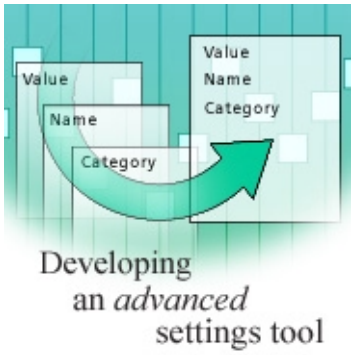
Notes.net

Iris Today



by
Jonathan
Coombs

Level: Advanced
Works with: Designer 5.0
Updated: 11/01/2001

The July 2001, *Iris Today* article, "**Applications settings tool: an alternative to profiles**," explained how to build a simple and reusable Application Settings tool that can store any number of application settings, either as text values or text lists. In some applications, this approach is really all that is needed. But many applications require more sophistication. For example, you may need to store settings of various different data types. Or you may want to present only a subset of the settings to the application administrators and hide the more technical ones.

This article explains how to take the simple Application Settings tool from the previous article and enhance it to provide a more customizable UI, allowing the developer to make the interface that all the other users see more user-friendly and robust. It explains how to support multiple data types, dynamic input validation, simplified script lookups, security at the setting level, Web formatting, and maintenance of multiple installations of the tool. The design techniques used to provide this functionality have a broad range of application and can be used when building many kinds of Notes applications, but I will describe them in the context of the Advanced Settings tool.

The Advanced Settings tool is backward compatible with the simple Setting documents. If you have already installed the simple Application Settings tool in an application and created some Setting documents, you can upgrade by removing the simple tool's design elements and pasting in the advanced ones. The Advanced Settings tool will automatically recognize the simple Setting documents and upgrade them as needed.

The Advanced Settings tool's core design elements, as well as some optional extensions, are available in the sample **Advanced Settings database** in the Iris Sandbox.

This article assumes a thorough understanding of designing Notes/Domino applications. We also recommend that you read the *Iris Today* article "**Applications settings tool: an alternative to profiles**" as background for this article.

Overview of the Advanced Settings tool

The simple Application Settings tool consisted of a Setting form for creating and editing individual Settings, a Settings UI (user-interface) view for viewing and opening them, and a Settings lookup view for accessing them through code.

The Advanced Settings tools uses the same basic design, but it splits the Settings UI view into two views: Admin Settings and Developer Settings. It also moves the Setting form's actions and header into a computed subform and provides other optional design elements.

Including these design elements and making a few changes to the Setting form enables the Advanced Settings tool to flexibly support a variety of new features:

- Distinguishing between developers and other users
- Supporting multiple data types

- Protecting your settings with input validation
- Protecting your settings using Authors fields
- Simplifying setting lookups in LotusScript
- Maintaining multiple installations of the tool
- Adapting the settings tool to your application without making design changes

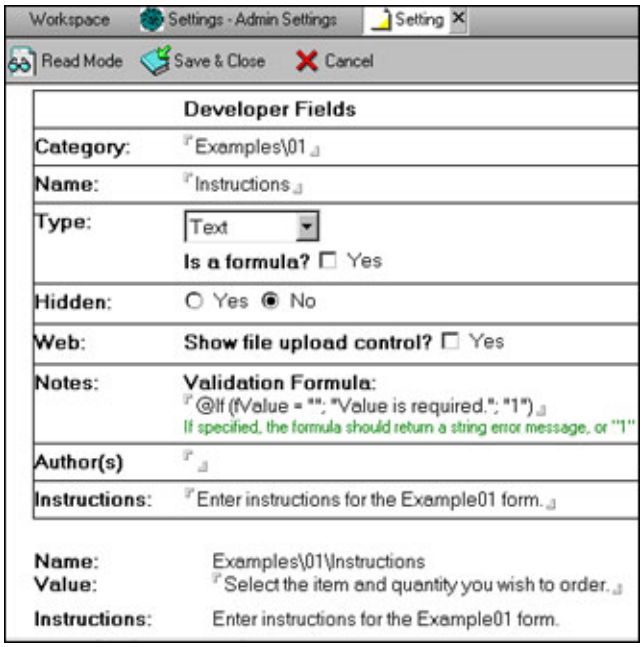
Let's explore each of these advanced capabilities.

Distinguishing between developers and other users

The simple Applications Settings tool did not distinguish between developers and application administrators, even though the two kinds of users generally use a given application (and that application's settings) quite differently. A developer typically creates and names each setting, writes instructions for it, and gives it a default value. An application administrator, on the other hand, should normally only be able to modify the default values of existing settings.

To make this distinction between developers, administrators, and other users, I've added the [developer] and [admin] roles to the Advanced Settings database's ACL. I've also added some hide-when formulas to the advanced Setting form and to the New Setting action in the Admin Settings view. Because of these hide-whens, the advanced Setting form allows developers to view and edit all fields, but it lets administrators edit only the Value field.

Here is the advanced Setting form:



The Category, Name, Value, and Instructions fields in the Setting form were explained in the **first article**. The other fields will be explained in this article.

The advanced Setting form has a Developer Fields table that is hidden from all nondevelopers. All fields in this table are editable, and the only editable field outside of the table is the Value field. Several descriptive fields (Category, Name, and Instructions) appear again outside of the Developer Fields table in computed-for-display fields for the benefit of nondevelopers.

The hide-when formula used for the Developer Fields table and New Setting action looks like this:

```
roles := @LowerCase(@UserRoles);
isdev := @IsMember("[developer]"; roles);
!isdev
```

This hide-when formula returns True if the current user has not been given the [developer] role. Since Notes ACL roles are case sensitive, the formula uses @LowerCase to make sure it will match on other variations such as [Developer] or [DEVELOPER].

Why worry about case variations in an ACL role? In a sample database created from scratch, I can naturally create roles using any case I choose. But what if I want to install a reusable tool that uses an [admin] role, into a database that has already defined an [Admin] role? The natural solution would be to put both roles in the ACL, but Notes unfortunately does not allow this. Because of this partial implementation of role case sensitivity, I prefer to use @LowerCase to ignore case in any formulas that check ACL roles. (Note that this technique will not work for Readers and Authors fields.)

Not only do developers and administrators edit different fields within each setting, but there can be settings that should be entirely restricted to developers. The Setting form contains a new field named fHidden, and the Admin Settings view (formerly known as Settings) uses this field to hide any setting whose fHidden field is set to Y. To access these hidden settings, a user must either use the hidden lookup view or install the optional Developer Settings view. (Of course, any user with Editor access who knows where to find these settings will be able to modify them.)

