A Pattern for Reusable RPG Code

with ILE

Presented by

Scott Klement

http://www.scottklement.com

"There are 10 types of people in the world.
Those who understand binary, and those who don't."

Objectives Of This Session

- Understand the "Pattern" Concept
- Important Concepts for Engineering ILE Applications
- Links to More Information

You'll want to be familiar with ILE concepts (procedures, modules, service programs) to get the most out of this session.

So You've Learned ILE…

Over the past few years of talking to folks who learn ILE concepts, I've noted:

- People leave the conference understanding the basic concepts
  - Procedures, modules, service programs, binding and activation groups
- People get back to the office and try to use them, but…
  - Can't find a use for the concepts
  - Find the concepts "too complicated" to use in the real world.
  - Can't get the concepts to fit into the way they're used to writing programs.
  - Find that ILE decreases their productivity.

There's more to learning ILE than understanding the basic concepts!

A Different Way of Thinking

People find it easy to learn RPG IV, but hard to incorporate ILE because it requires changing the way you think about your code.

You see, as we write code, we develop "patterns" in our minds

- Mental "templates" (or "skeletons") of how a program should work
- Over time these become habits!
- Or, even become the way we think about software development
- Indeed, amongst software engineers, many patterns have been written down and "formalized" – but this talk isn't about the specific, formal patterns.

When your only tool is a hammer, all problems start to look like a nail.

-- or –

When a hammer is the only tool you've ever used, a screwdriver seems alien, unnatural, and slow.

Simple Pattern Example (Screen)

```
01/01/08                        Customer Maintenance


       Customer to change:    12345


F3=Exit    F10=Add New Customer
```

The way we used to do things in green screens involved a pattern – this example illustrates the first screen of a customer maintenance program. Here the user keys in a customer number… this number is used to load a customer record that will be changed on the next screen.

Due to the limited amount of space in this presentation, I'm only showing the first screen.

Simple Pattern Example (RPG/400)

```
C           SCRN1     BEGSR
C           SCMSG     DOUEQ*BLANKS
 *
C                     EXFMTCUSTS1
C                     MOVE *BLANKS    SCMSG
 *
C           *IN03     IFEQ *ON
C                     MOVE *ON        *INLR
C                     RETRN
C                     ENDIF
 *
C           SCCUST    CHAINCUSTFILE             N99
C           *IN99     IFEQ *ON
C           *IN10     ANDEQ*OFF
C                     MOVELERR,1      SCMSG
C                     ENDIF
 *
C           *IN99     IFEQ *OFF
C           *IN10     ANDEQ*ON
C                     MOVELERR,2      SCMSG
C                     ENDIF
 *
C                     ENDDO
C                     ENDSR
```

Simple Pattern (Description)

The RPG/400 code sample on the last slide shows one pattern that I used in all of my old programs.

1. Put a "message" field on the screen.
2. Loop until the message field is blanks (no errors)
3. Display the screen.
4. Clear the message.
5. Check for exit keys (F3, F12)
6. Check the user's input (often combined with loading records for later)
7. Repeat loop (per 2, above)

In this pattern, the bulk of the work was usually in checking the user's input. On a more sophisticated screen, it might involve checking the validity of a price, verifying that we had adequate stock to fulfill an order, and so forth.

Essentially, the business logic was interwoven into the loop for the display logic.

Same Pattern, Different Program

```
01/01/08                    Add Item to Order

            Order:   61435

         Customer:   12345

             Item:   54321___

              Qty:   ___________

            Price:   _______________


F12=Cancel
```

This program adds an item to an existing customer order. This is the 2nd screen in the program – the order number and customer number have already been established – the user needs to type an item number, quantity and price for the new item.

Same Pattern Example (1 of 2)

```
C           SCRN1     BEGSR
C           SCMSG     DOUEQ*BLANKS
 *
C                     EXFMTADDITEMS2
C                     MOVE *BLANKS    SCMSG
 *
C           *IN12     IFEQ *ON
C                     LEAVESR
C                     ENDIF
 *
C           SCCUST    CHAINCUSTFILE             99
C           *IN99     IFEQ *ON
C                     MOVELERR,1      SCMSG
C                     ITER
C                     ENDIF
 *
C           SCITEM    CHAINITEMFILE             99
C           *IN99     IFEQ *ON
C                     MOVELERR,2      SCMSG
C                     ITER
C                     ENDIF
 *
C           PRCKEY    KLIST
C                     KFLD            CUZONE
C                     KFLD            CUTRAD
C                     KFLD            SCITEM
```


Same Pattern Example (2 of 2)

```
C           PRCKEY    CHAINPRCFILE              99
C           *IN99     IFEQ *ON
C                     MOVELERR,3      SCMSG
C                     ITER
C                     ENDIF
 *
C           SCPRIC    IFEQ 0
C                     Z-ADDPRPRIC     SCPRIC
C                     MOVELERR,4      SCMSG
C                     ITER
C                     ENDIF
 *
C           SCPRIC    IFLT PRLPRC
C                     MOVELERR,5      SCMSG
C                     ITER
C                     ENDIF
 *
C           SCPRIC    IFGT PRHPRC
C                     MOVELERR,6      SCMSG
C                     ITER
C                     ENDIF
 *
 * . . .   Code to validate quantity goes here . . .
 *
C                     ENDDO
C                     ENDSR
```

What's Wrong With That?

Business logic interwoven is into display, it's difficult to separate the two.

- What if I wanted to apply the same business rules in a non-interactive program?
- What if I wanted to have a different screen interface? (Web? GUI?)
- Or even a different 5250 application, for that matter?

Maybe I'd copy/paste/modify the code – then I'd have the business rule in many places!

