SOFTWARE DESIGN REPRESENTATION: ANALYSIS AND IMPROVEMENTS*

Peter Freeman

May, 1976

Technical Report #81

Department of Information and Computer Science
University of California
Irvine, CA 92717

*The project studied in this report was funded by the United Nations Development Programme and was carried out at the International Computer Education and Information Center in Budapest, Hungary.

1. PRELIMINARIES

1.1 INTRODUCTION

We are interested in design, especially the design of software*. One way of characterizing design is that it is a process of building a representation of the object to be created. We take it as self-evident that *some* representation of a piece of software exists from inception until a binary image is loaded into memory for execution. The issue we are concerned with in this paper is the form of that representation.

The reasons for studying design representation are probably well known to the reader, but are restated for the benefit of those new to methodological considerations: Representations, in some form other than contents of a human memory, are needed so that the information involved can be communicated to others; the limited capacity of the human

*It has been often observed that the methods used to design a wide range of objects are quite similar in many respects. This fact is especially true in the design of information processing systems that have both hard and soft elements. Thus, much of what we have to say here may apply equally well to hardware or software. We have chosen not to consider hardware explicitly in this paper, however, for two reasons: 1) hardware has a number of additional, very useful representations, the consideration of which would carry us too far afield and 2) the concrete example on which this study is based did not include any hardware design.

mind for technical detail must be augmented (represenations of current software systems often run to hundreds or even thousands of printed pages); the form of the representation may greatly affect the performance of various design tasks (review, backup, prediction of results); the representation of information affects our success or failure in producing more refined representations that are closer to the desired final result.

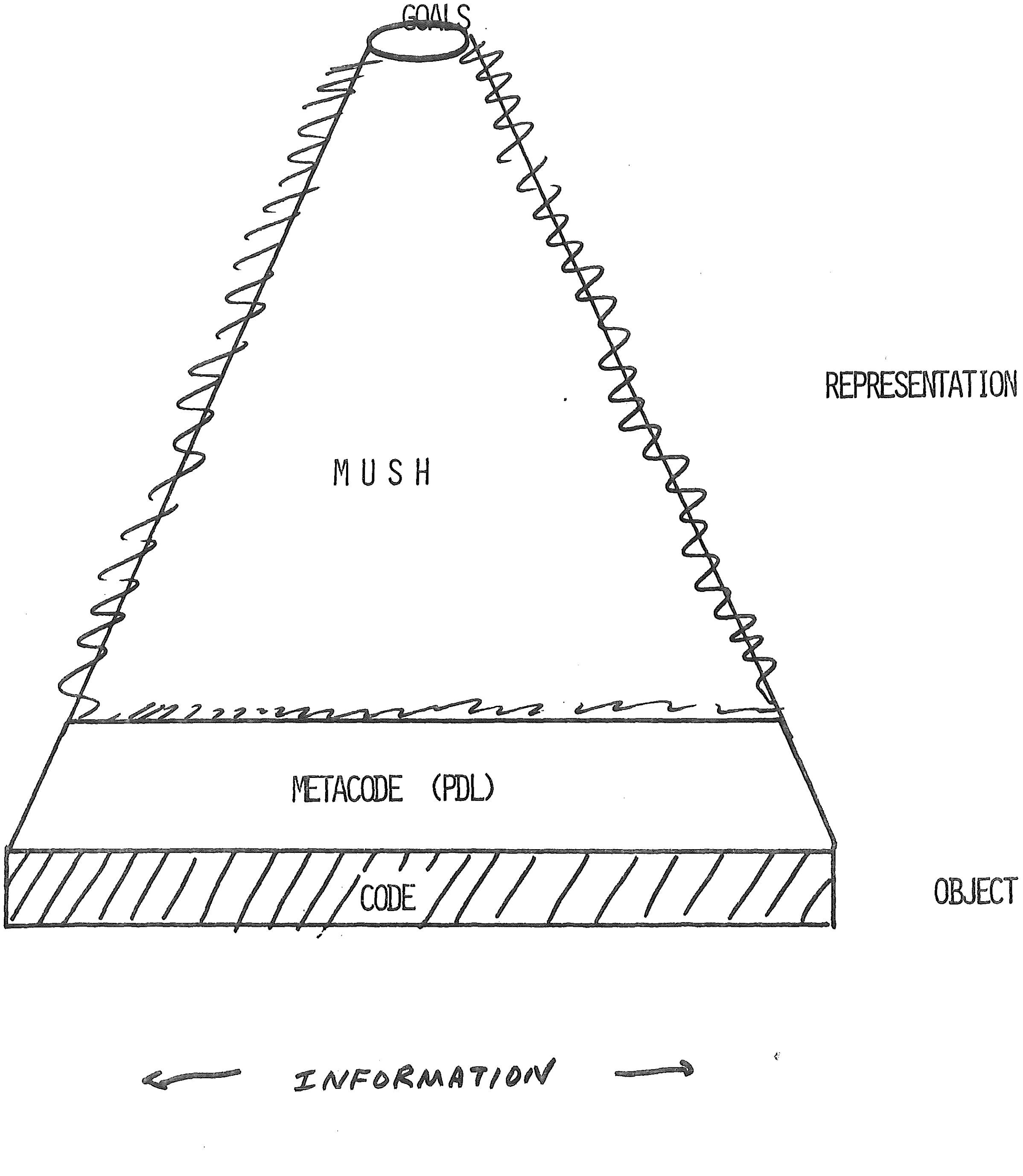
We have no panaceas to offer. What we have is a detailed example of the representations used on a particular project (actually, pieces of a detailed example too lengthy to present here), an analysis of those representations and their usage, and some suggestions of *how to improve the representation of software designs*. Following some preliminaries, we will describe the case study, analyze it and suggest improvements in that order.

