Form-Oriented Software Development

Steven P. Wartik*
Maria H. Penedo†

Computer Science Report No. TR-85-12
July 5, 1985

* Author's Address: Department of Computer Science, University of Virginia, Charlottesville, Virginia.
† Author's Address: TRW Defense Systems Group, Redondo Beach, California.

ABSTRACT

Software reusability is a key part of an effort to increase the productivity of software projects, and is important in building a software engineering environment. However, it is not enough for software to be reusable; for maximum effectiveness, the software should support tool integration, and should have a paradigm that developers can apply. This paper describes a reusable tool called FILLIN, a form-oriented user interface tool. FILLIN has been successfully employed in the Software Productivity System environment as a tool integration and reusability aid. It has helped establish a consistent interface across a wide variety of tools. The tool and its supporting paradigm are presented, along with an example, and the form-oriented view of data in a software development environment is discussed.

Key Words and Phrases: form, integration, man-machine interface, programming environment, software development paradigm, software reuse.

1. Introduction

An important problem in software environments is adding new tools that integrate well with other tools in the system. Ideally, new tools should be able to communicate cleanly with all existing tools. Furthermore, new tools should have a user interface that is consistent with existing ones. Meeting these two goals can simplify the process of learning a new tool's commands, help users understand new tools in terms of old software, provide guidelines for tool builders, and establish reusable tool libraries that reduce development effort.

Integration has received much attention, both in theory and practice. Toolpack[1], Interlisp[2], UNIX[3],* and the Software Productivity System[4] are examples of environments where integration plays an important role. The integration strategies vary widely among these environments. In Interlisp, for example, each "tool" is a LISP function whose characteristics can be modified simply by changing the function; each new tool is also a LISP function that is called directly by the LISP interpreter, and hence has access to the same environment that all other Interlisp tools do. In UNIX, new tools are built and integrated using several aids, including pipes and shell scripts. Pipes produce new tools by combining existing non-interactive tools. Other methods are used to build and integrate interactive software. UNIX tools are therefore more isolated from each other than Interlisp functions, so pipeline integration is not as flexible as the LISP environment but has fewer unexpected side effects. The relative merits of integration schemes pose an interesting problem to researchers. There does not appear to be a single "best" method; suitability depends on application, user community, and other factors[5].

Reusability often influences integration. In UNIX, where tools are invoked through the

* UNIX is a registered trademark of AT&T Bell Laboratories.

shell, tool builders will want to follow command syntax conventions; also, libraries such as the CURSES window management package[6] determine, through their capabilities, how a tool may behave. Such tools and libraries were originally written as reusable software packages; however, by encouraging the developer to conform to certain standards, they help make new tools easier to integrate. The environment's pertinent features are part of the re-used tools. and hence become part of the new tools as well. Therefore, integration strategies should account for reusability.

This paper studies one method of tool integration in the Software Productivity System (SPS). SPS is an environment developed by the TRW Software Productivity Project (SPP) to improve the productivity of software developers at TRW. The SPS is a set of tools built on top of the Berkeley variant of UNIX[7]. It includes the standard set of UNIX tools for text processing, software development, etc. SPP has also built, ported, and purchased many software products that are now included in the SPS. Purchasing or porting existing tools is often more cost-effective than re-implementing them. Thus the SPS includes tools from diverse sources. This creates a significant integration problem, as software vendors do not attempt to provide consistency with each others' tools. For example, the SPS includes the TEX text processing system[8] as well as the standard UNIX tool *troff*. Both perform similar functions, but TEX is highly interactive, whereas *troff* is batch-oriented. Both are necessary parts of the SPS; however, novice users are easily confused by the different tool styles.

Some vendors intend their UNIX utilities to be stand-alone environments. Such tools do not integrate well with standard UNIX tool set, and limits functionality to what the tool contains. We have avoided stand-alone tools, for the SPS changes constantly as its user community expands, and we have found no single product that satisfies everyone's needs. Tying the SPS to one environment is undesirable, since each project that uses the SPS introduces new tools that accommodate some new need. If an environment cannot

easily incorporate these new tools, its power is lessened.

This leaves the problem of finding a suitable paradigm for adding new tools to the SPS while maintaining some degree of user interface consistency and allowing for re-usable software. Since most SPS tools are interactive, man-machine interaction has always been a major consideration[9]. User interface standards were written[10], but we soon realized that relying on tools to provide much of the user interface fosters more effective standardization than do standards documents. Moreover, we had many tools to write, and lacked the man-power to duplicate complex user interfaces. We therefore decided to build man-machine interface tools. Two of these tools are prominent in the SPS user interface: MENU and FILLIN. MENU is a menu driver that gives novice users a consistent interface for invoking tools in the SPS. FILLIN supplies a form-oriented interface to users and to tools.