The Developer Settings view displays all settings using the original selection formula:

```
SELECT fDocType = "Setting"
```

The selection formula for the Admin Settings view excludes hidden settings:

```
SELECT (fDocType = "Setting") & (fHidden != "Y")
```

Supporting multiple data types

Most applications could contain various kinds of settings, including text settings supplying form pop-up help, text lists supplying keyword field options, name fields specifying workflow approvers, or even rich text settings containing canned e-mails or embedded resource files. The simple Setting form cannot store these kinds of data because it only has Value fields of types Text (fValue) and Text List (fValues). It also displays both fields at the same time, even though the application feature that uses a given setting will only use one of the two.

To better handle multiple data types, the advanced Setting form expects the developer to create all Setting documents and specify their data types. Then, when an administrator needs to modify a setting, all the settings are available in one view, and each document displays only the type of Value field necessary for the required type of setting. For example, the Value field in a setting of type Name List allows the administrator to select multiple names from the Address book, while the Value field in a document of type Rich Text provides Notes rich text field features.

The data type of an advanced Setting document is specified in the Type field on the advanced Setting form. This field allows a developer to specify which type of Value field to use, and each Value field is set to hide unless its type has been specified. Nondevelopers don't need to choose between

multiple Value fields and don't even see the Type field at all. In the example advanced Settings document shown above, the setting is of type Text, so the other kinds of Value fields are hidden.

The Type field, fType, is defined as a combo-box whose possible values are:

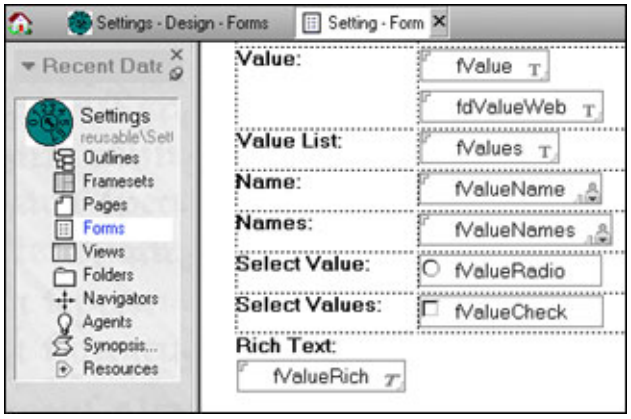
Text|T1
Text List|TM
Name|N1
Name List|NM
Rich Text|R1
Date|D1
Radio Button|S1
Check Box|SM

The item selected from this combo-box will determine which Value field is used to display and store this document's setting. To maintain backward compatibility with the simple Application Settings tool, the default value formula for fType is:

```
@If (
      (fValue = "") & (fValues != "");
            "TM";
      "T1"
)
```

That is, a Setting document whose type is undefined is assumed to be of type Text unless the Text List Value field is filled and the Text Value field is empty.

With the Type field in place, new kinds of Value fields can be supported by adding another option to the Type field's combo-box, adding a Value field of the new type to the form, setting the hide-when formula for the new Value field, and updating the formula for fValueString. (The formula for fValueString is described at the end of this section.) For example, here are various Value fields:



Name and Name List settings
The Name and Name List fields are very similar to the Text and Text List fields, except that they are of type Names and are set to "Use Address dialog for choices." For users with Notes clients, this can make some settings much more user-friendly. For example, one of my other reusable tools is a "nag agent" that loops through any view of workflow documents that are awaiting approval and reminds each document's reviewer to come review the document. Optionally, the agent can CC the application's administrators on each reminder e-mail. My nag agent tool expects this list

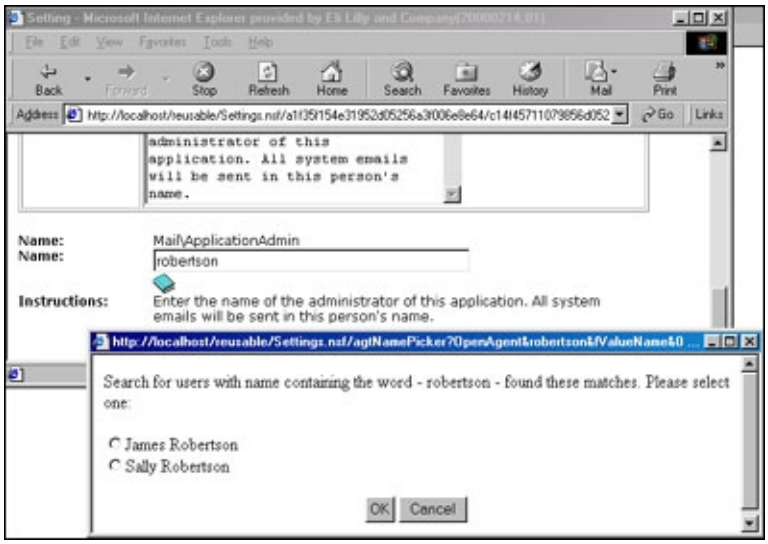
of administrators to be stored in a Name List setting document named AutoReminders\CopyTo. (This Setting document is provided in the **Advanced Settings database** under the Examples category, as are various other sample settings.)

Unfortunately, there is no simple way of duplicating the Notes Address dialog box for Web users. Sometimes an organization chooses to develop its own standard Web dialog box, which you can incorporate into the Advanced Settings tool to better support Name fields from the Web. If you have a Web dialog box of your own, you can use it instead. Or, you can use the basic Web name picker included with the Advanced Settings tool. It has an action hotspot next to the fValueName field that uses JavaScript to call the agtNamePicker agent and pass parameters to it:

```
window.open('http://' + document.forms[0].fdServerWeb.value + '/' +
document.forms[0].fdDBPathWeb.value + '/agtNamePicker?OpenAgent&' +

document.forms[0].fValueName.value + "&fValueName&0", 'VerifyUser',
'width=500,height=300,resizable=yes,scrollbars=yes')
```

The URL generated by this hotspot includes the server name and database path, an agent to run (agtNamePicker?OpenAgent), the value to search for, the field into which the name should be returned (fValueName), and whether that field supports multiple values. (Note that if you choose to remove or replace the supplied name picker, the display fields fdServerWeb and fdDBPathWeb can be removed as well.) The agtNamePicker agent essentially searches the Address Book for the supplied name in the Setting document and generates an HTML page listing the possible matches on that name as radio buttons. (To examine the agent's code, see the **agtNamePicker agent sidebar**.)



