Form-Oriented Interface Abstractions in Reusable Software

Steven P. Wartik

Computer Science Report No. TR-86-18
July 3, 1986

Author's Address:

Department of Computer Science
Thornton Hall
The University of Virginia
Charlottesville, Virginia 22903
804-924-1034
CSnet: spw@virginia

Abstract

Research in form-oriented user interfaces has produced many popular software systems. However, it has not adequately considered some of the fundamental concepts underlying a form abstraction. As a result, forms have not lived up to their potential. Forms should be an independent abstraction containing all the advantages of their paper equivalent, and a system supporting them should contain reusable software that aids tool builders in creating user interfaces.

In this paper we discuss what we believe is necessary to support a form abstraction properly. We discuss forms as objects independent of any data storage mechanism, and we show the utility of reusable software components centered around form-oriented interfaces. We illustrate our points through FILLIN, a reusable software package supporting forms, and we cover some recent extensions to FILLIN that show what reusable software requires to support forms. We present our experience with these extensions, and discuss our directions for future research.

1. INTRODUCTION

Forms are a popular abstraction in computer systems today. The concept of a form is simple and easily understood: everyone knows, from their day-to-day experiences, what it means to fill in a form. Researchers have also discovered that forms extend naturally into computer applications. They have been used successfully as database interfaces [11] and as database design tools [8]. Conversely, they have been used for command interfaces, as an aid to command construction [5]. They have also found application in tool integration and software reuse [16].

Forms are most useful in handling data that can be treated as a set of ordered N-tuples [15]. In a relational database, for example, each tuple in a relation can be mapped to an instance of a form by mapping each domain to one field in the form. In a network database, each record can be mapped to a form in a similar manner. For a command interface, a command and its arguments provide a similar, albeit less well defined, structure. More importantly, from a user's perspective, many applications map easily to a form-oriented structure. In automated office environments, forms can model addresses, employee history, phone books, appointment calendars, etc. In software development environments, forms can be used for problem reports, project status reports, design specification reports, etc.

Because many forms map easily to database structures, DBMS vendors typically supply a forms management package for querying databases and generating reports. While a DBMS is often a convenient storage manager for a set of forms, forms are not inherently related to databases. Indeed, from the user's perspective, a form is a convenient abstraction quite independent of a DBMS. Users perceive forms in two ways: as an object being manipulated by their computer system, and as a user interface style for data entry and modification. Because a system typically deals with a set of forms, a DBMS is frequently used to store the forms in the first case. For the second case, however, applications often work with a single form, not a large set, and the user does not think of a database being involved. That is, forms and databases are distinct entities, and, while forms are often conveniently mapped to databases, forms have other important uses as well. Thus there is a need to separate a data storage mechanism—DBMS or otherwise—and its man-machine interface.

This idea has been explored in the FILLIN package [16], a set of tools for manipulating forms independent of any back-end application. As explained elsewhere [2,10], it was developed as part of the Software Productivity System (SPS) to provide a uniform interface to a wide variety of tools, including tools for such software development tasks as software problem reports and unit development folders, and tools for such automated office functions as calendar management and interoffice correspondences. Moreover, it served as *reusable software* [14]: SPS required a uniform interface that could be useful to novices after minimal training and yet amenable to sophisticated expert use. A form-oriented interface seemed a reasonable approach, and, given the project's resources, software reuse was the only feasible solution [1].

Hence, FILLIN gave many of the advantages of form-oriented DBMS interfaces, yet was independent of a DBMS. SPS tools could, when needed, provide the bridge between FILLIN and a DBMS; some of the aforementioned tools worked in this fashion. Tool builders therefore have the best of all worlds: the power of a DBMS, the sophistication of a form-oriented interface, and the freedom to easily access any part of the host O/S environment.

However, FILLIN lacked some features typically provided by a DBMS form-oriented interface. For example, SPS implemented a "query-by-forms" tool for data retrieval from a central database of forms. FILLIN could not provide interactive review of the retrieved set of forms; a user had to exit the tool and use another to scan through the forms. The user had to learn two tools instead of one; also, there was an undesirable context switch from the interactive form-oriented retrieval to the non-interactive data scan.

