Surviving Client/Server: Credit Payments, Part 1

by Steve Troxell

About a year ago I had to develop a fairly involved credit accounting and payment system. The business rules of the system were quite novel and posed a few challenges. I thought it would be informative to share that experience with you and point out the reasoning behind some of the design choices. While you're not likely to encounter this exact situation yourself, many of the concepts can be generalized and may be helpful in learning client/server techniques.

The Problem

The situation is this: any customer may take out a credit slip for any amount, subject to caps on their available credit limit. The customer will repay the credit in full or make several partial payments until the balance is cleared. So far, this is fairly typical. The complicating factors are that a customer may have any number of credits outstanding at the same time and a single payment may be used to pay off more than one credit. For example, if I have a $1000 credit and a $2500 credit, I can make one payment of $3500 to cover them both. The payment is recorded as a single payment, but is applied to two outstanding credits.

In addition, a payment may consist of several forms of currency: cash, personal check, cashier's check, etc. The form, or method, of payment must be recorded separately. Therefore, we have a single payment with two dimensions to it: it may span one or more credits and may consist of one or more payment methods. Also, the allocation of each portion of the payment made against each credit must be accounted for. Back to the previous example with my $1000 and $2500 credits, I may make a payment of $3500 consisting of $900 in cash, $2100 in the form of a personal check and $500 in the form of a cashier's check. I wish to pay off the $1000 credit with $500 in cash and $500 from the personal check, with the remainder of the payment going against the $2500 credit ($400 in cash, $1600 from the personal check and the $500 cashier's check). All of this must be tracked.

If you think this is complicated, I've thrown out several other factors to simplify this from the real life case. The demo application described in this article is spartan by necessity. My intent is to focus on the overall techniques and principles. I have not devoted much space to fleshing out the application with full data validation and complete user interface dialogs. The Local InterBase demo application and database can be found on the disk in the SURVIVE directory.

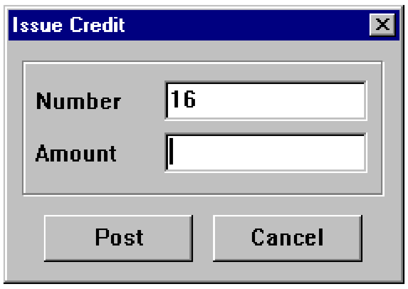
We'll discuss three main areas of this system: issuing the credit slips, recording payments to credits and reporting payment activity. Recording payments is by far the most complex element and we'll devote more space to that. From our analysis and prototyping, we've come up with the simplified screens shown in Figures 1 through 3. Of course, in a real system there would be much more detail than is shown here. Also, I've omitted the obvious function of selecting the customer before entering these screens.

Issuing Credits

Issuing the credit slip is a fairly straightforward and mechanical process so the dialog shown in Figure 1 serves our purpose of simply getting something into the database so that we can move on to the task of recording payments against credits. Listing 1 shows the table we will use to record credits. However, there is one interesting technique here worth exploring. The number used to uniquely identify the credit in the system is essentially an auto-increment field. However, we must display the number to be used in the dialog when it first appears, before we've posted the credit to the database and obtained the automatically generated number.

In addition, since credit numbers are tightly monitored by the business, we would like to account for all credit numbers used, even if the transaction was cancelled. For example, if we've brought up the `Issue Credit` dialog and obtained a credit number, then decided to cancel the dialog, we must account for the credit number that was "burned" since it cannot be reused. Therefore, if we find any gaps in the sequence of credit numbers, we have a good indication that someone has been tampering with the system (by issuing a credit then

