TRANSPORTATION ANALYSIS SIMULATION SYSTEM (TRANSIMS)

VERSION 1.0

Database Subsystem for IOC-1

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

March 1998

[Tab 6]

Contents

1. INTRODUCTION 5

2. CONCEPTS 7

2.1 DATA DIRECTORY 7
2.2 DATA SOURCE 7
2.3 DATA TABLE 7
2.4 TABLE FILE 7

3. CLASSES 8

3.1 C++ 8
3.1.1 TDbDirectory 11
3.1.2 TDbDirectoryDescription 13
3.1.3 TDbUserInformation 15
3.1.4 TDbSource 16
3.1.5 TDbSourceDescription 18
3.1.6 TDbTable 19
3.1.7 TDbTableDescription 21
3.1.8 TDbAccessor 22
3.1.9 TDbInserter 24
3.1.10 TDbField 26
3.1.11 TDbTableFile 27
3.1.12 TDbValue 32
3.1.13 TDbException 35

4. IMPLEMENTATION 37

4.1 C++ LIBRARIES 37
4.2 ORACLE DATABASE 37
4.3 TABLE FILE FORMAT 39
4.4 LIMITATIONS 40
4.4.1 Direct Data Manipulation 40
4.4.2 Object Interdependencies 40
4.4.3 Description Classes 40
4.5 PREDEFINED DATA DIRECTORY DESCRIPTIONS 41

5. SETUP 42

5.1 ORACLE DATABASE 42
5.2 FILE DATABASE 42
5.3 COMPILATION 43

6. EXAMPLES 44

6.1 RETRIEVING DATA 44
6.2 INSERTING DATA 44

7. FUTURE WORK 46

8. REFERENCES 47

1. 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.

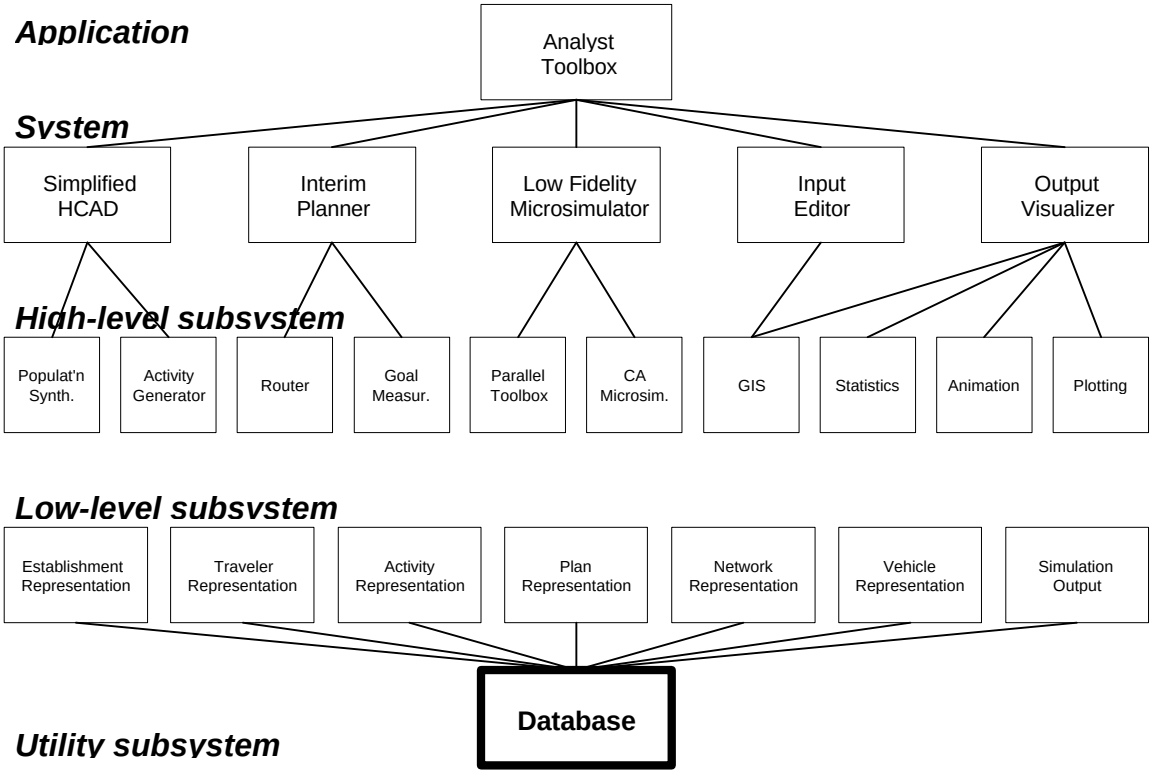

Figure 1: Location of the Database Subsystem in the TRANSIMS Software Architecture

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 subsystem can be compiled in four different versions:

- Solaris Oracle-based Booch version [the original version and the default]

- Solaris file-based Booch version
- Solaris file-based STL version
- Windows NT file-based STL version

The Oracle-based 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. Several database management utility programs are also available.

