

Getting comfortable with O-O

The introduction of object-oriented syntax in Version 10, Progress' Open Edge product, is the culmination of many years' effort and several previous attempts to add O-O capabilities to what is unequivocally the best business language ever built.

For those of us who have been married to the 4GL for many years, this new capability is a mix of appreciation and fear. Appreciation comes in knowing that Progress is staying abreast of contemporary programming standards regardless of whether we think they are an improvement over what we already know, or not. For example, SQL had been invented well before Mary Szekely started programming the 4GL. However, MIMS, the language that was the proving grounds for the founders of Progress, convinced them that SQL (which was not a standard then) was inadequate for complex business requirements such as recursive database queries, and the like. Thus, SQL was only reluctantly introduced to the language around version 5, and not fully supported until V9. By the same token, object oriented approaches were tried in 1989 and again in around 2000. In both cases, it was decided that there was not enough commercial demand for O-O coding structures.

In both cases, other environments, such as Oracle, SQL server, Java and .NET / C# have made these language components become standards in spite of themselves. So here we are, like it or not, facing the prospect of learning something new, once again.

But, if you've stayed abreast of most of the

developments of the last decade or so, you won't have a hard time getting started with the O-O ABL, or Object-Oriented Advanced Business Language. By the way, we think that the adoption of ABL as a moniker for the Progress language is a wonderful evolution beyond the 2-3-4GL path.

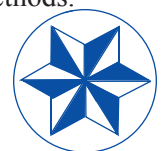
The vocabulary and concepts of object-oriented programming seem to have come straight out of academia, and at first don't seem to have a lot of relationship to real-world concepts, particularly the programming concepts we've been familiar with. However, with a little mental flexibility, and some loose comparisons with what we already know, you will find the "leap" from 4GL to ABL is not so great.

ABL as a moniker for the Progress language is a wonderful evolution beyond the 2-3-4GL path.

Classes

Let's start with the primary building block of the O-O environment: classes. A class is similar to a structured procedure in the 4GL. For those who aren't familiar with structured procedures, they are applications which are a bit more "restricted" than an "open" 4GL program. They have a definitions section, for variables, temp-tables, etc. And they can contain a block of functional code, the "main block". Otherwise, they can contain only internal procedures and functions.

If you remove the main block, and posit that a structured procedure could have only definitions, functions and procedures, you have a very close analogy to a class in the ABL. However, instead of procedures and functions, within a class you have methods.



A semi-formal description of a class would be something like: a body of code which has a state and behavior. So, this doesn't sound anything like a structured procedure. However, let's step back and reconsider a class as a code object. If one were to design an object, one would think that it would deal with one kind of "thing". Perhaps that thing might be a business object, such as a sales order and all its components, or perhaps more narrowly a single table and all of the operations which one would normally perform on it: create, delete, find, etc. Or perhaps the thing might be error management, or message handling, or security. And so on.

So what would a self-designed object look like? It would "contain" data, and the functionality to operate on that data. In other words, it might have a record buffer, (or, in the multi-tier world of today, an alternative thereto, such as temp-tables, prodatassets, etc.) And it would have the functionality, however structured, to add, delete, find, update said data.

This is approximately how a class is designed, with quite a bit more thought added to our simple concept. So, the state of the class is the data structures, and the behavior is the methods which are contained in the class.

So what does this mean about designing code? An awful lot of the work we all do is involved with some kind of legacy code. And we know how that was designed: as a good idea in the start, and mostly a bunch of willy-nilly additions to it over time. What does it look like now? Often, realistically, it is an amalgam of procedures hooked together rather tightly by means of parameters and shared structures such as variables and frames. Very rarely, such as in the case of the ADM model of Progress, there is a clear API or Application Programming Interface. However, it is much more rare in the day-to-day work we all so often do.

Consequently, if we were to start with the O-O ABL, we would have to begin anew with our thought processes. One would think more like architectural or molecular structures which are designed to fit together modularly from smaller, simpler structures. That is, a particular business class would need to have a clearly defined set of functionality, or methods, and APIs for those methods. Let's expand that a little. Let's say that we wanted to build a very finite, but standard set of methods into each business object. Suppose we start with the simplest

model, which is the best (perhaps only) way to start with a new programming paradigm. For example, a set of methods for maintaining a single "flat" file. Perhaps it is a control table, perhaps a table of message text. There are an awful lot of such maintenance programs in any robust application. And we have a pretty clear idea of what operations we perform on such a table. We browse records, find a specific one, create it update it, delete it. So we can envision what the object would look like. We would have a container for the records. Let's postulate a prodatasset. (For those not familiar with prodatassets, think, for this simple model, of a temp-table). We would have a corollary set of operations to populate the prodatasset; that is, moving data from the DB to the prodatasset, and back. This would include means of determining the lock and update status of any records removed from the DB, or about to be replaced. (Fortunately, prodatassets have a lot of this built in). In addition, we would have the operations we just spoke of. At first, conceptually, we would consider building one class per table we were going to maintain. However, this would proliferate code and make the same mess of our original good idea. So, most likely, we would move toward dynamic coding, and have one object to retrieve and replace data from the database, and one object to manage record functions (create, delete, find, etc). Later on, we might see the light and say that navigating a query is a single class object, while create, delete and update might be another. It doesn't matter how you end up, the point is you have to think much more modularly, a bit more abstractly, and certainly more clearly about how you're going to design code.

Let's take on some other O-O concepts in this article. One of the main ones we hear about is inheritance. Inheritance is really quite simple to understand for those who have been working with the contemporary 4GL. The 4GL has the concept of super-procedures, so for those familiar with super-procedures, inheriting another class is like adding a super-procedure to an existing 4GL program. For those not familiar with super-procedures, they are programs which contain internal procedures and functions. When a super-procedure is added to an existing program, it is as though those procedures and functions are "inserted into" the existing program. More formally, they are added to the "name space" of that program. What this means in the 4GL is the same thing as if the internal procedures were added at the front of the existing propath. Thus, inheritance means that all of the methods (again, think procedures or functions) of a

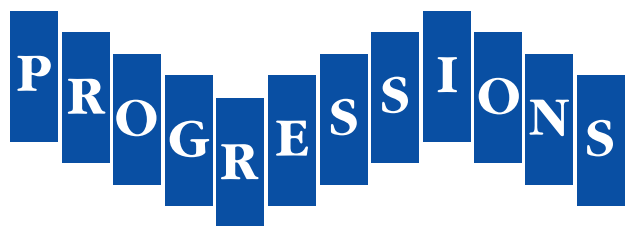
specific class (think structured procedure) are made “part of” the functionality of a class which inherits it. What this means is that we can compartmentalize functionality, such as query navigation, and whether we are dealing with temp-tables, prodatasets, or records, if we have a good, clear, flexible means of attaching a query to a given data object, (such as a dynamic query with handles), we can then place all of the query management in one class, and inherit that class in order to use all of its functionality in any data management application.