FILLIN has proven a useful tool in several ways. First, it is an excellent example of re-usable software[11], and has saved considerable effort in the building and maintenance of sophisticated tools. Second, it now provides a simple user interface that allows novice SPS users to quickly perform complex functions. Third, it has facilitated the construction of many rapid mockups of tools[12]. Fourth, it has provided SPS users with a form-oriented paradigm for writing new tools. FILLIN's definition, user interface, paradigm, and applicability to constructing software environments are the subjects of this paper. Section 2 presents an overview of FILLIN and forms. Sections 3 and 4 discuss its form definition and manipulation capabilities. Section 5 covers its man-machine interface. Section 6 is the form-oriented paradigm. Section 7 covers its tool-level interface, both at the shell and subroutine levels. Section 8 gives our experience with FILLIN. Section 9 presents our conclusions, and the directions of FILLIN.

2. An Overview of Forms and FILLIN

As the introduction mentioned, user interface standards are much more easily enforced

through tools than through policies and procedures. FILLIN is an example of an SPS package that helps in developing consistent user interfaces. It is a general-purpose package for performing operations on "forms". A form, from a user's perception, as a linear collection of data together with formatting information that determines its layout. FILLIN's purpose is to present this view of data to a user, together with a user interface with commands for creating, editing, and presenting (displaying for review or printing) a form. To a tool builder, FILLIN provides operators (*a' la* abstract data types[13]) to handle the internal structure of a form and the user interface. FILLIN is primarily a package for manipulating individual forms, since individual forms are what it presents to a user for filling. It can deal with collections of forms if necessary, but does not entail sophisticated algorithms for storing and retrieving forms. These tasks are done by tools that use FILLIN.

Forms as the basis for user interface have long been successful in conjunction with database management system environments (see, for example [14-16],). The advantages of forms are that:

- *They are easy to use.* Everyone has filled out paper forms. An electronic equivalent has proven to be a concept that all users understand.
- *They are simple to build.* The language used to build them has been mastered by many of our non-technical users, who take to form-building much more readily than programming.
- *A "form" is a widely-applicable concept.* Data in most tools can be represented using a form-oriented view. Certain interrelationships between different types of forms are difficult to represent, but can be achieved through methods discussed in section 7.

Most forms management systems are tied closely to a database management system. Unfortunately, database management systems are traditionally stand-alone environments; at best, they contain subroutine-level interface libraries that may be linked into programs. As

stated above, such systems remove much of the power of UNIX, and are not easily integrated into the SPS. We therefore chose to make FILLIN independent of any database management system.

Figure 1 shows a simple form. This form is from the SPS calendar management, and is the form users fill in to schedule an appointment. The SPS calendar management system was one of SPP's first tools to use FILLIN, and will serve as the example for this paper. It is a set of tools that handle personal appointment scheduling. It contains the following commands:

SCHEDULE The user gives this command a single argument—the date of the appointment—and is presented with the form shown in Figure 1. Filling in the fields schedules an appointment at the indicated time. If an appointment already exists at that time, the user is presented with the filled-in form and may make changes to it.

SUMMARY This command produces a summary of appointments for a given date.

RESCHED The user gives the date and time of an existing appointment, and a new date and time; the appointment is moved (or optionally, copied) to that date.

CLEARCAL This command removes appointments.

From a user's perspective, the form in Figure 1 has three parts: a header that describes its purpose, text labeling the data areas of the form, and data that the user enters in these areas. Forms are "filled in" by moving the cursor to the appropriate data area and typing in text. All forms filled through FILLIN are filled in this manner, no matter what the application. Therefore, the form user interface is consistent across all tools that use FILLIN.

Tools use FILLIN when they need to collect input from the user in a form-like way or when they need to manipulate form-like objects. For example, the calendar management

```
Appointment Form
February 24, 1985

Purpose: Automated Office Requirements Meeting
Location: Building R2/Room 2055          Duration: 90 minutes

Notes:
The meeting will cover the preliminary requirements for the new
automated office environment.  Before the meeting, prepare slides
covering the major tools to be discussed.
```

Figure 1: A Form for Calendar Appointments

system uses FILLIN:

- when a user first schedules an appointment,
- when a user reschedules an appointment, and
- when a user makes a change to an existing appointment.

Other tools that use the FILLIN package include a forms management system, a personal information storage and modification package, cross-referencing systems, and a bibliographical information entry tool. Some limitations of FILLIN (to be discussed later) can be circumvented by tools that add necessary constraints or additional functionality through their own software. FILLIN is used mainly for areas of the system related directly to user interface.

The FILLIN package consists of five parts:

1. A *form language* that describes the information of and display format for a form.
2. A *form data language* that defines a mapping between data and forms.
3. A set of subroutines for interpreting the form and data languages.
4. A set of subroutines for handling the user interface.
5. Command-level interfaces to (3) and (4).

The next sections discuss each of these.