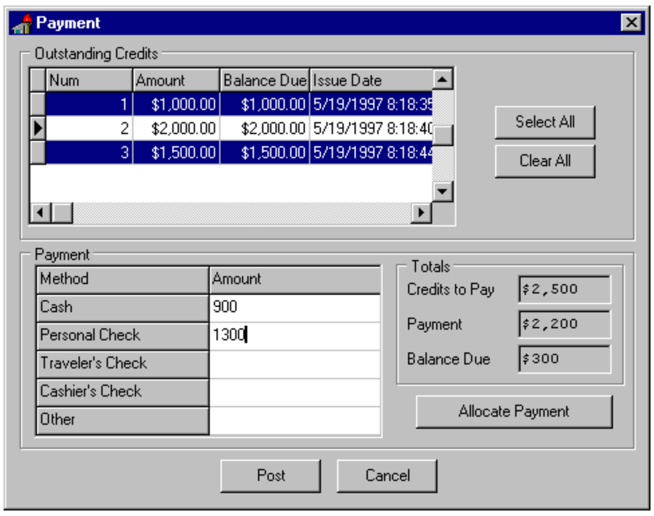
➤ *Figure 1*



Number: 16

➤ *Listing 1*

```
create table Credits
( CreditNo        integer not null primary key,
  Status          char(1) default 'V' not null,
  CustNo          integer not null,
  Amount          float default 0 not null,
  BalanceDue      float default 0 not null,
  IssueDateTime   date default 'now' not null );
```


Payment

Outstanding Credits

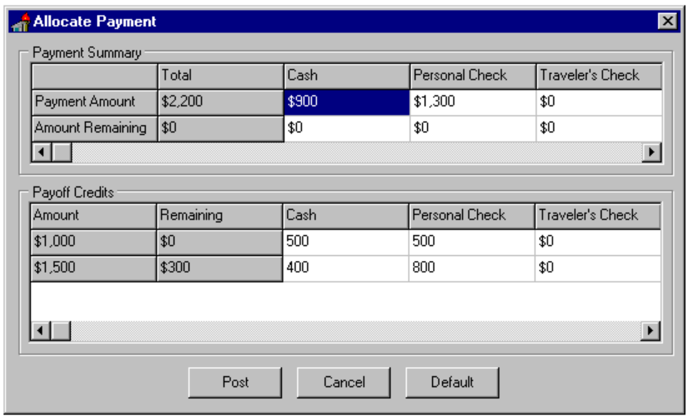
	Num	Amount	Balance Due	Issue Date
	1	$1,000.00	$1,000.00	5/19/1997 8:18:35
▶	2	$2,000.00	$2,000.00	5/19/1997 8:18:40
	3	$1,500.00	$1,500.00	5/19/1997 8:18:44

Payment

Method	Amount
Cash	900
Personal Check	1300
Traveler's Check	
Cashier's Check	
Other	

Totals

Credits to Pay	$2,500
Payment	$2,200
Balance Due	$300


Allocate Payment

Payment Summary

	Total	Cash	Personal Check	Traveler's Check
Payment Amount	$2,200	$900	$1,300	$0
Amount Remaining	$0	$0	$0	$0

Payoff Credits

Amount	Remaining	Cash	Personal Check	Traveler's Check
$1,000	$0	500	500	$0
$1,500	$300	400	800	$0

➤ *Left: Figure 2* ➤ *Above: Figure 3*

destroying all trace of it from the system so that it doesn't have to be repaid).

Here's a solution to this dilemma. Rather than use a true auto-increment field for the credit numbers, we use an InterBase generator (also because InterBase doesn't have true auto-increment fields). When the `Issue Credit` dialog is brought up, we immediately insert a new row into the `Credits` table. This new row is given a status of `void` and the credit number used is passed back into the `Issue Credit` dialog for display. Now, if the user cancels the dialog, we've already accounted for the credit number with a voided credit in the database (you may also decide to create a special status code just for cancelled transactions). If the user completes the transaction, we use the credit number to update the existing row in the `Credits` table.

Listing 2 shows the Delphi and SQL code behind the `Issue Credit` dialog. The stored procedure `CreditNew` is called when the form is displayed and takes care of generating the initial voided credit and returning the credit number. When the user clicks `Post` the `qry-CreditInsert` query is executed to update the existing record.