From practical experience, I know it’s possible to build a single application to create a prodataset, and another to navigate the prodataset, using a DB repository to feed these applications the name of the table, the fields which will be updated on-screen, etc. Thus, if a class were designed for a browser, another to instantiate fields, another to build the prodataset and another to navigate it, and yet one more for the record functions (create, etc), it would be feasible to have a half-dozen to a dozen classes to manage all flat-file management operations.

In subsequent articles, we will cover in more detail the specific syntax of a class, and how to build such an application scheme as we are talking about.

John Campbell

see page 14 for part two of this article (Classes: Syntax and structure)



Progressions

Winter Solstice:	John Campbell
Hanukkah:	Connie Campbell
Christmas:	Harriet Coates
April Fool’s Day:	Michael Bartlett

Progressions is prepared quarterly, which means 4 times a year...

Now that we are an electronic newsletter, we are offering the same price for everyone, no matter where you live! The annual subscription price is \$65.00 USD.

We accept Visa, Mastercard, and/or American Express (please include your expiration date and whether or not you have enough of a credit line for us to head for Katmandu or simply the border). All checks must be drawn on a US bank; checks and money orders must be in US dollars. In order to protect our contributors, the information in this periodical is copyrighted 1992-2007 White Star Software, Inc., and/or the author. All rights reserved worldwide and to the end of the cosmos.

Progressions

PO Box 250 Carbondale, CO 81623
970.963.3809 Voice 970.963.3548 Fax

Pessimistic locking on AppServer by Bogdan Brzowski

1. Introduction

In this article I would like to present a way to achieve *pessimistic locking* in a multi-tier application that uses the Progress AppServer. I presume that the reader knows the basics of the concurrency theory that lead us to the practical implementations known as *pessimistic locking* and *optimistic locking*. These basics are explained in ‘Concurrency in Progress Database’ article (Progressions, issue no 60).

When the user needs to amend any data in a database we need to provide him or her with the appropriate user interface and business logic. Our business logic tier should be aware of concurrency problems for example: the changes introduced by one user might be overwritten by another user. In general there are two strategies that allow us to avoid concurrency anomalies:

- Pessimistic locking database resources are locked exclusively whilst the data is being amended;
- Optimistic locking after data retrieval (in order to present the data in the user interface) all the locks are released. When the user commits the changes, all necessary database resources are locked again so the application can write the modifications into the database.

It is easy to achieve pessimistic locking in a traditional client-server application. We apply exclusive locks to the records that will be amended and after the user completes the modifications, we write the changed records to the database. When an exclusive lock is applied we can be sure that no other process will modify the same data, unfortunately during the modification time no other process can read the data with SHARE-LOCK or EXCLUSIVE options. Retrieving data with the NO-LOCK option is possible but this may result in an unreliable dataset as some of the data might have already been partially changed (especially when we are amending multiple records not just one).

Optimistic locking is a more sophisticated solution. Initially the data we want to amend is retrieved and presented in the user interface (for the purposes of this article, I don’t want to argue if this initial retrieval

should be done with either the SHARE-LOCK or NO-LOCK options). The retrieved data is held in internal buffers and is not being locked at all whilst the user is amending it. The main benefit is that during this time, other processes can retrieve this data in its pre-modified form, but when the user commits the changes things get more complicated:

- We must exclusively lock the data we want to amend. If it is not possible (because another process has also applied share or exclusive lock to the same data) then we have to dismiss the user’s changes;
- When the exclusive lock has been successfully applied we need to determine whether any other process has changed the same data that our user has amended. This requires a copy of the data before modification and this copy should be compared to the data that is exclusively locked before introducing the changes. If the copy is different then it means that another process has amended the same data in the meantime. If this is the case then we need to dismiss our modifications.

Optimistic locking offers better concurrency than pessimistic locking but sometimes it may result in dismissing the user’s work. This is why sometimes we may choose to use pessimistic locking, despite the poorer concurrency.

2. Pessimistic locking implementation

The presented solution presumes that the business logic tier (connected to appropriate database(s)) runs on Progress AppServer whilst the user interface is connected to the AppServer and is **not** connected to any relational database. Another presumption is that a given client session has one dedicated AppServer agent for itself. This solution is characteristic of AppServer operating in *State-reset* or *State-aware* mode and allows us to store the client session context in the AppServer session.

Transactions must be implemented on the AppServer side because the client side cannot directly access a database. The client side screen may include the following MAIN-BLOCK in its body:

```
MAIN-BLOCK:
DO ON ERROR UNDO, LEAVE
  ON END-KEY UNDO, LEAVE:
  RUN aslock.p PERSISTENT SET hLockerProc ON
  hAppServer.
  RUN initialiseScreen.
  ASSIGN
```

```

FRAME {&FRAME-NAME}:HIDDEN = FALSE
{&WINDOW-NAME}:HIDDEN = FALSE.
WAIT-FOR GO OF FRAME {&FRAME-NAME}.
RUN commitChanges.
END.
RUN unlockRecords IN hLockerProc NO-ERROR.
IF ERROR-STATUS:ERROR THEN
MESSAGE RETURN-VALUE
VIEW-AS ALERT-BOX ERROR.
DELETE PROCEDURE hLockerProc.

```

aslock.p is the AppServer side procedure (running persistently) that allows us to obtain pessimistic locking on AppServer. initialiseScreen is the procedure calling the aslock.p internal procedure in order to apply and hold exclusive lock(s) in the AppServer session. It also calls procedure(s) that retrieve the data and present this data on the screen in the form that allows updating the data. commitChanges is the procedure that writes the changes into the database and unlockRecords (from aslock.p) releases any exclusive locks applied to the records.

aslock.p has the following MAIN-BLOCK:

```

TRANSACTION-MODE AUTOMATIC.
DEF TEMP-TABLE ttLock NO-UNDO
FIELD hBuf AS HANDLE.

