The Workspace Manager: A Change Control System for APL

Rexford H. Swain
Independent Consultant
8 South Street
Washington, CT 06793
USA
203-868-0131

Daniel F. Jonusz
Manager, Product Safety Systems
Mobil Oil Corporation
P.O. Box 1031
Princeton, NJ 08543
USA
609-737-5598

ABSTRACT

This paper describes the Workspace Manager (WSM), a tool that helps to support and add discipline to APL system development and maintenance efforts.

The WSM acts as a repository of APL objects (variables, functions, and operators). Programmers use WSM tools to find where objects are used, edit objects, save changed objects, and request that objects be installed into (or erased from) production workspaces.

Periodically, the WSM installs new releases of production workspaces by merging new and changed objects into existing workspaces.

Audit trails are maintained for all of these activities, making it possible to review the change history of an object, compare different versions of an object, compare different releases of a workspace, revert to an old release of a workspace, and so on.

INTRODUCTION

Motive

Mobil maintains a suite of APL-based applications and a large database of health and safety information that is used to protect its employees, customers, and the public, and ensure compliance with state, Federal, and international regulations.

Health and safety data is manipulated by sophisticated logic to produce material safety data sheets and regulatory compliance reports. It is absolutely crucial that the results produced by the system be accurate. But the very nature of the regulatory environment is one of constant flux, so there is an almost continuous need for new and different logic and analysis techniques.

As the amount of mandated compliance legislation has surged over the years, we have had to increase our programming efforts to meet the increasingly complex regulatory requirements. What began as a one-developer application now has up to ten different developers writing code which is used by hundreds of end-users.

While it used to be simple to keep track of who was doing what, at this juncture it is vital that we have some automated means to manage and audit development activities.

Common code is proliferated throughout various applications in the system. When modifications are made, the developers must ensure that their changes will produce the desired result in the specific application under development, and, just as importantly, *not* have adverse effects in any other places they are used. All developers must also be wary of the possibility that one's work may counteract or conflict with another's.

Opportunity

In the past, in order to make a change to a production workspace, a programmer would go through the following procedure:

1. Invoke the application, and break out into the production workspace.
2. Edit and test; edit and test; edit and test...
3. Save new and/or changed code into a temporary file.

4. Send the temporary file to the "master" userid which owns the production workspace.

5. Log on to the master userid, invoke the application, and break out into the production workspace.

6. Copy in the temporary file.

7. Save the workspace.

8. Log off from the master userid.

We wished to streamline this process in general, and we developed some more specific goals. We wanted to be able to:

- Eliminate the bottleneck (and inconvenience) of logging on to the master userid.
- Cut down the number of mid-day changes to production applications.
- Resurrect an erased object.
- Keep a journal of all changes.
- Highlight the differences between similar objects.
- Highlight the differences between similar workspaces.
- Restore an earlier version of an object that has been changed.
- In an emergency, "roll back" an entire application to its state prior to some unfortunate change.
- Provide a mechanism that makes programmers aware of like-named objects that are used in more than one workspace, and whether or not their definitions are identical.
- After a shared object has been changed, allow programmers to easily install synchronized versions into the appropriate workspaces.
- Perhaps most importantly, reduce the possibility that two programmers would unknowingly make simultaneous but conflicting changes to a given program.

We wanted to do all this with a support system that was not too intrusive, not too difficult to learn, and not completely foreign to the experience of a relatively new programmer.

And we had to do all this within a corporate environment and budgeting situation that strongly prefers deliverable, visible application enhancements to the hard-to-quantify benefits of programmer productivity and application reliability.

Weapon

The Workspace Manager described here is implemented on a large IBM mainframe computer running the VM/CMS operating system, the APL2 language program, and the SQL/DS database management system.

IMPLEMENTATION

Note that the main focus of this paper is on the functionality of the WSM, rather than exactly how it is implemented.

Since there are so many differences among the wide variety of language and database systems used by APL developers, specific technical details would not be very useful to many readers anyway.

Instead, we have chosen to concentrate on the desirability of such a tool, and some of our significant design decisions.

Design Decisions

We designed the WSM to be a repository for everything necessary to build a production workspace, and keep a log of all editing and installation activity.

