THE VIEW LOTUS DEVELOPER 2006

LotusScript Coding Best Practices

Bill Buchan
HADSL

© 2006 Wellesley Information Services. All rights reserved.

What We'll Cover

- Introduction
- Structure
- Variables
- Comments
- Cut 'n' Paste Code
- Error Handling
- Functions and Subs
- Memory Structures
- Performance
- Design Patterns
- Summary and Questions

What We'll Cover

- Introduction
- Structure
- Variables
- Comments
- Cut 'n' Paste Code
- Error Handling
- Functions and Subs
- Memory Structures
- Performance
- Design Patterns
- Summary and Questions

What is this About?

- This talk aims to demonstrate LotusScript coding Best Practices, showing:
  - Why Best Practices are useful
  - How to save time and effort
- Best Practices?
  - Guidelines adopted by the marketplace to help increase quality
  - General assumption about custom business code:
    - It's developed for maintainability

Who am I?

- Bill Buchan
  - Dual PCLP in v3, v4, v5, v6, v7
  - 10+ years senior development consultancy for Enterprise customers
    - Learn from my pain!
  - 5+ Years code auditing
  - CEO of HADSL – developing Best Practice tools

Introduction to Best Practices:

- The first rule of Best Practices is:
  - There is no Best Practice!
- Seriously
  - Don’t use this as a prescriptive list!
  - Examine each one and decide whether it makes sense in your current context
  - Be open to new ideas all the time
- Attitude is better than prescriptive coding
  - No “Not invented here”
  - Collaborative programming:
    - Discuss ideas and approaches There will always be an improvement
  - The Humble Programmer
  - The Lazy Programmer

How to Code

- Architect well
  - Get the data model right
- Code for maintainability
  - Code updates cost more than new code!
  - Remember – the next person to maintain code might be you!
  - Keep it simple. Over-architecting is bad, too!
- Example:
  - A Scottish Insurance Company: In the 60's, decided to only use two digits for dates, full in the knowledge that this would have to change in Y2k
  - Assumed that the code would not last 40 years
  - The manager who made that decision was yanked back from retirement to be the Y2K manager
  - It rarely works out this well

How to Test

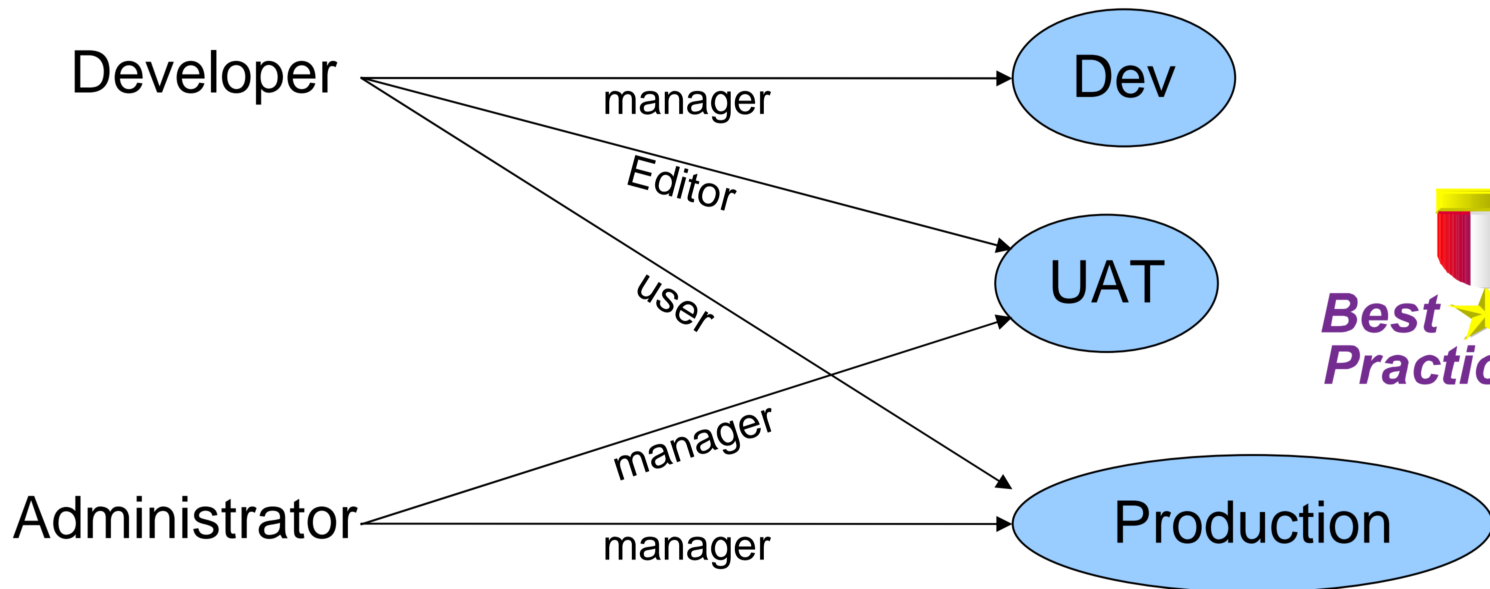
- Test soon, test often, test completely
  - Look at test-driven development
    - Write the specification
    - Write the structure
    - Write the tests
    - Write the code
  - Automatic testing means:
    - Far easier to regression test all the time
  - Minimise run-time errors by:
    - Always assuming that data operations outside the current database may fail
    - Trapping all run-time errors and providing an execution path that fails safely