```

The first instruction (TRANSACTION-MODE AUTOMATIC) can be used inside a persistent procedure running on the AppServer that is called from the client session. This instruction starts a transaction within AppServer session so that while our automatic transaction is open, every procedure executed on AppServer takes part in this transaction. ttLock temp-table stores buffer handles for the records being locked within the transaction.

The next step is to apply exclusive lock(s) to the records that are subject to amendment. We can do it inside the initialiseScreen internal procedure:

```

IF VALID-HANDLE (phLockerProc) THEN DO:
cCond = "WHERE DB_TABLE.PRIMARY_KEY EQ " +
STRING(iPrimaryKey).
RUN lockRecords IN phLockerProc
( INPUT "EXCL",
INPUT "DB_TABLE",
INPUT cCond ) NO-ERROR.
IF ERROR-STATUS:ERROR THEN DO:
MESSAGE RETURN-VALUE
VIEW-AS ALERT-BOX ERROR.
RETURN ERROR.
END.
END.

```

The cCond variable stores an example condition used to retrieve the record(s) that should be locked. Then the lockRecords internal procedure from aslock.p is called:

```

PROCEDURE lockRecords:
DEF INPUT PARAM cLockMode AS CHAR NO-UNDO.
DEF INPUT PARAM cTableName AS CHAR NO-UNDO.
DEF INPUT PARAM cQueryCond AS CHAR NO-UNDO.

DEF VAR cMsg AS CHAR NO-UNDO.
DEF VAR lOK AS LOG NO-UNDO.
DEF VAR hBuffer AS HANDLE NO-UNDO.
DEF VAR hQuery AS HANDLE NO-UNDO.
DEF VAR cExpr AS CHAR NO-UNDO.
DEF VAR hLockedBuffer AS HANDLE NO-UNDO.

proc:
DO ON ERROR UNDO, RETRY
ON STOP UNDO, RETRY:
IF RETRY THEN DO:
IF ERROR-STATUS:ERROR THEN
cMsg = ERROR-STATUS:GET-MESSAGE(1).
ELSE
cMsg = "Unidentified error".
UNDO, RETURN ERROR cMsg.
END.
IF cTableName EQ ? OR
cTableName EQ "" THEN
UNDO, RETURN ERROR "Table name not
specified".
CREATE BUFFER hBuffer FOR TABLE cTable-
Name NO-ERROR.
IF ERROR-STATUS:ERROR THEN
UNDO, RETRY.
CREATE QUERY hQuery NO-ERROR.
IF ERROR-STATUS:ERROR THEN
UNDO, RETRY.
LOK = hQuery:SET-BUFFERS(hBuffer) NO-ER-
ROR.
IF ERROR-STATUS:ERROR THEN
UNDO, RETRY.
IF NOT LOK THEN
UNDO, RETURN ERROR "Cannot set buffer
for table " + cTableName.
IF cQueryCond EQ ? THEN
cQueryCond = "".
ASSIGN
cExpr = "FOR EACH " + cTableName + " "
+ cQueryCond
cExpr = TRIM(cExpr).
LOK = hQuery:QUERY-PREPARE(cExpr) NO-ER-
ROR.
IF ERROR-STATUS:ERROR THEN
UNDO, RETRY.
IF NOT LOK THEN
UNDO, RETURN ERROR "Cannot prepare
query: " + cExpr.
LOK = hQuery:QUERY-OPEN NO-ERROR.

```

```

IF ERROR-STATUS:ERROR THEN
  UNDO, RETRY.
IF NOT LOK THEN
  UNDO, RETURN ERROR "Cannot open que-
ry".
  hQuery:GET-FIRST(NO-LOCK).
  DO WHILE hBuffer:AVAIL:
    IF VALID-HANDLE(hLockedBuffer) THEN
DO:
  DELETE OBJECT hLockedBuffer NO-ER-
ROR.
  IF ERROR-STATUS:ERROR THEN
    UNDO, RETRY.
  END.
  CREATE BUFFER hLockedBuffer FOR TABLE
cTableName NO-ERROR.
  IF ERROR-STATUS:ERROR THEN
    UNDO, RETRY.
  IF cLockMode BEGINS "EXCL" THEN
    LOK = hLockedBuffer:FIND-BY-
ROWID(hBuffer:ROWID,
      EXCLUSIVE,NO-WAIT) NO-ERROR.
  ELSE
    LOK = hLockedBuffer:FIND-BY-
ROWID(hBuffer:ROWID,
      SHARE,NO-WAIT) NO-ERROR.
  IF ERROR-STATUS:ERROR THEN
    UNDO, RETRY.
  IF hLockedBuffer:LOCKED THEN
    UNDO, RETURN ERROR "Requested re-
source is locked".
  CREATE ttLock NO-ERROR.
  IF ERROR-STATUS:ERROR THEN
    UNDO, RETRY.
  ttLock.hBuf = hLockedBuffer NO-ERROR.
  IF ERROR-STATUS:ERROR THEN
    UNDO, RETRY.
  hQuery:GET-NEXT(NO-LOCK).
  END.
END.
RETURN.
END PROCEDURE . /* lockRecords */

```

In this procedure a good practice for AppServer programming is included such that every potential error will be trapped and an error message with the associated error condition returned to the client as RETURN-VALUE. This is achieved by executing every command with the NO-ERROR option (which makes sense as AppServer session cannot generate message-boxes) and then examining the ERROR-STATUS handle. If any error has occurred then UNDO, RETRY is executed resulting in an error condition and the appropriate message returned (inside IF RETRY ... condition). Buffer handles for all locked records are stored inside ttLock temp-table, this temp-table is useful when we need to release the locks (inside unlockRecords from

aslock.p).

After successful lockRecords execution (inside initialiseScreen) we can retrieve the locked records (as temp-table or ProDataSet) and put their values into the appropriate widgets in order to amend these values. We could also call lockRecords as the first command inside the AppServer procedure that retrieves the records to amend. In this way we can decrease the number of AppServer calls from the client session (from 2 to 1), which results in better performance.

When the user commits the changes the appropriate procedure is called (commitChanges for example). This procedure writes modifications back to the database. Because of the pessimistic locking strategy we do not need to check if any other process has changed the records changed inside our session pessimistic locking means that during the time of modification, the records have an exclusive lock applied.

