    UNIQUE (SOURCE, NAME),
    FOREIGN KEY (SOURCE) REFERENCES SOURCE_INDEX
);
```

The NAME, COMMENT_TEXT, and TABLE_NAME fields map directly to the fName, fComment, and fTableName data members of TDbTableDescription instances, respectively. The dependencies between data tables correspond to entries in the metadata table DEPENDENT_TABLES:

```
--  Dependent tables schema.
CREATE TABLE DEPENDENT_TABLES (
    TABLE_NAME VARCHAR(50) NOT NULL,
    DEPENDENT_NAME VARCHAR(50) NOT NULL,
    PRIMARY KEY (TABLE_NAME, DEPENDENT_NAME),
    FOREIGN KEY (TABLE_NAME) REFERENCES TABLE_INDEX,
    FOREIGN KEY (DEPENDENT_NAME) REFERENCES TABLE_INDEX
);
```

See the appendix for instructions on how to create a TRANSIMS database. References [CAD 95; CG 93; Ve 95] provide information on database tuning and backup.

# IV. Usage

## *A. Examples*

The following examples show how to navigate, access, and add data to the database. The test and import programs in the appendix contain additional examples of using the subsystem.

### 1. Retrieving Data

The following example shows how to navigate through the metadata to retrieve data from a table.

```
//  Open the data directory.
TDbDirectory directory(TDbDirectoryDescription("IOC-1"));

//  Get the data source.
TDbSource nodeSource(directory, directory.GetSource("Node"));

//  Get the data table.
TDbTable nodeTable(nodeSource, nodeSource.GetTable(line));

//  Open an accessor for the data table.
TDbAccessor nodeAccessor(nodeTable);

//  Get the field.
TDbField idField(nodeTable.GetField("ID")),

//  Iterate over the records in the table.
for (nodeAccessor.GotoFirst(); nodeAccessor.IsAtRecord();
        nodeAccessor.GotoNext() {

    //  Declare the variable for receiving the data.
    NetNodeId id;

    //  Receive the data.
    fAccessor.GetField(fIdField, id);

    //  Print the result.
    cout << id << endl;

//  Delimit the iteration.
}
```

## 2. Inserting Data

The following example shows how to insert data into a table.

```
//  Open the directory.
TDbDirectory directory(TDbDirectoryDescription("IOC-1"));

//  Create the table description.
const TDbTableDescription description(“Test Table“,
        “This is a test table.“, “TEST_TABLE“);

//  Get the data source.
TDbSource source(directory, directory.GetSource(“Test Source“));

//  Create the fields for the table.
TDbSource::FieldCollection fields;
fields.Append(TDbField("StringField", TDbField::kString, 10));
fields.Append(TDbField("DoubleField", TDbField::kDouble));

//  Create the index for the table.
TDbSource::FieldCollection index;
index.Append(fields[1]);

//  Create the table.
source.CreateTable(description, fields, index);

//  Get the data table.
TDbTable table(source, description);
```

```
//  Open an inserter for the data table.
TDbInserter inserter(table);

//  Set the data to be inserted.
inserter.SetField(table.GetField("StringField"), "a string");
inserter.SetField(table.GetField("DoubleField"), 4.1);

//  Insert the record.
inserter.Insert();
```

### *B. Notes*

#### 1. Direct Data Manipulation

Raw SQL, DBtools.h++ objects, and the Oracle Call Interface (OCI) are all available for use within the database subsystem by the higher-level subsystems, but such use is not encouraged because it violates the principle of hiding the database implementation.

It is also possible to manipulate data externally using products such as SQLPLUS [KL 95; La 92]. However, care must be taken to keep the metadata in the table `SOURCE_INDEX`, `TABLE_INDEX`, and `DEPENDENT_TABLES` consistent and valid.

#### 2. Object Interdependencies

The following dependencies exists among instances of database objects:

- A `TDbSource` instance depends on the continued existence of the `TDbDirectory` instance that was passed to it in its constructor.
- A `TDbTable` instance depends on the continued existence of the `TDbSource` instance that was passed to it in its constructor.
- A `TDbAccessor` or `TDbInserter` instance depends on the continued existence of the `TDbTable` instance that was passed to it in its constructor.

If any of the above rules are violated, the dependent object will have a dangling pointer, and unpredictable behavior will occur.

#### 3. Description Classes

The descriptive classes `TDbDirectoryDescription`, `TDbUserInformation`, `TDbSourceDescription`, `TDbTableDescription`, and `TDbField` have equality and inequality operators that test the equality or inequality *only* of the data member that specifies the object uniquely (i.e., the test is not on all of the data members or on the object pointers). This allows one to search for descriptions in Booch sets using the `BC_Set::IsMember()` function.

## V. Future Work

Future work planned for the TRANSIMS data subsystem will focus on providing a highly portable alternative implementation that is not based on commercial products such as DBtools.h++ and Oracle. Performance enhancements and additional features such as temporary tables are also being considered.

## VI. References


[CAD 95] M. J. Corey, M. Abbey, and D. J. Dechichio, Jr., *Tuning Oracle*, (Berkeley, California: McGraw-Hill, 1995).

[CG 93] P. Corrigan and M. Gurry, *Oracle Performance Tuning*, (Sebastopol, California: O'Reilly & Associates, 1993).

[Ga 95a] J. Gay, *DBtools.h++ Access Library for Oracle*, Version 1, (Corvallis, Oregon: Rogue Wave Software, 1995).

[Ga 95b] B. O. Gallmeister, *POSIX.4: Programming for the Real World*, (Sebastopol, California: O'Reilly & Associates, 1995).

[Ke 94] T. Keffer, *Tools.h++ Introduction and Reference Manual*, Version 6, (Corvallis, Oregon: Rogue Wave Software, 1994).

[KL 95] G. Koch and K. Loney, *ORACLE: The Complete Reference*, Third Edition, (Berkeley, California: McGraw-Hill, 1995).

[La 92] R. F. van der Lans, *The SQL Guide to Oracle*, (Reading, Massachusetts: Addison-Wesley, 1992).

[Lo 94] K. Loney, *Oracle DBA Handbook*, (Berkeley, California: McGraw-Hill, 1994).

[Pl 92] P. J. Plauger, *The Standard C Library*, (Englewood Cliffs, New Jersey: Prentice Hall, 1992).

[Pl 95] P. J. Plauger, *The Draft Standard C++ Library*, (Englewood Cliffs, New Jersey: Prentice Hall, 1995).

[RW 94] Rogue Wave Software, *The C++ Booch Components*, Version 2.3, (Corvallis, Oregon: Rogue Wave Software, 1994).

[SL 95] S. Sulsky and K. L. Lohn, *DBtools.h++ User's Guide and Tutorial*, Version 1, (Corvallis, Oregon: Rogue Wave Software, 1995).

[Su 95] S. Sulsky, *DBtools.h++ Class Reference*, Version 1, (Corvallis, Oregon: Rogue Wave Software, 1995).

[Ve 95] R. Velpuri, *Oracle Backup & Recovery Handbook*, (Berkeley, California: McGraw-Hill, 1994).


## VII. APPENDIX: Booch Notation Diagrams

Figure 5 repeats the class diagram of Figure 2 using Booch notation and Figure 6 repeats the exception hierarchy of Figure 4 using Booch notation.

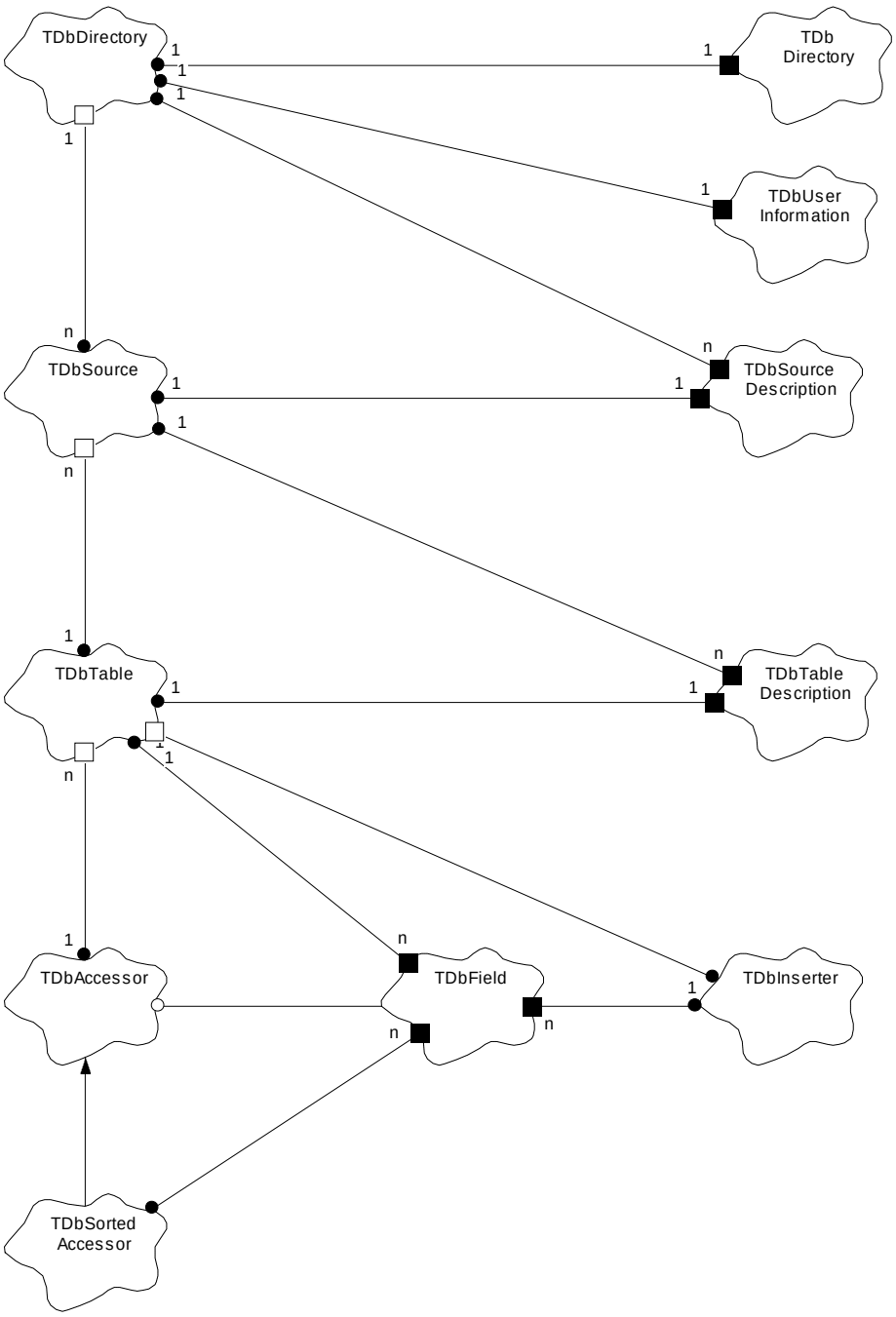


**Figure 5. Class diagram for the TRANSIMS database subsystem (Booch notation).**

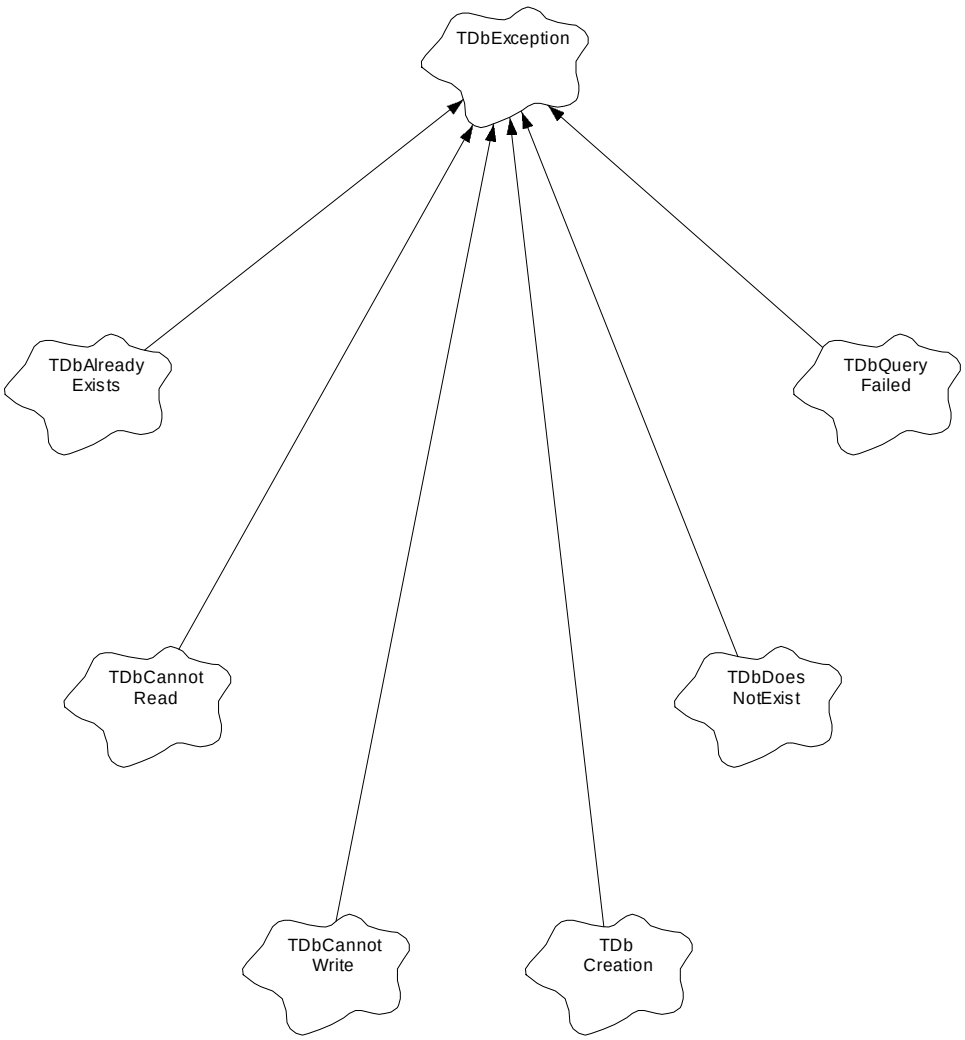


**Figure 6. Exception hierarchy for the TRANSIMS database subsytem (Booch notation).**

# VIII. APPENDIX: Database Creation

## *A. Oracle 7 Parameters*

Oracle 7 requires several configuration files containing database parameters. We provide a complete example configuration below. The file `inittransims.ora` is the main parameter file that sets general parameters and includes the TRANSIMS-specific parameter file `inittransims.ora`. That latter file defines control and dump file locations and logical names. The files `tnsnames.ora`, `listener.ora`, and `sqlnet.ora` configure the service names, the listener, and SQL*Net so that the client machines can access the database server. Refer to the Oracle administrator's manuals or Reference [Lo 94] for complete details on the format and meaning of the parameters below.

### 1. inittransims.ora

```
#   Project: TRANSIMS
#   Subsystem: Database
```

```
#   $RCSfile: inittransims.ora,v $
#   $Revision: 1.6 $
#   $Date: 1995/08/10 17:03:01 $
#   $State: Stab $
#   $Author: bwb $
#   Copyright © 1995 Regents of the University of California
#   All rights reserved


ifile = /disks/sousa2/oracle/transims/configtransims.ora

db_files = 20
db_file_multiblock_read_count = 16
db_block_buffers = 550
shared_pool_size = 6000000
log_checkpoint_interval = 10000
processes = 100
dml_locks = 200
log_buffer = 32768
sequence_cache_entries = 30
sequence_cache_hash_buckets = 23
max_dump_file_size = 10240
mts_dispatchers="ipc,1"
mts_max_dispatchers=10
mts_servers=1
mts_max_servers=10
mts_listener_address="(ADDRESS=(PROTOCOL=ipc)(KEY=transims))"
```

## 2. configtransims.ora

```
#   Project: TRANSIMS
#   Subsystem: Database
#   $RCSfile: configtransims.ora,v $
#   $Revision: 1.6 $
#   $Date: 1995/08/10 17:03:01 $
#   $State: Stab $
#   $Author: bwb $
#   Copyright © 1995 Regents of the University of California
#   All rights reserved


control_files = (/disks/sousa2/oracle/transims/control01.ctl,
                 /disks/sousa3/oracle/transims/control02.ctl,
                 /disks/sousa4/oracle/transims/control03.ctl)

#rollback_segments = ()
#rollback_segments = (rs00)
rollback_segments = (rs01,rs02)

background_dump_dest = /disks/sousa2/oracle/transims/bdump
core_dump_dest = /disks/sousa2/oracle/transims/cdump
user_dump_dest = /disks/sousa2/oracle/transims/udump

db_block_size = 4096
resource_limit = false
os_authent_prefix = ""
license_max_sessions = 8

db_name = transims
db_domain = sousa.tsasa.lanl.gov
```

## 3. tnsnames.ora

```
################
# Filename......: tnsnames.ora
# Name..........: LOCAL_REGION.world
# Date..........: 22-MAY-95 07:13:54
################
transims.world =
  (DESCRIPTION =
    (ADDRESS_LIST =
        (ADDRESS =
          (COMMUNITY = tsasa.world)
          (PROTOCOL = TCP)
          (Host = sousa.tsasa.lanl.gov)
```

