Office Information Systems, N. Naffah (ed.)

UNIFIED DIALOGUE MANAGEMENT IN THE CAROUSEL SYSTEM

Erik SANDEWALL

Software Systems Research Center
Linköping University, Linköping, Sweden

Abstract: The paper describes the design for a software environment which unifies the dialogue management support of a number of existing software tools, such as command-language handlers, operating-system shells, transition diagram interpreters, and forms management systems. The unification is accomplished without undue complexity: the resulting design is clean and inspectable.

A key notion in the design is the use of three orthogonal abstraction hierarchies, for interaction contexts, for interactive operations (roughly = commands), and for data types. These orthogonal hierarchies are used for obtaining multi-dimensional inheritance, i.e. mappings of several arguments, which may be stored in the system's data repository, and which inherit their values along the hierarchies represented in the arguments. Mappings with multi-dimensional inheritance are used in the invocation mechanism, i.e. for selecting the appropriate procedure to be executed in various situations. Currently used programming languages and software tools only use single-dimensional inheritance.

A working implementation called the CAROUSEL system, is available.

This research was supported by the Swedish Board of Technical Development under contract Dnr 80-3918.

1. Conventional approaches to dialogue management.

It is well known that the design and implementation of the dialogue with the user, is a substantial part of the implementation effort for software that is to be used interactively (see e.g. ref. BOE80 and WAS81). A large number of efforts have been made to provide relief. We distinguish the following approaches:

1. Software environments, i.e. software systems which communicate directly with the user, and which contain a number of application

programs, procedures, and/or modules which the user can invoke. This includes both command handlers for operating systems, and incremental programming systems.

In its simplest form, the command handler of an operating system merely allows the user to select one out of a number of different 'application programs'. The **UNIX** shell (ref. BOU78) is an example of a more powerful software environment, where the command language contains control primitives and a procedure mechanism. Interaction by command lines, which is natural in many applications, can then be implemented in the command language, rather than in the programming language that is used for the detailed processing. Dolotta and Mashey have reported that the command language often serves as the primary programming tool (ref. DOL80).

Incremental programming languages, such as **LISP** (ref. SAN78, WIN81, TEI78), **APL**, and **Smalltalk** (ref. BYT81) go one step further by completely integrating the programming language into the computing environment. LISP systems, in particular, contain a data repository which is used both for storing application data, and for storing procedure definitions, and all software management services such as editing, compiling, and generation management are done in that uniform software repository.

Both kind of software environments support one significant aspect of computer dialogue, namely the need for frequent revision, or in the words of Wasserman: *Our goal was to create a tool that would facilitate the "rapid prototyping" of the interactive dialogue in such a way that a user could interact with the "system" at an early phase of system development and so that the developer could easily modify the dialogue and present a revised interface.* (ref. WAS81).

2. Software tools for dialogue management: By software tools we mean software which perform specific functions, and which are intended to be used by the programmer in the engineering of an application system. The following are some groups of software tools for dialogue management:

2.1. Navigational tools, which maintain a topological model of a universe in which the user 'moves' in the course of the interaction. Most frequently the universe is a network (as in the following systems: **IDECS**: ref. HAG75, HAG80; **ZOG**: ref. ROB78, and its predecessor **PROMIS**: ref. SCH79; the **Transition Diagram Interpreter**: ref. WAS81; and the design proposed by Carlson and Metz: ref. CAR80). Nievergelt (ref. NIE79) gives a discussion of such systems. Systems for computer-aided instruction (e.g. **TUTOR**: ref. GHE75; **Coursewriter**: ref. MAR73) also maintain such models, but there the movement through the network is under the control of the system rather than the user. The universe may also be a tree (as in the **PRESTEL** system: ref. FOR79), or it may allow continuous movement for zooming in on significant data (**Spatial Data Management System**: ref. HER79).

In all these systems, the user navigates in a fixed universe. There are also systems where the universe *is* the user's information structure, particularly information management systems such as **PIE** (ref. GOL81) and **ED3** (ref. STR81).

2.2. Language tools, which simplify the implementation of a set of commands or a command language. This includes query languages for data bases, where different applications are realized as different information structures (e.g. using data base schemata), but the query language as such is a fixed resource. The **EUFID** system (ref. KAM78, BUR80) is a representative example of this large group of systems.

The query language approach is not sufficient when the application emphasizes active manipulation of data, rather than mere access to an existing data base. Office automation systems is one of the applications where active data manipulation plays a major role (see e.g. Morgan's survey article, MOR80). The interactive command language for an operating system has also that character. For such applications, one may use a syntax analysis tool (JOH78), but if the desired command language for the application merely consists of a command verb followed by one or a few arguments (or equivalent uses of function keys or of pointing into menues), one may prefer a tool which parses the command line in a standard way, and administrates the mapping from the command verbs to the procedures that define them. Facilities for macro expansion of commands and for 'help' style user support are more important than a rich syntax. The **INTRAC** system (ref. WIE79) is a good representative of this wide class of systems.

2.3. Layout tools, which support the presentation of data on the output medium. The major interactive layout tool is the forms management system, which supports presentation, entry, and editing of data in forms whose characteristics have been defined by the user. Forms management systems are available from the major manufacturers, and more sophisticated systems have been developed as parts of research projects, e.g. in the **Odyssey** system (ref. FIK80). In the **System for Business Automation** (ref. JONG80) and the **DIAL** system (ref. HAM80), forms management is available as a standard routine together with a procedural language and a data base. These systems as well as **LOIS** (ref. SAN79) and the system of Tsichritzis and Ladd (ref. LAD80) support forms flow between multiple user stations.

The classical report program generator was another kind of layout tool, but oriented towards the presentation of data on 'wallpaper' listings. The **Query-by-Example** system (ZLO75) has demonstrated that layout tools provide an alternative method for formulating queries, instead of using syntax-rich query languages.

Graphic systems (ref. NEW79) form another large group of layout tools, but they are not considered in the work reported here.