The last thing to do is to release the locks and delete our aslock.p procedure (running persistently on AppServer). The contents of unlockRecords (inside aslock.p) may be the following:

```

PROCEDURE unlockRecords:

  DEF VAR cMsg AS CHAR NO-UNDO.
  DEF VAR LOK AS LOG NO-UNDO.

  PROC:
  FOR EACH ttLock
    ON ERROR UNDO, RETRY
    ON STOP UNDO, RETRY:
    IF RETRY THEN DO:
      IF ERROR-STATUS:ERROR THEN
        cMsg = ERROR-STATUS:GET-MESSAGE(1).
      ELSE
        cMsg = "Unidentified error".
      UNDO, RETURN ERROR cMsg.
    END.
  IF VALID-HANDLE(ttLock.hBuf) THEN DO:
    DELETE OBJECT ttLock.hBuf NO-ERROR.
    IF ERROR-STATUS:ERROR THEN
      UNDO, RETRY PROC.
  END.
  DELETE ttLock NO-ERROR.
  IF ERROR-STATUS:ERROR THEN
    UNDO, RETRY PROC.

  LOK = THIS-PROCEDURE:TRANSACTION:SET-
COMMIT() NO-ERROR.
  IF ERROR-STATUS:ERROR THEN
    UNDO, RETRY PROC.
  IF NOT LOK THEN

```

```

UNDO, RETURN ERROR "Cannot perform
SET-COMMIT method".
END.

```

```

RETURN.
END PROCEDURE . /* unlockRecords */

```

The important step is to execute the SET-COMMIT method in order to commit all our previous changes performed in the AppServer session within our automatic transaction. If we do not execute this method then our changes will be dismissed.

Bogdan Brzozowski

Bogdan has been working as a PROGRESS 4GL programmer for ten years. He's the Software Department Manager in NOVUM company, which develops systems for cooperative banks all over Poland. He started working with V7 then worked to adapt their application to the V9 environment. He specializes in relational database design (using UML or ER diagram methodology), protecting data integrity in both 4GL and SQL-92 environments and implementing database transactions. He was a speaker at the Progress Developers World 2003 conference in Dublin where he presented "Protecting Data Integrity in 4GL-based and SQL-based Applications." You can contact Bogdan at:

brbogdan@op.pl or bogdan@novum.pl

Books Books Books

If you are looking for books on
Progress,
we are your one stop shop!!

Check us out on the web at

WWW.WSS.COM
and click on publications

Replication Options Explored by Adam Backman

As long as people have stored data there has been a need to protect this data from disaster. Most hardware vendors engineer redundancy in their systems to provide protection from the most common problems, such as component failure. But this redundancy does not help if your computer room catches fire. This is where site replication comes into play.

Over the years, people in the Progress community have had to rely on third parties if they wanted to do replication; fortunately, this is no longer the case. In this article we will explore the options available to you for replicating your data from one database to another. This can be done in a number of ways, each with their own set of benefits, drawbacks, and general considerations.

Replication Overview

Since Progress has supported after image journaling people have been taking the after image files from one database and applying them to a second database periodically to replicate the database. This is known as **log-based replication**.

The benefits of this method are:

- Cost (Licensing costs - cheap for backing up only the server)
- Reliable (This method has been around and tested for years)
- Asynchronous replication

The drawbacks are:

- No real-time replication
- You need to maintain the code to manage it, since there is no formally supported product for this.

There are several hardware vendors that support the replication of data across machines. This is more than mirroring, which just provides protection from component failure. This **hardware replication** is generally very expensive and only as effective as the vendor behind the technology.

Benefits:

Easy setup
Easy maintenance

Drawbacks:

Cost
Potential for database corruption if writes are not guaranteed

Progress now supports synchronous and near synchronous replication with **OpenEdge Replication** (Formerly Fathom Replication). This form of replication applies after image journal entries to a second database at the time they are created by the database manager. While Progress does support synchronous replication, it is not a recommended mode, as performance is not satisfactory. It is my opinion that there are too many limitations to the replication product prior to version 10.1A to implement it in production

Benefits:

Supported product
Near realtime replication
Generally lower cost than hardware replication

Drawbacks:

More setup
More to monitor and maintain
Another set of utilities to learn

All high availability systems should have after-imaging implemented. If your database does not have after-imaging implemented please refer to the Progress documentation on how to properly implement this feature. The Progress documentation does a very good job of demonstrating how to implement after-imaging.

Log-base replication

An alternative to other methods is log-based replication, developed by White Star Software. White Star has had customers using this method for years. This form of replication takes after image files from the production (source) database and applies them to the replication target database. This method is not real time and the target database cannot be opened for reporting. The main benefit is the cost. If you are going to use

the target database for backup purposes only then you will only need a single user license of your OpenEdge product. If you are planning on using this system as a warm standby you will need the appropriate licensing to support that purpose. Other benefits of this method are its simplicity, reliability, extensibility (add reporting or email confirmation), as well as the maturity of the code.

In addition, you can determine the frequency of the replication. The more frequent the replication the greater the number of after image files you should have on your source database in case the replication fails. More ai files equal more time to fix the issue before it becomes critical. If you don't want to set this up yourself please feel free to drop us a line and we will be glad to help. If you do want to set it up yourself please follow the steps below and start in a test environment and move to your production database when the code is tested and you are comfortable with the process. The code is delivered as-is and is without warranty: please use caution. The lawyers should be happy now.

Setup of Log-based replication

Download the scripts from wss.com. The scripts support Unix and Linux systems and will require some modification to support your environment. You can use these scripts as a model for your own process or modify them as outlined in the README to support your environment.

Once the process is in place you will need to implement after imaging for your database if it is not already implemented. Please note: You should have enough after image extents to support an entire day of after image switches. Example: If you switch after-image files once an hour you should have at least 24 after image extents so you will have plenty of time to respond to a failure.

Your production database will be referred to as the source database and the replicated database will be referred to as the target database.

Seeding the target database to start the replication process will require backing up the source database. This can be done either online or off line if you use the Progress backup utility. If you use an operating system backup you will need to do the process while the database is off line and then mark the database as

backed up with the `rfutil` command (`rfutil <dbname> -C mark backedup`).

At this point, you will need to mark any “full” after image extents as “empty” with the `rfutil` command (`rfutil <dbname> -C aimage extent empty`). This command will need to be completed for each “full” extent.