How to Test (cont.)

- Why test early ?
  - Costs rise as the tests fail further away from development
  - Cheapest to most expensive are:
    - Unit tests – by the developer
    - User acceptance tests (UAT) – by the tester
    - End-user failure – by the user
  - The more tests you perform in unit testing:
    - The cheaper your development will be
    - The more professional your colleagues will think you are
    - The more your boss will like you

Where to Test

- Developers should work in a development environment
- User acceptance testing (UAT) should occur in a test environment
- Administrators should deploy templates



Where to Test (cont.)

- Why?
  - It's a basic change control environment
  - You insulate developers from the production environment
  - No changing applications on the production environment
- Does this create more work?
  - Yes. The Administrator now has to deploy templates
- Does this help testing?
  - Absolutely. It shakes out any hard-coded errors
    - These account for 15%-20% of all errors!
- Does this help developers?
  - Yes. It means that someone else tests the application before it's promoted to production
- Does this save money?
  - Yes. It increases testing in non-user facing areas

What We'll Cover

- Introduction
- Structure
- Variables
- Comments
- Cut 'n' Paste Code
- Error Handling
- Functions and Subs
- Memory Structures
- Performance
- Design Patterns
- Summary and Questions

Structure – Code Structure

- Try to structure code so that it is composed of short logic functions
  - This encourages re-use
  - Shorter code is far easier to maintain.
  - Do not put 1,600 lines of code in a single function!
- Advanced Best Practices: use “Classes”
  - This binds a data structure and the code associated with the data in one place
  - Can extend this code structure to increase functionality
  - Encourages code re-use

Structure – Code Structure (cont.)

- Try to avoid building functions and subs with lots of parameters
  - An example of tight binding
  - Brittle
  - Always changing
  - You might find that two or three functions are more appropriate
  - Try passing a type or class instead
- Few parameters
  - More re-use
  - Simpler
- Hint:
  - If you have to use "_" —the continuation character—in a function declaration, stop yourself and re-evaluate!
  - If you always have to change functions, then re-evaluate!

Code Structure – Defensive Programming

- Defensive Programming is:
  - Assuming that the consumers (in most cases, you!) of your code will pass bad data
  - Checking all dependencies before making changes
- Example:

```
Function test(param1 as String, param2 as String) as integer
   Test = false
   if param1="" then exit function
   if param2="" then exit function

   ... ' Do some work

   Test = true
end function
```

Code Structure – Defensive Programming (cont.)

- Pros:
  - Far more robust code
  - Far better at dealing with and failing gracefully when things change
  - Better at dealing with transactions
    - All dependencies are checked before a transaction commit. Consistency is maintained
- Cons:
  - More code!
  - Redundant code?
  - Performance hit? Perhaps not. In our world, only a few operations actually take a long time:
    - *Opening a NotesView*
    - *Opening a NotesDocument*

What We'll Cover

- Introduction
- Structure
- Variables
- Comments
- Cut 'n' Paste Code
- Error Handling
- Functions and Subs
- Memory Structures
- Performance
- Design Patterns
- Summary and Questions

Variable Naming

- Should I adopt a variable naming methodology?
  - If it helps!
  - For instance:
    - docCurrent, viewDocuments, sSession indicates that this variable is a document, a view or a session
    - However, it falls down because there are over 70 notes classes in v7!
  - I tend to prefix:
    - Strings with “str”
    - Integers with “i”
    - Variants with “v”
- Do not use variable names that are not meaningful
  - EG: “elephant”, “donut”, “apple”
    - Unless you work in a zoo

Variable Naming (cont.)