- When logic has to change – it's a chore.
- Maybe I'd use a copy book??
  - But the code would have to be written carefully.
  - Changes to code in the copybook would require careful, detailed analysis.

Agility problems

- If it's difficult to change your code – your company becomes less "agile"
- Harder to change with the times.
- Your program code should be dictated by your business needs! Not vice-versa!
- IT becomes a hindrance to the company.

What's Wrong – Example

In the "Same Pattern Example" slides, I illustrated code to calculate a price

- Look up the customer's trade class and price zone.
- Look up a price record in the price list file
- If the user gave no price, use the list price
- If the user gave a price, make sure it's within the "high/low" range.

In the real world this might be more sophisticated

- Based on the current livestock prices (or other market pricing)
- Based on raw material costs, etc.

My code is interwoven into the display logic – so it ends up being repeated

- Retail Sales
- Food Service Sales
- EDI sales
- Web orders
- Batch updates for market pricing
- "Add-To" program (from example)

. . . Now management decides to price things differently . . .

Ground Rules for a New Pattern

It's important to take a new approach -- a new pattern. One that makes it easier to write re-usable business logic. Ultimately the goal is to write our business logic (such as how to calculate and validate a price) only once.

MVC (Model, View, Controller)

The code that implements business rules must be kept separate from the code that implements the user interface.

SOA (Service-Oriented Architecture)

Business logic should be organized into a set of re-usable "services".

Name Spaces

Business logic routines must be callable from anywhere without danger of naming conflicts.

Encapsulation

The business logic should know NOTHING about the display logic, and vice-versa. This prevents them from becoming interwoven.

More About Naming

A symbolic prefix is determined. Up to 7 chars.
- ORDER for service progam that works with orders.
- CUST for service program that works with customers
- PURCH for purchase orders
- FGI for finished goods inventory
- etc.

- Modules (*usually only one*) are named with that prefix, followed by an optional number, and language ID.
  - ORDERR4, ORDER2R4, ORDER3CL
  - CUSTCL, CUST2R4
  - etc.

- Subprocedures ("services") are prefixed as well.
  - ORDER_new()
  - ORDER_getHeader()
  - ORDER_getShipTo()
  - ORDER_getBillTo()
  - ORDER_getAllItems()
  - ORDER_setItem()
  - ORDER_checkItem()
  - ORDER_checkPrice()
  - ORDER_error()

TIP:
Namespaces also make it easier to read/debug the code. When you see a routine being called, you know which service program it's calling!

Compare:
`CheckAPrice()`
vs:
`ORDER_checkPrice()`

- Service program is same as "first" module.
- Copy book is the symbolic prefix with an _H (for "header") appended.

With Existing Naming Conventions

Many shops already have a naming convention in place that's based on the IBM i limit of 10 characters per object name. They use this convention to get all of the information they need into the name.

Here's an example of one such convention:

RGORP941 (or ORP941RG, or ORP941, etc)

RG = RPG language
ORP = application (abbreviation for order processing)
941 = number to make this program's name unique.

This works very well as the "prefix" under the new pattern. For example:

- **RGORP941_newOrder()**
- **RGORP941_getHeader()**
- **RGORP941_getItems()**
- Etc.

This makes the names in the older naming convention easier to understand, while still preserving the value of the naming convention.

M V C Pattern

The pattern I use for ILE applications is very much based on the "MVC" pattern.

M = Model -- this is your business logic and business rules. (also database)

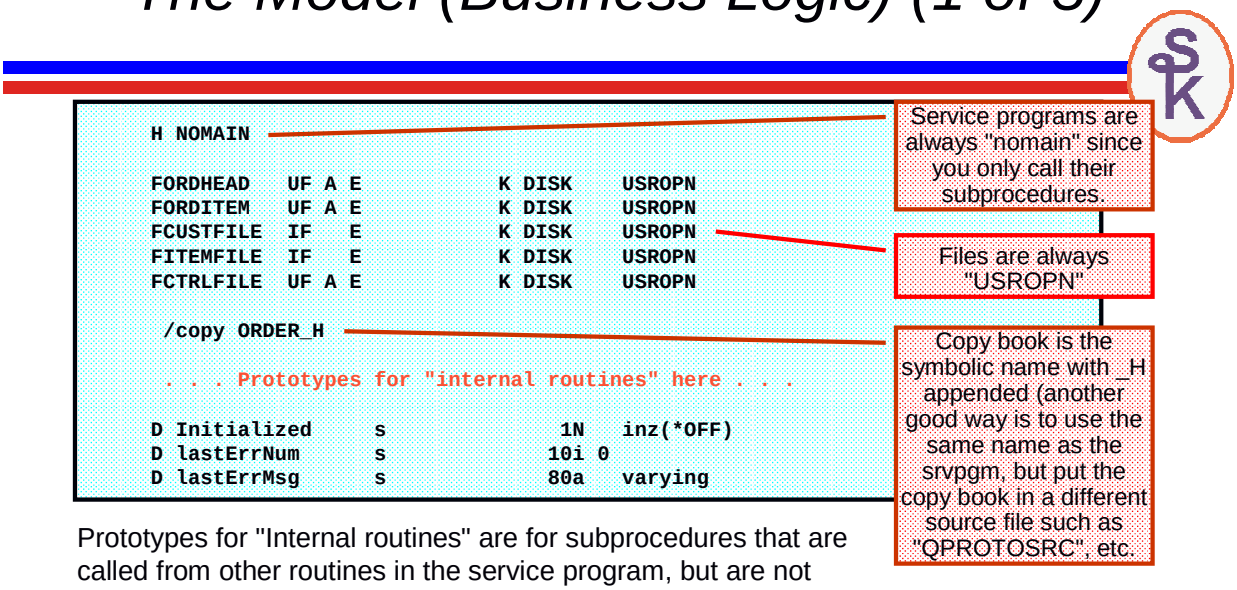
V = View -- this is the user interface. (Important: Not always a screen!)

