Reprinted from *Proceedings, Fifth International Conference on Software Engineering,*

SOFTWARE ARCHITECTURE BASED ON COMMUNICATING RESIDENTIAL ENVIRONMENTS

by

Erik Sandewall, Claes Strömberg, Henrik Sörensen

Software Systems Research Center
Linköping University
Linköping, Sweden

Abstract: This paper describes an alternative approach to software architecture, where the classical division of responsibilities between operating systems, programming languages and compilers, and so forth is revised. Our alternative is organized as a set of self-contained *environments* which are able to communicate pieces of software between them, and whose internal structure is predominantly descriptive and declarative. The base structure within each environment (its *diversified shell*) is designed so that it can accomodate such arriving software modules.

The presentation of that software architecture is done in the context of an operational implementation, the *SCREEN* system (System of Communicating REsidential ENvironments).

1. This paper describes alternative principles for software architecture.

Software design today is based on a number of universally accepted assumptions, particularly about the tasks that ought to be performed by operating systems, compilers, and other software tools, and about what tasks ought to be performed by 'application' programs. Each of those tools provides a package of services. For example, the practical operating system provides not only resource management, but also the top-level dialogue which allows the user to switch between various services, and to transfer data between them. The conventional compiler provides a number of services, such as data description (through declarations; also performed by the data base system), consistency checks, and code generation. The 'application' program is assumed to be in charge of e.g. all but the most trivial aspects of the user dialogue for the application.

This research was supported by the Swedish Board of Technical Development under contract Dnr 79-3919 and 79-3921.

The purpose of this paper is to question this conventional division of responsibilities between the major parts of software. We describe a different overall architecture of the software which we believe is more appropriate. Our view of software architecture has been implemented as an experimental but fully operational system, called the *SCREEN* ("System of Communicating REsidential ENvironments") system. Although the the main results that are reported in this paper is the architectural philosophy and not the piece of university-quality software, we shall make frequent references to *SCREEN*, in order to avoid the possible suspicion that the ideas are mere speculation, and to emphasize that the designs described here are the resuls of practical experimentation over several years.

2. Application example: an office information system.

For concreteness, let us start with one particular class of programs whose treatment relies heavily on the characteristics of *SCREEN*, and use it to explain how the system works and how it is used. In the development of the experimental Linköping Office Information System, LOIS (ref. P1, P2), we wanted to provide support for *information flow* in an office environment, i.e. the scenario where one user is able to instantiate a 'transaction' (e.g. a purchase order, or a request to have certain analyses made for a patient), send it to another user who (immediately or later) performs some operation on that 'transaction' (e.g. authorizes the purchase order), and passes it on again. (For a survey of some recent work on information-flow models, see ref. 1) Within one organization, we will usually have many such information-flow paths, and a many-to-many relationship between flow-paths and users: each path passes by several users, and each user has several paths come by.

In an office environment with personal computers for all employees, we will want each user's personal computer to account for his or her role in each of the information-flow paths that he participates in, and to do so in a well-engineered and coherent dialogue. Thus at run time, the complete system for the organization consists of one sub-system for each user. But at design time, we instead want to design each information-flow application separately, so that the roles performed by different users for this flow are co-ordinated and fit well together.

CH1627-9/81/0000/0144$00.75 © 1981 IEEE

In our project, we have designed a (primarily graphical) language for describing information flows, and an implementation which supports the design and debugging of an information-flow application, as well as the distribution of the segments of the flow to the separate user stations (ref. P3, P4). The application calls for an interesting and non-trivial structure in the receiving, "end-user" computing environment.

Let us consider this from the point of view of the end user. We assume that he or she already has an office information system with the standard facilities, such as:
- a computer mail system
- a personal data base system, where he or she may store data, organized according to the user's preferences using simple data definition commands, and where browsing and report generation from the data is also supported
- an agenda facility, which allows the user to maintain notes about pending duties of several types: "make the following phone calls: ...", "talk to the following people around the office...", "meetings: ...", "write the following reports with the following deadlines:...", etc.

Each of these sub-systems uses a reportoire of interactive commands, and some will also use other, orthogonal menues, e.g. the reportoire of agenda classes, or the reportoire of data sets in the personal data base.

