

# Object-Oriented Design of Database Stored Procedures

Michael Blaha, blaha@computer.org  
Bill Huth, bhuth@qbase.us  
Peter Cheung, peter.cheung@modelsoftcorp.com

## Abstract

*Object-oriented (OO) software engineering techniques are often used with programming languages, but they also apply to relational databases. OO techniques are not only helpful for determining database structure, but also for designing stored procedure code. In fact, we were surprised by the magnitude of the stored procedure benefits. OO techniques boost development productivity as well as the quality, clarity, and maintainability of the resulting code.*

## 1. Introduction

Object-oriented (OO) techniques are versatile. They are not only helpful for developing programs and structuring databases, but they are also effective for designing stored procedures. A **stored procedure** is programming code that runs in the address space of a database management system (DBMS). All the major DBMS products (such as SQL Server, Oracle, MySQL, DB2) now support stored procedures.

When we build database applications, we start by preparing a UML class model and then create a corresponding database structure [2]. One of the subsequent steps is to implement **methods**, that is the behavior for classes. We have found stored procedures to be helpful for implementing methods.

## 2. The SQL Server SP Language

We have been using SQL Server in our recent applications and emphasize it in this article. However, we believe that many of the techniques we discuss here would apply to other DBMSs.

SQL Server has three kinds of stored procedures: extended stored procedures (XPs), common-language-runtime stored procedures (CLRPs, new to SQL Server 2005 [1]), and interpreted stored procedures (SPs). Developers write XPs and CLRPs in a programming language (such as C or C++), compile them, and then link them into the database. In contrast, SPs are written with Microsoft's Transact-SQL language and are interpreted with linking an implicit part of the language.

For convenience, we have favored SPs in our applications. Of the three approaches, SPs are the easiest to write and the interpreter overhead is small compared to data-

base I/O. One nice feature of SPs is their support for optional input parameters. An SP may have optional inputs if: (1) the definition lists all outputs before inputs and (2) optional inputs have default values.

## 3. Financial Application Example

The examples in this article are based on an application for managing syndicated loans. The loans can be huge, involving billions of dollars, and can arise from lending to governments and multinational corporations. Participation in a loan must be spread across multiple lenders because the risk is too large for a single lender.

Figure 1 shows an excerpt of a UML class model for the application. An **Asset** is something of value and can be a **Currency** (such as US Dollars, Euros, and Yen), a **LoanInstrument**, or an **OtherAsset** (such as bonds and stock). An **Asset** can involve various **CounterPartyRoles**. Methods (SPs) link an **Asset** to a **CounterPartyRole** (*bindCPR*), unlink an **Asset** from a **CounterPartyRole** (*unbindCPR*), and retrieve the collection of **CounterPartyRoles** for an **Asset** (*getCPRs*).

A **CounterParty** is some organization that is relevant for the purpose of tracking loans. (The term **CounterParty** is banking jargon.) **CounterPartyRoleType** specifies the ways that **CounterParties** can be involved in loans; examples include borrower, lender, guarantor, and agent. A **CounterPartyRole** combines a **CounterParty** with a **CounterPartyRoleType**. The software not only tracks who is involved with **Assets**, but also tracks how they are involved. Methods can create a new object (*new*), delete an object (*delete*), modify data (*update*), and retrieve data (*getWithID*).

A **LoanInstrument** can be a **FacilityAgreement**, **Tranche**, or **TrancheItem**. A **FacilityAgreement** is an overall financial amount that is available to a borrower. A **Tranche** is a loan that is made under the auspices of a **FacilityAgreement**. **Tranches** are important because they provide the means for managing **Drawdowns** (borrowed funds). A **TrancheItem** is the portion of a **Tranche** covered by a particular lender. Thus a **Tranche** splits into **TrancheItems**, one for each lender.