We chose to retain *all* old versions of changed objects so that we would be able to re-build a workspace as of some prior date in case of a disastrous problem. Saving old versions would also mean that no program could be irretrievably lost after being erased by someone unaware of its importance.

We decided *not* to force programmers to use any particular editing and/or testing environment, since every individual has their own preferences and favorite tools. Nor would the WSM try to enforce documentation or coding standards.

We felt that programmers should *request* installations into production workspaces, but that the WSM should actually perform the installations on a periodic basis.

We decided that the WSM would build traditional APL workspaces, rather than dynamically incorporating functions from external files. This is feasible in a virtual memory environment, and makes the thousands of loads

and executions by end users more efficient. It also vastly simplifies life for programmers who want to use traditional active workspace oriented tools to search workspaces and generate function call trees and cross-references.

Overall, the idea was to have the WSM do its job without getting in the way of our talented programmers and their personal working habits.

Terminology

An object is a variable, function, or operator.

A new version number is assigned to each distinct definition of an object with a given name. The syntax *OBJECT*`.`*VERNO* (for instance, *FOO*`.5`) is used to refer to a specific version of an object.

A workspace is a collection of objects.

A new release number is assigned to a workspace as it is put into production. The syntax *WSID*`.`*RELNO* (for instance, *NEWS*`.18`) is used to refer to a specific release of a workspace.

You check out (edit) an object that you wish to change.

You check in (save) an object after you have changed it.

Overview

There are four fundamental steps in the cycle of maintaining APL objects and workspaces with the WSM. Each step is invoked with a full-screen WSM tool:

1. *FINDOBJ*: Find an object (select desired version)

2. *EDITOBJ*: Check out object for editing

3. *SAVEOBJ*: Check in modified (or new) object

4. *INSTOBJ*: Request installation into workspaces

Rather than subject you to a formal description of these four main tools, we will show examples of how a programmer might use them in the all-too-common process of fixing a bug.

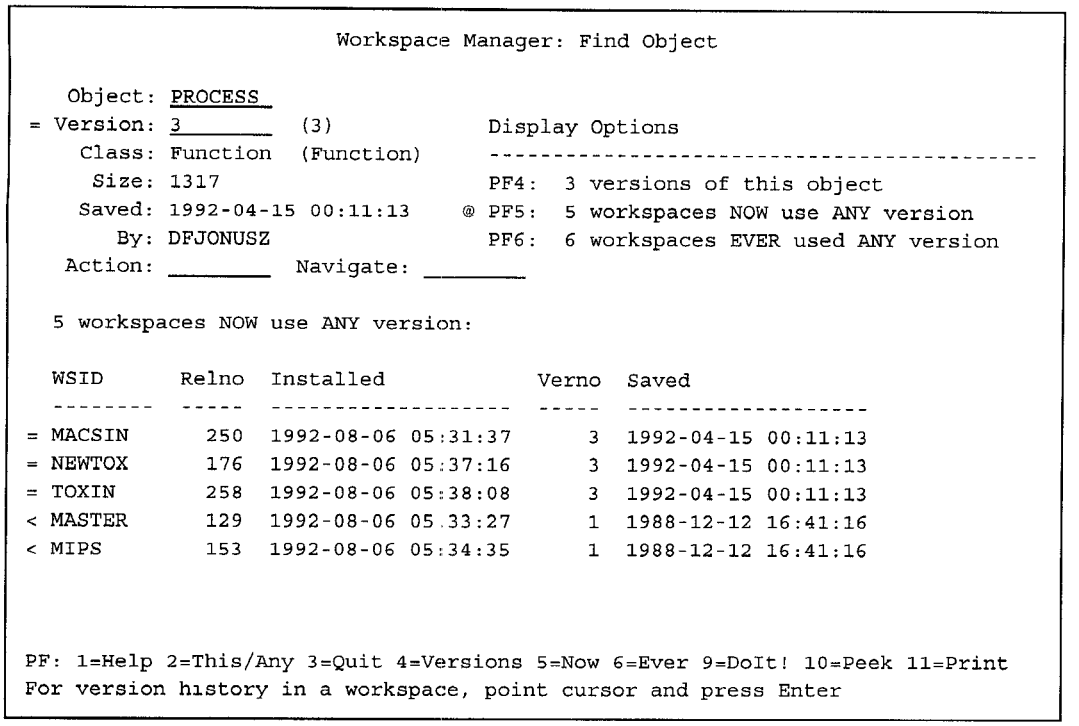

Figure 1: Sample *FINDOBJ* Panel

Step 1: FINDOBJ

The tool *FINDOBJ* is used to identify an object and its usage.