- Might be a 5250 (green screen display)
- Might be a batch job, and the user interface is a spooled file.
- Might be a web interface
- Etc.

C = Controller -- the piece of code that controls the flow of the application.

- Calls the correct "model" routines and "View" routines
- Called in the right order to make an application.
- Passes data between them

The Model (Business Logic) (1 of 5)



```
H NOMAIN

FORDHEAD   UF A E           K DISK    USROPN
FORDITEM   UF A E           K DISK    USROPN
FCUSTFILE  IF   E           K DISK    USROPN
FITEMFILE  IF   E           K DISK    USROPN
FCTRLFILE  UF A E           K DISK    USROPN

 /copy ORDER_H

  . . . Prototypes for "internal routines" here . . .

D Initialized     s              1N   inz(*OFF)
D lastErrNum      s             10i 0
D lastErrMsg      s             80a   varying
```

Service programs are always "nomain" since you only call their subprocedures.

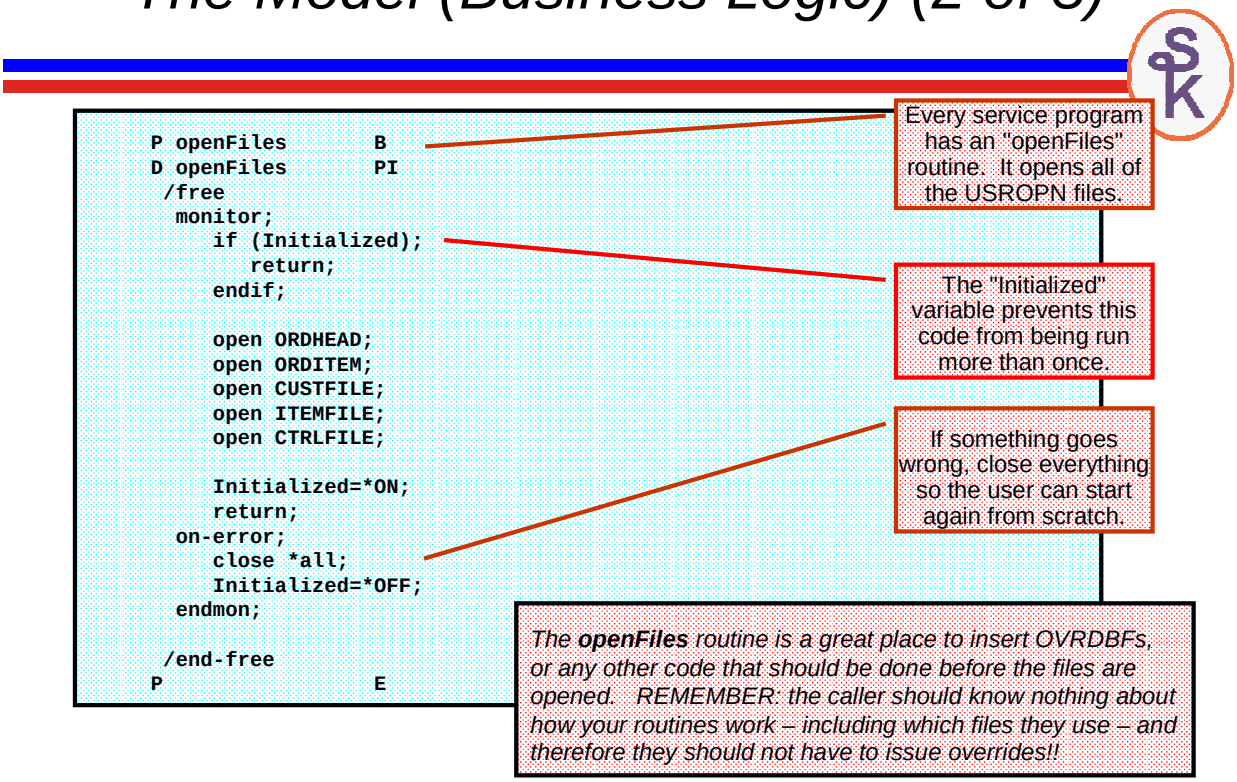
Files are always "USROPN"

Copy book is the symbolic name with _H appended (another good way is to use the same name as the srvpgm, but put the copy book in a different source file such as "QPROTOSRC", etc.

Prototypes for "Internal routines" are for subprocedures that are called from other routines in the service program, but are not available outside of the service program. Restricting who can call routines improves "agility" – when you know that nothing else calls a routine, it's easy to change it.

Prototypes for exported routines are in the copy book.

The Model (Business Logic) (2 of 5)



```
P openFiles       B
D openFiles       PI
 /free
  monitor;
     if (Initialized);
        return;
     endif;

     open ORDHEAD;
     open ORDITEM;
     open CUSTFILE;
     open ITEMFILE;
     open CTRLFILE;

     Initialized=*ON;
     return;
  on-error;
     close *all;
     Initialized=*OFF;
  endmon;

 /end-free
P                 E
```

