A Large Mainframe REXX Application

Design Aspects for a Large-scale Mainframe REXX Application

Anthony Rudd, Datev eG, Nuremberg, Germany

Anthony.Rudd@datev.de

May 6, 2004

Background

The described large-scale REXX mainframe application was developed in-house to provide an interactive workbench to support the tasks performed during the program development life cycle.

Statistics - size

	Number	LOCs
Execs	190	12,000
including kernel	(57)	(5,000)
Panels	120 (50 help)	12,200
General help panels	50	
Skeletons	60	2,200
Messages (individual)	110	

LOCs = lines of code (including comments and blank lines)

Statistics – performance comparison

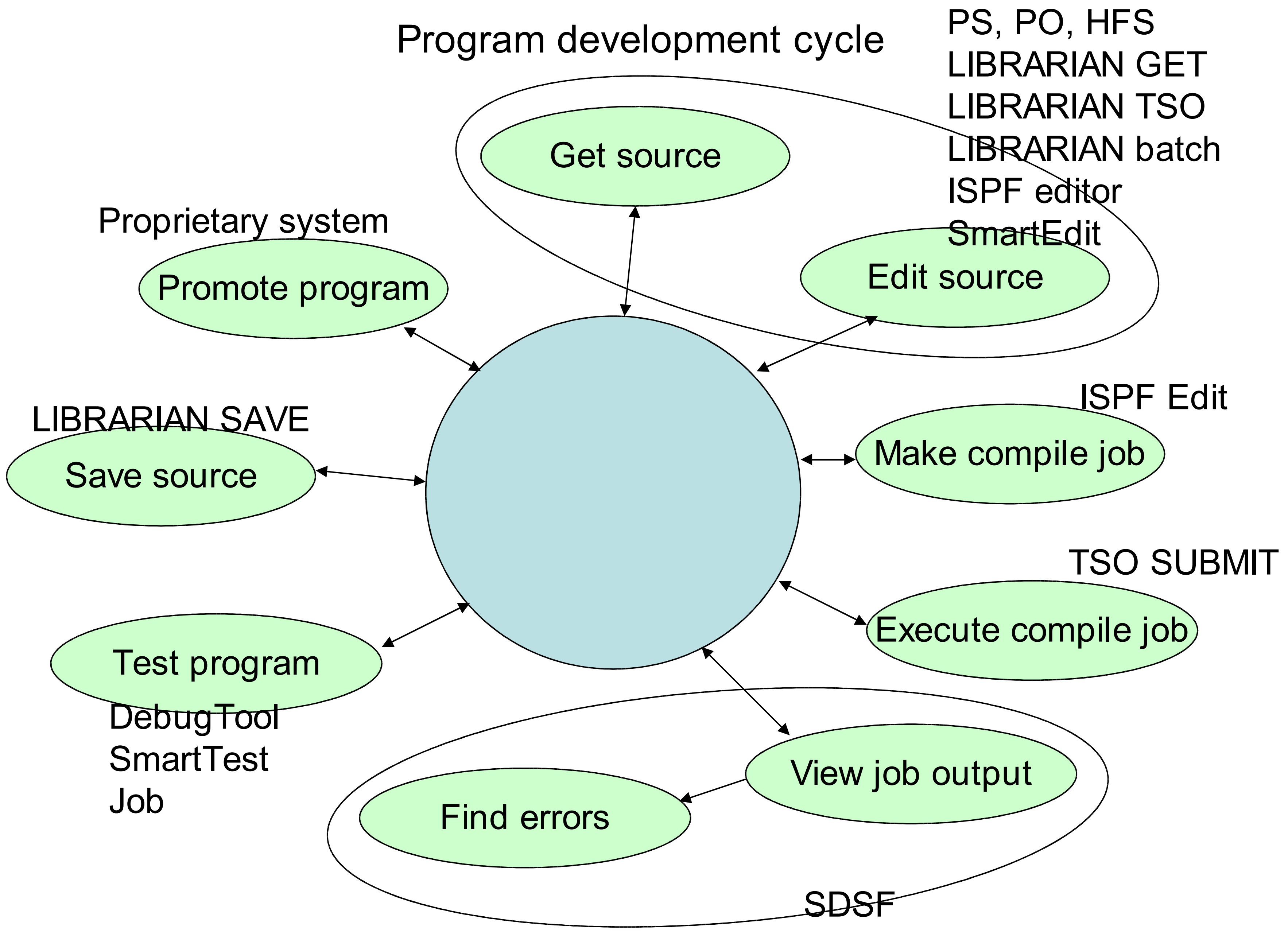
	SUs
Compiled version	95,002
Interpreted version	97,562
Modular variant	183,680
Programs not preloaded	115,556

SUs = service units (a weighted measure of resource usage)

The modular variant = no composite kernel

Programs not preloaded = programs loaded dynamically

Program development cycle



The typical program development life cycle involves:

3. Retrieve the latest source program version from the archiving system (in our case, LIBRARIAN).
4. Edit the program (ISPF or SmartEdit depending on the language, file type or user preference).
5. Build (locate) the compilation job using appropriate procedures. These procedures will depend on the “processors” involved (such as CICS precompiler, DB2 preprocessor, in-house documentation system, etc.). Customise the compilation job with the appropriate libraries (macro libraries, copy book libraries, object (load) libraries), options (compiler, binder), etc.
6. Submit the compilation job.
7. Locate the associated job output in SDSF and check the step return codes.
8. If compilation errors occurred, look through the list for errors, note the error and location; in a separate window, depending on the error, correct the program, JCL, etc., and resubmit the job.
9. If the compile job completed successfully, submit an appropriate test job or invoke an interactive debugger.
10. Depending on the outcome of the test, either continue at step 2) or store the source in the archiving system.
11. Promote program to production (in our case, a proprietary system).

The large number of combinations (processors, libraries, options, etc.) for each specific program entry make a computer-supported system essential.

The host workbench is a REXX-based application that makes wide use of ISPF (and other) components:

- Execs – processing logic, edit macros, action bar processing, panel exits
- Panels – general input, table display, general help
- Skeletons – job control creation, parameter files, input parameters
- Tables (temporary) – display of filtered “program” entries (project, last-used date, etc.), display of temporary results, profile for user-specific inter-session parameters
- ISPF services (ISPF editor, LM services, etc.)
- VSAM – “program” entries
- DB2 – query user-specific database information (such as assigned collection Ids)
- Program interfaces (SmartEdit, LIBRARIAN editor, etc.)

Background

Partially for historical reasons, VSAM was chosen as data storage medium:

- An in-house REXX interface for database-like VSAM file processing was available

- DB2 not used because of “bureaucratic” reasons (easier to change), user files rather than central “file”

General design criteria

- User-friendliness (in the widest sense)
- Modular design
- Ease of maintenance (expandability (new compilers, changed libraries, etc.)).

Modular design

Small independent modules
 = ease of maintenance
 = massive performance penalty (approx. 1 second per module;
 an action typically requires the use of 10 modules
 = an unacceptable 10 second response time!)

Solution: combine individual modules in a monolithic module either manually or using an in-house REXX preprocessor (#include statements) – also as input to the REXX compiler (although only limited performance gain because of the extensive use of external services (principally ISPF)).

User-friendliness

- Integrate components required for the program development process (for example, allocate a library using appropriate default values).
- Where possible, prompt for input (reference list of most recently-used files, available compilers, options).

The user should not need to know anything or should at least be able to obtain the required information, directly if possible.

This requires the integration of program development tools (for example, selectable list of file names, selectable list of member names, etc.).

Problem:
The interfaces to such tools are often programs.
As with individual exec modules, the repeated loading (and deletion) of programs is time consuming (use count = 0).

Solution:
Preload such programs.
The subsequent load sets the use count > 0, namely the program will not be automatically deleted after use.

Ease of maintenance

Maintenance in such an application covers several facets, principally:

- Ease of making changes

- Ease of locating errors

Ease of making changes

As with most applications, the described developer's workbench will be subject to changes.

Some changes are known in advance, in this case, different library names, compiler options, etc. I classify such "frequent" changes as *class 1 changes*.

I classify rare changes as *type 2 changes*. A typical type 2 change in this application would be the addition of a new programming language.

Class 1 changes

Class 1 changes are "planned" changes.

"Global variable" definitions are contained in a separate exec. All subsequent references use the global variable names. Furthermore, the names of the global variables are themselves contained in a fixed global variable.

Definition of variables:

```
$vn = "$proclis $libr $ppli $ebd $cics $eyeball",
     "$pcconv $xedicnv",

     "$seconv2", /* 09.07.2003 */

     "$lked $db2bpkg", /* 02.04.2004 */
…
     "$viast",

     "$cobload cicsload asmload"

"VPUT (" $vn "$VN) SHARED"
```

The application execs then retrieve $VN to get the list of names, which is then used as the subsequent input to retrieve the specific values:

```
"VGET ($vn)"
"VGET ("$vn")"
```

Library names use a similar scheme.

For example

```
cobload = 'IGY.V2R2.SIGYCOMP'
```

The skeletons used to generate the job control then use the symbolic name, in this case, COBLOAD.

For example:

```
//CMP     EXEC PGM=IGYCRCTL,REGION=20M,
//             PARM=&COPT
//STEPLIB DD   DSN=&cobload,DISP=SHR
```

Obviously, it would also be possible the parameterise the name of the compiler. However, such changes occur so infrequently that it is not worthwhile.

Class 2 changes

Class 2 changes are infrequent, "unplanned" changes.

REXX as interpretative language (together with ISPF skeletons (and panels)) is well-suited for such changes. In the simplest case, the name of the associated skeleton. If the "job creation engine" is designed to process the (dynamic) list of processes, it will automatically handle the addition of a new compiler. In the described developer's workbench, approximately 1 hour (note: this does not include any extensive plausibility processing that might be required in panels).