This and other problems discussed below led us to examine the concept of a form-oriented interface in more detail. The resulting extensions to FILLIN form the subject of this paper. We treat these extensions from the point of view of form-oriented interfaces as reusable software components. Section 2 gives an overview of FILLIN. Section 3 covers the problem area in detail. Section 4 presents our solutions. Section 5 discusses our conclusions, and the state towards which FILLIN is evolving.

2. AN OVERVIEW OF FILLIN

In this section we give a brief overview of FILLIN. It is a tool package implemented on the Unix[1] operating system [6], concerned primarily with user interfaces. It presents users with a form-oriented view of data, allowing them to enter and modify data by "filling-in-the-blanks" of a form, displayed as a two-dimensional image on a CRT screen. This is a general-purpose interface style that is adaptable to a wide range of tools.

FILLIN operates on *forms*, where a form is a linear collection of data, together with formatting information. The formatting information determines the layout of the data on a *form image*, which is the visual representation of a form—*i.e.*, what users view when dealing with forms. Users perceive a form image as consisting of two parts: the areas where they enter data, called *form data fields*, and surrounding text that identifies the data areas, called *trim*. Form data fields contain textual data—anything that can be represented on a CRT screen. They can be *single-line* or *multi-line*. Single-line fields contain exactly one line of text, of a fixed width (hence a fixed maximum size). Multi-line fields, as their name implies, may contain any number of text lines; however, since a form provides a specific area for each datum, not all of this information is shown if more exists than will fit on the screen. Both single-line and multi-line data fields have a data type, such as "integer", that defines the syntax of data in the field. Data types are given as regular expressions. They prevent most common input errors; as explained in Section 4, a more sophisticated method of error detection is available.

The purpose of FILLIN is to present the above concepts to a user, through a user interface with commands for creating, editing, and presenting (displaying for review or printing)

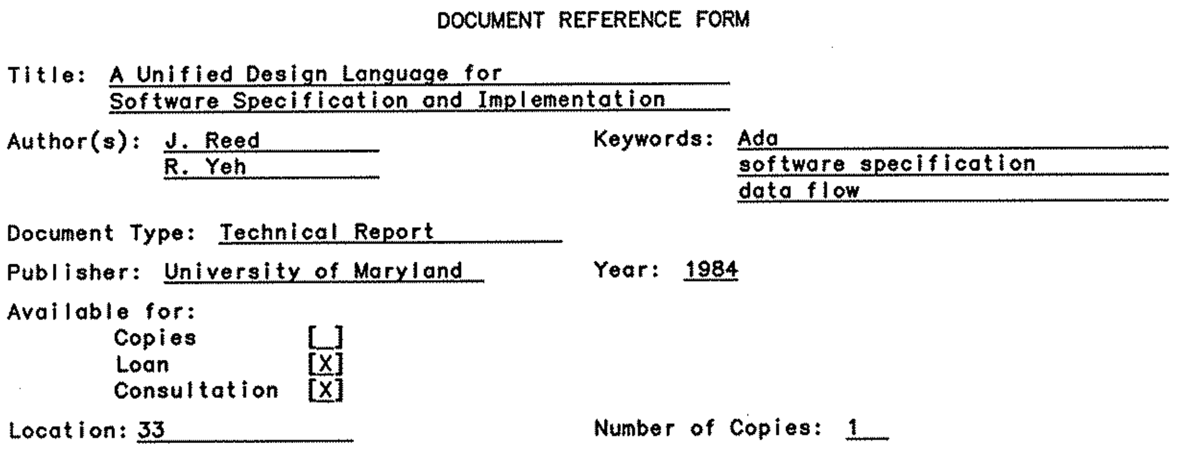
1. Unix is a registered trademark of AT&T Bell Laboratories.

a form image. To a user, FILLIN provides a view of data that corresponds to what he or she expects to do with an ordinary paper form. To a tool builder, FILLIN provides a set of abstractions (data types together with operations) for manipulating instances of forms. While it can deal with collections of form instances if necessary, its algorithms for storing and retrieving form instances are deliberately simple. Tools use FILLIN when they need to collect from a user input that is conveniently entered as a form, or when they need to manipulate form-like objects. Applications that require storing and retrieving many forms provide their own DBMS, and establish an interface between the DBMS and FILLIN. Applications that deal with only a few forms generally use FILLIN's slower but simpler form storage mechanism. For example, the SPS forms management system [10], which was designed to be able to handle a large project's needs, uses the IDM/500 database machine [3]. The SPS calendar management system, designed for maintaining a personal appointment schedule, uses FILLIN's capabilities, storing form instances as ordinary files in the Unix directory hierarchy.