1.2 A VIEW OF DESIGN

A *design* is a representation of an object. Fundamentally, a design is a collection of information that tells us something about the object we want to create. Thus, the *design process* can be viewed as an information-gathering process that continually expands the design by putting more and more information into the representation until a termination point is reached. Typically, we terminate a design when we have enough information to implement the object. Figure 1 illustrates our informational view of design.

It follows then that design is a process of deciding what information to include in the design and what information not to include. This has led us in an earlier paper [Freeman, 1975] to propose that it is primarily the decisions made in the course of design that must be represented.

FIGURE 1 : ONE VIEW OF DESIGN



1.3 REPRESENTAION *VS* DOCUMENTATION

It is important to distinguish representations from documentation. We, along with most people concerned with software design, have not been careful about making this distinction in the past. One of the main points we want to stress in this paper is the need for this distinction.

The current use of the word "documentation" in software creation usually denotes information describing an object such as a program, an operating procedure, or the results of a design process. This word has the further connotation in many instances, especially in design where the object does not yet exist, of representing everything we know about the object.

It is precisely this connotative use of the word "documentation," however, that we believe has prevented clear thinking about design representation. In the case of a program that exists, it is understood that the documentation *augments* the actual object and that we must go to the object itself as the final authority on questions of substance. (This in fact is a standard maxim given to those attempting to understand a program). In the case of a program *design*, however, the "documentation" *is all that exists*. Thus, it is more than information about an object (i.e., documentation), it is a representation of the object being created.

By not being clear about what our flowcharts, lists, diagrams, and voluminous prose generated during design really constitute, we have not thought clearly about the role of these *representational forms*. Of course, everyone recognizes that the stacks of paper generated during a design process contain in theory all the information we have available about the object (usually, there is additional, unrecorded information). Yet, few people have specifically treated such *ad hoc* collections of information as a representation in the same way that an architect views graphical representations of a building or an engineer views mathematical representations of a physical process. Just as other design professions have carefully evolved (and in many cases consciously created) the representational forms they use, so must we do the same for software design.