Suppose, for example, you become aware of a problem in a function named *PROCESS*. First you import the WSM tools into your active workspace; then you execute the *FINDOBJ* tool, naming the object you have in mind:

```
) IN WSM
FINDOBJ 'PROCESS'
```

The panel above (Figure 1) appears.

(For publication purposes, input fields in the panel are distinguished by underlining; on a 3270-family display terminal, standard field colors and intensity are used to differentiate the various field types.)

In the upper-left quadrant of the panel, you see that the *most recent* version of *PROCESS* is version number 3, a function, saved on 1992-04-15 at 00:11:13 by userid DFJONUSZ. (The version of *PROCESS* that is in your *active workspace* is shown in parentheses to the right of the version and class.)

The upper-right quadrant of the panel shows some statistics about *PROCESS* and the PF keys to press for three different displays in the lower half of the panel. The "at" symbol next to PF5 indicates that you are currently looking at the list of workspaces that *now* use *any* version of the object.

The lower half of the panel contains the display requested by PF4 or PF5 or PF6. In this example, you see that there are five workspaces currently using two distinct versions of *PROCESS*.

If the problem with *PROCESS* was reported to you by a user of the workspace *MACSIN*, then you learn that the current production release of *MACSIN* is release number 250 which was installed on 1992-08-06 at 05:31:37, and that *MACSIN* uses version number 3 of *PROCESS*.

You also become aware that whatever you do to fix the problem, it better work in workspaces *NEWTOX* and *TOXIN* as well. And you might even look into version 1 and see about bringing those up to date too.

Usually you would just press PF3 to quit after looking at this information. Or, if the selected object was not already in your active workspace, you could type "Get" in the "Action" field and then press PF9.