- Variables that only exist for a short period of time should have short names:
  - For example; Loop counters - “i”
  - But only if they are easily viewable on a screen
- Complex variables – such as NotesDocument, etc.
  - Longer names – but less than 15-20 characters
  - Make them “CamelCase” (humped)
    - For example: docPersonOldDir, instead of DOCPERSONOLDDIR
- Use UPPERCASE for 'Consts'
  - An old “C” habit where UPPERCASE indicated preprocessor variables
- Beware the accidental “rude” variable name
  - Especially in multi-lingual environments

Variable Existence

- A big part of “Code Complete” is:
  - Variables only exist as long as they are useful
- Within a large function, therefore:
  - Don’t declare all variables at the top
  - Declare the variables as you need them
  - The goal is to have the meaningful parts of a sub-section of code visible in one screen
- Why?
  - This cuts down the time required to understand & debug
    - A large part of debugging is looking at each variable and understanding where it's being changed and where it's being used. If it's all visible in one screen, this is far simpler.
  - Which in turn cuts maintenance costs

Variable Declarations

- Always use Option Declare. Why?
  - If you don't declare the variable in advance then it will default to a variant
  - This gives you a 10x performance overhead as the variant is data-converted on each operation
  - If you mistype the variable – easy to do – you end up with an implicit variable declaration – and two separate variables
- Always put code in Script Libraries instead of forms and shared actions
  - Because Option Declare is switched off for these
- Remember:
  - Declaring your variables in advance means that the LotusScript compiler is ensuring type correctness
  - It's better to shake out syntax errors in unit testing than during user testing

What We'll Cover

- Introduction
- Structure
- Variables
- Comments
- Cut 'n' Paste Code
- Error Handling
- Functions and Subs
- Memory Structures
- Performance
- Design Patterns
- Summary and Questions

What to Put in Comments

- Comments should be inserted into code to aid understanding
  - Document the why, not the what
    - Especially if you have to introduce a non-intuitive fix – such as an edge condition
  - A competent programmer can read your code
  - But don't over-comment!
- Author/Age/Last Change comments:
  - Rarely get updated
  - Mistrust them
- Comments such as the following help no one
  - "This should never get executed"
  - "Here there be dragons"

Meta Code in Comments

- A good use of comments is to write meta-code
  - For instance, when first writing a function, write the code-branches as a series of comments
  - Leave them in once the code is complete

```
Function test (param1 as String, param2 as String) as integer
    ' check the parameters and bail out if incorrect.

    ' Add Param 1 to param 2

    ' Check it's valid. If so – update...

    ' Errorhandler
end function
```

Comments Tricks

- Remember:
  - Comments consume code-space in LotusScript
  - There's a 64k – about 1,400 line – limit for each code-sequence
    - This should not normally be an issue
    - Classes, of course, are all declared in Declarations
    - In terms of classes, you can "subclass" to break up large class declarations
- Check out LotusScript Doc:

  - http://www.lsdoc.org/
  - It's a "JavaDoc for LotusScript"
  - You can annotate code to make the comments clearer

What We'll Cover

- Introduction
- Structure
- Variables
- Comments
- Cut 'n' Paste Code
- Error Handling
- Functions and Subs
- Memory Structures
- Performance
- Design Patterns
- Summary and Questions

Introduction to Cut 'n' Paste Code

- Code re-use is a good thing
  - However it should be treated with caution
  - The code should be examined to determine
    - Whether it's complete
    - Whether it's appropriate for this situation
    - Whether it requires context
    - Whether it's legal to use in this context
  - If you don't understand it
    - It will bite you later on
  - Remember:
    - You will be responsible for the application

How to use Cut 'n' Paste Code:

- Try to isolate new complex functions in separate script libraries
  - Aids debugging, isolation, etc.
  - Try to understand the context of the function
  - Isolate the impact of the function
- With all those caveats in mind, check out:
  - http://www.openNTF.org.
  - http://www.keysolutions.com/NotesFAQ.
- Remember:
  - To leave in accreditations
  - To thank the original developers
  - To check back to see if there have been updates/fixes

Resource

What We'll Cover

- Introduction
- Structure
- Variables
- Comments
- Cut 'n' Paste Code
- Error Handling
- Functions and Subs
- Memory Structures
- Performance
- Design Patterns
- Summary and Questions

Error Handling Introduction

- An Error is thrown at run-time when:
  - A resource is unavailable: database, server, memory
  - File I/O breaks
  - Etc., etc.
- Execution stops at the point of error
- Sets error information:
  - Error$ - a textual description
  - Err – a numeric error code
  - Erl – the line number of the error
- Jumps to the error handling routine
  - If there is no error handling routine in the current code-sequence, jumps to calling function
  - If top-level function has no error handler, then default error handling is used