Via inheritance from **Asset**, the objects in the different subclasses can have **CounterPartyRoles**. Thus **TrancheItem** inherits **CounterPartyRoles**. The model does not en-

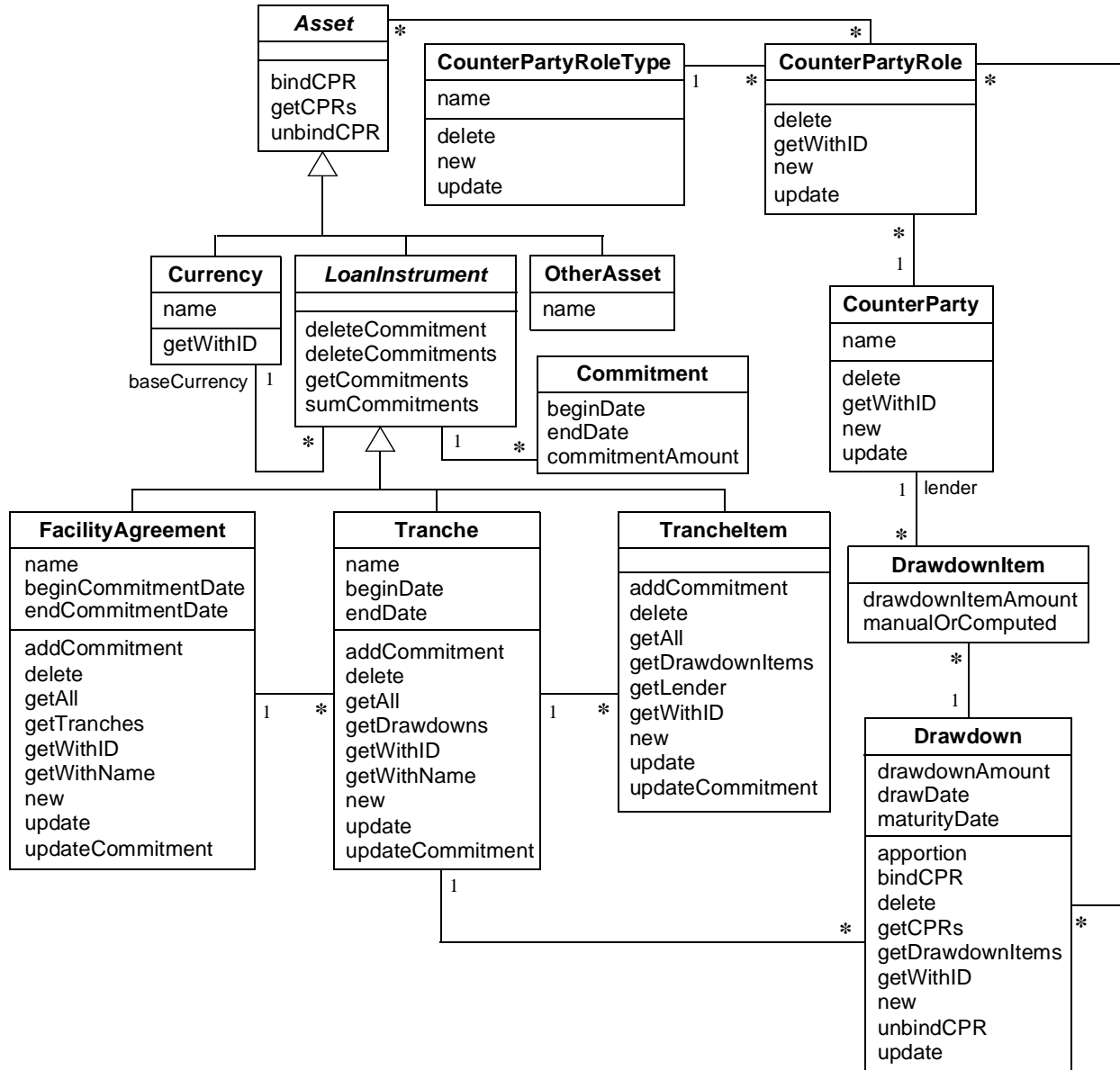


Figure 1 Excerpt of UML class model with public methods

force some business constraints: (1) each *TrancheItem* has exactly one lender and (2) of the many *TrancheItems* for a *Tranche*, exactly one *TrancheItem* pertains to each lender. Since an *Asset* can be bound to any number of *CounterPartyRoles*, the model cannot enforce these constraints. However the database structure can store the data and SPs can then enforce the constraints.