Martin (ref. MAR73) has specified a catalogue of 23 different

dialogue methods. Our three main groups seem to cover most of them.

One may argue that these groups of tools merely reflect three important aspects of interactive application programs, namely the language aspect (which accounts for how messages to the system are composed), the navigational aspect (which makes it possible to have context-dependent interpretation of commands or other messages), and the presentation aspect (which controls the layout of the communication medium, e.g. the screen). Each tool provides support for one such aspect of the intended application system.

Besides environments and tools, a third line of approach is:

3. Programming language features which facilitate writing dialogues. The **Smalltalk** language and system (ref. BYT81) demonstrates how the **class** concept originally developed in **Simula** can be helpful for window management. String processing and interface to a data base management system are also significant features in programming languages that are to be used for programming dialogues (ref. WAS81).

A common characteristics of these three lines of approach is that they provide *a structure within which procedures or other program entities are stored,* or what we shall call a *software structure.* In the operating-system shell, it is the file directories that constitute the software structure; in the programming languages the software structure is defined by the global structuring concepts of the language: blocks, procedures, classes, abstract data types, and the like. In the interactive software tools, finally, the software structure is special-purpose: a node in a transition diagram, or a field in the description of an electronic form, are often associated with one or several procedures, which are triggered when the respective entity is used.

This observation explains why there are two standard ways of implementing software tools, depending on the host programming language. Incremental programming languages (e.g. LISP) allow the programmer to introduce new software structures, which means that the tool can be implemented in an 'interpretive' way: the software structure can be available at run-time. Conventional programming languages, oriented towards compilation and static type checking, require the use of a pre-processor which 'compiles' the software structure, and the procedures attached to it, into one single program or module. Johnson and Lesk (ref. JOH78) describe compiling language tools in Unix. The conventional programming language preempts the choice of software structures.

2. Support of dialogue management in office systems.

The main result in the present work is *the design of a software tool for dialogue management* which is *unified* in the sense that it

provides the essential services of the three groups of tools above, and *concise* in the sense that it has a clean structure and an inspectable definition. (The key idea for combining the simplicity and the generality is a new way of using abstraction hierarchies, which will be described in later sections).

An upper bound on the complexity of the system is provided by the present implementation, which is 30 pages (1500 lines) of prettyprinted Lisp expressions. (The procedural kernel of the system is less than 100 lines). The system is called **CAROUSEL** (ref. SAN81), and is available as a paper listing from the author.

As for all other kinds of software, the challenge has not merely been to provide all kinds of services in one single system, but to preserve the conceptual simplicity of the system as the various services are included.

The need for a unified and concise design became apparent in our own work on the **LOIS** Linköping Office Information System (ref. SAN80), where a number of dialogue management tools were used extensively, both for implementation of various application services, and as a basis for office modelling tools, particularly for modelling and implementing information flow between workstations. The tools were quite useful, but the software maintenance problems for the tools, and the interface problems between the various tools, caused us to look for a more concise global design.

Similar experience has been reported by other researchers, in informal discussions and sometimes in publications. Winograd has written: "*It is in the domain of interaction that there is currently the most to be gained from developing bodies of descriptive structures to be shared by system builders. There are already many pieces that can be incorporated, ... Currently each of these is in a world and formalism of its own. Given a sufficiently flexible tool for describing and integrating interaction packages, this level of description will be one of the basic building blocks for all systems.* (ref. WGR79).

Before we proceed to a description of the unified design, we must however first review the character of office information systems, and the applicability of the currently available tools.

Office information systems are characterized by a relatively large number of moderately complex services, such as the familiar computer mail, personal calendar, personal agenda ('tickler file'), text-file management, and personal data base. In what follows, we shall assume that the reader is familiar with such application programs through his or her own use of them.

Other and more specific services are also important, such as for travel management, scheduling of rooms and equipment, project management, etc. In fact, most of the applications can be characterized as 'XX management systems', where XX ranges over messages, appointments, text files, trips, rooms, and so forth.

Query-language tools are clearly not adequate for the construction of such office applications, since the emphasis in the applications is on customized sets of simple commands, and on very good 'help' services, rather than on complex queries in a rich and flexible language.

Navigational tools create a dilemma. If the network structure is simple, then the user who is action oriented and wants to get a specific thing done, may easily feel constrained by the fixed structure of the available topology. This problem is often met by allowing the navigation space to grow, so that all conceivable user needs can be met: the PROMIS system has 35.000 nodes ('frames') in its network (ref. ROB79). However, building such a network requires a lot of work, and carries with it a danger of costly redundancy in the network, and costly maintenance. Finally, a significant problem for the users is reportedly that they lose their orientation in such a vast space.

Layout tools are often useful, but solve only a part of the whole problem.

Tools which support application-specific command languages seem to be the most natural ones to use, since conventional OIS services (such as mail systems) carry on a simple command dialogue with the user. The problem is that such tools (in their usual, fairly simple form) will only handle one application service at a time. A more general solution is desirable, particularly since most of these services have largely similar command sets: there is a 'create object' command, a 'print' command, an 'edit' command, commands for storing away objects in various 'files', etc. The similarity is a natural consequence of the applications being 'XX management systems'. When the users of the system complain that 'the same thing is called different names in different services', the complaint means that the similarity has not been exploited in the implementation of the total system. That is not only a disadvantage for the end user; it is also a symptom of costly software redundancy.

Although many of the important commands are analogous throughout the various application services, we also have commands which are specific for one application, or a limited class of applications. For example, mail systems is an example of 'browsing' applications where the concept of the 'current object' is defined (e.g. the mail message that is presently being looked at). Operations such as 'next' are restricted to such applications. It follows, therefore, that a simple extension of the query language strategy would not be adequate: a general-purpose command language, complemented by a data definition for each application, would not make it possible to have application-specific commands.

Besides the simple, command-driven services which are 'xx management systems', there are also 'substantial' services which do *something* (e.g. computations) besides data management. However, when such services are to be used interactively, the management of the user interaction often accounts for a surprisingly large part of the

software volume (in terms of both lines-of-code and programmer-hours). Boehm reports experiments where the parts of the program which really perform the task at hand, accounted for only 2-3 % of the software (BOE80). The same considerations as we have discussed for 'XX management systems' also apply when tools are to be used for the interaction part of 'substantial' services.