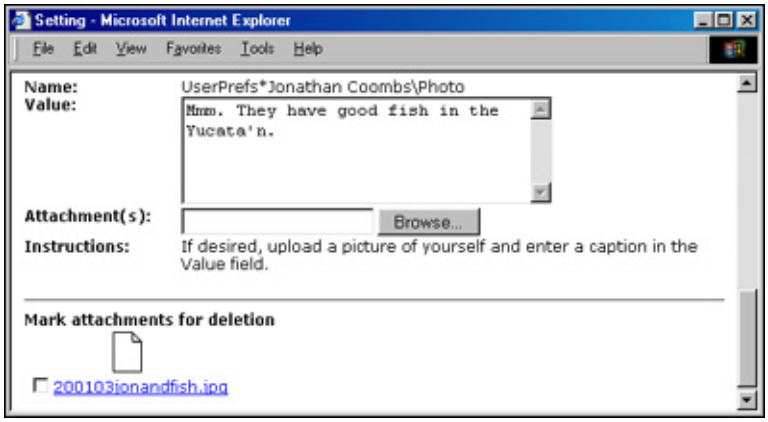
Rich Text settings

Rich Text fields are useful for storing many different kinds of data, including images, formatted text, and attachments. Except for a few important caveats, the Advanced Settings tool can support all of these for both Notes and Web clients.

From the Notes client, all you need to do is create a setting and specify its type to be Rich Text. The Value field will natively support all formatting and file attachments from the Notes client. (If you want the administrator to attach files to a setting, it is a good idea to mention the File - Attach menu command in the setting's instructions.)

For the benefit of the Web client, the fValueRich field is set to "Display Using Java Applet." This nifty Domino option allows Web users to use a limited set of text formatting features. Note that Web users who open Rich Text settings will be prompted to download the rich text applet. This applet formats rich text quite differently than the Notes client, so you should avoid using both kinds of clients on these kinds of settings. Hopefully, future versions will be a little more compatible, but for now I try to avoid editing any Rich Text fields from the Web.

The Java Rich Text applet does not support file attachments, so I've also placed a File Upload control and a checkbox field (fShowUpload) on the Setting form. The upload control will only display if fShowUpload is checked. (Incidentally, the upload control can be displayed to the Web client for all types of settings, not just Rich Text.)



Date settings

I have yet to use a setting of type Date, but I've included this type in order to support all the main data types. A setting of this type might be useful if you were to create a scheduled agent that should only run when the application administrator has specifically set it for a certain date (for example, see Examples\05\DeleteLastYear in the **Advanced Settings database**.)

Invalid dates entered through Web clients can generate type mismatch errors before input validation has a chance to catch them. Instead of creating a sophisticated JavaScript validation routine, I've chosen to present a separate field of type Text (fValueDateWeb) to Web users. This field's input translation formula attempts to validate the Web field and synchronize it with the Notes field (fValueDate):

```
@If ( @ClientType = "Notes"; @Return(fValueDate); "");
webdate := @TextToTime(fValueDateWeb);
newdate := @If (
      @IsError(webdate);
            "";
      webdate
);
FIELD fValueDate := newdate;
newdate
```

From a Notes client, the value from the Notes field is simply copied into the Web field. From a Web client, the Web field's value is validated, cleared if invalid, and copied into the Notes field. Clearing invalid entries is less than ideal, but it is a simple way of avoiding type mismatch errors.

Formula settings

One of the most powerful features of LotusScript is the Evaluate function, which allows LotusScript routines to use the functionality of many formula language functions. In a nutshell, the Evaluate function takes a string containing a formula and returns the result of the formula as a variant. For example, the following LotusScript code will print the current user's common name to the status bar:

```
Dim varTemp As Variant
varTemp = Evaluate("@Name([CN]; @Username)")
Print varTemp(0)
```

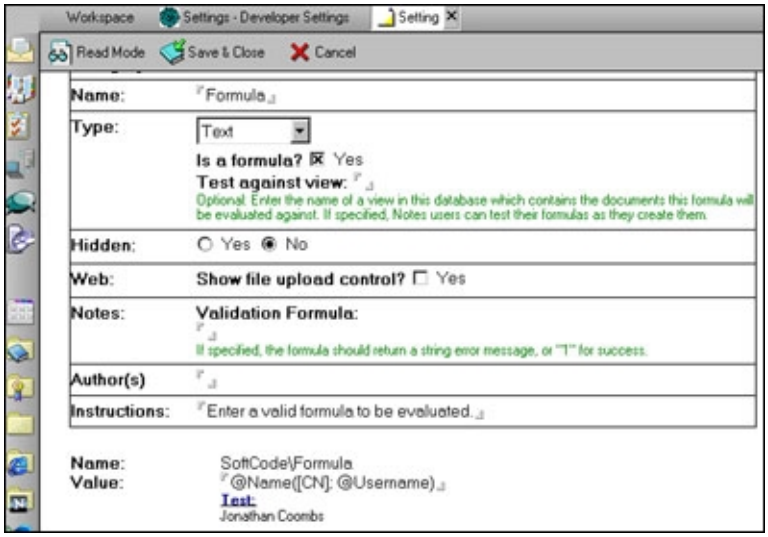
For more information about the Evaluate function, refer to the **Domino R5 Designer Help** and the *Iris Today* article, "**Simplifying your LotusScript with the Evaluate statement**."

While developing an application, you may realize that a portion of your LotusScript code will need to change in the future in order to accommodate changes in the users' needs. Whenever possible, it's a good idea to convert this portion of code into a formula and store it as a setting. It can then be looked up dynamically when it needs to be used, and passed to the Evaluate function. The agtTestSoftCode agent and SoftCode\Formula setting illustrate this approach. The agent evaluates the formula stored in the Setting document and prints out the first line of the result:

```
Sub Initialize  'agtTestSoftCode
      Dim s As New NotesSession, vwSettings As NotesView
      Dim strFormula As String, varResult As Variant
      Set vwSettings = s.CurrentDatabase.GetView("vwLookSettings")
      strFormula =
vwSettings.GetDocumentByKey("SoftCode*Formula").fValue(0)
      varResult = Evaluate (strFormula)
      Print varResult(0)  'print the first line of the result
End Sub  'agtTestSoftCode
```

"Soft-coding" features strategically in this manner can add a lot of flexibility to your applications. On the other hand, soft-coded features also make your code dependent on the Advanced Settings tool. If the tool is not installed properly, or if a formula setting has been deleted, the code that depends on it will fail. If the formula setting is quite complex, losing it is tantamount to losing part of your database design. With these considerations in mind, I usually choose to store a small number of soft-coded formulas as settings.