Figure 1 shows an example of a form image, similar to that used by the SPS library management system. This form is used to catalogue documents in the library, giving their location (as a folder in a set of filing cabinets) and cross-listing them under a set of keywords for reference and indexed retrieval.[2] This form has several multi-line data fields: title, author, and keywords. The other fields are single-line, some with data types. For example, the year is a four-digit integer.

The library management system uses FILLIN for several tasks. First, new references are recorded by filling in a copy of the above form. Second, existing references are modified by editing the form through FILLIN. Third, users search the library database by filling fields of the above form, using a "query-by-forms" (QBF) style [11]. In each case, FILLIN

2. In the examples, the areas where users enter data (the form data fields) are underlined; when using the tool, this is obvious by context.



DOCUMENT REFERENCE FORM

Title: A Unified Design Language for Software Specification and Implementation

Author(s): J. Reed, R. Yeh

Keywords: Ada, software specification, data flow

Document Type: Technical Report

Publisher: University of Maryland

Year: 1984

Available for:

- Copies [ ]
- Loan [X]
- Consultation [X]

Location: 33

Number of Copies: 1

Figure 1: Library Management System Form Image

supplies form-oriented portion of the user interface. For example, the tool that enters new references works by invoking FILLIN with an instance for the reference form. FILLIN then places a form image of this instance on the screen, and the user enters the desired information. When he or she exits FILLIN, the library tool is given access, through a set of FILLIN routines, to the data the user entered in the form. It then takes this data and stores it in the database.

FILLIN therefore serves as reusable software for user interfaces. It provides a set of procedures, functions and data types that support an object-oriented view of software [4]. The principle objects are forms, data fields, and the terminal's screen. Using the technique outlined in the last paragraph with variations that depend on such issues as where the data is to be stored, whether data is being created or modified, what sort of error checking is needed, etc., tools achieve a sophisticated interface with minimal effort.

FILLIN also provides reusability in another way. All functions available through the subroutine library are also available at the Unix shell level, through a set of command-level application tools. In Unix, the shell language is often used for rapid prototyping [12]; the Unix pipe mechanism and the rich tool package make construction of many applications straightforward, if somewhat slow and not always robust [7]. FILLIN's tools provide access to this capability. More importantly, the rapid prototypes that are built through FILLIN are usually similar in design to the eventual implementation. Therefore, FILLIN also supports reusability of design components [9]: the prototype's structure can (but need not) drive the software's design. Because the same abstractions exist at both the shell and C levels, the transformation of the user interface from prototype to implementation is almost immediate.

3. REQUIREMENTS FOR A COMPLETE FORM ABSTRACTION

The version of FILLIN as described in the previous section was valuable in providing a consistent user interface; also, it integrated easily into the SPS environment. However, FILLIN did not provide a complete form abstraction. The paradigm discussed in the previous section required that FILLIN be in complete control during the time when a user filled a form; the tool using FILLIN did not regain control until the user indicated he or she was finished. This introduced two user interface context switches, from the tool's to FILLIN's back to the tool's. As a result, several important functions could not be performed within a form-oriented context:

1. *Form Validation.* FILLIN could handle simple syntactic validation of data entered for an individual data field (requiring that a field is a positive integer, for instance) but could do nothing beyond that. It could not require that one data field's be filled in only if another is (*e.g.*, first name makes no sense unless a last name is given).

2. *Review a Set of Forms.* This operation is analogous to flipping through a stack of paper forms. It can be done for *N* forms by invoking FILLIN *N* times, but the continual context switch is irritating.

