

As with all template changes, don't do this to the copy in your Progress install directory. Instead, if it isn't already there, make a copy of it into a matching directory structure in your application code layout and change that copy.

## 5.5.2 Understanding The *submitValidation* Procedure

Whenever a SmartDataObject saves a database record, it does this by calling the *submitRow* procedure. One of the parameters this procedure receives is a CHR(01) separated list containing the names of all fields to be updated and their associated values. This list becomes very important in a number of procedures.

The *submitRow* procedure does a number of things, including setting any needed foreign key values for any newly added records. Another very important function it performs is to call the *submitValidation* procedure. This validation procedure has a lot of built-in functionality that is not always used, but the hooks are in place just in case they might be needed. The *submitValidation* procedure receives the list of values to be updated from the *submitRow* procedure and does a number of checks and processes on each field in this list. The checks performed are:

1. Make sure the field exists within the *RowObject* temp-table
2. Make sure the field is marked in the SDO as being updateable
3. Converts any “?” value to a true unknown value
4. Stores the changed value into the associated *RowObject* field
5. Verifies that the value matches the field's format
6. Tries to run a *<fieldname>Validate* procedure

Once it has done this for every field to be updated, it tries to run the *RowObjectValidate* procedure to check the entire *RowObject* temp-table record. The *submitValidation* procedure is the driving force for all of the client-side validation performed within the SmartDataObject.

## 5.5.3 Validating Individual Fields

The functionality we are concerned about right now is for field-level validation. For every field that has been modified in the current record, the *submitValidation* procedure will try to run another procedure made up of the name of the field plus the key word “Validate”. Thus, if we made changes to the *Customer.Name* and *Customer.Address* fields, the *submitValidation* procedure would automatically attempt to run the *nameValidate* and *addressValidate* procedures. The *RUN* statement is done with *NO-ERROR* so missing procedures will not cause a problem.

These field-level procedures do not normally exist and must be added by the developer. They are not an override of a normal SmartObject class entry, so the developer will have to manually add them. It is by following the naming convention that these new

procedures are tied into the automatic processing flow for the SmartDataObject. So one of the first things to check when one of these procedures compiles fine but doesn't seem to be run is to verify that the name is correct.

Each field-level validation procedure will be passed a character value as an input, so be sure to add code defining this input parameter. Of course, you can name the parameter whatever you want. The character string that is passed to each field validation procedure is the screen value of that field. This means that if you are validating a date, decimal or some other non-character field, you will probably want to perform some form of data type conversion before you test.

A word of caution is needed when creating these field-level validation procedures. They are run on the client-side of the SDO, so if it is deployed in an AppServer environment, there will not be any available database connections. By default, the Section Editor checks the **DB-Required** flag on any newly created sections. This would force Progress to automatically ignore the procedure in a split SDO. Therefore, to make sure it works properly, you'll need to shut off the **DB-Required** flag for any field-level validation procedure that you add.

The last piece of this puzzle is that when *submitValidation* calls each field validation procedure, it checks for a return string. If something non-blank is returned, the SDO knows that some sort of error occurred. It will continue running all of the other field level validation routines but the transaction will be stopped. All of the error messages will be shown in one dialog box and the user's cursor will automatically be placed in the first field that showed an error. In many ways, this is much better than the behavior seen with the old **VALIDATE** options used in field phrases. Since all of the field-level validation procedures are processed at once, the user will see more than just the first error message. This will let them make a handful of corrections, if necessary, before trying to again save the transaction.

```

/** get the screen value */

DEFINE INPUT  PARAMETER pcNameValue AS CHARACTER  NO-UNDO.

/** test for ? or "" values */

IF pcNameValue = "" THEN RETURN "Customer name may not be blank.".

ELSE IF pcNameValue = ? THEN RETURN "Customer name may not be unknown.".

```

**Figure 161 - Field Level Validation Procedure**

This type of validation is intended for the first cut at testing the user’s input. It should be used only for testing this field in isolation to anything else. In other words, this is not the place to make sure that one field has an appropriate value based on the data in another field. Also, this is a client-side validation procedure. Do not do anything that requires a reference to the database in this piece of code. These field-level validation procedures take the place of the most basic parts of the Data Dictionary *VALIDATE* phrases.

