Rules for the Design of End User Languages

U. Engelmann, H.P. Meinzer

German Cancer Research Center
Institute of med. and biol. Informatics

Heidelberg, FRG

Introduction

Tools for the implementation of fourth generation languages are now available (SCHA83). These tools facilitate the production of interactive systems. A language is defined by a formal grammar in which the semantic actions are embedded (MEI83).

This process of implementation yields software systems for unexperienced users like, in the medical field, doctors and nurses. The languages are problem oriented and built to avoid hardware or software restrictions. A user expresses his demands in his concepts and language and must not specify how a job is executed on the machine. The systems are always specialized for the solution of defined tasks. It is assumed that an increase of performance of 7:1 can be gained if compared to third generation software.

The creation of good interactive systems can be supported by a careful design. The authors develop systems for end ~~users~~ user since ten years and won a lot of experience which is presented here in the form of rules for the design of fourth generation languages.

The design of a language

There are basically only three ways of man-machine communication, the sequential dialog (question and answer), the menue technique (pick one of a set of commands) and the command languages. The third alternative is the method of choice as it is more powerful and flexible than the other methods of interaction (MEI83). The following ideas can support the design of command controlled systems.

- sets of commands

  Functionally correlated commands should be sampled as a well defined subset of the language. A collection of correlated commands in sets also supports the training of users. A user can concentrate on these subsets of his special interest and will not be confused by the others.

- active and parametric commands

  It is absolutely necessary that every command has only very few parameters, two is maximum, zero is best. This requires the possibility for the definition and update of defaults, which is done by the parametric commands. Active (or procedural) commands initialize and execute a task under the conditions defined by defaults. Every set of commands should be split in a list of its procedural and the associated parametric commands.

- information commands

  There are two groups of information commands. The first includes answers to questions like

  - which file am I working with?
  - what is the actual value of a default?
  - how did I proceed to the actual state of my session?

  The commands can be named STATUS, DEFAULT or HISTORY. Other information commands answer the questions

  - which commands are available?
  - what are the possible parameters for a certain command?
  - what does a command do?

  These commands could be HELP (or simply ?) or EXPLAIN. Another category of commands support system management functions. These parts of a language have a more general character and should deal with the following features:

  - extension of a language by a user without changing the grammar.
  - creation of files or procedures that collect a set of commands for the solution of a given problem. It must be possible to include parameters and control structures.
  - definition of synonyms and abbreviations of commands under control of a user.
  - concatenation of commands to permit more than one function per line. Examples are 'AND', semicolon ';' and comma ','. This allows the use of basically simple commands to form longer and rather 'natural'looking input lines.
  - lines of comments usually make no sense in an interactive command language. They are very helpful if procedures or batch-jobs are developed.
  - conversational and batch mode must use the same commands. In special applications like e.g. image processing some routines are very time consuming. It should be possible to execute them in batch mode.

Features of a language

Close control of user yields useful hints on the quality of a system. Norman (NOR83) studied and classified user errors into five groups:

- mode error
- description error

- consistence error
- capture error
- activation error

From this classification the following design rules are derived.

1. Reduce the number of modes.
   Ideally there are no modes. Users get very easily confused, this requires a lot of complicated explanations.

2. Always clearly indicate the actual mode.
   As it is not possible to use no modes (at least you need WAITING and RUNNING) it must be made obvious where you are. Bad systems force a user to use trial and error methods where he tries to recognize a mode by its (different) error messa-ges.

3. Avoid ambignity.
   An example can be the editor 'VI' (UNIX, Berkeley Release) where 'd' and 'D' and 'CTRL-d' have different meanings. More bad examples are to be found everywhere. Ambignity can be very harmful as unwanted but harmful decisions are possibly in-itialized.

4. Design a consistent language.
   Consistence errors occur if for example the sequence of parameters of a command is not obvious, or worse changing from command to command. A user is mislead by wrong analogies. If a new command is inserted into a language its consistence with the existing ones must be checked both in respect to its parameters and its name and function.

5. Support the users memory.
   A number of activation errors are based on people's short memory. An interupt of a session (e.g. by a telephone call) results in an unfinished process (out of sight is out of mind). Incomplete sequences of actions should be indicated, questions to missing answers should be repeated within a certain time.

6. Avoid prompting.
   Prompts are questions or remarks of the machine like
   - do you really want to erase this dataset?
   - this file already exists, select another name.

   These parts of a dialog are popular with users, especially with unexperienced ones. As the remarks are always the same the attention paid to them is reduced by the time and then they cause trouble again because users apply potentially dangerous standard answers to the standard questions. It is bad that the machine does not learn anything but constantly repeats the same message to the same error (MEI83). Especially disturbing are prompts if the language is used in a procedure or batch job. As the answer to a question is missing, the next command line is taken for the answer, of course not understandable to the machine. Worst case are repeated prompts, the whole process ends up in a mess.