```
          (Port = 1525)
        )
    )
    (CONNECT_DATA =
       (SID = transims)
       (GLOBAL_NAME = transims.world)
    )
  )
```

### 4. listener.ora

```
################
# Filename......: listener.ora
# Name..........: sousa.world
# Date..........: 22-MAY-95 07:13:54
################
LISTENER =
  (ADDRESS_LIST =
        (ADDRESS=
          (PROTOCOL=IPC)
          (KEY= transims.world)
        )
        (ADDRESS=
          (PROTOCOL=IPC)
          (KEY= transims)
        )
        (ADDRESS =
          (COMMUNITY = tsasa.world)
          (PROTOCOL = TCP)
          (Host = sousa.tsasa.lanl.gov)
          (Port = 1525)
        )
  )
STARTUP_WAIT_TIME_LISTENER = 0
CONNECT_TIMEOUT_LISTENER = 10
TRACE_LEVEL_LISTENER = OFF
SID_LIST_LISTENER =
  (SID_LIST =
    (SID_DESC =
      (SID_NAME = transims)
      (ORACLE_HOME = /usr/oracle)
    )
  )
PASSWORDS_LISTENER = (CF4F2809E5A7CC8F)
```

### 5. sqlnet.ora

```
################
# Filename......: sqlnet.ora
# Name..........: sousa.world
# Date..........: 22-MAY-95 07:13:54
################
AUTOMATIC_IPC = ON
TRACE_LEVEL_CLIENT = OFF
SQLNET.EXPIRE_TIME = 0
NAMES.DEFAULT_DOMAIN = world
NAME.DEFAULT_ZONE = world
SQLNET.CRYPTO_SEED = "32207116214195"
```

## *B. SQL Scripts for Database Creation*

In order to create a new TRANSIMS database in Oracle 7, one must execute the SQL scripts below in order. The first script, `MakeDatabase1.sql`, creates redo logs and a system tablespace. The second script, `MakeDatabase2.sql`, generates the data dictionary, rollback segments, and additional tablespaces. When using these scripts, one must adjust the rollback segment parameters in `configtransims.ora` appropriately.

The script `Database.sql` generates the tables that store TRANSIMS metadata (`SOURCE_INDEX`, `TABLE_INDEX`, and `DEPENDENT_TABLES`). The scripts

`MakeRoll.sql` and `MakeUser.sql` create a role and a user, respectively, for TRANSIMS access to the database. These three scripts are written in standard SQL and will work on databases other than Oracle 7.

## 1. MakeDatabase1.sql

```
--  Project: TRANSIMS
--  Subsystem: Database
--  $RCSfile: MakeDatabase1.sql,v $
--  $Revsion$
--  $Date: 1995/08/10 17:02:21 $
--  $State: Stab $
--  $Author: bwb $
--  Copyright © 1995 Regents of the University of California
--  All rights reserved


--  Create the database.
CONNECT INTERNAL

STARTUP PFILE=/disks/sousa1/oracle/dbs/inittransims.ora NOMOUNT

CREATE DATABASE transims
  MAXINSTANCES 1
  MAXLOGFILES 16
  LOGFILE
    GROUP 1
      ('/disks/sousa2/oracle/transims/redo1a.log',
        '/disks/sousa2/oracle/transims/redo1b.log')
      SIZE 1m,
    GROUP 2
      ('/disks/sousa3/oracle/transims/redo3a.log',
        '/disks/sousa3/oracle/transims/redo3b.log')
      SIZE 1m,
    GROUP 3
      ('/disks/sousa4/oracle/transims/redo3a.log',
        '/disks/sousa4/oracle/transims/redo3b.log')
      SIZE 1m
  DATAFILE
    '/disks/sousa3/oracle/transims/system01.dbf'
    SIZE 100m;

DISCONNECT

--  Next steps:
--    .  Startup the database.
--    .  Continue with MakeDatabase2.sql.
```

## 2. MakeDatabase2.sql

```
--  Project: TRANSIMS
--  Subsystem: Database
--  $RCSfile: MakeDatabase2.sql,v $
--  $Revsion$
--  $Date: 1996/04/10 15:30:18 $
--  $State: Rel $
--  $Author: bwb $
--  Copyright © 1995 Regents of the University of California
--  All rights reserved


--  Create the data dictionary.
@/disks/sousa1/oracle/rdbms/admin/catalog.sql


--  Create a rollback segment in system.
CREATE ROLLBACK SEGMENT rs00
  TABLESPACE system;

ALTER ROLLBACK SEGMENT rs00 ONLINE;


--  Create additional tablespaces.
```

```
CREATE TABLESPACE rbs
  DATAFILE
    '/disks/sousa4/oracle/transims/rbs01.dbf' SIZE 100m,
    '/disks/sousa4/oracle/transims/rbs02.dbf' SIZE 100m,
    '/disks/sousa4/oracle/transims/rbs03.dbf' SIZE 100m
  DEFAULT STORAGE (INITIAL 1m NEXT 1m PCTINCREASE 0 MINEXTENTS 9 MAXEXTENTS 50);

CREATE TABLESPACE temp
  DATAFILE
    '/disks/sousa5/oracle/transims/temp01.dbf' SIZE 825m,
    '/disks/sousa19/oracle/transims/temp02.dbf' SIZE 825m,
    '/disks/sousa20/oracle/transims/temp03.dbf' SIZE 825m
  DEFAULT STORAGE (INITIAL 825k NEXT 825k PCTINCREASE 5);

CREATE TABLESPACE users
  DATAFILE
    '/disks/sousa6/oracle/transims/users01.dbf' SIZE 825m,
    '/disks/sousa7/oracle/transims/users02.dbf' SIZE 825m,
    '/disks/sousa8/oracle/transims/users03.dbf' SIZE 825m,
    '/disks/sousa11/oracle/transims/users04.dbf' SIZE 825m,
    '/disks/sousa12/oracle/transims/users05.dbf' SIZE 825m,
    '/disks/sousa13/oracle/transims/users06.dbf' SIZE 825m,
    '/disks/sousa14/oracle/transims/users07.dbf' SIZE 825m
  DEFAULT STORAGE (INITIAL 825k NEXT 825k PCTINCREASE 5);

CREATE TABLESPACE idx
  DATAFILE
    '/disks/sousa9/oracle/transims/idx01.dbf' SIZE 825m,
    '/disks/sousa10/oracle/transims/idx02.dbf' SIZE 825m
  DEFAULT STORAGE (INITIAL 825k NEXT 825k PCTINCREASE 5);


--  Create additional rollback segments.
CREATE PUBLIC ROLLBACK SEGMENT rs01
  TABLESPACE rbs
  STORAGE (OPTIMAL 100m);

CREATE PUBLIC ROLLBACK SEGMENT rs02
  TABLESPACE rbs
  STORAGE (OPTIMAL 100m);

ALTER ROLLBACK SEGMENT rs01 ONLINE;
ALTER ROLLBACK SEGMENT rs02 ONLINE;

ALTER ROLLBACK SEGMENT rs00 OFFLINE;
REM DROP ROLLBACK SEGMENT rs00;


--  Change user tablespaces.
ALTER USER sys TEMPORARY TABLESPACE temp;
ALTER USER system TEMPORARY TABLESPACE temp;


--  Next steps:
--    .  Shutdown the database.
--    .  Edit the profile to include the new rollback segments.
--    .  Startup the database.
```

## 3. Database.sql

```
--  Project: TRANSIMS
--  Subsystem: Database
--  $RCSfile: Database.sql,v $
--  $Revision: 1.5 $
--  $Date: 1995/06/21 15:00:34 $
--  $State: Stab $
--  $Author: bwb $
--  Copyright © 1995 Regents of the University of California
--  All rights reserved


--  Source index schema.
CREATE TABLE SOURCE_INDEX (
    NAME VARCHAR(50) NOT NULL,
    COMMENT_TEXT VARCHAR(250),
```

```
    PRIMARY KEY (NAME)
);


--  Table index schema.
CREATE TABLE TABLE_INDEX (
    TABLE_NAME VARCHAR(50) NOT NULL,
    SOURCE VARCHAR(50) NOT NULL,
    NAME VARCHAR(50) NOT NULL,
    COMMENT_TEXT VARCHAR(250),
    PRIMARY KEY (TABLE_NAME),
    UNIQUE (SOURCE, NAME),
    FOREIGN KEY (SOURCE) REFERENCES SOURCE_INDEX
);


--  Dependent tables schema.
CREATE TABLE DEPENDENT_TABLES (
    TABLE_NAME VARCHAR(50) NOT NULL,
    DEPENDENT_NAME VARCHAR(50) NOT NULL,
    PRIMARY KEY (TABLE_NAME, DEPENDENT_NAME),
    FOREIGN KEY (TABLE_NAME) REFERENCES TABLE_INDEX,
    FOREIGN KEY (DEPENDENT_NAME) REFERENCES TABLE_INDEX
);
```

## 4. MakeRole.sql

```
--  Project: TRANSIMS
--  Subsystem: Database
--  $RCSfile: MakeRole.sql,v $
--  $Revision: 1.6 $
--  $Date: 1995/08/10 17:02:21 $
--  $State: Stab $
--  $Author: bwb $
--  Copyright © 1995 Regents of the University of California
--  All rights reserved.


--  STAFF role.
CREATE ROLE STAFF;


--  Privileges for STAFF role.
GRANT
    ALTER SESSION,
    CREATE CLUSTER,
    CREATE PROCEDURE,
    CREATE SEQUENCE,
    CREATE SESSION,
    CREATE SYNONYM,
    CREATE TABLE,
    CREATE TRIGGER,
    CREATE VIEW
  TO STAFF;
```

## 5. MakeUser.sql

```
--  Project: TRANSIMS
--  Subsystem: Database
--  $RCSfile: MakeUser.sql,v $
--  $Revision: 1.6 $
--  $Date: 1995/08/10 17:02:21 $
--  $State: Stab $
--  $Author: bwb $
--  Copyright © 1995 Regents of the University of California
--  All rights reserved


--  User creation.
CREATE USER ioc1
  IDENTIFIED BY lanl
  DEFAULT TABLESPACE USERS
  TEMPORARY TABLESPACE TEMP
  QUOTA UNLIMITED ON USERS;
```

```
--  User privileges.
GRANT STAFF TO ioc1;
ALTER USER ioc1
  DEFAULT ROLE STAFF;
```

# IX. APPENDIX: Source Code

This appendix contains the complete C++ source code for the database subsystem classes.

## *A. TDbAccessor Class*

### 1. Accessor.h

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: Accessor.h,v $
//  $Revision: 2.1 $
//  $Date: 1996/02/20 22:18:17 $
//  $State: Exp $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


#ifndef TRANSIMS_DB_ACCESSOR
#define TRANSIMS_DB_ACCESSOR


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <DBS/Table.h>
#include <DBS/TableDescription.h>
#include <DBS/Field.h>


//  Include DBtools.h++ header files.
#include <rw/db/reader.h>


//  A table accessor provides facilities for navigating and editing a data
//  table's records.
class TDbAccessor
{
  public:

    //  Open an accessor for the specified table.
    TDbAccessor(TDbTable& table);

    //  Create a copy of the specified accessor.
//  TDbAccessor(const TDbAccessor& accessor);

    //  Make the accessor a copy of the specified accessor.
//  TDbAccessor& operator=(const TDbAccessor& accessor);

    //  Return the accessor's data table.
    TDbTable& GetTable();

    //  Return the number of records in the accessor's table.
    size_t GetRecordCount() const;

    //  Return whether the accessor is at a record.
    bool IsAtRecord() const;

    //  Advance to the next record.  The exception TDbCannotRead is thrown if
    //  the record cannot be read.
    void GotoNext();

    //  Position the accessor on the first record.  The exception TDbCannotRead
    //  is thrown if the record cannot be read.
    virtual void GotoFirst();
```

```
    //  Get the value for the specified field.  The exception TDbCannotRead is
    //  thrown if the record cannot be read; the exception TDbDoesNotExist is
    //  thrown if the specified field does not exist.
    void GetField(const TDbField& field, char& value) const;
    void GetField(const TDbField& field, unsigned char& value) const;
    void GetField(const TDbField& field, short& value) const;
    void GetField(const TDbField& field, unsigned short& value) const;
    void GetField(const TDbField& field, int& value) const;
    void GetField(const TDbField& field, unsigned int& value) const;
    void GetField(const TDbField& field, long& value) const;
    void GetField(const TDbField& field, unsigned long& value) const;
    void GetField(const TDbField& field, float& value) const;
    void GetField(const TDbField& field, double& value) const;
    void GetField(const TDbField& field, string& value) const;

    //  Return the reader instance for the accessor.  Note that this exposes the
    //  implementation.
    RWDBReader& GetReader();

  protected:

    //  Each table accessor is associated with a table.
    TDbTable* fTable;

    //  Each table accessor has a database table.
    RWDBTable fDbTable;

    //  Each table accessor has a reader.
    RWDBReader fReader;

    //  The reader may be at a record or not.
    bool fIsAtRecord;
};


#endif // TRANSIMS_DB_ACCESSOR
```

## 2. Accessor.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: Accessor.C,v $
//  $Revision: 2.1 $
//  $Date: 1996/02/20 22:18:17 $
//  $State: Exp $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include TRANSIMS header files.
#include <DBS/Exception.h>
#include <DBS/Accessor.h>


//  Open an accessor for the specified table.
TDbAccessor::TDbAccessor(TDbTable& table)
  : fTable(&table),
    fDbTable(table.GetDatabase().table(table.GetDescription().GetTableName())),
    fReader(fDbTable.reader()),
    fIsAtRecord(FALSE)
{
}


//  Return the accessor's data table.
TDbTable& TDbAccessor::GetTable()
{
    return *fTable;
}


//  Return the number of records in the accessor's table.
size_t TDbAccessor::GetRecordCount() const
{
```

```
    try {
        RWDBSelector selector = fTable->GetDatabase().selector();
        selector.from(fDbTable);
        selector << rwdbCount();
        RWDBReader reader = selector.reader(fTable->GetConnection());
        size_t result;
        reader();
        reader >> result;
        return result;
    } catch(RWDBStatus status) {
        throw TDbException(status.message());
    }
}


//  Return whether the accessor is at a record.
bool TDbAccessor::IsAtRecord() const
{
    return fIsAtRecord;
}


//  Advance to the next record.  The exception TDbCannotRead is thrown if the
//  record cannot be read.
void TDbAccessor::GotoNext()
{
    try {
        fIsAtRecord = fReader() != 0;
    } catch(RWDBStatus status) {
        throw TDbCannotRead(status.message());
    }
}


//  Position the accessor on the first record.  The exception TDbCannotRead is
//  thrown if the record cannot be read.
void TDbAccessor::GotoFirst()
{
    try {
        fReader = fDbTable.reader();
        fIsAtRecord = fReader() != 0;
    } catch(RWDBStatus status) {
        throw TDbCannotRead(status.message());
    }
}


//  Get the value for the specified field.  The exception TDbCannotRead is
//  thrown if the record cannot be read; the exception TDbDoesNotExist is thrown
//  if the specified field does not exist.
void TDbAccessor::GetField(const TDbField& field, char& value) const
{
    if (!fIsAtRecord)
        throw TDbCannotRead("The accessor is not at a record.");
    if (!fTable->GetFields().IsMember(field))
        throw TDbDoesNotExist("Field does not exist.");
    try {
        ((RWDBReader&) fReader)[field.GetName()] >> value;
    } catch (RWDBStatus status) {
        throw TDbCannotRead(status.message());
    }
}

void TDbAccessor::GetField(const TDbField& field, unsigned char& value) const
{
    if (!fIsAtRecord)
        throw TDbCannotRead("The accessor is not at a record.");
    if (!fTable->GetFields().IsMember(field))
        throw TDbDoesNotExist("Field does not exist.");
    try {
        ((RWDBReader&) fReader)[field.GetName()] >> value;
    } catch (RWDBStatus status) {
        throw TDbCannotRead(status.message());
    }
}
```

