3

Procedures: How, When, and Why to Build Them

The previous chapters explored some theory. Now, it's time for some hands-on work. Let's leverage your OPM RPG knowledge with a simple and practical example of using procedures.

Here are just a few quick notes first, to avoid confusion:

- What I refer to here as a "procedure" is also known as a subprocedure, because for some, the term "procedure" applies only to the main program flow procedure, similar to an OPM program. I'll use the term "main program flow procedure" or "main program flow module" to refer to the latter concept.
- I'm presenting RPG code in fixed format until chapter 8, in which I'll help you, dear reader, make the transition (as smoothly as possible) to free format. Let's take one step at a time to keep things as simple as possible. After all, as I mentioned in the introduction, this is not a quick reference guide, but a "slow" one!

- You might notice that I use parameters in the procedures, but don't present or discuss the (many) keywords related to them. All of that will be explained in a later chapter as well.

If you are an experienced RPG programmer, you might argue that the OPM scenario presented in chapter 1 and shown in Figure 3.1 is too old-fashioned and simplistic.

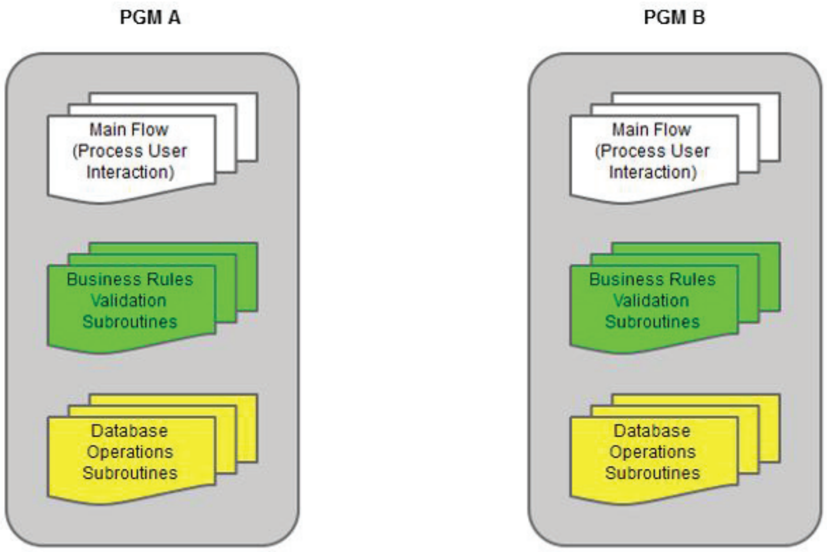
PGM A
Main Flow (Process User Interaction)
Business Rules Validation Subroutines
Database Operations Subroutines
PGM B
Main Flow (Process User Interaction)
Business Rules Validation Subroutines
Database Operations Subroutines

Figure 3.1: The traditional OPM scenario

You are right, of course. Even in OPM, it's possible to avoid duplicating the same code over and over again in multiple programs. To be fair, a more accurate scenario would be the one depicted in Figure 3.2, in which the code related to business rules is isolated in several standalone programs (`BR1`, `BR2`, ... `BR`*x*). These programs are called as needed by `PGM A` and `PGM B`.

How to Build Your First Procedure, Using Your OPM Knowledge

I'll use the more realistic and up-to-date scenario in Figure 3.2 as my starting point for the creation of a simple procedure. I'll stick to the scenario described in the first chapter: `PGM A` and `PGM B` both handle inventory, but they do slightly different things. While A imports inventory items from a cargo manifest CSV file, B handles the user inventory management via screen interaction. They both use some of the same business rules, which I've isolated into programs BR*x* in this new scenario.