When an information flow application has been designed, and broken down into the pieces that are to be sent to the personal sub-systems for each of several users, how is each flow-piece to be *integrated* into its user's personal system? The easiest way is to say that the segment defines one new 'program', parallell to the mail 'program', the agenda 'program', and all the others. For example, the information flow between a ward in a hospital and the chemical laboratory, would generate one program for the head nurse in the ward, which might be called "laboratory data", and which would contain sub-commands such as "order analysis for patient", "look through today's returned analyses", "look at patient's journal" (because returned data for a patient are assumed to be accumulated automatically to the patient's journal), "find out status of pending analysis", etc. (These fragments occur at the beginning and at the end of the flow-path for each laboratory request). Some of the other stations for this information flow would contain only one or a few operations, if implemented in the same way.

This straight-forward solution is sometimes also the best one. But very often the system becomes more habitable for the user if the segment of the information flow is incorporated into one of the other services. For example, the administrator who has to look over and authorize purchase orders before they can be processed further, but who also needs to be reminded about this duty in order to do it, would prefer to see her part of the purchase order's path as one further agenda category, where it could rely on the presumed machinery of the agenda facility for reminding the user about deadlines, using appropriate devices such as bells, whistles, or color graphics.

Similarly, the user who is on an information flow because he wants to hoard the information in his personal data base, for later browsing, but who is not interested in seeing it when it arrives, would rather have *his* segment of the information flow connect straight into the personal data base. An finally, the user who files a request for vacation once or twice per year, and gets back a positive or negative answer a few days later, would be best served if the answer comes back as a message in the ordinary computer mail (although the manager who approved the application may have seen it differently).

Programs which effect information flow in an organization are conceptually a limited class of programs, but they are also a class of programs which represent a very large volume of actual data processing. These two circumstances taken together are the rationale for designing a special-purpose language for describing information flows. Other classes of programs which satisfy the same requirements, and for which specialized languages have frequently been designed and implemented, are for programming language processors (see e.g. ref. 2), for report generating programs, for data base queries, and for interactive forms management on a display screen. One of our basic design considerations has been to support such special-purpose languages in general; we shall return to that point.

3. A diversified shell provides powerful support for applications.

Let us now switch from the special case of information-flow programs, to the extrapolation to more general issues. The first point we want to make is that *the computer system's (= the operating system's) top-level dialogue with the user should be considerably more developed than what is customary today* in conventional interactive operating systems. The information-flow application provided an example where the shell should contain basic services such as a mail handler, a personal data base handler, and an agenda facility, and where 'application programs' were embedded into the shell at various locations. We shall use the term *diversified shell* for a user dialogue facility with such a reportoire of services, oriented towards certain application situations or computer/user interaction modes.

By employing the term *shell* for the computer's top-level dialogue handler, as previously used in the Multics and Unix (ref. 3) operating systems, we indicate that procedural services and easy coupling of modules are taken for granted in the shell, and that we are adding yet another aspect when going to the diversified shell.

In a conventional operating system, there is only one way of adding more programs, at least from the perspective of the common programmer, and that is by storing the object code for the program as a file, so that it can be invoked by a 'RUN' command. In our concept for an office information system, the mail handler, the agenda handler, and the other parts of the shell define a number of different such points where 'programs' may be added. Each

point makes implications about when the program will be invoked, or made available to the user, and what conventions it has to follow in order to operate correctly.

Notice that the increments do not consist *only* of procedures, however. Commands, agenda categories, etc. are data objects which are associated both with procedural information (e.g. the code for executing the command) and descriptive information (e.g. the help text that goes with the command).

Systems programmers who are working with operating systems in practice already use these techniques, for operating systems actually contain additional 'dispatch' points where 'programs' for specific, system-oriented purposes may be added, e.g. drivers for various types of memory devices or I/O devices. However, these additional dispatch points are not used (and are not supposed to be used) by regular programmers.

The techniques described here are classical in the context of Lisp programming. What we have done is to extend them to classes of applications for which Lisp is not usually being used, and to relate them to issues which are significant for software engineering.

It is interesting to compare also with the CODASYL group proposal for an *end user facility* (ref. 4). Although that proposal only described an EUF with a fixed set of facilities, it generalizes easily to being a specification for a diversified shell, to which services could be added incrementally.