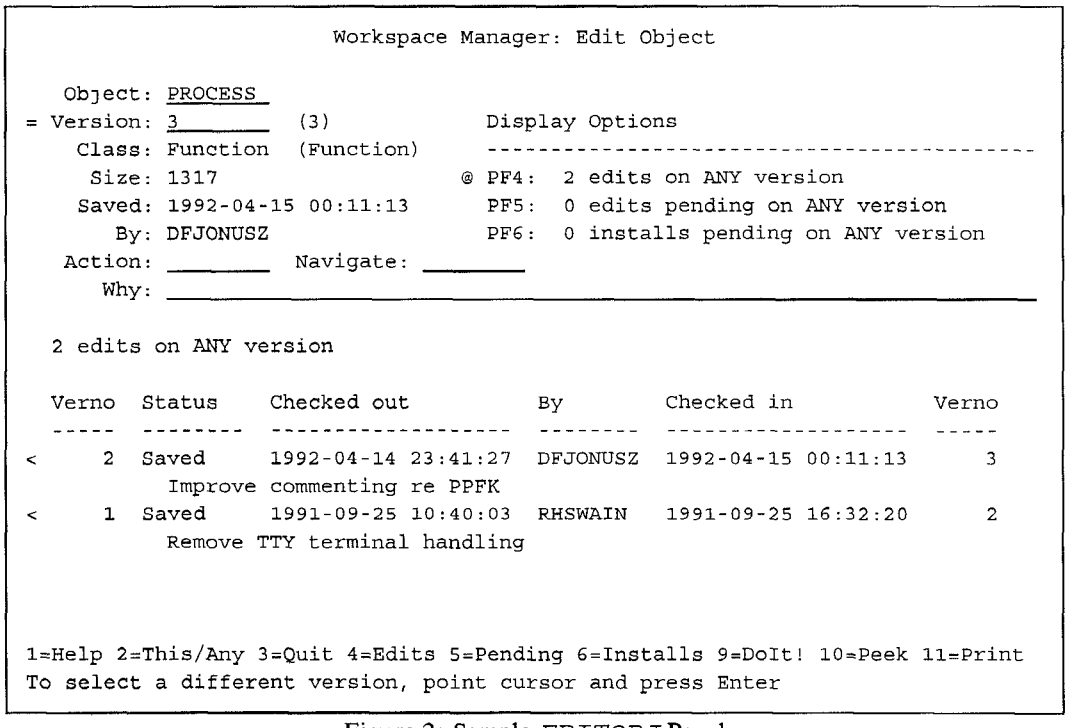

Figure 2: Sample *EDITOBJ* Panel

Step 2: EDITOBJ

Having decided which version of an object you want to work on, *EDITOBJ* lets you check out the object. This is a mechanism that allows other programmers to see that you are in the process of changing the object, and thus helps to avoid conflicts when two people, unbeknownst to each other, go to work on the same object.

Like the *FINDOBJ* panel, the upper-left quadrant of the panel identifies the object/version that you have selected, and the upper-right quadrant shows the three possible displays.

In this example, you would execute:

```
EDITOBJ 'PROCESS'
```

The panel above (Figure 2) appears.

You see that there have been two previous changes to *PROCESS*. Version 1 was changed on 1992-09-25 by userid RHSWAIN, who described the work as "Remove TTY terminal handling", and became version 2 when it was saved. Version 2 was then changed during the night of 1992-04-14 and became version 3.

If someone else had *PROCESS* checked out already, *EDITOBJ* would default to the PF5 display, so you would immediately see who and why. At that point, it would be up to you to negotiate with the other programmer.

In order to actually "check out" this object, you would type "E" (for Edit) in the "Action:" field, tab to the "Why" field and type an explanation of what you are planning to do, and then press PF9.

The panel display would then change to show that there are now three edits, with one edit pending. Your new pending edit would appear at the top of the display, showing version 3, a status of "Editing", and a blank "Checked in" timestamp. (See also Figure 3.)

Note that *EDITOBJ* does not attempt to help you actually edit the object you select -- you have simply posted a notice that you are in the process of changing this object. Any other programmers who attempt to check out *PROCESS* now will default to the PF5 display and see that you have work in progress.

You are now free to use your own favorite techniques to edit and test.

