Criteria for a standard command language based on data abstraction

by DAVID BEECH
Hewlett-Packard Company
Palo Alto, California

ABSTRACT

A solution is offered to some fundamental problems that have thwarted previous efforts to develop a standard command language. The technical approach is based on the form of modularity provided by data abstraction, and this is introduced from the point of view of the end user, together with a discussion of the advantages and disadvantages that might be perceived at this level. This leads to the statement of a simple but stringent set of criteria for the inclusion of functional capabilities in a standard command language and the testing of various candidates against them. Some candidates are accepted and others rejected, resulting in an initial proposal for the scope of a standard command language that is small and simple enough to have a hope of success.
INTRODUCTION

Command languages are at a crucial stage of their development, with considerable pressure to define a standard command language, but a dearth of good proposals. Many users are becoming impatient for a uniform method of access as they are confronted with an ever wider variety of systems and heterogeneous networks, and they very reasonably hope that a standard command language will be a distinct improvement over existing command languages. Yet there have been committees working within the American National Standards Institute since 1969 without evident success. Other national groups have engaged in preliminary skirmishes, CODASYL has tried its hand, and there is even a danger that the ADA infantry will aim to persuade us that what is good for embedded military personnel is good for us too. Now the International Standards Organization has been called upon to find a way of bringing order out of chaos.

A dozen years with so little progress suggest that there are some fundamental problems that have not been properly addressed. It is the thesis of this paper that these problems can be attacked by means that are sound in theory and viable in practice. Some surgery is required—or, more precisely, the application of Ockham's razor, the philosophical principle of conceptual economy. The result could be a standard command language that would make a rational start to providing uniformity for the user and would offer a framework within which more widespread standardization could evolve.

Some of the fundamental problems besetting previous efforts have been:

1. A lack of criteria for placing any bounds on the potentially large set of commands that might be included as a defined part of the command language
2. A lack of consensus about the detailed semantics of system functions to be invoked by commands
3. The difficulty of making richness of system function comprehensible to the user
4. The temptation to make command languages too much like programming languages

Since some form of modularity holds promise as an approach to the solution of each of these problems, and data abstraction has been extensively developed as a means of achieving modularity in programming languages,¹⁻⁴ this is the conceptual tool I shall employ. The operations that can be performed on a system will be modularized, i.e., partitioned into sets of operations that can be performed on different types of object, such as particular types of database system or text editor. The command language will then provide for such operations to be invoked; but the definitions of their semantics will reside within the various object types, whose potential for standardization becomes a set of separable questions. These should be addressed by specialists in their functional areas, who could most effectively be organized into distinct standards committees within a modular framework.

This will lead to solutions to each of the problems listed earlier:

1. Criteria will be proposed which limit the command language itself to a handful of commands.
2. Semantic controversies will then be kept within the confines of particular types of object and need not always produce an outright winner; e.g., more than one type of database model may be offered.
3. Modularity helps the user by reducing arbitrary complexity to more comprehensible interactions within and between types of object.
4. The command language will be deprived of general-purpose programming capabilities; the implication is that programming languages must also be able to invoke the operations accessible from the command language.

USER VIEWPOINT

The data abstraction concept

An intuitive way of describing data abstraction is that data are pictured as residing in black boxes that conceal their representation. All that is known is the set of operations that may be applied to a particular type of black box, and the definition of the responses that will be returned (Figure 1). The responses may depend on previous operations, so one way of modeling this is to think of the black boxes as having states that may be changed by operations.

How does this affect the command language user's view of a system? First, consider users of a conventional command language. They issue sequences of commands to one large black box; e.g.,

```
logon beech
copy MCL MCL2
edit MCL2
compile MCL2
run
```

In this example the verbs are all distinct, since they are all being interpreted at the same level, as it were, by a single black box. This is the monolithic approach: the commands all belong to one language, symbolized by their being described...
in one massive manual. It is the degenerate case of data abstraction.

I must emphasize that, for the purposes of this paper, I shall work with this verb-and-operands model of the essential information in a command without attending to important, but secondary, questions of alternative representation via syntactic sugaring, special keys, menus, prompting, and so forth.

Some commands, such as "edit," probably put the user into a mode in which subsequent inputs are directed to the text editor rather than the command language processor, and these inputs may include a separate sublanguage of commands to be used at this deeper level until it is decided to return to the top level. Certainly I shall retain the concept that a command passes the user's input/output port to the operation being invoked, and I shall return to the topic of modes shortly. But this is still a restricted way of offering modularity, which is especially clumsy when issuing a single command to a particular component of a system, requiring three commands in all: to enter the mode, issue the useful command, and return from the mode.