A corollary of our argument is that *operating systems should serve a very signficant role for program structuring.* It is interesting to go back to the classical term 'the stored-program computer', where the computer is supposed to contain one single program, and apply it to the personal-computer situation. In this view, **the program** consists of all the various pieces of executable code that are stored in (primary or secondary) memory, for example on files. The top-level structure of that program consists of the operating system, which is able to invoke lower-level parts of **the program** which are stored on files, by doing a RUN operation. In this view, **the program** is also able to amend itself, under user control (usually indirect user control), so that new services are added and old services are sometimes deleted. This is done of course by adding or deleting program files. What we are talking about in this paper is to reform and improve the currently very primitive architecture for **the stored program** of the personal computer, so that its ability to modify itself (i.e. to facilitate its own software maintenance) is recognized as an asset, and supported as much as possible.

From this perspective, it is a pity that the sub-discipline for operating systems thinks of the O.S. as being *essentially* an administrator for shared resources, and nothing else. Brinch-Hansen (ref. 5, page 1) writes:

> An *operating system* is a set of manual and automatic procedures that enable a group of people to share a computer installation efficiently.

The role of the operating system as a skeleton program onto which more specific (sub-)programs are to be attached, is equally important.

A third observation is that *the diversified shell must provide a data repository which can hold those data which are common for the shell and the various programs that are attached to it.* Operating systems in the Multics family (such as Tops-20 and Unix) allow one program to invoke another, i.e. to perform the equivalent of the RUN command. However, if the caller and the callee want to share data, only crude facilities are available. Incremental programming systems such as Lisp (ref. 6, 7, 8) and APL provide a model here: they allow one environment to hold a large number of procedures, all of which may be invoked by the user, but which may also invoke each other, and which share the common data space of the Lisp 'sysout' or the APL 'workspace'. An operating system might provide that service on a global level and in a less language-dependent way.

One example where such a design would be very useful was provided by Boehm in (ref. 9), which describes how a small, interactive application software product was developed in a controlled experiment. One of the hypotheses was that "most of the code in (such a) product is devoted to housekeeping", i.e. error processing, mode management, user amenities, and moving data around. This hypothesis was confirmed by the two independent development teams in the experiment, which used only 2% and 3% of the code for implementing the actual cost model that was the real purpose of the program. The quoted figures seem extreme, but the general observation seems to recur frequently.

In a computing environment where the shell provides a global data repository and an interactive command language, one should be able to implement such applications by writing their core (those 2-3 % of the old code) as programs which operate on data in the global data repository, and rely on services or general-purpose commands that are already in the diversified shell for "moving data around" and other housekeeping chores.

Finally, *a diversified shell is the ideal target structure for special-purpose languages.* In a discussion of the use of program generators, particularly in Unix, Johnson and Lesk (ref. 2, p. 2156) write:

> Program generators have been used for some time in business data processing, typically to implement sorting and report generation applications. Usually, the specifications used in these applications describe the entire job to be done /.../ In contrast, our program generators might better be termed module generators; the intent is to provide a single module that does an important part of the total job. The host language, augmented perhaps by other generators, can provide the other features needed in the application...

We also use that strategy, but just having module generators leaves open the task of putting together the generated modules into a working system. By contrast, a diversified shell offers to the generators a well-defined framework into which they can insert the generated modules. - Also, we prefer using an interpretive technique called superroutines (ref. P5), rather than the compiler-like technique of a program generator.

4. Residential environments should be able to send and receive pieces of software.

One of the salient features of our architecture, as implemented in the *SCREEN* system is that it is organized as a number of *residential environments* (this term was defined in ref. 6), i.e. almost autonomous systems, each of which has its own diversified shell, and whatever contents have been placed into it. Software updates are often performed by having one environment send a patch of software (programs and/or data) to one or more others. Each environment typically serves one user, and is thought of by that user as one piece of software equipment, or a 'software individual', because it has a unique existence through time, and it changes gradually over time as the result of successive software updates that are sent to it from outside, or made on it by the user.

Environments are similar to ADA *configurations* (ref. 10), since they are built up from available modules, but they differ from configurations in being able to receive updates and to modify themselves as well. Implementation-wise, in *SCREEN* they are Interlisp 'sysouts' (analoguous to APL 'workspaces')(one environment may be one single 'sysout' or several of them which invoke each other).

This section describes the reasons for, as well as some interesting consequences of that architecture.

The information-flow scenario above provided one reason why different end users need their private *end-user environments*, i.e. customized systems which have the same shell, but filled with different contents. Other reasons are because each user may need his or her own customized reportoire of data base queries, report specifications, edit macros, etc., and because a user which is able to modify the appearance of his or her system is likely to be more satisfied with it (just like having control of your physical working environment is psychologically valuable).