Now you can restore the backup to the target machine. This database will have the same storage capacity requirements as the source with the exception of the after image files. The target database will not have after imaging enabled so they will not be needed. Once this step is complete you can start the replication process.

The replication process works as follows:

1. Mark the current busy after image extent as full and switch to the next extent
2. Cleanup archived after image extents older than 3 days on both local and target systems
3. Backup and compress the oldest full after image extent on the local machine
4. Move the oldest full after image extent to the target system
5. Apply the oldest full after image extent to the target database
6. Compress the oldest full after image extent on target system

Repeat steps 3-6 until there are no more full extents

The only maintenance is to monitor any logs

Books Books Books

If you are looking for books on
Progress,
we are your one stop shop!!

Check us out on the web at

WWW.WSS.COM
and click on publications

generated by the process on an ongoing basis.

Setup of OpenEdge Replication

Copy and modify the `source.repl.properties` and `target.repl.properties` files from the OpenEdge replication installation directory. These files will be placed in the database directory on the source and target machine(s). The names of the files will be changed to `<dbname>.repl.properties` when they are moved into place.

Sample `source.repl.properties`

```
[server]
  control-agents=agent1
  database=source
  transition=manual
  transition-timeout=600
```

```
[control-agent.agent1]
  name=agent1
  database=source
  host=moth
  port=4501
  connect-timeout=120
  replication-method=async
  critical=0
```

Sample `target.repl.properties`

```
[agent]
  name=agent1
  database=source
  listener-minport=4387
  listener-maxport=4500
```

1. Shut down the source database
2. Move the properties files into the database directories on the source and target machines
3. Enable site replication on the source database (`proutil <dbname> -C enablesitereplication source`)
4. Backup the database. Use Progress backup or OS utility and `rfutil` to mark the database as backed up.
5. Restore the backup on the target machine
6. Enable site replication on the target (`proutil <dbname> -C enablesitereplication target`)
7. Start the source database (`proserve <dbname> -DBService replserv`)
8. Start the target database (`proserve <dbname> -DBService replagent -S replication_port_number`)

The replication port number is derived from the `<dbname>.repl.properties` file on the source system.

This would be 4501 from the example file above.

This can now be monitored with the dsrutil utility that is included in the OpenEdge replication product.

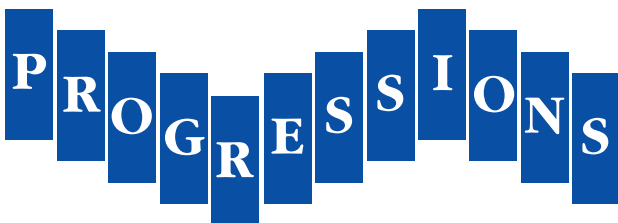
I hope this article encourages you to look at your options to better protect your data. You have many options across many price ranges so consider the value of your data and develop a strategy that make sense for your business. At White Star Software, we help people develop and maintain these plans every day so if you have questions please feel free to drop us a line, we would be glad to help.

Adam Backman

Adam has over 20 years of experience in the software field. He worked at Progress Software Corporation for 8 years in various capacities. Since 1995, Adam has run the Database Performance Tuning and Administration division of White Star Software where he provides international training, consulting, and writing services centered on Progress databases and infrastructure. He has been a presenter at Progress Software conferences and other events since 1992.

Adam Backman specializes in Database Administration Training, Performance Tuning, UNIX/Linux Administration Training, Hardware Configuration, Disaster Recovery Planning, Database and System Administration, Administrator Mentoring, and System Migration.

You can reach Adam at: adam@wss.com
603.897.0297



Look Ma, Power XML by Paul Guggenheim

In Version 9, Progress provided the developer with basic and detailed 4GL features for handling xml files. The CREATE-NODE and APPEND-CHILD methods allow the programmer to construct a sophisticated, hierarchical xml tree that can represent records for a given table and fields.

Our first program example, ethnicwritexmlv9.p uses the ethnic table in the school database consisting of the 2 fields ethnicid and description. It builds the xml tree in memory with CREATE-NODE and APPEND-CHILD methods inside the for each ethnic block. The save method on the document handle writes the xml information to the desired file

```
/*ethnicwritexmlv9.p - create xml document
   from ethnic table with formatting */
```

```
define variable hdoc as handle.
define variable hroot as handle.
define variable hformat as handle.
define variable hrow as handle.
define variable hfield as handle.
define variable htext as handle.
define variable hbuf as handle.
define variable hdbfld as handle.
define variable i as integer.
```

```
/* create the necessary objects. */
create x-document hdoc.
create x-noderef hroot.
create x-noderef hrow.
create x-noderef hfield.
create x-noderef htext.
create x-noderef hformat.
```

```
/* get a buffer for the ethnic table. */
hbuf = buffer ethnic:handle.
```

```
/* set up a root node. */
hdoc:create-node (hroot, "ethnics", "element").
hdoc:append-child (hroot).
```

```
for each ethnic:
```

```
  hDoc:CREATE-NODE (hformat, ?, "text").
  hformat:node-value = "~n~t".
  hRoot:APPEND-CHILD (hformat).
```

```
/* create a ethnic row node. */
  hdoc:create-node (hrow, "ethnic", "element").
  /* put the row in the tree */
```

```

hroot:append-child (hrow).

/* Treat ethnicid as an attribute of this
element.
The remaining fields will be created as
elements. */
hrow:set-attribute ("ethnicid", string
(ethnicid)).

hDoc:CREATE-NODE (hformat, ?, "text").
hformat:node-value = "~n~t~t".
hrow:APPEND-CHILD (hformat).

/* add the other fields as elements. */
repeat i = 1 to hbuf:num-fields:
  hdbfld = hbuf:buffer-field (i).

/* we already did ethnicid above so skip
it. */
if hdbfld:name = "ethnicid" then next.

hDoc:CREATE-NODE (hformat, ?, "text").
hformat:node-value = "~n~t~t".
hrow:APPEND-CHILD (hformat).

/* create an element with the field name
as the tag.
Note that the field name is the same as
the element
name. The rules for allowed names in
xml are less
stringent than the rules for progress
column names. */
hdoc:create-node (hfield, hdbfld:name,
"element").
hrow:append-child (hfield).

/* create new field to hold data of
datafield as child of datafield. */
hdoc:create-node (htext, "", "text").

/* node to hold value. */
hfield:append-child (htext).

/* assign data to field with text node */

if hdbfld:buffer-value = ?
then
htext:node-value = "?".
else
htext:node-value = string (hdbfld:buffer-
value).

end. /* repeat i = 1 to hbuf:num-fields */
end. /* for each ethnic */

/* write the xml node tree to an xml file. */
hdoc:save ("file", "ethnicv9.xml").

/* delete the objects. note that deleting
the document
object deletes the dom structure under it