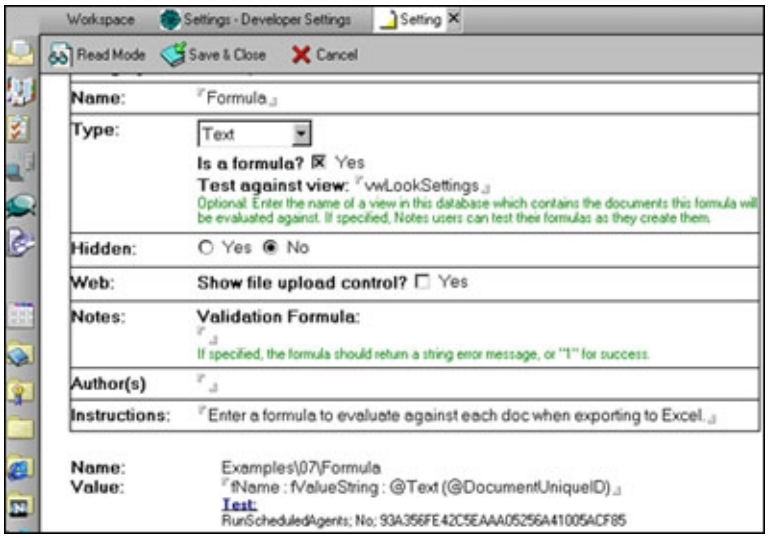
Storing formulas as ordinary text settings works adequately but can be somewhat tedious to debug and test. Instead of adding a completely new type to support formula settings, I have enhanced the Text type to provide formula testing.



To enable formula testing on a given Text setting, check Yes for the Is a formula? (fIsFormula) field. This sets the flag field fIsFormula to True, unhiding the Test hotspot at the bottom of the form. The Test hotspot runs the Evaluate function against the formula stored in the Value field and displays the result. (For another example of a formula setting, take a look at Examples\06\Formula in the **Advanced Settings database**.)

Some formulas are meaningful at the database level, but many others are designed to be used in the context of a specific Notes document. For example, @DocumentUniqueID and @IsResponseDoc have no meaning except in reference to a document. The result of evaluating the formula @Responses in isolation is zero. But if you pass a document object into the Evaluate function along with the formula, it will be evaluated against that document and will return a legitimate value. Since this is the kind of formula I use most often, I added a Test Against View field (fFormulaView). If a view is specified in fFormulaView, clicking the Test hotspot will evaluate the formula in the Value field against a document in the specified view.

For example, if I were writing a routine to export all the documents in a view to Excel, I might have the export routine use a formula that computes the exact cell values that should be exported. The Test hotspot would allow me to debug the formula before attempting an export:



The LotusScript code behind the Test hotspot looks like this:

```
Sub Click(Source As Button)
    Dim ws As New NotesUIWorkspace, s As New NotesSession
    Dim db As NotesDatabase, vwTest As NotesView, doc As
    NotesDocument
    Dim strFormula As String, docTest As NotesDocument
    Set db = s.CurrentDatabase
    Set doc = ws.CurrentDocument.Document
    strFormula = doc.fValue(0)
    If strFormula <> "" Then
        'A formula exists
        varResult = ""
        If doc.fFormulaView(0) = "" Then
            'Evaluate the formula
            varResult = Evaluate (strFormula)
        Else
            'Evaluate the formula against a document
            Set vwTest = db.GetView(doc.fFormulaView(0))
            If Not (vwTest Is Nothing) Then
                iFormulaPreview_g = iFormulaPreview_g + 1
                Set docTest =
                vwTest.GetNthDocument(iFormulaPreview_g)
                If docTest Is Nothing Then
                    'Loop back to the first doc
                    iFormulaPreview_g = 1
                    Set docTest =
                    vwTest.GetNthDocument(iFormulaPreview_g)
                End If
                 varResult = Evaluate (strFormula, docTest)
            End If
        End If
        doc.ftFormulaTest = varResult
        doc.GetFirstItem("ftFormulaTest").SaveToDisk = False
        Call ws.CurrentDocument.Refresh
    End If
End Sub 'Hotspot : Click
```

This code first makes sure a formula has actually been placed in the Value field. If so, it then checks the fFormulaView field to determine whether the formula should be evaluated at the database level or against a document. The syntax for the first case should look familiar:

```
varResult = Evaluate (strFormula)
```

In the second case, the view specified in the fFormulaView field is opened and a document in the view is accessed. The formula is then evaluated against that document:

```
varResult = Evaluate (strFormula, docTest)
```

But which document in the view should the formula be tested against? Whenever a Setting document is opened, the Setting form's PostOpen event initializes a global counter variable named iFormulaPreview_g to zero. Each time the user clicks the Test hotspot, this counter is incremented. This allows the user to test the formula against a variety of documents.

Finally, I wanted the formula result to be displayed but not saved with the Setting document. Fortunately, the NotesItem class provides a SaveToDisk property to support temporary fields:

```
doc.GetFirstItem("ftFormulaTest").SaveToDisk = False
```

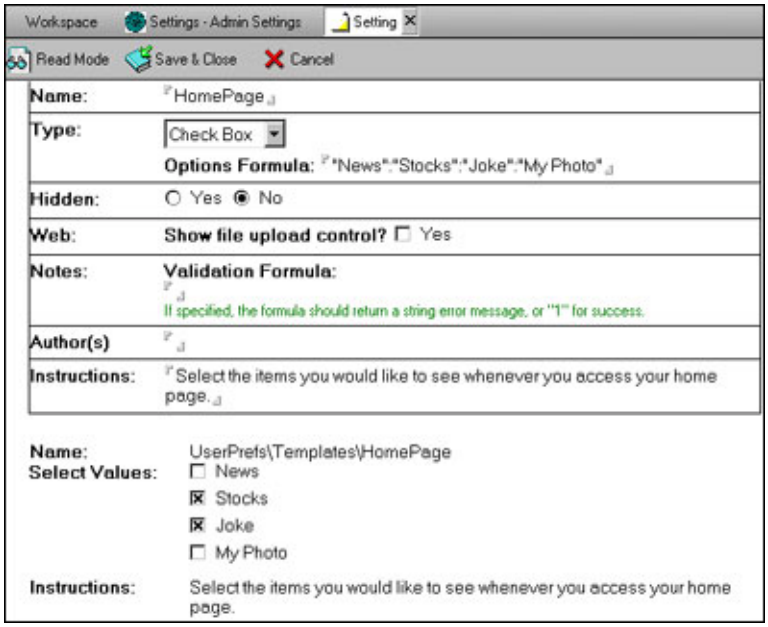
The temporary field (ftFormulaTest) is displayed through a computed-for-display field (fdFormulaTest), so no trace is left on the back-end document.