At the same time, there are strong reasons why the personal systems should not be self-contained, but instead should be able to communicate software between them. Although advanced tools, such as query languages and forms handlers, enable end users to make some software additions by themselves, certainly the end user will need to be helped in some cases by a programmer. The programmer should then be able to implement the new service on *his* personal system, which could contain professional tools for testing and debugging, support for special-purpose languages for professional use, tools for keeping track of what software updates have been sent to which users, and other software development services. Since new services are often implemented using special-purpose tools, rather than a general-purpose programming language, we shall use the term *development environment*, rather than the possibly more narrow term 'programming environment', for the environment used by the 'programmer'.

Although software updates may also be made directly in the end-user environment, the use of separate development environments is advantageous not only because extra tools may be made available, but also because many services may be needed by several users, and because (like in information-flow applications) some services require different but co-ordinated pieces of software to be sent to different end-user environments.

However, if pieces of software are developed in one development environment (DE) and transferred for execution in another environment (EUE), special care must be taken that they are inserted correctly. There are in principle three possible (although non-exclusive) ways of handling that problem:
- DE:s contain a lot of 'knowledge' about the internal structure of the various EUE:s
- there is a central data base which contains descriptions of the structure of the EUE:s
- each EUE contains 'knowledge' about its own structure, so that incoming software updates can contain abstract specifications of where they are to be inserted, and the EUE decides actively where the patch should be inserted.

We avoid the first approach, because multiple DE:s are needed, and *SCREEN* uses a combination of the second and third approaches, with as much use of self-description in the EUE as possible. One touchstone for the design of the self-description has been the following: each environment must be able to re-generate itself, i.e. it must be able to generate a 'program' which, when executed within a blank Lisp system, will load into that system those modules which are necessary in order to re-create a rejuvenated equivalent of the originating environment.

There are potentially other kinds of environments, such as *end-user programming environments*, which enable an end user to specify the system's behavior patterns in application-oriented terms, and *control environments* which autonomuously control some aspect of the total system's behavior without user dialogue, e.g. causing certain programs to be run at certain hours (like a batch controller). We have however not yet tried using these openings.

5. Structures built from concepts (named objects) provide an updatable structure in each environment.

Through the global data base or by other means, the DE:s and the EUE:s must share some knowledge about the EUE:s structure. It is clearly a good idea to express that knowledge in terms of *what* the EUE:s do, rather than *how* they do it. For the reasons described above, the mapping from the *what* model to the *how* model is stored in the EUE. It is therefore also natural to store at least parts of the *what* model itself in the EUE.

There is also another good use for that information in the EUE. When the end user has to deal with a continously changing computer system, there must be some characteristics in the system which remain stable, and there should also be a frame of reference which allows the user and the system to communicate about how today's system structure relates to last week's structure.

An analogous problem of finding one's way around in a changing environment occurs in a city where construction

often goes on, but only locally. The street map in the city then serves as a framework for the inhabitant, and allows him to accomodate to the changes.

These considerations led to two crucial design decisions: First, *the software structure is organized around concepts, structures, and processes which are seen and used by the end user, or which are useful for describing the application environment.*

In *SCREEN*, e.g. as used for the LOIS office information system, the top-level structure is to give the user a choice between several *contexts* (i.e. sub-systems such as mail, agenda, personal data base). Within each context, there is usually additional choices to be made in one or more dimensions, e.g. the choice between commands in most of the contexts, the choice between agenda categories, between data sets in the personal data base, etc.

Secondly, since this basic framework of contexts, commands, categories, etc. is needed for multiple uses when the end-user environment is running, it *is represented explicitly as a data structure*, not only in DE:s but also in EUE:s. In that data structure, each context name has attached information about what commands are available in it, what local variables are bound in the context, what expressions have to be executed when the context is entered, etc. (We will refer to this structure as a 'data structure', rather than as a 'data base', because it is internal to the environment, rather than globally available data).

The information that is stored in the data structure for each environment, can clearly be viewed as specifications for that environment. In our architecture, these specifications are not a "source" from which the "executable" system is constructed or generated; instead they are *the dominant structure in* the executable system.