3. The FILLIN Form Language

The *fillin form language* is used to define a template for a form, that is, the format and contents of a form. These templates are stored in "template files," which are read by FILLIN routines and converted into an internal form structure. A template file consists of a *header* and one or more *form data lines*. The header is textual information that appears at the top of the form for the reader's benefit. The form data lines each consist of one or more *fields* that define the form contents. Each field is either a *data* field or a *label* field. Data fields define the data contents of the form; they are the fields users fill. Each data field corresponds to a data location where data may be stored and modified. Label fields are textual information that identify (for the user) the purposes of data fields. Tools generally ignore label fields; from their perspective, only the user data is important.

Figure 2-a is the form language definition of the appointment form in Figure 1. Figure 2-b shows a blank form, as the user first sees it on invoking SCHEDULE. The first two lines of Figure 2-a are the header, terminated by the line containing a single period. When FILLIN first displays the form, it copies the header verbatim to the top of the screen, as seen in Figure 2-b (the date is added by the SCHEDULE tool; see section 7). The remainder of the file consists of the form data lines containing the fields. Fields in template files are separated by vertical bars (|). In the figure, the first data line contains three fields, the second contains seven, etc. Empty fields are permitted, so two adjacent vertical bars or an empty line each count as one field.

Fields have three parts, all optional: *attributes*, a *tag*, and *data*. The information inside angle brackets defines a field's attributes. Each field has a specific set of attributes that tell FILLIN how the field should be displayed, and how it may be filled. A default set of attribute values is built into FILLIN; attributes may be specified to override this default. For example, the field '`<h>Purpose:`' has the '`h`' attribute, which tells FILLIN that the field's contents (the string '`Purpose:`') are to be shown using a highlighted display mode, such as

```
Appointment Form

<h>Purpose:| |<dw40>(purpose)
<h>Location:| |<dw20>(location)|<w10>|<h>Duration:| |<dw20>(duration)

<h>Notes:
<dm4>
```

Figure 2-a: Template File for an Appointment Form

```
Appointment Form
February 24, 1895

Purpose:
Location:                    Duration:

Notes:
```

Figure 2-b: Empty Appointment Form

---

reverse video or underlining, if the terminal is capable of doing so (here, highlighted fields are shown using underlining). The next field is a space, used to obtain white space before a data field. The last area on the line is defined by the field `<dw40>(purpose)`. The 'd' attribute means the field is a data field, a place where the user can enter text. The 'w40' attribute means the field is forty characters wide. It occupies forty spaces on the screen, and up to forty characters of data can be entered in it. The field is shown empty in Figure 2-b; in Figure 1, it has been filled with the text "Automated Office Requirements Meeting".

Tags are the strings in parentheses in Figure 2-a. Tags give fields symbolic names. When a tool wants to reference a field's value, it can do so using the field's ordinal position within the form, or using the tag. The former method is quick and acceptable for simple applications, but is confusing in large forms; tags are a more useful referencing scheme, for the same reasons that a constant is often better than a literal in a programming language. Most of the tags in Figure 2-a have values similar to the label fields that precede them, so it is tempting to conclude that the information in the label fields is redundant. However, the labels are arbitrary text, whereas the tags must often fit naming

conventions not established by FILLIN. For example, in the document cross-referencing system that uses FILLIN, the tags correspond to domain names in a relational database management system. This DBMS limits domain names to twelve characters and requires them to be alphanumeric strings. Text in label fields, which is intended to be readable by users, often does not conform to this convention; for instance, "Last Name" is not a valid tag. The redundancy between label fields and field tags is therefore both necessary and desirable.

The last data line in Figure 2-a is for a data field in which a user may place arbitrary notes. This field has the attribute 'm4', meaning it is a *multi-line* field. The other fields in the template file are *single-line* fields. A single-line field can have only one line of text. Its width is specified by the 'w' attribute, fixing both the area it occupies in the form and how much text it can hold. A multi-line field also occupies a fixed area, but can contain arbitrary amounts of text. The amount users see when filling the form is controlled by the integer following the 'm'. In the example, there can be as many notes as desired; however, if a user should enter more than four lines, only the last four will be visible during entry, and only the first four will be seen after the user moves to another field (see below). The effect is similar to a continuation page in a paper form; it gives the form a consistent appearance while being filled in. When the form is presented for output, all information is shown.

Forms may be as long as desired; FILLIN places no restrictions on how many data lines or data fields are present, and will scroll a terminal's screen as necessary to allow access to all fields. In our experience, most forms are short enough to fit on a single twenty-four line terminal screen.

4. FILLIN Form Data Files

Whereas template files store information about the contents and format of a form, *form data files* store, in UNIX files, the data that users enter in forms. Therefore, format

information is not replicated in each form instance. These files serve two purposes:

1. They provide a consistent, readable language for programs that use form data in repeated invocations.
2. They are a temporary data buffer for programs that use the UNIX "fork-exec" paradigm[3].