Suppose we call a black box an object and conceive of our outermost system object as being populated by interior objects that communicate with each other by means of operations and responses (Figure 2). Then we would achieve full flexibility if we could immediately name both an object and a particular operation to be performed on it, say with a qualified name of the form obj.op. Commands with certain simple names like "logon" might then be acted upon directly by the command language processor object, while those such as "mymail.read" would merely relay the "read" operation to the object "mymail," which might be my electronic mail system. Definition of such a "read" command would then be the business of "mymail," not of the command language, and the name need only be unique among the operations of "mymail." It is as though the command language processor object offered a "run" operation, which acted on any composite name to relay the specified command to (and response from) the specified object.

Thus the visible difference to the command language user could be very slight, but even this much difference might be too great! If there were only one "read" operation available to (or normally used by) a particular user, why should it be necessary to qualify the name just for the sake of some principle of modularity? This is a valid complaint, and I shall take it as a requirement for the naming scheme later that it should be possible, via controlled defaults or synonyms, to use single words to represent composite command names.

We are now in a position to see how the designer of an object type may choose to offer its functions only within a special mode, or always by direct invocation from the command language (or from programs), or both ways. The modal approach would put only one operation, e.g., "mymail.start," into the definition of the object; this would then include in its semantics the possibility of a dialogue, including data inputs from the user that had the form of commands directed to the mailer in its private language. The direct approach would put the specific operations into the interface, and the two approaches could, if desired, be combined to allow equivalent function to be obtained by either means.

Instances and names

We envisage command languages as being merely users of operations that are defined and implemented in programming languages. (Note that these programming languages do not even have to be abstract type languages. They must just be able to implement call routines that provide the semantics desired of an abstract type.) An abstract type is used by creating one or more instances of the type and performing permitted operations on these instances.

We must consider here the important distinction between general languages of the abstract type (e.g., CLU, 1 ALPHARD, 2 PLAIN 3 ), with potentially multiple instances of a type; and "module" languages (e.g. MODULA-2 4 ), which are similar but permit at most one instance of a module. This affects the way that a language specifies the creation and naming of instances; and the naming, at least, is bound to be relevant to the command language user. With a module approach, there is no need to distinguish between the name of the module and the name of the instance, whereas with an abstract type a command must indicate the instance and not the type (e.g., "mymail" and not "mailer.type"). I prefer the abstract type approach as a more natural way of treating multiple instances than having multiple modules whose equivalence (apart from name) has to be determined by inspection. However, the difference to most command language users would normally be negligible, since they would just know what names to apply to the objects they wanted to use without worrying how they were derived.
Advantages and disadvantages

I have already touched on the ways in which this form of modularity can work to the advantage of the user: in making large systems more comprehensible by dividing them into manageable pieces, in encouraging well-specified interfaces, and in providing a route to a useful initial standard rather than none at all. Beyond this, it will allow for experience to be gained with other types of object before deciding whether they are ripe for standardization. This will include experimentation with alternative ways of doing similar things, an approach that is much harder to manage in a monolithic language.

It is time now to look for possible disadvantages. I have already stated that I intend to develop a naming scheme to make it possible to avoid the burden of name qualification. But, going below the syntax to the underlying modularization, what if the abstraction made by the system designer does not coincide with that which is most natural to the user? For example, commands to file, editor, formatter, and photo-composer types of object may appear to the user to be all operating on an object of type "document." The use of synonyms will provide a superficial solution here, and this may often be sufficient. The simplest method is to replace all name qualification by single command names, so that the modularity is no longer visible. The more ambitious method, that of introducing differently grouped name qualifications, could be just as easy as a matter of naming; but the transformation of the semantics to fit the new conceptualization could be a very difficult exercise. The net result is that one has to live to some extent with the structure of a given system in order to retain one's sanity. One can always do at least as well as with the monolithic description, and usually much better, but one cannot easily produce arbitrary reconceptualizations.

A slightly more discomfiting criticism of this form of abstraction is its inherent asymmetry. One object is essentially selected as the principal operand of an operation, and the others are passed to it in subsidiary roles. This is often a difficult exercise. The net result is that one has to live to some extent with the structure of a given system in order to retain one's sanity. One can always do at least as well as with the monolithic description, and usually much better, but one cannot easily produce arbitrary reconceptualizations.