Error Handling Effects

- If an error occurs in the middle of a critical sequence of updates, possibly some updates will happen, and some will not
  - Serious data consistency errors
  - Get all your validation done before committing transactions
  - Perform all the transaction "commits" at the same time
    - In our case, all the "saves"
  - Defensive coding minimises this effect
  - Do not use "on error goto next"
- You must deal with errors

How to Handle Errors

- Two approaches:
  - Error trap in almost every routine
    - Very granular
    - Lots of information
    - Minimal chance of breaking transactions
    - However, lots of code
  - Error trap at the top
    - Very simple
    - Minimal code
    - You will lose context – you might not be able to get valid data on how the error happened
    - You might break transactions – and therefore compromise data integrity
- I recommend:
  - Error trap in most routines!

Writing a Minimal Error Trapping Routine

- If you are going to write error trapping in most routines:
  - Keep it simple
  - Keep it cut'n'paste friendly
- This means that:
  - It might be difficult obtaining the name of the function that the error occurred in
  - Check out getThreadInfo() or lsi_info()

Example Function with Error Trapping

```
Function test(param1 as String) as integer
   Test = false

   on error goto errorhandler

   ' do something here!

   Test = true

exitFunction:
   exit function
errorhandler:
   call raiseError()
   resume exitfunction

end function
```

Example 'RaiseError()' Routine

```
Function raiseError() as integer
   raiseError = false
   ' Remember. No error trapping here!

   Print “Run time error: (“ & cstr(err) & _
      “: “ & Error$ & “ at line: “ & cstr(erl)



   raiseError = true
exit function
```

Better 'RaiseError()' Routine

```
Function raiseError() as integer
   raiseError = false

   Print "Run time error: (" + cstr(err) & _
      ": " & Error$ & " at line: " & cstr(erl) & _
      "called from: " & getThreadInfo(11) & _
      " in module: " & getThreadInfo(10)

   raiseError = true
exit function
```

Example: OpenLog

- OpenLog is an OpenNtf Project:
  - http://www.openntf.org
- It provides a simple framework for collecting error logging
  - To use in your database:
    - Copy the Script libraries from the OpenLog database
    - Update the code:

```
Function test(param1 as String) as integer
    Test = false
    on error goto errorhandler
    ' ...
    Test = true

exitFunction:
    exit function
errorhandler:
    call logError()
    resume exitfunction

end function
```

OpenNtf.org

- Example Log output:

Logged Event

Event Type:	Error
Event Time:	26/02/2004 03:47:27
Severity:	0

User Name	Julian Robichaux/nsftools	**Client Version**
Effective Name	Julian Robichaux/nsftools	Release 5.0.10 March 22, 2002 Build 166
Access Level	6: Manager	
Database Roles	[NotifyCreator]	

Error Num	13	**Error Msg**	Type mismatch	**Server**	
Error Line	3	**Language**	LotusScript	**Database**	c:\lotus\notes5\data\OpenLog.nsf
Stack Trace				**Agent**	Test LS Logging
				Method	ERRORSUB1

Error 13 on line 3 in function ERRORSUB1: Type mismatch

Other Logging

- So far we have discussed logging run-time errors
  - You may wish to log other levels of event. For instance:
    - Verbose: All variables values plus
    - Normal: Code events plus
    - Minimal: Errors
- Why?
  - You may wish to record non-error events
  - You may wish to produce performance information
- The OpenLog solution may not scale to this level
  - It creates a new document for each error raised
  - If you were to introduce verbose logging
    - The log database would become huge
    - You may see performance issues keeping up with logging

What We'll Cover

- Introduction
- Structure
- Variables
- Comments
- Cut 'n' Paste Code
- Error Handling
- **Functions and Subs**
- Memory Structures
- Performance
- Design Patterns
- Summary and Questions

Functions and Subs: Introduction

- A function or subroutine:
  - Is a good way of decomposing problems
  - Is a good way of packaging up a common solution
  - Should carry as little context as possible to maximize re-use
    - Always assume someone will cut'n'paste
  - Try and minimise the number of calling parameters
  - Decide what your error handling strategy will be
  - Beware short Function/Sub names:
    - You might “collide” with functions in newer versions of LotusScript in the future, e.g., ArraySplit
    - Not a bad idea to prefix all functions or Subs with a name-space prefix such as “MyCompany_.”