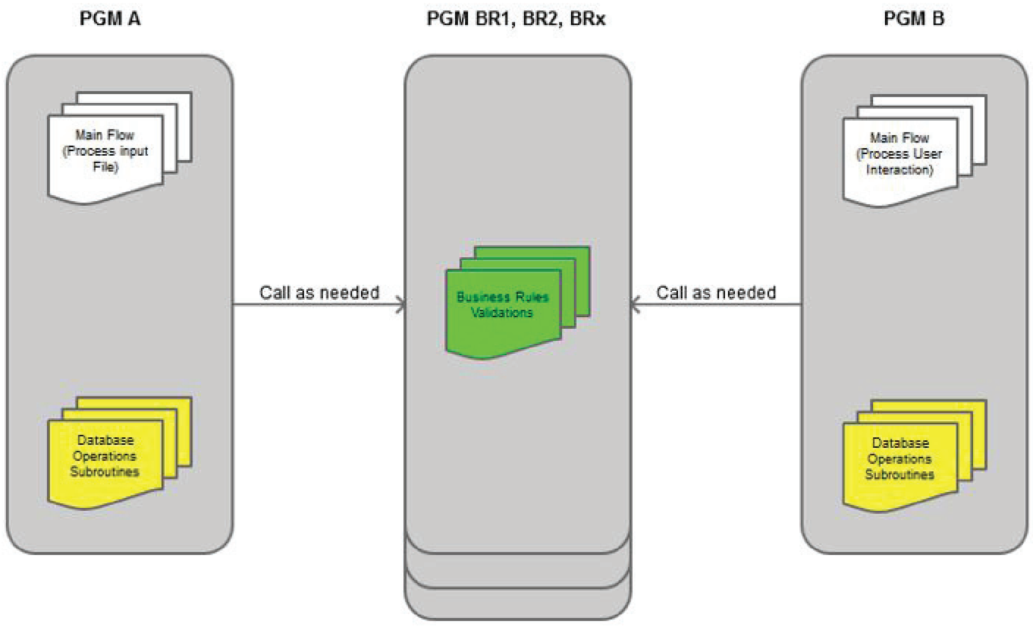
PGM A
PGM BR1, BR2, BRx
PGM B
Main Flow (Process input File)
Database Operations Subroutines
Call as needed
Business Rules Validations
Call as needed
Main Flow (Process User Interaction)
Database Operations Subroutines

Figure 3.2: An evolved, more current, and more realistic OPM scenario

Let's say that BR1, one of those programs, translates the supplier's item ID into the company's item ID. PGM A would call BR1 whenever it needs to import a new item, passing the external item ID and supplier ID codes, and receiving the internal item ID. This means that a PLIST would be defined in PGM A to call BR1 with the necessary variables:

```
C     PL_BR1        PList
C                   Parm                     P_ExtItmID
C                   Parm                     P_SupID
C                   Parm                     P_IntItmID
```

The structure should be familiar, so let me just explain the parameter names:

- P_ExtItmId is the external (supplier) item ID.

- P_SupID is the ID of the supplier (just in case different suppliers use the same item ID for different things).
- P_IntItmId is the internal item ID that is returned by the program.

Now let's transform BR1 into a procedure, step by step.

To use a procedure, I need to "tell" the program how to call it, the same way I would do for a program. For that, I'll use something called *prototype definition*. This definition must be used in every program or service program that uses the procedure. Since the idea here is to reuse code instead of duplicating it, I usually create a source member in a separate source file, named QCPYLESRC, with all the module's procedure prototypes. I then include it in the program or service program where I need to use it, via a /COPY or /INCLUDE instruction. In fact, as you can see below, the prototype definition (shown as part of the QCPYLESRC/BR_INV_PR source member) is actually quite similar to PLIST:

```
 * ----------------------------------------------------------------------*
 *   Prototype . : BR_INV_PR                                              *
 *   Description : Inventory Related Procedures                           *
 *   Author .... : Rafael Victoria-Pereira                                *
 *   Date ...... : March 2014                                             *
 *   Changes ... :                                                        *
 * ----------------------------------------------------------------------*

 * ----------------------------------------------------------------------*
 *   Convert the External Item ID into the Internal Item ID
 * ----------------------------------------------------------------------*
D CvtItmId        PR
D P_ExtItmID                    50
D P_SupId                      256
D P_IntItmID                    50
```

Then, all I need to do is include this definition in my program/service program, using either /COPY or /INCLUDE:

```
* Prototype definition for Inventory-Related Procedures
 /COPY QCPYLESRC,BR_INV_PR
```

Later in this chapter, I discuss which to choose, depending on the situation.