For example, the calendar management system has separate commands to enter and present appointments, and the form data for an appointment must be preserved on entry to be accessible for review. The SUMMARY tool expects SCHEDULE to create its data files in a certain format. Plainly, the two programs could be written to cooperate, using some format of their own. However, this situation occurs in many SPS tools. Creating code for each would have been contrary to our principle of software reuse. The other choice was to use an existing DBMS available for UNIX; candidates included INGRES, TROLL, and the IDM/500 database machine. Because of their size, these systems have a startup overhead of several seconds. This is acceptable if using a "calendar environment" (where all commands would be part of one tool, allowing the database to remain open through a series of transactions), but is not desirable on UNIX, where commands are invoked from the shell to allow for easy pipeline integration. Furthermore, most "queries" in a software development environment are simple, and concentrate on a single, small set of data (such as a form). Our experiments with the aforementioned and other systems convinced us that their power was unnecessary for most of our applications. Because the form-oriented view of data was to be used throughout the SPS, FILLIN's data format was selected as a standard means for data storage. For UNIX, this is generally significantly faster than a DBMS.

Form data files are used mostly for long-term data storage, but they are also helpful in tool integration. Tool builders often "integrate" two UNIX tools by "forking" a process and having the parent wait for the child, which "execs" (transforms itself into) some other

command. The child will communicate with the parent either through "pipes" or through intermediate files. FILLIN uses the latter. A parent process, such as the calendar management system, will fork and have its child execute FILLIN. While the parent waits, the user enters data in the calendar form. The FILLIN process then exits, leaving its data in a prearranged file that the parent reads and interprets. This approach, common among UNIX tools, is much quicker than integrating subroutines into a program. It also results in small and independent tools, which are easier to maintain.

Data files are generally created by FILLIN. They are automatically written when the user has finished filling a form and exits FILLIN. A data file is a text file (FILLIN does not permit non-textual data to be entered in its fields). It has the same number of "fields" as the form from which it was created. Each field is at least one line long, and contains the data that the user entered in the form. If the corresponding field in the form template has a tag, the tag name is placed before the user data, surrounded in parentheses; if the field has no tag, the tag line is omitted. User data for a single-line field occupies exactly one line (if the user did not enter a value, the line is empty), and appears in the data file exactly as it was shown on the screen. User data for a multi-line field is placed in the data file in its entirety, (not just the amount that appeared on the screen), terminated by a line containing a single period.

Figure 3 shows a data file for the appointment form in Figure 2, as it appears when FILLIN creates it. It can be re-used as data for a subsequent invocation of FILLIN (in which case the form would, on start-up, appear exactly as in Figure 2-a), or it can be interpreted by the calendar manager to schedule an appointment for the given date, or both. Note that the fields are in the same order as in the template file. FILLIN creates data files this way, although for fields with tags the order of fields in the data file is unimportant; FILLIN will read the data into the proper form field. Furthermore, fields can be absent from the data file; FILLIN will leave the field in the corresponding form empty on

```
(purpose)
Automated Office Requirements Meeting
(location)
R2/2055
(duration)
90 minutes
The meeting will cover the preliminary requirements for the new
automated office environment.  Before the meeting, prepare slides
covering the major tools to be discussed.
.
```

Figure 3: Example Data File

initialization. This is useful when creating mockups of new systems that use existing forms, particularly if the mockup does not need data from all the fields.

5. The FILLIN User Interface

The most important feature of the FILLIN package is in how it provides a consistent man-machine interface to tools in the SPS. Users do not have to learn different styles of data entry, only the different ways tools are invoked. Once inside a tool that uses FILLIN, forms are filled in exactly the same manner, regardless of the tool's function. The interface itself is straightforward. It is modeled after how one fills a paper form: move the cursor to the appropriate place (data field) in the form and type in the data.

An important attribute of a fill-in-the-blank interface is the ability to move between fields. This must be quick and simple, as users want to switch fields often and not necessarily sequentially. In FILLIN, the control-N and carriage return keys move the cursor to the next data field, and the control-P key moves it to the previous data field. This approach is adequate when moving to an adjoining field. However, it is too slow when moving half-way through a multi-page form, so the FILLIN interface also includes commands for scrolling quickly through a form. The control-F key moves to the next page of the form, or half-way to the end of the form if the current page is the last. The control-B key moves to the previous page of the form, or half-way to the top of the form if the current page is the first. Because most forms are short, this approach is quite

effective in quickly reaching fields. Even in the longest forms written, no field is more than ten keystrokes away. A "search" command that would scan the form for a textual string and stop at the corresponding data field was once considered, but we realized that, for most forms, it would require more keystrokes (and time) than the scrolling commands.

Once the user has positioned the cursor on the desired field, he may enter or change its data. Data entry involves no complex commands; text is typed directly into the data field. All a beginning user needs to know is how to move between fields and how to quit. The form in Figure 1 could be filled by typing 'Automated Office Requirements Meeting' (when FILLIN begins, it positions the cursor at the beginning of the first data field), moving to the next field and typing 'R2/2055', etc. If the user enters text in a single-line data field that already contains a value, the old field is erased. A simple window editor is also available for single-line fields. However, single-line fields are typically so short that users prefer to retype the field rather than enter the several editing commands required to change a few characters.