The data structure that we use in *SCREEN* is a **frame** system as developed in artificial intelligence research (for an introduction to frames, see e.g. Winston (ref. 7, page 291 ff). It is constructed from named objects (corresponding to what is otherwise often called 'concepts') and relationships between them, implemented using 'slots'. Therefore, the elementary piece of information is the **assignment** of a value to an **object**. (This is an assignment in a static sense - the object has a value - rather than in the dynamic sense of an assignment operation such as the algolish :=). One object may have several **slots**, and an assigned value in each of the slots. The assignment can be given explicitly, inherited, computed when needed, or augmented when needed, all using data-driven procedures (cf. Winston, ibid, page 211 ff).

Concepts are often associated with procedures or other pieces of code. Consider for example the **agenda** (tickler file) facility in the office information system, whose standard version allows the user to ask the EUE to later remind her of phone calls, computer mail that remains to be read, and other duties. An external contributor (e.g. a user, or another environment) may now add more classes of tasks to the agenda, so that an agenda printout may be

4 PHONE CALLS TO MAKE OF WHICH 1 ASAP
11 NEW MESSAGES IN COMPUTER MAIL
LISP CONFERENCE PAPERS DUE NEXT WEEK
6 LEAVE OF ABSENCE REQUESTS PENDING
14% OF WORKING TIME THIS WEEK SCHEDULED FOR MEETINGS

Here the last three items are the printouts for three additional, and user-specific objects in the type **agenda-tasks**. The agenda facility of course knows how to look up all currently relevant agenda-tasks, and for each of them to look up and invoke the little data-driven procedure which inspects the current data structure to compute the figures, and composes the appropriate printout line. The user can proceed directly from this agenda menue to doing the work it suggests (for example pointing at line 4 will bring her into the routine for looking at and approving leaves of absence), or she may merely find out more about the work (for example look closer at the list of intended phone calls). In all cases, her actions causes the invocation of other programs or data attached to the agenda-task.

In many cases, the user obtains multiple, orthogonal options, again often physically represented by menue choices. For example, the services for handling text files, in the LOIS office information system, work with a number of different object types, including:
text files (the objects of direct interest to the user. Reports from the data base may be implemented by associating some file names with procedures for generating the file automatically when needed);
classification categories of text files: teaching, research, etc.;
format of text files: letter, memo, contract, etc. (Each format may be associated with e.g. a procedure which generates the beginning of a file automatically)
formatter used to operate on text files. The formatter object is associated with information both about the pieces of software that make up the formatter, the conventions for calling it, and the command conventions that are used in the text file;
language (English, Swedish, etc.) in which the text is written. Has implications e.g. for the choice of fonts, for the choice of reserved words in the formatter ("page", "chapter", etc)., and for hyphenization rules. Much of this information is best given by procedures associated with the language name.

Classical programming languages such as Simula certainly allow one or more procedures to be associated with identifiers (e.g. Simula class names) in the program. However, the language definition in those cases does not account for treating those same identifiers as known entities at run-time. The examples given above show how many kinds of named objects which are needed at run-time, may also need to be associated with procedures. A **case** statement is in fact a mapping from symbols to pieces of code; our architecture represents that mapping more explicitly by slot assignments.

6. Superroutines are an improvement over program generators.

In every applications area, there is a need for special tools. In office automation, tools for data base queries, report generation, and forms handling are commonplace. There are two common strategies for such tools:

- general-purpose programs with parameters, where some aspects of the program's behavior can be controlled by setting the parameters. A description of a form (giving X/Y coordinates for all fields) is an example of a parameter which is a data structure.
- program generators which translate from a specification language, to a source program in a programming language, which may then be further compiled.

As a rule, program generators may provide greater flexibility. Often they are organized so that there is a fixed top-level structure for the generated program, but some positions in the program called *handles* are variable, and their contents are to be written out in the specification, or are computed from information in the specification.

For example, in the parser generator YACC (ref. 2), there is one handle where code fragments (written in C) associated with each production in the syntax, are incorporated into a case statement, and one handle called yylex where a lexical procedure may be inserted, hand-written or generated using the separate tool Lex.

The procedures or code fragments which are stored as slot values in the data structures of our environments, would be contributions to program generators that we might have in our system. However, since the procedures are already in the environment used at execution time, no program generation is in fact necessary - these code fragments can be called directly.

The technique that we are using then has been called *superroutines* (ref. P5). It is the interpretive equivalent of program generators. The superroutine is identical to the 'fixed, top-level structure' for the generated program, but it is available at run-time, and the variable parts of the program are either computed when needed and executed immediately, or stored in the data structure, and retrieved and executed when needed.