3. The unified design.

We shall describe the basic ideas of the generalized tool in terms of an interpretive implementation, where the tool is a software environment which may be used by an end user, and whose detailed behavior is determined by application descriptions of various kinds, which are stored in the system. The extension to a compiling strategy does not seem to offer any problems in principle, but only a lot of extra work. (Our own implementation, the CAROUSEL system, is interpretive).

The basic behavior of the software environment is to repeatedly make interactions with the user. In each interaction, the user specifies an *operation*, e.g. 'print', 'enter', or 'delete'. He may also specify *parameters* for the operation. The methods for specifying operations and parameters may vary: operations may be entered by typing a command verb, pressing a function key, or pointing at a menue; parameters may be typed in, or derived from the current state of the machine. For example, the current cursor location may be a parameter.

At each interaction, the current state of the machine and the system determines that it is in a particular *context*, e.g. 'mail management' or 'address directory management'. Thus different applications are realized not as separate programs, but as separate contexts within the generalized tool. The methods for inputting operations and parameters may be defined for each context.

For each interaction, the system looks up the definition of the specified operation in the present context, and executes it. Thus there is an operation-defining mapping

od: contexts * operations --> procedures

which is stored in the data repository of the system. Although each combination of arguments for **od** may have its own value stored separately, *the system obtains its power from the observation that the values of* **od** *may very often be specified for abstractions of the first and the second arguments.*

The use of abstractions for the first argument realizes the observation that we made above, namely that analogous operations may occur in several applications (= contexts). Thus we have a partial ordering -> on contexts so that **c -> cc** means that **cc** is an *abstraction* of **c** (while **c** is a particularization of **cc**), and we make the rule that if the data repository contains

c -> cc
od(cc,op) = p

but no definition for **od(c,op)**, then the listed value **p** will be used instead ('inherited'). The rule is extended transitively.

Since the major difference between applications is that they use different data structures, the context-independent definition **p** of the operation **op**, must often be directed by the data description ('declaration'; 'schema'; we shall use the word 'prototype') that is associated with the specific context **c**. For example, the operation for 'entering a new object' will characteristically prompt the user for the successive fields or properties that should be defined for objects in the current context.

A necessary prerequisite for this arrangement is that the various applications or contexts use a common data base structure for application data. It is convenient to use the same data repository for both application data and system data (such as the **od** mapping), and we shall discuss the structure of the data repository in more detail below. Let us remark here, however, that the data base allows tree structured objects, where e.g. a conventional record can be a node in the tree, with the record's fields as daughter nodes, and a sequence of records can be a higher level node, with the successive records in the sequence as daughters.

Quite often, different contexts require the definitions of operations to be *almost* the same. The 'enter' operation, again, may have to ask the user for the desired type of the new objects (in contexts where objects of several types are maintained in parallell), and may have to insert the object into one or more files (e.g. into the current file, in a browsing-type context). Thus besides the operation-definition mapping **od**, there are other mappings from contexts (or other kinds of objects, such as operations, or types) to auxiliary procedures which are invoked by the operation's main procedure.

The same mechanism turns out to be quite useful for abstraction on operations as well. When the basic operations such as 'print', 'enter', 'edit', etc. are relieved of their context-specific details, the remainder is in most cases a tree traversal, which is performed in parallel over the current object and the prototype for the current object. We can therefore extend the use of abstractions to the second argument of the operation defining mapping **od**: the standard operations have a common abstraction 'traverse', and all cases of standard operations in standard contexts are accounted for by one single entry in the data repository,

od(common-cx, traverse) = travproc

where **travproc** does a tree traversal, and invokes both context-specific and operation-specific procedures at well-defined points.

The most important operation specific procedure is the *leaf procedure,* i.e. the procedure which determines what the operation is to do whenever it reaches a leaf. A 'print' operation prints out the current value; an 'enter' operation prompts the user for a new value, etc. Other operation specific procedures are needed for various kinds of

exception conditions, for example the *missing daughter procedure:* if the tree traverser expects to see a branch to a particular daughter, which however does not exist in the current data base, the action of the traverser depends on the choice of operation: a 'get' and a 'delete' operation will bypass the missing subtree, whereas an 'enter' operation will create an appropriate subtree in this place.

The various operation-specific procedures are often shared between groups of operations. Thus we do not need to define each such procedure for each operation; instead we define an abstraction hierarchy on the operations, and associate the operation specific procedures with nodes on different levels in that hierarchy, with inheritance in the obvious way. For example, operations which act on one single term in a record ('get', 'put',...) often have counterparts which do the same operation on the whole record ('print', 'enter',...). Corresponding pairs in these sets of operations can share abstractions, and beyond it they need only one single procedure to define them - a procedure which is invoked in the process of identifying the tree structure that is going to be traversed.

We can see how concepts from programming languages carry over to the generalized interaction tool, but they re-appear in new forms. As could be expected, the concept of *type* is also significant. It comes in most importantly as follows: among the 'leaf' operations for the various operations or operation abstractions, several must also depend on the type of the value in the leaf. Printout, data entry, editing, etc. is done differently depending on the current type. Such leaf operations therefore use another mapping of two arguments, the type-leaf definition

tld: operation * type --> procedure

which specifies a procedure for doing that operation on leaves containing a value of that type. As for **od**, the possibility of inheritance in each of the two arguments turns out to be very useful.

Besides the instances of tree traversal, there are also other kinds of operations. Some, such as the operation for switching contexts, or the operation for asking for help, are defined in all contexts, and depend on context-specific information, but do not participate in any operation abstraction, and do not use the traversal procdure. Others, such as the operation for moving to the next item in the browsing sequence, are defined over a more restricted context abstraction, and use no operation abstraction. Again others, such as the operations for maintaining catalogs (such as mailing-lists, or sets of particular groups of messages in a computer mail system) can be implemented as temporary shifts into an *alternative context* (which is derived from the current context), and performing one of the standard operations Some additional examples of operations will be discussed in section 6.