In multi-line fields, which may contain large amounts of data, being able to erase text so easily is undesirable. Thus, changing a multi-line field that has data in it requires using a text editor. Users can change the text of a multi-line field using the editor mentioned above (another SPS utility), or invoke the editor of their choice on the contents of the multi-line data field at which FILLIN's cursor was positioned. This approach both obviates the need to create yet another editor and provides access to some extremely sophisticated editors available on UNIX.

For more general processing, FILLIN allows fields to be "filtered" through the UNIX pipe capability[3]. The user types control-A and enters a UNIX shell-level command; this command takes as its input the data of the field, and the field's contents become the output of the command. For example, typing '↑Adate<cr>' will place the current date and time in the current field in the standard UNIX format (↑A stands for control-A and <cr> for

carriage return). Typing '↑Asort<cr>' sorts the contents of a multi-line field. This is a simple yet powerful feature of FILLIN since most shell commands may be used on the contents of any form field.

6. A Form-Oriented Software Development Paradigm

FILLIN was originally intended simply as a re-usable tool. It has grown into an SPP paradigm for software development, a consequence of the vast amount of data in a typical software project that is naturally representable as a form. This includes problem reports, unit development folders, personnel records, configuration management reports, and many other necessary project components[17]. FILLIN has become an influence on any proposed tool; developers look first to determine if its interface is suitable, and try to model the tool's function in terms of FILLIN. If FILLIN is chosen, then tool development proceeds along the following lines:

1. The system is perceived as a set of one or more abstract objects that can be expressed as forms.
2. The relevant operations on the objects are determined.
3. First versions of the forms are created, as FILLIN template files. These are tested until their layout and content are deemed satisfactory.
4. The user interface to the operations is decided.
5. An implementation scheme using FILLIN is chosen for the abstract operations.
6. Shell-level commands are created that access the operations and form objects.

This is not a rigorous, enforced methodology, but guidelines for accessing FILLIN so as to easily construct certain classes of software. The preceding sections have shown how FILLIN has been designed to promote tool integration. The paradigm shows how to create form-oriented software through FILLIN. Aside from the large savings resulting from the re-use

of software, the paradigm has produced an SPS environment with a pervasively consistent user interface.

As an example, consider developing the calendar management system under this paradigm. The steps would be as follows:

1. "Appointments" are the abstract objects that will be expressed as forms. There are other relevant abstractions, such as dates, but they are not form objects. Hence, they must be implemented using some other mechanism.
2. The operations on appointments are scheduling, summarizing, rescheduling, and clearing.
3. A form template file for appointments is created. Prospective users experiment with it, through FILLIN, until they are satisfied with its content and layout.
4. The user interface is chosen to be four separate commands, which is most convenient for UNIX.
5. Each of the four commands is implemented. Because the calendar management system is heavily used, C would be preferable to shell scripts. SCHEDULE and SUMMARY would use the FILLIN library; however, because form data files are ordinary UNIX files, RESCHED and CLEARCAL would not, working instead with the UNIX primitives to move and delete files.
6. The tools are made into generally accessible shell-level commands.

This is not an argument that all data should be expressed using forms, nor should the reader believe that all SPS tools use FILLIN. In some applications, no inherent data separation exists (such as in text formatting), so little is gained using forms. For forms with few fields, it is often quicker to eliminate the form-filling step and work from the command level, supplying all arguments directly. Furthermore, FILLIN does not understand concurrent access to data files; additional software is needed to prevent overwriting and to

implement protection schemes more sophisticated than those of UNIX. Good software design can provide re-usable layers on top of FILLIN, however; the forms management system, which uses FILLIN and supports concurrency, has been used to implement a tool description library.

Step 5 of the paradigm can require a significant amount of work. Users have occasionally argued that FILLIN is too small, and that it would be improved by more built-in functionality of the sort traditionally found in database management systems, such as report generation across forms. Our objective was never to invent something that could be used as a complete development environment, but rather as re-usable software aid to user interface consistency. Unfortunately, each form type has operations that are germane to the application. For example, a useful operation that may be applied to the calendar management system is to take an appointment scheduled for Tuesday and copy it to all Tuesdays from now until year's end. This operation must be performed using primitive database operations that are not usually available without programming. Thus, it is not clear that we would gain much from extending FILLIN to allow this. Few SPS tools contain duplicated code, and we take this as an indication that FILLIN is approximately the right size. This is discussed further in the section 8.

7. Interfaces to FILLIN

FILLIN is often invoked as a UNIX shell-level command. This method is simplest, results in smaller, independent modules, and provides enough power for most applications. Certain programs, however, will choose to access FILLIN by linking to routines through a subroutine library. There are several reasons why this may be desirable:

1. *Efficiency.* UNIX extracts a small time penalty in forking and executing a process. Starting FILLIN in this manner delays the appearance of the form by a slight but noticeable amount. This is rarely significant, particularly when the parent program

starting FILLIN requires some initialization of its own. However, if the parent performs little or no initialization, then the form can be made to appear almost instantaneously, an effect that is much more pleasing to the user.