Along a different line, a disadvantage of adopting an initial command language that is not all-inclusive is that "the good is the enemy of the best": it may delay or prevent the arrival of the universal command language. This objection had to be included for the pleasure of refuting it, since the history of efforts to date supports the view that the search for an impressively comprehensive command language is itself the principal agent of delay and appears capable of indefinitely delaying the production of anything whatsoever.

Finally, I shall surely be assailed for believing that it is sufficient for the command language to be able to perform operations on objects without also being able to construct procedures, to iterate and branch and declare its own variables, and generally to aspire to be a programming language. The defense here is that it is extremely difficult to design a good programming language, so the choice would be between a poor result and even longer delays; and that a new programming language is not needed for the purpose of putting logic around commands—existing programming languages are already extensively known and supported and can generally serve quite well. Improvements in language and implementation may be necessary in some cases for interactive use and convenient access to system function, but the payoff will then be enormous in terms of the avoidance of artificial discontinuities between what can be expressed in the command language and what in the programming language. One method of embedding system function in programming languages has been discussed in detail elsewhere.

Taking stock

The results so far are mildly encouraging in improving comprehensibility, quite strong in delegating areas of disagreement to particular abstract types and in establishing a boundary between command languages and programming languages, and far too successful (for some tastes) in limiting the function included in the command language per se. Everything apart from "run" could be delegated to the abstract types, and this approach has been successfully embodied in the Lilith machine. A command language standard with this single command in a suitable embodiment ought to be achievable, but it might be considered to miss an opportunity to introduce more widespread uniformity. Are there other operations which should not be left to designers of individual abstract types? If so, we should be prepared to admit them, and we accordingly propose slightly more generous criteria for inclusion of functional capability.

CRITERIA FOR INCLUSION

A particular functional capability should be considered for inclusion in a standard command language if and only if

1. It provides one of the following:
   a. The means for a user to begin or end a session in which commands are issued to a system
   b. A general means of invoking operations conceived as acting on abstract types of object whose semantics are not defined within the command language
   c. Action which it is desirable to define uniformly for all or most abstract types accessible from the command language

2. It is not already available in the command language, except perhaps with extreme inconvenience.

We proceed now to consider some candidates for inclusion. Invocation has to be present, and we shall not attempt to deal here with questions of its syntax or treatment of operand
types. Otherwise, apart from LOGON and LOGOFF, we shall be dealing with functionality which is common to multiple abstract types. Operations will be named in capitals where they correspond to potential explicit commands in a command language.

Possible ways of treating operations applicable to multiple types are

- A recursive subtype structure (cf. SIMULA), where the subtype inherits the operations of the parent type
- A restricted two-level structure with all specific types known to the command language nested within a general object type
- Separate abstract types, some of whose operations are implicitly involved in command language operations such as "run"

The last method is selected for its simplicity and adequacy for our purposes.

**LOGGING ON and OFF**

The question of the inclusion of some form of LOGON and LOGOFF has been prejudged in its favor by one of the criteria. The reason for this strong line is the importance of enabling users to get started in a simple, standard way rather than receiving a disastrous first impression of the complexity and idiosyncrasies of computer systems. Symmetrically, but of less importance, users should be able to take their leave without embarrassment.

However, it is necessary to cater to the spectrum from the one-person computer that does not require any identification of the user to the highly protected system that has elaborate authorization if necessary, and LOGOFF will allow for some implementation-defined cleanup.

LOGON interacts with access control (see below) in that before it is performed (in systems that require it) only HELP and LOGON are available. The semantics of LOGON allow it to give the user some initial access rights, obtained from a "user authorization" object. It also interacts with naming (see below) in establishing an initial name space for the user. LOGOFF reverts to the HELP or LOGON situation.

**Help**

A uniform method of seeking help should be available to command language users so that they do not need too much recursive help in using this facility. The HELP command addressed to the "command language processor" object will sometimes produce information directly related to that object, e.g., how to logon; or influenced by that object, e.g., what menu of commands is available to the user at a given point; or even customized by that object as a result of forming an intelligent model of the user during previous interactions. But much of the information desired will be about the operations that may be invoked on other abstract types, and these types will be required to include "help" operations which may be invoked by the command language processor in order to support its uniform helpfulness. This does not preclude the direct invocation of these "help" operations, or indeed the provision of other more specialized kinds of help for particular abstract types.