The traversal procedure captures the fact that several operations share the same control structure: it is analogous to the *mapcar*-type functions in LISP, and an example of a *control abstraction* in the sense of Liskov (LIS77). (Hägglund (HAG80) has extended the concept of control abstraction to the case when one descriptive structure

allows several different interpreters).

In the CAROUSEL implementation, the use of the control abstraction has led to the following proportion between various types of information in the source file: the traversal procedure is about 90 lines of prettyprinted procedures, the prettyprinted representations of the information for the central contexts and commands is about 500 lines, and low-level procedures for manipulating the data representation and miscellaneous interface procedures is about 800 lines of prettyprinted procedures. Since the information in the latter two groups can be specified and analyzed on a procedure-by-procedure or definition-by-definition basis, and only the executive procedure need be analyzed as a single chunk, this represents a successful modularization.

4. The structure of the data repository in CAROUSEL.

The data repository should be able to store both the information in the applications, and the information required by the system itself. In particular, it should be able to store the abstraction hierarchies for contexts, operations, and value types, and the various mappings from these kinds of entities to procedures (such as **od** and **tld**).

Presumably, most data base structures would be able to represent this information. In our CAROUSEL implementation, the data repository is organized around *entities* and *frames*. The 'entity' concept is used as in Chen's original paper (CHE76), i.e. entities are representations in the data repository, of concrete or abstract objects which bear a significance for the computer user. Persons, cars, computer terminals, operations, and contexts are examples of entities.

Entities are organized in a taxonomical structure using the -> relation. Besides its use for the system information, which has already been described, it is well known that such structures are also useful for modelling application data (see e.g. ref. QUI69 and MYL80).

The CAROUSEL system contains not only a transitive -> relation, but also a set membership relation =>. The latter is however not necessary for the purposes described in the present paper.

Each entity is associated with a *frame*, i.e. a tree structure where the terminal nodes may contain a value (e.g. a string, a number, a text file, another entity, or a set of entities), and where non-terminal nodes have a number of daughters, which are again frames, recursively. Each arc from a non-terminal node to a daughter is marked with a *tag*. Access to information in the data repository is usually specified as the name of an entity, whose frame is to be accessed, and a sequence of tags which are to be used for selecting the appropriate daughter in the successive steps down into the frame.

As usual in such systems, the -> relation on entities defines an inheritance condition for the frames, i.e. if a value or subtree

corresponding to a certain tag sequence is missing in the frame for a concept **c**, and **c** -> **cc**, then a default is provided by the leaf or sub-tree in the frame of **cc** under the same tag sequence.

When a mapping **m** of n arguments **x1**, **x2**, ... **xn** is to be stored in the data repository, it is convenient to store the value in the frame of **x1**, under the tag sequence **m**, **x2**, ... **xn**. The inheritance rule for frames then accounts for the use of abstractions in the first argument of **m**.

For mappings such as **od** which use abstractions in both arguments, additional machinery is required. In CAROUSEL that is arranged by allowing the tags in the frame to be not only entities, but also pairs (**r**, **e**) where **r** is a relation symbol such as ->, and **e** is an entity. If a non-terminal node in the data repository has a daughter whose tag is (->, **e**), and an access is made where the access chain requires the tag **c**, then that daughter is selected iff **c** -> **e**. The scheme is generalized to other relations in the obvious way; tags 'without' a relation symbol are considered to contain the equality relation. Combined with the inheritance rule for entities, this mechanism is sufficient for representing mappings which use abstractions in more than one argument.

Generalized tags may also be occur in the access chain, which is very useful. The tree traversal operations usually do not want to traverse the whole frame (tree structure) that is actually stored in the data repository, but only a selection of it which is described by the current 'view' (in the usual data-base sense of that word) of the data repository. This is conveniently arranged, in many cases, by defining the view as an access chain where some of the links have the form (->, **e**), which means that it matches all actual tag values in the corresponding place in the data repository, which agree with or are particularizations of **e**.

The actual system allows other relations besides = and -> in the tags, including tags which cause the match condition to be computed by a procedure. Propositional connectives on the tags would be a straight-forward extension. Such facilities are needed for more complex applications, but what has been described so far is sufficient for the information required by the system itself.

The information structure described here is of course not new. Sibley has remarked (ref. SIB80), possibly with some exaggeration, that virtually all current ideas with respect to data modelling have been fairly well described already in the 1960's, and some even in the 1950's. The 'entity' concept is very widespread, under various names. 'Frame' structures as described above, i.e. discrimination trees which interact with a taxonomical (= abstraction) hierarchy, have been particularly much used in artificial intelligence research. Carbonell (ref. CAR80B) discusses inheritance from an A.I. viewpoint, and provides additional references. The choice of such a widely used representation makes it likely that it is satisfactory for common applications.

The use of the data repository for system purposes is not restricted to the storage of procedures as leaves in the frames: there is an analogous need to store brief verbal descriptions of entities, to be used for documentation and for on-line user assistance ('help information'). The structure for the procedures that has just been described, applies equally well for descriptions: just as procedures can be shared among the members of an abstraction, so can the corresponding textual description, and just as some subordinate parts of an inherited procedure may need to be specified separately for each participant of the abstraction, the same applies for some parts of the inherited text. For example, the brief description of the abstract 'add' operation may be "add one more member to the <field> field in the <item>". When this string is inherited to the 'cc' command in the 'mail' context, the local information fills in the slots to create "add one more member to the CC field in the message". The substitution of "message" for "<item>" was defined by the 'mail' context, and the substitution of 'CC' for '<field>' was because we have

cc -> ownadd -> add

where **ownadd** is the abstraction of commands which add one more member to the field whose tag equals the command name.