Recently there has been increased concern taken with software design. A number of books (e.g. [Lucas, 1974], [McGowan and Kelly, 1975], and [Yourdon, 1975]) and papers (e.g. [DeRemer and Kron, 1975], [Freeman, 1976b], and [Stevens, Myers, and Constantine, 1974]) have appeared that treat various aspects of software design.*

* There are several attempts to develop a more comprehensive representation for system design that will include hardware and software [for example, Chu, 1975]. Our purpose here is not to survey the field, however, but to try to increase our understanding of the issues by close examinatin of one particular example.

Some of these have explicitly addresssed the issue of representation: Structure charts [Stevens, Myers, and Constantine, 1974] and program description languages (PDLs) [Caine and Gordon, 1975] are both forms for representing program designs and have been used with some success. Peters and Tripp [1976] survey a number of graphical design representation schemes.

Our own work on software design rationalization [Freeman, 1975] provided another form for representing a design; while typically requiring too much effort to carry out in the full format we earlier proposed, its ability to capture the decisions being made during design led us to one of the representational forms discussed below*.

----

* Software design rationalization is a technique for capturing not only the design decisions, but the reasoning that led to them. Specifically, one is instructed for each decision, to list a set of alternative choices, an evaluation for each alternative, and the reason for the one actually chosen. While it is clear that this provides a great amount of information not normally found in a design, which may be very useful to later attempts to understand the design, it is not clear that the added effort of producing this detailed "rationalization" is cost-effective. The decision statement representation discussed below appears to be a more natural representation for actual design situations.

In the remainder of this paper we will illustrate and analyze the usage of several representational forms. Not surprisingly, we will find that although paying careful attention to the representational issues outlined above appears to have improved matters, we are still a long ways from having a truly satisfactory representation for software design.

2.0 THE CASE STUDY

2.1 OVERVIEW

The analysis we will present in later sections is based on a case study of an actual design project. The Remote Text Editor (RTE) project was carried out by a group of five people under the direction of the author; it resulted in the design and implementation of a small time-sharing system. The work was performed at the International Computer Education and Information Center (Szamok) in Budapest, Hungary.

A sizeable fragment of the design, along with a description of the representational forms used, is presented in [Freeman, 1976a]. References in this paper to the "case study" should be read as implicit references to that paper. Some of that material is reproduced in this section for ease of reference along with some new material.

2.2 FORMS OF REPRESENTATION USED

Six main representational forms were used in the RTE project; each is defined and illustrated below:

English prose: Ordinary narrative descriptions.

example: The FS consists of a collection of reentrant programs. These programs, taken as a group, are a part of each process-type in the RTE system.

Lists: Simple lists of items which are usually a single word or phrase.

example: required control functions:

start editing
start copying
start directory manipulation
submit job
quit subsystem
logon
logoff

Decision statements: Simple English statements that record a decision with little elaboration. Each decision is represented by a single statement.

example: Files will be accessible in two modes: block or record.

Functional module specifications: Inputs, outputs, function, calling sequences, and constraints that define a program module; presented in a fixed format with little or no narrative explanation.

example:

PROTECT(filename, owner-id, user-process, rtn-code)

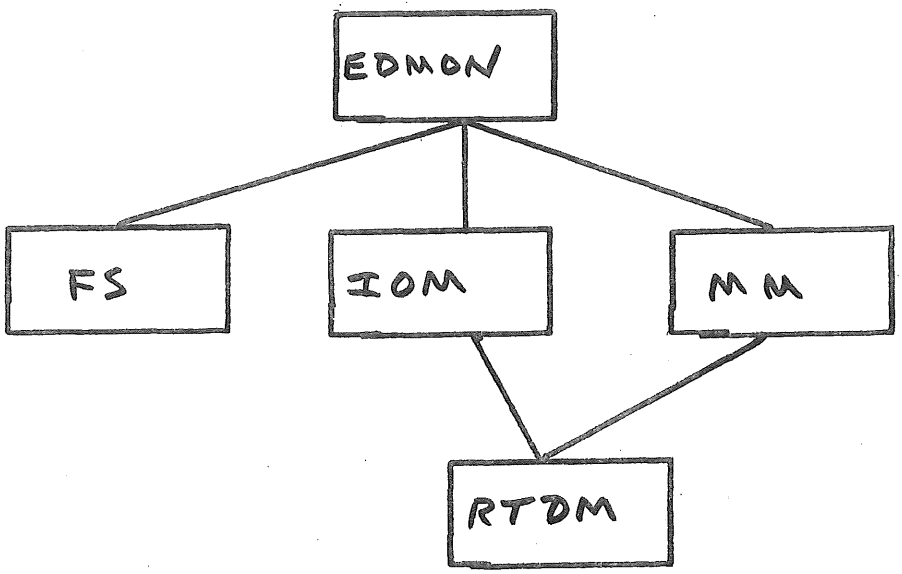
function: Mark the specified file read-only