A *LoanInstrument* may have any number of *Commitments* that are summed to limit the amount available for *Drawdowns*. Each *Commitment* has begin and end dates; thus the total amount available varies over time.

The methods to delete, get, and sum *Commitments* do not vary across subclasses, so *LoanInstrument* specifies

them. In contrast, methods to add and update *Commitments* have constraints that vary, so the subclasses (*FacilityAgreement*, *Tranche*, and *TrancheItem*) individually specify them. Similarly, the *new*, *delete*, *update*, and *getWithID* methods also vary by subclass. By definition, each *TrancheItem* has a single lender that is assigned when a *new TrancheItem* is created. The *getLender* method retrieves a *TrancheItem*'s lender.

The *FacilityAgreement* *Commitments* limit *Drawdowns* for a borrower. The *Tranche* *Commitment* limits *Drawdowns* for a loan and the *TrancheItem* *Commitments* limit exposure for a lender to a loan. Ultimately, the sum of the *Commitments* for a *Tranche* cannot exceed the sum of

the *Commitments* for its *TrancheItems*. (The total funds from lenders must be able to cover the withdrawals of a borrower.) This is another constraint that SPs must check.

A **Drawdown** is a removal of funds within the scope of a *Tranche*. A *Tranche* can involve multiple *Drawdowns* that occur on various dates—the *drawDate* must lie within the begin and end dates for the *Tranche*. There are additional constraints. The total *Drawdowns* cannot exceed the *Commitment* for its *Tranche*. Furthermore, the total *Drawdowns* cannot exceed the total *Commitment* for its *FacilityAgreement* (traverse from *Drawdown* to *Tranche* and from the *Tranche* to *FacilityAgreement*).

A **DrawdownItem** is the portion of a *Drawdown* for a particular lender. Consequently, a *Drawdown* may be associated with any relevant *CounterPartyRoles*, except for lenders. The apportioning of a *Drawdown* to *DrawdownItems* is normally consistent with the splitting of its *Tranche* into *TrancheItems* (*manualOrComputed* = “computed”), but sometimes a *DrawdownItem* is manually adjusted (*manualOrComputed* = “manual”).

The methods can link a *Drawdown* to a *CounterPartyRole* (*bindCPR*), unlink a *Drawdown* from a *CounterPartyRole* (*unbindCPR*), and retrieve the collection of *CounterPartyRoles* for a *Drawdown* (*getCPRs*). There is a method to *apportion* a *Drawdown* to *DrawdownItems* consistent with the splitting of its *Tranche* into *TrancheItems*. The *apportion* method sets *manualOrComputed* to “computed”.

Reference [3] presents another application for which we used OO techniques to design stored procedures—soft-coded values that can handle miscellaneous data for objects.

#### 4. SP Naming

A large application can have many SPs and naming can quickly become an issue. Developers must be able to find SPs with the desired functionality, avoid duplication (SPs with the same logic and different names), and avoid name clashes. We found it helpful to organize SPs about class names, using the convention <prefix>\_<className>\_<operationName>.

- **Prefix.** The possibilities are *usp* (public SP) and *isp* (private SP) as the next section explains.
- **Class name.** We took class names from the UML model and did not abbreviate them.
- **Operation name.** An *operation* specifies behavior that applies across classes. Each pertinent class provides its own method to implement the operation. For brevity, some SPs have abridged operation names.

We found the SP naming convention to be effective, more than we had anticipated. We were readily able to find SPs and guess in advance what the name should be. With the convention, the SPs are clearly tied to the class model and can also be searched via the operation name. Good names

are important in that they improve understandability, documentation, and productivity.

#### 5. Public vs. Private

A **public** SP is available to other applications and the user interface. A **private** SP is only for the use of other SPs in the same application. In the syndicated loan application about 90% of the SPs were public and 10% were private. Figure 1 shows only public methods.

