4GL Coding Worst Practices
aka: The Keyword Forget List According to Tom
Tom Bascom, White Star Software

Abstract: Just because you CAN-DO() it doesn't mean that you should do it. Programmers are drawn to bad examples likes moths to flame. No matter how many carefully crafted comments you put around that one usage of a special case some programmer will find it and suddenly propagate it throughout your application.

Come to this session to learn why CAN-DO(), FIND FIRST, RELEASE and many, many other coding worst practices should be banned! Or at least learn why you should not continue to use these worst practices just because "everyone else always does it that way".
4GL Coding Worst Practices
aka: The Keyword Forget List According to Tom

Tom Bascom, White Star Software
tom@wss.com
A Few Words about the Speaker

• Tom Bascom: Progress user & roaming DBA since 1987
• Partner: White Star Software & DBAppraise, LLC
  – Expert consulting services related to all aspects of Progress and OpenEdge.
  – Remote database management service for OpenEdge.
  – Author of: protop
  – Simplifying the job of managing and monitoring the world’s best business applications.
  – tom@wss.com
• I do not work for Progress...
• ... and after this we can be pretty sure that I never will ;)
• The content of this session is heavily laced with opinion!
“Deprecated” = not quite “banned” 😕

knowledgebase.progress.com/articles/Article/P67734

• Code will still “work” but it is strongly recommended to start modifying your code.
• Technical Support will do their best to help.
• No escalations will be issued on deprecated statements and clauses.
• Not guaranteed that old code with deprecated statements will still work in future releases.
Progress’ Officially Deprecated Features

• UPDATE EDITING
• GO-PENDING
• CHOOSE
• SCROLL
• PUT SCREEN
• IS-ATTR-SPACE
• GATEWAYS
• SQL (within the 4gl)
Tom’s List!
DOS

• Even worse than Windows!
DOS

• Even worse than Windows!
• And UNIX and VMS and BTOS for that matter
• These statements bind your code to a particular OS.
• Your code is less flexible and more brittle...
• Instead use OS-COMMAND VALUE( cmd )
  – CMD can be stored in a config table.
  – Perhaps as a template that you use with SUBSTITUTE
if opsys = "unix" then
    dbaCmd =
    "\$DLC/bin/proutil &1 -C dbanalys -Bp 10 >" +
    " \$PROTOP/dbanalys/&2.dba 2>&1 &&"
.
else
    dbaCmd =
    "%DLC%~\bin~\proutil &1 -C dbanalys -Bp 10 >"
    " %PROTOP%~\dbanalys~\&2.dba 2>&1 &&"
.
os-command silent value( substitute( dbaCmd, pdbname(1), ldbname(1))).
Unquoted File Names

- Unfortunately, this code works:
  ```
  output to test.
  put "hello kitty".
  ```

- IMHO that should be a compiler error.

- Allowing unquoted file names is a bug waiting to happen:
  ```
  define variable test as character no-undo.
  test = "xyzzy".
  output to test.
  put "hello kitty".
  ```
CALL

- This is the old “HLC” interface.
- You have to relink your _progres executable to use it.
- The capabilities that it supports have been replaced by the ability to call shared libraries, even on UNIX:
  
  http://dbappraise.com/ppt/shlib.pptx
DEFINE WORK-TABLE (WORKFILE)

• Pre-dates temp-tables and prodatasets.
• No indexes - so searches are slow
• No fancy, modern features like READ-JSON etc.
• Limited by available RAM
• Ungraceful failures when you run out of RAM
  – Sessions crashes
  – Maybe even the whole machine 😞
ACCUMULATE etc.

• Aggregate phrases, in general, are too complex and finicky.
• AVERAGE, TOTAL, etc.
• IMHO it is easier, and a lot clearer, to explicitly create the intermediate variables and do the calculations.
• Much less likely to be messed up by a maintenance programmer.
There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.

— C.A.R. Hoare, The 1980 ACM Turing Award Lecture

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

Brian W. Kernighan and P. J. Plauger in *The Elements of Programming Style.*
• Validation expressions have no usefulness in modern code. They apply too late in the data entry cycle to be useful (validations are performed just before data is written to the db). Don't use them. You're on the wrong path.

• If you change a validation expression that change isn't noticed by code until you recompile.

• Validation expressions made good demos back in the day but have never really been useful for real code.

• The role of validation expressions is better filled by a “rules engine” (such as Corticon).
• Almost always a sign that the programmer is confused about record locking or transaction scope and does not read documentation very carefully.
Verifies that a record complies with mandatory field and unique index definitions. It clears the record from the buffer and unites it to the database if it has been changed.
RELEASE and Record Locks

message "before find".
find customer exclusive-lock where cust-num = 2.
display cust-num name discount.
message "before update".
update discount.
message "before release customer".
release customer.
message "after release customer".