inputs: filename
owner-id
user-process

outputs: done
file open - no action
user = owner
owner does not exist
user does not exist

Graphical forms: Flow charts and system structure charts indicating intermodule dependencies of control and data flow.

example (structure chart)



Metacode: A PDL (program description language) Algol-like language consisting of ordinary control flow statements (if-then-else, do-while, repeat-until, exit, exit-with, case) and free-form English statements for conditions and operational statements. Ordinary programming statements are permitted, but are discouraged. Data structures are either implicit or logically (not physically) defined.

```
example:  begin
               if owner = user then exit
     with('user =owner');
               if file is open then exitwith('file in use');
               get directory for owner;
               if none then exitwith('owner does not
     exist');
               find file entry;
               if none then setreturncode('file does not
     exist');
               else begin
                mark file read-only;
                setreturncode('done');
                end
               release directory;
     end
```

2.3 STAGES OF DESIGN IN THE RTE PROJECT

This project consisted of several definite temporal stages. Each stage also corresponded to the level of decisions being made:

Stage Ø: Specifications by customer

Basic objectives of project were defined. Organizational resources (people and machines) committed. Overall schedules set. Represented by: prose.

Stage 1A: Detailed specification

Functional specs of system were developed by design team and advisors. Operated as co-routine with Stage 1B. Both involved decisions at about the same level. Represented by: prose, lists.

[Note: Throughout, the "level" of decisions referred to is merely the mode; often at a low level some high-level decisions were made and vice versa].

Stage 1B: Overall structural organization of system

Major decisions regarding main pieces of the system and how they interact. List of main ideas of how to build each piece (not complete). Represented by: prose, lists, structure charts.

Stage 2: Preliminary design of each facility

Functions and interactions of each facility were considered and major decisions made about how they would work. Basic input formats and data structures set. Represented by: decision statements.

Stage 3: Refined design of each facility

Each decision made in Stage 2 was directly expanded by considering its implications, the decisions made at Stage 2 for interacting facilities, and more details of the underlying hardware and RTDM system (a real-time disk monitor supplied by the manufacturer). Represented by: decision statements.

Stage 4: Preliminary module design

The decisions of previous stages were examined for implicit definition of modules for each facility. Common service routines were identified. Modules were specified by stating inputs, outputs, and functions precisely. Some rough metacode was written. Represented by: prose, module specifications, structure charts, and metacode.

Stage 5: Detailed module design

All needed modules were specified in metacode. Data structures were specified at the logical level. Repcesentated by: functional module specifications, metacode.

2.4 REPRESENTATIONAL NEEDS

In this section we will summarize the representational needs experienced in each stage of the RTE project described above. For each stage we will describe the information inputs and outputs to the stage (these comprise the main items to be represented), the constraints considered at that stage, the types of decisions made, and the dominant representational forms actually used.