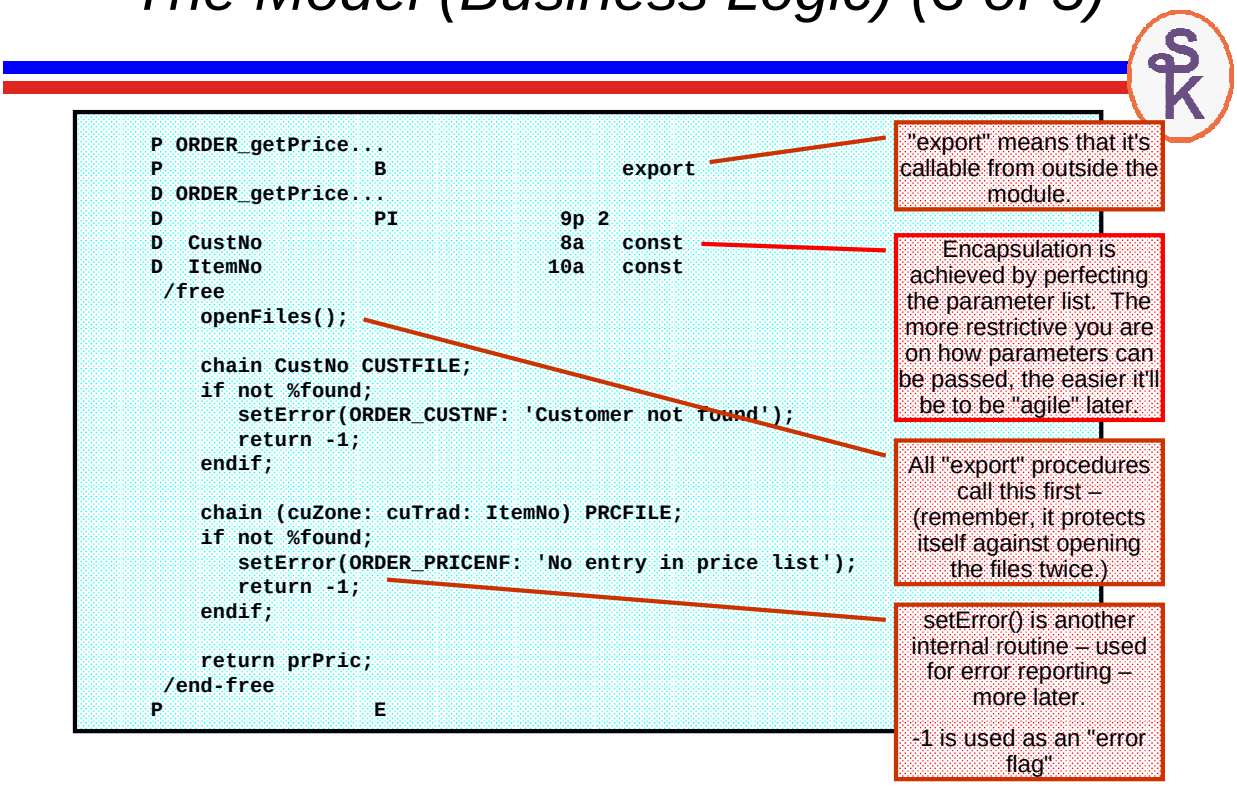
Every service program has an "openFiles" routine. It opens all of the USROPN files.

The "Initialized" variable prevents this code from being run more than once.

If something goes wrong, close everything so the user can start again from scratch.

*The **openFiles** routine is a great place to insert OVRDBFs, or any other code that should be done before the files are opened. REMEMBER: the caller should know nothing about how your routines work – including which files they use – and therefore they should not have to issue overrides!!*

The Model (Business Logic) (3 of 5)



```
P ORDER_getPrice...
P                 B                   export
D ORDER_getPrice...
D                 PI             9p 2
D  CustNo                         8a   const
D  ItemNo                        10a   const
 /free
    openFiles();

    chain CustNo CUSTFILE;
    if not %found;
       setError(ORDER_CUSTNF: 'Customer not found');
       return -1;
    endif;

    chain (cuZone: cuTrad: ItemNo) PRCFILE;
    if not %found;
       setError(ORDER_PRICENF: 'No entry in price list');
       return -1;
    endif;

    return prPric;
 /end-free
P                 E
```

"export" means that it's callable from outside the module.

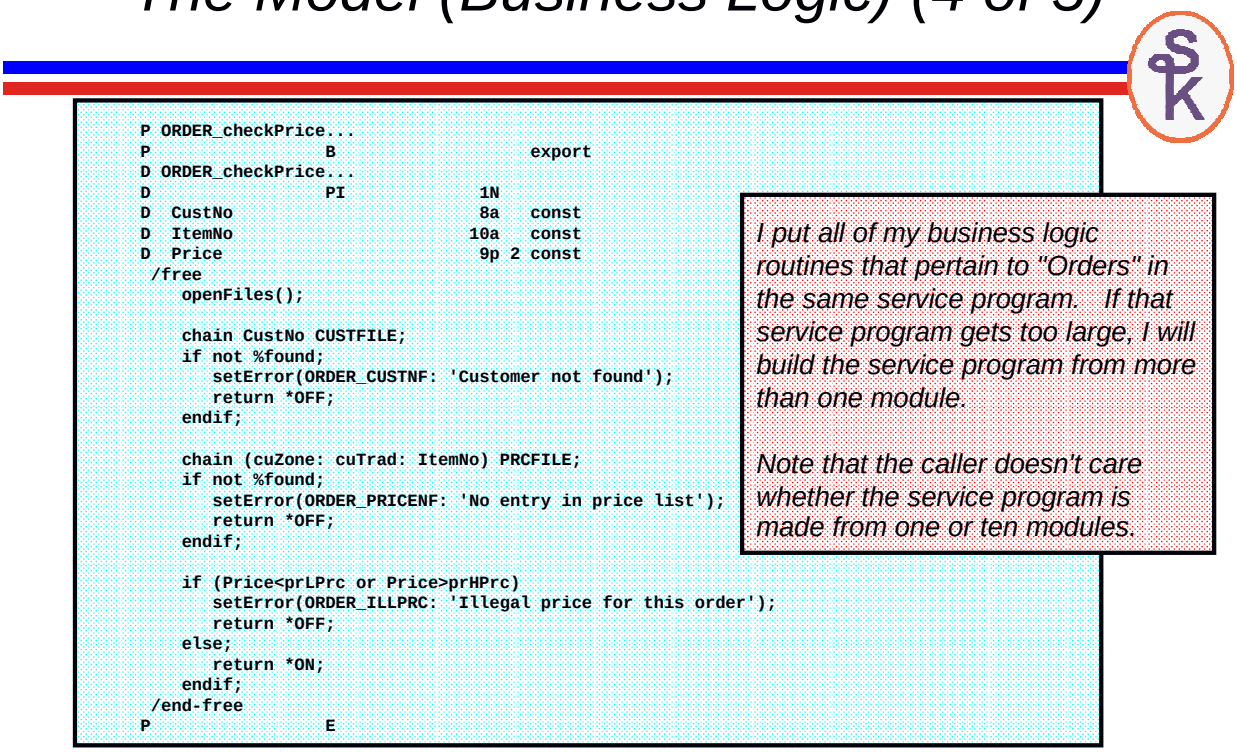
Encapsulation is achieved by perfecting the parameter list. The more restrictive you are on how parameters can be passed, the easier it'll be to be "agile" later.