Payments

Handling payments of these credits is where all the action is. Figure 2 shows the main payment screen. The grid at the top shows all the outstanding credits for the customer. The user may select one or more of these credits to be paid with a single payment. The sum of all the balances of the selected credits is shown in the `Credits To Pay` box. The grid at the bottom lists all the possible payment methods. The user enters the appropriate amount for each method and the total of all methods is the total for the entire payment and is shown in the `Payment` box. If a payment does not total up to the amount of the credits to pay, then a balance due is calculated and one or more of the credits will be partially paid. These credits will remain outstanding although presumably with a lower balance due than before.

Payment Method Codes

The list of payment methods seems a mundane aspect of the system at first glance. However, quite a bit of forethought went into the design of the payment method codes.

While the list of possible method types will probably remain static and unchanging for some time, there is the possibility that new payment methods may be added in the future. In addition, different sites for the same system may want different combinations of payment methods. For these reasons, payment methods were defined in a table in the database. This way, the set of methods and codes can be easily customized for individual sites, and new methods can be added at will.

Our payment method table is defined and populated as shown in Listing 3. The `Sequence` column defines the order in which the methods will be displayed in the method grid of the application (see Figure 2). A defined sequence is necessary since we cannot rely on physical order and alphabetical order on either of the other columns may not be desirable. This is important to keep in mind when designing tables containing such lists of items, especially if you elect to use simple numeric codes rather than mnemonic alphanumeric codes.

For example, suppose our code values were simply the digits 1, 2, 3 and so on. Developers will generally elect to sort on the code value when displaying the list in combo box dropdowns, grids, etc. Now the field is serving two purposes: defining the code value for the item and the sequence of that item relative to other items in the list. Once data has been written to the database, the sequence of items is locked in forever unless you recode the data in the database.

Notice we have an `Other` item which is a catch-all for any type of payment method we haven't specifically accounted for. Suppose we add a new payment method a year after installation of the system. Our numeric code sequence forces us to use the next number in sequence as the code value for this new method. Unfortunately, we are also sorting on this column so all new code values sort out after `Other` in any list of all items. Clearly it would be more desirable for a catch-all category like `Other` to always sort out to the bottom of the list. The alternative is to recode `Other` to a

➤ *Listing 2*

```
Delphi unit:
unit fmIssue;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, DB, DBTables;
type
  TfrmIssue = class(TForm)
    ...
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure btnPostClick(Sender: TObject);
  private
  public
    CustomerNo: LongInt;
    procedure PopulateForm;
  end;
var frmIssue: TfrmIssue;
function ShowMarkerIssueDlg(aCustomerNo: LongInt): TModalResult;
implementation
{$R *.DFM}
uses dmData;
function ShowMarkerIssueDlg(aCustomerNo: LongInt): TModalResult;
begin
  Application.CreateForm(TfrmIssue, frmIssue);
  try
    with frmIssue do begin
      CustomerNo := aCustomerNo;
      PopulateForm;
      Result := ShowModal;
    end;
  finally
    frmIssue.Release;
  end;
end;
procedure TfrmIssue.PopulateForm;
begin
  { Generate a voided credit record and obtain the credit number }
  with dmDataModule.spCreditNew do begin
    ParamByName('iCreditNo').Clear;
    ParamByName('iCustNo').AsInteger := CustomerNo;
    ExecProc;
    edtNumber.Text := IntToStr(ParamByName('oCreditNo').AsInteger);
  end;
end;
procedure TfrmIssue.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
end;
procedure TfrmIssue.btnPostClick(Sender: TObject);
begin
  { Update the existing credit record }
  with dmDataModule.qryCreditIssue do begin
    ParamByName('CreditNo').AsInteger := StrToInt(edtNumber.Text);
    ParamByName('Amount').AsFloat := StrToFloat(edtAmount.Text);
    ExecSQL;
  end;
end;
end.

spCreditNew stored procedure:
create procedure CreditNew(iCreditNo integer, iCustNo integer)
  returns (oCreditNo integer)
as begin
  if (iCreditNo is null) then
    iCreditNo = gen_id(Gen_CreditNo, 1);
  oCreditNo = :iCreditNo;
  insert into Credits (CreditNo, CustNo)
    values (:iCreditNo, :iCustNo);
end;

qryCreditIssue query:
update Credits
  set Status = 'I', Amount = :Amount,
      BalanceDue = :Amount, IssueDateTime = 'now'
  where CreditNo = :CreditNo
```