2. CONCEPTS

2.1 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.

2.2 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.

2.3 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.

2.4 Table File

A table file is an ASCII text file that contains data formatted in a relational format (i.e., each record in the table has a tuple of fields) and a header specifying the field metadata. These files are portable between operating systems and can be used to import and export data.

3. CLASSES

3.1 C++

The database subsystem has classes for locating, describing, and manipulating data and metadata. Figure 2 and Figure 3 show the relationships between classes.

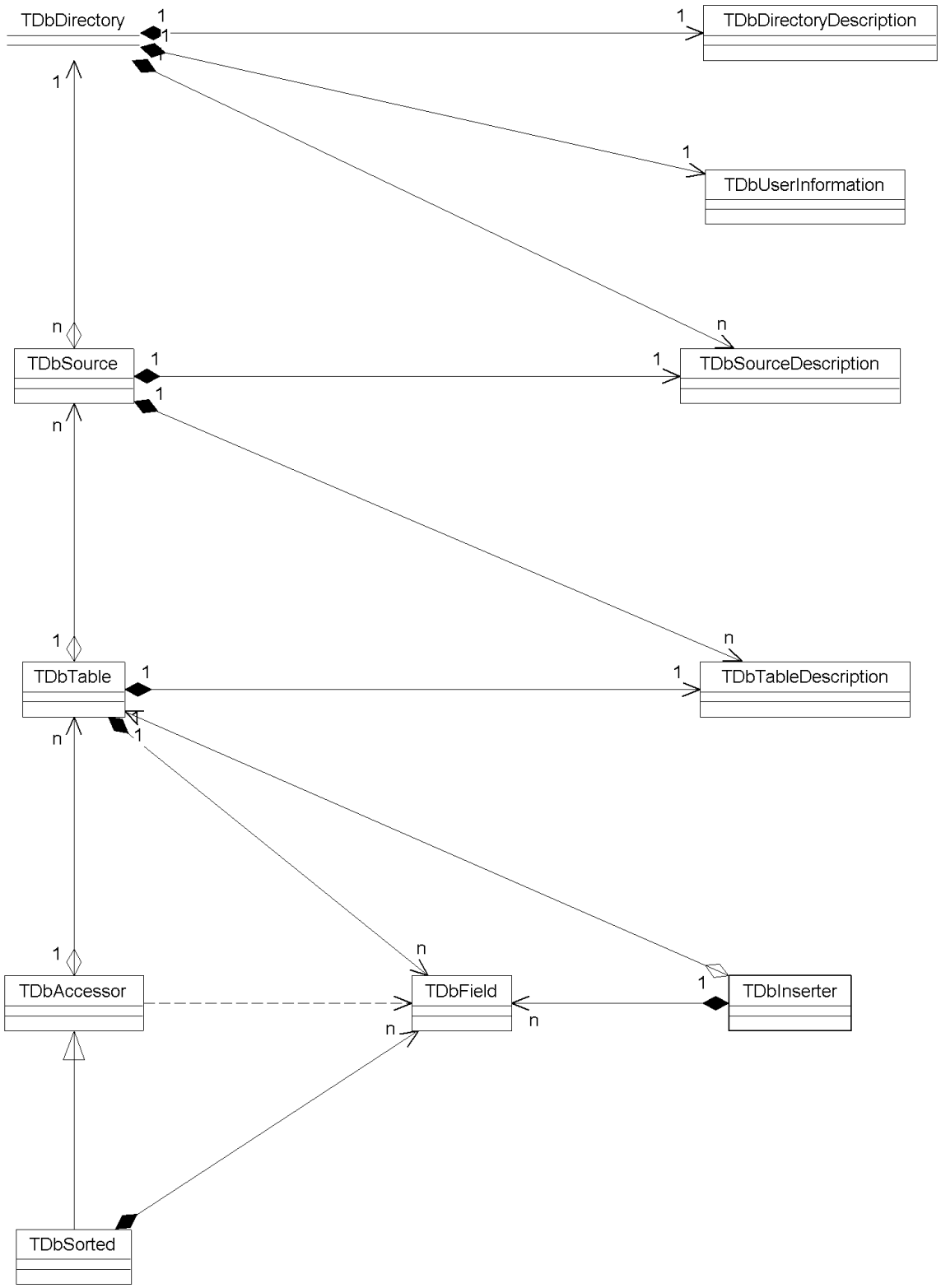

Figure 2: Class Diagram for the TRANSIMS Database Subsystem (unified notation)