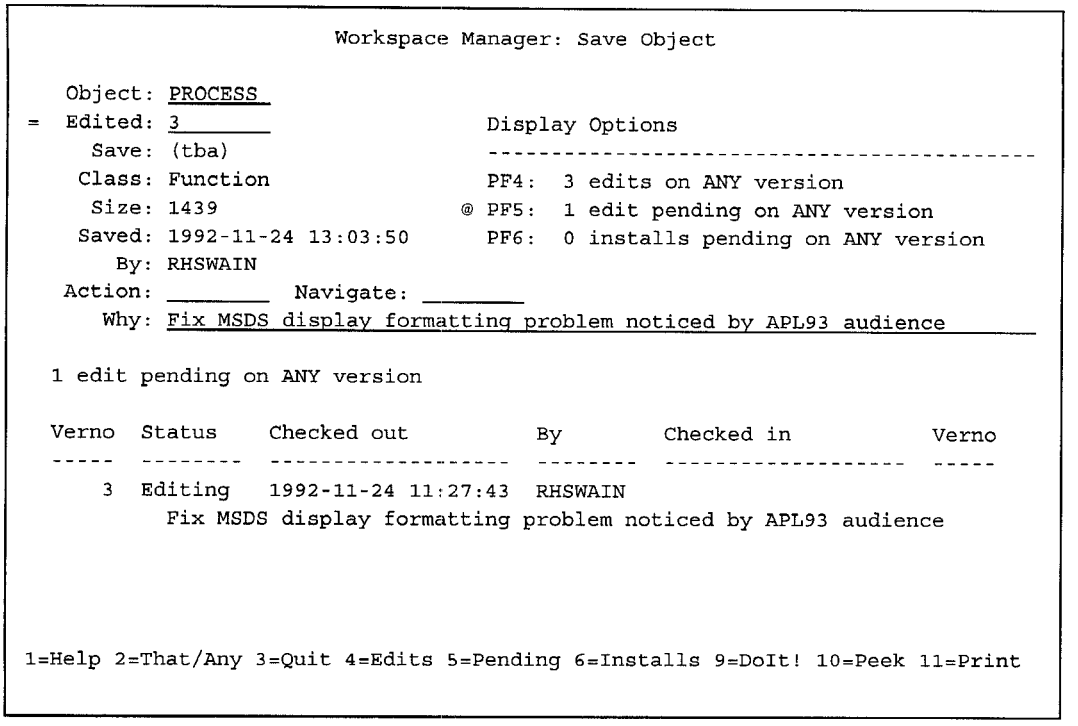

Figure 3: Sample *SAVEOBJ* Panel

Step 3: SAVEOBJ

Having completed and tested your changes, *SAVEOBJ* lets you check in your object, saving it into the WSM database and closing out the edit. When your modified object is saved, a new version number is assigned to it.

Continuing our example, you execute:

```
SAVEOBJ 'PROCESS'
```

The panel above (Figure 3) appears.

SAVEOBJ looks in the edit log to see if you have an edit in progress on *PROCESS*. Since you do, it knows what version you started working on, and is able to retrieve the explanation you gave. (You could change the explanation at this point if it turned out that the problem was something other than what you originally thought.)

To actually save your new version, you type "S" (for Save) in the "Action" field and press PF9.

The WSM then increments the version number, and the panel changes so that version "4" replaces "(tba)" in the "Save" field, and the current time and version number 4 appear in the "Checked in" and "Version" columns.

At this point, you would probably want to proceed to *INSTOBJ* to request that your new version be installed into one or more production workspaces. Rather than pressing PF3 to quit *SAVEOBJ* and then executing *INSTOBJ*, you could use the WSM navigation feature. You would type "I" (for Install) in the "Navigate" field and press PF9.

Find/Edit/Save/Install Sequence

While you *usually* proceed in sequence through the four find/edit/save/install steps, this is not always the case:

- You might stop after the save step if you wanted another programmer to test your new function before installation. Or you might want to postpone requesting an install until you had finished saving several interdependent functions -- you would want them all to be installed together.

- If you want to save an entirely new object into the WSM, you would skip the edit step (there is nothing to check out) and go directly to the save step (you will be saving version 1).

- If you want to install an *existing* object into a workspace, you would go directly from find to install.

```
                    Workspace Manager: Install Object

   Object: PROCESS
=  Edited: 3                        Display Options
  Install: 4                        ----------------------------------------
    Class: Function                 PF4:  8 installs on ANY version
     Size: 1439                     PF5:  0 installs PENDING on ANY version
    Saved: 1992-11-24 13:03:50    @ PF6:  5 workspaces NOW use ANY version
       By: RHSWAIN
   Action: ________  Navigate: ________
    WSIDs: MACSIN NEWTOX TOXIN
      Why: Fix MSDS display formatting problem noticed by APL93 audience

  5 workspaces NOW use ANY version

  WSID      Relno  Installed            Verno  Saved
  --------  -----  -------------------  -----  -------------------
= MACSIN      250  1992-08-06 05:31:37      3  1992-04-15 00:11:13
= NEWTOX      176  1992-08-06 05:37:16      3  1992-04-15 00:11:13
= TOXIN       258  1992-08-06 05:38:08      3  1992-04-15 00:11:13
< MASTER      129  1992-08-06 05:33:27      1  1988-12-12 16:41:16
< MIPS        153  1992-08-06 05:34:35      1  1988-12-12 16:41:16
1=Help 2=This/Any 3=Quit 4=Installs 5=Pending 6=Wss 9=DoIt! 10=Peek 11=Print
```

Figure 4: Sample *INSTOBJ* Panel

Step 4: INSTOBJ

Having saved your new version of *PROCESS*, you use *INSTOBJ* to request that the WSM install it into one or more production workspaces. You execute:

```
INSTOBJ 'PROCESS'
```

The panel above (Figure 4) appears.

The WSM examines the edit log and determines that the "parent" of *PROCESS*.4 was *PROCESS*.3. Since *PROCESS*.3 is currently used in three workspaces, the WSM guesses that you want to install your new version into those same three workspaces -- so they are automatically defaulted into the "WSIDs" field.