SQL Server does not distinguish between public and private. So we established a convention to indicate the difference. A prefix of *usp* (user stored procedure) denotes a public SP; a prefix of *isp* (internal stored procedure) denotes a private SP. Developers must be disciplined in adhering to this convention—*isp* SPs are solely for internal use.

Similarly, we distinguished between private views (prefix of *iv*) and public views (prefix of *uv*).

Private SPs are helpful in a number of situations. For example, superclass SPs that are overridden by subclasses should be declared as private. Similarly, computational artifacts are not part of the published interface and should also be declared as private.

We found that it was more difficult to determine the class owner for private methods than for public methods—this is not surprising. Public SPs have intrinsic meaning; private SPs are really just for implementation convenience.

#### 6. Coping with the Lack of Inheritance

**Inheritance** is a mechanism that organizes classes by their similarities and differences. An operation is invoked on an object and the language infrastructure automatically resolves the operation to a method in the inheritance hierarchy. Microsoft’s SQL Server language is not an OO language and consequently lacks inheritance and support for method resolution. As a result, client SPs must know where an operation is implemented in the class hierarchy and directly call the appropriate method.

We considered defining shell methods to effect method resolution. For example, we could define *usp\_FacilityAgreement\_deleteCommitment* and *usp\_Tranche\_deleteCommitment* as shell methods that call *isp\_LoanInstrument\_deleteCommitment*. But this would lead to many shell methods—it would roughly double the SPs for the syndicated loan application. Furthermore, we would need to maintain all the shell methods for parameter changes. We decided that the benefit of method resolution was not worth the additional clutter and maintenance hassle.

Our workaround for the lack of inheritance was to just directly call SPs at their particular location in the inheritance

hierarchy. We infrequently move methods within the hierarchy so direct calling has not been a problem. We have been using a simple tool that searches all SPs for occurrences of a string, so we can quickly find the callers of an SP.

We handle destructors differently. (A *destructor* is a method that deletes an instance of a class.) We use referential integrity and specify *on delete cascade* for each subclass relative to its superclass. So in essence, destructors are built into the fabric of the database structure. Whenever we need to delete an object that is described by an inheritance hierarchy, we just delete the root-level superclass record and let deletion cascade down the inheritance tree.

We have avoided multiple inheritance with SQL Server.

## 7. Error Handling

Our SPs have extensive error checking. In general, the presence of an error causes an SP to do nothing and return an error message. For convenience, we use strings instead of numeric error codes. The syntax for error strings is *className* + '\_' + *operationName* + '\_' + *errorMessage*. (We have tried to make error messages consistent across classes and operations.)

It is often convenient to define a transaction, tentatively write the data, check the data, and then rollback if necessary. The pseudocode in Figure 2 uses the UML's Object Constraint Language [5] to illustrate this approach. First update a *Drawdown*'s *drawdownAmount* and/or *drawDate*. Then get the *Tranche* (*T*) for the *Drawdown*. Then check a business rule for *T*—the total *drawdownAmount* cannot exceed the total *commitmentAmount* as of the *drawDate*.

```
BEGIN TRANSACTION
  update data for aDrawdown
  T := aDrawdown.Tranche
  check that
    sum(T.Drawdown.drawdownAmount) <=
    sum(T.Commitment.commitmentAmount)
    as of aDrawdown.drawDate
COMMIT or ROLLBACK TRANSACTION
```

**Figure 2 Error checking.** Transactions can ease error checking.

## 8. Substituting SQL Code for Programming

SQL is a powerful language and we were able to reduce development effort by substituting SQL code for programming code. SQL Server can flexibly mix stored procedures with views and functions.

- **User-defined functions.** User-defined functions behave the same as SQL Server's built-in functions. Both can be freely combined with SQL code. About the only restriction is that a function call cannot be an argument in a

stored procedure invocation. Instead, you must first evaluate the function and store it in a variable. Then you pass the variable as an argument.

As an example, several SPs perform currency conversion. The SPs include a function that converts an amount from a source currency to a target currency. The conversion function is called within a SQL statement.