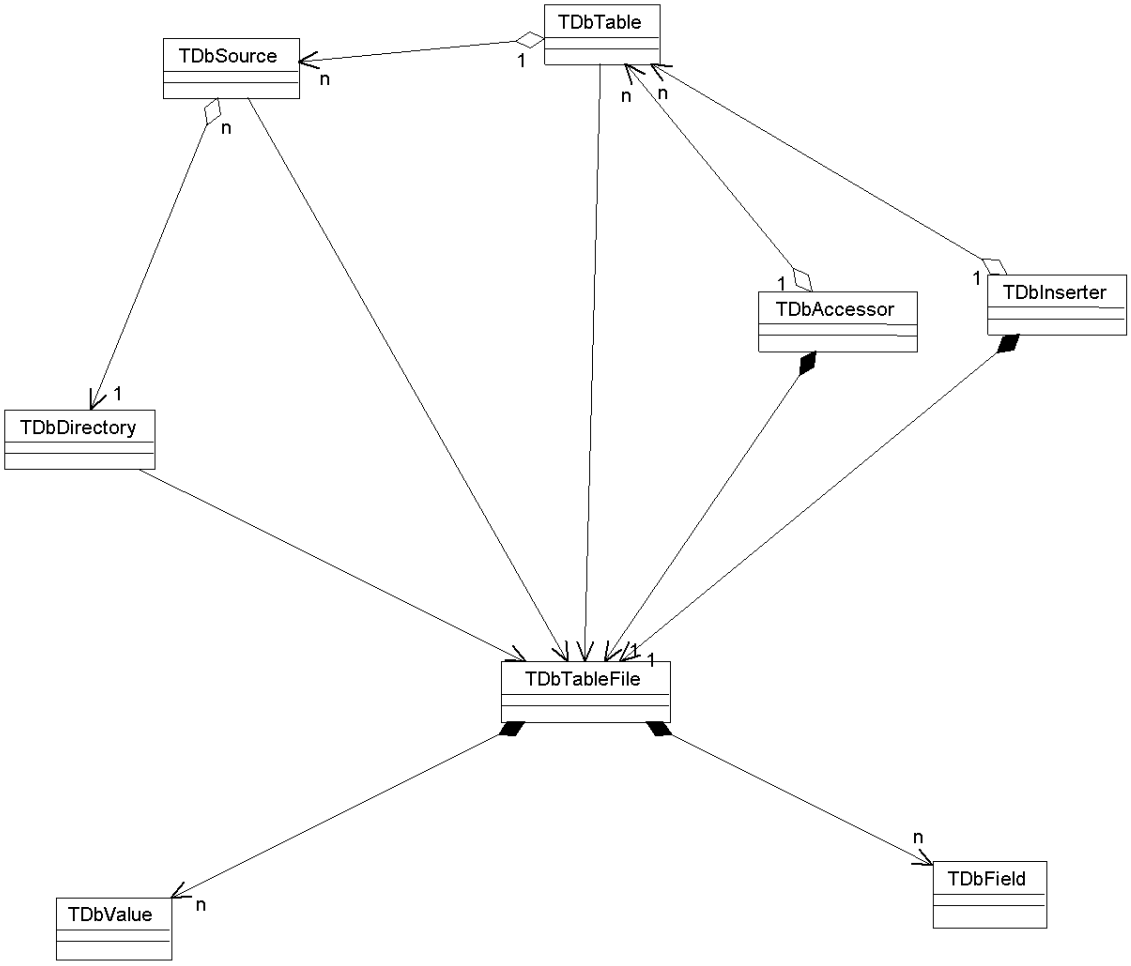

Figure 3: Relationship of the Database Subsystem Classes TDbTableFile and TDbValue with the Other Classes (unified notation)

3.1.1 TDbDirectory

The data directory manages the available data sources and provides services for finding and manipulating the data sources.

Public access:

```
typedef BC_TUnboundedSet<const TDbSourceDescription,
      25U, BC_CManaged> SourceDescriptionSet
typedef BC_TSetActiveIterator<const
      TDbSourceDescription>
      SourceDescriptionSetIterator
typedef set<TDbSourceDescription,
      less<TDbSourceDescription> >
      SourceDescriptionSet
typedef SourceDescriptionSet::iterator
      SourceDescriptionSetIterator
typedef SourceDescriptionSet::const_iterator
      SourceDescriptionSetConstIterator
```

Type definitions.

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

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. Available only if `DBS_FILES` has not been defined.

`RWDBDatabase& GetDatabase()`
Return the database instance for the data directory. Note that this exposes the implementation. Available only if `DBS_FILES` has not been defined.

`RWDBConnection& GetConnection()`
Return the connection instance for the data directory. Note that this exposes the implementation. Available only if `DBS_FILES` has not been defined.

`const string& GetDirectoryName() const`
Return the directory name for the data directory. Note that this exposes the implementation. Available only if `DBS_FILES` has been defined.

Private access:

`void UpdateSourceDescriptions()`
Update the source descriptions.

`TDbDirectoryDescription fDescription`
Each data directory has a description.

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

`RWDBDatabase fDatabase`
Each directory is connected to a database.

`RWDBConnection fConnection`
Each directory has an explicit connection.

`string fDirectoryName`
Each directory corresponds to a file system directory.

3.1.2 TDbDirectoryDescription

A data directory description uniquely specifies a data directory.