3. *Query Processing.* QBE (Query-by-Example) [18] and QBF [11] are popular query styles; IDE's TBE [17] uses a similar style to help a user browse through relations in a database. The SPS forms management system had a query facility similar in functionality to QBF. It was clumsier, however, because FILLIN required a context switch to the command level after the fields of a form were filled.

4. *Apply a Command to a Form Instance.* While commands could be applied to individual fields, it was not possible, when filling the form, to apply a command to the entire form at once.

5. *Form-Specific Commands.* Different forms being used for different ends, it is desirable to permit application-builders to embed commands within FILLIN that may be applied only to specific forms.

All these problems are rooted in context switching. Other tools can implement solutions, at the price of constant context switches. Arguably, certain functions in a forms-management system should *not* be performed in a form-oriented style: in the SPS forms management system, such operations as assigning permissions to folders of forms, deleting a set of forms, and listing the names of all forms (each form in the SPS forms management system has an associated name that is not necessarily in one of its data fields) are implemented, quite reasonably, through command-oriented interaction. However, in certain contexts a form-oriented interface is preferable, and switches from the context should be delayed until a command-oriented interface is needed.

Although existing DBMS' and their form-oriented interfaces would have solved the problems for certain tools, a form-oriented interface was required for software that could not use a DBMS. We accordingly studied the form-oriented abstraction to determine the additional characteristics necessary for creating reusable software that can perform the above functions. Such a package could, as discussed in Section 2, be used with or without a DBMS, thereby satisfying everyone's needs. The next section explains our approach.

4. AN EXTENDED FORM ABSTRACTION

Several factors guided the enhancements to FILLIN. All specifications, design and implementation decisions were viewed in light of the following:

1. *Interface Consistency.* FILLIN's uniform interface had to be preserved across all tools that relied on it. Tools with extendible command sets, such as the EMACS text editor

[13], are powerful but notorious for the problems they cause when one user attempts to try another's remapped command sequences.

2. *Tool Integration.* Much of the power in building tools on Unix stems from easy access to existing tools. We wished to expand, not reduce, the contexts where FILLIN could be used.

3. *Keep the User Interface Separate from Tool Functionality.* FILLIN designers have, over the years, resisted many attempts to add features that are not related to user interface. For example, FILLIN forms are linear, but hierarchically-structured information is sometimes advantageous. However, we have found it difficult to implement such structure without incorporating the nuances of a particular application that demands it. In the interests of keeping the FILLIN and the interface it presents simple—both to the user and the tool-builder—we have avoided including such features.

4. *Provide Similar Capabilities at the Shell and C Levels.* As discussed in Section 2, FILLIN's software development paradigm requires that any function in the FILLIN library should be available to shell-level writers as well, and preferably simpler to write (albeit slower to execute).

5. *Create Reusable Software.* Extensions had to be done in a way that preserved FILLIN's role as a reusable tool, so that the concept of a form could appear in many tools. Little duplicate software exists in the tools that use FILLIN (for those portions concerned with user interfaces, at least), an indication that FILLIN offers good reusability.

For each of the requirements, it was necessary to consider what operations should be provided to implement them. We sought common solutions, to keep the tool simple. Two schemes were sufficient to solve all the requirements: one for form validation, and another for the other four. We now discuss each.

4.1 Form Validation

A form instance is *valid* if it meets a predefined set of conditions. Conditions may apply to any instance of the form type, or may be application-dependent. Examples of conditions include syntactic checks on data field values, requirements that certain fields be filled (*e.g.*, in the reference form of Figure 1, a title, document type, location and number of copies must be given; all other values are optional), relationships between data field values within the form instance (*e.g.*, a book must come from a publishing company, not a university), and relationships between data field values and values outside the form instance (*e.g.*, the location must be in an existing folder or a new folder that fits into some pre-established numbering scheme, perhaps located in a database).