2. *Data Integrity.* Certain checks on a user's data are not possible through FILLIN. Through the subroutine library, programmers can make arbitrary checks.

3. *Data Storage.* Some systems, such as the SPS forms management system, use FILLIN's data files as the underlying data storage mechanism. Making queries on forms thus requires interpreting data files. To preserve the FILLIN abstraction, FMS uses FILLIN-provided operations to read the data files.

This section covers the command and subroutine-level interfaces to the FILLIN package. It presents examples of using both, showing why each is superior for certain applications.

7.1 The Command Level Interface to FILLIN

There are two commands available at the shell level: FILLIN and FILLOUT. The syntax for FILLIN is as follows:

```
fillin template_file data_file text
```

This tells FILLIN to begin execution using the named template file as the form. The "text" is placed just below the header (as in the date from Figure 1). If the data file does not exist or is empty, FILLIN creates an empty form and places it on the screen for the user to fill. If the data file contains data, then that data is placed into the form, and the form is presented for the user to modify. When the user gives the command to exit, FILLIN writes the data from the modified form into the named data file and quits.

Another command, FILLOUT, lets users view forms on the screen or send them to a printer. Its syntax is identical to that of FILLIN:

```
fillout template_file data_file text
```

However, FILLIN is seldom invoked in this manner, because users do not want to be troubled with remembering the association between form types and data files. Instead, a tool is built that invokes FILLIN to collect the data; on completion, the tool may extract data from the data file, or arrange for the data file to be stored in some useful location. For example, the calendar management system works by creating a directory file for each date that contains an appointment, and storing individual appointments in a file whose name is the time of the appointment. The concatenation of these names therefore form the data file name, and is passed to FILLIN as the second argument. To form this name, it needs to extract the time and date, which always occur on lines 6 and 8, respectively, of the data file (so for new appointments, it actually must create the data file elsewhere, extract the date and time, create the directory for the date, and finally move the data file to its new location).

This approach is excellent for creating mockups. Figure 4 shows a "shell script" (a set of commands, placed in a file, that are executed by the UNIX command interpreter as though typed on the keyboard; the notation "$n" means to substitute the *n*'th argument to the script) that simulates much of the functionality of the scheduling and summary tools, using the aforementioned approach. The real tool is far more complex; it includes such features as conflict resolution and date and time normalization. However, it uses the same interface to FILLIN, choosing the simplicity of the command-level interface over the (here unnecessary) power of the subroutine library.

7.2 The Subroutine Level Interface to FILLIN

One objective of building the subroutine package was to create abstract data types that programs could use as a standard way to refer to FILLIN forms. Accordingly, FILLIN subroutines manipulate two new data types called `TEMPLATE` and `FIELD`. `TEMPLATE` objects are an internal representation of a form. They contain the information found in a form template file, along with filled-in data. A `TEMPLATE` is created by the `read_template` subroutine, which

```
date=$1
if test ! -d "$date"
then mkdir "$date"
fi
fillin /usr/local/lib/templates/APPT /tmp/cal-data
time=`sed -n -e 6p /tmp/cal-data`
mv /tmp/cal-data "$date/$time"
```

Figure 4-a: A Mockup of the Calendar Management System Scheduler

```
date=$1
chdir "$date"
for time in *
do
     fillout /usr/local/lib/templates/APPT "$time"
done
```

Figure 4-b: A Mockup of the Calendar Management System Summarizer

reads a template file and returns an appropriate data structure. Several other operations may be performed on TEMPLATE objects (through the indicated subroutines):

1. A template may be "merged" with a template data file. This places the data field values of a specified data file in the fields of a specified template.
2. Any field of a template may be accessed. Data fields may be accessed by their ordinal position, or by their tag. Label fields may be accessed by their tag.
3. Names and values of a data field may be accessed.
4. The values of each field of a template may be set. As with accessing, fields are referenced either by position or by their tag.
5. The template can be presented to the user for filling or editing.
6. A data file can be created from the values of the template data fields.
7. A data file can be shown to the user as a form.
8. Miscellaneous functions exist including interrupt-handling, error-handling, and cleanup.