```

```

also. */

delete object hbuf.
delete object hdoc.
delete object hroot.
delete object hrow.
delete object hfield.
delete object htext.

To read this xml file back into memory, our second
example, ethnicreadxmlv9.p, uses the load method to
read the previously saved xml file back into memory.
The program then uses the get-document-element and
get-child methods to traverse the tree and display the
xml information.

/* ethnicreadxmlv9.p - read xml file back in
and display data */

def var hdoc      as handle.
def var hroot    as handle.
def var htable   as handle.
def var hfield   as handle.
def var htext    as handle.
def var recno    as int.
def var i        as int.
def var j        as int.
def var k        as int.
DEF VAR tablename AS CHAR FORMAT "x(15)".
def var infilename as char label "Input File"
format "x(25)" init "ethnicv9.xml".

def var keyfield  as char format "xxx".
def var fieldname as char format "x(15)".
def var fieldvalue as char format "x(25)".

create x-document hdoc.
create x-noderef hroot.
create x-noderef htable.
create x-noderef hfield.
create x-noderef htext.

repeat:

hide frame a.

update infilename.

hdoc:load ("file", infilename, false).
hdoc:get-document-element (hroot).

define frame a
tablename keyfield fieldname fieldvalue
with down no-box no-labels stream-io.

recno = 0.
output to terminal paged.
do i = 1 to hroot:num-children with frame a
stream-io:

```

12

```
hroot:get-child (htable, i).

/* filter out text formatting */
if htable:subtype ne "element" then next.

tablename = htable:NAME.

recno = recno + 1.

display recno @ keyfield tablename.

/* read field names */
do j = 1 to htable:num-children with frame
a:

    htable:get-child (hfield, j).

    /* filter out text formatting */
    if hfield:subtype ne "element" then next.

    display hfield:name format "x(15)" @
fieldname.

    /* should be just one child for value */
    do k = 1 to hfield:num-children with
frame a:
        hfield:get-child (htext, k).
        display htext:node-value format
"x(25)" @ fieldvalue.
        down.
    end. /* k = 1 to ... read field values
should be only one value */

    down.
end. /* j = 1 to ... read field names */

page.

end. /* do i = 1 to ... read records */
output close.

end. /* repeat */
```

At the time, the developer community was thrilled to have the ability to handle xml files, even though several lines of code had to be written to handle the xml processing. Enter OpenEdge and ProDataSets. Because of the complex nature of ProDataSets, more powerful 4GL constructs needed to be provided. 4GL developers now have the ability to read and write xml data and schema from temp-tables and ProDataSets in a single method statement.

In ethnicwritexml.p, we have cut down a 95 line program to 14 lines!

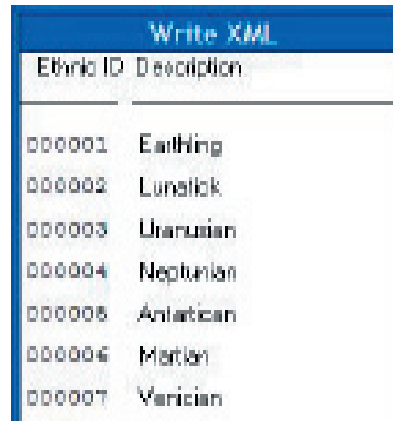
```
/* ethnicwritexml.p */

define temp-table tethnic like ethnic.
```

```
for each ethnic with title "Write XML":
    create tethnic.
    buffer-copy ethnic to tethnic.
    display tethnic.
end.
```

```
buffer tethnic:write-xml("file","ethnic.
xml",true /* formatted */).
buffer tethnic:write-xmlschema("file","ethnic
schema.xml",true /* formatted */).
```

```
delete object tethnic.
```



Ethnic ID	Description
000001	Earthling
000002	Lunatic
000003	Uranian
000004	Neptunian
000005	Antarian
000006	Martian
000007	Verician

It actually creates two xml files, one for schema

```
<?xml version="1.0" ?>
_ <xsd:schema xmlns:xsd="http://www.
w3.org/2001/XMLSchema" xmlns="" xmlns:
prodata="urn:schemas-progress-com:xml-pro-
data:0001">
_ <xsd:element name="tethnic" prodata:
proTempTable="true" prodata:undo="true">
_ <xsd:complexType>
_ <xsd:sequence>
_ <xsd:element name="tethnicRow" minOc-
curs="0" maxOccurs="unbounded">
_ <xsd:complexType>
_ <xsd:sequence>
<xsd:element name="ethnicId" type="xsd:int"
nillable="true" prodata:format="999999" pro-
data:label="Ethnic ID" />
<xsd:element name="Description" type="xsd:
string" nillable="true" prodata:
format="x(25)" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
_ <xsd:unique name="description">
<xsd:selector xpath="//tethnicRow" />
<xsd:field xpath="Description" />
</xsd:unique>
_ <xsd:unique name="ethnicId" prodata:
primaryIndex="true">
<xsd:selector xpath="//tethnicRow" />
```

```
<xsd:field xpath="ethnicId" />
</xsd:unique>
</xsd:element>
</xsd:schema>
```

and one for data

```
<?xml version="1.0" ?>
_ <tethnic xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance">
_ <tethnicRow>
<ethnicId>1</ethnicId>
<Description>Earthling</Description>
</tethnicRow>
_ <tethnicRow>
<ethnicId>2</ethnicId>
<Description>Lunatick</Description>
</tethnicRow>
_ <tethnicRow>
<ethnicId>3</ethnicId>
<Description>Uranusian</Description>
</tethnicRow>
_ <tethnicRow>
<ethnicId>4</ethnicId>
<Description>Neptunian</Description>
</tethnicRow>
_ <tethnicRow>
<ethnicId>5</ethnicId>
<Description>Antartican</Description>
</tethnicRow>
_ <tethnicRow>
<ethnicId>6</ethnicId>
<Description>Martian</Description>
</tethnicRow>
_ <tethnicRow>
<ethnicId>7</ethnicId>
<Description>Venician</Description>
</tethnicRow>
</tethnic>
```

which use the `write-xmlschema` and `write-xml` methods respectively. Notice that the `true` option is used in both methods. This formats the xml output with carriage returns and tabs so it can be easily read.