```
void TDbAccessor::GetField(const TDbField& field, short& value) const
{
    if (!fIsAtRecord)
        throw TDbCannotRead("The accessor is not at a record.");
    if (!fTable->GetFields().IsMember(field))
        throw TDbDoesNotExist("Field does not exist.");
    try {
        ((RWDBReader&) fReader)[field.GetName()] >> value;
    } catch (RWDBStatus status) {
        throw TDbCannotRead(status.message());
    }
}

void TDbAccessor::GetField(const TDbField& field, unsigned short& value) const
{
    if (!fIsAtRecord)
        throw TDbCannotRead("The accessor is not at a record.");
    if (!fTable->GetFields().IsMember(field))
        throw TDbDoesNotExist("Field does not exist.");
    try {
        ((RWDBReader&) fReader)[field.GetName()] >> value;
    } catch (RWDBStatus status) {
        throw TDbCannotRead(status.message());
    }
}

void TDbAccessor::GetField(const TDbField& field, int& value) const
{
    if (!fIsAtRecord)
        throw TDbCannotRead("The accessor is not at a record.");
    if (!fTable->GetFields().IsMember(field))
        throw TDbDoesNotExist("Field does not exist.");
    try {
        ((RWDBReader&) fReader)[field.GetName()] >> value;
    } catch (RWDBStatus status) {
        throw TDbCannotRead(status.message());
    }
}

void TDbAccessor::GetField(const TDbField& field, unsigned int& value) const
{
    if (!fIsAtRecord)
        throw TDbCannotRead("The accessor is not at a record.");
    if (!fTable->GetFields().IsMember(field))
        throw TDbDoesNotExist("Field does not exist.");
    try {
        ((RWDBReader&) fReader)[field.GetName()] >> value;
    } catch (RWDBStatus status) {
        throw TDbCannotRead(status.message());
    }
}

void TDbAccessor::GetField(const TDbField& field, long& value) const
{
    if (!fIsAtRecord)
        throw TDbCannotRead("The accessor is not at a record.");
    if (!fTable->GetFields().IsMember(field))
        throw TDbDoesNotExist("Field does not exist.");
    try {
        ((RWDBReader&) fReader)[field.GetName()] >> value;
    } catch (RWDBStatus status) {
        throw TDbCannotRead(status.message());
    }
}

void TDbAccessor::GetField(const TDbField& field, unsigned long& value) const
{
    if (!fIsAtRecord)
        throw TDbCannotRead("The accessor is not at a record.");
    if (!fTable->GetFields().IsMember(field))
        throw TDbDoesNotExist("Field does not exist.");
    try {
        ((RWDBReader&) fReader)[field.GetName()] >> value;
    } catch (RWDBStatus status) {
        throw TDbCannotRead(status.message());
    }
```

```
}

void TDbAccessor::GetField(const TDbField& field, float& value) const
{
    if (!fIsAtRecord)
        throw TDbCannotRead("The accessor is not at a record.");
    if (!fTable->GetFields().IsMember(field))
        throw TDbDoesNotExist("Field does not exist.");
    try {
        ((RWDBReader&) fReader)[field.GetName()] >> value;
    } catch (RWDBStatus status) {
        throw TDbCannotRead(status.message());
    }
}

void TDbAccessor::GetField(const TDbField& field, double& value) const
{
    if (!fIsAtRecord)
        throw TDbCannotRead("The accessor is not at a record.");
    if (!fTable->GetFields().IsMember(field))
        throw TDbDoesNotExist("Field does not exist.");
    try {
        ((RWDBReader&) fReader)[field.GetName()] >> value;
    } catch (RWDBStatus status) {
        throw TDbCannotRead(status.message());
    }
}

void TDbAccessor::GetField(const TDbField& field, string& value) const
{
    if (!fIsAtRecord)
        throw TDbCannotRead("The accessor is not at a record.");
    if (!fTable->GetFields().IsMember(field))
        throw TDbDoesNotExist("Field does not exist.");
    try {
        RWCString temp;
        ((RWDBReader&) fReader)[field.GetName()] >> temp;
        value = temp;
    } catch (RWDBStatus status) {
        throw TDbCannotRead(status.message());
    }
}


//  Return the reader instance for the accessor.  Note that this exposes the
//  implementation.
RWDBReader& TDbAccessor::GetReader()
{
    return fReader;
}
```

## *B. TDbDirectory Class*

### 1. Directory.h

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: Directory.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


#ifndef TRANSIMS_DB_DIRECTORY
#define TRANSIMS_DB_DIRECTORY


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <DBS/UserInformation.h>
#include <DBS/DirectoryDescription.h>
```

```
#include <DBS/SourceDescription.h>


//  Include Booch Components header files.
#include <BCStoreM.h>
#include <BCSetU.h>


//  Include DBtools.h++ header files.
#include <rw/db/dbase.h>
#include <rw/db/dbmgr.h>
#include <rw/db/connect.h>


//  The data directory manages the available data sources and provides services
//  for finding and manipulating the data sources.
class TDbDirectory
{
  public:

    //  Type definitions.
    typedef BC_TUnboundedSet<const TDbSourceDescription, 25U, BC_CManaged>
            SourceDescriptionSet;
    typedef BC_TSetActiveIterator<const TDbSourceDescription>
            SourceDescriptionSetIterator;

    //  Open the specified data directory.  The exception TDbDoesNotExist is
    //  thrown if the specified data directory does not exist.
    TDbDirectory(const TDbDirectoryDescription& description, const
            TDbUserInformation& user = TDbUserInformation());

    //  Create a copy of the specified directory.
//  TDbDirectory(const TDbDirectory& directory);

    //  Make the directory a copy of the specified directory.
//  TDbDirectory& operator=(const TDbDirectory& directory);

    //  Return the data directory's description.
    const TDbDirectoryDescription& GetDescription() const;

    //  Return whether the specified data source is available.
    bool HasSource(const string& name) const;

    //  Get the specified data source's description.  The exception
    //  TDbDoesNotExist is thrown if the specified data source does not exist.
    const TDbSourceDescription& GetSource(const string& name) const;

    //  Get the descriptions of the available data sources.
    const SourceDescriptionSet& GetSources() const;

    //  Create a new data source in the data directory.  The exception
    //  TDbAlreadyExists is thrown if the data source already exists.
    void CreateSource(const TDbSourceDescription& description);

    //  Delete the specified data source from the data directory.  The exception
    //  TDbDoesNotExist is thrown if the source does not exist.
    void DeleteSource(const TDbSourceDescription& source);

    //  Perform an SQL query.  The exception TDbQueryFailed is thrown if the
    //  query fails.  Note that this exposes the relational database model.
    void Query(const string& statement);

    //  Return the database instance for the data directory.  Note that this
    //  exposes the implementation.
    RWDBDatabase& GetDatabase();

    //  Return the connection instance for the data directory.  Note that this
    //  exposes the implementation.
    RWDBConnection& GetConnection();

  private:

    //  Update the source descriptions.
    void UpdateSourceDescriptions();

    //  Each data directory has a description.
```

```
    TDbDirectoryDescription fDescription;

    //  Each data directory has several data sources, identified by their
    //  descriptions.
    SourceDescriptionSet fSourceDescriptions;

    //  Each directory is connected to a database.
    RWDBDatabase fDatabase;

    //  Each directory has an explicit connection.
    RWDBConnection fConnection;
};


#endif // TRANSIMS_DB_DIRECTORY
```

## 2. Directory.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: Directory.C,v $
//  $Revision: 2.1 $
//  $Date: 1995/09/29 16:28:11 $
//  $State: Exp $
//  $Author: roberts $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include TRANSIMS header files.
#include <DBS/Exception.h>
#include <DBS/Directory.h>


//  Include DBtools.h++ header files.
#include <rw/db/table.h>
#include <rw/db/select.h>
#include <rw/db/reader.h>
#include <rw/db/inserter.h>
#include <rw/db/deleter.h>
#include <rw/db/result.h>


//  Define the error handler for DBtools.h++ errors.
static void ExceptionHandler(const RWDBStatus& status)
{
    throw status;
}


//  Define the hash function for source descriptions.
static BC_Index SourceDescriptionHashValue(const TDbSourceDescription&
        description)
{
    return HashValue(description.GetName());
}


//  Open the specified data directory.  The exception TDbDoesNotExist is thrown
//  if the specified data directory does not exist.
TDbDirectory::TDbDirectory(const TDbDirectoryDescription& description, const
        TDbUserInformation& user)
  : fDescription(description),
    fSourceDescriptions(SourceDescriptionHashValue),
    fDatabase(RWDBManager::database(description.GetAccessLibrary(),
            description.GetServerName(), user.GetUserName(), user.GetPassword(),
            description.GetDatabaseName())),
    fConnection(fDatabase.connection())
{
    if (fDatabase.status().errorCode() == RWDBStatus::dbNotFound)
        throw TDbDoesNotExist("Data directory does not exist: " +
fDatabase.status().message());
    else if (fDatabase.status().errorCode() != RWDBStatus::ok)
        throw TDbException(fDatabase.status().message());
```

```
    fDatabase.setErrorHandler(ExceptionHandler);

    UpdateSourceDescriptions();
}


//  Return the data directory's description.
const TDbDirectoryDescription& TDbDirectory::GetDescription() const
{
    return fDescription;
}


//  Return whether the specified data source is available.
bool TDbDirectory::HasSource(const string& name) const
{
    return fSourceDescriptions.IsMember(name);
}


//  Get the specified data source's description.  The exception TDbDoesNotExist
//  is thrown if the specified data source does not exist.
const TDbSourceDescription& TDbDirectory::GetSource(const string& name) const
{
    for (SourceDescriptionSetIterator i(fSourceDescriptions); !i.IsDone();
            i.Next())
        if (i.CurrentItem()->GetName() == name)
            return *i.CurrentItem();
    throw TDbDoesNotExist("Data source does not exist.");
}


//  Get the descriptions of the available data sources.
const TDbDirectory::SourceDescriptionSet& TDbDirectory::GetSources() const
{
    return fSourceDescriptions;
}


//  Create a new data source in the data directory.  The exception
//  TDbAlreadyExists is thrown if the data source already exists.
void TDbDirectory::CreateSource(const TDbSourceDescription& description)
{
    if (HasSource(description.GetName()))
        throw TDbAlreadyExists("Data source already exists.");

    try {
        RWDBTable table = GetDatabase().table("SOURCE_INDEX");
        RWDBInserter inserter = table.inserter();
        inserter << description.GetName() << description.GetComment();
        inserter.execute(GetConnection());
    } catch(RWDBStatus status) {
        throw TDbException(status.message());
    }

    UpdateSourceDescriptions();
}


//  Delete the specified data source from the data directory.  The exception
//  TDbDoesNotExist is thrown if the source does not exist.
void TDbDirectory::DeleteSource(const TDbSourceDescription& description)
{
    if (!HasSource(description.GetName()))
        throw TDbDoesNotExist("Data source does not exist.");

    try {
        RWDBTable table = GetDatabase().table("SOURCE_INDEX");
        RWDBDeleter deleter = table.deleter();
        deleter.where(table["NAME"] == description.GetName());
        deleter.execute(GetConnection());
    } catch(RWDBStatus status) {
        throw TDbException(status.message());
    }

    UpdateSourceDescriptions();
```

```
}


//  Perform an SQL query.  The exception TDbQueryFailed is thrown if the query
//  fails.  Note that this exposes the relational database model.
void TDbDirectory::Query(const string& sql)
{
    try {
        fDatabase.executeSql(sql, GetConnection());
    } catch(RWDBStatus status) {
        throw TDbQueryFailed(status.message());
    }
}


//  Return the database instance for the data directory.  Note that this exposes
//  the implementation.
RWDBDatabase& TDbDirectory::GetDatabase()
{
    return fDatabase;
}


//  Return the connection instance for the data directory.  Note that this
//  exposes the implementation.
RWDBConnection& TDbDirectory::GetConnection()
{
    return fConnection;
}


//  Update the source descriptions.
void TDbDirectory::UpdateSourceDescriptions()
{
    fSourceDescriptions.Clear();
    try {
        RWDBTable table = GetDatabase().table("SOURCE_INDEX");
        RWDBReader reader = table.reader(GetConnection());
        while (reader()) {
            RWCString name;
            RWCString comment;
            reader >> name >> comment;
            fSourceDescriptions.Add(TDbSourceDescription(name, comment));
        }
    } catch(RWDBStatus status) {
        throw TDbException(status.message());
    }
}
```

## *C. TDbDirectoryDescription Class*

### 1. DirectoryDescription.h

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: DirectoryDescription.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


#ifndef TRANSIMS_DB_DIRECTORYDESCRIPTION
#define TRANSIMS_DB_DIRECTORYDESCRIPTION


//  Include TRANSIMS header files.
#include <GBL/Globals.h>


//  A data directory description uniquely specifies a data directory.
class TDbDirectoryDescription
```

```
{
  public:

    //  Construct a directory description.
    TDbDirectoryDescription(const string& name, const string& comment = "",
            const string& databaseName = "transims", const string& server =
            string(fgServerName), const string& accessLibrary =
            string(fgAccessLibrary));

    //  Construct a copy of the specified directory description.
//  TDbDirectoryDescription(const TDbDirectoryDescription& description);

    //  Make the directory description a copy of the specified directory
    //  description.
//  TDbDirectoryDescription& operator=(const TDbDirectoryDescription&
//          description);

    //  Return the name of the directory.
    const string& GetName() const;

    //  Return the comment for the directory.
    const string& GetComment() const;

    //  Return the database name for the directory.
    const string& GetDatabaseName() const;

    //  Return the name of the server.
    const string& GetServerName() const;

    //  Return the name of the access library.
    const string& GetAccessLibrary() const;

    //  Return whether the directory description has the same name as the
    //  specified directory description.
    bool operator==(const TDbDirectoryDescription& description) const;

    //  Return whether the directory description has a name different from the
    //  specified directory description.
    bool operator!=(const TDbDirectoryDescription& description) const;

    //  Return the name of the default server.
    static const string GetDefaultServerName();

    //  Set the name of the default server to the specified name.
    static void SetDefaultServerName(const string& serverName);

    //  Return the name of the default access library.
    static const string GetDefaultAccessLibrary();

    //  Set the name of the default access library to the specified name.
    static void SetDefaultAccessLibrary(const string& accessLibrary);

  private:

    //  A directory has a name.
    string fName;

    //  A directory has a comment.
    string fComment;

    //  A directory has a database name.
    string fDatabaseName;

    //  A server is required for using a directory.
    string fServerName;

    //  An access library is required for using a directory.
    string fAccessLibrary;

    //  There is a default server.
    static char fgServerName[1000];

    //  There is a default access library.
    static char fgAccessLibrary[1000];
};
```

```
#endif // TRANSIMS_DB_DIRECTORYDESCRIPTION
```

## 2. DirectoryDescription.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: DirectoryDescription.C,v $
//  $Revision: 2.1 $
//  $Date: 1995/10/10 22:29:56 $
//  $State: Exp $
//  $Author: stretz $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include TRANSIMS header files.
#include <DBS/DirectoryDescription.h>


//  Initialize the default server name.
char TDbDirectoryDescription::fgServerName[1000] = "";


//  Initialize the default access library.
#ifdef SUN4
// SunOS 4.x
char TDbDirectoryDescription::fgAccessLibrary[1000] = "/usr/rogue/lib/librwora.so";
#else
// SunOS5.x and Others
char TDbDirectoryDescription::fgAccessLibrary[1000] = "librwora.so";
#endif


//  Construct a directory description for the specified database name.
TDbDirectoryDescription::TDbDirectoryDescription(const string& name, const
        string& comment, const string& databaseName, const string& serverName,
        const string& accessLibrary)
  : fName(name),
    fComment(comment),
    fDatabaseName(databaseName),
    fServerName(serverName),
    fAccessLibrary(accessLibrary)
{
}


//  Return the name of the directory.
const string& TDbDirectoryDescription::GetName() const
{
    return fName;
}


//  Return the comment for the directory.
const string& TDbDirectoryDescription::GetComment() const
{
    return fComment;
}


//  Return the database name for the directory.
const string& TDbDirectoryDescription::GetDatabaseName() const
{
    return fDatabaseName;
}


//  Return the name of the server.
const string& TDbDirectoryDescription::GetServerName() const
{
    return fServerName;
}
```

```
//  Return the name of the access library.
const string& TDbDirectoryDescription::GetAccessLibrary() const
{
    return fAccessLibrary;
}


//  Return whether the directory description has the same name as the specified
//  directory description.
bool TDbDirectoryDescription::operator==(const TDbDirectoryDescription&
        description) const
{
    return GetName() == description.GetName();
}


//  Return whether the directory description has a name different from the
//  specified directory description.
bool TDbDirectoryDescription::operator!=(const TDbDirectoryDescription&
        description) const
{
    return GetName() != description.GetName();
}


//  Return the name of the default server.
const string TDbDirectoryDescription::GetDefaultServerName()
{
    return fgServerName;
}


//  Set the name of the default server to the specified name.
void TDbDirectoryDescription::SetDefaultServerName(const string& serverName)
{
    strcpy(fgServerName, serverName);
}


//  Return the name of the default access library.
const string TDbDirectoryDescription::GetDefaultAccessLibrary()
{
    return fgAccessLibrary;
}