All "export" procedures call this first – (remember, it protects itself against opening the files twice.)

setError() is another internal routine – used for error reporting – more later.

-1 is used as an "error flag"

The Model (Business Logic) (4 of 5)



```
P ORDER_checkPrice...
P                 B                   export
D ORDER_checkPrice...
D                 PI             1N
D  CustNo                        8a   const
D  ItemNo                       10a   const
D  Price                         9p 2 const
 /free
    openFiles();

    chain CustNo CUSTFILE;
    if not %found;
       setError(ORDER_CUSTNF: 'Customer not found');
       return *OFF;
    endif;

    chain (cuZone: cuTrad: ItemNo) PRCFILE;
    if not %found;
       setError(ORDER_PRICENF: 'No entry in price list');
       return *OFF;
    endif;

    if (Price<prLPrc or Price>prHPrc)
       setError(ORDER_ILLPRC: 'Illegal price for this order');
       return *OFF;
    else;
       return *ON;
    endif;
 /end-free
P                 E
```

I put all of my business logic routines that pertain to "Orders" in the same service program. If that service program gets too large, I will build the service program from more than one module.

Note that the caller doesn't care whether the service program is made from one or ten modules.

The Model (Business Logic) (5 of 5)

```
P setError        B
D setError        PI
D   ErrId                       10i 0 const
D   ErrMsg                      80a   varying const
 /free
    lastErrNum = ErrId;
    lastErrMsg = ErrMsg;
 /end-free
P                 E



P ORDER_error     B                   export
D ORDER_error     PI            80A   varying
D   ErrorId                     10i 0 options(*nopass:*omit)
 /free
    if %parms>=1 and %addr(ErrorId)<>*null;
       ErrorId = lastErrNum;
    endif;
    return lastErrMsg;
 /end-free
P                 E
```

Error handling… error message and id are set by setError. To find out what happened, the caller can always call ORDER_error to get the last error number and message.

Note that this is NOT "stateless" – which has drawn some criticism (but see the "stored procedure wrappers", later…)

Model Copy Book (1 of 2)

```
D ORDER_PRICENF...
D                 c                   const(1101)
D ORDER_CUSTNF...
D                 c                   const(1102)
D ORDER_ILLPRC...
D                 c                   const(1103)
D ORDER_ERROR_ITEM_NOT_FOUND...
D                 c                   const(1104)


 ****************************************************
 * Order_checkPrice(): Check the price of an item
 *
 *    CustNo = (input) Customer number
 *    ItemNo = (input) item number
 *     Price = (input) price to validate
 *
 * Returns *ON when price is valid, *OFF otherwise.
 ****************************************************
D ORDER_checkPrice...
D                 PR             1N
D  CustNo                        8a   const
D  ItemNo                       10a   const
D  Price                         9p 2 const

. . .  Other prototypes here . . .
```

Everything in the copy book will be used by callers – so should use the "ORDER" prefix.

Note that names can be up to 4096 long. Use . . . to extend them, if needed.

Model Copy Book (2 of 2)

```
**************************************************
* Template for 1 line of an order
**************************************************
D Order_Item_t    ds                  qualified
D                                     based(Template)
D   LineNo                       3p 0
D   ItemNo                       8a
D   Qty                          5p 0
D   Price                        9p 2
D   Desc                        20a

**************************************************
*  Order_loadItems(): Routine to load the items
*     on an existing order on disk.
*
*   OrderNo = (input)  Order to load from disk
*     Count = (output) number of items on order
*      Item = (output) array of items on order
*
* Returns *ON when successful, *OFF otherwise.
**************************************************
D Order_loadItems...
D                 PR             1N
D  OrderNo                      10a   const
D  Count                         3p 0
D  Item                               likeds(Order_Item_t) dim(999)
```

It's very important that anything referenced by "like" or "likeds" (or other similar tools) be provided in the same copy book – this helps provide proper encapsulation.

In this case, the data structure referenced with LIKEDS is defined in the same copy book (and prefixed into the ORDER name space)

BASED(Template) stops RPG from allocating memory for the variable

Model – Closing Thoughts

- Remember that the goal is to engineer (not just throw together) an interface
- That interface should be re-usable from just about anywhere.
- Think of it as creating your own API (because that's what it really is) for your application
- Or, you might think of it as creating your own programming language, with op-codes specifically for your business rules.
- Some folks like to "externalize" their database – i.e. put the database in a separate module from the business rules. (in that case, you'd have MVCD – Model, View, Controller, Database).
  - Useful if you might want to use a different method of storing data some day.
  - Maybe switch to SQL Server? Oracle? MySQL?
  - Maybe switch to using XML? Stream files?
  - Personally, I don't see that as likely, so I'm happy to put my database logic in with the business logic. (I don't see my database changing any time soon.)