Functions and Subs: Overview

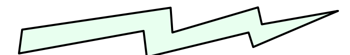
- Functions return a value
  - For very simple functions:
    - It's fine that the return value is a value
    - Make sure those values are consistent in your problem-space
  - For more complex functions:
    - It's best to return whether the function itself has worked properly
    - Pass back return values via the calling parameters
- Subs:
  - Difficult to see how they can be useful in a defensive coding world
    - How do you know they have worked ?

Functions and Subs: Approach

- Try to minimize the size of the function
  - Consider splitting up anything over 100 lines
- Don’t be afraid to create a low-level function and a number of higher-level exploitation functions
- Code Hiding:
  - Explicitly document your externally visible functions by prefixing them with Public
  - Don't be afraid to hide helper functions in your script library by prefixing them with Private
- If you find you are passing a lot of contextual information between functions, investigate types or classes to bundle information:
  - You can then pass a single object between functions
  - Simpler to add another item of data in future
  - Consider wrapping the code into a class

Functions and Subs: Approach (cont.)

- Place all functions and subs into script libraries
  - Better grouping of functions
  - Easier to debug
  - Easier to code – full syntax checking
    - You don't get full syntax checking on form events or shared actions!
  - Easier to reuse



GOTCHA!

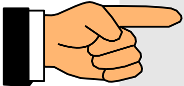
Functions and Subs: Tricks

- Use Variants to “loosely bind” parameters
  - Then you can pass in containers such as arrays or lists into the same function

```
Function test(param as variant) as integer
   test = false

   if (isContainer(param)) then
      forall thisThing in Param
         call test(thisThing)
      end forall
   else
      Print “I've been passed: “ + param
      test = true
   end if

end function
```

Functions and Subs: Tricks (cont.)

- Use Variants to “loosely bind” parameters
  - To make the function more useful




```
Function ListCollection(p as variant) as integer
   test = false

   if (typeName(p) = "NOTESVIEW") or _
   (typeName(p) = "NOTESDOCUMENTCOLLECTION") then
      ' decompose this collection into a number of
      ' single document calls to myself
      dim doc as NotesDocument
      set doc = P.getFirstDocument
      while (not doc is nothing)
         call listCollection(doc)
         set doc = p.getNextDocument(doc)
      wend
   else
      ' We've been passed a single document.
      ' Process it.
      Print "I have been passed doc: " + doc.Title(0)
      test = true
   end if

end function
```

What We'll Cover

- Introduction
- Structure
- Variables
- Comments
- Cut 'n' Paste Code
- Error Handling
- Functions and Subs
- Memory Structures
- Performance
- Design Patterns
- Summary and Questions

Memory Structures: Introduction

- Use memory structures to store collections of temporary information
  - Design your data structure correctly and save effort
- Arrays
  - Store multiple values of the same type in an array
- Lists
  - Associate a terminal value with a lookup value
- Types
  - Bundle a number of different types of data into one structure
- Classes
  - Bundle a number of different types of data into one structure
  - Associate the code for that structure

Memory Structures: Arrays

- Arrays:
  - Are non-ordered
    - That is, they retain their insertion order, but are not sorted by value
  - Can be of fixed or dynamic size
  - Store any other variable type
- Example:

```
Dim myFixedArray(1) as String

myFixedArray(0) = “First”
myFixedArray(1) = “Second”

Print myFixedarray(0)
Print myFixedarray(1)
```

```
Dim myDynamicArray() as String
redim myDynamicArray(0)

myDynamicArray(0) = “First”

redim preserve myDynamicArray(ubound(myDynamicArray)+1)

myDynamicArray(ubound(myDynamicArray)) = “Second”

forall thatEntry in myDynamicArray
    Print “That entry is: “+ thatEntry
end forall
```

Memory Structures: Arrays (cont.)

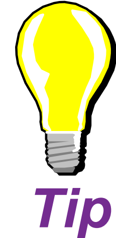
- By default, they count from zero
  - Don't change this by using “option base”
  - You can guarantee that someone will either:
    - Cut'n'paste this code somewhere else
    - Remove your “option base” directive!

Heads Up!

- Extending arrays is a relatively expensive operation:
  - When building collections of more than 20 data items, don't extend the array item by item

Memory Structures: Lists

- Lists are:
  - Non-ordered
    - That is, the memory layout does not reflect their value or their index value
  - Indexed by a unique lookup value
  - Very fast & memory efficient
- Example:




```
Dim myList List as String

MyList (1) = “First”
MyList (2) = “Second”

forall listEntry in myList
     Print “This list entry is: “+ listEntry
end forall
```

Memory Structures: Types

- Types:
  - Help bind together multiple values
- Example:

```
Type Customer
      name as NotesName
      UNID as String
      Department as String
end type

Dim newCustomer as Customer

newCustomer.Name = new NotesName(“Joe Bloggs/Acme”)
newCustomer.UNID = doc.UniqueID
newCustomer.UNID = doc.Department(0)
```

Memory Structures: Types (cont.)