The diversity of these conditions illustrates the difficulty of implementing form validation. The ideal solution is a language that can express constraints such as the above. Unfortunately, the few examples given above show the inherent complexity involved. Our experience has shown that a candidate language, to be useful, needs the power of a small programming language, and must be able to communicate with a DBMS, file system, and host environment. Implementing such a language would be a huge task, and would create software dependent on a particular DBMS. Furthermore, the language would probably duplicate features of a DBMS, and would certainly duplicate features of a programming language; therefore, from the standpoint of reusing software, designing and implementing such a language is a poor approach. Even implementing the language through a DBMS' integrity system has its drawbacks; such a system would be easy to write, but many DBMS integrity systems are incomplete. Moreover, one can always conceive of an integrity constraint that cannot be satisfied by a DBMS' facilities because it involves data outside the database (*e.g.*, the form is invalid if a particular user is logged on, but log records are not stored in the database).

A better approach is to express validation conditions in the application languages used to write the tools that invoke FILLIN—in this case, **C** and the Unix shell. This reuses existing software, including important aids such as interpreters and debuggers. Such tools already exist on Unix for C and the shell. Moreover, one can still talk to a DBMS, or any other part of the host environment, in the usual way.

Another issue is when to validate user input data. There are two possibilities. First, each datum can be checked as it is entered. Second, an entire form may be validated *en masse* when a user indicates he or she has finished filling it. The latter seems friendlier; it lets users discover and correct errors without continual interruptions of error messages. Moreover, forms can legitimately be in a temporarily inconsistent state (*e.g.*, for the example with names given above, one should be able to enter the first name before the last). Therefore, we choose to validate only complete forms.

Validation proceeds as follows. FILLIN classifies a form, in a manner explained below, as either valid or invalid. If it is valid, then FILLIN exits. If the form is invalid, the validation process should display the reason why. If the reason can be explained tersely, it can be made to fit on a single line at the bottom of the screen. Otherwise, it may disrupt the display (by scrolling the screen or placing text at arbitrary positions, for instance); if so, the form image must be redrawn. Three possible "exit statuses" are thus defined: valid, invalid, and invalid needing redisplay. Validation cannot change the form's data fields; it may only determine the correctness of the form.

Validation may be done with equal facility at the shell or C interface level. In C, the tool-builder gives, as an argument to a FILLIN routine, the name of a routine to call when the user issues the exit command. The routine is called just before switching from full-screen to command-oriented context, and is passed as an argument the current form instance. Through projection operations provided by FILLIN, the validation routine checks the values in the form data fields and either returns a value indicating that the form is valid (in which case FILLIN exits) or displays an error message and returns a value indicating that the form is invalid, and must not be accepted in its current state.

At the shell level, the tool-builder provides not a routine, but a shell command. The Unix shell is a self-contained programming language; in fact, its string-manipulation facilities are superior to those found in most conventional programming languages, and are excellent for the types of comparisons that validation often involves. This command has access to all the data field values through a mechanism that will be explained shortly. If the command exits with a status of zero, the form is presumed valid; if it exits with a non-zero status, the form is presumed invalid, and FILLIN refuses to exit. The following validation script requires that a last name be given, displaying an error message if one is not:

```
if [ "`fieldvalue ftf {} lastname`" != "" ] ; then exit 0 ; fi
echo You must give a last name.
exit 1
```

This works as follows. The above is given as an argument to FILLIN when it is invoked, along with the form template file stored in a file named **ftf**, which describes the form instance's data content. When the user types the quit command, the above is executed, after first replacing all instances of "{}" with the name of a form data file containing the data currently in the form. **fieldvalue** is a command that prints, on the standard output, the value of the named data field in the form template file—here, "lastname". The Unix shell notation `...` means to execute the command between the quotes and substitute its output for the quoted text. Therefore, if the data field is filled, a non-empty string *x* will be substituted, and the test "`"x" != ""`" will succeed, causing the command to exit with zero status. Otherwise, **echo** prints an error message, and the command exits with a non-zero status.

Form validation applies only to the form instance. Applications could reference other data areas if needed. For example, one tool that uses FILLIN and INGRES must determine if a name that is entered in a form instance exists in a relation. It uses the INGRES shell command **printr** to retrieve the pertinent relation, and the Unix utility **grep** to search for the name. This technique is application-specific, and the form does not depend on it in any way.