Just a note about the procedure name: instead of BR1, I'm using a (slightly) clearer name for the procedure: CvtItmID. I like to use a verb to identity the procedure type ("Cvt" being short for "convert" in this case), followed by the subject ("Item ID"). You could argue that Convert_Item_ID would be even clearer (and you'd be right!), but it's important to avoid getting carried away with long names. Besides, Convert_Item_ID wouldn't "fit" in the space reserved for the function name in the D-line (although that's not really a problem). In the next chapter, you'll learn how to solve this "problem." For now, just keep in mind that you need to find the right balance between readability and maintainability. Excessively long names cost you additional time while coding. They also increase the chance of misspelling when you need to call the procedure (especially if you still use Source Entry Utility—SEU).

Speaking of calling, the way to call a procedure also differs from calling a program:

```
C                   CallP     CvtItmID(P_ExtItmID : P_SupID : P_IntItmID)
```

CALLP is used instead of CALL, and the parameters follow the procedure name enclosed by parentheses and separated by a colon, just as in a built-in function. You can also use a slightly different notation, similar to a program call with parameters:

```
C                   CallP     CvtItmID(P_ExtItmID :
C                                      P_SupID    :
C                                      P_IntItmID )
```

OK, that's how the procedure is defined and called. Now let's see how to create it! Here's an example of a simple procedure structure, without the actual code:

```
 *----------------------------------------------------------------------*
 * Convert the External Item ID into the Internal Item ID
 *----------------------------------------------------------------------*
P CvtItmID        B                   EXPORT
D CvtItmID        PI
D P_ExtItmId                    50
D P_SupId                       10  0
D P_IntItmId                    50
```

Continued

```
C*
C* The procedure's code goes here
C*
P CvtItmId        E
```

As I explained in chapter 1, procedures are part of modules. A module can contain one or more procedures. This means that the compiler needs to know where each procedure begins and ends. The P-lines you see above delimit the CvtItmId procedure. Note that the first P-line, which contains the procedure name (positions 7 to 21), also has the keyword EXPORT. This means that this procedure will be exported by the module with the CvtItmId. (Since RPG is case insensitive, you can and should use mixed case or some other form of writing to make names more easily understandable, but for RPG, it really doesn't matter.)

The EXPORT keyword means that this procedure will be available to the programs and/or service programs that are bound to this module. This line also has a "B," indicating the beginning of the procedure. Similarly, the other P-line, which ends the procedure, has an "E" in the same position, thus telling the compiler that the code of the procedure is contained between the two P-lines.

Everything in between the P-lines is somewhat similar to a standard OPM program. I say "somewhat similar" because right after the beginning of the procedure comes the procedure interface (PI). The D-lines shown here are used to define the procedure's parameters, just as an *ENTRY PLIST would do in an OPM program—and just as in an OPM program, you don't need to specify a procedure interface if your procedure doesn't have parameters. However, you should always include a PI line for consistency between procedures.

Note that this list begins with a D-line that repeats the procedure name and has "PI" in positions 24–25. Having the procedure name in the PI line is optional, but I recommend that you always specify the procedure name with your procedure interface because doing so makes it easier to create the prototype definition for the copy member. You simply copy the "PI" block of lines to the respective member in QCPYLESRC, replace the "PI" with "PR," and you're done! This line marks the start of the procedure interface. Also, note that the D-lines don't have the usual "S," "C," or "DS" in positions 24–25 for the definition type because they are part of the prototype interface. It's also possible to define variables within a procedure (more on this later); these variables would be defined using the usual notation in positions 24–25 ("S," "C," or "DS").

Finally, there would be a bunch of C-lines containing the actual procedure code, just as in a regular OPM program. One of these lines would assign a value to P_IntItmID, thus allowing the procedure to return the internal item ID to the calling program, just as it would in an OPM program.

When and Why You Should Build a Procedure

The next big question is when to create a procedure. How should an OPM program's subroutines be transformed into procedures? Should the procedures be exact copies of the subroutines? Which parameters should be passed between these procedures?