//  Set the name of the default access library to the specified name.
void TDbDirectoryDescription::SetDefaultAccessLibrary(const string&
        accessLibrary)
{
    strcpy(fgAccessLibrary, accessLibrary);
}
```

## *D. TDbException Class*

### 1. Exception.h

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: Exception.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


#ifndef TRANSIMS_DB_EXCEPTION
#define TRANSIMS_DB_EXCEPTION


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
```

```
//  A database exception signals the failure of a member function.
class TDbException
{
  public:

    //  Construct an exception with the specified message text.
    TDbException(const string& message = "Database error.");

    //  Construct a copy of the specified exception.
//  TDbException(const TDbException& exception);

    //  Make the exception a copy of the specified exception.
//  TDbException& operator=(const TDbException& exception);

    //  Return the message text for the exception.
    const string& GetMessage() const;

  private:

    //  Each exception has a message.
    string fMessage;
};


//  This exception is thrown when an attempt is made to create something that
//  already exists.
class TDbAlreadyExists
  : public TDbException
{
  public:

    //  Construct an exception with the specified message text.
    TDbAlreadyExists(const string& message = "Already exists.");

    //  Construct a copy of the specified exception.
//  TDbAlreadyExists(const TDbAlreadyExists& exception);

    //  Make the exception a copy of the specified exception.
//  TDbAlreadyExists& operator=(const TDbAlreadyExists& exception);
};


//  This exception is thrown when an attempt to read data fails.
class TDbCannotRead
  : public TDbException
{
  public:

    //  Construct an exception with the specified message text.
    TDbCannotRead(const string& message = "Cannot read.");

    //  Construct a copy of the specified exception.
//  TDbCannotRead(const TDbCannotRead& exception);

    //  Make the exception a copy of the specified exception.
//  TDbCannotRead& operator=(const TDbCannotRead& exception);
};


//  This exception is thrown when an attempt to write data fails.
class TDbCannotWrite
  : public TDbException
{
  public:

    //  Construct an exception with the specified message text.
    TDbCannotWrite(const string& message = "Cannot write.");

    //  Construct a copy of the specified exception.
//  TDbCannotWrite(const TDbCannotWrite& exception);

    //  Make the exception a copy of the specified exception.
//  TDbCannotWrite& operator=(const TDbCannotWrite& exception);
};
```

```
//  This exception is thrown when an attempt is made to access something that
//  does not exist.
class TDbDoesNotExist
  : public TDbException
{
  public:

    //  Construct an exception with the specified message text.
    TDbDoesNotExist(const string& message = "Does not exist.");

    //  Construct a copy of the specified exception.
//  TDbDoesNotExist(const TDbDoesNotExist& exception);

    //  Make the exception a copy of the specified exception.
//  TDbDoesNotExist& operator=(const TDbDoesNotExist& exception);
};


//  This exception is thrown when a query fails.
class TDbQueryFailed
  : public TDbException
{
  public:

    //  Construct an exception with the specified message text.
    TDbQueryFailed(const string& message = "Query failed.");

    //  Construct a copy of the specified exception.
//  TDbQueryFailed(const TDbQueryFailed& exception);

    //  Make the exception a copy of the specified exception.
//  TDbQueryFailed& operator=(const TDbQueryFailed& exception);
};


//  This exception is thrown when table creation fails.
class TDbCreationFailed
  : public TDbException
{
  public:

    //  Construct an exception with the specified message text.
    TDbCreationFailed(const string& message = "Creation failed.");

    //  Construct a copy of the specified exception.
//  TDbCreationFailed(const TDbCreationFailed& exception);

    //  Make the exception a copy of the specified exception.
//  TDbCreationFailed& operator=(const TDbCreationFailed& exception);
};


#endif // TRANSIMS_DB_EXCEPTION
```

## 2. Exception.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: Exception.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include TRANSIMS header files.
#include <DBS/Exception.h>


//  Construct an exception with the specified message text.
TDbException::TDbException(const string& message)
```

```
  : fMessage(message)
{
}


//  Return the message text for the exception.
const string& TDbException::GetMessage() const
{
    return fMessage;
}


//  Construct an "already exists" exception with the specified message text.
TDbAlreadyExists::TDbAlreadyExists(const string& message)
  : TDbException(message)
{
}


//  Construct a "cannot read" exception with the specified message text.
TDbCannotRead::TDbCannotRead(const string& message)
  : TDbException(message)
{
}


//  Construct a "cannot write" exception with the specified message text.
TDbCannotWrite::TDbCannotWrite(const string& message)
  : TDbException(message)
{
}


//  Construct a "does not exist" exception with the specified message text.
TDbDoesNotExist::TDbDoesNotExist(const string& message)
  : TDbException(message)
{
}


//  Construct a "query failed" exception with the specified message text.
TDbQueryFailed::TDbQueryFailed(const string& message)
  : TDbException(message)
{
}


//  Construct a "creation failed" exception with the specified message text.
TDbCreationFailed::TDbCreationFailed(const string& message)
  : TDbException(message)
{
}
```

## *E. TDbField Class*

### 1. Field.h

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: Field.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


#ifndef TRANSIMS_DB_FIELD
#define TRANSIMS_DB_FIELD


//  Include standard C header files.
#include <stddef.h>
```

```
//  Include TRANSIMS header files.
#include <GBL/Globals.h>


//  A field is a column in a data table.

class TDbField
{
  public:

    //  Field types.
    enum EType {kUnknown, kChar, kUnsignedChar, kShort, kUnsignedShort, kInt,
            kUnsignedInt, kLong, kUnsignedLong, kFloat, kDouble, kString};

    //  Create a field with the specified name and type.
    TDbField(const string& name, EType type = kUnknown, size_t size = 0);

    //  Construct a copy of the specified field.
//  TDbField(const TDbField& field);

    //  Make the field a copy of the specified field.
//  TDbField& operator=(const TDbField& field);

    //  Return the name of the field.
    const string& GetName() const;

    //  Return the type of the field.
    EType GetType() const;

    //  Return the size of the field, if any.
    size_t GetSize() const;

    //  Return whether the field has the same name as the specified field.
    bool operator==(const TDbField& field) const;

    //  Return whether the field has a name different from the specified field.
    bool operator!=(const TDbField& field) const;

  private:

    //  A field has a name.
    string fName;

    //  A field has a type.
    EType fType;

    //  A field may have a size (if it is a string).
    size_t fSize;
};


#endif // TRANSIMS_DB_FIELD
```

## 2. Field.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: Field.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include TRANSIMS header files.
#include <DBS/Field.h>


//  Create a field with the specified name and type.
TDbField::TDbField(const string& name, EType type, size_t size)
  : fName(name),
```

```
    fType(type),
    fSize(size)
{
}


//  Return the name of the field.
const string& TDbField::GetName() const
{
    return fName;
}


//  Return the type of the field.
TDbField::EType TDbField::GetType() const
{
    return fType;
}


//  Return the size of the field, if any.
size_t TDbField::GetSize() const
{
    return fSize;
}


//  Return whether the field has the same name as the specified field.
bool TDbField::operator==(const TDbField& field) const
{
    return GetName() == field.GetName();
}


//  Return whether the field has a name different from the specified field.
bool TDbField::operator!=(const TDbField& field) const
{
    return GetName() != field.GetName();
}
```

## *F. TDbInserter Class*

### 1. Inserter.h

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: Inserter.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


#ifndef TRANSIMS_DB_INSERTER
#define TRANSIMS_DB_INSERTER


//  Include Booch Components header files.
#include <BCStoreM.h>
#include <BCMapU.h>


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <DBS/Table.h>
#include <DBS/TableDescription.h>
#include <DBS/Field.h>



//  This class is used for inserting data into a table.
class TDbInserter
```

```
{
  public:

    //  Open an inserter for the specified table.
    TDbInserter(TDbTable& table);

    //  Destroy an inserter.
    ~TDbInserter();

    //  Make a copy of the given inserter.
//  TDbInserter(const TDbInserter& inserter);

    //  Make the inserter a copy of the given inserter.
//  TDbInserter& operator=(const TDbInserter& inserter);

    //  Return the inserter's data table.
    TDbTable& GetTable();

    //  Insert the current record into the table.  Note that the data may
    //  not be immediately written to disk.  The exception TDbCannotWrite is
    //  thrown if the data cannot be written.
    void Insert();

    //  Flush any changes in the table not yet written to disk.
    void Flush();

    //  Set the value of the specified field.  The exception TDbDoesNotExist is
    //  thrown if the field is not in the table.
    void SetField(const TDbField& field, char value);
    void SetField(const TDbField& field, unsigned char value);
    void SetField(const TDbField& field, short value);
    void SetField(const TDbField& field, unsigned short value);
    void SetField(const TDbField& field, int value);
    void SetField(const TDbField& field, unsigned int value);
    void SetField(const TDbField& field, long value);
    void SetField(const TDbField& field, unsigned long value);
    void SetField(const TDbField& field, float value);
    void SetField(const TDbField& field, double value);
    void SetField(const TDbField& field, const string& value);
    void SetField(const TDbField& field);

  private:

    //  Type definitions.
    typedef BC_TUnboundedMap<TDbField, RWDBValue*, 30U, BC_CManaged> FieldMap;
    typedef BC_TMapActiveIterator<TDbField, RWDBValue*> FieldMapIterator;

    //  Set the value of the specified field.  The exception TDbDoesNotExist is
    //  thrown if the field is not in the table.
    void SetField(const TDbField& field, RWDBValue* const value);

    //  Each table inserter is associated with a table.
    TDbTable* fTable;

    //  Each table inserter has a database table.
    RWDBTable fDbTable;

    //  Each table inserter has its own database connection.
    RWDBConnection fConnection;

    //  Each table insert has a phrase book for the database.
    RWDBPhraseBook fPhraseBook;

    //  Each table insert has a field map holding the current values of the
    //  fields.
    FieldMap fFields;
};


#endif // TRANSIMS_DB_INSERTER
```

## 2. Inserter.C

```
//  Project: TRANSIMS
//  Subsystem: Database
```

```
//  $RCSfile: Inserter.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include DBTools.h++ header files.
#include <rw/db/result.h>


//  Include TRANSIMS header files.
#include <DBS/Exception.h>
#include <DBS/Inserter.h>



//  Define the hash function for fields.
static BC_Index FieldHashValue(const TDbField& field)
{
    return HashValue(field.GetName());
}


//  Open an inserter for the specified table.
TDbInserter::TDbInserter(TDbTable& table)
  : fTable(&table),
    fDbTable(table.GetDatabase().table(table.GetDescription().GetTableName())),
    fConnection(fTable->GetDatabase().connection()),
    fPhraseBook(fTable->GetDatabase().phraseBook()),
    fFields(FieldHashValue)
{
    try {
        fConnection.autoCommit(FALSE);
        fConnection.beginTransaction();
    } catch(RWDBStatus status) {
        throw TDbException(status.message());
    }
}


//  Destroy an inserter.
TDbInserter::~TDbInserter()
{
    for (FieldMapIterator i(fFields); !i.IsDone(); i.Next())
        delete *i.CurrentValue();

    try {
        fConnection.commitTransaction();
    } catch(RWDBStatus status) {
        throw TDbException(status.message());
    }
}


//  Return the inserter's data table.
TDbTable& TDbInserter::GetTable()
{
    return *fTable;
}


//  Insert the current record into the table.  Note that the data may not be
//  immediately written to disk.  The exception TDbCannotWrite is thrown if the
//  data cannot be written.
void TDbInserter::Insert()
{
    string start = "INSERT INTO " + fDbTable.name();
    string fields = "  (";
    string values = "  VALUES(";
    bool first = TRUE;
    for (FieldMapIterator i(fFields); !i.IsDone(); i.Next()) {
        const TDbField& field = *i.CurrentItem();
        const RWDBValue& value = **i.CurrentValue();
```

```
        if (first) {
            first = FALSE;
            fields += "\"" + field.GetName() + "\"";
            values += value.asString(fPhraseBook);
        } else {
            fields += ", \"" + field.GetName() + "\"";
            values += ", " + value.asString(fPhraseBook);
        }
    }
    fields += ")";
    values += ")";

    try {
        fTable->GetDatabase().executeSql(start + "\n" + fields + "\n" + values,
                fConnection);
    } catch(RWDBStatus status) {
        throw TDbCannotWrite(status.message());
    }
}


//  Flush any changes in the table not yet written to disk.
void TDbInserter::Flush()
{
    try {
        fConnection.commitTransaction();
        fConnection.beginTransaction();
    } catch(RWDBStatus status) {
        throw TDbException(status.message());
    }
}


//  Set the value of the specified field.  The exception TDbDoesNotExist is
//  thrown if the field is not in the table.
void TDbInserter::SetField(const TDbField& field, RWDBValue* const value)
{
    if (!fTable->GetFields().IsMember(field))
        throw TDbDoesNotExist("Field not found.");

    if (fFields.IsBound(field)) {
        delete *fFields.ValueOf(field);
        fFields.Rebind(field, value);
    } else
        fFields.Bind(field, value);
}

void TDbInserter::SetField(const TDbField& field, char value)
{
    SetField(field, new RWDBValue(value));
}

void TDbInserter::SetField(const TDbField& field, unsigned char value)
{
    SetField(field, new RWDBValue(value));
}

void TDbInserter::SetField(const TDbField& field, short value)
{
    SetField(field, new RWDBValue(value));
}

void TDbInserter::SetField(const TDbField& field, unsigned short value)
{
    SetField(field, new RWDBValue(value));
}

void TDbInserter::SetField(const TDbField& field, int value)
{
    SetField(field, new RWDBValue(value));
}

void TDbInserter::SetField(const TDbField& field, unsigned int value)
{
    SetField(field, new RWDBValue(value));
}
```

```
void TDbInserter::SetField(const TDbField& field, long value)
{
    SetField(field, new RWDBValue(value));
}

void TDbInserter::SetField(const TDbField& field, unsigned long value)
{
    SetField(field, new RWDBValue(value));
}

void TDbInserter::SetField(const TDbField& field, float value)
{
    SetField(field, new RWDBValue(value));
}

void TDbInserter::SetField(const TDbField& field, double value)
{
    SetField(field, new RWDBValue(value));
}

void TDbInserter::SetField(const TDbField& field, const string& value)
{
    SetField(field, new RWDBValue(value));
}

void TDbInserter::SetField(const TDbField& field)
{
    if (fFields.IsBound(field))
        delete *fFields.ValueOf(field);
    fFields.Unbind(field);
}
```

## *G. TDbSortedAccessor Class*

### 1. SortedAccessor.h

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: SortedAccessor.h,v $
//  $Revision: 1.1 $
//  $Date: 1996/02/20 22:18:17 $
//  $State: Exp $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


#ifndef TRANSIMS_DB_SORTEDACCESSOR
#define TRANSIMS_DB_SORTEDACCESSOR


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <DBS/Accessor.h>


//  A sorted table accessor provides facilities for navigating and editing a
//  data table's records.
class TDbSortedAccessor
  : public TDbAccessor
{
  public:

    //  Open an accessor for the specified table, sorted by the specified
    //  fields.
    TDbSortedAccessor(TDbTable& table, const TDbSource::FieldCollection&
            fields);

    //  Create a copy of the specified accessor.
//  TDbSortedAccessor(const TDbSortedAccessor& accessor);

    //  Make the accessor a copy of the specified accessor.
//  TDbSortedAccessor& operator=(const TDbSortedAccessor& accessor);
```

```
    //  Position the accessor on the first record.  The exception TDbCannotRead
    //  is thrown if the record cannot be read.  Note that calling this
    //  function is an expensive operation.
    void GotoFirst();

    //  Return the sort fields for the table.
    const TDbSource::FieldCollection& GetSortFields() const;

    //  Return the reader instance for the accessor.  Note that this exposes the
    //  implementation.  The reader is not necessarily sorted.
//  RWDBReader& GetReader();

 private:

    //  Each sorted table accessor has the fields sorted in a particular
    //  order.
    TDbSource::FieldCollection fSortFields;
};


#endif // TRANSIMS_DB_SORTEDACCESSOR
```

## 2. SortedAccessor.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: SortedAccessor.C,v $
//  $Revision: 1.1 $
//  $Date: 1996/02/20 22:18:17 $
//  $State: Exp $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include TRANSIMS header files.
#include <DBS/Exception.h>
#include <DBS/SortedAccessor.h>


//  Open an accessor for the specified table, sorted by the specified
//  fields.
TDbSortedAccessor::TDbSortedAccessor(TDbTable& table, const
            TDbSource::FieldCollection& fields)
  : TDbAccessor( table),
    fSortFields(fields)
{
}


//  Position the accessor on the first record.  The exception TDbCannotRead is
//  thrown if the record cannot be read.  Note that calling this function is
//  an expensive operation.
void TDbSortedAccessor::GotoFirst()
{
    try {
        RWDBSelector selector = fTable->GetDatabase().selector();
        for (TDbTable::FieldSetIterator i(fTable->GetFields()); !i.IsDone();
                i.Next())
            selector << fDbTable[(*i.CurrentItem()).GetName()];
        for (TDbSource::FieldCollectionIterator j(fSortFields); !j.IsDone();
                j.Next())
            selector.orderBy(fDbTable[(*j.CurrentItem()).GetName()]);
        //ISSUE(bwb): The following statement has failed for some tables.
        //The problem appears to be in DBtools.h++.
        fReader = selector.reader();
        fIsAtRecord = fReader() != 0;
    } catch (RWDBStatus status) {
        throw TDbCannotRead(status.message());
    }
}


//  Return the sort fields for the table.
```

```
const TDbSource::FieldCollection& TDbSortedAccessor::GetSortFields() const
{
    return fSortFields;
}
```

## *H. TDbSource Class*

### 1. Source.h

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: Source.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