<table>
<thead>
<tr>
<th>RECID</th>
<th>Table Flags</th>
<th>Usr</th>
</tr>
</thead>
<tbody>
<tr>
<td>386</td>
<td>2 X X</td>
<td>6</td>
</tr>
<tr>
<td>386</td>
<td>2 S X</td>
<td>6</td>
</tr>
</tbody>
</table>

White Star Software
Use proper record and transaction scoping and you will never need to use RELEASE:

define buffer upd_cust for customer.

for each customer no-lock:
  if customer.discount > 10 then
    do FOR upd_cust transaction: /* strong scope the “upd_cust" buffer */
      find upd_cust exclusive-lock where upd_cust.custNum = customer.custNum.
      upd_cust.discount = 10.
    end.
  end.
CAN-DO()

• A long, long time ago...
• There were no list manipulation functions.
• LOOKUP, ENTRY etc. was not in the 4gl...
• Someone noticed that CAN-DO could be subverted to perform this common task:

  display can-do( "a,b,c", "z" ).
CAN-DO ... more history

• A *lot* of code was created by a very early, very successful VAR using this technique.

• Unfortunately a great many Progress programmers have been exposed to that code.

• That does not, however, make it a “best practice” or even a “good idea”.

• At best it is a “bad habit”.

White Star Software
Just Because you CAN-DO...

... doesn’t mean that you SHOULD-DO

Using a security function to perform string operations is misleading. Rather than having plain English self-documenting code you are abusing a function to achieve a purpose that it was never intended for.
Using the CAN-DO function as a substitute for the LOOKUP function constitutes a misuse of the CAN-DO function and is strongly discouraged.

The CAN-DO function is explicitly designed to match user-id patterns. Using it for arbitrary string comparisons is not intended and will provide undefined results.
CAN-DO() alternative

• Bad:
  
  if can-do( ‘ME,NH,VT,MA,CT,RI’, ‘NH’ ) = true then ...

• Good:
  
  if lookup( ‘NH’, ‘ME,NH,VT,MA,CT,RI’ ) > 0 then ...
But CAN-DO() supports wild-cards!

• 99% of those uses are covered with BEGINS, MATCHES or COMPARE

• But if you really need it:

/* credit to Kurt Gunderson, PEG July 23 2015 */

function myCANDO returns logical ( input t as char, input s as char ):

    return (( lookup( "*", s ) <> 0 or lookup( t, s ) <> 0 )
      and ( not lookup( "!" + t, s ) <> 0 ))

end function.
... about those “wild-cards”

- CAN-DO( "a,b,#", "#" ) returns false!
- CAN-DO uses some magic characters:
  - .: Match a single character
  - *: Match multiple characters
  - !: Negate a match
  - #: Distinguish GRANTable vs. non-GRANTable permissions in SQL89
  - @: Used to represent the blank userid (as of oe11)
  - “ “: Spaces are not allowed
- Other quirks could be added at any time!
And if that all that wasn’t Bad Enough

- CAN-DO() in a WHERE clause *forces* client-side selection and sorting.
CAN-DO() Summary

• CAN-DO is a **security** function!

• Special characters in the data will cause it to produce unexpected results! (aka **bugs**)

• Because it is all about userids it **must** be evaluated by the **client**.
  – So if it is in a WHERE clause you **will** have a table scan!
  – CAN-DO cannot and will not be evaluated server-side.
OF

- Implied WHERE clause
- Uses a "common names that are indexed imply JOIN" convention.
- Obscures table relationships.
- Like VALIDATE, OF “makes a good demo”.
- But should never be used in real life.
USE-INDEX

• Overrides Progress’ optimizer:
  – You are probably not smarter than the compiler*
  – I certainly am not.

• Forces a single index to be used.

• WHERE clause mismatches will result in very poor performance.

• On the bright side USE-INDEX provides an ordering:
  – May not be clear or obvious – index naming conventions can be missing, misleading or wrong.
  – May conflict with BY phrase and cause client side sorting.

* Trying to outsmart a compiler defeats much of the purpose of using one.
   — Kernighan and Plauger, The Elements of Programming Style. 32
• Still think you should use it?
  – Prove it!
  – Using realistic and meaningful test data.
  – Provide detailed and reproducible test cases in your comments so that future programmers will recognize and celebrate your brilliance.
FIND FIRST (and LAST)

- Reflexive and automatic use on each and every FIND does NOT improve your code.