Stage 0: Customer Specifications

inputs: basic needs of the sponsoring organization
physical resources available
people resources available

outputs: work order
constraints on the task (technical and organizational)

decision types: information and action needs
[not made by designers]

representations used: verbal instructions and agreements
memoranda defining the task

Stage 1: Detailed Specifications

inputs: work order
knowledge and experience with other systems
detailed knowledge of available hardware and software
knowledge of application area

outputs: detailed list of user functions
initial structuring of system into facilities
listing of major functions for each facility
dominant interactions between facilities

constraints: capabilities of machine, software, and design staff
user needs and level of sophistication

decision types: functional capabilities
system structure

representations used: prose
lists
structure charts

Stage 2: Preliminary Facility Design

inputs: outputs of Stage 0 and 1
models of other systems
analogous information handling situations
requirements of facilities defined in Stage 1
probable structure of facilities defined in Stage 1

outputs: sets of decisions

constraints: hardware capabilities and structure
Stage 2 decisions

decision types: detailed function, structure and format descriptions.

representations used: decision statements
formal module specifications

Stage 3: Detailed Facilitiy Design

inputs: outputs of previous stage
decisions of Stage 2
knowledge of other systems
requirements - global and of other facilities

outputs: detailed decisions based on decisions of Stage 2
some module specifications

some constraints on functions, structures, and formats

constraints: hardware
Stage 2 decisions

decision types: detailed function, structure, and format descriptions

representations used: decision statements
formal module specifications

Stage 4: Preliminary Module Design

inputs: outputs of all previous stages
programming knowledge

outputs: required module list
service module list
rough design of major modules
module specifications, including further decisions

constraints: previous decisions
programming requirements of system and machine

decision types: most format and structure
some further functional requirements

representations used: lists
metacode at high level
some prose

Stage 5: Detailed Module Design

inputs: previous stage outputs
programming knowledge
rough module designs of previous stage

outputs: detailed data structures (logical)
detailed metacode (logical)
precise input/output formats

constraints: previous decisions
programming structures

decision types: program structures
housekeeping structures

representations used: metacode
diagrams
prose (documentation)

2.5 NARRATIVE DESCRIPTION OF THE RTE PROJECT

The design of the entire system followed the temporal stages described above. For example, Stage 1 activities were completed before Stage 2 activities were begun. A certain amount of backtracking was necessary, as in any design effort; but, with the exception of the RJE (Remote Job Entry) communicator, the design of the other facilities* proceeded straightforwardly.

In Stages 1 to 3 there was a large amount of interaction between the design of different parts of the system. Beginning with Stage 4, the interaction was more limited and localized.

The original project goals were laid down by Szamok management verbally and in memoranda, but in Hungarian. No direct translation was available. They were the usual type of high-level general directions of management to a technical organization. They set resource levels and defined certain global constraints on the form and function

* The total RTE system consists of several major pieces, described as "facilities": memory manager, process manager, i/o manager, file system, editor, command interpreter, RJE communicator, and RJE initiator. See [Freeman, 1976a] for more details.

of the system.

Considerable effort was expended on the RTE project before the author joined it. One of the results of this activity was a more detailed statement of specifications for the system embodied in a report that was given back to management for review.

When the author joined the project, it was soon evident that although the major pieces of the system had been identified, the specifications for each of these pieces were not well understood. Further, the required user functions were not clearly delineated.

A series of group work meetings were held over several days at which we first worked out a comprehensive view of the system as the user will see it, and then developed a functional specification of each major facility. Note that we did not develop detailed specifications of some facilities such as the command language interpreter and copy system which do not interact significantly with other facilities.

Starting from the functional specifications , we began to devise software structures that would satisfy the specifications. We worked top-down by considering analogs of the subsystems, listing requirements and constraints, and identifying major decisions. After choosing an overall structure, we concentrated on details.

The decisions and information generated at the first stage came about largely through free-association and brainstorming. Some decisions were explicitly elaborated later, but primarily the first stage formed a basis for the more organized decision making of the next stage. At the end of Stage 1 we had a clear representation of the requirements.