(You still get a list of *all* workspaces that use *any* version of *PROCESS* -- this is an opportunity to consider bringing the other workspaces up to date.)

After making any desired changes in the "WSIDs" and "Why" fields, you type "Copy" in the "Action" field, and then press PF9. (Another possible action is "Erase".)

Your installation request then becomes "pending" and is actually performed the next time that the WSM installer job runs.

Object Syntax

When you invoke any of the four find/edit/save/install tools, there a variety of ways to identify the version of an object that you want. If you supply just an object name as an argument, you select the most recent version of the object. But this is only one of six possibilities:

OBJECT	latest version
OBJECT.-1	previous version
OBJECT.4	version 4
WSID.OBJECT	version in latest ws release
WSID.-2.*OBJECT*	version in ws 2 releases ago
WSID.48.*OBJECT*	version in release 48 of ws

The most frequently used form is *WSID.OBJECT*. For example, if you want to check out the version of object *PROCESS* that is being used in workspace *MASTER*, you could rely on the WSM to figure out the version number by executing:

```
EDITOBJ 'MASTER.PROCESS'
```

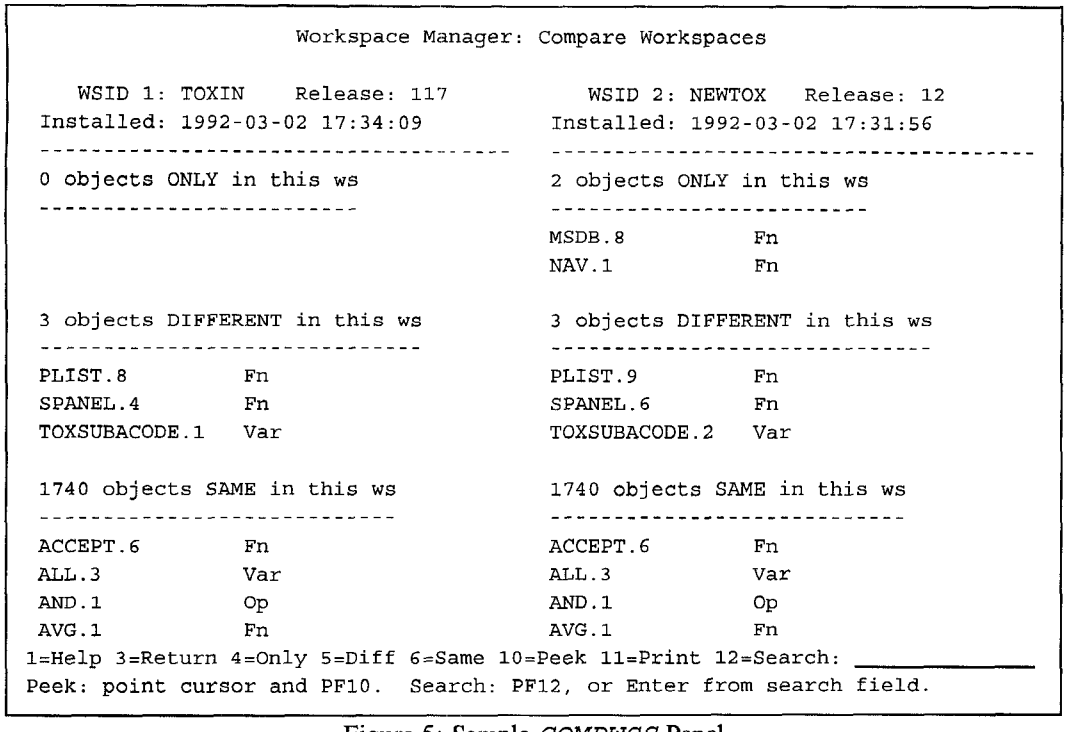

Figure 5: Sample *COMPWSS* Panel

MULTI-OBJECT TOOLS

For programmers with more than a few objects to change (or who wish to avoid panels), the WSM has four non-interactive multiple-object tools with pluralized names:

```
        FINDOBJS 'objs'
'why'   EDITOBJS 'objs'
        SAVEOBJS 'objs'
'wss'   INSTOBJS 'objs'
```

THE WSM INSTALLER

Periodically, a job known as the WSM installer is invoked. It creates new releases of production workspaces by performing all pending installation requests in existing workspaces.