The Controller (Program Flow)

```
D Step            s             10i 0
 /free

    select;
    when   step = 0;
        if ( myVIEW_getOrder(Order:Cust)=*OFF);
           *inlr = *on;
           return;
        else;
            step = step + 1;
        endif;

    when   step = 1;
        if ( ORDER_loadHeader(Order:Cust) = *OFF );
            myView_setError( Order_Error() );
            step = step - 1;
        else;
            step = step + 1;
        endif;

    when   step = 2;
        if ( myVIEW_editHeader() = *OFF );
            myView_setError( Order_Error() );
            step = 0;
        else;
            step = step + 1;
        endif;
```

I typically write my 5250 controller so that it can go forward or backward by adding/subtracting froma "step" variable.

From a View routine, F12 might cause it to go back a step.

From a Model routine, an error might cause it to go back to the last screen.

Controller Notes

- Usually the shortest piece of the application (shorter than the model or view)
- only controls the flow of the application.
- calls the model and view routines and passes the data between them.
- Called the "glue" that holds the program together.
- Theoretically you can replace only the view module and you'll have the same application with a different user interface (such as converting green screen front-end to a Windows GUI front-end)
- However, it's been my experience that any time the view changes, the controller has to change as well, since the flow of the program is so closely tied to the user's input.
- Consequently, I frequently "cheat" and put the controller and view together into a single object. I'll write a *PGM object for the controller and view, and it'll call a *SRVPGM application for the model.

The View (User Interface)

Notes about a view module for 5250 (green screen) displays.

There's more than just DDS to the view – there's RPG code, too!

All of the logic that relates to the user interface, broken into procedures.

Includes

- DDS for display file
- Code that runs EXFMT, etc.
- Handling of function keys, page up, page down, etc.
- Loading, reading, etc of subfiles.
- Separate procedure to display each screen. (as needed)
- Separate procedures to clear/load subfiles (as needed)
- Separate procedures to print to print files (as needed)

Sample View Procedure

```
FACMEORDS  CF   E             WORKSTN INDDS(DspFunc)

D DspFunc         ds                  qualified
D   Exit                        1n    overlay(DspFunc:03)
D   Cancel                      1n    overlay(DspFunc:12)
  .
  .
  .
P VIEW_getOrder    B
D VIEW_getOrder    PI             1N
D   Order                        10a
D   Cust                          8a
 /free

  exfmt ORDENTS1;
  scErrMsg = *Blanks;

  if (dspFunc.Exit);
     return *OFF;
  endif;

  Order = scOrder;
  Cust  = scCust;
  return *ON;

 /end-free
P                 E
```

```
01/01/08                  Order Entry


            Order:   61435
          Customer:  12345


F3=Exit
```

Routines in the view typically take parameters that get moved to/from a screen or print file – handling one screen at a time.

Then they display the screen and handle keys (all stuff you already know how to do.)

In this case, I return *OFF when the user presses the exit key (F3) or *ON otherwise.

Combined View & Controller

I have discovered that any time the view changes, the controller usually changes as well. Consequently, I often put the view and controller in the same module.

- One less source member to maintain.
- I can call my "model" routines directly from the view procedures, which is a little more like the "old pattern".

In that scenario:

- Main procedure ("mainline") of my program has the controller logic
- Separate subprocedure for each screen.
- The screen subprocedures will call routines from the model directly to do validations, retrieve stuff (like the getPrice example), descriptions that go with item numbers, etc.

Sample Combined View/Controller

In this example, I'm calling the routines in the model directly from the view. (Some would say you shouldn't do that…)

```
01/01/08                    Add Item to Order

                Order:   61435

             Customer:   12345

                 Item:   54321___

                  Qty:   __________

                Price:   ______________

F12=Cancel
```

```
/free
  dou scMsg = *blanks

    exfmt ADDITEMS2;
    scMsg = *blanks;

    // handle F12 here…

    if (scPrice = 0);
        scPrice = ORDER_getPrice( scCust: scItem );
        if (scPrice = -1);
            scMsg = ORDER_error();
        endif;
    else;
        if (ORDER_checkPrice( scCust: scItem: scPrice) = *OFF);
            scMsg = ORDER_error();
        endif;
    endif;

  enddo;
```

Now all programs can call these same routines – only one place to change them.

The underlying routines can change any way I want them to, as long as the parameters stay the same, none of the callers have to change.

Get the Idea?

I hope you get the idea for the 5250 view. I didn't want to spend too much time on it, since I figure you already understand the idea, since most of the code is stuff you've done before.

If you're still unsure, please download the complete sample applications from the articles I've written, and walk through the code… you should find it easy enough to follow.

It's important to understand that the "view" isn't purely for 5250 screens. Any interface that gets the needed input/output can be considered a "view". Consider these ideas:

- A web interface – providing a much more modern user interface.
- A batch program – instead of an interactive user, read the input from a file – like a "script" (similar to what you do when you do batch FTP?)
- Or perhaps data from an EDI document
- Printer output is also "user interface".
- SQL stored procedures or Web Services can provide a "view" that's running on a program on a completely different computer.

CGIDEV2 as a Controller/View

A native ILE web tool such as raw CGI, CGIDEV2, eRPG SDK, CGILIB, etc, can be used to run our business logic with no changes. Since this is an ILE language, it can call the subprocedures directly.