5. Relationships to conventional programming languages.

A conventional programming language contains modes of expression both for procedural behavior (such as conditional expressions and loops), and for what we have called software structures above. The design described here, as well as its implementation in the **CAROUSEL** system, only considers the software structure, and can be used together with any reasonable language for procedural behavior. In that respect it is similar to O.S. shells and to software tools. However, we have seen how familiar concepts in programming languages, such as data abstraction (ref. LIS74), and inheritance of information in a hierarchy (ref. DAHL72) have carried over to the design described here. In this way it continues the trend that was formulated by Jones (JON77), namely increasing the number of concepts that are shared between programming languages and operating systems.

Our software structure has been explained in terms of the -> relation, and the mappings **od** and **tld** which identify a procedure definition in the data repository, and which allow inheritance in both their arguments. The similarities and differences with the software structures of conventional programming languages, are most easily seen if they also are presented as similar mappings, instead of the usual BNF-oriented notation.

In **algol 60** (ref. NAU62), the significant entity for software structure is the *block*. If we let -> be the inclusion relation between blocks, so that **bb** -> **b** means that the block **bb** is enclosed within the block **b**, then algol uses a procedure-defining mapping

pd: blocks * procedure-names --> procedure-definitions

where **pd** allows inheritance in its first argument, just like **od**, but not

in its second argument (where the -> relation is not defined).

Since our intention is only to give a cue about the similarities and differences between the **CAROUSEL** design and a few programming languages, we restrict the discussion to the identification of procedures.

The **Smalltalk** language (ref. BYT81) contains a *class* hierarchy, where -> is the relation from a class to its superclass. Each class has a number of *methods* (roughly: procedures), each characterized by a *selector*. Methods are invoked when a *message* is sent to an *object* which is a member of a class. Let **mc(obj)** be the smallest class of which the object **obj** is an instance. The methods which are given in a class definition express a method-defining mapping

md: classes * selectors --> methods

With this notation, the appropriate method to be invoked when a message with the selector **s** is sent to an object **obj** is **md(mc(obj), s)**, where again we notice that **md** inherits in its first argument, but not in the second one.

Smalltalk has inherited the class concept from the **Simula** language (ref. BIR73), where **cc -> c** is expressed syntactically as

c **class** *cc* ...

followed by the rest of the description for **cc**. However, since it remained in the **algol 60** tradition, **Simula** has its class hierarchy within a block hierarchy, which means that the **Simula** counterpart of the **mc** mapping is defined using inheritance in the block hierarchy, and that we must separate between the name and the definition of a class (since the same identifier may have several definitions as a class, in different blocks).

In summary, the rules for identifying procedure definitions (or what closely corresponds to them) in a number of programming languages, are similar to **CAROUSEL**'s rules since they are defined over a hierarchy of some kind of entities, and allow inheritance in that hierarchy, but at least the programming languages considered here only allow inheritance in one argument of the identification mapping: *one-dimensional inheritance.*

Inheritance mechanisms in other areas also tend to favor one-dimensional inheritance. Carbonell's classification of inheritance mechanisms (ref. CAR80B) focuses on the inheritance of properties between *objects* (while we prefer to think of inheritance characteristics for *mappings*). One then has a choice between allowing a single superior for each object, which results in quite rigid structures, or allowing several superiors, which may be too general (leading to a lack of structure). Focusing instead on inheritance for mappings, provides in fact a structured and restricted way of allowing inheritance from several superiors.

Many operating systems allow a 'path' technique for arranging locality of files, commands, etc. Bourn (ref. BOU78 p. 1981) and Dolotta (ref. DOL80 p. 47f.) have discussed the use of such structures among Unix

users, and remark on the use of paths through directories that represent successively larger organizational structures: assignment, programmer, small group of programmers, whole project. The resulting, single-dimensional inheritance pattern reflects the organizational hierarchy.

Some cases of true multi-dimensional inheritance also exist in the literature. The TAXIS system (ref. MYL80B) allows the user to define composite objects using functions (in the predicate calculus sense of the word), e.g. "the enrollment of a particular student in a particular course", and such a composite object may inherit properties along the hierarchies of its arguments, e.g. "the enrollment of a particular student in any CS course"; "the enrollment of a foreign student in any course".

Hewitt's ACTOR system is reported (informal discussion) to allow two-dimensional inheritance in the following way: it is similar to Smalltalk (ref. BYT81), but every message is an object, and has a position in the abstraction hierarchy which was determined when it was created. When a message is sent to (or rather, arrives at) an object, the resulting event is determined by inheriting for both the message and the receiving object. - Both TAXIS and ACTOR are of course highly experimental systems or designs.

Returning to conventional programming languages, we have noticed that single-dimensional hierarchies are often expressed using textual inclusion. If textual representations of programs are considered significant, it would be a straight-forward exercise to design a notation along the lines of **algol 60** or **Simula** for expressing the **od** and **tld** mappings in **CAROUSEL**. However, in a computing environment which is interactive both for development and for production use of programs, the textual representation of the whole program is of marginal significance, and the primary realization of the software structure is in terms of mappings and, implementation-wise, as structures in the software repository. A variety of different methods may be used for displaying excerpts of the structure to the user.

6. Realization of various services.

The claim for this work is that the services of traditional software tools have been unified and generalized. We have already shown how that works with respect to the implementation of the command language, and the possibility of defining commands across applications. We shall now discuss how the other types of applications are handled.

Navigational systems based on state-transition models can clearly be easily accomodated by letting each state be a context. It is advantageous to let all nodes in state transition networks be particularizations of one common abstraction, which serves as the range for the operations that manage the context-switching.

If one wants to have contexts (nodes) whose primary purpose is to ask one question to the user, get the answer, and proceed to the next node, then the command parsing procedure for such contexts should assume that there is one standard, default operation 'ANSWERIS', and that what the user types in is the single parameter for that operation. (In the MEDICS application system built on IDECS, such nodes are often used; ref. ELF80). Exception-creating commands whereby the user can ask e.g. "what is the purpose of this question" or "what are the allowable answers" can be easily accomodated: the parsing procedure should recognize them as ways of overriding the default operation.