In `ethnicreadxml.p`, we now only use 32 lines versus 81 for version 9.

```
/* readethnic.p */

define var thand as handle.
define var tbuf as handle.
define var qh as handle.
define var fh1 as handle.
define var fh2 as handle.

create temp-table thand.

thand:read-xml("file","ethnic.xml","merge","e
thnic.schema.xml",false).
```

```
tbuf = thand:default-buffer-handle.
```

```
create query qh.
```

```
qh:set-buffers(tbuf).
```

```
qh:query-prepare("for each " + tbuf:name).
qh:query-open().
qh:get-first().
```

```
do while tbuf:available with down title
"Read XML":
    display tbuf:buffer-field("ethnicid"):buf-
fer-value
        label "Ethnic ID" format "999999"
        tbuf:buffer-field("description"):
buffer-value
        label "Description" format
"x(25)".
    qh:get-next().
end.
```

```
delete object qh.
```

```
delete object thand.
```

Read XML	
EthnicID	Description
1	Earthling
2	Lunatick
3	Uranusian
4	Neptunian
5	Antartican
6	Martian
7	Venician

Not only does it read the xml file and display the information, it also loads the information into a temp-table (or a ProDataSet). Make sure the `override-default-mapping` is set to `false` in the `read-xml` method. The `true` value creates a temp-table with CLOB and BLOB fields which cannot be components of indexes. (As of this writing, the `true` value causes the Progress session to end abnormally. Progress is aware of this and is working to fix it for version 10.1B)

In summary, Open Edge offers powerful new xml methods for reading and writing xml files into temp-tables and ProDataSets. However, the developer still has the detail tools from version 9 should the application require them.

Paul Guggenheim

Paul Guggenheim & Associates was founded in 1984. Paul has been training Progress students since 1986. He has developed 6 comprehensive high-quality Progress training courses and provides both public training classes in the Chicago area and private training classes on-site.

In addition, Paul provides clients with consulting services that include database design, web integration, performance tuning, ADM2 application development and GUI programming. Custom solutions include hand-held, WebClient, and graphical applications.

Clients range from industries such as manufacturing, distribution, banking, health care, insurance and non-profit.

Paul has spoken at several regional, national and international conferences and has published articles in Progressions. Paul is also the chairperson of the Chicago Area Progress Users Group.

Paul Guggenheim & Associates, Inc.
1788 Second Street, Suite 201
Highland Park, IL 60035
Phone: (847) 926-9800
Fax: (847) 926-9805
E-mail: paul@pgasmarts.com

Classes: Syntax and structure by John Campbell

Classes are very similar to structured procedures. They contain methods, described below. Their suffix, unlike a .P, is a .CLS. They are quite structured. This requirement demands that their purpose be clearly articulated, and that they be well-designed. Also, unlike procedures, if they refer to other classes, they must refer to them in a specific place within propath. This structure is a benefit in terms of clear implementation, but something of a bane to long-time Progress programmers, who are used to being able to manipulate propath at will. You still can manipulate propath, in order to see different versions of a class, but the propath issues become a bit more complex and structured.

Structure

You can have a definitions section, as well as a simple block of code; otherwise, all of the content of classes are methods. So, a simple class might look like the following:

```
class Item:
  define temp-table tItem
  field tItemNum like item.itemnum
  field tItemName like item.itemname
  field tOnHand like item.Onhand
  field tPrice like item.Price.

  define query qItem for item.

  define variable hQuery as handle.
  define variable WherePhrase as char.
  define variable ByPhrase as char.
  define variable QueryPhrase as char.

  method public void SetQuery():
    hQuery = query qItem:handle.
  end.

end class.
```

This is simply an example of how a class would look, rather than a fully functional class.

Instantiating a class

First of all, a variable has to be defined to point to the class. This is a bit like defining a handle to point to a persistent procedure. Then, the class has to be instantiated. This is a bit like running the procedure and pointing the handle to it. Once it is instantiated, its

methods can be invoked.

Here is the syntax for instantiating a class:

```
define variable ItemClass as class item no-undo.
```

```
ItemClass = new item().
```

This would be similar to the following in the 4GL:

```
define variable hItem as handle no-undo.
```

```
run item.p persistent set hItem.
```

Methods: review, syntax and invocation

A method is similar to an internal procedure or function within a structured procedure. However, it does have some differences. For example, a method declares whether its information is public, private or protected, and whether it returns a value or not. A **public** method is available to the class that defined it, all those classes which inherit it, as well as to other classes which references the defining class. A **protected** method is available to the defining class and all those which inherit it. A **private** method is available only to the defining class, just as there are private functions and internal procedures in a structured procedure. Although the analogy is imperfect, think of the difference between global variables, new shared variables, and local variables, and apply this concept to a procedure or function, and you have a rough idea of the difference in these declaration modifiers for a method.

Method return-types and parameters

Just as a function or procedure can return a value, so can a method. A **void** method is declared if it does not return any value. (see the above example for the syntax of a void method). Otherwise, a class can return any of the standard datatypes, as well as another class. (We'll avoid how this works, for now to keep things simple).

So, in the example above, since the method setquery does not return any value, it is declared as void. Also, it does not take any input values. A method which returns a value as well as having an input parameter would look like the following:

```
class item:
method public decimal ReturnPrice
(input ptItemNum as int):
```

```
for first tItem
where tItemNum = ptItemNum no-lock:
return tItem.tPrice.
end.
return ?.
end method.
end class.
```

In this example, the method is public, so it is available to any class which can access this class. It returns a decimal value, so its “return-type” is declared as such. In addition, it takes pItemNum as an input parameter.

Here is an example of invoking the method for ReturnPrice.

```
define variable ItemClass as class item no-undo.

ItemClass = new item().

display ItemClass:ReturnPrice(10).
```

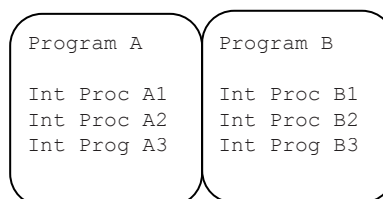
Not so tough, once you've worked a bit with handles and functions.