In another view, the superroutine is a generalization of the parameterized program, where the parameters are not restricted to flags or codes, but they may also be procedures or pieces of programs, which are stored in the data structure, and invoked indirectly through data access chains.

For example, *SCREEN* contains a superroutine for forms handling called IFORM, which accepts a form layout description as a parameter, and where each field in the form may be associated with a number of procedures, such as:
- a procedure for checking the correctness of values entered by the user
- a procedure for transforming values entered by the user, to their appropriate, internal form, and for performing any forward side-effects of the entry of these data
- an inverse procedure for printing out values in the data base in appropriate format on the screen

as well as a number of others. The major advantages of the interpretive character of the superroutine are that errors are more easily controlled, updates are performed more easily, and it is easier to make multiple use of the same information (i.e. the information is more 'declarative' rather than 'procedural').

Superroutines may be said to be interpreters for special-purpose languages, e.g. the forms description language. However, one must realize that the forms description (as well as several other of the special-purpose languages) is only used internally in the data structure of the environment. For the user, the forms description is entered by "drawing" the form on the screen, and is edited in the same fashion. It would in fact be better to reserve the term 'language' for the expressions typed by the user *and* by the computer during that dialogue, and to use the term 'knowledge representation' for the large expression in a formal language that describes the whole form. Thus superroutines are more precisely used as *interpreters for special-purpose knowledge representations* (which have often been communicated using special-purpose dialogue languages).

SCREEN environments contain a limited and fairly fixed reportoire of superroutines (for command dialogue, form-oriented dialogue on the screen, report generation, search in the data base, etc., as well as for a number of system-oriented purposes), together with a large structure of conceptual entities which stores information ranging from specifications and documentation, to information which directs the superroutines. Also, it should be clear at this point that *the diversified shell is just one of those superroutines* - it is a fixed procedure with a lot of handles, where the contributions from various applications are attached.

With some simplification, we may then distinguish two kinds of contents in environments: fixed, procedural parts, namely the diversified shell and the library of superroutines, and a data structure part which is variable in the sense that it is tailored to each user, and that it changes over time, more fluently than the procedural parts. These two parts interrelate because the procedural parts often access the data structure to retrieve procedures which are immediately invoked, and which in their turn may often call one of the standard superroutine tools.

7. On programming techniques and programming style.

Environments organized with those two kinds of contents, encourage the programmer to make software updates by extending the data-structure part of an environment. Some heuristics for the programmer (or more generally, the person that develops or extends the software) are:
- when a system is extended with new features which are analoguous to existing ones, for example support for

additional output devices, or introduction of a new kind of structured inter-user messages: try making the extension by introducing additonal named objects into the environment, and providing it with the descriptive information that is needed for execution, documentation, and maintenance;
for all kinds of system extensions, always look out for concepts which are defined in the application, and try to organize the software around it. This technique is useful both because it suggests a structure for the software, and because it prepares the ground for future addition of analoguous features.

For the end user, this software structure hopefully means that he or she can often form a reasonable model of how the software works and how it changes. Users can frequently communicate in terms of embedded languages, at least with a programmer and sometimes (given enough support software) even with the computer system.

The use of application-oriented concepts as a software framework also makes it easier to arrange that the system's behavior, and the changes in its behavior, is comprehensible for the end user. When operating the system, the end user regularly encounters situations where he has a choice of options, be they commands, formats, data set, language, or whatever, and where changes during the evolution of the software are often merely reflected by additions to a menue, or warning texts that a certain, previously existing item has been changed. Requests for software changes that are sent from the user to a software developer, are of course organized in the same ways.

Not surprisingly, updates that are sent from one environment to another often have the same structure, i.e. they consist of definitions for a small number of new objects, assignments to slots for these objects, and assignments to new slots for old objects which already exist in the receiving environment.

This style of working with software marks another departure from old habits. When working with traditional programming languages, we become used to thinking of data as an appendix to the program: the program *is essentially* an algorithm, and it *contains* declarations, and data structures are *created* dynamically as the program is executed. Using *SCREEN*, we think instead of the update as being essentially *a collection of data*, built from objects and relationships, although some attributes of some of those objects may be interpretable as procedures.

8. Incremental advising is useful when the application does not naturally offer named objects (concepts).

The technique of making software updates by introducing new, named objects, or adding slot assignments to existing objects, is not always sufficient. Although we can always resort to conventional code updates or classical Interlisp-style advising (ref. 11), other and more precise techniques are desirable.