Figure 5 shows an example of using these routines. This is a simplified version of the

```
main(argc, argv)
        int     argc;
        char    *argv[];
{
        TEMPLATE        user_template;
        char            *template_name,
                        *datafile_name;

        template_name = argv[1];        /* Extract the template file name. */
        datafile_name = argv[2];        /* Extract the data file name. */

        user_template = read_template_file(template_name);
        merge_datafile(user_template, datafile_name);
        fillin_driver(user_template);
        write_template_to_datafile(user_template);
        fillin_cleanup(user_template);
        exit(0);
}
```

Figure 5: The FILLIN Driver Routine

shell-level version of FILLIN. The program consists of a main routine that interprets command-line arguments, extracting the name of the template file desired and the data file to be used. The template file is read and the resulting structure is merged with the data file; this structure is then passed to the driver routine, allowing the user to place data in the form's fields. After the user quits, the driver routine returns, the data template structure is put back in the data file, and the routine exits. This program is linked with the subroutine library to produce the FILLIN driver. The actual routine includes more error-handling and diagnostic capabilities, but this example suffices to illustrate both the basic structure of FILLIN and how one uses the subroutine library.

Applications may use this calling sequence to invoke FILLIN. Often, they will want to set field values before displaying the form to the user. For example, some applications that use FILLIN automatically "timestamp" the form by placing the current date in a predetermined field. This may be done directly after reading the template, using the `set_field_value` routine. The form designer will create a field with a tag called "date" and use code such in Figure 6. This is a general mechanism that allows full control over data values assigned to fields. As remarked earlier, it is difficult to anticipate the range of

```
TEMPLATE template;
char     *timestamp;
...
template = read_template(template_file_name);
timestamp = /* current date */;
set_field_value(template, "date", timestamp);
```

Figure 6: "Timestamping" a Form

built-in functions that users will require; thus, instead of placing in the template language the ability to automatically read in the date, time, user login name, terminal port, etc. (a small sample of the built-in values we have seen), and having to modify FILLIN each time a user requires a new built-in value, we opted to give the application-writer complete freedom at the expense of additional code.

The routines that interpret template and data files may be used independently of the other routines to provide a simple relational database mechanism that, for small application systems in the SPS, is faster than a full DBMS. A template provides a descriptor of the attributes of a relation. Thus, a set of data files created from a single template type is analogous to a relation. One can step through each tuple relatively simply, as shown in Figure 7. This code fragment prints the purpose of all meetings that are held in building R2. What would replace the pseudo-code depends on where the tuples are stored. For a typical application, the program might interpret each file in a directory as a FILLIN data file. In that case, the condition for the while-loop would be replaced by a test for end-of-file in reading the directory, and the pseudo-statement for merging the tuple would be a call to `merge_datafile` with the next file in the directory as an argument. Alternately, the application might be storing its data using a DBMS. If so the "merging" would be a series of calls to `set_field` (or perhaps `set_named_field`) to convert the tuple data into a FILLIN structure.

This code can be tailored to provide more sophistication, if needed. Applications of

```
TEMPLATE t;
FIELD    f;
char     *loc;
...
t = read_template("template_file");
while ( more tuples remain ) {
        merge next tuple into t;
        f = named_field(t, "location");
        loc = field_value(f);
        if ( loc[0] == 'R' && loc[1] == '2' )
                printf("%s\n", field_value(named_field(t, "purpose")));
}
```

Figure 7: Stepping Through A Set of Tuples

FILLIN such as the forms management system contain subroutines that extend this code to include the projection and selection operations of a relational algebra. This can be done by adding code in the above loop that accepts only tuples with certain field values, or that prints only certain fields. Clearly, there is a point of diminishing returns where using the query power of an existing DBMS becomes more effective. For simple systems, however, FILLIN is often easier to use. Moreover, it is portable among different database management systems. Our benchmarks show that the IDM/500 database machine is faster than a DBMS implemented in software, but not all remote SPS sites can afford one. The FILLIN scheme solves this problem.

8. Experience with FILLIN

Our experience with FILLIN has been very positive. It is used in many SPS tools, including the calendar system, the forms management system, and the interoffice correspondence tool. SPP personnel have created many form types for project support. SPS users, including secretarial, administrative and technical personnel, have also created their own form types. These form types support office, business, personal, and technical activities.

Due to its role as an SPS user interface tool, FILLIN has undergone many changes since its inception four years ago. There have been three distinct versions, each with intermediate modifications. The original package did not include the subroutine library; programs

using FILLIN had to read and interpet the form data file. The second version included the subroutine-level support for manipulating form data files, and also added the field tag concept. The current version saw significant changes to the user interface, including the support for arbitrary editors and scrolling discussed previously.

User interface issues have always presented the most challenging problems in FILLIN. Intermediate revisions have incorporated users' suggestions on what they believed would make forms entry easiest. Usually, experimentation has proven the only way to resolve conflicting opinions. For example, the "next-field" key was originally carriage return; in the second version, it was control-N, in a subsequent unreleased version it was a tab, and in the current version it became carriage return or control-N. Not everyone agrees on what makes the best user interface, but what exists now seems a reasonable tradeoff. This version has been in use for almost two years, and users are satisfied with it.

The FILLIN package continues to grow in response to needs for additional support. While template files are text files and can be created using any text editor, users expressed the desire for a "what-you-see-is-what-you-get" screen-oriented approach. Understanding the spatial relationship between fields in a template file can be confusing, especially for a novice user, and a screen-oriented tool eliminates having to examine them. Therefore, SPP developed a template file editing tool called FILLEDIT that allows users to create and modify forms without needing to understand the FILLIN form language.