- **Views.** Views can be handy for combining simple and aggregate data. For example a view could return *Tranche* data along with the total *drawdownAmount* as of a date.

We also used views to dynamically compute derived data. We found the performance to be reasonable, as long as we carefully wrote the views. We tested our views to verify their performance. Given acceptable performance, it is better to compute derived data as needed rather than store redundant data that must be kept consistent across updates to underlying base data.

- **Layers of views.** Layers of views can be helpful for effecting a sequence of calculations. For example, we wrote a series of views that computes *Commitment* time intervals for *LoanInstrument* and sums the applicable *commitmentAmounts* for each time interval.

These techniques reduce portability because they exploit SQL Server features. However, we believe the techniques could be ported to another DBMS.

As a matter of policy we try to minimize the use of cursors. Cursors increase the size of stored procedures, take longer to write, and are more error prone than SQL statements. Where possible, we place computation into a SQL statement rather than loop through a collection with programming code. Given the combination of SQL queries, user-defined functions, and views, we seldom need cursors.

Some authors have noted the drawbacks of nulls in databases [4]. We agree that nulls are troublesome, but it is not practical to entirely forego nulls. For example, nulls naturally arise from outer joins. We permitted nulls in our implementation.

## 9. Idiom for Record Sets

SQL code naturally deals with record sets—the inputs to an SQL query are tables and the result is a table. In contrast, programming code deals with variables and individual records. Consequently there is a mismatch between SQL and programming languages.

Stored procedures have ordinary input and output parameters that are intended for simple variables (integer, date, text, and so forth). So how can a stored procedure return a set of records? The SQL Server documentation is unclear about how to resolve this problem. We considered several possibilities.

- **Hardcoding.** Hard code the columns for a fixed number of records. For example, we could write an SP that takes

a *facilityAgreementID* as input and returns the *name*, *beginDate*, and *endDate* for each of its *Tranches*. With hardcoding, the output parameters would be *name1*, *beginDate1*, *endDate1*, *name2*, and so forth. This is a clumsy approach that we would only pursue as a last resort. It is awkward to limit results to a fixed number of records and the large number of output parameters would make the code verbose and difficult to read.

- **String of IDs.** Return many IDs folded into a long string. A separate function gets a record at a time using the ID. This is also a clumsy approach. The string length would limit the number of IDs and we would need to write additional SPs to manipulate ID strings.
- **XML string.** Store multiple records within an XML string. Once again the string length would limit the amount of data. Construction and parsing of the string would be straightforward.
- **Cursor.** SQL Server cursors are easy to use within an SP, but it is not obvious how to maintain a cursor across repeated SP calls. This does not appear to be a viable solution.

- **Record handshaking.** We could handshake on each output record, using a parameter to hold state. The SP returns one output record at a time as Figure 3 illustrates. For brevity we have omitted some error checking. This approach is a reasonable option and is similar to the cursor approach, but with a cursor SQL Server maintains state. With record handshaking, application code maintains state. Database caching would mitigate the performance impact of handling one record at a time.
- **Select query.** Define the SP as a *select* query. This works but has an odd syntax and is poorly documented. Figure 4 shows an example.
- **Callee temp table.** In our experiments, we could not create a *local* SQL Server temp table within an SP and have it persist across multiple invocations. (A *global* temp table works, but would be visible to other applications.)
- **Caller temp table.** The calling SP creates a *local* SQL Server temp table. The called SP knows the temp table name and stores the record set in the temp table for use by the calling SP. Figure 5 illustrates the idea.
- **Table-valued function.** SQL Server lets developers define a function that returns a table. Such a function is es-

```

/* Get the next Tranche for a FacilityAgreement (in ascending name order). */
CREATE PROCEDURE usp_FacilityAgreement_getNextTranche (
    @p_trancheID          INTEGER          OUTPUT,
    @p_name               VARCHAR(50)      OUTPUT,
    @p_beginDate          DATETIME         OUTPUT,
    @p_endDate            DATETIME         OUTPUT,
    @p_return             VARCHAR(50)      OUTPUT,
    @p_facilityAgreementID INTEGER,
    @p_priorTrancheID    INTEGER          = NULL ) AS

IF @p_priorTrancheID IS NULL BEGIN /* get the first Tranche */
    SELECT  @p_trancheID = T.trancheID, @p_name = T.name,
            @p_beginDate = T.beginDate, @p_endDate = T.endDate
    FROM Tranche AS T
    WHERE T.name IN (SELECT MIN(name) FROM Tranche
                    WHERE facilityAgreementID = @p_facilityAgreementID)
END
ELSE BEGIN /* get the next Tranche */
    SELECT  @p_trancheID = T1.trancheID, @p_name = T1.name,
            @p_beginDate = T1.beginDate, @p_endDate = T1.endDate
    FROM Tranche AS T1
    WHERE T1.name IN ( SELECT MIN(T2.name) FROM Tranche AS T2, Tranche AS T3
                      WHERE T2.name > T3.name AND T3.trancheID = @p_priorTrancheID AND
                        T2.facilityAgreementID = @p_facilityAgreementID )
END

IF @p_trancheID IS NULL BEGIN /* trying to retrieve, after the last Tranche */
    SET @p_return = 'FacilityAgreement_getNextTranche_priorTrancheIDisLast';
    RETURN
END