#ifndef TRANSIMS_DB_SOURCE
#define TRANSIMS_DB_SOURCE


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <DBS/Directory.h>
#include <DBS/SourceDescription.h>
#include <DBS/TableDescription.h>
#include <DBS/Field.h>


//  Include Booch Components header files.
#include <BCStoreM.h>
#include <BCSetU.h>
#include <BCCollU.h>


//  Include DBtools.h++ header files.
#include <rw/db/dbase.h>
#include <rw/db/dbmgr.h>
#include <rw/db/connect.h>


//  A data source organizes the different data tables (versions) that may exist
//  for a type of data.
class TDbSource
{
  public:

    //  Type definitions.
    typedef BC_TUnboundedSet<const TDbTableDescription, 25U, BC_CManaged>
            TableDescriptionSet;
    typedef BC_TSetActiveIterator<const TDbTableDescription>
            TableDescriptionSetIterator;
    typedef BC_TUnboundedCollection<TDbField, BC_CManaged> FieldCollection;
    typedef BC_TCollectionActiveIterator<TDbField> FieldCollectionIterator;

    //  Open the specified data source.  The exception TDbDoesNotExist is
    //  thrown if the specified source does not exist.
    TDbSource(TDbDirectory& directory, const TDbSourceDescription& description);

    //  Construct a copy of the specified source.
//  TDbSource(const TDbSource& source);

    //  Make the source a copy of the specified source.
//  TDbSource& operator=(const TDbSource& source);

    //  Return the data source's description.
    const TDbSourceDescription& GetDescription() const;

    //  Return whether the specified data table (version) of the data source
    //  exists.
    bool HasTable(const string& name) const;
```

```
    //  Return the description for the specified data table (version) of the
    //  data source.  The exception TDbDoesNotExist is thrown if the specified
    //  source does not exist.
    const TDbTableDescription& GetTable(const string& name) const;

    //  Return the set of descriptions of the available data tables (versions)
    //  of the data source.
    const TableDescriptionSet& GetTables() const;

    //  Create a new data table (version) of the data source with the specified
    //  description and using the specified SQL statement.  The exception
    //  TDbAlreadyExists is thrown if the specified table already exists.  The
    //  exception TDbCreationFailed is thrown if the table cannot be created.
    //  Note that this exposes the relational database model.
    void CreateTable(const TDbTableDescription& description, const string& sql);

    //  Create a new data table (version) of the data source with the specified
    //  description and with the specified fields and primary index.  The
    //  exception TDbAlreadyExists is thrown if the specified table already
    //  exists.  The exception TDbCreationFailed is thrown if the table cannot
    //  be created.
    void CreateTable(const TDbTableDescription& description, const
        FieldCollection& fields, const FieldCollection& index);

    //  Delete the specified data table from the data source.  The exception
    //  TDbDoesNotExist is thrown if the table does not exist.
    void DeleteTable(const TDbTableDescription& description);

    //  Return the directory for the source.
    TDbDirectory& GetDirectory();

    //  Return the database instance for the data source.  Note that this
    //  exposes the implementation.
    RWDBDatabase& GetDatabase();

    //  Return the connection instance for the data source.  Note that this
    //  exposes the implementation.
    RWDBConnection& GetConnection();

  private:

    //  Update the table descriptions.
    void UpdateTableDescriptions();

    //  Each data source is associated with a data directory.
    TDbDirectory* fDirectory;

    //  Each data source has a description.
    TDbSourceDescription fDescription;

    //  Each data source has several tables, identified by their descriptions.
    TableDescriptionSet fTableDescriptions;
};


#endif // TRANSIMS_DB_SOURCE
```

## 2. Source.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: Source.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include TRANSIMS header files.
#include <DBS/Exception.h>
#include <DBS/Source.h>
```

```
//  Include DBtools.h++ header files.
#include <rw/db/table.h>
#include <rw/db/select.h>
#include <rw/db/reader.h>
#include <rw/db/inserter.h>
#include <rw/db/deleter.h>
#include <rw/db/result.h>


//  Define the hash function for table descriptions.
static BC_Index TableDescriptionHashValue(const TDbTableDescription&
        description)
{
    return HashValue(description.GetName());
}


//  Open the specified data source.  The exception TDbDoesNotExist is thrown if
//  the specified source does not exist.
TDbSource::TDbSource(TDbDirectory& directory, const TDbSourceDescription&
        description)
  : fDirectory(&directory),
    fDescription(description),
    fTableDescriptions(TableDescriptionHashValue)
{
    if (!directory.GetSources().IsMember(description))
        throw TDbDoesNotExist("Data source does not exist.");

    UpdateTableDescriptions();
}


//  Return the data source's description.
const TDbSourceDescription& TDbSource::GetDescription() const
{
    return fDescription;
}


//  Return whether the specified data table (version) of the data source exists.
bool TDbSource::HasTable(const string& name) const
{
    return fTableDescriptions.IsMember(name);
}


//  Return the description for the specified data table (version) of the data
//  source.  The exception TDbDoesNotExist is thrown if the specified source
//  does not exist.
const TDbTableDescription& TDbSource::GetTable(const string& name) const
{
    for (TableDescriptionSetIterator i(fTableDescriptions); !i.IsDone();
            i.Next())
        if (i.CurrentItem()->GetName() == name)
            return *i.CurrentItem();
//ISSUE(bwb): The following line causes a fault in RWString::~RWString.
//  throw TDbDoesNotExist("Data table does not exist.");
    static const string message = "Data table does not exist.";
    throw TDbDoesNotExist(message);
}


//  Return the set of descriptions of the available data tables (versions) of
//  the data source.
const TDbSource::TableDescriptionSet& TDbSource::GetTables() const
{
    return fTableDescriptions;
}


//  Create a new data table (version) of the data source with the specified
//  description and using the specified SQL statement.  The exception
//  TDbAlreadyExists is thrown if the specified table already exists.  The
//  exception TDbCreationFailed is thrown if the table cannot be created.  Note
//  that this exposes the relational database model.
```

```
void TDbSource::CreateTable(const TDbTableDescription& description, const
        string& sql)
{
    if (HasTable(description.GetName()))
        throw TDbAlreadyExists("Data table already exists.");

    RWDBResult result;
    try {
        result = GetDatabase().executeSql(sql, GetConnection());
        RWDBTable table = GetDatabase().table("TABLE_INDEX");
        RWDBInserter inserter = table.inserter();
        inserter << description.GetTableName() << fDescription.GetName() <<
                description.GetName() << description.GetComment();
        inserter.execute(GetConnection());
    } catch(RWDBStatus status) {
        try {
            for (RWDBTable table = result.table(); table.isValid(); table =
                    result.table())
                table.drop();
        } catch (RWDBStatus) {
        }
        throw TDbCreationFailed(status.message());
    }

    UpdateTableDescriptions();
}


//  Create a new data table (version) of the data source with the specified
//  description and with the specified fields and primary index.  The
//  exception TDbAlreadyExists is thrown if the specified table already
//  exists.  The exception TDbCreationFailed is thrown if the table cannot
//  be created.
void TDbSource::CreateTable(const TDbTableDescription& description, const
        TDbSource::FieldCollection& fields, const TDbSource::FieldCollection&
        index)
{
    string sql = "CREATE TABLE " + description.GetTableName() + " (";

    bool first = TRUE;
    for (FieldCollectionIterator i(fields); !i.IsDone(); i.Next()) {
        const TDbField& field = *i.CurrentItem();
        if (first) {
            first = FALSE;
            sql += "\n";
        } else
            sql += ",\n";
        sql += "  \"" + field.GetName() + "\" ";
        switch (field.GetType()) {
            case TDbField::kChar:
            case TDbField::kUnsignedChar:
                sql += "NUMBER(3)";
                break;
            case TDbField::kShort:
            case TDbField::kUnsignedShort:
                sql += "NUMBER(5)";
                break;
            case TDbField::kInt:
            case TDbField::kUnsignedInt:
            case TDbField::kLong:
            case TDbField::kUnsignedLong:
                sql += "NUMBER(10)";
                break;
            case TDbField::kFloat:
                sql += "FLOAT(63)";
                break;
            case TDbField::kDouble:
                sql += "FLOAT(126)";
                break;
            case TDbField::kString:
                {
                    char temp[10];
                    sprintf(temp, "%u", field.GetSize());
                    sql += "VARCHAR(" + string(temp) + ")";
                }
                break;
```

```
            case TDbField::kUnknown:
                throw TDbCreationFailed("Invalid field type.");
                break;
        }
    }

    first = TRUE;
    for (FieldCollectionIterator j(index); !j.IsDone(); j.Next()) {
        if (first) {
            first = FALSE;
            sql += ",\n  PRIMARY KEY (\"" + (*j.CurrentItem()).GetName();
        } else
            sql += "\", \"" + (*j.CurrentItem()).GetName();
    }
    if (!first)
        sql += "\")\n";

    sql += ")";

    CreateTable(description, sql);
}


//  Delete the specified data table from the data source.  The exception
//  TDbDoesNotExist is thrown if the table does not exist.
void TDbSource::DeleteTable(const TDbTableDescription& description)
{
    if (!HasTable(description.GetName()))
        throw TDbDoesNotExist("Data table does not exist.");

    try {
        RWDBTable table = GetDatabase().table("TABLE_INDEX");
        RWDBDeleter deleter = table.deleter();
        deleter.where(table["NAME"] == description.GetName());
        deleter.execute(GetConnection());
        GetDatabase().table(GetTable(description.GetName()).GetTableName()).
                drop();
    } catch(RWDBStatus status) {
        throw TDbException(status.message());
    }

    UpdateTableDescriptions();
}


//  Return the directory for the source.
TDbDirectory& TDbSource::GetDirectory()
{
    return *fDirectory;
}


//  Return the database instance for the data source.  Note that this exposes
//  the implementation.
RWDBDatabase& TDbSource::GetDatabase()
{
    return fDirectory->GetDatabase();
}


//  Return the connection instance for the data source.  Note that this exposes
//  the implementation.
RWDBConnection& TDbSource::GetConnection()
{
    return fDirectory->GetConnection();
}


//  Update the table descriptions.
void TDbSource::UpdateTableDescriptions()
{
    fTableDescriptions.Clear();
    try {
        RWDBTable table = GetDatabase().table("TABLE_INDEX");
        RWDBSelector selector = GetDatabase().selector();
        selector << table["NAME"] << table["COMMENT_TEXT"] <<
```

```
                table["TABLE_NAME"];
        selector.where(table["SOURCE"] == fDescription.GetName());
        RWDBReader reader = selector.reader(GetConnection());
        while (reader()) {
            RWCString name;
            RWCString comment;
            RWCString tableName;
            reader >> name >> comment >> tableName;
            fTableDescriptions.Add(TDbTableDescription(name, comment,
                    tableName));
        }
    } catch(RWDBStatus status) {
        throw TDbException(status.message());
    }
}
```

## I. TDbSourceDescription Class

### 1. SourceDescription.h

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: SourceDescription.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


#ifndef TRANSIMS_DB_SOURCEDESCRIPTION
#define TRANSIMS_DB_SOURCEDESCRIPTION


//  Include TRANSIMS header files.
#include <GBL/Globals.h>


//  A data source description uniquely specifies a data source within a data
//  directory.
class TDbSourceDescription
{
  public:

    //  Construct a source description.
    TDbSourceDescription(const string& name, const string& comment = "");

    //  Construct a copy of the specified source description.
//  TDbSourceDescription(const TDbSourceDescription& description);

    //  Make the source description a copy of the specified source description.
//  TDbSourceDescription& operator=(const TDbSourceDescription& description);

    //  Return the name of the source.
    const string& GetName() const;

    //  Return the comment for the source.
    const string& GetComment() const;

    //  Return whether the source description has the same name as the specified
    //  source description.
    bool operator==(const TDbSourceDescription& description) const;

    //  Return whether the source description has a name different from the
    //  specified source description.
    bool operator!=(const TDbSourceDescription& description) const;

  private:

    //  A source has a name.
    string fName;

    //  A source has a comment.
```

```
    string fComment;
};


#endif // TRANSIMS_DB_SOURCEDESCRIPTION
```

## 2. SourceDescription.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: SourceDescription.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include TRANSIMS header files.
#include <DBS/SourceDescription.h>


//  Construct a source description.
TDbSourceDescription::TDbSourceDescription(const string& name, const string&
        comment)
  : fName(name),
    fComment(comment)
{
}


//  Return the name of the source.
const string& TDbSourceDescription::GetName() const
{
    return fName;
}


//  Return the comment for the source.
const string& TDbSourceDescription::GetComment() const
{
    return fComment;
}


//  Return whether the source description has the same name as the specified
//  source description.
bool TDbSourceDescription::operator==(const TDbSourceDescription& description)
        const
{
    return GetName() == description.GetName();
}


//  Return whether the source description has a name different from the
//  specified source description.
bool TDbSourceDescription::operator!=(const TDbSourceDescription& description)
        const
{
    return GetName() != description.GetName();
}
```

# *J. TDbTable Class*

## 1. Table.h

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: Table.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
```

```
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


#ifndef TRANSIMS_DB_TABLE
#define TRANSIMS_DB_TABLE


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <DBS/Directory.h>
#include <DBS/Source.h>
#include <DBS/TableDescription.h>
#include <DBS/Field.h>


//  Include Booch Components header files.
#include <BCStoreM.h>
#include <BCSetU.h>


//  Include DBtools.h++ header files.
#include <rw/db/dbase.h>
#include <rw/db/dbmgr.h>
#include <rw/db/connect.h>


//  Each data table contains fields (columns/attributes) and records
//  (rows/tuples).  Each record has a unique key (primary index) and may have
//  secondary keys (indexes).
class TDbTable
{
  public:

    //  Type definitions.
    typedef BC_TUnboundedSet<const TDbField, 25U, BC_CManaged> FieldSet;
    typedef BC_TSetActiveIterator<const TDbField> FieldSetIterator;
    typedef BC_TUnboundedSet<const TDbTableDescription, 25U, BC_CManaged>
            TableDescriptionSet;
    typedef BC_TSetActiveIterator<const TDbTableDescription>
            TableDescriptionSetIterator;

    //  Open the specified table.  The exception TDbDoesNotExist is thrown if
    //  the specified table does not exist.
    TDbTable(TDbSource& source, const TDbTableDescription& description);

    //  Construct a copy of the specified table.
//  TDbTable(const TDbTable& table);

    //  Make the table a copy of the specified table.
//  TDbTable& operator=(const TDbTable& table);

    //  Return the data table's description.
    const TDbTableDescription& GetDescription() const;

    //  Return the set of data tables on which the data table depends.
    const TableDescriptionSet& GetDependentTables() const;

    //  Add a the specified table to the dependents.  The exception
    //  TDbAlreadyExists is thrown if the specified table is already a
    //  dependent.
    void AddDependentTable(const TDbTableDescription& description);

    //  Remove the specified table from the dependents.  The exception
    //  TDbAlreadyExists is thrown if the specified table is already a
    //  dependent.
    void RemoveDependentTable(const TDbTableDescription& description);

    //  Return whether the specified field exists in the data table.
    bool HasField(const string& name) const;

    //  Return the specified field of the data table.  The exception
    //  TDbDoesNotExist is thrown if the specified field does not exist.
    const TDbField& GetField(const string& name) const;
```

```
    //  Return the set of available fields in the data table.
    const FieldSet& GetFields() const;

    //  Return the source for the table.
    TDbSource& GetSource();

    //  Return the directory for the table.
    TDbDirectory& GetDirectory();

    //  Return the database instance for the data table.  Note that this
    //  exposes the implementation.
    RWDBDatabase& GetDatabase();

    //  Return the connection instance for the data table.  Note that this
    //  exposes the implementation.
    RWDBConnection& GetConnection();

  protected:

    //  Update the dependent tables.
    void UpdateDependents();

    //  Update the fields.
    void UpdateFields();

  private:

    //  Each data table is associated with a data source.
    TDbSource* fSource;

    //  Each data table has a description.
    TDbTableDescription fDescription;

    //  Each data table has dependent tables.
    TableDescriptionSet fDependents;

    //  Each data table has several fields.
    FieldSet fFields;
};


