HEWLETT-PACKARD GENERAL SYSTEMS USERS GROUP

FEBRUARY, 1980 MEETING

DATABASE SERIES

HIDING DATA STRUCTURES IN PROGRAM MODULES

Brian C. Mullen

I.S.P. Information Systems Planning Corp.
4180 Lions Avenue
North Vancouver, BC
V7R 3S4
(604) 980-9101

How can you convert a data structure used in almost every program in your application system from a KSAM file to an IMAGE database without recompiling any application programs that access this data structure?

In our case, we had a common code table structure that was initially implemented as a KSAM file. These code tables were accessed by almost every program in the system. In order to avoid the overhead and other problems associated with KSAM, it was decided to convert to an IMAGE data structure.

How was this flexibility achieved?

The details of the physical implementation of the Common Codefile were hidden within the subprograms that accessed it.

What did the CODEFILE contain?

All the codetables such as PROVINCE/STATE codes, MONTHS, job-codes, account-codes which are relatively stable, small and numerous.

Usually they are hardcoded in data divisions or loaded directly in small files using utilities. Adequate maintenance and documentation procedures are hard to justify and are seldom done.

To access, maintain and report the contents of any of the codetables all we required was:

one maintenance program

one report program

One set of access routines

A new CODETABLE can be added and the only additional code is the application code that uses the new code values.

The logical structure of the common code file is

```
COMMON-CODE-FILE

    CODE-TABLE-HEADER
        TABLE-ID
        TABLE-NAME
        CODE-LENGTH
        CODE-TYPE
        TABLE-STATUS
        NUMBER-OF-VALUES
        VALUE-DESCRIPTION-MAX
        PARENT-TABLE-ID

        CODE-TABLE-VALUE-DETAILS
            CODE-VALUE
            CODE-DESCRIPTION


            PARENT-VALUE
```

Clearly there are advantages to defining a data structure as abstractly as possible. We do this through the set of operations to access this data structure.

This would include:

CREATE a CODEFILE

OPEN the CODEFILE

ADD a new TABLE to the CODE-FILE
using TABLE-ID and Parameters

ADD a new CODE-VALUE to an existing CODE-TABLE
using TABLE-ID, CODE-VALUE, CODE-DESCRIPTION

MODIFY an existing code entry in a table
using TABLE-ID, CODE-VALUE, new parameters
using TABLE-ID

GET a code description
using the TABLE-ID and CODE-VALUE

GET the CODE-TABLE parameters using the TABLE-ID

CHECK a CODE-VALUE is exists
using TABLE-ID, CODE-VALUE, RETURN-CODE

DELETE a code entry from an existing table

DELETE a complete CODE-TABLE

In our case, we included some translation capabilities between tables within this CODEFILE but let me keep the example simple.

This definition is similar to some of the operational definitions you will run across in textbooks for data stacks.

Nowhere within this definition has the actual physical implementation been specified.

Initially the CODEFILE was implemented as a KSAM file.

When the overhead due to extra data segments with KSAM was realized we converted the CODEFILE to an IMAGE data structure

This was worthwhile because almost every program in the system accessed the CODEFILE.

Because of the implentation approach we were able to do this without any changes to our existing application programs. However some practical problems have to be overcome.

To access this data structure
some common information must be shared between these routines.

In other words, some of the data must be external
or global with respect to these routines

This shared information usually includes file control information setup by the OPEN operation prior to accessing the CODEFILE.
It depends upon the physical implementation of the CODEFILE

In FORTRAN this would be no problem.
The shared information would be defined in a NAMED COMMOM area.

In COBOL there is no GLOBAL or NAMED COMMOM facility.

If our COBOL programs are not modular, this shared data can be stored in a CONTROL-BLOCK defined in the DATA DIVISION and
passed to each of the subroutines as a parameter in the CALL.

If we attempt to build truly modular COBOL programs with SUBPROGRAMS we encounter the problem of STAMP coupling.
When the data is passed as a parameter in the CALL and
the program structure is multi-level, the shared data must be defined at the highest node in the tree that contains
all the branches using this data.

In addition, the data must be included in every set of CALL parameters between the definition node and the using nodes in the program tree.

Now we really have a design problem!

Before we can build any part of the program we have know the common data and where it is used.
We must define the contents of a GLOBAL-DATA-BLOCK that must be passed with every CALL to all subprograms.

When we are concerned with a single data structure such as a CODEFILE this is not too difficult, but if we build a system with a mutitude of data structures then the problem becomes truly complicated.

We have not successfully uncoupled our modules.
Rather the GLOBAL-DATA-BLOCK is very dependent on our application and the architecture of the system we are implementing.

We cannot predict which of the subprograms may at some time in the future is required to CALL a utility subprogram that may require a piece of this GLOBAL data.

If we do not provided access to this GLOBAL data it is unlikely when the need arises that a programmer will reuse a reusable piece of code.

There are several approaches to overcomming this problem.
Some people are experimenting with them now and I look forward in the future to hearing of there experiences in future talks.

One builds an SPL outer block with the GLOBAL data.
This data is accessed by subprograms via calls to an SPL routine.

An alternative is to combine the routines that use the shared data into a common subprogram.
The individual routines are then invoked as seperate entry points.

This routine must be non-dynamic so that the shared data is preserved between calls.

This requires these routines be loaded into individual program files and cannot be stored in seperate SLs.

Our CODEFILE access routines, were implemented as SPL routines. and our COBOL programs were implemented as PERFORMED SECTIONS and not as SUBPROGRAMS. We had earlier backed away from the subprogram approach when we encountered the problems with global data.

There is a philosophical reason for this talk.

Our success with the CODEFILE has made us more aware of the importance of identifying DATA STRUCTURES in a system and defining them abstractly by the operations we must perform on them.

It provides a quideline for uncoupling physical data structures and application code.

However it also highlites some of the real difficulties in developing modular systems, which in turn are a key to building re-usable code and improving our productivity.

People who work in the FORTRAN world are much more successful at building and using subroutine libraries, than we have been in the COBOL commercial systems world.

> Partly this has been accomplished through neccessity. Few programmers could recreate the highly specialized, complex FORTRAN sub-routines even if they wanted to.
>
> But partly it has been because FORTRAN supports features that allow the creation of highly modular systems. These features are not available in COBOL.

One of the major obstacles to building a modular system in COBOL is the inability to localize Data.
All data is declared globally within the DATA DIVISION.

Because we cannot create local and global data in COBOL, to build modular programs in COBOL we must learn to hide our data structures within COBOL subprograms and implement the access routines as multiple ENTRY points to these subprograms.

Clearly if we want to build libraries of reusable routines we must tackle the problem of standardization of the interfaces so we can plug the reusable routines together to build programs and utimately systems.