➤ *Listing 3*

```
create table PaymentMethods
(
  PayMethodCode       char(2) not null primary key,
  PayMethodName       char(20) not null,
  Sequence            smallint not null
);
commit;
insert into PaymentMethods values ('CS', 'Cash',           1);
insert into PaymentMethods values ('CK', 'Personal Check',    2);
insert into PaymentMethods values ('TC', 'Traveler''s Check', 3);
insert into PaymentMethods values ('CC', 'Cashier''s Check',  4);
insert into PaymentMethods values ('OT', 'Other',             5);
```

higher value to make room for the new code, and then recode all references to `Other` in the database to the new value. A better approach would be to give `Other` an arbitrarily high value like 9, but that puts a cap on the growth potential of the table until you hit `Other` eventually.

The independent `Sequence` field avoids these issues. It also carries the side benefit that the order of methods in the list can be customized from site to site if needed without impacting the underlying logic of the system.

Granted, this is a pretty in-depth analysis for something as relatively trivial as a payment method code table. But, like everything, it's easier to learn the principles on the small cases in order to recognize the issues and then apply the techniques to larger cases.

But we're not done with the payment methods table yet. The payment methods are not likely to change very frequently but could change. We could store the payment methods in local tables on the workstations to improve throughput across the network. But if a change should be necessary, all the local tables need to be updated and there is always the likelihood of one or more being missed. We've elected to keep the lookup table in the central database server and load up a string list internally when the application starts up. The application always uses the internal string list from that point on. This way, the data is freshened every time the application starts up, but if a change is necessary, the central copy of the table can be changed and you can be assured that every workstation will receive the updated table once they are restarted. Sometimes I'll load the internal string lists upon user login instead of program start up. This is most helpful in applications that are generally up 24 hours a day, 7 days a week. All internal data is refreshed upon a new user login rather than requiring a restart of the application.

The data module unit handles this for us. It declares a `PaymentMethodsList` string list which we can use throughout the application. When the data module is created, it runs a query to fetch the payment method codes and then populates this list (see Listing 4). The payment method name is kept in the string list while the list's Objects array contains a pointer to a null-terminated string containing the method's code value.

➤ *Listing 4*

```
unit dmData;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, DB, DBTables;
type
  TdmDataModule = class(TDataModule)
  ...
end;
var
  PaymentMethodsList: TStringList;
  dmDataModule: TdmDataModule;
implementation
{$R *.DFM}
procedure TdmDataModule.dmDataModuleCreate(Sender: TObject);
var
  Code: PChar;
begin
  PaymentMethodsList := TStringList.Create;
  with qryPaymentMethodsGet do begin
    Open;
    try
      while not Eof do begin
        Code := StrAlloc(Length(FieldByName('PayMethodCode').AsString));
        StrPCopy(Code, FieldByName('PayMethodCode').AsString);
        PaymentMethodsList.AddObject(FieldByName('PayMethodName').AsString,
          TObject(Code));
        Next;
      end;
    finally
      Close;
    end;
  end;
end;
end.

qryPaymentMethodsGet query:
select * from PaymentMethods order by Sequence
```

Conclusion

Unfortunately, that's all the space I have this month. Next month, we'll complete our look at this credit system and explain the multidimensional payment processing in depth.

We'll also look at a simple TDBGrid descendant component that allows us to do the multiple non-consecutive selection which is required in the payment dialog.

Steve Troxell is a Senior Software Engineer with TurboPower Software. He can be reached by email at stevet@turbopower.com or on CompuServe at STroxell.