To answer these questions and others you might have, let's try to deconstruct the concept of a procedure. It's supposed to be reusable and work together with other procedures, like building blocks, to create modular solutions to the challenges today's RPG programmers face. This implies that a procedure should be small and simple, so that it can be used in as many situations as possible. It should also be compatible with other procedures. In other words, each procedure should be independent and, within reason, self-sufficient.

Our example so far has been the inventory programs scenario depicted in Figure 3.2, earlier in this chapter. It shows two main programs, PGM A and PGM B, which manage the inventory master file in different ways. While A imports inventory items from a cargo manifest CSV file, B handles the user inventory management via screen interaction. They both call some external programs, identified here as BR1, BR2 ... BR*x*.

These "BR" programs are the perfect candidates for procedures, but you need to be careful. If the idea is to build something that you can reuse, you need to keep it as simple as possible. Let's say the BR2 program serves the purpose of checking whether an item exists in inventory and is updating its quantity as specified.

Figure 3.3 depicts a scenario in which BR2 is transformed into procedures. Imagine that program BR2 has a subroutine, among others, that checks whether an item already exists in stock, and that it also updates the inventory. When BR2 was turned into a module, it was split into several procedures. As I explained at the beginning of this chapter, a module can have one or more procedures. BR2's subroutine "Check if item exists in inventory and update it" becomes procedures "Check if item exists in inventory" and "Update inventory" because there will be some situations in which only one of these two operations will need to be performed. Somewhere in PGM A or PGM B you might have a piece of code that checks whether an item exists in inventory, but doesn't update it, or vice versa.

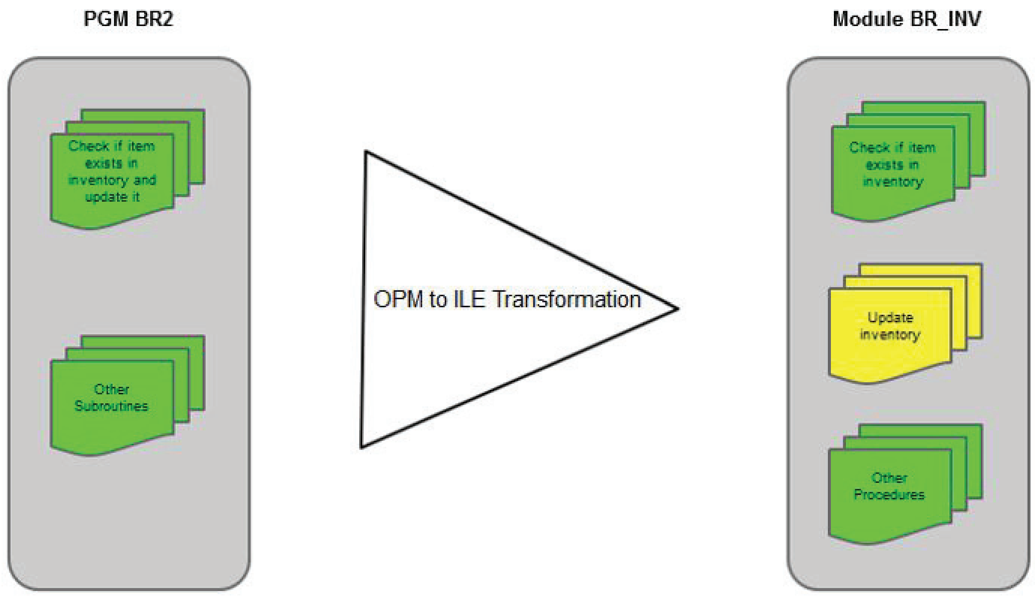
PGM BR2
Check if item exists in inventory and update it
Other Subroutines
OPM to ILE Transformation
Module BR_INV
Check if item exists in inventory
Update inventory
Other Procedures

Figure 3.3: Transformation of the BR2 OPM program into a module with several procedures

Procedures should also be compatible with one another. If you have several procedures that might be used together or are related to the same thing (like inventory business rules, in this example), they should share some common parameters or some sort of key to access the data on which they operate. This needs special care when you transform subroutines into procedures, because while the OPM program had global variables that were available to all the subroutines, the same won't happen, at least not automatically, with the procedures. Whatever variables you define *inside* your procedure won't be available *outside* that procedure.