An example of a compiler declaration:

```
$c = "STEP=CMP;SKL=SEWBC;",
     "DDIN=SYSIN;DDOUT=SYSLIN;",
     "DDLIB=SYSLIB=&ulibcmp CBC.TEST.SCLB3H.H",
     "SYS1.SCEEH.H SYS1.SEZACMAC,",
     "LKED=SYS1.SCEELKED CBC.TEST.SCLB3DLL;"
```

with the associated entry in the processor list:

```
$proclis = 'ASM $asm' 'COBOL $cob' 'C/370 $c'
```

This associates the C/370 programming language with the **`$c`** compiler declaration, which in turn uses the **`SEWBC`** skeleton to generate the compilation job.

Ease of debugging

An interactive system, such as the application being described, also requires an interactive debugging capability.

Solution:
A command is provided to accept the names of the REXX execs to be traced. These names are stored in a global (ISPF shared pool) variable. Every REXX exec has prologue code that checks whether its name is contained in this list. If yes, REXX TRACE is initiated, otherwise tracing is turned off.

Disadvantage:
Only output to the screen.

Trace

The dynamic trace capability requires some means with which the name of the module to be traced can be specified.
In the described application, a command. TR *modulename*.

Implementation: This command adds the name to a list stored as an ISPF shared variable (<tracenam>), for example, **`SEWB$BO SEWB$B1`**.

Each exec has a prologue that passes its name to the trace processing function, in the application, **`SEWBTR`**.

The trace processing function then returns either the trace activation command (for example, **`TRACE ?R`**) or a null string depending on whether the exec name is contained in the list of execs to be traced. This returned command is then INTERETed by the invoking exec.

Typical trace output:

```
PROC:SEWB$BO
  1399 *-*     PARSE ARG key parm
       >>>       "APGM1"
       >>>       "                  "
  1400 *-*     boop = lop /* Zeilenoperation, BO, SE,... */
       >>>       "BO"
  1401 *-*     msg  = SEWBGOUT(key)
  1425 *-*      SEWBGOUT:
  1426 *-*      INTERPRET SEWBTR(SEWBGOUT) /* TRACE */
  4073 *-*        SEWBTR:
  4075 *-*        TRACE OFF
       >>>        "TRACE OFF"
  1426 *-*       TRACE OFF
       >.>       ""
```

Trace output, continued:

```
  1402 *-*      IF msg = '' /* Job beendet */
       >>>        "1"
  1403 *-*       THEN
       *-*       msg = BrowseOutput()
  1406 *-*        BrowseOutput:
  1407 *-*        /* CALL OUTTRAP msg.,,'NOCONCAT' */
  1408 *-*        "VGET (boold) PROFILE"
       >>>          "VGET (boold) PROFILE"
  1409 *-*        IF boold <> 'O' & boop <> 'SE'
       >>>          "1"
  1410 *-*         THEN
       *-*         rc = SEWB$BO2(jobid)
  5027 *-*          SEWB$BO2:
  5028 *-*          INTERPRET SEWBTR(SEWB$BO2) /* TRACE */
  4073 *-*            SEWBTR:
  4075 *-*            TRACE OFF
       >>>            "TRACE OFF"
  5028 *-*           TRACE OFF
***
```

Trace, continued

A second trace capability is a high-level trace of labels that provides a general overview of the processing path. This capability is set when the application is activated. The application allows the specification of a switch that is stored as an ISPF shared variable (<trmode>), for example, SEWB /T.

Typical high-level trace output:

```
TRACE:SEWB001
   207 *-*  GetStorageClass:
TRACE:SEWB005
    93 *-*  ReadRec:
   188 *-*   SetUser:
TRACE:SEWB003
   176 *-* TABDISLoop:
***
```

Trace, invocation

Typical exec prologue:

```
/* REXX - SEWB$BO - BrowseOutput */
SEWB$BO:
  INTERPRET SEWBTR(SEWB$BO) /* TRACE */
```

Trace, implementation

```
/* REXX - SEWBTR - Trace */
SEWBTR:
PARSE ARG procname
ADDRESS ISPEXEC "VGET (tracenam trmode)"
IF (trmode = 1) THEN SAY 'PROC:'procname TIME('L')
IF WORDPOS(procname,tracenam) > 0 THEN DO
  SAY 'PROC:'procname
  RETURN "TRACE ?R"
END
RETURN ''
```

The trace processing exec (SEWBTR) handles both trace variants.

Summary

- REXX as interpretative language (together with ISPF services) made it well-suited (ideal) to this application.

- Investing time in the design phase can save (a lot of) time in the maintenance phase.

- Avoid repetitive loading of programs (time-intensive) – preload such programs.

- Loading REXX execs from a library is very time-intensive – pack such execs together in a kernel.

- "Nobody is perfect" – plan for errors. Provide troubleshooting facilities.