4.2 Form Scanning, Querying, and Command Extension

To achieve a reusable form-oriented interface that can satisfy the other requirements, we focus on context switches. Consider scanning: a user should be able to scan through a set of electronic forms with the ease of flipping through a stack of paper forms. Desirable operations, then, include "flip to next form," "flip to previous form," and "flip to a random point in the set of forms" (useful for a set of lexicographically-ordered forms).

Querying uses a similar style. The common technique in QBE-oriented DB interfaces is to present the user with a blank form image (or relation schema). The user then fills in the data areas corresponding to desired values, and instructs the system to retrieve the set of tuples corresponding to the filled-in values. The tuples are retrieved and shown to the user.

Context is important in these tasks. The SPS software required many context switches to perform the above functions, and, while better suited to the particular level of abstraction for the problem than the equivalent DBMS software, was not as convenient to use. For instance, scanning required generating a synopsis of each form (expressed as a few key fields) in a command-oriented style. To review all data fields of a set of forms involved a separate Unix shell command for each form. Thus there was a continual switch between command-oriented and form-oriented context. The switches had no functional purpose other than that mandated by the structure of the software.

The above discussion tacitly assumes that FILLIN commands are available to invoke the operations. By default, they are not. Such commands are application-dependent; scanning commands are only necessary for applications dealing with a set of forms. Since many FILLIN applications use only one form at a time (*e.g.*, in the SPS calendar management tool a user gives one "create" command per appointment to be scheduled) the operations for manipulating multiple forms would sometimes be semantically meaningless, and such operations introduce potential user confusion. Also, there is an important user interface distinction between the commands "scan next tuple in database" and "scan next appointment in today's calendar" that a generic "scan next" command does not express. For these reasons, we sought to avoid built-in knowledge of what it means to work with a set of forms.

Our solution therefore uses command extension as the basis for implementing all the above operations. All the above operations, in addition to many others, may be introduced through command extension in a manner explained presently. The technique is extremely powerful from the tool builder's point of view. There is one problem: much of FILLIN's ease of use stems from the simple interface, consistent across all applications. The basic interface remains unchanged, but added commands require knowledge of what commands relate to what tool; moreover, tool builders can use different key sequences to invoke similar commands across tools, or give the same key sequence different meanings across tools. Projects must impose standards to correct this, a disadvantage as by default FILLIN automatically provides an identical interface. However, we have not as yet found this to be troublesome.

An extended command consists of three parts: a key sequence to invoke it, an action to be performed when the key is depressed, and a help message describing the command. To keep the interface simple, the key sequence is always two characters, the first of which is the ASCII key **ESC**, and the second of which is any ASCII character *c* such that **ESC-*c*** is not already a FILLIN command. A tool may have as many commands as there are keys.

As with validation, an extended command may be specified at either the **C** level or the shell level, where the action is a **C** subroutine or a Unix shell command, respectively. Pressing the two-key sequence causes the action. It is given the current form instance as a parameter.

An action can have many effects (including none). It may communicate with the user, if more information is needed to complete the action. It may also communicate with other parts of the system, sending the current values in the form instance to whatever data storage mechanism is being used. Conversely, it can fetch new data field values, altering the form image.

Two types of support are provided to implement the above. First, operators are available to effect the communication with a user without switching from the form-oriented context; this is done by using the screen's bottom line as a data entry window. A one-line dialogue is usually sufficient, but an action may use any type desired for more complex interaction. Usually, this corrupts the screen, requiring a special exit status (see below).

Second, as with validation, the action must provide an exit status that indicates the effect. Four exit statuses are defined:

1. An action may have no effect.
2. An action may change data field values, requiring redisplay of the form image with the new values.
3. An action may have no effect on the data field values but may alter the form image, requiring redisplay of the original form image.
4. An action may change both data field values and the form image, requiring redisplay of the form image with the new values.

FILLIN provides four corresponding status values that an action may return.

The following example illustrates these concepts. Suppose we are developing a tool to scan a set of forms of the type shown in Figure 1. This might require the following commands:

- Determine the set of forms (perhaps identifying some relation in a database, mapping tuples to form instances) and retrieve the first one in the set, placing a form image containing it on the screen.

