Comparing a Form-Based and a Language-Based User Interface for Instructing a Mail Program

Robin Jeffries Jarrett Rosenberg

Hewlett-Packard Laboratories
Palo Alto, CA 94304

Abstract

In the domain of interaction languages, forms have been found to be of value in allowing users, especially non-programmers, to specify objects and operations with a minimum of training, time, and errors. Most of that research, however, has been on the use of data base query languages. The present research found that in a procedural task of specifying mail filtering instructions, non-programmers using a form were as fast as programmers using a procedural language, although programmers using the form were faster still.

Résumé

Il est bien connu que l'utilisation de formulaires peut aider les usagers, spécialement les non-programmeurs, à spécifier les opérations qu'ils veulent faire exécuter à l'ordinateur dans un minimum de temps, avec un minimum d'erreurs et sans requérir un long apprentissage. Les études antérieures ont surtout porté sur l'interrogation de bases de données. La présente recherche montre que dans une tâche qui consiste à définir des filtres pour sélectionner un sous-ensemble de messages électroniques, les non-programmeurs remplissant des formulaires sont aussi rapides que des programmeurs exécutant la même tâche à l'aide d'un langage procédural. En fait, les programmeurs eux-mêmes sont plus rapides en utilisant le formulaire que la langage procédural.

Keywords: forms, design trade-offs, interaction styles

Introduction

As computers become both more widespread and more powerful, there is a growing demand to make this increased functionality more easily available to programmers and non-programmers alike. Form-based interaction languages have become an attractive means for user interaction with systems ([2], [5], [10], [11]). We define a form as a user interface which presents the user with a template to be filled out. The template may be dynamic, allowing the user to add or subtract instances of fields from the visible form. With a system that supports a pointing device and pop-up menus, fields can have an associated menu of choices that represent the valid data formats. Most data entry is then done by selecting from the menu. The user is only required to type the data items that cannot be reduced to menu selections; the form clearly delineates where such data entry is required and provides hints as to the valid format of that data.

Such forms are a congenial interface for many reasons:

- they are accessible to non-programmers because of the familiar metaphor of filling out a form. In particular, many office tasks map well onto the form-filling model.
- by allowing the user to call up the currently available options on demand, the range of possibilities covered by the form is made apparent, fewer demands are made on the user's memory, and the form is made easy to learn.
- by constraining the options that can be entered at any point, they forestall many syntactic errors, as well as potentially meaningful constructs that are beyond the capabilities of the system.
- by reducing typing, forms allow users to complete the task more quickly and prevent them from making many minor syntactic errors.
- they allow the user to select whichever manner of entry is faster: seeing and pointing or remembering and typing.
- they provide the user with a template that represents a prototypic solution to a class of tasks; this can potentially lessen the amount of effort needed to construct a solution.

We propose that forms may be a useful means for non-programmers to specify procedural instructions. Besides their other advantages, forms could provide a more declarative style of task construction. Non-programming computer users find it difficult to specify instructions as procedures, and it has been hypothesized that a declarative specification language can minimize the errors that vague, incomplete or erroneous procedure descriptions produce. (Miller, 1980)

Form-based languages have been used to specify instructions in a few existing systems. One notable example is the Query By Example (QBE) system of Zloof [10]. This is a data base query language where one fills out a table that mirrors the representation of the underlying data base. Entries in the table can specify data values, constraints, or actions. A study of the usability of QBE [8] showed that it was easy to learn and that subjects were able to formulate a larger proportion of correct queries with it than they could with a more traditional query language (SQL). Other advantages were that people made substantially fewer clerical errors and that it was well remembered even after a six-week interval. In another study comparing QBE, a traditional query language, and an algebraic language [1], error rates were approximately equal across groups, but subjects wrote queries much more quickly in QBE. (It took more than three times as long to write queries in the algebraic language.) QBE has been extended to handle a more general class of instructions as Office-procedures By Example (OBE [11]). which includes data base access, mail, reports, word processing etc., showing that the form model of instruction can encompass a large class of tasks. A similar forms-based application development environment is the FORMAL system [5], which can handle an even broader class of tasks.