Public access:

```
TDbDirectoryDescription(const string& name, const
      string& comment = "", const string& databaseName
      = "", const TDbUserInformation& userInformation
      = TDbUserInformation() , 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 TDbUserInformation& GetUserInformation() const
```

Return the user information.

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

Return the name of the server. Available only if `DBS_FILES` has not been defined.

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

Return the name of the access library. Available only if `DBS_FILES` has not been defined.

```
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.

`bool operator<(const TDbDirectoryDescription& description) const`
Return whether the directory description has a name preceding the specified directory description.

`static const string GetDefaultServerName()`
Return the name of the default server. Available only if `DBS_FILES` has not been defined.

`static void SetDefaultServerName(const string& serverName)`
Set the name of the default server to the specified name. Available only if `DBS_FILES` has not been defined.

`static const string GetDefaultAccessLibrary()`
Return the name of the default access library. Available only if `DBS_FILES` has not been defined.

`static void SetDefaultAccessLibrary(const string& accessLibrary)`
Set the name of the default access library to the specified name. Available only if `DBS_FILES` has not been defined.

Private access:

`string fName`
A directory has a name.

`string fComment`
A directory has a comment.

`string fDatabaseName`
A directory has a database name.

`TDbUserInformation fUserInformation`
User information is required for using a directory.

`string fServerName`
A server is required for using a directory.

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

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

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

3.1.3 TDbUserInformation

An instance of user information contains the information necessary for a user to access a database.

Public access:

```
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.

Private access:

```
string fUserName
```
A user information instance has a user name.

```
string fPassword
```
A user information instance has a password.

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

```
static char fgPassword[1000]
```
There is a default password.

3.1.4 TDbSource

A data source organizes the different data tables (versions) that may exist for a type of data.

Public access:

```
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
typedef set<TDbTableDescription,
      less<TDbTableDescription> > TableDescriptionSet
typedef TableDescriptionSet::iterator
      TableDescriptionSetIterator
typedef TableDescriptionSet::const_iterator
      TableDescriptionSetConstIterator
typedef vector<TDbField> FieldCollection
typedef FieldCollection::iterator
      FieldCollectionIterator
typedef FieldCollection::const_iterator
      FieldCollectionConstIterator
```
Type definitions.

```
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. Available only if `DBS_FILES` has not been defined.

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

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. Available only if `DBS_FILES` has not been defined.

```
RWDBConnection& GetConnection()
```

Return the connection instance for the data source. Note that this exposes the implementation. Available only if `DBS_FILES` has not been defined.

```
const string& GetDirectoryName() const
ivate
```

Return the directory name for the data directory. Note that this exposes the implementation. Available only if `DBS_FILES` has been defined.

```
void UpdateTableDescriptions()
```
Update the table descriptions.

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

```
TDbSourceDescription fDescription
```
Each data source has a description.

```
TableDescriptionSet fTableDescriptions
```
Each data source has several tables, identified by their descriptions.

3.1.5 TDbSourceDescription

A data source description uniquely specifies a data source within a data directory.

Public access:

```
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.

```
bool operator<(const TDbSourceDescription&
     description) const
```
Return whether the source description has a name preceding the specified source description.

Private access:

`string fName`
A source has a name.

`string fComment`
A source has a comment.

3.1.6 TDbTable

Each data table contains fields (columns/attributes) and records (rows/tuples). Each record may have a unique key (primary index).

Public access:

```
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
typedef set<TDbField, less<TDbField> > FieldSet
typedef FieldSet::iterator FieldSetIterator
typedef FieldSet::const_iterator
      FieldSetConstIterator
typedef set<TDbTableDescription,
      less<TDbTableDescription> > TableDescriptionSet
typedef TableDescriptionSet::iterator
      TableDescriptionSetIterator
typedef TableDescriptionSet::const_iterator
      TableDescriptionSetConstIterator
```

Type definitions.

`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 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. Available only if `DBS_FILES` has not been defined.

`RWDBConnection& GetConnection()`
Return the connection instance for the data table. Note that this exposes the implementation. Available only if `DBS_FILES` has not been defined.

`const string& GetDirectoryName() const`
`otected`
Return the directory name for the data directory. Note that this exposes the implementation. Available only if `DBS_FILES` has been defined.

`void UpdateDependents()`
Update the dependent tables.

`void UpdateFields()`
Update the fields.

Private access:

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

`TDbTableDescription fDescription`
Each data table has a description.

`TableDescriptionSet fDependents`
Each data table has dependent tables.

`FieldSet fFields`
Each data table has several fields.

3.1.7 TDbTableDescription

A data table description uniquely specifies a data table.

Public access:

```
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.