Considerations:

- Web applications are always stateless.
- Stateless calls make the controller work completely differently.
- Use an "action" variable passed from the browser to keep track of which step of your application needs to be performed next.
- As long as your model is stateless – or stateless with the request scope (such as my error handling) – it will work nicely as a web application

Unfortunately, a sample web application is too much to fit into this presentation. But, see Paul Tuohy's article "Pattern Recognition: Adopting the Pattern" (System iNEWS magazine) for a detailed description and complete sample code.
(There's a link at the end of this presentation.)

More about Encapsulation

Encapsulation is a very important concept for writing re-usable code, and is perhaps ***the*** single most important concept in reducing maintenance and improving agility.

Encapsulation is almost entirely based on writing a strict, well-defined interface (prototype and PI) and making sure that it's the only communication between your model and it's caller. (the controller or combined controller/view)

```
P ORDER_checkPrice...
P                 B                   EXPORT
D ORDER_checkPrice...
D                 PI             1N
D  CustNo                        8a   const
D  ItemNo                       10a   const
D  Price                         9p 2 const
```

Only use EXPORT when a routine MUST be called from outside your service program.

- Greatly reduces the analisys required when you want to change the interface.

Always use CONST or VALUE if a parameter is input-only.

- More self-documenting ("this parameter is clearly input-only")
- Greatly reduces analysis of callers if something changes
- Greatly reduces the code you have to review if a bug is found in production.
- Makes the code easier to re-use!!

Binder Langauge Considerations

- Binder language exports from the SERVICE PROGRAM, whereas the EXPORT keyword exports from the MODULE.
- MODULE exports can be shared by routines that are bound-by-copy (direct binding), but if they're not in the binder language, they can't be called from external programs.
- This provides better encapsulation when you have multiple modules – you can limit procedure calls to be only within that service program!
- ILE binds procedures by ***NUMBER – always add new procedures to the END.***
- To keep service program maintenance to a minimum – make changes backward-compatible.
  - Always add new procedures to the end.
  - Signatures don't protect against parameter issues (unless you MAKE them)
  - Only change parameters in a compatible way.
  - Provide wrappers or "compatibility" procedures when necessary.

Utilize Binder Language

```
 STRPGMEXP SIGNATURE('ORDERR4 ver 1.00')
    EXPORT SYMBOL(ORDER_new)                  #1
    EXPORT SYMBOL(ORDER_loadHeader)           #2
    EXPORT SYMBOL(ORDER_loadItems)            #3
    EXPORT SYMBOL(ORDER_saveHeader)           #4
    EXPORT SYMBOL(ORDER_saveItem)             #5
    EXPORT SYMBOL(ORDER_checkItem)            #6
    EXPORT SYMBOL(ORDER_checkPrice)           #7
    EXPORT SYMBOL(ORDER_checkQuantity)        #8
    EXPORT SYMBOL(ORDER_error)                #9
ENDPGMEXP
```

- ***Always*** use a hard-coded signature.
  - ***EXPORT(*ALL)*** requires re-binding with every change to the export list, which can discourage programmers from using small, re-usable routines.
  - Using *CURRENT and *PRV (with SIGNATURE(*GEN)) does not provide any additional protection, but makes maintenance more cumbersome.
  - You can still force a signature mismatch by changing the version number, in the (very unusual) situations where you need to do that!
- Think about the things you'll do to exports and how easy this makes them:

  Add (very common), Change (very common), rename (unusual), remove (unusual)

Binding Directory Considerations

(do not confuse BNDDIR with binder language)

Using binding directories correctly greatly simplifies maintenance

- If you write software for in-house use, use a single BNDDIR for all *SRVPGMs
- If you work for a software house, use a single BNDDIR for each product you sell.

For example, where I work (an in-house shop), I have a binding directory called MAIN that's always in my library list.

- We add every service program to the binding directory (Model or otherwise)
- Since we use "name spaces" (prefixes) for all exported routines, they never collide.
- The system sorts out which service program to use, and where to find it.
- All I need to code is the /COPY to get the prototypes.

```
     H BNDDIR('MAIN': 'QC2LE')
     H OPTION(*SRCSTMT: *NODEBUGIO)
      /if defined(*CRTBNDRPG)
     H DFTACTGRP(*NO) ACTGRP('KLEMENT')
      /endif

      /copy ORDER_H
      /copy CUST_H
      /copy DATEUTIL_H
```

Compiling / Binding

Because the same binding directory is always used, and it's specified in the H-spec, the commands to compile and bind are very easy.

Other considerations:

- Don't use the binding directory for modules – that only seems to confuse things.
  (If modules are named according to my spec, it's easy enough to bind them.)
  (Any module called from multiple locations should be put in a SRVPGM!)
- Activation group
  - Service programs should always use *CALLER
  - Programs should use *NEW if called directly from a menu, and you want all files to close when the user returns to the menu.
  - Programs should otherwise use ACTGRP('MYNAME') where MYNAME is a standard name you've decided on for your company.
  - My company (Klement Sausage Co) uses ACTGRP('KLEMENT')
  - Don't use QILE, too many other people use that, and you might conflict.
  - Don't use *CALLER for a program – causes problems with RCLRSC
  - Don't use (or rarely use) *NEW for web apps, hurts performance too much.
- PDM options 14 and 15 work nicely – as does WDSC/RDi compile commands, but you have to do the CRTSRVPGM at the command line.