Radio Button and Checkbox settings
It is often best to limit a setting's possible value to a predefined or computed list of options. The fValueRadio and fValueCheck provide this functionality through single-selection radio buttons and multiple-selection checkboxes, respectively.

For a setting of this type, the advanced Setting form requires a valid Notes formula in the fOptionsFormula field that will return a list of values to present to the user. For example, you might have an on/off switch setting that enables or disables all the scheduled agents in your application. (Those agents would run, but each would check this setting before doing anything.) For example, in the **Advanced Settings database**, the Examples\03\RunScheduledAgents setting's Radio Button Options Formula looks like this:

```
"Yes|y":"No|n"
```

Here's how the UserPrefs\Templates\HomePage setting's Check Box Options Formula appear in the Settings document:



The Exiting event of the fOptionsFormula field evaluates the entered formula, places the resulting list in the hidden fOptions field, and refreshes the document. This refreshes the fValueRadio or fValueCheck field's option list to match the current formula. (Since we have already seen examples of the Evaluate function, the Exiting event's code is not shown here.)

Updating the fValueString formula
As described in the **first article**, a hidden computed field, fValueString, stores the current value of a setting as a single text value. This makes it easier to display the setting in a view or access its value to be processed as text. The formula for fValueString uses the type of the current setting to determine which Value field contains the setting's value and which conversion formula should be used:


```
@If (
      fType = "T1"; fValue;
      fType = "TM"; @Implode(fValues; @NewLine);
      fType = "N1"; @Name([CN]; fValueName);
      fType = "NM"; @Implode(@Name([CN]; fValueNames); @NewLine);
      fType = "D1"; @Text(fValueDate);
      fType = "S1"; fValueRadio;
      fType = "SM"; @Implode(fValueCheck; @NewLine);
      fType = "R1"; fValueString;
      ""
)
```

The types of fields that support multiple values are converted to multi-line strings using @Implode, and name fields are converted to text using @Text. There is no formula capable of extracting the unformatted text of a rich text field into text, so fValueString ignores the Rich Text type. Instead, the form's QueryClose event uses the GetFormattedText method (in NotesRichTextItem) to accomplish this task:

```
Sub Queryclose(Source As Notesuidocument, Continue As Variant)
      'If a rich text setting, put plain text into fValueString.
      Dim doc As NotesDocument
      Dim rtitem As NotesRichTextItem
      Dim strTemp As String
      Set doc = Source.document
      If bSaveAttempted And ( Cstr( doc.fType(0) ) = "R1" ) Then
            ' QuerySave has set bSaveAttempted to True,
            '   and this a Rich Text setting.
            Set rtitem = doc.GetFirstItem( "fValueRich" )
            strTemp = rtitem.GetFormattedText( False, 0 )
            doc.fValueString = strTemp
            Call doc.save (False, False)
      End If
End Sub
```

Tying up loose ends with a WebQuerySave agent
You may have noticed that QueryClose code only solves the rich text problem for fValueString if the user has a Notes client, and that the Exiting event of the fOptionsFormula field also only works from Notes. Web clients do not directly support such form events as PostOpen, Exiting, QuerySave, or QueryClose. Instead, Domino allows the functionality of some of these events to be duplicated in custom WebQueryOpen and WebQuerySave agents.

I have created one such agent, named Web Save frmSetting, to duplicate the functionality of the frmSetting.fOptionsFormula.Exiting and frmSetting.QueryClose events. If installed, this agent is launched on the WebQuerySave event (otherwise, the agent's absence is ignored and its functionality is unavailable to Web users):

```
@Command([ToolsRunMacro]; "agtSaveFrmSetting");
@True
```

The ToolRunMacro command returns True if the agent runs successfully, and False otherwise. This formula, however, returns True either way because the Web save agent is an optional design element.

Protecting your settings with input validation

The main purpose for making the Setting form more user-friendly is to reduce the likelihood that an application administrator will misunderstand and accidentally break a setting. Using the Address dialog box or a predefined set of checkboxes can help protect certain kinds of settings, but

others can be best protected by input validation.

Of course, a specific setting's input validation formula cannot be hard-coded into the generic Setting form. Once again the Evaluate function provides an excellent alternative: storing the formula in a special hidden field. On QuerySave, the Setting form evaluates any formula stored in this field (fValidation) against the current setting document before allowing the document to be saved:

```
Sub Querysave(Source As Notesuidocument, Continue As Variant)
    Dim doc As NotesDocument, varTemp As Variant, strTemp As String
    'If specified, run the validation formula
    Set doc = Source.Document
    If doc.fValidation(0) <> "" Then
        On Error Goto tagErrorHandler  'enable the validation error
        handler
        varTemp = Evaluate (doc.fValidation(0), doc)
        On Error Goto 0  'disable the validation error handler
        strTemp = Cstr (varTemp(0))
        If strTemp <> "1" Then
            If strTemp = "" Then strTemp = "Document did not pass
            validation."
            Messagebox strTemp, , "Validation"
            Continue = False 'Don't allow the save
        End If
    End If
tagEnd:
    Exit Sub
tagErrorHandler:
    Messagebox "Error #" + Cstr(Err) + ": " + Error$, , "Error in validation
    formula"
    Continue = False 'Don't allow the save
    Resume tagEnd
End Sub  'frmSetting.QuerySave
```

Since this code only mimics Notes validation, the output of the formula in the fValidation field needs to be a little different than that of a standard validation formula. Instead of returning @Success or @Failure, I chose to require the formula to return 1 on success and an error message on failure. This way, the QuerySave code can simply check for the 1 value and recognize anything else as an error message to be displayed. If the validation formula itself causes an error, the error handler returns the Notes error message and still prevents the user from saving.

Perhaps the simplest and most common use of field validation is to require that the field in question contain a value. In a setting of type Text, this kind of validation formula might look like this:

```
@If (fValue = ""; "Value is required."; "1")
```

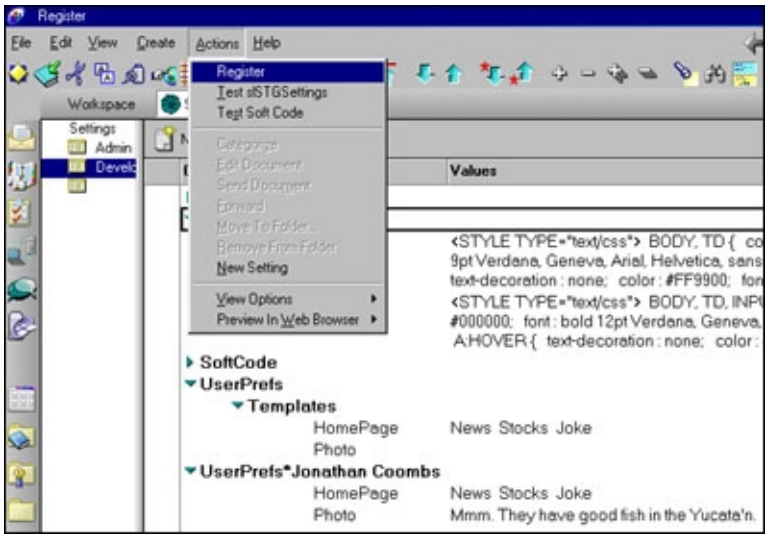
Note that the input validation feature is not currently supported for Web clients, since they do not execute the QuerySave code.