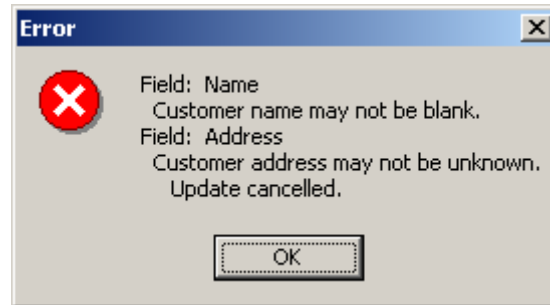


Figure 162 - Example Of Field Validation Responses

As you can see from the dialog box shown above, all of the field-level validation procedures will be run, no matter what. The fields are validated in their tab order. If there are multiple errors, each field along with its associated error message will be collected. Once the *submitValidation* procedure has finished running all of the field validation routines, it checks to see if any character strings were returned. If so, it displays them all in a single dialog box. The other nice part is that the user’s cursor is automatically placed in the first field to show an error.

Keep in mind that these hooks for validation work for only the most basic type of validation. They are client-side routines, so there will be no easy access to database fields. The only thing you have ready access to is the screen value of that field. So testing for blank or unknown values works. Testing for dates in relation to the current data is also recommended. Testing to see if a value is one of a set of possible values doesn’t work unless you either hard-code those values or have the SmartDataObject somehow obtain the list of valid values ahead of time. Don’t try to make this hook the end-all for your validation and stretch it to accomplish more than it should.

### 5.5.4 Making Sure A Field Gets Validated

You’ve got your database designed, you’ve come up with validation for many of the fields to make sure all input data is correct and you’ve built your SmartDataObjects with all of these field-level validation procedures to implement your designs. But when you put together a maintenance screen and go to update an existing record, some of the expected validation doesn’t happen. You’ve still got blank values going into the database or other problems that you had hoped to prevent.

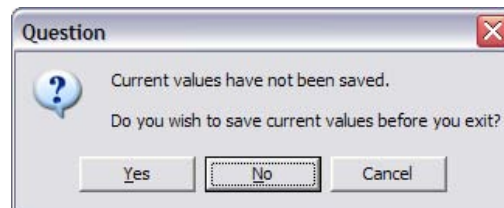
The reason for this is quite simple. Validation only happens on fields which are written back to the database. So it is only the fields that were changed and need to be saved that are checked. If you read the later sections on working with SmartViewers, you’ll learn more about the circumstances under which fields are saved.

For now, I’ll give you the short answer. Set the **MODIFIED** attribute of a field to **TRUE** to force that field to be validated (and saved). There are a number of places that this could be done, but one simple location would be in an override of the **updateRecord** procedure from the data visualization class. This is the procedure invoked by the SmartToolBar when its “Save” button is pressed and it publishes the named event “updateRecord”.

You might consider putting this type of logic in **submitRow** instead, but this is too late in the process. At the time **submitRow** is run, we’ve already executed the **collectChanges** procedure and have built a list of all of the modified fields and their new values. There are other uses for this procedure, which will be discussed in other sections.

This method is great for enforcing a mandatory style behavior. If you have fields that you might initialize as blank but want to make sure the user enters some value, this is the perfect method to make sure validation occurs on that field.

Be careful with using this “trick”. One of the default behaviors of ADM2 is warning the user when they try to leave a screen that has been modified.



**Figure 163 - Pending Changes Warning**

If you add a new record, do not change anything and then close the window, the ADM2 architecture knows it can safely close that window immediately because nothing really has been done. If you add a new record, make a change to one of the fields and then try to close the window, things behave differently. It is because the **dataModified** property of the visualization object has now been set to true. When you try to close the window, you will get a question about saving the current values before exiting. If you force a field to be seen as modified upon adding a new record, you may get this question when you do not really warrant it. Instead, force the field to be flagged as modified during the beginning part of the save process, so that it will be gathered up with all of the other user-changed fields without interfering with any normal processing.