In Stage 2 we undertook to identify and make the major decisions that would determine more precisely the structure of subsystems. We did this by considering the requirements and the hazy structural outline produced in Stage 1B, and by trying to identify design features for which there seemed to be alternatives. If such a feature seemed low-level (i.e. not having global implications) we ignored it; otherwise we made a decision and recorded it exactly as shown in the case study. Alternatives for each decision were considered -- often at length -- in the manner of design rationalization [Freeman, 1975].

The decision statements in Stage 3 were direct expansions and implications of the decision statements in Stage 2. For each decision statement in Stage 2 we asked what implications, sub-decisions, and co-decisions were possible. The resulting decisions were then made, as in Stage 2, by consideration of alternatives. While sub-decisions of different Stage 2 decisions sometimes necessitated coordination, we did not have to make major revisions to the Stage 2 decisions.

After Stage 3 we felt the decision statement refinement had carried us as far as it could and we set out to specify actual program modules. While Stage 4 generated a number of decisions and information that later (Stage 5) permitted us to produce detailed module designs, it was not a cleanly represented stage of the design. Bluntly, the representation used in Stage 4 was a "mush."

Out of the "mush" of Stage 4 we proceeded to identify needed program modules. In Stage 5 we designed these modules in sufficient detail that we could then code and test them. Our decisions were all represented in the standard organization of module specifications and the metacode used to describe the logic.

3. ANALYSIS

Throughout the following we assume some familiarity with the case study summarized in Section 2 and more fully described in [Freeman, 1976a].

3.1 APPROACH

Our primary purpose in this study is to increase our understanding of software design representation by careful consideration of a particular case study. Having presented what occurred in this particular project, we now want to identify some cause and effect relationships.

A necessary first step in an analysis of this sort, identifying the constituent parts, has already been done. In what follows, we will limit most of our analysis to two particular forms: decision statements and metacode. While other forms were used, these two were dominant.

Our approach to this analysis has three parts. First, we will identify several important features of any representation along which the two forms of interest can be evaluated. Because we are interested in the total representation of a design, we will also evaluate the two forms taken together along these same dimensions.

estimate value

Second, we will assess the impact of these forms (and some of the others used in the RTE project) on the design process. We will identify specific ways in which the forms helped or hindered us as we designed.

Third, we will assess the impact of the design process on the forms. We want to understand to what extent the choice of forms was dictated by the design steps and to what extent those that were chosen were modified because of the design process.

As with any complex and unformalized endeavor, these evaluations are necessarily subjective. It is clear that whenever possible, we should strive to obtain objective evaluations. Yet, it is also clear that most progress in improving the state-of-the-art in software engineering is based on subjective evaluations due to the extreme difficulty of obtaining objective data of significance. Further, careful objective studies usually must be preceded by subjective analyses that point the way toward areas for deeper investigation.

3.2 ANALYSIS BY PARTS

3.2.1 Underlying Factors -

As the theory and practice of a particular representation is developed, highly detailed analyses can be carried out. For example, we can analyze a programming language along a large number of well-understood technical and practical dimensions (see, for instance, [Sammet 1971]). Unfortunately, this is not yet the case for design representations. One result of studies such as this should be a gradual establishment of the proper dimensions to use in analyzing representations. Thus, we must first discuss briefly the factors we consider important for analysis.

A representation is a collection of information expressed in graphical, mathematical, or written form. Its main reason for existence is to facilitate two fundamental operations of design: review and elaboration. A design must be reviewed at a number of times for a number of reasons: 1) the designers themselves must understand parts designed by colleagues and refresh their memories on their own decisions, 2) peers must review it for technical accuracy, 3) managers must review it to determine resource allocations for future stages, 4) and customers must review it to see if it meets specifications and expectations. Later, a design may be reviewed by students, researchers, and other designers in order to understand the conceptual basis and logical structure of the object that has been created.

Elaboration is the basic operation by which we achieve progress in design. Even if a design is not strictly organized into logical levels, any design process is fundamentally one of continually elaborating the information in the design. In effect, designers apply operators (albeit in a complex manner) to the information represented at one stage of a design to obtain the next, more elaborated representation. A design representation must facilitate these two main operations of design.