- Pros:
  - Binds data together
  - Functions can then use this type as a parameter
  - Makes the data structure easier to extend
  - You can copy simple types
- Cons:
  - Functions which deal with this type have to understand the data associated with this type
    - The functions are therefore tightly bound to this type
  - More complex types – containing arrays or lists – cannot be automatically copied
    - You have to write code for this

Memory Structures: Classes

- Classes
  - Bind data and code together
  - May be extended from another class
  - May contain (“Encapsulate”) other classes
- Example

```
Class Customer
      Public name as NotesName
      Public UNID as String
      Public Department as String

      Public Sub new(doc as NotesDocument)
            me.Name = new NotesName(doc.name(0))
            me.UNID = doc.UniqueID
            me.UNID = doc.Department(0)
      end sub
end type

Dim newCustomer as Customer(doc)
Print “My customer name is: “ + newCustomer.Name
```

Memory Structures: Combinations

- Combining Lists and Classes is particularly useful:

```
Dim CustomerList List as Customer

dim doc as NotesDocument
set doc = vCustomers.getFirstDocument

while not doc is nothing

	dim C as new Customer(doc)
	set CustomerList(C.name) = C
	set doc = vCustomers.GetNextDocument(doc)
wend

forall thisCustomer in CustomerList
	Print “Customer: “ + thisCustomer.Name + “ is in Department: “ + _
		thisCustomer.Department
end forall
```

Memory Structures: Encapsulation

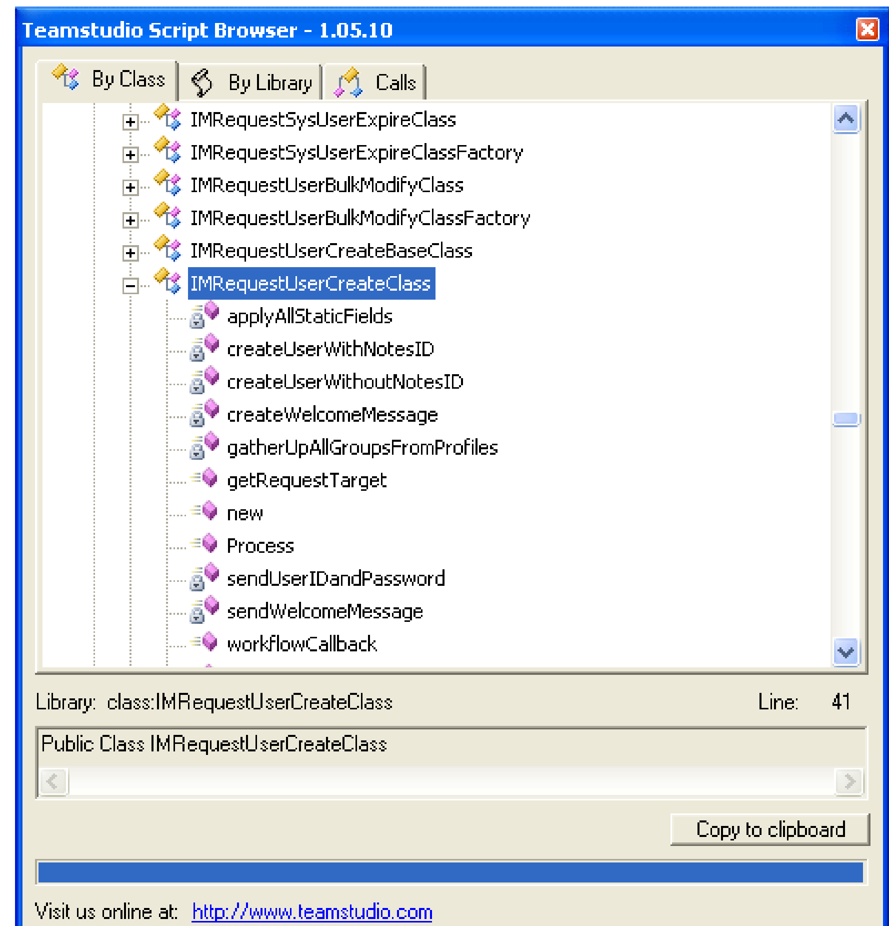
- Classes can contain other classes:

```
Class Customers

    Public CustomerList List as Customer

    Public function load(vCust as notesView) as integer
        dim doc as NotesDocument
        set doc = vCust.getFirstDocument

        while not doc is nothing
            dim C as new Customer(doc)
            set CustomerList(C.name) = C
            set doc = vCust.GetNextDocument(doc)
        wend
    end function

    Public Function Print as Integer
        forall thisCustomer in CustomerList
          Print "Customer: " + thisCustomer.Name + " is in Department: " + _
            thisCustomer.Department
        end forall
    end function
end class
```

```
Dim CL as new CustomerList
call CL.load(viewCustomers)
call CL.Print
```