7. Permit an UNDO

Even the finest program design can not make failsave systems, it can happen that a user applies a wrong command. In the case of a parametric command there is no problem as a second input corrects the first. If an unwanted action was executed real harm could have been initialized and in this case a function UNDO is very helpful. In an editor e.g. DELETE commands are candidates for a reverse action. A nice example is found in the LISA concept where a file is finally lost only at the end of a session. No physical destruction is executed but some kind of logic operation that can be reverted (LISA quotation).

8. Permit a software reset.

A common joke of computer people is the remark, if nothing else helps and you and the system lost control of the interaction then use the 'hardware reset'. This indicates a complete electric cut off of the power supply. This is not only not elegant and time consuming for a new start but also dangerous as incomplete action or open files can be the result. We found a command like RESET to be helpful. It reinitialises the dialog, all defaults are set to their standard values. The machine is put into a defined and well known state.

9. Use concepts of the end users.

The system must be as close to the concept and language of the users as possible. Training and use is facilitated a lot.

10. Take special care of error messages.

Every programmer can tell stories of misleading or not understandable error messages. Tests have shown that improved error messages can increase the system performance by factor two (BRO82, BRO83, SHN80) not to speak of the subjective acceptance of a system. There are three main aspects:

- syntax errors

  In an interactive environment the syntax errors are identified by the parser. The location of the error in the input line can be indicated and a suggestion for correct continuation can be presented. A standard message like "unknown command" is less useful.

- semantic errors

  Actions usually depend on their semantic context, e.g. a certain command only makes sense if another action has been executed before. If the interactive system has been defined by its grammar and the correlated semantic actions special error checking routines can be inserted at the semantic level. These error actions should be given priority before executing the others. On this level very sophisticated error detection can be implemented. An error and a possible solution can be described in much more detail as compared to the syntax errors.

- psychologic aspects

  The negative image of computers to a great deal stems from the negative experience of user in the case of errors. As interactive systems lead users to an experimental approach ("let us try and see what happens"). "The user learns weather a system is a friend or a foe, when he makes errors" (BRO83).

Shneiderman described a few suggestions to message design (SHN82). He asks for readable and understandable error messages, the efficiency of which should be tested thoroughly by end users under control of a system designer. All messages that urge a user or scold him are bad. Good message are positive in form and action. Negative words like INVALID, ILLEGAL, ERROR or INCORRECT should be avoided. An error should be indicated, and a possible solution explained in detail, A message must be in clear text, don't use error codes that have to be analysed with the help of a reference manual. A user should feel that HE controls the system and not vice versa.

While developing and maintaining large end user systems we found an error documentation feature very helpful. The number and location of both syntactic and semantic errors can identify the two deficiencies in the concept of a system. It is then possible to update the system, the instruction manuals, the error messages and the education at the right point.

References

[ACS84] Archer, J.E. jr., Conway, R., Schneider, F.B.: User Recovery and Reversal in Interactive Systems. ACM Transactions on Programming Languages and Systems 6, No 1 (1984) 1-19.

[BRO82] Brown, P.J.: My system gives excellent error messages - or does it? Software Practice and Experience 12, No. 1 (1982) 91-94.

[BRO83] Brown, P.J.: Error Messages: The neglected area of the man/machine interface? Communications of the ACM 26, No. 4 (1983) 246-249.

[ENG85] Engelmann, U., Meinzer, H.P.: Bessere Mensch/Maschine Schnittstellen durch mehr Beachtung des Benutzerfehlers. Angewandte Informatik 1985 (in print).

[MEI83] Meinzer, H.P.: Der Dialog zwischen Mensch und Maschine in der biologisch-medizinischen Forschung. Dissertation, Fakultät für Theoretische Medizin, Universität Heidelberg (1983).

[NOR83] Norman, D.A.: Design Rules Based on Analyses of Human Error. Communications of the ACM 26, No. 4 (1983) 254-258.

[SHN80] Shneiderman, B.: System message design: Guidelines and experimental results. In Badre, A., Shneiderman, B. (eds.): Directions in Human-Computer Interactions. Norwood, N.J.: Ablex Publishing Co. (1982).

[SHN82] Shneiderman, B.: Designing Computer System Messages. Communications of the ACM 25, No. 9 (1982) 610-611.

[TEI75] Teitelman, W.: INTERLISP Reference Manual. Xerox PARC, Palo Alto, Calif., Dec. 1975.

[VER78] Verhofstad, J.S.M.: Recovery techniques for Database Systems. Comput. Surv. 10, No. 2 (1978) 167-195.

[WIL83] Williams, G.: The LISA Computer System. Byte 2 (1983).