Extensions to the form language are often proposed. The most common criticism of FILLIN is that it cannot model hierarchical data. This is desirable, but difficult to implement in a general fashion; the suggestion has come from three parties, with three *ad hoc* ideas on how it should be done. Moreover, it is not clear that FILLIN is the right place for this constraint, since FILLIN is primarily a user interface tool. While it can, if necessary, provide a data storage medium, applications with hierarchical data relationships tend to be sufficiently large and complex as to warrant a formal DBMS. In that case, a tool

can provide the bridge between the FILLIN interface and the database. Most forms are linear, not hierarchical; it is the relationship between different form types that is not one-to-one. Therefore, the user interface does not have to model hierarchical relationships. There are, however, forms whose data has inherent many-to-one relationships. We are attempting to devise a general solution to this problem.

Another proposed extension is to associate data types with data fields. A field's attributes could specify that the field can contain integers, floating-point values, dates, etc. This change is useful and applicable to many tools, but also requires additional support for error-processing. Generic messages such as "invalid integer value" are not always helpful enough. So far it has proven easier to let the applications check the values rather than building them into FILLIN. These are some examples of the difficulties we have encountered in trying to make what at first seem simple extensions of a general-purpose tool.

9. Conclusions

In this paper we have presented an important tool that has been widely used and re-used in the SPS environment. We have discussed its general capability, and some of our experiences with its development and use, from both users' and implementors' points of view.

FILLIN's most important aspects are preservation of user interface across tools, integration ability to existing tools, and reusability in building other tools. The FILLIN package has been extensively re-used as a man-machine interface package and as an abstract data type package that deals with form-like objects. It is a building block for environment support. Its success is due in part to the utility of forms, and in part to how easily it integrates with other tools at both the command and subroutine levels. The SPS development continues, placing emphasis on tools similar to those presented in this paper, and we believe that FILLIN will continue to play an important role.

Acknowledgements

We would like to acknowledge all members of the Software Productivity Project at TRW for their hard work and creative ideas, and our user community for their valuable feedback.

REFERENCES

[1] L. Osterweil, "Toolpack—An Experimental Software Development Environment Research Project," *IEEE Trans. on Software Eng. SE-9*, 6 (Nov. 1983), pp. 673-685.

[2] W. Teitelman and L. Masinter, "The Interlisp Programming Environment," *Computer 14*, 4 (Apr. 1981), pp. 25-34.

[3] B. Kernighan and J. Mashey, "The UNIX Programming Environment," *Computer 14*, 4 (Apr. 1981), pp. 12-24.

[4] B. Boehm, M. Penedo, A. Pyster, E. Stuckle and R. Williams, "A Software Development Environment for Improving Productivity," *Computer 17*, 6 (June 1984), pp. 30-44.

[5] L. Osterweil, "Software Environment Research—The Next Five Years," *Software Eng. Notes 6*, 4 (Aug. 1981), pp. 15-24.

[6] K. Arnold, *Screen Updating and Cursor Movement Optimization: A Library Package*, Unix Programmer's Manual (4.2 Berkeley Software Distribution), Volume II-D, Berkeley, CA, 1983.

[7] *UNIX Programmer's Manual*, 4.2 Berkeley Software Distribution, Berkeley, CA, Aug. 1983.

[8] D. Knuth, *The TEXbook*, Addison Wesley, Reading, MA, 1984.

[9] B. Schneiderman, *Software Psychology: Human Factors in Computer and Information Systems*, Winthrop Publishers, 1980.

[10] M. Penedo and A. Pyster, "Software Engineering Standards for TRW's Software Productivity Project," *Proc. 2nd Software Engineering Standards Application Workshop*, San Francisco, CA, May 1983.

[11] T. Standish, "An Essay on Software Reuse," *IEEE Trans. on Software Eng. SE-10*, 5 (Sep. 1984), pp. 494-497.

[12] S. Squires (ed), "Special Issue on Rapid Prototyping," *Software Eng. Notes 7*, 5 (Dec. 1982).

[13] J. Guttag, *The Specification and Application to Programming of Abstract Data Types*, PhD Thesis, Computer Science Dept., University of Toronto, 1975.

[14] *INGRES VIFRED (Visual Forms Editor) User's Guide*, Relational Technology Inc., Berkeley, CA, 1982.

[15] M. Mannino and J. Choobineh, "Research on Form Driven Database Design and Global View Design," *IEEE Computer Society Technical Committee on Database Engineering 7*, 4 (Dec. 1984), pp. 58-63.

[16] D. Tsichritzis, "Form Management," *Comm. ACM 25*, 7 (July 1982), pp. 453-478.

[17] E. Goldberg, "Applying Corporate Software Development Policies," *Proc. AIAA Third Software Life Cycle Management Conference*, 1978.