Incremental advising is such a technique. When it is used, a procedure may contain a named *advise point*, i.e. a point to which incremental code ('advise') may be added, and other modules may contain advise to the advise point. (An update message that is sent from one environment to another, may constitute such a module). Each piece of advise therefore has the form 'module M gives advise to advise point P that <code>'. Several pieces of advise may be given from different modules to the same advise point, and the origins of these pieces of advise are maintained separately, so that individual pieces of advise may be deleted, or passed on to other environments.

Advise points may be either named objects (like ordinary procedures) or slots of arbitrary objects (i.e. the advise point is a data driven procedure).

Incremental advising is used for many purposes in the software for the *SCREEN* system itself, and in the superimposed LOIS office information system. For example, there are advise points for the entry and exit of a context, for temporary detachment from an environment (Interlisp 'sysout' function) and for the corresponding resumption; for the regeneration of an environment, and so forth. Various services give incremental advise to these advise points; for example, the resumption of an environment has incremental advise for bringing it up to date with what may have happened while it was asleep, e.g. picking up various kinds of messages, as contributed by modules that make use of the respective kinds of messages.

9. Tools for administrating large numbers of named objects should build 'upwards' from the elements of the data structure.

The number of objects and slot assignments grows quite rapidly. The present LOIS system already contains many hundreds of them. Conceptual tools and software tools for administrating them are therefore necessary. *SCREEN* suggests and supports a certain uniform structure on the data. This structure is useful both inside environments, and for defining the extraction of modules from environments and the incorporation of modules into environments. The structure is based on two orthogonal concepts, the *block* concept and a fairly conventional *type* concept. They can be characterized as pragmatic and semantic, respectively: types characterize what data have the same structure; blocks describe what data are to be processed or used together.

The type concept is the familiar one: each object has a type, which implies the existence of certain slots in the object, and sometimes also restrictions on the possible value in the slot. We allow for both the conventional type mechanism, where the type name is a new object whose type is types, and for A.I.-style IS-A hierarchies where slots and slot values may be inherited from superior objects in the hierarchy.

While the data structure is resident in an environment, it is viewed as merely a large collection of slot-value assignments. When information is to be transferred from one environment to another, one usually has to transfer a number of slot assignments. Some possible strategies are to let the sending environment compute by a dependency analysis which slot assignments will be needed by the receiver for a given purpose, or to let the receiving environment ask for assignments when they are needed.

Both of these strategies are likely to be time-consuming. We have therefore chosen instead to use a *block* concept for meeting the same objective.

In principle, a block is simply a set of object/slot pairs. Relative to the current contents of an environment at one point in time, it defines a set of object/slot/value triples. Some operations on blocks that may be performed within environments are:

- *dumping* the block, i.e. putting its set of object/slot/value triples in textual form on a file (sequential file or direct-access file)
- *loading* the block, i.e. an environment may incorporate the contents of a block as dumped by another environment
- *checking* well-formedness of the contents of a block within an environment
- *presentation* of a block (or the union of several blocks) to the user in a well readable format

The block concept originates from the Interlisp *makefile* technique (ref. 11), where blocks are dumped to and loaded from sequential text files. We have used that facility in *SCREEN*, but the sequential-file technique has many draw-backs, and we are gradually shifting to a direct-access technique. The use of blocks for other purposes besides dumping and loading has also turned out to be very convenient. In particular, blocks are the donors of incremental advise as described in the previous section.

The important lesson to be learnt from that experience is: in designing tools for controlling masses of software, do not start by assuming the currently available storage devices (e.g. text files), and then think of ways of filling them with contents. Instead, *first make clear what are your elements of information, then how you want to group it, then how what operations you have on those aggregates of information, and finally, how you want to store them.* If you still think you can use ordinary text files, you have probably done something wrong.

10. The meta level should be handled in the same way as the object level.

The currently running, *SCREEN*-based software consists of about ten environments, containing software that is organized as several hundred blocks, and using named objects in about one hundred different types, each with potentially a considerable number of slots. The whole system is required to change dynamically, not only because of evolving research ideas, but also because of new application requirements.

It goes without saying that maintenance tools are needed for such a structure. In fact, one additional reason for organizing software using the frame-oriented data structure, was the idea that documentation could be integrated into the same data structure in a simple, convenient, and natural fashion.