Protecting your settings using Authors fields

By default, any Notes document that doesn't have an Authors field can only be edited by users with Editor access or higher. In some applications, you may wish to give certain users access to modify settings without giving them full Editor access. The fAuthors field is a standard Authors field that allows a developer to enter the names or roles of users who should be allowed to modify the current setting.

For example, many applications allow registered users to maintain personal profiles and individualized settings. You could use the Advanced Settings

tool to allow users to customize the way they interact with a Notes/Domino application. To see a partial implementation of this in the **Advanced Settings database**, run the Register agent by selecting Actions - Register. This agent creates a set of user preference settings for the current user based on a set of templates:



The Register agent makes copies of all the documents in the UserPrefs\Templates category, re-categorizes the copies under UserPrefs* *User Name*, and adds the user's name to the fAuthors field.

Simplifying setting lookups in LotusScript

In the **first article**, I recommended using standard documents instead of profile documents, but I also mentioned that using standard documents can involve more overhead. Although they are easier to maintain and copy from one database to another, they are more tedious to access from code. I know of no way around this in formula language, but I have created a script library named slSTGSettings that can help simplify setting lookups in LotusScript.

For example, the STGGetSettingValue function provides access to setting values in much the same way as @GetProfileField provides access to profile document fields. Without this function, a very concise setting lookup in script might look like this:

```
Dim varSet As Variant
Dim s As New NotesSession, vw As NotesView
set doc = s.CurrentDatabase.GetView("vwLookSettings")
varSet = vw.GetDocumentByKey("SoftCode*Formula",True).fValue(0)
```

Using STGGetSettingValue, the lookup code is much simpler:

```
Use "slSTGSettings"
Dim varSet As Variant
varSet = STGGetSettingValue ("SoftCode*Formula")
```

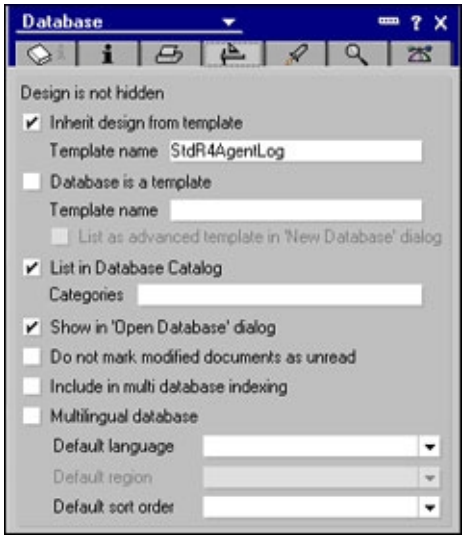
The STGGetSettingValue function takes care of finding the requested setting in a lookup view, determining which Value field to use, and returning the current value. (To see this function's source code, see the **STGGetSettingValue function sidebar**.)

Maintaining multiple installations of the tool

In the **first article**, I made a passing reference to "element-level design

inheritance," which is quite similar to database-level design inheritance but is more finely tuned.

Suppose you have a Notes application template stored in an NTF file—the standard Notes log template is a good example. When you create a new log database based on this template, the new database is set to inherit its entire design from the database that claims to be the StdR4AgentLog template (in this case, alog4.ntf). Choosing to refresh the design of the database (File - Database - Refresh Design) causes the entire design of olog.nsf to be updated to match the design of alog4.ntf. This means that if your organization were to modify the standard Agent Log template, those modifications could be easily rolled out to all the agent logs in the organization. This is known as database-level design inheritance, and it is specified in the Database properties box.



Note: Remember that database-level design inheritance is linked by these database properties alone, and not by the filename of the template database. So you must be careful not to give multiple template databases the same template name.

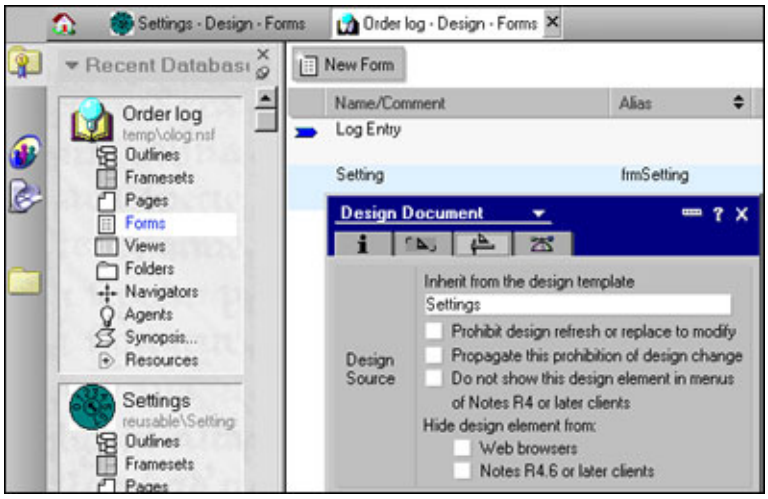
Database-level inheritance is commonly used to refresh an application's design from the development server to the production server. Some companies also use a test server between development and production. In this three-layer scenario, the databases on the development and test servers would both be given a template name (for example, Order Log), and the databases on the test and production servers would both be set to inherit design from that template.

Element-level design inheritance is a bit trickier than database-level inheritance. Individual design elements can be marked to inherit their designs from specific templates, independently of any database-level inheritance. During a design refresh, first any unmarked elements are refreshed from the database-level template (if there is one), and then any marked elements are refreshed from their specified templates. (Note that a design refresh will delete any unmarked elements not found in the template, but it will never delete a marked element.)