- Retrieve the next form instance in the set, and replace the current form image with the new form instance; or, if the end of the set has been reached, issue an appropriate diagnostic.

The "action" for the first command is to change the form data fields, which requires a change to the form image; therefore, the status is "Fields but not screen changed." This would be implemented by the following steps:

a. Locate the set of forms to be scanned.

b. Retrieve the first form from the data store.

c. Create a form instance whose data field values are taken from the retrieved form.

d. Exit, with status indicating that the data fields were altered.

The first two steps are purely database operations, and involve no FILLIN primitives. The last two steps cause FILLIN to create the appropriate form image.

For the second command, if more forms are in the set, the action again changes the form data fields, requiring the same status. The algorithm for implementing this is similar to the above, except that step a) is unnecessary and step b) retrieves the next rather than the first data form. If, on the other hand, no forms remain, a diagnostic message should be printed. The change to the screen might require that the original form image be redrawn.

The reader should note that again only the interface is being considered. The "action" may obtain the data from any source, and may place it in any source. A relational database has been used for simplicity, but tools using FILLIN have used many data management schemes, from INGRES to the standard Unix directory structure.

4.3 Experience

The extended form abstraction has been used in several new tools. One, an INGRES database browser, is discussed below. Several are extensions to existing tool environments that already use FILLIN. For example, the reference tool had no easy way to modify existing references. Forms had to be selected through command-oriented context, then modified through form-oriented context. Using extended commands, we implemented a QBF-style modification tool for selecting and editing a reference.

Validation has also proven useful. An annoying feature of the original reference tool was that it could not verify that a title was given until after FILLIN had exited. Switches occurred from form-oriented context, to command-oriented, then back to form-oriented, each time the title was missing. While the user did not have to type any commands, the form needed to be redisplayed for each switch to form-oriented context, which looked unnatural. The new reference tool validates using a script similar to the one on page 11, without requiring the screen to be redrawn.

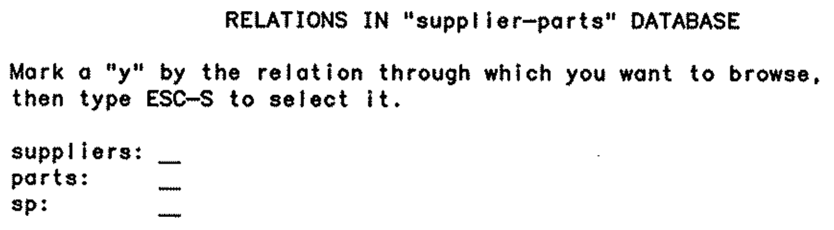
We conclude this section by discussing, briefly, the INGRES browsing tool. It is implemented using algorithms similar to those given at the end of the last section. The tool consists of nine shell scripts, averaging 25 lines. The tool can browse through any relation in a given database. It can scan in either direction, either sequentially or searching for the next (or previous) tuple whose values match a given pattern.

With one exception, all interaction is done through forms. The exception is in selecting the pattern when searching for the next form, which is done in a line-oriented style (but without disturbing the form image, so no context switch occurs). Commands are implemented as selection in forms, essentially a menu-oriented style. For example, a user selects a particular relation through which to browse by using a form such as that shown in Figure 2. Moving to the appropriate data field, typing a "y", and then pressing the extended command **ESC-S**, switches one to a form for browsing through a relation. Such a form is shown in Figure 3.

The browser was a prototype tool, to experiment with the concepts discussed in this paper. implementation time was about two hours. Even so, it is fairly robust. Since FILLIN handles most of the user interface, users must enter data in a form-oriented style, which structures the data and makes error detection straightforward.

5. CONCLUSIONS

This paper has discussed reusable software for applications hat can benefit from form-oriented user interfaces. It has covered the original concept of form abstraction in FILLIN, and has shown what was needed to extend the concept into something that much more



```
                RELATIONS IN "supplier-parts" DATABASE

Mark a "y" by the relation through which you want to browse,
then type ESC-S to select it.

suppliers: __
parts:     __
sp:        __
```

Figure 2: Selecting a Relation for Browsing

closely models the better features of paper forms. The extensions have allowed solutions to be constructed for the problems we encountered.