That hypothesis has in fact been confirmed when working with the system. It has been natural to store various kinds of documentary information in the *SCREEN* environments. Also, just as named objects and object types are introduced for various applications, the task of software maintenance gives rise to a number of object types: *environments* (each of which has a name), modules or what we call *blocks* (each of which of course has a name also), *facilities* i.e. software development undertakings (the handling of fonts could e.g. be one facility, containing a font generator, a font editor, an addition to the formatter for accounting for fonts, etc. Thus a facility contains a number of blocks, subsets of which are given to different environments, and it is also associated with a set of programmers, a set of documents, etc.)

An extensive description of the documentation structure that is built within the *SCREEN* system, has been published elsewhere (ref. P6). Let us just emphasize again one crucial characteristics, namely that the meta level task of describing the system is handled in the same way as object level tasks, and using the same techniques and the same software tools.

11. Conclusion.

Software design has been sub-divided into the tasks of designing operating systems, programming languages, program generators, data base systems, and so forth. Sub-disciplines of computer science and software engineering have dedicated themselves to studying each of these kinds of software. Unfortunately, the top-level design is obsolete. It does not change easily, partly because of old habits, and perhaps also because each sub-discipline focuses on its own task, and takes the rest of the world more or less for granted. The purpose of this paper has been to suggest that we should take a fresh look at those basic design assumptions.

Acknowledgements: The philosophy of software architecture that has been described here, is based on the philosophy that is prevalent in the A.I. community of Lisp users. In particular, we owe a lot to discussions with David Barstow, Cordell Green, Howard Shrobe, Jerry Sussman, and Terry Winograd.

The authors of this paper have worked jointly on the *SCREEN* software. The paper was written by Erik Sandewall.

References.

1. Clarence A. Ellis and Gary J. Nutt: "Computer Science and Office Information Systems". *Computing Surveys*, Vol.12, No. 1, pp. 27-60, March 1980.

2. S.C. Johnson and M.E. Lesk: "Language Development Tools". *The Bell System Technical Journal*, Vol. 57, No. 6, Part 2, pp. 2155-2176, July-August 1978.

3. S.R. Bourne: "The UNIX Shell". *Ibid.*, pp. 1971-1990.

4. Codasyl End User Facility Task Group: "A Progress Report on the Activities of the Codasyl End User Facility Task Group". *FDT Bulletin*, issued by ACM-SIGMOD, Vol. 8, No. 1, 1976.

5. Per Brinch-Hansen:: *Operating System Principles.* Prentice-Hail, 1973.

6. Erik Sandewall: "Programming in the Interactive Environment: the LISP Experience". *ACM Computing Surveys*, Vol. 10, No. 1, pp. 35-72, March 1978.

7. Patrick H. Winston and Berthold K. Horn: *LISP.* Addison-Wesley, 1981.

8. Eugene Charniak, C. Riesbeck, and D. McDermott: *Artificial Intelligence Programming.* Lawrence Erlbaum Associates, Hillsdale, N.J.

9. Barry W. Boehm: "Developing Small-Scale Application Software Products: Some Experimental Results". S.H. Lavington (ed): *Information Processing 80*, pp. 321-326 (participants edition), North-Holland, 1980.

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

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

Project bibliography.

This section lists previous, relevant papers from the research activites within which the SCREEN system developed.

P1. Erik Sandewall et al.: "Linköping Office Information System - an Overview of Facilities and Design". *Data* (Copenhagen) January-February, 1979.

P2. Erik Sandewall et al.: "Provisions for flexibility in the Linköping Office Information System". *Proc. (U.S.) National Computer Conference*, 1980

P3. Erik Sandewall, Erland Jungert, Gunilla Lönnemark, Katarina Sunnerud, Ove Wigertz: "A tool for design and development of medical data processing systems". *Proc. Medical Informatics Berlin*, Sept. 1979

P4. Erik Sandewall: "A description language and pilot-system executive for information-transport systems". *Proc. 1979 Conf. on Very Large Data Bases, Rio de Janeiro.*

P5. Erik Sandewall: "Why Superroutines are Different from Subroutines". *Report LiTH-MAT-R-79-28*, Linköping University, 1979.

P6. Erik Sandewall, Henrik Sörensen, Claes Strömberg: "A System of Communicating Residential Environments". *Proc. 1980 LISP Conference*, Stanford, CA.