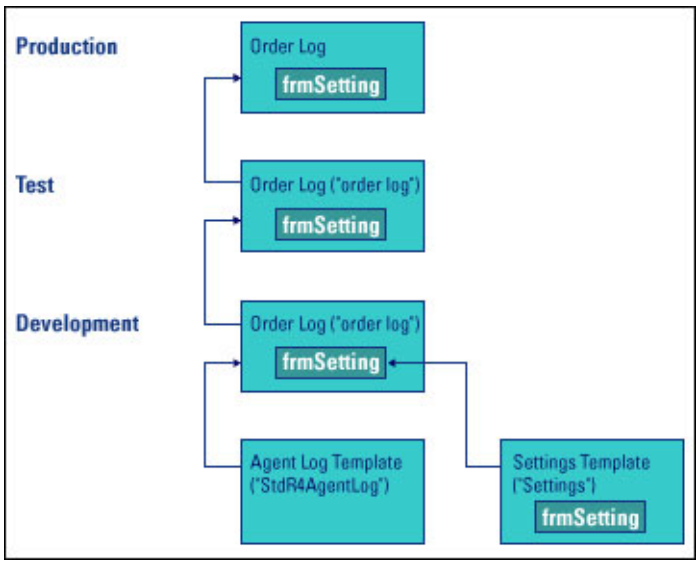
I like to maintain a central template of reusable tools (including the Advanced Settings tool) on the development server, and use element-level inheritance to roll out changes to those tools. Whenever I fix or enhance an existing tool, I do so in the template, ensuring that the change will be refreshed to all my application's templates on the development server. (Of

course, this means I have to make sure that each fix or enhancement is compatible with all the applications that use the tool.)

For example, I might take a standard log database and add some functionality to it that enhances its ability to support an Orders database. If the database as a whole inherits from StdR4AgentLog, any new elements I create would need to be set to inherit from their own template. If I installed the Advanced Settings tool, I would set its design elements to inherit from Settings as shown here:



In a three-layer development environment, the inheritance structure would end up looking something like this:



Of course, this approach saves the most time when used with complex, self-contained packages or tools. It could be used to centrally manage the design of either the simple or advanced Settings tool, but it might not be worth the effort for the simpler one.

For more information concerning design inheritance, refer to the **Domino R5 Designer Help**.

Adapting the Settings tool to your application

without making design changes

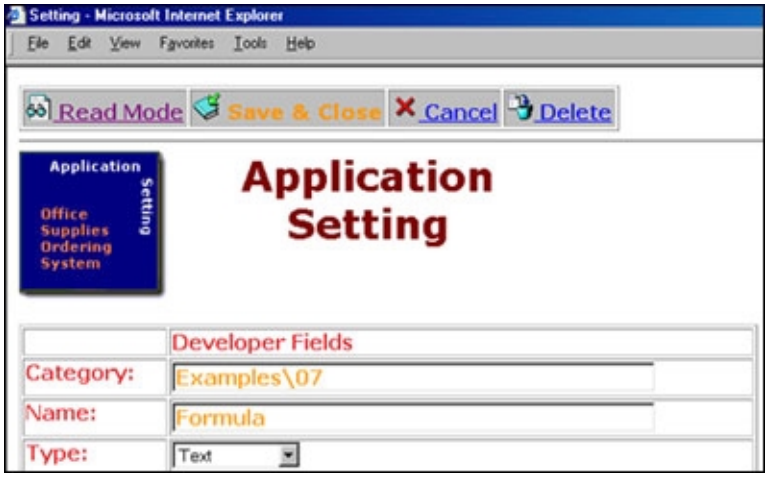
Most successful applications present users with a clear and consistent interface. After going to all the trouble of installing a reusable tool into an application and setting up element-level design inheritance, it would pain me to throw it all away just to make the tool's interface match the application's interface. There really is no perfect solution to the dilemma between reusability and usability, but there are techniques that can help.

Many installation-specific parts of a form's design can be moved into a computed subform. The Advanced Settings tool comes with its header and actions already separated out into the sfrmHeader subform. If necessary, design inheritance can be turned off for just this one element and it can be customized to match the look and feel of the application it is installed in.

In addition, in Domino applications, the appearance of an entire Web form can be controlled through cascading stylesheets (CSS). The advanced Setting form can format itself for Web users using the stylesheet stored in the optional SettingForm\Styles Setting document. The form's HTML Head Content property looks for this setting and, if it finds it, inserts the contents of its Value field between the HTML page's <HEAD> tags:

```
REM "If styles have been supplied via a setting doc, use them.";
temp := @DbLookup("Notes":"NoCache"; ""; "vwLookSettings";
"SettingForm*Styles"; "fValue");
styleblock := @If( @IsError(temp); ""; temp);
styleblock
```

CSS and modern Web browsers can give you a lot of control over the appearance of an HTML page without making any formatting changes to the HTML code itself. This can enable you change the font, color scheme, and spacing of a Web form without modifying the form's design. For example, in the **Advanced Settings database**, try swapping SettingForm\Styles2 for SettingForm\Styles. You get something like this:



Personally, I'm not so sure my customers would go for the orange and red look, but imagine how much more fun prototyping sessions can be with CSS.

Conclusion

There are many things to consider when developing an application—multiple kinds of users and software clients, input validation, security, usability, and testing just to name a few. Developing flexible, reusable tools like the Advanced Settings tool can be the key to progressively increasing the quantity and quality of applications you can build. You can further leverage the impact of design reuse across many applications by taking advantage of element-level design inheritance.

ABOUT JONATHAN COOMBS

Jonathan is a software developer for **Joseph Graves Associates, Inc.** in Indianapolis. JGA is a full service consulting firm that delivers quality IT services and customized e-commerce, Internet, and document management software solutions. Jonathan's professional interests include software reuse, Lotus Notes and Domino technology, and computational linguistics. He can be reached at **jcoombs@jgraves.com**.

]