However, the most common interaction languages for specifying procedures are formal languages, which typically have none of the advantages listed above for forms, but have instead greater generality. Nevertheless, the relative ease of use of forms and language-based interfaces may vary for tasks of different complexity, and for different types of users. Since a form interface is designed to facilitate a particular class of tasks, users should find it easy to construct tasks that are relatively close to the form's prototypic one. However, once a task deviates too much from this ideal, it should become more difficult to transform it into the representation required by the form. A language, on the other hand, should be less sensitive to such variation; a much broader range of tasks should be consistent with its representation. Furthermore, insofar as forms encourage a more declarative style of task solution, this may make it less easy to solve complex procedural problems. Welty and Stemple [9] compared two data base query languages that varied primarily in their proceduralness (though neither was form-based) and found that non-programmers made more errors on complex queries using the less procedural language.

Equally of interest are the relative advantages of forms and languages for programmers and non-programmers. Non-programmers should clearly find forms easier to use, based on the well-known problems they have with many programming-related constructs [6]. This should be particularly true in applications that will be used intermittently: users will be unwilling to undergo extensive training to learn to use them, and are more likely to forget the detailed aspects of the language between uses. Programmers, on the other hand, may be more comfortable, and thus more efficient, in a language-based environment. Furthermore, research in the domain of data-base query languages has consistently shown that programmers do better on almost all measures of task performance than non-programmers (e.g., learning, task completion time, errors) [7]. The difficulty appears to be in translating from the way people think naturally about queries into the logical structure required by the data base; therefore it is reasonable to expect comparable results when specifying instructions, since the logical structure of the task to be performed is an important component. Thus a form would be a particularly useful interface if it could enable non-programmers to achieve a similar level of performance as programmers.

An experiment

As a way of investigating some of these issues, we examined the suitability of a forms-based interface for specifying procedural instructions. We were interested in whether the forms interface was more effective than a language-based one for either programmers or non-programmers, and whether the complexity of the task affected which interface was more useful. To this end we constructed two simple and (hopefully) natural interfaces, one form-based and the other a language, and had both programmers and non-programmers use them to specify a variety of mail filtering procedures in a hypothetical mail system.

Subjects. There were 18 subjects, half of them male, all in their twenties and thirties, and all employees of Hewlett-Packard Laboratories. Twelve of the subjects were professional Lisp programmers accustomed to the same programming environment, the HP Common Lisp Development Environment, which is built on top of an EMACS-like editor and does not make extensive use of mouse-oriented or graphical interaction modes. The remaining six subjects (five females; one male) were selected from among the secretarial and technical staff at the same location. They were all experienced computer users, but they were not programmers. They used a variety of computing environments in their professional lives. Two of the non-programmers were familiar with mouse-oriented, graphical interaction modes; the others were not. All of the subjects were familiar with one or more computer mail systems.

Tasks. We selected filtering mail messages [3] as a domain that would be familiar to most computer users and for which it should be possible to write a variety of procedural instructions of varying complexity. Two interfaces were developed, one form-based and one language-based, that allowed users to construct rules for manipulating their mail messages in a hypothetical mail system.

The 16 problems were constructed so that half of them were logically simple and half logically complex (examples are given in Figure 1). The complex problems required more than one form in the forms condition and complex parenthesization in the language condition. We hypothesized that subjects using the form might find the complex tasks relatively more difficult, since they had to, in effect, treat them as two distinct tasks.

We devised two very different interfaces to the hypothetical mail filtering system. The *form-based interface* displayed a template in an IF–THEN format (see Figures 2 and 3a). Subjects could specify the pattern to be matched

Simple Tasks

"Notify me when new mail about a meeting or appointment arrives."

"Every Saturday at 3:00 AM, move all unread messages which are answered to the Pending folder."

"Remail all messages from Bill about skiing sent between Jan 1 and Jan 30 to Ann and then delete them."

Complex Tasks

"Delete messages from Bob and messages more than 300 lines long."

"Notify me when new mail from Meg or about a meeting, lunch, or party arrives."

"Every day at 4:00 PM, notify me if I have tagged mail, or mail bigger than 20 lines which is unread."

Figure 1. Examples of Experimental Tasks.