```
bool operator<(const TDbTableDescription&
      description) const
```
Return whether the table description has a name preceding the specified table description.

Private access:

`string fName`
A table has a name.

`string fComment`
A table has a comment.

`string fTableName`
A table has a table name (in the database).

3.1.8 TDbAccessor

A table accessor provides facilities for navigating and editing a data table's records. The records are not necessariliy retrieved in a particular order. The accessor may or may not be at a record (see Figure 4). 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.

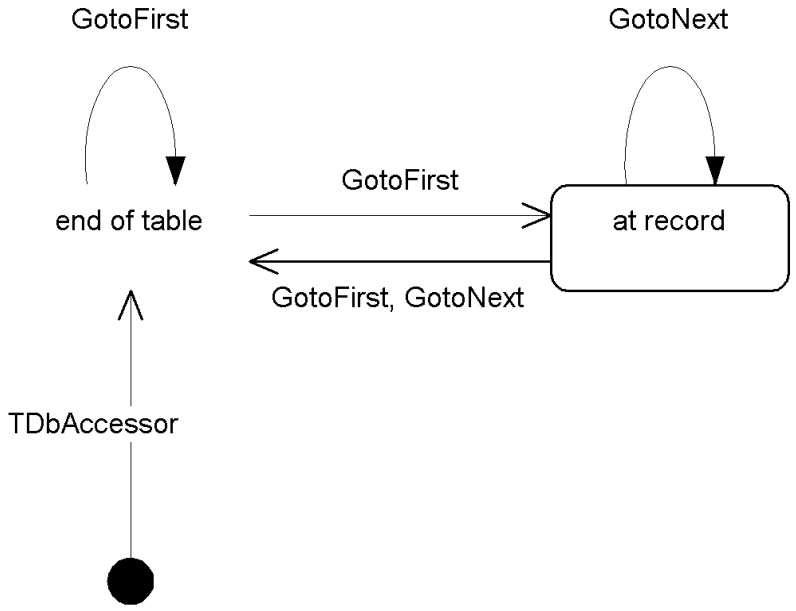

Figure 4: Table Accessor

Public access:

`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. Available only if `DBS_FILES` has not been defined.

Protected access:

`TDbTable* fTable`
Each table accessor is associated with a table.

`RWDBTable fDbTable`
Each table accessor has a database table.

`RWDBReader fReader`
Each table accessor has a reader.

`TDbTableFile fFile`
Each table accessor has a file.

`bool fIsAtRecord`
The reader may be at a record or not.

3.1.9 TDbInserter

This class is used for inserting data into a table.

Public access:

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

`~TDbInserter()`
Destroy an inserter.

`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.

Private access:

```
typedef BC_TUnboundedMap<TDbField, RWDBValue*, 30U,
     BC_CManaged> FieldMap
typedef BC_TMapActiveIterator<TDbField, RWDBValue*>
     FieldMapIterator
```

Type definitions.

```
void SetField(const TDbField& field, RWDBValue* const
     value)
```

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

`TDbTable* fTable`

Each table inserter is associated with a table.

`RWDBTable fDbTable`

Each table inserter has a database table.

`RWDBConnection fConnection`

Each table inserter has its own database connection.

`RWDBPhraseBook fPhraseBook`

Each table insert has a phrase book for the database.

`FieldMap fFields`

Each table insert has a field map holding the current values of the fields.

`TDbTableFile fFile`

Each table has a file.

3.1.10 TDbField

A field is a column in a data table.

Public access:

```
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.

```
const string& GetFormat() const
```
Return the format 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.

```
bool operator<(const TDbField& field) const
```
Return whether the field has a name preceding the specified field.

Private access:

```
string fName
```
A field has a name.

```
EType fType
```
A field has a type.

`size_t fSize`
A field may have a size (if it is a string).

3.1.11 TDbTableFile

A table file implements a relational table on a file system.

Public access:

```
typedef BC_TUnboundedCollection<TDbField,
      BC_CManaged> FieldCollection
typedef BC_TCollectionActiveIterator<TDbField>
      FieldCollectionIterator
typedef BC_TUnboundedCollection<TDbValue,
      BC_CManaged> ValueCollection
typedef BC_TCollectionActiveIterator<TDbValue>
      ValueCollectionIterator
typedef vector<TDbField> FieldCollection
typedef FieldCollection::iterator
      FieldCollectionIterator
typedef FieldCollection::const_iterator
      FieldCollectionConstIterator
typedef vector<TDbValue> ValueCollection
typedef ValueCollection::iterator
      ValueCollectionIterator
typedef ValueCollection::const_iterator
      ValueCollectionConstIterator
```

Type definitions.

`TDbTableFile(const string& filename, bool writeable = false)`
Open a table file with the specified file name, making it writeable if requested. A `TDbDoesNotExist` exception is thrown if the file does not exist.

`TDbTableFile(const string& filename, const FieldCollection& fields, const FieldCollection& index = FieldCollection(), bool overwrite = false)`
Create a table file with the specified file name, fields, and index. A `TDbAlreadyExists` exception is thrown if the file already exists and overwriting is not requested.

`~TDbTableFile()`
Close a table file.

`static void Delete(const string& filename)`
Delete the specified table. A `TDbDoesNotExist` exception is thrown if the file does not exist.

`void Begin()`
Go to the beginning. A `TDbCannotRead` exception is thrown if the position cannot be set.