**Creation and destruction**

Creation and destruction are the most fundamental operations common to different object types, serving as a prerequisite for, or veto on, all other operations on an object. Do they satisfy our criteria? The answer is clearly "yes" if uniformity is interpreted loosely and "no" if it is taken strictly. The major problems are with the widely differing parameterization of creation for different object types and the varying semantics of destruction of types of objects that may be shared or may cause cascading destruction of other objects.

We propose that CREATE and DESTROY be grudgingly admitted, with uniformity in the names of the commands and in their interaction with the naming and access control of the objects they deal with; but that they allow for type-dependent parameters and semantics beyond this.

**Naming**

The "run" operation must be able to resolve names of operations on any abstract type and possibly names being passed as operands. Its semantics become very weak if this resolution is system-defined, and it would be helpful to users of different systems to have a common method of name qualification and aliasing to overcome the need for names to be unique systemwide (or networkwide). The earlier requirement for synonyms and default name qualification can be satisfied by this more general approach, which we accept as satisfying the criteria.

We therefore postulate an abstract type "Name Manager" (NM), with an operation "resolve" implicitly used by the command language "run" and an operation "name" used by abstract types when intended names are passed to them in a "create" operation. A particular instance of an NM is associated with a user in the "user authorization" object, and it provides the initial name environment after a successful LOGON. Otherwise, the NM behaves like any other object accessible from the command language, and explicit operations on it may be invoked in the normal way. The NM abstract type can be defined to correspond to a conventional directory structure, i.e., a tree with additional links to make it a network, since there seems to be reasonable consensus that this is a satisfactory model. The set of explicit operations could be RENAME, REMOVE, EQUATE, and EXPAND (applied to an incomplete, possibly ambiguous, name).

**Access control**

If a system supports any form of access control, it is desirable to apply it as a uniform scheme across all types of object for it to be effective; so this appears to be another...
strong prima facie candidate. With the data abstraction model, it is attractive to base access control on subsetting the permitted operations of the abstract types, with other refinements that we cannot pursue here. A similar approach is proposed to that used for naming: a “check access” operation is implicitly used by the command language “run,” and other operations on the “Access Control Manager” (ACM) are explicitly available for direct invocation.

A system that does not wish to support access control can appear to the user as one that allows all operations except the explicit operations on the ACM. It can then use its normal optimized implementation, rejecting the ACM operations but not checking anything else.

We therefore admit access control to the command language, with explicit operations that could be GRANT and REVOKE.

**Accounting**

Accounting is another function relevant to all types of object. It could be treated similarly to naming and access control, since command language actions implicitly use system resources, and explicit operations could be provided that were directed to the “accounting manager” object. However, the latter operations might not be very widely accessible, and there is little consensus about the best way of charging for consumption of resources. Therefore, the decision proposed here is that the semantics of the command language actions should allow for system-defined accounting to be performed and that the command language should treat any accounting manager as an ordinary object about which it has no special knowledge.

**Concurrent invocation**

In the absence of a consensus on good language for concurrent invocation, we propose an interim treatment until suitable forms can take their rightful place alongside synchronous invocation in the command language. Particular types of concurrency manager might exist in different systems, with operations like START an operation on some other object, ENQUIRE STATUS, and WAIT. An alternative approach, available with programming languages that offer concurrency and that support access to the desired operations on objects, is to invoke a processor of such a programming language and express the concurrency requirements in its language.

**CONCLUSION**

I have discussed the applicability of data abstraction to command languages and arrived at the view that it provides a good conceptual structure for issuing commands to operate on different types of object, leaving the initial definition and implementation of object types to fully-fledged programming languages.

The modularity inherent in this approach suggested some stringent criteria that could be applied to reduce to soluble proportions the problems of designing a potential standard command language. Applying these criteria, we admitted only the following commands:

- “Run” the operations on instances of abstract types
- LOGON and LOGOFF
- HELP
- CREATE and DESTROY
- Implicit “resolve” and “name”, and certain explicit commands, to a name manager
- Implicit “check access”, and certain explicit commands, to an access control manager

This is not to deny the possibility of or need for standardization of other system functions. On the contrary, it would encourage the timely and efficient consideration of such matters by experts working within a modular structure of separate committees, some of which already exist.

**ACKNOWLEDGMENTS**

Meetings of IFIP Working Group 2.7 have provided a helpful environment for the development of these ideas.

**REFERENCES**