(boolean combinations of various message components), when the task should be carried out (tasks could be set up to run once, at regular intervals, or when new mail arrived), and the actions to be performed (we included most of the actions available in a typical mail system.) By clicking on any field with the mouse, subjects could bring up a menu of choices to be inserted into that field. Places where data (e.g., name of sender) needed to be inserted were indicated with angle brackets (see, for example, the `Msg body contains:` field in Figure 2). The mouse or the arrow keys could be used to move the cursor to different parts of the form.

The form assumes that the various fields and subfields are conjoined with implicit ANDs. To override this assumption, separate forms must be used. Data within a field or subfield can be connected with AND, OR, or NOT; thus, for example, one could refer to mail that is `Read AND NOT Answered OR Tagged`. Unfilled fields are assumed to match anything. An example of a filled out form is given in Figure 3a.

For the purposes of the experiment softkeys were provided to bring up a new form within the same task, bring up a copy of the current form, or to reset the current form. Only one form was visible at a time, and it was not possible to return to previous forms.

The interface we implemented did not do any syntactic checking of the forms users created. The interface limited the ways in which items could be composed, but it was still possible to create syntactically invalid forms (e.g., `To: Joe AND`) that would not be detected by the interface.

```
IF
  When: Now
  Header contains:
  Msg body contains: <string>
  Folder name is: <string>
  Msg length is:
  Msg is:
  Date sent is:
THEN
```

Figure 2. A Blank Form.

The *language-based interface* provided essentially the same functionality as the form-based interface in a Pascal-like syntax. (See Figure 3b.) As an attempt to optimize the language for this task domain, the primitives were made specific to the mail filtering task and as mnemonic as possible. The environment we provided did no syntactic or semantic checking of the programs subjects wrote.

Procedure. Subjects were tested individually. All the non-programmer subjects were assigned to the forms condition; we felt that non-programmers would be unable to do the tasks in the language condition without substantially more training than we gave subjects in this experiment and chose not to include such a group. We did, however, ask two of the non-programmers to do some of the tasks using the language interface, in order to get a rough estimate of how difficult the interface would be for non-programmers to use. The programmer subjects were randomly assigned to the form or language condition, and did not know that they would be returning later to be in the other condition. They returned 12 to 16 weeks later to do the same tasks using the other interface.

The experimenter described to subjects the mail filtering task and the interface (form or language) they would use to write instructions. Form subjects were shown a worked-out example, and the experimenter demonstrated the use of the mouse, the pop-up menus, and the function keys. Language subjects had available a BNF description of the grammar at all times and were shown two worked-out examples. Then all subjects worked through four practice problems.

The problem to be solved was displayed at the top of a VDT screen, with a form or an editing buffer filling the rest of the screen. The form groups used the form interface developed especially for the experiment. The language group used their normal programming environment, which had been modified to save the contents of the editing buffer along with a time stamp at the end of each task. A similar data collection method was used for the form groups.

The practice problems consisted of three simple problems and one complex problem. During the practice tasks, the experimenter corrected any errors or misconceptions and pointed out to subjects any inefficient strategies they were using. After all four practice problems had been solved, the experimenter reviewed the major points and verified that the subject understood the nature of the task.

```
IF
  When:
    Every day at: 5:00 AM
  Header contains:
  Msg body contains: <string>
  Folder name is: In-Tray
  Msg length is:
  Msg is: Read AND NOT Tagged
  Date sent is:
    More than 2 days ago
THEN
  Move-to: Old-Mail
```

```
WHEN daily(5:00 AM)
   FOREACH msg IN "In-Tray"
      IF read(msg) AND
         NOT tagged(msg) AND
         date-sent(msg) > 2 days ago
      THEN
         move-to-folder(msg, "Old-Mail")
```

Figure 3. Solutions for a Long Simple Task. (a) Form, (b) Language.

Next the subject solved 16 experimental problems, eight simple and eight complex. The problems were grouped into four blocks of four. Two different orders were used, each on half of the subjects. The entire process took about an hour.

When the programmers returned for their second session about 3 months later, they were given the same tasks in the other order and the opposite interface. Afterwards, subjects were asked to compare the tasks used in the two sessions. All subjects perceived them as similar, but no one recognized that the exact same tasks were used in both sessions.