`bool GetNextRecord()`
Get the next record, returning whether there was one. A `TDbCannotRead` exception is thrown if the record cannot be read.

`void AppendRecord()`
Append the current record. A `TDbCannotWrite` exception is thrown if the record cannot be appended.

`void DeleteRecord()`
Delete the current record. A `TDbCannotWrite` exception is thrown if the record cannot be deleted.

`void Flush()`
Flush any buffered data.

`int GetRecordCount() const`
Return the number of records. A `TDbCannotRead` exception is thrown if the table cannot be read.

`const FieldCollection& GetFields() const`
Return the fields.

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

Get the value of a field. A `TDbDoesNotExist` exception is thrown if the field does not exist.

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

Set a field to the specified value. A `TDbDoesNotExist` exception is thrown if the field does not exist.

```
void ClearField(const string& field)
void ClearField(const TDbField& field)
```

Clear a field. A `TDbDoesNotExist` exception is thrown if the field does not exist.

Protected access:

```
TDbField ReadFieldDescription()
```

Read the next field description from the file.

```
TDbValue ReadField(const TDbField& field)
```

Read the next field from the file.

`void UnreadRecord()`
Undo the last record read from the file.

`void UnreadField()`
Undo the last field read from the file.

`void ReadDelimiter(bool record = false)`
Read the next delimiter from the file.

`void SkipRecord()`
Advance one record.

`void WriteFieldDescription(const TDbField& field)`
Write the field description next on the file.

```
void WriteField(const TDbField& field, const
     TDbValue& value)
```
Write the field next on the file.

`void WriteDelimiter(bool record = false)`
Write the delimiter next on the file.

`int PeekChar() const`
Return the next character to be read, or -1 if there is no character.

`bool IsAtFieldDescription() const`
Return whether the file is at a field description.

`bool IsAtRecord() const`
Return whether the file is at a record.

`bool IsAtDeletedRecord() const`
Return whether the file is at a deleted record.

`bool IsAtEnd() const`
Return whether the file is at its end.

`void Lock(bool writelock)`
Lock the file.

`void Unlock()`
Unlock the file.

Private access:

`TDbTableFile(const TDbTableFile&)`
Do not allow table files to be cloned.

`TDbTableFile& operator=(const TDbTableFile&)`
Do not allow table files to be copied.

```
FILE* fFile
```
Each table file has a handle.

```
FieldCollection fFields
```
Each table file has a field collection.

```
ValueCollection fValues
```
Each table file has a value collection.

```
long fOffset
```
Each table file has an offset for the start of data.

```
static const TDbField kHeader
static const TDbField kHeader
static const TDbField kType
static const TDbField kType
static const TDbField kSign
static const TDbField kSign
static const TDbField kSize
static const TDbField kSize
static const TDbField kName
static const TDbField kName
static const TDbField kIndicator
```
There is a header in a table file.

3.1.12 TDbValue

A value represents typed data.

Public access:

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

```
TDbValue()
TDbValue(char value)
TDbValue(unsigned char value)
TDbValue(short value)
TDbValue(unsigned short value)
TDbValue(int value)
TDbValue(unsigned int value)
TDbValue(long value)
TDbValue(unsigned long value)
TDbValue(float value)
TDbValue(double value)
TDbValue(const string& value)
```
Create a value.

```
~TDbValue()
```
Destroy a value.

```
TDbValue(const TDbValue& value)
```
Construct a copy of the specified value.

```
TDbValue& operator=(const TDbValue& value)
```
Make the value a copy of the specified value.

```
TDbValue& operator=(char value)
TDbValue& operator=(unsigned char value)
TDbValue& operator=(short value)
TDbValue& operator=(unsigned short value)
TDbValue& operator=(int value)
TDbValue& operator=(unsigned int value)
TDbValue& operator=(long value)
TDbValue& operator=(unsigned long value)
TDbValue& operator=(float value)
TDbValue& operator=(double value)
TDbValue& operator=(const string& value)
```
Assign the specified value.

```
EType GetType() const
```
Return the type of the value.

```
bool IsUnknown() const
bool IsChar() const
bool IsUnsignedChar() const
bool IsShort() const
bool IsUnsignedShort() const
bool IsInt() const
bool IsUnsignedInt() const
bool IsLong() const
bool IsUnsignedLong() const
bool IsFloat() const
bool IsDouble() const
bool IsString() const
```
Return whether the value has a specified type.

```
bool operator==(const TDbValue& value) const
bool operator!=(const TDbValue& value) const
```
Return whether the value is different from the specified value.

```
char AsChar() const
unsigned char AsUnsignedChar() const
short AsShort() const
unsigned short AsUnsignedShort() const
int AsInt() const
unsigned int AsUnsignedInt() const
long AsLong() const
unsigned long AsUnsignedLong() const
float AsFloat() const
double AsDouble() const
string AsString() const
```

Return the data.