Many of the solutions found in FILLIN are taken from those implemented in database management systems, whose interface styles have proven popular and useful. The important contribution of this paper is the isolation of these ideas: the consideration of forms independent from a DBMS or any other facet of a programming environment. Therefore, they can be reused by any application, database or otherwise. Form-oriented data can be treated not as a relation, but as a form, which is often a better abstraction for an application.

We have attempted to construct general-purpose solutions, avoiding task-specific ones when possible. For example, the most common type of validation is to verify that a data field has been assigned a value. Some people have suggested adding an option to FILLIN that requires users to fill certain fields before exiting. The option would be useful but difficult to implement such that a tool's abstract view of data is preserved. Generic error messages such as "you must fill field *x*" do not tell a user much. The example on page 11 requires a filled field and can report any message desired. In any event, filled fields are only one type of validation that may be needed.

FILLIN's model has been paper forms, and its user interface is derived from the operations one would expect to perform on them. A reviewer once suggested associating actions with data fields, and automatically invoking the action when a data fields are filled. This construct clearly has no analogy with paper forms, which is of course not a sufficient rea-

```
            RELATION "suppliers" IN "supplier-parts"
        100 tuples.

        Type ESC-N to select the next tuple,
             ESC-P to select the previous one,
             ESC-F to search forward,
             ESC-B to search backward.

Attribute   Type  Value
sno         c3    S1
sname       c10   Smith
status      i2    20
city        c15   New York
```

Figure 3: Browsing through a Relation

son to reject it. So far, however, we have not found a situation where it is absolutely necessary; usually it is needed for validation of a data field, which we feel is currently handled in an acceptable manner. In other situations, extended commands can often perform the same task. One situation where it would be better than what currently exists in FILLIN is in the INGRES browser. Selecting a relation could be done by placing an "y" in the appropriate data field, which is easier than the current scenario of entering a "y" and then typing **ESC-S**.

We can conclude, then, that an automated form-oriented interface has more to offer over a paper form than just the ease of data storage and retrieval. Productivity gains through form-oriented interfaces will come about through re-thinking the concept of a form, and determining what interface features an automated form can lend that a paper form cannot. Through FILLIN, we are currently studying this question.

REFERENCES

[1] B. Boehm, *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[2] 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.

[3] *IDM/500 Reference Manual*, Britton-Lee, Inc., Los Gatos, CA, 1981.

[4] B. Cox, *Object Oriented Programming: An Evolutionary Approach*, Addison Wesley, Reading, MA, 1986.

[5] P. Hayes and P. Szekely, "Graceful Interaction through the COUSIN Command Interface," *International Journal of Man-Machine Studies 19*, 3 (Sep. 1983), pp. 285-305.

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

[7] R. Manis and M. Meyer, *The Unix Shell Programming Language*, Howard W. Sams & Co., Indianapolis, IN, 1986.

[8] 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.

[9] Y. Matsumoto, "Some Experience in Promoting Reusable Software: Presentation in Higher Abstract Levels," *IEEE Trans. on Software Eng. SE-10*, 5 (Sep. 1984), pp. 502-512.

[10] M. Penedo and S. Wartik, "Reusable Tools for Software Engineering Environments," *Proc. AIAA'85*, Long Beach, CA, Oct. 1985.

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

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

[13] R. Stallman, "EMACS, the Extensible, Customizable Self-Documenting Display Editor," *Proc. ACM SIGPLAN Notices SIGOA Symposium on Text Manipulation*, Portland, OR, June 1981, pp. 147-156.

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

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

[16] S. Wartik and M. Penedo, "Form-Oriented Software Development," *IEEE Software 3*, 2 (Mar. 1986), pp. 61-69.

[17] A. Wasserman, "The Unified Support Environment: Tool Support for the User Software Engineering Methodology," *Proc. Softfair*, Washington, D.C., June 1983, pp. 145-153.

[18] M. Zloof, "Query-by-Example: A Data Base Language," *IBM Systems Journal 16*, 4 (1977), pp. 324-343.