The installer is automatically initiated (by the VM Schedule product) on a daily basis. It is scheduled to run during off-prime hours in order to avoid changes in the midst of a business day (and also to avoid prime-time chargeback rates!).

For each workspace with any pending installation requests, it links to the appropriate disks, invokes APL2 (using the "controlled invocation" feature), loads the workspace (bypassing the latent expression), performs all pending copy and/or erase actions, saves the workspace, and updates the WSM directories accordingly.

Inevitably, it is occasionally necessary to run an unscheduled install because of an urgent need to fix some problem. In this case, the installer may be executed "manually", and the administrator may interactively select which workspaces and/or objects to install.

OTHER TOOLS

Compare Workspaces

The WSM tool *COMPWSS* may be used to compare two different workspaces (or two releases of the same workspace). A sample display is shown above (Figure 5).

Compare Objects

Another WSM tool, *COMPOBJS*, compares two objects. This is particularly useful when investigating the difference between two versions of a given function. The display shows lines which are different (or not found) in either of the two versions.

Various Queries

Many other tools are available to query the histories of edits, installs, and releases; simulate pending installations; show object-by-workspace usage maps; and so on.

EVALUATION

The Workspace Manager has been in active use for more than a year, enabling us to draw some conclusions about our environment, the WSM itself, and our own practices.

System Environment

The combination of REXX, APL2, and SQL make an *extremely* powerful software development environment. (Of course, similar implementations would be feasible using other operating systems, APL dialects, and/or database managers.) Two facilities were especially helpful:

- The APL2 "namespaces" facility (associated processor 11) offered a very clean way to "package" and isolate the WSM code from the programmer's application code, ensuring that there were no name conflicts and localization/shadowing problems. The WSM code essentially disappears from the programmer's active workspace when not in use. All users share the most up-to-date version of the WSM code, even if saved in private workspace.

- The SQL "logical unit of work" facility alleviated two common difficulties with data in shared files: the problem of properly sequencing events when two users simultaneously request a read-and-update, and the all-or-nothing issue when more than one table must be changed at once.

Workspace Manager

After putting the WSM into production, we immediately saw many of the benefits we had anticipated. In particular, being able to see the change history of whole workspaces as well as individual objects has been invaluable.

The biggest problem with the Workspace Manager is that it only manages workspaces! When we install application enhancements that involve modifications to APL objects together with changes to the structure of SQL tables, we still must be careful to synchronize the WSM workspace installations with SQL tables changes. We have a "test" copy of our database that is very helpful for development and testing, but we typically wait for a weekend to put this type of change into production.

There have been some complaints from programmers about the difficulty of orchestrating WSM saves and installs that involve many workspaces and different versions of many objects. But we believe that these problems are inherent to an application environment that involves so many workspaces containing similarly named but different objects. In fact, if the WSM had been in use earlier, many of these difficulties might never have arisen.

Discipline

Programmers who work on applications that are distributed to many remote systems (e.g., by disk or tape) learn to go through a *very* careful testing phase before releasing new software, simply because the difficulty of distributing a fix is relatively high. Conversely, in our experience, programmers in a centralized time sharing environment tend to be less careful about testing, because the "cost" of fixing a bug is relatively low -- they know that a change can be made (for *all* users) in a matter of minutes.

Our hope was that by discouraging ad-hoc application changes and scheduling relatively few installations, we would promote a more careful coding and testing discipline. While it is difficult to measure this sort of thing, we believe that we are seeing some improvement.

CONCLUSION

Clearly, the WSM is not perfect, but it certainly is a step in the right direction. While there is much that *could* be done to enhance the WSM, our feeling is that we are now getting about 80% of the possible benefit after expending only about 20% of the development effort.

REFERENCES

John M. Mizel, *Using SCSS to Manage APL2 Development Projects*; APL92 Conference Proceedings.

Bob Bykerk, *APL Object Manager*; APL88 Conference Proceedings.

David B. Allen, et. al., *LOGOS, An APL Programming Environment*; APL86 Conference Proceedings.