Private access:

```
EType fType
```

A value has a type.

```
union { char fChar
unsigned char fUnsignedChar
short fShort
unsigned short fUnsignedShort
int fInt
unsigned int fUnsignedInt
long fLong
unsigned long fUnsignedLong
float fFloat
double fDouble
string* fString
}
```

A value has data.

3.1.13 TDbException

A database exception signals the failure of a database subsystem function. Each exception has a message. Figure 5 shows the hierarchy of exception classes.

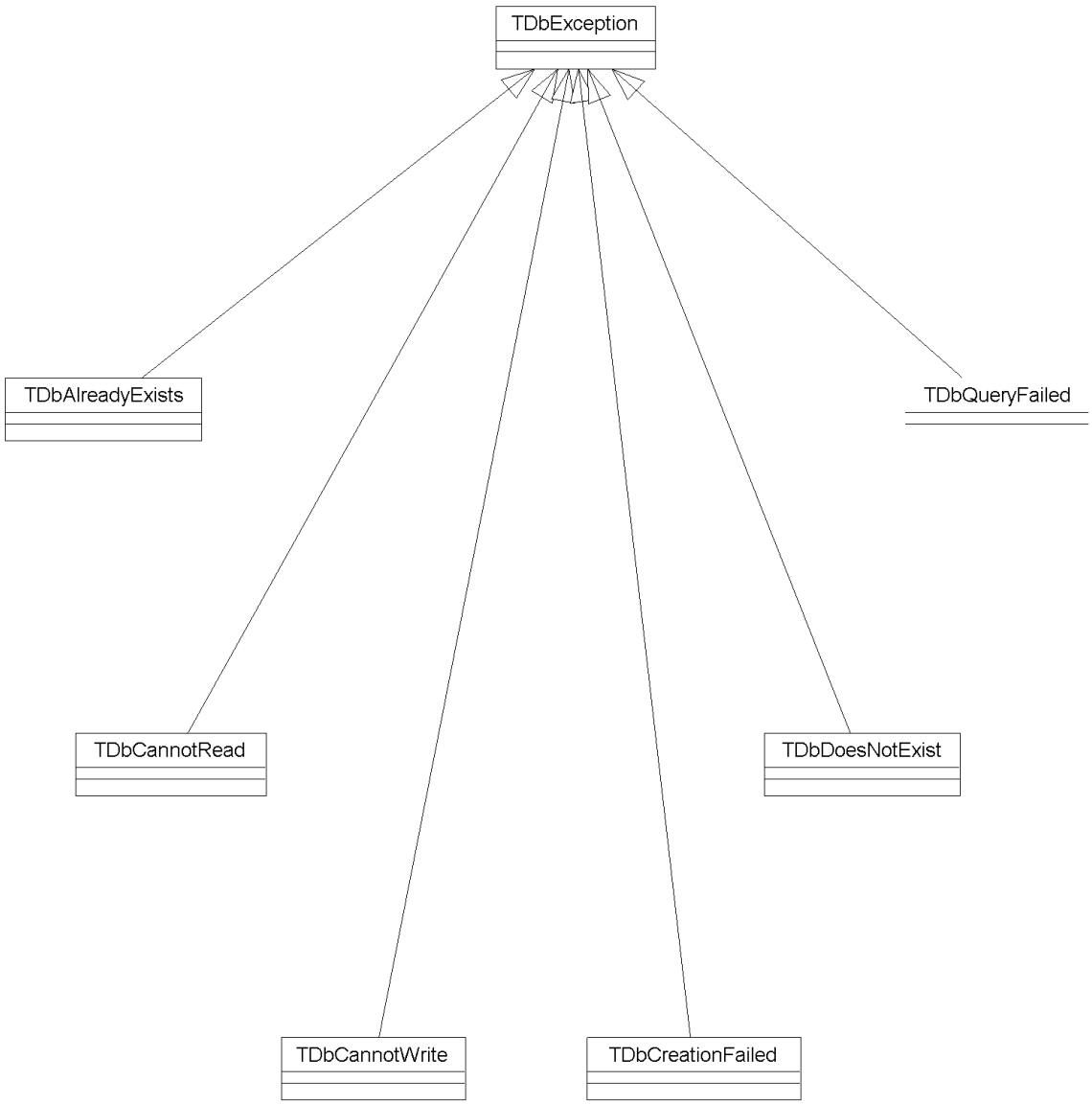

Figure 5: Exception Hierarchy (unified notation)

```
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.

```
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.

4. IMPLEMENTATION

4.1 C++ Libraries

The Oracle-based 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 subsystem also uses the Tools.h++ library [Ke 94] via the DBtools.h++ library dependence on Tools.h++.

The Booch-based implementation of the database subsystem uses the Booch Components [RW 94] extensively for C++ container classes.

All versions of the database subsystem use 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).

4.2 Oracle Database

The DBtools.h++ library requires five parameters in order to establish a connection with a relational database [Ga 95a]:

- *Database Name:* This is the name of the Oracle database.
- *Server Name:* This is the name of the machine on which the Oracle database resides. This can be left blank because SQL*NET is able to locate the database from the `ORACLE_SID` and `TWO_TASK` environment variables.
- *Access Library:* This is the name of the DBtools.h++ shared library for Oracle.
- *User Name:* This is the user name under which the TRANSIMS data resides in the Oracle database.
- *Password:* This is the password for the user name.