Inheritance

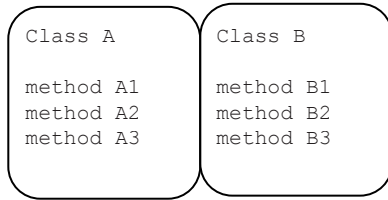
Review

In the article in the previous issue of Progressions, we discussed the concept of inheritance. As mentioned, inheritance works a bit like adding a super-procedure to an existing procedure.

Let's take a look at a diagram to make this clearer. Suppose you have a program (program A) with a series of internal procedures. And, we have another program (B) with its own set of internal procedures. When B is attached to A as a super-procedure, A can call any internal procedure in B without any more syntax than “run”. For example, in A, one can say “run B1” if B is a super-procedure.



By the same token, classes which inherit another class, also “inherit” the ability to call any method within the inherited class without any additional syntax.



Syntax:

A statement in class A, if it inherited class B, could invoke a method in class B as if it were “in” class A.

```

class A inherits B:
end class.
+++++
define variable ClassA as class item no-undo.

ClassA = new A().

display ClassA:B1().
  
```

This is fairly abstract, so let’s talk about a more concrete example. We’ll start with something pretty simple, so the mechanism is clear.

First, class B:

```

class B:

method public void ViewMessage
(pMessage as char,pType as char):
case pType:
when "error" then
message pMessage view-as alert-box
error.
when "warning" then
message pMessage view-as alert-box
warning.
otherwise
message pMessage view-as alert-box.
end case.
end method.

end class.
  
```

Nothing fancy: only one class whose “job” it is to present a message box with a message type (error, warning, etc) and some message text.

Now, suppose that class A inherits B:

```

class A
inherits B:
define variable MessageText as char.

method public void CallMessage
(PmesgType as char):
ViewMessage (MessageText,pMesgType).
end method.

method public void SetMessageText
(pMessageText as char):
MessageText = pMessageText.
end method.

end class.
  
```

In this class, there is a method to set the message text, and one to call the message handler. We’ll look at this in more detail in a moment. Our last piece of code is the main program that instantiates the “parent” class and takes action.

```

define variable ClassA as class A no-undo.

ClassA = new A().

ClassA:SetMessageText("Item not found").

ClassA:CallMessage("error").
  
```

First, our calling program instantiates class A. Then, we invoke the method “SetMessageText” to set the text for the message. Next, we invoke the method (in A) to call the message, with the message type of Error. Note that there is no reference to class B here.

The key to inheritance is that when the method CallMessage is invoked in A, it can automatically invoke the method ViewMessage in B without any other reference to B (no class variable, no instantiation).

Now, let’s talk about a more practical, although more complex, example for the real world.

Suppose you were using dynamic prodatasets to manipulate data. (If you are not yet familiar with prodatasets, any time you encounter the word prodataset, simply replace it with the word temp-table). Next, suppose you had a class whose job it was to create and build a particular dynamic prodataset. Suppose you had another class whose job it was to create a dynamic

query, and navigate that query. And, suppose you had a program whose job it was to create a dynamic browse for a specific table, and then invoke a dynamic query, which used a dynamic prodataset, to display the records for any table: customer, order, item, etc. You can do all of this using inheritance.

Here's how it would work. The class for the dynamic query would inherit the class for the dynamic prodataset. The calling program (our browser) would invoke the dynamic query, which automatically "invokes" (instantiates) the code for the dynamic prodataset. Then, as the primary program knows which table and fields to display in the browser, it simply has to make the invocations to the dynamic query program, and not worry about the prodataset at all.

The benefit of object-orientation is that code components can be very specific in their functionality,

and designed to "fit" together very seamlessly. This also lends to being able to modify one component without breaking the others which connect to it.

The downside to this architecture is that it takes a great deal more time to design such objects well: flexible and modular, and at the same time maintainable. 4GL programmers who are used to building code by simply getting started, and evolving their thought process as they write their code, will find that this "old" mindset no longer works. The code structure has to be designed before it can be coded, rather than after.

John Campbell



JARGON™

- **Develop powerful Wireless applications in our new V3 Jargon Writer!**
- **Run wireless apps either online or offline when out of coverage area**
- **Create "bolt-on" solutions to your Progress app in days, not months.**
- **Integrate barcode scanners, printers and other wireless peripherals**
- **Deploy with AppServer or WebSpeed. Save money with AppServer.**
- **Enjoy easy, powerful wireless deployment for under \$150 per handheld.**
- **Leverage your 4GL experience - no need to master Java, C++, or HTML.**
- **Download a FREE Evaluation Copy or contact us for a personal online demo**

JARGON SOFTWARE Inc
 708 North First Street, Suite 432
 Minneapolis, Minnesota 55401
 612 338 1175 | fax 612 338 2974
www.jargonsoft.com (On-line demo!)
info@jargonsoft.com

All brand and product names are the property of their respective owners.



Table of Contents

Pleasure is very seldom found where it is sought; our brightest blazes of gladness are commonly kindled by unexpected sparks.

-Samuel Johnson

Pessimistic locking on AppServer	Bogdan Brzozowski 4
Replication Options Explored	Adam Backman 7
Look Ma, Power XML	Paul Guggenheim..... 10
Classes: Syntax and structure	John Campbell 14

GOT HEADACHES?

Do Any of these Describe You?

You have a new Progress-based package you're implementing but want some customizations.

The users of your existing application are asking for numerous modifications.

You want to upgrade; you've made customizations; you don't want to spend a fortune "going back" to vanilla code but you don't want to lose what you've implemented.

Do You Need to:

Add or remove some fields in an existing screen, change some labels or defaults, or even rewrite the screen entirely...

Add your own tracking screens in the middle of the application flow...

Put your own messages, validation or extended business rules into the application...

Add fields or indexes to existing tables...

Get better rollup information, and get it in real-time...

TailorPro is the solution to all these problems, and many more. It is a Progress-based application which allows you to insert your own application logic anywhere in your current programs. It enables you to add fields not tracked by your application, and perform data entry to them in the "middle" of existing screens. No source code or DB changes required!

TailorPro Saves You Money by:

Cutting operating costs in your company by making your users more efficient.

Dramatically reducing the implementation costs for custom application components.

Eliminating the future costs of upgrading a customized application.

Look into the *TailorPro* features on our new website: **www.tailorpro.biz**

voice: 970.963.3809

fax: 970.963.3548