Sample Compile/Bind Commands

Compile the model (a single-module service program – most common for models):

```
CRTRPGMOD MODULE(ORDERR4) DBGVIEW(*LIST)
CRTSRVPGM SRVPGM(ORDERR4)
ADDBNDDIRE BNDDIR(MAIN) OBJ((ORDERR4 *SRVPGM))
```

If it uses SQL (including result sets for stored procedure wrappers):

```
CRTSQLRPGI OBJ(ORDERR4) OBJTYPE(*MODULE) DBGVIEW(*SOURCE)
CRTSRVPGM SRVPGM(ORDERR4)
ADDBNDDIRE BNDDIR(MAIN) OBJ((ORDERR4 *SRVPGM))
```

When it uses multiple modules (substitute CRTSQLRPGI if it uses SQL)

```
CRTRPGMOD MODULE(ORDERR4) DBGVIEW(*LIST)
CRTRPGMOD MODULE(ORDER2R4) DBGVIEW(*LIST)
CRTRPGMOD MODULE(ORDER3R4) DBGVIEW(*LIST)
CRTSRVPGM SRVPGM(ORDERR4) MODULE(ORDER*)
ADDBNDDIRE BNDDIR(MAIN) OBJ((ORDERR4 *SRVPGM))
```

Compile a combined controller/view that calls the model:

```
CRTBNDRPG PGM(ORDVIEWR4) DBGVIEW(*LIST)
```

Scott's BUILD Tool

I've written a tool that lets you put the various steps required to compile a program in comments at the top of your program. The tool is designed to work nicely from all of the environments:

- Command-line
- PDM
- WDSC

It automates the entire process, including the CRTSRVPGM and adding to the binding directory.

See the "More Information" links for a link to the article where you can learn more and download the code.

(Requires System iNetwork Pro membership – but no additional costs…)

Reusing the Model from Non-ILE Apps

As ILE objects, service programs can only be called from other ILE code, right? Wrong. Here are a few ways that you can call a service program from a non-ILE language:

- Web Services
- External Stored Procedures (SQL)

Stored procedures are callable from just about anywhere:

- .NET, ASP, Java, PHP, NET.DATA, Visual Basic, C/C++, even Microsoft Office!
- Can be on the same machine, or different machine (via ODBC or JDBC)

I always write a separate ILE sub procedure to be called from the stored procedure – never call the "regular" ILE procedure directly. This stored procedure interface will call the "regular" routine, but will do some massaging of the data. (I call this a "wrapper")

Why use a wrapper?

- Result sets for output parms (Meta data for returned variables)
- Enables ILE to call directly
- Enables a façade over the error handling

Sample Stored Procedure Wrapper

```
P ORDER_checkPrice_sp...
P                 B                   export
D ORDER_checkPrice_sp...
D                 PI
D  CustNo                        8a   const
D  ItemNo                       10a   const
D  Price                         9p 2 const

D Result7         ds                  qualified occurs(1)
D  MsgId                        10i 0 inz
D  Msg                          80a   varying inz
 /free
      %occur(Result7) = 1;
      if ( Order_CheckPrice(CustNo: ItemNo: Price) = *OFF);
         Result7.Msg = ORDER_error(Result7.MsgID);
      endif;

 /end-free
C/exec SQL set Result sets Array :Result7 for 1 Rows
C/end-exec
P                 E
```

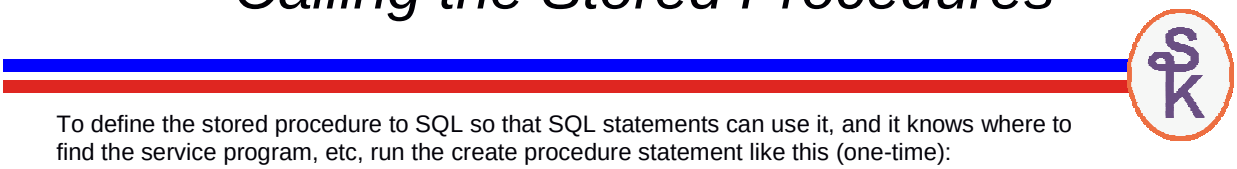
Input data comes from SQL as parameters.

The information in this DS (field names, sizes, data types) will be communicated as "meta data" in the result set. Callers can use that information

Calls the "regular" routine and the error message routine – so there's no duplication of code – still just one place to change rules.

Returns the output data to SQL as a result set.

Calling the Stored Procedures



To define the stored procedure to SQL so that SQL statements can use it, and it knows where to find the service program, etc, run the create procedure statement like this (one-time):

```
CREATE PROCEDURE ORDER_CHECKPRICE(
         IN CustNo    CHAR(10),
         IN ItemNo    CHAR(8),
         IN Price     DECIMAL(9,2)
       )
       LANGUAGE RPGLE
       NOT DETERMINISTIC
       CONTAINS SQL
       EXTERNAL NAME 'SCOTTLIB/ORDERR4(ORDER_CHECKPRICE_SP)'
       PARAMETER STYLE GENERAL;
```

Now the procedure can be called from an SQL statement like this one:

```
CALL ORDER_CHECKPRICE('12345', '54321', 19.27);
```

A result set might look like this:

MSGID – INTEGER	***MSG – VARCHAR(80)***
1101	'Customer not found'

More Information

In System iNetwork Programming Tips newsletter:

Feb 14, 2008 issue, "Writing Reusable Service Programs" (Associate Membership)
http://systeminetwork.com/article/writing-reusable-service-programs

Jan 24, 2008 issue, "A General Purpose BUILD Tool" (Pro Membership)
http://systeminetwork.com/article/general-purpose-build-tool

In System iNEWS magazine (ProVIP membership, or print copy):

Feb 2007 issue, "Pattern Recognition Eases Modern RPG Programming"
http://systeminetwork.com/article/pattern-recognition-ease-modern-rpg-programming

Paul Tuohy, October 2007 issue: "Pattern Recognition: Adopting the Pattern"
http://systeminetwork.com/article/pattern-recognition-adopting-pattern

Paul Tuohy, Feb 2007 issue, "Considerations for a Successful ILE Implementation"
http://systeminetwork.com/article/considerations-successful-ile-implementation

This Presentation

You can download a PDF copy of this presentation from:

http://www.scottklement.com/presentations/

Thank you!