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 (
    SOURCE VARCHAR(50) NOT NULL,
    NAME VARCHAR(50) NOT NULL,
    TABLE_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
);
```

References [CAD 95] [CG 93] [Ve 95] provide information on database tuning and backup.

4.3 Table File format

Table files are standard UNIX-style ASCII text files. Lines are delimited by a new-line character (`\n`). The first line must be "TRANSIMS Database File V1". The following lines provide the field metadata, one line for each field. They are in tab-delimited format with the columns given in Table 1.

Table 1: Columns for field descriptions in table files

Column Number	Column Name	Interpretation
1	Type	type of data
2	Sign	whether the data is signed
3	Size	size of the data
4	Name	name of the field (no more than 50 characters)

The valid field types are specified in Table 2.

Table 2: Valid field types for table files

Type	Sign	Size	C++ Data Type
I	S	1	`char`
I	U	1	`unsigned char`
I	S	2	`short`
I	U	2	`unsigned short`
I	S	4	`int` or `long`
I	U	4	`unsigned int` or `unsigned long`
F	X	4	`float`
F	X	8	`double`
S	X	x	`string` with length no more than x

The data records follow the field metadata. Each line represents a record in tab-delimited format. Column `(i + 1)` contains the data for the `i`-th field in text format. The first field contains a plus (`+`) if the record is valid or a minus (`-`) if the record has been deleted and should be ignored.

4.4 Limitations

4.4.1 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.

4.4.2 Object Interdependencies

The following dependencies exist 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.

4.4.3 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.

4.5 Predefined Data Directory Descriptions

The header file Directories.h contains the global objects shown in Table 3.

Table 3: Predefined data directories

C++ Object	Name	Oracle Location	File System Location
`extern const TDbDirectoryDescription kgTest`	Test Directory	test/lanl@transims	/home/projects/transims/database/test
`extern const TDbDirectoryDescription kgIOC1`	IOC #1	ioc1/lanl@transims	/home/projects/transims/database/ioc1
`extern const TDbDirectoryDescription kgCS1`	Case Study #1	cs1/lanl@transims	/home/projects/transims/database/cs1
`extern const TDbDirectoryDescription kgPKG1`	Package #1	pkg1/lanl@transims	/home/projects/transims/database/pkg1
`extern const TDbDirectoryDescription kgIOC2`	IOC #2	ioc2/lanl@transims	/home/projects/transims/database/ioc2
`extern const TDbDirectoryDescription kgTRANSIMS1`	TRANSIMS #1	transims1/letmein@transims	/home/projects/transims/database/transims1

5. SETUP

5.1 Oracle Database

Oracle 7 requires several configuration files containing database parameters. The file inittransims.ora is the main parameter file that sets general parameters and includes the TRANSIMS-specific parameter file configtransims.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.

In order to create a new TRANSIMS database in Oracle 7, one must execute the SQL scripts described 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.

5.2 File Database

To set up the file-based database subsystem, one must put a copy of the files SOURCE_INDEX.tbl, TABLE_INDEX.tbl, and DEPENDENT_TABLES.tbl in the file-system directory corresponding to the database subsystem data directory.

5.3 Compilation

Table 4 shows the compilation settings for the various versions of the database subsystem.

Table 4: Compilation settings for the database subsystem

Operating System	Data Storage	Container Classes	Compiler Flags	Make File	Suffix for Libraries and Executables
Solaris	Oracle	Booch		Makefile	
Solaris	files	Booch	-DDBS_FILES	Makefile.FILES	.FILES
Solaris	files	STL	-DGBL_STL -DDBS_FILES -DDBS_STL	Makefile.STL	.STL
Windows NT	files	STL	-DGBL_STL -DDBS_FILES -DDBS_STL	n/a	.STL

6. EXAMPLES

The following examples show how to navigate, access, and add data to the database.

6.1 Retrieving Data

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

```
//  Open the data directory.
TDbDirectory directory(kgIOC1);
//  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.
}
```

6.2 Inserting Data

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

```
//  Open the directory.
TDbDirectory
directory(TDbDirectoryDescription(kgCS1));
//  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();
```

7. FUTURE WORK

Future work planned for the TRANSIMS database subsystem will focus on the following areas:

- add generic data compression via a `TDbCompressedTableFile` class
- provide a import and export capability with Hierarchical Data Format (HDF) files
- add table editing features to the Java graphical interface for ManageDatabase
- support remote usage of the Java graphical interface for ManageDatabase
- optimize the C++ code (including inlining)
- fix copy semantics in `TDbInserterModified`
- construct a new `TDbSortedAccessor` class
- add indexing to file-based tables
- add a capability for large data directories
- add sorting capabilities
- add temporary table capabilities