Among *layout tools*, we consider particularly forms management systems. Such systems should consist of two parts: a *design tool* for entering a forms description, and a *production tool* for data entry, data editing and data presentation using an existing form.

The production-tool services are conveniently set up by embedding the formatting information (such as the X and Y coordinates of each field) in the prototype. When the general-purpose traverser procedure traverses the current frame and its prototype in parallel, it is the latter that determines the choice and order of frames. This is natural both because the prototype defines the current view, and because the current frame may sometimes be empty, e.g. during data entry. The prototype also contains the information about the types of the values in the leaves. For these reasons, we can obtain layout descriptions simply by extending the type hierarchy downwards into instances which contain coordinate information. For example, if a standard prototype for an address-directory service stipulates that the 'name' field should have the type **string**, we may compose an alternative prototype where the type in the 'name' field is **string4**, where

string4 -> string
tld(string4, print) = pri4

where **pri4** might be a procedure which prints its argument in position (X = 4, Y = 0) on the screen. (More likely, it will be a procedure which does a cursor movement to a position given as another leaf in the frame of **string4**). All other information for **string4** is inherited from **string**. By creating a prototype from such further instantiated types, we obtain what is effectively a forms description that can be understood by the general-purpose traverse procedure.

The resulting forms descriptions are somewhat elaborate, and it is not convenient to enter them manually. However, the most convenient way of entering a new form is anyway to 'paint' it: the user moves the cursor around the terminal screen, and makes commands such as 'create a data field here' or 'delete that data field' when the cursor is in appropriate positions, and preferably using control keys or functions keys on the keyboard. Thus the characteristic commands are 'put', 'delete', 'move', 'add information to', and so on, just as in the other applications. Form entry is therefore gracefully accomodated using contexts which operate on a fixed frame (viz. the forms description that is currently being edited), in which the node for the

whole form has the individual form fields as daughters, and discriminates between them using the X/Y coordinates as a tag. This allows us to treat the characteristic form editing operations (insert field, delete field, etc.) as particularizations of the general operations put, delete, etc., while the parameter is an X/Y coordinate which is defined by the present position of the cursor.

By contrast, the representation of the form as a modified prototype for production use, requires that the record's field name be used as a tag, and the X/Y coordinates be represented as values in leaves. The transformation between the design tool's representation and the production-time representation, is a simple and symmetric reversal of the tag and one of the leaves, in each of the nodes that represent a form field.

Data base queries can be expressed in several ways. If a query is to be formulated as one single, complex expression using e.g. Boolean connectives, then the facilities described in this paper are not adequate (unless the context mechanism is used to implement an ATN parser). However, the design does support the case where the query is expressed as a succession of short commands which make restrictions, joins, and other operations on a 'working relation'. Also, since design of forms is accomodated, it should be possible to state queries in the present system using Query-by-Example-like techniques (ref. ZLO75).

Operating system shells provide a command dialogue with the user. The design described here has all three characteristics of a shell: it maintains a data repository (of which the file directory is a special case); it performs a command dialogue; and it is able to invoke programs or procedures that are stored in the data repository. Thus it is a generalization of the conventional O.S. shell.

Although the presentation here has by necessity been brief, it has hopefully been sufficient to demonstrate in principle how the services of conventional software tools and software environments have been unified.

7. Discussion.

Many authors have emphasized the need for systematic and powerful methods for dialogue management, for example Terry Winograd as quoted in section 2 above. A statement by Hoare (as cited in ref. WEG79) also applies very well: "*In many applications, algorithms play almost no role, and certainly present almost no problem. The real problem is the mass of detailed requirements; and the only solution is the discovery or invention of general rules and abstractions which cover the many thousands of cases with as few exceptions as possible.*

The present work has been done in the same spirit as both those quotations express. We have shown that interactive user services which are traditionally provided by operating systems and a number of software tools, can be provided in a unified way by a simple and

concise interactive system. Operating systems and software tools are usually considered to be on a level 'above' the programming language. This is particularly true from the viewpoint of classical algorithmic languages, which maintain strict separation of programs and data, where the software tools must be implemented as program generators. The results presented here suggest that concepts and services can and should be moved 'upwards', from the realm of the procedural programming language, to the realm of a *software environment* that combines the salient features of operating systems and software tools.

In the traditional division of responsibility between the software environment (realized by the operating system) and the application program (expressed in the programming language), the software environment provides a fairly narrow and well-defined set of services, and all the application-specific things as well as the continous dialogue, happen within each application program. The application is realized as a monolithic program at least at execution time: when the end user has passed 'through' the operating system shell 'into' the application program, he or she has no contact with the operating system (except for low-level data management and communication services that are clearly 'below' the level of the application program).

The present work suggests a development where the computing environment takes over much more of what is presently done in the application program: it takes responsibility for the *user dialogue*, and for *data management* or "moving data around" in the words of Boehm (ref. BOE80). Furthermore, through the use of the control abstractions, the context machinery, and the embedding of directive information in prototypes, it takes responsibility for a large part of *control*, which in the definition of Salter (ref. SAL76) is "*the mechanism that activates functions in the desired sequence*", or (in interactive situations, we might add), the mechanism that activates functions at the desired time.

Instead of having a large monolithic application program, encapsulated in the thin shell provided by the operating system, we therefore see having an *application description* which consists of number of separate *entity descriptions* (in our case, *frames*), which are individually stored in the data repository provided by the software environment, and whose role is to *steer* the general-purpose behavior of the system, rather than to define the behavior from scratch for each application.

This viewpoint provides an different perspective on the relationship between programming languages and programming environments. Speculatively, we argue that it is wrong to design a programming language first, and think about the design of the programming environment afterwards, as has been the case in the ADA effort. Since hierarchical structures are needed both in the programming language and in the programming environment, it would be reasonable to use the same hierarchy and a uniform representation for both purposes. A software repository which can represent hierarchies, and accomodate various kinds of software structures, may allow and

encourage representations of programs which do not appear easily when language design is done in the framework of the classical **algol 60** syntax tradition. The two-dimensional inheritance that plays a crucial role for the generality and simplicity of the CAROUSEL design, emerged in the data repository representation of the software.