In order to provide an estimate of how the non-programmers would have performed using the language interface, we asked two of the non-programmers to attempt the language condition. Both of these subjects had had some exposure to programming. We used the same training materials as the programmers received, but the experimenter provided much more coaching. The subjects worked through only eight of the 16 problems.

Results. Since the 12 programmers' data constituted a within-subjects design, they were analyzed using a $2 \times 2 \times 8$ repeated measures ANOVA involving interface (form or language), complexity and problems, with problems nested within levels of complexity. The data from the six non-programmers were analyzed separately because they had only been in the forms condition.

The means and standard deviations for the various conditions are given in Table 1. The ANOVA for the programmers' data revealed a difference between the form interface and the language ($F_{(1,11)} = 47.44, MSE = 2227$, $p < .001$). There were also differences among the different problems, both for programmers ($F_{(14,154)} = 22.68, MSE = 767$, $p < .001$) and non-programmers ($F_{(15,75)} = 8.95, MSE = 4046, p < .001$). This variability among problems may be responsible for the lack of any detectable effect of task complexity on completion time. None of the other main effects or interactions were statistically significant.

The medians and interquartile ranges for each group's median completion times are shown in Figure 4. It can be seen that the programmers using the form were 24% faster than non-programmers using the form (86 sec. vs. 113 sec., $p < .001$ using the Wilcoxon test), while non-programmers using the form were not reliably different from the programmers using the language. The two non-programmers that attempted the task using the language took 2-3 times longer than when they used the form.

The number and types of errors were equally distributed over the different subjects and conditions, with subjects making some sort of error on roughly half of the problems. Subjects using the language, of course, made many minor syntactic errors which the form forstalled.

Discussion. These results clearly illustrate the basic advantage of form-based interaction: forms can not only provide much faster performance than using a formal procedural language, they can even allow non-programmers to equal the performance of programmers using such a language.

We also note that the advantages of forms are not limited to non-programmers. The programmers in our experiment also showed a significant speedup using forms. The programmers' familiarity with language-based interactions and with the specific editing environment used to compose the tasks did not overshadow the facilitation the form provides. However, as expected, the advantages of this sort of form seem to be greatest for non-programmers.

Table 1. Means and Standard Deviations for Programmers' Form and Language Completion Times, by Task Type.

	Lang.	Form	Combined
Simple	84.7	118.9	101.8
	(50.3)	(43.0)	(49.7)
Complex	88.8	120.9	104.9
	(42.4)	(42.8)	(45.5)
Combined	86.8	119.9	
	(46.5)	(42.8)	

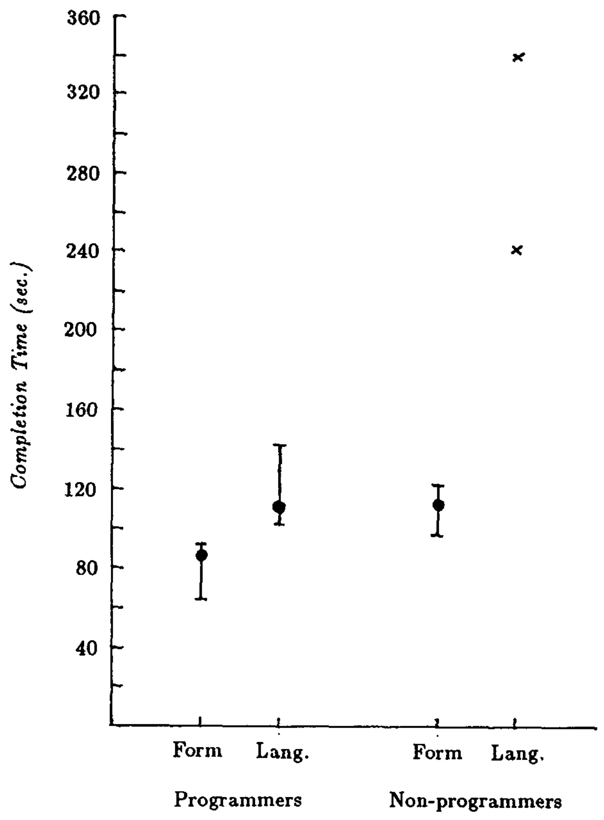

Figure 4. Medians and Interquartile Ranges for Each Group's Completion Times.