Memory Structures: Classes: Implementation

- Classes are all defined in the declarations section.
  - Difficult to navigate and edit
  - Keep classes small – 1000 lines maximum
- No Class Browser
  - Check out http://www.teamstudio.com for Craig Schumann's LotusScript Class Browser



Memory Structures: Usage

- Use these memory structures to:
  - Build intermediate information in memory
  - Cache intermediate results
- Much more efficient to use memory than:
  - Re-open Notes documents multiple times
- Where would I use them ?
  - Directory or data synchronization tools
  - Reporting or analysis tooling
  - Database content checking tools
  - Complex data structure management

What We'll Cover

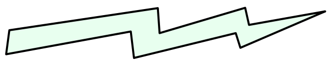
- Introduction
- Structure
- Variables
- Comments
- Cut 'n' Paste Code
- Error Handling
- Functions and Subs
- Memory Structures
- Performance
- Design Patterns
- Summary and Questions

Performance: Introduction

- I said "Architect for Maintainability"
  - This is still true
  - However, sometimes – especially on large data runs – you need to know how to perform better
  - Don't prematurely optimize
- Expensive operations:
  - Opening a NotesView
  - Opening large number of NotesDocuments
- Cheap operations:
  - Holding something in memory
- Why ?
  - CPU is now very very cheap
  - Memory is now very cheap
  - Disk I/O – whilst faster – is still thousands of times slower than accessing memory

Performance: Views

- Fast
  - Use "NotesViewEntries" to parse data from existing view summary information
- Medium
  - Use NotesViews to iterate through documents
  - Use NotesDocumentCollections to iterate through documents
- Slow
  - Looking up every single document
- Very slow
  - Opening up every single document, and using "extended class syntax" to obtain data fields:
    - Always use "GetItemValue" instead
      - *This will reduce incorrectly named fields*
      - *You can get type conversion*
      - *doc.GetItemValue("field")(0) is faster than doc.Field(0)*



GOTCHA!

What We'll Cover

- Introduction
- Structure
- Variables
- Comments
- Cut 'n' Paste Code
- Error Handling
- Functions and Subs
- Memory Structures
- Performance
- Design Patterns
- Summary and Questions

Design Patterns: Introduction

- Design patterns are:
  - Generally accepted Best Practices for designing part of an application
  - Prevalent in the Java world
  - A good way of solving common problems
- Design patterns are not:
  - A replacement for actually designing the application!
  - To be used prescriptively
    - After all, if all of our job could be automatically performed, we would not be required!
- We shall discuss:
  - A common Notes design pattern
  - An object oriented design pattern: Singleton

Design Pattern: Notes Replication

- Before Notes/Domino v6, scheduled agents were not allowed to open databases on other servers
  - This led to catalog.nsf style designs, where:
    - A single database was replicated to multiple servers
    - A "run on all servers" agent collected data
    - All the data was replicated to all servers
  - This in turn leads to a huge replication overhead, pushing all that data back to all replicas
- In v6.x, you can assign "Trusted Servers"
  - Defined in the Server Document on the security Tab
  - List all servers that this server is allowed connections from
  - Agents can then open databases on other servers

Trusted Servers vs Replication: Application

- Architecturally, you may now:
  - Have a single central database
  - A single scheduled agent
  - Now pushes and pulls data to and from remote servers
- This changes the architecture of applications
  - Reduces data replication
  - Increases response rate – no longer waiting for replication schedules
  - Depends on reliable network links

Design Pattern: Singleton

- The Singleton is:
  - A method of ensuring that only one object of a particular type is created
  - Also a very fine Scottish malt whiskey
    - Do not mix these two up!
- Particularly useful for:
  - Configuration class objects
    - An object that holds global configuration information about your application

Design Pattern: Singleton Implementation

- Create a class in a script library
  - Let's call it "ConfigurationClass"
  - Let's create it in script library "Class:ConfigurationClass"
  - It contains all global setup information for an application
  - It loads itself from a profile doc in the current database
  - It is used by the application to get configuration variables

```
Class ConfigurationClass

	Public ConfigVar1 as String

	Sub new ()
		dim sSession as new NotesSession
		dim dbThis as NotesDatabase
		set dbThis = sSession.CurrentDatabase
		dim docProfile as NotesDocument
		set docProfile = dbThis.GetProfileDocument("Configuration")

		ConfigVariable1 = docProfile.GetItemValue("ConfigVariable1")(0)
	end sub
end class
```

```
Use "class:ConfigurationClass"

sub initialize()
	dim C as new ConfigurationClass()

	Print "Config is: " + C.ConfigVar1
end sub
```

Design Pattern: Singleton Implementation (cont.)