#endif // TRANSIMS_DB_TABLE
```

## 2. Table.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: Table.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include TRANSIMS header files.
#include <DBS/Exception.h>
#include <DBS/Table.h>


//  Include DBtools.h++ header files.
#include <rw/db/table.h>
#include <rw/db/select.h>
#include <rw/db/reader.h>
#include <rw/db/inserter.h>
#include <rw/db/deleter.h>
#include <rw/db/result.h>
#include <rw/db/schema.h>


//  Define the hash function for table descriptions.
static BC_Index TableDescriptionHashValue(const TDbTableDescription&
        description)
{
```

```
    return HashValue(description.GetName());
}


//  Define the hash function for fields.
static BC_Index FieldHashValue(const TDbField& field)
{
    return HashValue(field.GetName());
}


//  Open the specified table.  The exception TDbDoesNotExist is thrown if the
//  specified table does not exist.
TDbTable::TDbTable(TDbSource& source, const TDbTableDescription& description)
  : fSource(&source),
    fDescription(description),
    fDependents(TableDescriptionHashValue),
    fFields(FieldHashValue)
{
    UpdateDependents();
    UpdateFields();
}


//  Return the data table's description.
const TDbTableDescription& TDbTable::GetDescription() const
{
    return fDescription;
}


//  Return the set of data tables on which the data table depends.
const TDbTable::TableDescriptionSet& TDbTable::GetDependentTables() const
{
    return fDependents;
}


//  Add a the specified table to the dependents.  The exception
//  TDbAlreadyExists is thrown if the specified table is already a dependent.
void TDbTable::AddDependentTable(const TDbTableDescription& description)
{
    if (fDependents.IsMember(description.GetName()))
        throw TDbAlreadyExists("Data table already dependent.");

    try {
        RWDBTable table = GetDatabase().table("DEPENDENT_TABLES");
        RWDBInserter inserter = table.inserter();
        inserter << fDescription.GetTableName() << description.GetTableName();
        inserter.execute(GetConnection());
    } catch(RWDBStatus status) {
        throw TDbException(status.message());
    }

    UpdateDependents();
}


//  Remove the specified table from the dependents.  The exception
//  TDbAlreadyExists is thrown if the specified table is already a dependent.
void TDbTable::RemoveDependentTable(const TDbTableDescription& description)
{
    if (!fDependents.IsMember(description.GetName()))
        throw TDbDoesNotExist("Data table not dependent.");

    try {
        RWDBTable table = GetDatabase().table("DEPENDENT_TABLES");
        RWDBDeleter deleter = table.deleter();
        deleter.where(table["TABLE_NAME"] == fDescription.GetTableName() &&
                table["DEPENDENT_NAME"] == description.GetTableName());
        deleter.execute(GetConnection());
    } catch(RWDBStatus status) {
        throw TDbException(status.message());
    }

    UpdateDependents();
```

```
}


//  Return whether the specified field exists in the data table.
bool TDbTable::HasField(const string& name) const
{
    return fFields.IsMember(name);
}


//  Return the specified field of the data table.  The exception TDbDoesNotExist
//  is thrown if the specified field does not exist.
const TDbField& TDbTable::GetField(const string& name) const
{
    for (FieldSetIterator i(fFields); !i.IsDone(); i.Next())
        if (i.CurrentItem()->GetName() == name)
            return *i.CurrentItem();
    throw TDbDoesNotExist("Field does not exist.");
}


//  Return the set of available fields in the data table.
const TDbTable::FieldSet& TDbTable::GetFields() const
{
    return fFields;
}


//  Return the source for the table.
TDbSource& TDbTable::GetSource()
{
    return *fSource;
}


//  Return the directory for the table.
TDbDirectory& TDbTable::GetDirectory()
{
    return fSource->GetDirectory();
}


//  Return the database instance for the data table.  Note that this exposes the
//  implementation.
RWDBDatabase& TDbTable::GetDatabase()
{
    return fSource->GetDatabase();
}


//  Return the connection instance for the data table.  Note that this exposes
//  the implementation.
RWDBConnection& TDbTable::GetConnection()
{
    return fSource->GetConnection();
}


//  Update the dependent tables.
void TDbTable::UpdateDependents()
{
    fDependents.Clear();
    try {
        RWDBTable indexTable = GetDatabase().table("TABLE_INDEX");
        RWDBTable dependentTable = GetDatabase().table("DEPENDENT_TABLES");
        RWDBSelector selector = GetDatabase().selector();
        selector << indexTable["NAME"] << indexTable["COMMENT_TEXT"] <<
                indexTable["TABLE_NAME"];
        selector.where(dependentTable["TABLE_NAME"] ==
                fDescription.GetTableName() && dependentTable["DEPENDENT_NAME"]
                == indexTable["TABLE_NAME"]);
        RWDBReader reader = selector.reader(GetConnection());
        while (reader()) {
            RWCString name;
            RWCString comment;
            RWCString tableName;
```

```
            reader >> name >> comment >> tableName;
            fDependents.Add(TDbTableDescription(name, comment, tableName));
        }
    } catch(RWDBStatus status) {
        throw TDbException(status.message());
    }
}


//  Update the fields.
void TDbTable::UpdateFields()
{
    fFields.Clear();
    try {
        RWDBTable table = GetDatabase().table(fDescription.GetTableName());
        table.fetchSchema(GetConnection());
        RWDBSchema schema = table.schema();
        for (size_t i = 0; i < schema.entries(); ++i) {
            RWDBColumn column = schema.column(i);
            TDbField::EType type;
            size_t size = 0;
            switch (column.type()) {
                case RWDBValue::Char:
                    type = TDbField::kChar;
                    break;
                case RWDBValue::UnsignedChar:
                    type = TDbField::kUnsignedChar;
                    break;
                case RWDBValue::Short:
                    type = TDbField::kShort;
                    break;
                case RWDBValue::UnsignedShort:
                    type = TDbField::kShort;
                    break;
                case RWDBValue::Int:
                    type = TDbField::kInt;
                    break;
                case RWDBValue::UnsignedInt:
                    type = TDbField::kUnsignedInt;
                    break;
                case RWDBValue::Long:
                    type = TDbField::kLong;
                    break;
                case RWDBValue::UnsignedLong:
                    type = TDbField::kUnsignedLong;
                    break;
                case RWDBValue::Float:
                    type = TDbField::kFloat;
                    break;
                case RWDBValue::Double:
                    type = TDbField::kDouble;
                    break;
                case RWDBValue::String:
                    type = TDbField::kString;
                    size = column.storageLength();
                    break;
                default:
                    type = TDbField::kUnknown;
                    break;
            }
            fFields.Add(TDbField(column.name(), type, size));
        }
    } catch(RWDBStatus status) {
        throw TDbException(status.message());
    }
}
```

## *K. TDbTableDescription Class*

### 1. TableDescription.h

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: TableDescription.h,v $
```

```
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


#ifndef TRANSIMS_DB_TABLEDESCRIPTION
#define TRANSIMS_DB_TABLEDESCRIPTION


//  Include TRANSIMS header files.
#include <GBL/Globals.h>


//  A data table description uniquely specifies a data table.
class TDbTableDescription
{
  public:

    //  Construct a table description.
    TDbTableDescription(const string& name, const string& comment = "", const
            string& tableName = "");

    //  Construct a copy of the specified table description.
//  TDbTableDescription(const TDbTableDescription& description);

    //  Make the table description a copy of the specified table description.
//  TDbTableDescription& operator=(const TDbTableDescription& description);

    //  Return the name of the source.
    const string& GetName() const;

    //  Return the comment for the source.
    const string& GetComment() const;

    //  Return the table name for the source.
    const string& GetTableName() const;

    //  Return whether the table description has the same name as the specified
    //  table description.
    bool operator==(const TDbTableDescription& description) const;

    //  Return whether the table description has a name different from the
    //  specified table description.
    bool operator!=(const TDbTableDescription& description) const;

  private:

    //  A table has a name.
    string fName;

    //  A table has a comment.
    string fComment;

    //  A table has a table name (in the database).
    string fTableName;
};


#endif // TRANSIMS_DB_TABLEDESCRIPTION
```

## 2. TableDescription.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: TableDescription.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved
```

```
//  Include TRANSIMS header files.
#include <DBS/TableDescription.h>


//  Construct a table description.
TDbTableDescription::TDbTableDescription(const string& name, const string&
        comment, const string& tableName)
  : fName(name),
    fComment(comment),
    fTableName(tableName)
{
}


//  Return the name of the source.
const string& TDbTableDescription::GetName() const
{
    return fName;
}


//  Return the comment for the source.
const string& TDbTableDescription::GetComment() const
{
    return fComment;
}


//  Return the table name for the source.
const string& TDbTableDescription::GetTableName() const
{
    return fTableName;
}


//  Return whether the table description has the same name as the specified
//  table description.
bool TDbTableDescription::operator==(const TDbTableDescription& description)
        const
{
    return GetName() == description.GetName();
}


//  Return whether the table description has a name different from the specified
//  table description.
bool TDbTableDescription::operator!=(const TDbTableDescription& description)
        const
{
    return GetName() != description.GetName();
}
```

## *L. TDbUserInformation Class*

### 1. UserInformation.h

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: UserInformation.h,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


#ifndef TRANSIMS_DB_USERINFORMATION
#define TRANSIMS_DB_USERINFORMATION


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
```

```
//  An instance of user information contains the information necessary for a
//  user to access a database.
class TDbUserInformation
{
  public:

    //  Construct a user information instance.
    TDbUserInformation(const string& userName = string(fgUserName), const
            string& password = string(fgPassword));

    //  Construct a copy of the specified user information.
//  TDbUserInformation(const TDbUserInformation& information);

    //  Make the user information a copy of the specified user information.
//  TDbUserInformation& operator=(const TDbUserInformation& information);

    //  Return the user name.
    const string& GetUserName() const;

    //  Return the password.
    const string& GetPassword() const;

    //  Return the default user name.
    static const string GetDefaultUserName();

    //  Set the default user name.
    static void SetDefaultUserName(const string& userName);

    //  Return the default password.
    static const string GetDefaultPassword();

    //  Set the default password.
    static void SetDefaultPassword(const string& password);

  private:

    //  A user information instance has a user name.
    string fUserName;

    //  A user information instance has a password.
    string fPassword;

    //  There is a default user name.
    static char fgUserName[1000];

    //  There is a default password.
    static char fgPassword[1000];
};


#endif // TRANSIMS_DB_USERINFORMATION
```

## 2. UserInformation.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: UserInformation.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include TRANSIMS header files.
#include <DBS/UserInformation.h>


//  Initialize the default user name.
char TDbUserInformation::fgUserName[1000] = "ioc1";
```

```
//  Initialize the default password.
char TDbUserInformation::fgPassword[1000] = "lanl";


//  Construct a user information instance.
TDbUserInformation::TDbUserInformation(const string& userName, const string&
        password)
  : fUserName(userName),
    fPassword(password)
{
}


//  Return the user name.
const string& TDbUserInformation::GetUserName() const
{
    return fUserName;
}


//  Return the password.
const string& TDbUserInformation::GetPassword() const
{
    return fPassword;
}


//  Return the default user name.
const string TDbUserInformation::GetDefaultUserName()
{
    return fgUserName;
}


//  Set the default user name.
void TDbUserInformation::SetDefaultUserName(const string& userName)
{
    strcpy(fgUserName, userName);
}


//  Return the default password.
const string TDbUserInformation::GetDefaultPassword()
{
    return fgPassword;
}


//  Set the default password.
void TDbUserInformation::SetDefaultPassword(const string& password)
{
    strcpy(fgPassword, password);
}
```

# X. APPENDIX: Test Program

This appendix contains the complete C++ and SQL source code for the database subsystem test program.

## *A. Test.C*

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: Test.C,v $
//  $Revision: 2.1 $
//  $Date: 1996/02/20 22:18:17 $
//  $State: Exp $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include Standard C++ header files.
```

```
#include <iostream.h>


//  Include TRANSIMS header files.
#include <DBS/Exception.h>


//  Function prototypes.
extern bool TestDirectory();
extern bool TestSource();
extern bool TestTable();
extern bool TestField();
extern bool TestAccessor();
extern bool TestSortedAccessor();
extern bool TestInserter();
extern bool TestUserInformation();
extern bool TestDirectoryDescription();
extern bool TestSourceDescription();
extern bool TestTableDescription();
extern void TestCleanup();


//  Main program.
int main(int, char*[])
{
    cout << "NOTE: If this program exits abnormally, it is necessary" << endl;
    cout << "to run the script TestCleanup to clean up the database." << endl;
    cout << endl;

    try {

        bool fail = FALSE;

        cout << "Database Subsystem Tests"
            << " [$Revision: 2.1 $]"
            << endl;

        fail |= !TestDirectory();
        fail |= !TestSource();
        fail |= !TestTable();
        fail |= !TestField();
        fail |= !TestAccessor();
        fail |= !TestSortedAccessor();
        fail |= !TestInserter();
        fail |= !TestUserInformation();
        fail |= !TestDirectoryDescription();
        fail |= !TestSourceDescription();
        fail |= !TestTableDescription();

        cout << (fail ? "  F" : "  No f") << "ailures occured." << endl;

    } catch(const TDbException& exception) {

        cout << endl;
        cout << "Unexpected database subsystem exception occurred--aborting." <<
                endl;
        cout << "(" << exception.GetMessage() << ")" << endl;

    } catch(...) {

        cout << endl;
        cout << "Unexpected unknown exception occurred--aborting." << endl;

    }

    TestCleanup();

    return 0;
}
```

## B. TestAccessor.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: TestAccessor.C,v $
```

```
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include Standard C++ header files.
#include <iostream.h>


//  Include Standard C header files.
#include <math.h>


//  Include TRANSIMS header files.
#include <DBS/Exception.h>
#include <DBS/Directory.h>
#include <DBS/Source.h>
#include <DBS/Table.h>
#include <DBS/Field.h>
#include <DBS/Accessor.h>


//  Test accessor class.
bool TestAccessor()
{
    cout << "  Accessor Class Tests"
        << " [$Revision: 2.0 $]"
        << endl;

    bool anyFail = FALSE;
    bool fail;

    TDbDirectory directory(TDbDirectoryDescription("IOC-1"));
    TDbSource source(directory, directory.GetSource("First Test Source"));
    TDbTable table(source, source.GetTable("Test Table 1A"));
    directory.Query("INSERT INTO TESTTABLE1A VALUES('ABC', 2, 123.321)");
    directory.Query("INSERT INTO TESTTABLE1A VALUES('DEF', -3, 456.654)");
    directory.Query("INSERT INTO TESTTABLE1A VALUES('GHI', 789, 789.987)");
    directory.Query("INSERT INTO TESTTABLE1A VALUES('JKL', 123654, 1456.654)");
    directory.Query("INSERT INTO TESTTABLE1A VALUES('MNO', -100000, 2456.654)");
    TDbAccessor accessor(table);

    cout << "    DB-AC-010: Constructor and DBtools.h++ access.";
    fail = !accessor.GetReader().isValid();
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-AC-020: Table retrieval.";
    fail = &table != &accessor.GetTable();
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-AC-030: Record counting.";
    fail = accessor.GetRecordCount() != 5;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-AC-040: Not at record.";
    fail = accessor.IsAtRecord();
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-AC-050: Get field before start.";
    try {
        string value;
        accessor.GetField(table.GetField("STRINGFIELD"), value);
        fail = TRUE;
    } catch(const TDbCannotRead&) {
        fail = FALSE;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;
```

```
cout << "    DB-AC-060: Go to first record.";
try {
    accessor.GotoFirst();
    fail = FALSE;
} catch(const TDbException&) {
    fail = TRUE;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-AC-070: At record.";
fail = !accessor.IsAtRecord();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-AC-080: Get a string value.";
{
    string value;
    accessor.GetField(table.GetField("STRINGFIELD"), value);
    fail = value != "ABC";
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-AC-090: Get an unsigned char value.";
{
    unsigned char value;
    accessor.GetField(table.GetField("INTEGERFIELD"), value);
    fail = value != 2;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-AC-100: Get a float value.";
{
    float value;
    accessor.GetField(table.GetField("REALFIELD"), value);
    fail = fabs(value - 123.321) > 1e-6;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-AC-110: Go to next record.";
try {
    accessor.GotoNext();
    fail = FALSE;
} catch(const TDbException&) {
    fail = TRUE;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-AC-120: Get a char value.";
{
    char value;
    accessor.GetField(table.GetField("INTEGERFIELD"), value);
    fail = value != -3;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-AC-130: Get a double value.";
{
    double value;
    accessor.GetField(table.GetField("REALFIELD"), value);
    fail = fabs(value - 456.654L) > 1e-12;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-AC-140: Get a short value.";
{
    short value;
    accessor.GetField(table.GetField("INTEGERFIELD"), value);
    fail = value != -3;
}
```

```
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

accessor.GotoNext();

cout << "    DB-AC-150: Get an unsigned short value.";
{
    unsigned short value;
    accessor.GetField(table.GetField("INTEGERFIELD"), value);
    fail = value != 789;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

accessor.GotoNext();

cout << "    DB-AC-160: Get an unsigned int value.";
{
    unsigned int value;
    accessor.GetField(table.GetField("INTEGERFIELD"), value);
    fail = value != 123654L;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-AC-170: Get an unsigned long value.";
{
    unsigned long value;
    accessor.GetField(table.GetField("INTEGERFIELD"), value);
    fail = value != 123654L;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

accessor.GotoNext();

cout << "    DB-AC-180: Get an int value.";
{
    int value;
    accessor.GetField(table.GetField("INTEGERFIELD"), value);
    fail = value != -100000L;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-AC-190: Get a long value.";
{
    long value;
    accessor.GetField(table.GetField("INTEGERFIELD"), value);
    fail = value != -100000L;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-AC-200: Read failure (invalid cast).";
try {
    unsigned char value;
    accessor.GetField(table.GetField("STRINGFIELD"), value);
    fail = TRUE;
} catch(const TDbCannotRead&) {
    fail = FALSE;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-AC-210: Invalid field.";
try {
    unsigned char value;
    accessor.GetField(TDbField("A Non-Existent Field"), value);
    fail = TRUE;
} catch(const TDbDoesNotExist&) {
    fail = FALSE;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;
```

```
    accessor.GotoNext();

    cout << "    DB-AC-220: At end.";
    fail = accessor.IsAtRecord();
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-AC-230: Get field past end.";
    try {
        string value;
        accessor.GetField(table.GetField("STRINGFIELD"), value);
        fail = TRUE;
    } catch(const TDbCannotRead&) {
        fail = FALSE;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

## C. TestCleanup.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: TestCleanup.C,v $
//  $Revision: 2.1 $
//  $Date: 1995/10/10 22:31:45 $
//  $State: Exp $
//  $Author: stretz $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include DBtools.h++ header files.
#include <rw/db/dbase.h>
#include <rw/db/dbmgr.h>
#include <rw/db/result.h>


//  Clean up database.
void TestCleanup()
{
#ifdef SUN4
    RWDBDatabase database = RWDBManager::database("/usr/rogue/lib/librwora.so", "",
"ioc1", "lanl",
            "transims");
#else
    RWDBDatabase database = RWDBManager::database("librwora.so", "", "ioc1", "lanl",
            "transims");
#endif

    database.executeSql("DROP TABLE TESTTABLE1A");
    database.executeSql("DROP TABLE TESTTABLE2A");
    database.executeSql("DROP TABLE TESTTABLE2B");
    database.executeSql("DROP TABLE TESTTABLE2D");

    database.executeSql("DELETE FROM DEPENDENT_TABLES\n"
            "  WHERE TABLE_NAME IN (SELECT TABLE_NAME\n"
            "    FROM TABLE_INDEX\n"
            "    WHERE SOURCE IN ('First Test Source', 'Second Test Source'))");

    database.executeSql("DELETE FROM TABLE_INDEX\n"
            "  WHERE SOURCE IN ('First Test Source', 'Second Test Source')");

    database.executeSql("DELETE FROM SOURCE_INDEX\n"
            "  WHERE NAME IN ('First Test Source', 'Second Test Source')");
}
```