- It is NOT a “standard”.
- Nor is it a “best practice”.
- Nor does it “always work”.

- Yes, I know it is all over the place in certain code bases.
Unique FINDs

• FIND is designed to return exactly one or zero records.
• 99.44% of FIND statements should be for UNIQUE records.
• This is one of Progress’ big advantages over SQL.
• If the WHERE clause specifies a unique record then FIRST adds no value.
• Worse – it confuses the maintenance programmer by implying that there /should/ be an ordered result-set.
Unique FIND FIRST performance

• It is NOT faster.
• It does NOT “eliminate a check for ambiguous records”.

\[
\begin{align*}
\text{FIND FIRST customer NO-LOCK.} \\
\text{FIND customer NO-LOCK WHERE custNum = 1.} \\
\text{FIND FIRST customer NO-LOCK WHERE custNum = 1.}
\end{align*}
\]

• All of the statements above take the same time to run and have the same “logical” impact on the db engine.
• All statements execute the same number of “logical IO ops” (ProTop, PROMON or VST “block access”).
• Feel free to test it yourself!
Faster FIND with FIRST?

• But what if FIRST does actually make a query faster?
  – You have not specified UNIQUE criteria!
  – You are missing an appropriate index to match your WHERE clause.
  – Maybe your WHERE clause isn’t doing what you think it should be doing?
Returning the Wrong Record Faster!

*A program that produces incorrect results twice as fast is infinitely slower.*

— John Osterhout
FIND SECOND?

• You used FIND FIRST anyway… what are you doing about the second record?

• If there actually is a second record and you are actually doing something with it:
  – How did you specify the ordering?
  – If you don’t care about order – what does FIRST mean?
  – Are you treating it exactly the same as the FIRST record from a 3NF perspective?
  – Are you processing the entire result set? Why didn’t you use FOR EACH?
"It Always Works"

- This usually means that the programmer does not want to deal with:
  
  More than one Customer records found by a unique FIND. (3166)

- Adding FIRST will “make it go away”.

- It also makes your result is potentially wrong:
  - What if you forgot a component of the index?
  - Or didn’t know that a previously unused feature has been enabled by the users?
  - Or the users suddenly create a second magical record?
Magic FIRST Records

- Records that are special by convention.
- No specific attribute identifies the usage.
- A clear violation of Third Normal Form.

```
find first customer no-lock where custNum > 0.
display custNum name discount.
defaultDiscount = discount.

find first customer no-lock where name > "".
display custNum name discount.
defaultDiscount = discount.
```
FIND FIRST Summary

- FIND FIRST is almost always a sign of lazy programming
- It does not improve performance
- It can create bugs and mask existing bugs
- Some code bases are infested with it – but there is no reason to make the problem worse by continuing the habit
for first customer no-lock where discount > 10:
leave.
end.
display custNum name discount.

for each customer no-lock where discount > 10 by discount:
leave.
end.
display custNum name discount.

for first customer no-lock by discount:
leave.
end.
display custNum name discount.

/* continue... */
define variable minDiscount as decimal no-undo initial 99999.
define variable minDiscountCustNum as integer no-undo.

for each customer no-lock:
    if discount < minDiscount then
        assign
            minDiscount = discount
            minDiscountCustNum = custNum
    end.

find customer no-lock where custNum =
minDiscountCustNum.display custNum name discount.

/* continue... */
### FOR FIRST (results)

<table>
<thead>
<tr>
<th>Cust Num</th>
<th>Name</th>
<th>Discount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Lift Tours</td>
<td>35%</td>
</tr>
<tr>
<td>1067</td>
<td>Armadale Fitness</td>
<td>15%</td>
</tr>
<tr>
<td>1</td>
<td>Lift Tours</td>
<td>35%</td>
</tr>
<tr>
<td>6</td>
<td>Fanatical Athletes</td>
<td>0%</td>
</tr>
</tbody>
</table>

- So what does “FIRST” mean?
FOR FIRST (and LAST)

• Given the same WHERE clause:
  – FIND only ever uses one index
  – FOR *could* use multiple indexes

• Sorting occurs **after** selection.
  – FIRST and LAST produce a single record (thus there is nothing to sort).
  – Based on whatever index was chosen.
  – BY is an index selection tie-breaker – it is **not** the primary optimization.
  – Selection might not match your hoped for order.
Summary

• DOS et al
• Unquoted File Names
• CALL
• DEFINE WORKFILE
• ACCUMULATE & Aggregate Phrases
• VALIDATE

• RELEASE
• CAN-DO
• OF
• USE-INDEX
• FIND FIRST
• FOR FIRST
Questions?
Suggested Additions?
Thank You!