It should be noted that the clear advantages of forms shown in this study may be subject to other constraints. First, only independent tasks were considered here. In both the form-based and language-based interface, tasks could potentially interact in pernicious ways, and our results do not address which interface (if either) would be helpful in detecting or managing these interactions. Second, for an application of the sort considered here, which would be used intermittently, our subjects' level of skill was reasonably representative of the familiarity real users would have with the system; however, the advantages of forms may be reduced in situations where users have substantial practice with the interface.

Conclusions

To what factors can we attribute the advantages that accrued to forms in this study? Clearly, one way in which forms assist users is in the various bookkeeping functions they serve. The ability to select options from pop-up menus provides much of this assistance. People can be faster and more accurate because they are reminded of the available choices and they simply have less to type. Some of the form's advantage may be attributed to the better mapping it affords between the task as understood by the user and the representation required by the interface. We can speculate on various reasons for this—that forms allow people to express tasks declaratively, that they allow them to represent a task as a prototype plus deviations, or that they provide a familiar model for enumerating the solution. Further research is certainly needed to determine which, if any, of these reasons accounts for the form's superiority. Although we were not able to detect any interaction of problem complexity and interface type, this may have been due to the high variability among the tasks used; Welty and Stemple [9] found that as task complexity increased performance with their nonprocedural language decreased. This suggests that as task complexity (and hence problem-solving time) increases, the advantages of a form may become less important.

We must also consider how generalizable these results are to other examples of form and language interfaces, and different kinds of tasks. Our experiment sampled only a small part of this large space, and other research will be needed to confirm that these findings are a general property of form interfaces in many domains.

What do these results suggest for system designers? First, they support previous findings that forms are in general a useful interaction language, especially for non-programmers, because of their qualities of saving effort, reducing memory load, and structuring the user's interaction with the system in a way that is close the prototypical tasks the user wants to perform. Furthermore, they suggest that forms can allow non-programmers to equal programmers using a formal language. They also suggest that the advantages of forms are not limited to non-programmers; programmers may also find them a useful technique for simple task specifications, especially for casual and infrequent use.

Acknowledgments. We would like to thank Craig Zarmer and Mike Creech for their programming help, and our colleagues for being willing subjects. Special thanks go to Jean Gascon for his helpful comments and assistance with the data analysis.

References

1. Greenblatt, D., and J. Waxman, A study of three database query languages. in B. Shneiderman, ed., *Databases: Improving Usability and Responsiveness.* New York: Academic Press. 1978.

2. Hayes, P., and P. Szekely, Graceful interaction through the COUSIN command interface. *Int'l. J. Man-Machine Studies.* 19, 1983.

3. Malone, T., Grant, K., and F. Turbak, The information lens: an intelligent system for information sharing in organizations. in *Proc. CHI '86 Human Factors in Computing Systems.* (Boston, April 1986). New York: ACM 1986.

4. Miller, L., Natural language programming: styles, strategies, and contrasts. IBM Watson Research Center Research Report RC 8687 Dec. 1980.

5. Shu, N. C. A forms-oriented and visual-directed application development system for non-programmers. in *IEEE Computer Society Workshop on Visual Languages*. Hiroshima, Japan: Computer Society Press, 1984.

6. Soloway, E., Ehrlich, K., Bonar, J., and J. Greenspan, What do novices know about programming? in A. Badre and B. Shneiderman, eds., *Directions in Human-Computer Interaction*. Norwood, N.J.: Ablex. 1982.

7. Thomas, J., Psychological issues in the design of database query languages. in M. Shaw and M. Coombs, eds., *Designing for Human-Computer Communication*. New York: Academic Press. 1983.

8. Thomas, J. and J. Gould, A psychological study of query by example. in *Proc. Nat. Comp. Conf.* Arlington, Va.: AFIPS Press. 1975.

9. Welty, C. and D. Stemple, Human factors comparison of a procedural and a nonprocedural query language. in *ACM Trans. Database Systems*. 6, 1981.

10. Zloof, M., Query by example. in *Proc. Nat. Comp. Conf.* Arlington, Va.: AFIPS Press. 1975.

11. Zloof, M., QBE/OBE: a language for office and business automation. *IEEE Computer*. May 1981.