## D. TestDirectory.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: TestDirectory.C,v $
```

```
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include Standard C++ header files.
#include <iostream.h>


//  Include TRANSIMS header files.
#include <DBS/Exception.h>
#include <DBS/Directory.h>


//  Test directory class.
bool TestDirectory()
{
    cout << "  Directory Class Tests"
        << " [$Revision: 2.0 $]"
        << endl;

    bool anyFail = FALSE;
    bool fail;

    TDbDirectory directory(TDbDirectoryDescription("IOC-1"));

    cout << "    DB-DI-010: Constructor and DBtools.h++ access.";
    fail = !directory.GetDatabase().isValid();
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DI-020: Constructor and DBtools.h++ access failure.";
    try {
        TDbDirectory directory2(TDbDirectoryDescription("NODIRECTORY"),
                TDbUserInformation("NOUSER", "NOPASSWORD"));
        fail = TRUE;
    } catch(const TDbException&) {
        fail = FALSE;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DI-030: Retrieval of directory description.";
    fail = directory.GetDescription() != TDbDirectoryDescription("IOC-1");
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    directory.CreateSource(TDbSourceDescription("First Test Source",
            "Here is the comment for the first test source."));
    directory.CreateSource(TDbSourceDescription("Second Test Source",
            "Here is the comment for the second test source."));

    cout << "    DB-DI-040: Source creation.";
    fail = !directory.HasSource("First Test Source");
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DI-050: Source creation failure.";
    try {
        directory.CreateSource(TDbSourceDescription("First Test Source"));
        fail = TRUE;
    } catch(const TDbAlreadyExists&) {
        fail = FALSE;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DI-060: Source existence.";
    fail = !directory.HasSource("Second Test Source");
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DI-070: Source absence.";
```

```
    fail = directory.HasSource("A Non-Existent Source");
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DI-080: Source retrieval.";
    const TDbSourceDescription& source =
            directory.GetSource("First Test Source");
    fail = source.GetName() != "First Test Source" || source.GetComment() !=
            "Here is the comment for the first test source.";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DI-090: Source retrieval failure.";
    try {
        directory.GetSource("A Non-Existent Source");
        fail = TRUE;
    } catch(const TDbDoesNotExist&) {
        fail = FALSE;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DI-100: Source set retrieval.";
    const TDbDirectory::SourceDescriptionSet& set = directory.GetSources();
    TDbDirectory::SourceDescriptionSetIterator iterator(set);
    fail = !set.IsMember(TDbSourceDescription("First Test Source")) ||
            !set.IsMember(TDbSourceDescription("Second Test Source"));
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DI-110: SQL query.";
    try {
        directory.Query("UPDATE SOURCE_INDEX SET COMMENT_TEXT = '' WHERE"
                " NAME = 'First Test Source'");
        fail = FALSE;
    } catch(const TDbQueryFailed&) {
        fail = TRUE;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DI-120: SQL query failure.";
    try {
        directory.Query("NOT AN SQL STATEMENT");
        fail = TRUE;
    } catch(const TDbQueryFailed&) {
        fail = FALSE;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    directory.CreateSource(TDbSourceDescription("Third Test Source"));
    directory.DeleteSource(TDbSourceDescription("Third Test Source"));

    cout << "    DB-DI-130: Source deletion.";
    fail = directory.HasSource("Third Test Source");
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DI-140: Source deletion failure.";
    try {
        directory.DeleteSource(TDbSourceDescription("Fourth Test Source"));
        fail = TRUE;
    } catch(const TDbDoesNotExist&) {
        fail = FALSE;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

## E. TestDirectoryDescription.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: TestDirectoryDescription.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include Standard C++ header files.
#include <iostream.h>


//  Include TRANSIMS header files.
#include <DBS/DirectoryDescription.h>


//  Test directory description class.
bool TestDirectoryDescription()
{
    cout << "  Directory Description Class Tests"
        << " [$Revision: 2.0 $]"
        << endl;

    bool anyFail = FALSE;
    bool fail;

    TDbDirectoryDescription dirDesc("Name", "Comment", "Database", "Server",
            "Library");
    TDbDirectoryDescription::SetDefaultServerName("Default Server");
    TDbDirectoryDescription::SetDefaultAccessLibrary("Default Library");

    cout << "    DB-DD-010: Retrieval of directory name.";
    fail = dirDesc.GetName() != "Name";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DD-020: Retrieval of comment constructed explicitly.";
    fail = dirDesc.GetComment() != "Comment";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DD-030: Retrieval of database name constructed explicitly.";
    fail = dirDesc.GetDatabaseName() != "Database";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DD-040: Retrieval of server name constructed explicitly.";
    fail = dirDesc.GetServerName() != "Server";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DD-050: Retrieval of access library constructed"
            "explicitly.";
    fail = dirDesc.GetAccessLibrary() != "Library";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DD-060: Inequality operator.";
    fail = dirDesc != TDbDirectoryDescription("Name");
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DD-070: Equality operator.";
    fail = dirDesc == TDbDirectoryDescription("Another");
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DD-080: Retrieval of comment constructed from default.";
    fail = TDbDirectoryDescription("").GetComment() != "";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;
```