```

**Figure 3 Record handshaking.** A set can be retrieved one record at a time.

```
CREATE PROCEDURE usp_FacilityAgreement_getTranches (
    @p_facilityAgreementID          INTEGER ) AS

SELECT    trancheID, name,
          beginDate, endDate
FROM Tranche
WHERE facilityAgreementID = @p_facilityAgreementID
```

**Figure 4 Select query.** SQL Server can return a record collection.

```
/* If #temp_Tranches exists, get the Tranches for a FacilityAgreement.*/
CREATE PROCEDURE usp_FacilityAgreement_getTranches (
    @p_return          VARCHAR(50)          OUTPUT,
    @p_facilityAgreementID          INTEGER ) AS

IF EXISTS (SELECT * FROM tempdb.dbo.sysobjects
    WHERE id = object_id('[tempdb].[dbo].[#temp_Tranches]')) BEGIN
    DELETE FROM #temp_Tranches WHERE facilityAgreementID = @p_facilityAgreementID

    INSERT INTO #temp_Tranches
        (facilityAgreementID, trancheID, name, beginDate, endDate)
    SELECT facilityAgreementID, trancheID, name, beginDate, endDate
    FROM Tranche
    WHERE facilityAgreementID = @p_facilityAgreementID
END
```

**Figure 5 Caller temp table.** The SP stores records in a temp table with a predefined name.

essentially a parameterized view. Table-valued functions have restrictions. For example, they cannot update the database and cannot execute SPs. Also there is no provision to return an error code for an unexpected error.

Of the possible options, six are viable (string of IDs, XML string, record handshaking, select query, caller temp table, and table-valued function). We have been using the select query in most of our applications.

When possible, we imposed a meaningful order when retrieving a collection of records. For example, when retrieving the *Tranches* for a *FacilityAgreement*, we order the *Tranches* by name.

## 10. Idiom for Updates

We have also devised an idiom for updates. We want to be able to update an entire record or any combination of fields within a record, all with the same SP. We do this by defaulting the input to an ‘impossible’ value for each updatable field. Then we can detect genuine updates. Some examples of the ‘impossible’ values are:

- String = '@#%\$&'
- ID = -1
- Date = '1/1/1753'

We define a transaction and update each field, one at a time, within the transaction. For each field, we check the input value. If it is not the ‘impossible’ value, we update

the field—otherwise, we do nothing. The performance of these updates is acceptable, since there is a locality of reference for I/O (the database schema physically clusters an object’s values together on the disc).

For semantic reasons, we disallow update of some fields. For example, *usp\_Tranche\_update* cannot update *facilityAgreementID*.

For mass updates, an alternative is to process an XML file of bulk updates. Figure 6 demonstrates the approach.