agtNamePicker agent

```
Sub Initialize 'agtNamePicker
	'This agent searches the Address Book for the supplied name and generates an HTML page listing the
	possible matches on
	'  that name as radio buttons.
	'Sample call (JavaScript in an action hotspot):
	'window.open('http://' + document.forms[0].ServerAndPath.value + '/agtNamePicker?OpenAgent&' +
	document.forms[0].fUserAdd.value + "&fUser&1", 'VerifyUser',
	'width=500,height=300,resizable=yes,scrollbars=yes')
	'Parameters:
	'- You should pass exactly three parameters in the URL calling this agent, in this order:
	'  - strSearchFor (string): The search value the user typed in
	'  - strDestField (string): The field in which to store the selected name
	'  - bMulti (0 or 1): Whether strDestField supports multiple values. If true, new names are separated using
	commas
	'Known bugs:
	'- Netscape 6 ignores the OK button
	'
	'Sample output:
	'<HTML>
	'<!-- Lotus-Domino (Release 5.0.5 - September 22, 2000 on Windows NT/Intel) -->
	'<HEAD>
	'</HEAD>
	'<BODY TEXT="000000">
	' <script> function addNames() {
	'   window.opener.document.forms[0].fValueName.value = Temp.value;
	'   window.close();
	'   }
  ' </script>
	'Search for users with name containing the word - duck - found these matches. Please select one:<br><br>
	'<INPUT NAME="Temp" TYPE=hidden VALUE="">
	'<INPUT TYPE=radio NAME="SelectUser" VALUE="Daffy Duck" onClick="Temp.value='Daffy Duck'">Daffy
	Duck<br>
	'<INPUT TYPE=radio NAME="SelectUser" VALUE="Donald Duck" onClick="Temp.value='Donald
	Duck'">Donald Duck<br>
	'<BR><DIV ALIGN=center><INPUT TYPE=button onClick="addNames()" VALUE="OK">
	'<INPUT TYPE=button onClick="window.close()" VALUE="Cancel"><DIV>
	'</BODY>
	'</HTML>

	Dim session As New NotesSession, doc As NotesDocument
	Dim strSearchFor As String
	Dim varMatches As Variant
	Dim varTemp As Variant, strTemp As String, strNewValue As String
	Dim varParms As Variant, strDestField As String, bMulti As Integer, bFound As Integer

	On Error Goto Errorhandler

	Set doc = session.DocumentContext
```

```
'Parse out the three parameters from the CGI string.
'Examples: "?OpenAgent&Smith&fUserName&1",  "?OpenAgent&&fUserName&0"
strTemp = |@Explode ( @Right("| + doc.Query_String_Decoded(0) + |";"&"); "&" ) |
varParms = Evaluate (strTemp)
bFound=False  'Assume the worst: no match will be found
If Ubound (varParms) = 2 Then
    'We got all the parameters
    strSearchFor = varParms(0)
    strDestField = varParms(1)
    bMulti = Cint( varParms(2) )
    strNewValue = "Temp.value"

    strTemp = |@unique(@NameLookup ( [Exhaustive]; "| + strSearchFor + |" ; "Fullname"))|
    varMatches = Evaluate(strTemp)
    If (Ubound(varmatches) >= 0) And (varmatches(0) <> "") Then
        'Match(es) found! Build JavaScript and radio buttons to present the names and process the user's
        selection.
                                                    bFound = True
        'Create the script called by the OK button
        Print | <script> function addNames() {|
        If bMulti Then
            'The OK button should append the selected name to any existing names in the strDestField
            field.
            Print | var strOld = window.opener.document.forms[0].| + strDestField + |.value;|
            Print | if ( strOld == "") {|
            Print |   window.opener.document.forms[0].| + strDestField + |.value = | + strNewValue + |;|
            Print | }|
            Print | else {|
            Print |   window.opener.document.forms[0].| + strDestField + |.value = strOld + ", " + | +
            strNewValue + |;|
            Print | }|
        Else
            'The OK button should write the selected name over any existing name in the strDestField field.
            Print |   window.opener.document.forms[0].| + strDestField + |.value = | + strNewValue + |;|
        End If
        Print | window.close(); } </script>|
        'Create the body that displays the matches
        Print "Search for users with name containing the word - " & strSearchFor & " - found these matches.
        Please select one:<br><br>"
        Print |<INPUT NAME="Temp" TYPE=hidden VALUE="">|
        Forall varMatch In varMatches
            'Create a radio button for this match
            Print |<INPUT TYPE=radio NAME="SelectUser" VALUE="| & varMatch & |"
            onClick="Temp.value='| & varMatch & |'">| & varMatch & |<br>|
        End Forall
        'Create the OK and Cancel buttons
        Print |<BR><DIV ALIGN=center><INPUT TYPE=button onClick="addNames()" VALUE="OK">|
        Print |<INPUT TYPE=button onClick="window.close()" VALUE="Cancel"><DIV>|
    End If
End If

If Not(bFound) Then
    'No matches found. Just create a message and a Close button.
    If strSearchFor = "" Then
        Print "Please enter a name to search for. <br>"
    Else
        Print "Search for users with name containg the word - " & strSearchFor & " - did not find any
        matches. <br>"
    End If
    Print |<BR><DIV ALIGN=center><INPUT TYPE=button onClick="window.close()"
    VALUE="Close"></DIV>|
End If
```



```
    Exit Sub

ErrorHandler:
    Print "Error " & Str(Err) & ": " & Error$
End Sub 'agtNamePickerS
```

]

STGGetSettingValue function

```
Const C_SETTINGS_VIEW = "vwLookSettings"

Public Function STGGetDocument (strView As String, strKey As String) As NotesDocument
	' Given a doc's view and key, return the doc
	Dim s As New NotesSession
	Set STGGetDocument = s.CurrentDatabase.GetView(strView).GetDocumentByKey(strKey,True)
End Function 'STGGetDocument

Public Function STGGetSettingValue (strName As String) As Variant
	' Created by Jonathan Coombs on 9-15-2000
	' Given a Setting doc's key, look up the doc and return the appropriate field value as a variant
	' Use the type field to determine which value field to use. Only return an array if the setting type
	'   supports multiple values.
	' This function provides the Settings tool with a degree of modularity. If you use it to handle
	'    all of your lookups, none of your calling code needs to directly refer to the fields on frmSetting.
	' Warning: The "r1" type is not yet fully supported. See STGTest() and STGGetSettingRTF() for details.

	Dim doc As NotesDocument, item As NotesItem
	Dim strType As String, strError As String, varReturn As Variant

	'If an error occurs, ignore it and return null
	strError = ""
	On Error Goto tagError

	Set doc = STGGetDocument (C_SETTINGS_VIEW, strName)
	strType = Lcase$ ( Cstr ( doc.fType(0) ) )

	Select Case strType
	Case "t1"
		varReturn = doc.fValue(0)
	Case "tm"
		varReturn = doc.fValues
	Case "n1"
		varReturn = doc.fValueName(0)
	Case "nm"
		varReturn = doc.fValueNames
	Case "r1"
		'Warning: This case currently only works in debug mode.
		Set item = doc.GetFirstItem ("fValueRich")
		Set varReturn = item 'Warning: This will return an object, not a simple value
	Case "d1"
		varReturn = doc.fValueDate(0)
	Case "s1"
		varReturn = doc.fValueRadio(0)
	Case "sm"
		varReturn = doc.fValueCheck
	Case Else
		varReturn = strError
```

```
        End Select

tagEnd:
        If strType = "r1" Then
                Set STGGetSettingValue = varReturn
        Else
                STGGetSettingValue = varReturn
        End If
        Exit Function

tagError:
        varReturn = strError
        Resume tagEnd

End Function 'STGGetSettingValue
```