```
    cout << "    DB-DD-090: Retrieval of database name constructed from"
            " default.";
    fail = TDbDirectoryDescription("").GetDatabaseName() != "transims";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DD-100: Retrieval of server name constructed from default.";
    fail = TDbDirectoryDescription("").GetServerName() != "Default Server";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DD-110: Retrieval of access library constructed from"
        " default.";
    fail = TDbDirectoryDescription("").GetAccessLibrary() != "Default Library";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DD-120: Retrieval of default server name.";
    fail = TDbDirectoryDescription::GetDefaultServerName() != "Default Server";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-DD-130: Retrieval of default access library.";
    fail = TDbDirectoryDescription::GetDefaultAccessLibrary() !=
            "Default Library";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occured." << endl;
    return !anyFail;
}
```

## F. TestField.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: TestField.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include Standard C++ header files.
#include <iostream.h>


//  Include TRANSIMS header files.
#include <DBS/Exception.h>
#include <DBS/Directory.h>
#include <DBS/Source.h>
#include <DBS/Table.h>
#include <DBS/Field.h>


//  Test field class.
bool TestField()
{
    cout << "  Field Class Tests"
        << " [$Revision: 2.0 $]"
        << endl;

    bool anyFail = FALSE;
    bool fail;

    TDbDirectory directory(TDbDirectoryDescription("IOC-1"));
    TDbSource source(directory, directory.GetSource("First Test Source"));
    TDbTable table(source, source.GetTable("Test Table 1A"));
    const TDbTable::FieldSet& fields = table.GetFields();

    cout << "    DB-FE-010: Retrieval of name constructed explicitly.";
    fail = TDbField("Name").GetName() != "Name";
```

```
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-FE-020: Retrieval of type constructed explicitly.";
    fail = TDbField("", TDbField::kUnknown).GetType() != TDbField::kUnknown ||
            TDbField("", TDbField::kUnsignedChar).GetType() !=
            TDbField::kUnsignedChar || TDbField("", TDbField::kShort).GetType()
            != TDbField::kShort || TDbField("",
            TDbField::kUnsignedShort).GetType() != TDbField::kUnsignedShort ||
            TDbField("", TDbField::kInt).GetType() != TDbField::kInt ||
            TDbField("", TDbField::kUnsignedInt).GetType() !=
            TDbField::kUnsignedInt || TDbField("", TDbField::kLong).GetType() !=
            TDbField::kLong || TDbField("", TDbField::kUnsignedLong).GetType()
            != TDbField::kUnsignedLong || TDbField("",
            TDbField::kFloat).GetType() != TDbField::kFloat || TDbField("",
            TDbField::kDouble).GetType() != TDbField::kDouble || TDbField("",
            TDbField::kString).GetType() != TDbField::kString;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-FE-030: Retrieval of size constructed explicitly.";
    fail = TDbField("", TDbField::kUnknown, 324).GetSize() != 324;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-FE-040: Inequality operator.";
    fail = TDbField("Name") != TDbField("Name", TDbField::kString, 10);
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-FE-050: Equality operator.";
    fail = TDbField("Name") == TDbField("Name Another");
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-FE-060: Retrieval of type constructed from default.";
    fail = TDbField("").GetType() != TDbField::kUnknown;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-FE-070: Retrieval of size constructed from default.";
    fail = TDbField("").GetSize() != 0;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-FE-080: Retrieval of string field type.";
    fail = table.GetField("STRINGFIELD").GetType() != TDbField::kString;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-FE-090: Retrieval of string field size.";
    fail = table.GetField("STRINGFIELD").GetSize() != 100;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-FE-100: Retrieval of integer field type.";
    fail = table.GetField("INTEGERFIELD").GetType() != TDbField::kLong;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-FE-110: Retrieval of real field type.";
    fail = table.GetField("REALFIELD").GetType() != TDbField::kDouble;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

## G. TestInserter.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: TestInserter.C,v $
//  $Revision: 2.0 $
```

```
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include Standard C++ header files.
#include <iostream.h>


//  Include Standard C header files.
#include <math.h>


//  Include TRANSIMS header files.
#include <DBS/Exception.h>
#include <DBS/Directory.h>
#include <DBS/Source.h>
#include <DBS/Table.h>
#include <DBS/Field.h>
#include <DBS/Inserter.h>
#include <DBS/Accessor.h>


//  Return whether two real numbers are the same.
static bool Equal(REAL a, REAL b)
{
    return fabs(a - b) < 1e-5;
}


//  Test inserter class.
bool TestInserter()
{
    cout << "  Inserter Class Tests"
        << " [$Revision: 2.0 $]"
        << endl;

    bool anyFail = FALSE;
    bool fail;

    TDbDirectory directory(TDbDirectoryDescription("IOC-1"));
    TDbSource source(directory, directory.GetSource("First Test Source"));
    TDbTable table(source, source.GetTable("Test Table 1A"));
    directory.Query("DELETE FROM TESTTABLE1A");
    TDbAccessor accessor(table);

    cout << "    DB-IN-010: Constructor and DBtools.h++ access.";
    TDbInserter inserter(table);
    fail = FALSE;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-IN-020: Table retrieval.";
    fail = &table != &inserter.GetTable();
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-IN-030: Flush with nothing pending.";
    try {
        inserter.Flush();
        fail = FALSE;
    } catch(const TDbException& exception) {
        fail = TRUE;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-IN-040: Setting fields.";
    try {
        inserter.SetField(table.GetField("STRINGFIELD"), "SELECT");
        inserter.SetField(table.GetField("INTEGERFIELD"), -3);
        inserter.SetField(table.GetField("REALFIELD"), 4.1);
        fail = FALSE;
    } catch(const TDbCannotWrite& exception) {
```

```
        fail = TRUE;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-IN-050: Inserting record.";
    try {
        inserter.Insert();
        inserter.Flush();
        string s;
        int i;
        double f;
        accessor.GotoFirst();
        accessor.GetField(table.GetField("STRINGFIELD"), s);
        accessor.GetField(table.GetField("INTEGERFIELD"), i);
        accessor.GetField(table.GetField("REALFIELD"), f);
        fail = s != "SELECT" || i != -3 || !Equal(f, 4.1) ||
                accessor.GetRecordCount() != 1;
    } catch(const TDbCannotWrite&) {
        fail = TRUE;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-IN-060: Invalid field.";
    try {
        inserter.SetField(TDbField("XYZ"), 321);
        fail = TRUE;
    } catch(const TDbDoesNotExist&) {
        fail = FALSE;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-IN-070: Flushing records.";
    inserter.SetField(table.GetField("STRINGFIELD"), "ABC");
    inserter.Insert();
    fail = accessor.GetRecordCount() != 1;
    inserter.Flush();
    fail |= accessor.GetRecordCount() != 2;
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

## H. TestSortedAccessor.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: TestSortedAccessor.C,v $
//  $Revision: 1.1 $
//  $Date: 1996/02/20 22:18:17 $
//  $State: Exp $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include Standard C++ header files.
#include <iostream.h>


//  Include Standard C header files.
#include <math.h>


//  Include TRANSIMS header files.
#include <DBS/Exception.h>
#include <DBS/Directory.h>
#include <DBS/Source.h>
#include <DBS/Table.h>
#include <DBS/Field.h>
#include <DBS/SortedAccessor.h>
```

```
//  Test sorted accessor class.
bool TestSortedAccessor()
{
    cout << "  Sorted Accessor Class Tests"
        << " [$Revision: 1.1 $]"
        << endl;

    bool anyFail = FALSE;
    bool fail;

    TDbDirectory directory(TDbDirectoryDescription("IOC-1"));
    TDbSource source(directory, directory.GetSource("First Test Source"));
    TDbTable table(source, source.GetTable("Test Table 1A"));
    TDbSource::FieldCollection sortFields;
    sortFields.Append(table.GetField("INTEGERFIELD"));
    TDbSortedAccessor accessor(table, sortFields);

    cout << "    DB-SA-010: Constructor and DBtools.h++ access.";
    fail = !accessor.GetReader().isValid();
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-SA-020: Not at record.";
    fail = accessor.IsAtRecord();
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-SA-030: Go to first record.";
    try {
        accessor.GotoFirst();
        fail = !accessor.GetReader().isValid();
    } catch(const TDbException&) {
        fail = TRUE;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-SA-040: At record.";
    fail = !accessor.IsAtRecord();
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-SA-050: Get a string value.";
    {
        string value;
        accessor.GetField(table.GetField("STRINGFIELD"), value);
        fail = value != "MNO";
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-SA-060: Go to next record.";
    try {
        accessor.GotoNext();
        fail = FALSE;
    } catch(const TDbException&) {
        fail = TRUE;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-SA-070: Get another string value.";
    {
        string value;
        accessor.GetField(table.GetField("STRINGFIELD"), value);
        fail = value != "DEF";
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

## I. TestSource.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: TestSource.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include Standard C++ header files.
#include <iostream.h>


//  Include TRANSIMS header files.
#include <DBS/Exception.h>
#include <DBS/Directory.h>
#include <DBS/Source.h>
#include <DBS/Table.h>


//  Test source class.
bool TestSource()
{
    cout << "  Source Class Tests"
        << " [$Revision: 2.0 $]"
        << endl;

    bool anyFail = FALSE;
    bool fail;

    TDbDirectory directory(TDbDirectoryDescription("IOC-1"));
    TDbSource source(directory, directory.GetSource("Second Test Source"));

    cout << "    DB-SO-010: Constructor and DBtools.h++ access.";
    fail = !source.GetDatabase().isValid();
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-SO-020: Constructor failure for non-existent source.";
    try {
        TDbSource source2(directory, TDbSourceDescription("Not A Source"));
        fail = TRUE;
    } catch(const TDbDoesNotExist&) {
        fail = FALSE;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-SO-030: Retrieval of source description.";
    fail = source.GetDescription().GetName() != "Second Test Source" ||
            source.GetDescription().GetComment() !=
            "Here is the comment for the second test source.";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    source.CreateTable(TDbTableDescription("Test Table 2A",
            "Here is the comment for the second test source's first table.",
            "TESTTABLE2A"), "CREATE TABLE TESTTABLE2A (ITEM VARCHAR(10))");
    source.CreateTable(TDbTableDescription("Test Table 2B",
            "Here is the comment for the second test source's second table.",
            "TESTTABLE2B"), "CREATE TABLE TESTTABLE2B (ITEM VARCHAR(10))");

    cout << "    DB-SO-040: Table creation.";
    fail = !source.HasTable("Test Table 2A");
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-SO-050: Table creation failure due to SQL.";
    try {
        source.CreateTable(TDbTableDescription("Test Table 2C", "",
                "TESTTABLE2C"), "NOT AN SQL STATEMENT");
        fail = TRUE;
```

```
} catch(const TDbCreationFailed&) {
    fail = FALSE;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-SO-060: Table creation failure due to names.";
try {
    source.CreateTable(TDbTableDescription("Test Table 2A", "",
            "TESTTABLE2A"), "CREATE TABLE TESTTABLE2A (ITEM VARCHAR(10))");
    fail = TRUE;
} catch(const TDbAlreadyExists&) {
    fail = FALSE;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-SO-070: Table existence.";
fail = !source.HasTable("Test Table 2A");
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-SO-080: Table absence.";
fail = source.HasTable("A Non-Existent Table");
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-SO-090: Table retrieval.";
const TDbTableDescription& table = source.GetTable("Test Table 2A");
fail = table.GetName() != "Test Table 2A" || table.GetComment() !=
        "Here is the comment for the second test source’s first table." ||
        table.GetTableName() != "TESTTABLE2A";
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-SO-100: Table retrieval failure.";
try {
    source.GetTable("A Non-Existent Table");
    fail = TRUE;
} catch(const TDbDoesNotExist&) {
    fail = FALSE;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-SO-110: Table set retrieval.";
const TDbSource::TableDescriptionSet& set = source.GetTables();
TDbSource::TableDescriptionSetIterator iterator(set);
fail = !set.IsMember(TDbTableDescription("Test Table 2A")) ||
        !set.IsMember(TDbTableDescription("Test Table 2B"));
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-SO-120: Directory retrieval.";
fail = &directory != &source.GetDirectory();
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

source.CreateTable(TDbTableDescription("Test Table 2C",
        "Here is the comment for the second test source’s third table.",
        "TESTTABLE2C"), "CREATE TABLE TESTTABLE2C (ITEM VARCHAR(10))");

cout << "    DB-SO-130: Table deletion.";
source.DeleteTable(TDbTableDescription("Test Table 2C"));
fail = source.HasTable("Test Table 2C");
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-SO-140: Table deletion failure.";
try {
    source.DeleteTable(TDbTableDescription("Test Table 3"));
    fail = TRUE;
} catch(const TDbDoesNotExist&) {
    fail = FALSE;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
```

```
    anyFail |= fail;

    cout << "    DB-SO-150: Table creation w/o SQL.";
    {
        TDbSource::FieldCollection fields;
        fields.Append(TDbField("StringField", TDbField::kString, 10));
        fields.Append(TDbField("DoubleField", TDbField::kDouble));
        fields.Append(TDbField("FloatField", TDbField::kFloat));
        fields.Append(TDbField("UnsignedLongField", TDbField::kUnsignedLong));
        fields.Append(TDbField("LongField", TDbField::kLong));
        fields.Append(TDbField("UnsignedIntField", TDbField::kUnsignedInt));
        fields.Append(TDbField("IntField", TDbField::kInt));
        fields.Append(TDbField("UnsignedShortField", TDbField::kUnsignedShort));
        fields.Append(TDbField("ShortField", TDbField::kShort));
        fields.Append(TDbField("UnsignedCharField", TDbField::kUnsignedChar));
        fields.Append(TDbField("CharField", TDbField::kChar));
        TDbSource::FieldCollection index;
        index.Append(fields[0]);
        index.Append(fields[3]);
        source.CreateTable(TDbTableDescription("Test Table 2D",
                "Here is the comment for the second test source's 4th table.",
                "TESTTABLE2D"), fields, index);
        TDbTable table2(source, source.GetTable("Test Table 2D"));
        TDbTable::FieldSet fields2 = table2.GetFields();
        fail = fields.Length() != fields2.Extent();
        for (TDbSource::FieldCollectionIterator i(fields); !i.IsDone();
                i.Next())
            fail |= !fields2.IsMember(*i.CurrentItem());
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

## J. TestSourceDescription.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: TestSourceDescription.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include Standard C++ header files.
#include <iostream.h>


//  Include TRANSIMS header files.
#include <DBS/SourceDescription.h>


//  Test source description class.
bool TestSourceDescription()
{
    cout << "  Source Description Class Tests"
        << " [$Revision: 2.0 $]"
        << endl;

    bool anyFail = FALSE;
    bool fail;

    TDbSourceDescription sourDesc("Name", "Comment");

    cout << "    DB-SD-010: Retrieval of name constructed explicitly.";
    fail = sourDesc.GetName() != "Name";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-SD-020: Retrieval of comment constructed explicitly.";
```

```
    fail = sourDesc.GetComment() != "Comment";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-SD-030: Inequality operator.";
    fail = sourDesc != TDbSourceDescription("Name");
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-SD-040: Equality operator.";
    fail = sourDesc == TDbSourceDescription("Another");
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-SD-050: Retrieval of comment constructed from default.";
    fail = TDbSourceDescription("").GetComment() != "";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

## K. TestTable.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: TestTable.C,v $
//  $Revision: 2.1 $
//  $Date: 1996/01/02 22:17:03 $
//  $State: Exp $
//  $Author: stretz $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include Standard C++ header files.
#include <iostream.h>


//  Include TRANSIMS header files.
#include <DBS/Exception.h>
#include <DBS/Directory.h>
#include <DBS/Source.h>
#include <DBS/Table.h>


//  Test table class.
bool TestTable()
{
    cout << "  Table Class Tests"
        << " [$Revision: 2.1 $]"
        << endl;

    bool anyFail = FALSE;
    bool fail;

    TDbDirectory directory(TDbDirectoryDescription("IOC-1"));
    TDbSource source(directory, directory.GetSource("First Test Source"));
    source.CreateTable(TDbTableDescription("Test Table 1A",
            "Here is the comment for the first test source's first table.",
            "TESTTABLE1A"),
            "CREATE TABLE TESTTABLE1A (\n"
            "  STRINGFIELD VARCHAR(100) NOT NULL,\n"
            "  INTEGERFIELD NUMBER(38,0),\n"
            "  REALFIELD FLOAT(126),\n"
            "  PRIMARY KEY (STRINGFIELD)\n"
            ")");
    TDbTable table(source, source.GetTable("Test Table 1A"));

    cout << "    DB-TA-010: Constructor and DBtools.h++ access.";
    fail = !table.GetDatabase().isValid();
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;
```

```
cout << "    DB-TA-020: Constructor failure for non-existent table.";
try {
    TDbTable table2(source, source.GetTable("A Non-Existent Table"));
    fail = TRUE;
} catch(const TDbDoesNotExist&) {
    fail = FALSE;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-TA-030: Retrieval of table description.";
fail = table.GetDescription().GetName() != "Test Table 1A" ||
        table.GetDescription().GetComment() != "Here is the comment for the"
        " first test source's first table." ||
        table.GetDescription().GetTableName() != "TESTTABLE1A";
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-TA-040: Addition of dependent table.";
try {
    table.AddDependentTable(TDbSource(directory, directory.GetSource(
            "Second Test Source")).GetTable("Test Table 2B"));
    fail = FALSE;
} catch (const TDbException&) {
    fail = TRUE;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-TA-050: Addition of dependent table failure.";
try {
    table.AddDependentTable(TDbSource(directory, directory.GetSource(
            "Second Test Source")).GetTable("Test Table 2B"));
    fail = TRUE;
} catch (const TDbException&) {
    fail = FALSE;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-TA-060: Retrieval of dependent table set.";
const TDbTable::TableDescriptionSet& set = table.GetDependentTables();
TDbTable::TableDescriptionSetIterator iterator(set);
fail = set.Extent() != 1 || !set.IsMember((string) "Test Table 2B");
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-TA-070: Field existence.";
fail = !table.HasField("INTEGERFIELD");
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-TA-080: Field absence.";
fail = table.HasField((string) "A Non-Existent Field");
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-TA-090: Field retrieval.";
try {
    TDbField field = table.GetField("STRINGFIELD");
    fail = FALSE;
} catch(const TDbException&) {
    fail = TRUE;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;

cout << "    DB-TA-100: Field retrieval failure.";
try {
    TDbField field = table.GetField("A Non-Existent Field");
    fail = TRUE;
} catch(const TDbException&) {
    fail = FALSE;
}
cout << (fail ? " [failed]" : " [passed]") << endl;
anyFail |= fail;
```

```
    cout << "    DB-TA-110: Field set retrieval.";
    const TDbTable::FieldSet& set2 = table.GetFields();
    TDbTable::FieldSetIterator iterator2(set2);
    fail = set2.Extent() != 3 || !set2.IsMember((string) "STRINGFIELD") ||
            !set2.IsMember((string) "INTEGERFIELD") || !set2.IsMember((string)
            "REALFIELD");
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-TA-120: Source retrieval.";
    fail = &source != &table.GetSource();
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-TA-130: Directory retrieval.";
    fail = &directory != &table.GetDirectory();
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-TA-140: Removal of dependent table.";
    try {
        table.RemoveDependentTable(TDbSource(directory, directory.GetSource(
                "Second Test Source")).GetTable("Test Table 2B"));
        fail = FALSE;
    } catch (const TDbException&) {
        fail = TRUE;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-TA-150: Removal of dependent table failure.";
    try {
        table.RemoveDependentTable(TDbSource(directory, directory.GetSource(
                "Second Test Source")).GetTable("Test Table 2B"));
        fail = TRUE;
    } catch (const TDbException&) {
        fail = FALSE;
    }
    cout << (fail ? " [failed]" : " [passed]") << endl;
    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

## L. TestTableDescription.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: TestTableDescription.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include Standard C++ header files.
#include <iostream.h>


//  Include TRANSIMS header files.
#include <DBS/TableDescription.h>


//  Test table description class.
bool TestTableDescription()
{
    cout << "  Table Description Class Tests"
        << " [$Revision: 2.0 $]"
        << endl;

    bool anyFail = FALSE;
    bool fail;
```

```
    TDbTableDescription tabDesc("Name", "Comment", "Table");

    cout << "    DB-TD-010: Retrieval of name explicitly constructed.";
    fail = tabDesc.GetName() != "Name";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-TD-020: Retrieval of comment explicitly constructed.";
    fail = tabDesc.GetComment() != "Comment";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-TD-030: Retrieval of table name explicitly constructed.";
    fail = tabDesc.GetTableName() != "Table";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-TD-040: Inequality operator.";
    fail = tabDesc != TDbTableDescription("Name");
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-TD-050: Equality operator.";
    fail = tabDesc == TDbTableDescription("Another");
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-TD-060: Retrieval of comment constructed from default.";
    fail = TDbTableDescription("").GetComment() != "";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-TD-070: Retrieval of table name constructed from default.";
    fail = TDbTableDescription("").GetTableName() != "";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

## M. TestUserInformation.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: TestUserInformation.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  Copyright © 1995 Regents of the University of California
//  All rights reserved


//  Include Standard C++ header files.
#include <iostream.h>


//  Include TRANSIMS header files.
#include <DBS/UserInformation.h>


//  Test user information class.
bool TestUserInformation()
{
    cout << "  User Information Class Tests"
        << " [$Revision: 2.0 $]"
        << endl;

    bool anyFail = FALSE;
    bool fail;

    TDbUserInformation userInfo("Name", "Password");
    TDbUserInformation::SetDefaultUserName("Default Name");
    TDbUserInformation::SetDefaultPassword("Default Password");
```

```
    cout << "    DB-UI-010: Retrieval of user name constructed explicitly.";
    fail = userInfo.GetUserName() != "Name";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-UI-020: Retrieval of password constructed explicitly.";
    fail = userInfo.GetPassword() != "Password";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-UI-030: Retrieval of user name constructed from default.";
    fail = TDbUserInformation().GetUserName() != "Default Name";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-UI-040: Retrieval of password constructed from default.";
    fail = TDbUserInformation().GetPassword() != "Default Password";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-UI-050: Retrieval of default user name.";
    fail = TDbUserInformation::GetDefaultUserName() != "Default Name";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << "    DB-UI-060: Retrieval of default password.";
    fail = TDbUserInformation::GetDefaultPassword() != "Default Password";
    cout << (fail ? " [failed]" : " [passed]") << endl;
    anyFail |= fail;

    cout << (anyFail ? "    F" : "    No f") << "ailures occurred." << endl;
    return !anyFail;
}
```

## N. TestCleanup.sql

```
--  Project: TRANSIMS
--  Subsystem: Database
--  $RCSfile: TestCleanup.sql,v $
--  $Revision: 1.6 $
--  $Date: 1995/07/18 14:27:42 $
--  $State: Rel $
--  $Author: bwb $
--  Copyright © 1995 Regents of the University of California
--  All rights reserved


--  Remove tables.
DROP TABLE TESTTABLE1A;
DROP TABLE TESTTABLE2A;
DROP TABLE TESTTABLE2B;
DROP TABLE TESTTABLE2C;
DROP TABLE TESTTABLE2D;

--  Remove test data table dependencies.
DELETE FROM DEPENDENT_TABLES
  WHERE TABLE_NAME IN (SELECT TABLE_NAME
    FROM TABLE_INDEX
    WHERE SOURCE IN ('First Test Source', 'Second Test Source'));

--  Remove test data tables.
DELETE FROM TABLE_INDEX
  WHERE SOURCE IN ('First Test Source', 'Second Test Source');

--  Remove test data sources.
DELETE FROM SOURCE_INDEX
  WHERE NAME IN ('First Test Source', 'Second Test Source');

--  Commit changes.
COMMIT;
```

# XI. APPENDIX: Import Utility

This appendix contains the complete C++ source code for the database subsystem data import utility.

## A. Import.C

```
//  Project: TRANSIMS
//  Subsystem: Database
//  $RCSfile: Import.C,v $
//  $Revision: 2.0 $
//  $Date: 1995/08/04 13:19:25 $
//  $State: Rel $
//  $Author: bwb $
//  U.S. Government Copyright 1995
//  All rights reserved


//  Include Standard C++ header files.
#include <fstream.h>
#include <iostream.h>


//  Include TRANSIMS header files.
#include <GBL/Globals.h>
#include <DBS/Exception.h>
#include <DBS/Directory.h>
#include <DBS/Source.h>


int main(int argc, char* argv[])
{
    if (argc != 2) {
        cerr << "Usage: Import file" << endl;
        cerr << endl;
        cerr << "The text file must have the following format:" << endl;
        cerr << "    table name" << endl;
        cerr << "    table comment" << endl;
        cerr << "    table SQL name" << endl;
        cerr << "    source name" << endl;
        cerr << "    CREATE TABLE statement" << endl;
        cerr << "    . . ." << endl;
        cerr << "    );" << endl;
        cerr << "    column names, in order and separated by commas" << endl;
        cerr << "    data, comma and single-quote delimited" << endl;
        cerr << "    . . ." << endl;
        cerr << "    " << endl;
        cerr << "    [repeat all of the above for additional tables" << endl;
        cerr << "See the file \"SampleNetwork.dat\" for an example." << endl;
        return 0;
    }

    string sql;

    try {

        TDbDirectory directory(TDbDirectoryDescription("IOC-1"));

        ifstream infile(argv[1]);

        while (!infile.eof()) {

            char line1[1000];
            char line2[1000];
            char line3[1000];

            infile.getline(line1, 1000);
            infile.getline(line2, 1000);
            infile.getline(line3, 1000);
            const TDbTableDescription table(line1, line2, line3);
            cout << "Processing " << table.GetName() << " . . . ";

            infile.getline(line1, 1000);
            TDbSource source(directory, directory.GetSource(line1));
```

```
            sql = "";
            for (infile.getline(line1, 1000); strcmp(line1, ");") != 0;
                    infile.getline(line1, 1000)) {
                sql.append(line1);
                sql.append("\n");
            }
            sql.append(")");
            source.CreateTable(table, sql);

            infile.getline(line1, 1000);
            const string columns = string(line1);

            for (infile.getline(line1, 1000); strcmp(line1, "") != 0;
                    infile.getline(line1, 1000)) {
                const string values(line1);
                sql =
                    "INSERT INTO " + table.GetTableName() + "\n"
                    + "  (" + columns + ")\n"
                    + "  VALUES(" + values + ")";
                directory.Query(sql);
            }

            cout << "done." << endl;

        }

        cout << "end." << endl;

    } catch(const TDbException& dbException) {

        cerr << endl;
        cerr << "Database exception: " << dbException.GetMessage() << endl;
        cerr << "Last SQL statement: " << endl;
        cerr << sql << endl;

    } catch(...) {

        cerr << endl;
        cerr << "Unknown exception." << endl;

    }

    return 0;
}
```