However, the issue is not only about whether the textual representation or the data-base representation of a program should be the primary one. It is also concerned with what kinds of entities should be used in the hierarchy for the software structure.

deRemer and Kron (ref. REM76) have made the distinction between 'programming in the small' (i.e. writing individual procedures) and 'programming in the large' (i.e. organizing the global structures where the procedures are components). We agree about that distinction, but disagree with their view of what 'programming in the large' is about. deRemer and Kron describe a module interconnection language, MIL 75, for programming in the large, and write: "*The universe of discourse of MIL 75 consists of three sets: resources, modules, and systems. Resources are atomic, ... e.g. variables, constants, procedures, data types, etc.*" (third page of paper). We believe instead that the universe of discourse for programming-in-the-large should be *entities in the system's behavior as seen be the end user:* operations, contexts, forms, data files, features that can be turned on and off, printers where the output may appear, etc. These are the things that it is important to describe. In the course of describing them, it is frequently necessary to associate procedures or other program objects with them, but that is a secondary circumstance.

In addition to describing the system's behavior, programming-in-the-large should also be concerned with building models of the application. That issue has not been addressed by the present paper, but previous projects in our laboratory (ref. ELF80, SAN80) indicate that once we have tools for modelling the system's behavior, the next higher level of modelling the application domain can be built with considerable convenience.

Acknowledgements.

Sture Hägglund has strongly influenced the work described here, and also given valuable remarks about earlier drafts of the manuscript, and a number of valuable references. Anders Ström has argued for a long time that data abstraction should be incorporated into our software tools.

Implementation.

The present implementation of the design described here, **CAROUSEL**, is written in **Interlisp** for the DEC-20 computers. With the exception of the scetch for a query facility, all services described in this paper

have been implemented in the existing **CAROUSEL** system.

References.

BIR73 Graham Birtwistle et al: *Simula begin.* Auerbach, 1973.

BOE80 Barry W. Boehm: *Developing Small-scale Application Software Products: Some Experimental Results.* **In** S.H. Lavington (ed): Information Processing 80. North-Holland, 1980.

BOU78 S.R. Bourne: *The UNIX Shell.* The Bell System Technical Journal, Vol. 57, No. 6, July-August 1978, pp. 1971-1990.

BUR80 John F. Burger: *Semantic Database Mapping in Eufid.* Proceedings of the ACM-SIGMOD 1980 International Conference on Management of Data (Peter P. Chen, ed.), pp. 67-74. ACM, 1980.

BYT81 BYTE Magazine, Vol. 6, No. 8, August 1981: special issue on Smalltalk.

CAR80 Eric D. Carlson and Wolfgang Metz: *Integrating Dialog Management and Data Base Management.* **In** S.H. Lavington (ed): Information Processing 80. North-Holland, 1980.

CAR80B Jaime G. Carbonell: *Default Reasoning and Inheritance Mechanisms on Type Hierarchies.* Proceedings of the Workshop on Data Abstraction, Databases, and Conceptual Modelling. SIGPLAN Notices, Vol. 16, Nr. 1, January 1981.

CHE76 Peter P. Chen: *The Entity-Relationship Model - Towards a Unified View of Data.* ACM Transactions on Data Base Systems, Vol. 1, No. 1, (1976).

DAHL72 Ole-Johan Dahl and C.A.R. Hoare: *Hierarchical Program Structures.* **In** O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare: *Structured Programming.* Academic Press, 1972.

DOD80 U.S. Department of Defense: *Requirements for Ada Programming Support Environments - "Stoneman".* 3D1079 Pentagon, February 1980.

DOL80 T.A. Dolotta and J.R. Mashey: *Using a Command Language as the Primary Programming Tool.* **In** D. Beech (ed): *Programming Language Directions,* pp. 35-55. North-Holland, 1980.

ELF80 Johan Elfström et al.: *A Customized Programming Environment for Patient Management Simulations.* Proceedings of 3rd World Conference on Medical Informatics, Tokyo, 1980.

FIK80 Richard E. Fikes: *Odyssey: A Knowledge-Based Personal Assistant.* To appear in *Artificial Intelligence.*

FOR79 M. Ford: *PRESTEL - The British Post Office Viewdata Service.* Proceedings of the 1979 International Conference on Communications. IEEE, June 1979.

GHE75 J. Ghesquiere, C. Davis, and C. Thompson: *Introduction to TUTOR.* Report, Computer-Based Research Laboratory, University of Illinois, 1975.

GOL81 Ira Goldstein: *A Network Representation for Office Information Systems.* 1981 Office Automation Conference Digest, pp 367-369. AFIPS, 1981.

HAG75 Sture Hägglund and Östen Oskarsson: *IDECS2 Users' Guide.* Report DLU 75/3, Datalogilaboratoriet, Uppsala University (Sweden), 1975.

HAG80 Sture Hägglund et al: *Specifying Control and Data in the Design of Educational Software.* Proceedings, CAL81 Symposium on Computer Assisted Learning, Leeds, 1981.

HAM77 M. Hammer, W.G. Howe, V.J. Kruskal, and I. Wladawsky: *High Level Programming Language for Data Processing Applications.* Communications of the ACM, vol. 20, no. 11, 1977, pp. 832-840.

HAM80 Michael Hammer and Brian Berkowitz: *DIAL: A Programming Language for Data Intensive Applications.* Proceedings of the ACM-SIGMOD 1980 International Conference on Management of Data (Peter P. Chen, ed.), pp. 75-92. ACM, 1980.

HER80 Christopher F. Herot: *Spatial Management of Data.* ACM Transactions on Database Systems, Vol. 5, Nr. 4, pp. 493-513 (December, 1980).

JOH78 S.C. Johnson and M.E.Lesk: *UNIX Time-Sharing System: Language Development Tools.* The Bell System Technical Journal, Vol. 57, No. 6, July-August 1978, pp. 2155-2176.