## 11. Conclusion

We have found OO techniques to be helpful for designing database stored procedures. The benefits were greater than we had expected. OO techniques have improved the clarity of our thought, tightened the coupling between the database structure and the stored procedures, and improved our ability to understand, organize, and manage the resulting code mass. We recommend the following practices.

- **SP names.** Each SP implements a method and consequently belongs to a class. Therefore you should organize SP names about class names. The convention <prefix>\_<className>\_<operationName> is helpful.
- **Public vs. private.** OO languages distinguish public methods (freely accessible) from private methods (limited access) and SPs should do likewise. SPs have no intrinsic support for privacy, so we recommend the work-

```

CREATE PROCEDURE usp_Drawdown_updateMany (
    @drawdownInfo VARCHAR(1000) =
        '<Drawdowns>
        <Drawdown did="1" trancheID = "1" drawdownAmount="2455633"
        drawDate="1/1/2003" maturityDate="1/1/2010" />
        <Drawdown did="2" trancheID = "1" drawdownAmount="376599"
        drawDate="1/3/2003" maturityDate="11/7/2010" />
        <Drawdown did="3" trancheID = "1" drawdownAmount="8134431"
        drawDate="12/2/2004" maturityDate="1/1/2012" />
        </Drawdowns>' )
AS
DECLARE @hdoc INTEGER
DECLARE @tblDrawdownInfo TABLE (did INT, trancheID INT, drawdownAmount MONEY,
    drawDate DATETIME, maturityDate DATETIME)

/* Put the info into a temporary table variable */
EXEC master.dbo.sp_xml_preparedocument @hdoc OUTPUT, @drawdownInfo

INSERT INTO @tblDrawdownInfo (did, trancheID, drawdownAmount, drawDate, maturityDate)
SELECT did, trancheID, drawdownAmount, drawDate, maturityDate
FROM openxml (@hdoc, '/Drawdowns/Drawdown')
WITH (did INT, trancheID INT, drawdownAmount MONEY, drawDate DATETIME,
    maturityDate DATETIME) temp

/* Remove the internal xml representation. */
EXEC master.dbo.sp_xml_removedocument @hdoc

UPDATE D
SET drawdownAmount = TDDI.drawdownAmount,
    drawDate = TDDI.drawDate,
    maturityDate = TDDI.maturityDate
FROM Drawdown D
    INNER JOIN @tblDrawdownInfo TDDI ON TDDI.did = D.drawdownId

RETURN

```

**Figure 6 Bulk update using XML.** XML can handle bulk inserts and updates.

around of a prefix (such as ‘usp’ for public and ‘isp’ for private) to indicate intent. Software developers must have the discipline for adhering to the intent.

- **Inheritance.** SPs lack support for inheritance. Nonetheless, it is still helpful to think about inheritance during conceptual design. We found the best workaround to be direct calling of SPs and foregoing the use of polymorphism.
- **Substituting SQL code for programming.** SQL is a powerful language and developers should take advantage of it. Many DBMSs can flexibly mix SPs with views and functions, making it convenient to substitute concise SQL code for verbose programming code.
- **Idiom for record sets.** SPs have no natural mechanism for returning a record collection. We considered various alternatives and determined that string of IDs, XML

string, record handshaking, select query, caller temp table, and table-valued function are viable options.

- **Idiom for updates.** If you follow our suggestions, an update SP can update any combination of fields for a class.

## 12. References

- [1] Bob Beauchemin, Niels Berglund, and Dan Sullivan. *A First Look at SQL Server 2005 for Developers*. Boston: Addison-Wesley, 2004.
- [2] Michael Blaha and William Premerlani. *Object-Oriented Modeling and Design for Database Applications*. Upper Saddle River, NJ: Prentice Hall, 1998.
- [3] Michael Blaha. *Designing and Implementing Softcoded Values*. IEEE-CS ReadyNote, 2006.
- [4] C.J. Date. *Relational Database: Writings 1991-1994*. Boston: Addison-Wesley, 1995.
- [5] Jos Warner and Anneke Kleppe. *The Object Constraint Language*. Boston: Addison-Wesley, 1999.