- The code so far does not load a singleton
  - It loads a new copy every time it's called
- To make ConfigurationClass a singleton
  - Create a global function called "GetConfiguration"
  - Create a global variable called CurrentConfiguration
  - Make the class "Private"

```
Use "class:ConfigurationClass"

sub initialize()
        dim C as ConfigurationClass()
        set C = getConfiguration

        Print "Config is: " + C.ConfigVar1
end sub
```

```
Private CurrentConfiguration as ConfigurationClass

Public Function getConfiguration as ConfigurationClass
        if CurrentConfiguration is nothing then
                set CurrentConfiguration = new ConfigurationClass()
        end if

        set getSingleton = CurrentConfiguration
end function
```

What We'll Cover

- Introduction
- Structure
- Variables
- Comments
- Cut 'n' Paste Code
- Error Handling
- Functions and Subs
- Memory Structures
- Performance
- Design Patterns
- Summary and Questions

Resources

- Code Complete 2
  - Steve McConnell, Microsoft Press
- http://www.openNtf.org
- http://www.keysolutions.com/Notesfaq
- http://www.notes.net
  - For forum and for support database
- My “Object Orientated LotusScript” Presentation

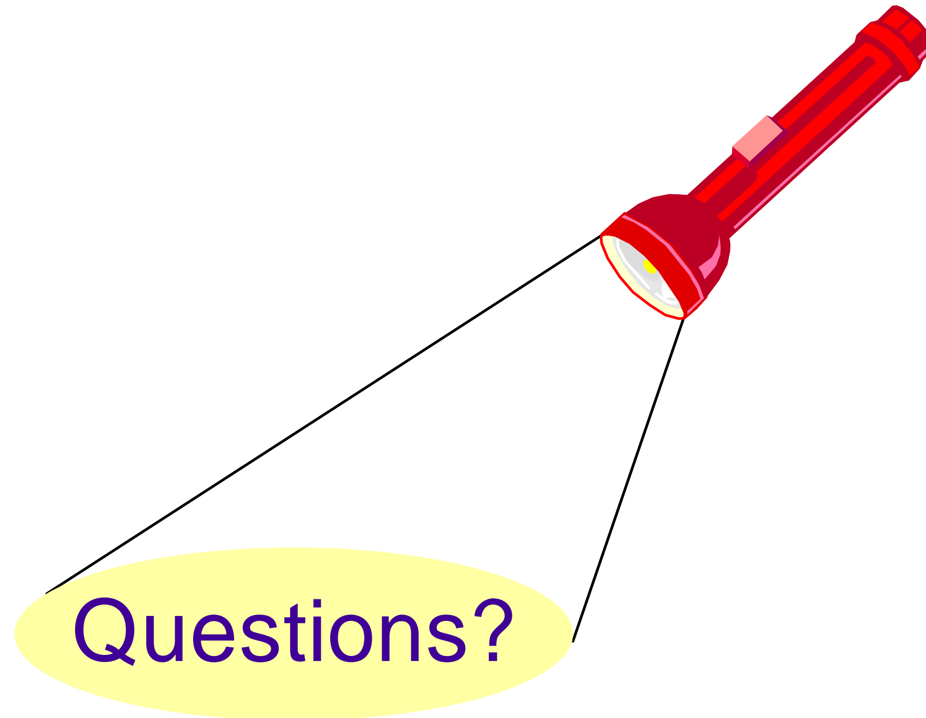
Resource

Key Points to Take Home

- Best Practices
  - Don't be too prescriptive – use them as guidelines
  - Sometimes using Best Practices means more work
  - Always be willing to learn new techniques
  - Always be willing to discuss implementation with other parties – this will always pay off
  - Always code for maintenance. It's the highest cost
  - Make your code easy to deploy
    - Your Administrators will be more forgiving
  - You are not alone – use the on-line forums to investigate

Checklist

Your Turn!



How to Contact Me:
Bill Buchan
Bill@hadsl.com