JON77 Anita K. Jones: *The Narrowing Gap Between Language Systems and Operating Systems.* **in** B. Gilchrist (ed): Information Processing 77, pp. 869-873. North-Holland, 1977.

JONG80 S. Peter deJong: *The System for Business Automation (SBA): A Unified Application Development System.* **in** S.H. Lavington (ed): Information Processing 80, pp. 469-474. North-Holland, 1980.

HER79 Christopher F. Herot: *Spatial Managament of Data.* ACM Transactions on Database Systems, 1979.

KAM78 I. Kameny et al.: *EUFID: The End-User Friendly Interface to Data Management Systems.* Proceedings of the Fourth International Conference on Very Large Data Bases, Berlin, September 1978.

LAD80 Ivor Ladd and D.C. Tsichritzis: *An Office Form Flow Model.*

Proceedings of the 1980 National Computer Conference, pp. 533-539. AFIPS Conference Proceedings, Vol. 49.

LIS74 Barbara H. Liskov and Steve Zilles: *Programming with Abstract Data Types.* Proceedings of the ACM Conference on Very High Level Languages, SIGPLAN Notices, Vol. 9, April 1974, pp. 50-59.

LIS77 B. Liskov, A. Snyder, R. Atkinson, and C. Scaffert: *Abstraction Mechanisms in CLU.* Communications of the ACM, Vol. 20, Nr. 8, pp. 564-576 (August 1977)

MAR73 James Martin: *The Design of Man-Computer Dialogues.* Prentice-Hall, 1973

MOR80 Howard Lee Morgan: *Research and Practice in Office Automation.* Invited paper, in S.H. Lavington (ed): Information Processing 80. North-Holland, 1980.

MYL80 John Mylopoulos, Philip A. Bernstein, and Harry K.T. Wong: *A Language Facility for Designing Database-Intensive Applications.* ACM Transactions on Database Systems, Vol. 5, No. 2, June 1980, pp. 185-207.

MYL80B John Mylopoulos and H.K.T. Wong: *Some features of the TAXIS data model.* Proc. 6th Annual Conf. on Very Large Data Bases, Montreal, Sept. 1980

NAU62 Peter Naur (ed): *Revised Report on the Algorithmic Language Algol 60.* Regnecentralen, Copenhagen, 1962.

NEW79 W. Newman and R. Sproull: *Principles of Interactive Computer Graphics.* McGraw-Hill, 1979.

NIE79 J. Nievergelt and J. Weydert: *Sites, modes and trails: telling the user of an interactive system where he is, what he can do, and how to get to places.* Proceedings of IFIP Conference on the Methodology of Interaction. North-Holland, 1979.

QUI68 M.R. Quillian: *Semantic Memory.* in: M. Minsky (ed): *Semantic Information Processing*, pp. 216-270. MIT Press, 1968.

REM76 Frank deRemer and Hans H. Kron: *Programming-in-the-Large Versus Programming-in-the-Small.* IEEE Transactions on Software Engineering, June 1976.

ROB78 G. Robertson: *Some Design Considerations for the ZOG Man-Computer Interface.* Proceedings of the Third NATO Advanced Study Institute on Information Science, Chania, Greece, 1978.

ROB79 G. Robertson, D. McCracken and A. Newell: *The ZOG Approach to Man-Machine Communication.* Report CMU-CS-79-148, Carnegie-Mellon University, 1979.

SAL76 Kenneth G. Salter: *A Methodology for Decomposing System Requirements into Data Processing Requirements.* Proceedings of the 2nd International Conference on Software Engineering, IEEE, 1976.

SAN78 Erik Sandewall: *Programming in an Interactive Environment: The LISP Experience.* ACM Computing Surveys, Vol. 10, Nr. 1, pp. 35-72, March 197 8

SAN79 Erik Sandewall: *A Description Language and Pilot-System Executive for Information-Transport Systems.* Proceedings of the Fifth International Conference on Very Large Data Bases, Rio de Janeiro, 1979.

SAN80 Erik Sandewall, Göran Hektor et al.: *Provisions for Flexibility in the Linköping Office Information System (LOIS).* Proceedings of the 1980 National Computer Conference, pp. 569-577. AFIPS Conference Proceedings, Vol. 49.

SAN81 Erik Sandewall: *SCREBAS Provisional Reference Manual.* Internal report, Software Systems Research Center, Linköping University (Sweden), March, 1981.

SCH79 J. Schultz and L. Davis: *The Technology of PROMIS.* Proceedings of the IEEE, September, 1979

SIB80 E.H. Sibley: *Database Management Systems _Past and Present_* Proceedings of the Workshop on Data Abstraction, Databases, and Conceptual Modelling. SIGPLAN Notices, Vol. 16, Nr. 1, January 1981.

STR81 Ola Strömfors and Lennart Jonesjö: *The Implementation and Experience of a Structure-Oriented Text Editor.* Proceedings of ACM SIGPLAN/SIGOA Symposium on Text Manipulation, 1981.

TEI78 Warren Teitelman et al.: *Interlisp Reference Manual.* Xerox Palo Alto Research Center, Palo Alto, CA, 1978

WAS81 Anthony I. Wasserman: *User Software Engineering and the Design of Interactive Systems.* Proceedings of the 5th International Conference on Software Engineering, IEEE, 1981.

WEG79 Peter Wegner: *Research Directions in Software Technology.* MIT Press, 1979.

WIE79 Johan Wieslander: *Interaction in Computer Aided Analysis and Design of Control Systems.* Thesis, Department of Automatic Control, Lund University (Sweden), 1979.

WGR79 Terry Winograd: *Beyond Programming Languages.* Communications of the ACM, Vol. 22, Nr. 7, pp. 391-401 (July 1979)

WIN81 Patrick Henry Winston and Berthold Klaus Paul Horn: *LISP.* Addison-Wesley, 1981.

ZLO75 M. M. Zloof: *Query by Example.* Proceedings of the 1975 National Computer Conference, pp. 431-437. AFIPS Conference Proceedings, Vol. 44.