Let's say that PGM B needs to remove an item from inventory. It will first call the "Check if item exists in inventory" procedure spawned from BR2, and then it will call the "Update inventory" procedure. These two procedures might have existed as a subroutine in BR2 and shared a global program variable that stored the item ID, but when they were transformed into procedures, they became independent from one another. However, they need to share the item ID as a key, so that they can work together. The most explicit way of sharing the data is passing it as a parameter, common to both procedures. This has other advantages, like making debugging easier, for instance.

Finally, procedures should be, within reason, self-sufficient. In other words, a procedure should be able to perform whatever operation it needs to perform while keeping calls to other procedures at a minimum. Of course, there will be situations in which this is simply not possible, but you should try to minimize those situations. How? Well, having procedures that perform a simple and well-delimited operation, like the two examples presented here, is a way.

Let the "main" program flow module (remember M1 and M2 from the previous example?) take care of the orchestration of the procedures to produce the desired result. This is important because when you compile your code, you might run into a "chicken or egg" situation: you need to recompile module X, but this module has a procedure that uses another procedure or function from module Z. If module Z also uses something from module X, you won't be able to recompile either of them! Also keep in mind the aforementioned need for simplicity and reusability; if your procedures are small and self-contained, they will be easier to understand, debug, and use over and over again, instead of writing new (or duplicated) code.

A final note about this example: you might have noticed that the "Update inventory" procedure (the one in the middle) in Figure 3.3 is a lighter shade than the other procedures (in the electronic version of this book, the "Update inventory" procedure is yellow). In fact, it's the same color as the "Database Operations" subroutines in Figure 3.2. This is not a coincidence. Procedure "Update inventory" shouldn't be in module BR_INV because it's a database operation procedure, and BR_INV is a business rules module. Updating the inventory is definitely not a business rule. It depends on one or more business rules, but it's not a business rule, so it shouldn't be there. For the moment, let's keep it simple. I'll come back to this detail later in the book, and explain where the procedure belongs and why.

A Side Note: /COPY or /INCLUDE?

You're probably wondering why I keep writing "/COPY or /INCLUDE" every time I mention the prototype definition copy member. Both directives have the same syntax and purpose.

As you probably gathered from the previously shown sample code regarding the inclusion of the prototype definition, the purpose is to include a piece of code from another source member in the one that contains the /COPY or /INCLUDE. The syntax is quite simple: /COPY (or /INCLUDE) followed by a blank space, the library and name of the source file, and a comma followed by the name of the source member. (You can skip the library, and the compiler will look for the filename in the library list.) Here is an example:

```
/COPY QCPYLESRC,M3_PR
```

It's also possible to have a /COPY or /INCLUDE inside another /COPY or /INCLUDE; this process is called *nesting*. You can have up to 32 levels of nesting by default, with a maximum of 2,048, but it's possible to change this value in the COPYNEST keyword of the H-specs. Just be careful not to repeat a /COPY that you used in an upper level in a lower level, thus causing an infinite loop.

Until you start using embedded SQL in your RPG code, you can use either directive. If you want to use /INCLUDE when you compile a module with embedded SQL, you need to specify RPGPPOPT(*LVL2) in the Create SQL ILE RPG Object (CRTSQLRPGI) command. I'll discuss this compilation command later, in the context of embedded SQL in RPG (chapter 11). *LVL2 is not the default value for the keyword, so you have to either change it at compilation time, or simply use /COPY instead of /INCLUDE and leave the default as it is.

Let's go over a simple example: PGM A, a "regular" RPG program, uses some procedures from BR_INV. To do this, the following line is included in the program's code:

```
/COPY QCPYLESRC,BR_INV_PR
```

However, BR_INV_PR, BR_INV's copy member, contains the prototype definitions for BR_INV and some additional definitions (constants and work variables) that facilitate the use of the procedures. These additional definitions are also used in other modules, so they don't actually exist in the BR_INV_PR copy member; they're included via another /COPY instruction. This is an example of a nested copy—a copy member inside another copy member. Because PGM A is not an SQL embedded RPG program, this nesting doesn't cause any problems.