The factors we have chosen for analysis are directly related to these two primary design functions:

clarity - are the representations clear and easy to understand?

completeness - do they permit all needed information to be represented?

accuracy - can facts and relationships be expressed precisely?

consistency - are similar items expressed in similar ways? are items expressed in the same way at different levels of specificity?

ease of use - do people find them natural and simple to use?

cost-effectiveness - are they expensive to produce and/or maintain?

3.2.2 Information Elements Of A Design -

A fundamental requirement of any design representation is that it be able to represent the elements of the object in question. In the case of software design, we can identify the following kinds of information:

decisions - choices among alternatives;

relations - logical connections between other elements, either physical or informational;

constraints - statements of conditions that should not be broken by the design; may include constraints on the process itself, but these are usually outside the representation;

open questions - an important part of any design is determining what is not known or what is partially known; this information needs to be a part of the design representation (although one might argue that it is properly a part of the design documentation);

requirements - much the same as open questions;

functions - descriptions of what something is to do;

structures - programs, data structures, control structures, formats.

Bear in mind in the following that these are the types of information that went into the representations we are evaluating. The case study [Freeman, 1976a] provides numerous examples and the analysis presented above in Section 2.4 also illustrates the range of information present in this design.

3.2.3 Summary Evaluation -

Table 1 presents a summary evaluation of the underyling factors described above. Only those forms (metacode and decision statements) which are used to any great extent are evaluated; their combined effect* (since they formed the bulk of the representation) are also evaluated.

* By "combined effect" we mean "taken together to form a whole" and are not implying their use side-by-side on the same page.

	decision statements	metacode	decision statements & metacode overall
clarity	8	9	5
completeness	6	10	4
accuracy	9	7	8
consistency	contradictions 5	7	5
ease of use	8	10	7
cost to use	10	10	10

scale: 1 — 5 — 10
bad — good

Table 1: Summary Evaluation by Factors

Table 2 presents a summary evaluation of the ability of metacode and decision statements to represent the information in elements in our design.

Information Elements	Decision Statements	Metacode
decisions	10	4
relations	3	10
constraints	8	3
open questions	0	3
requirements	0	0
functions	10	8
structures	2	10

scale: 0 — 10
unable to represent — easy to represent

Table 2: Summary Evaluation by Information Represented

In the next two sections we amplify and illustrate these evaluations.

3.2.4 Discussion Of Factors -

Clarity (8/9/5)*

The clarity of decision statements is high. For example, consider the LFS Preliminary Decision Statements in the case study. It is easy to see quickly what the primary features of the LFS will be by reading these decision statements. When they are later refined, the ramifications of the higher-level features are readily apparent.

What is missing is a representation of the interrelation of decisions. Thus, the connection between the decisions on file sharing and file protection is unrepresented.

Metacode is generally easy to follow as evidenced by the samples in the case study. The logic is given in unambiguous form through the use of structured control statements while unnecessary details are suppressed and the operations are represented in natural language that is easier to comprehend (in general) than straight programming language statements.

----

* (8/9/5) are the evaluations of clarity given in Table 1 for decision statements, metacode, and the two forms taken together.

is not as good as either alone because of the gap between them. That is, they are rather different forms of representation and do not interface well. As we progressed through the stages of design there was no problem as long as we were within a single form, but when we worked back and forth between the two forms the combination did not clearly represent the design. This is most clearly seen in the case study in Stage 4 where we attempted to create a metacode representation directly from decision statements.

need a relational rep. bet the representations.

Completeness (6/10/4)

Decision statements are only fair along the dimension of completeness. A reasonable amount of the design was not captured by them, due largely to the fact that they did not express immediately the implications of design decisions, such as the interrelationships between parts of the system. In the case of metacode, on the other hand, when something is written down, we can determine its relationship to other parts, but only on a local basis. This is clearly a consequence of the fact that metacode, as a form, has a definite structure while natural language statements do not.

There is structure between levels of decision statements, however, and this removes some of the problems. For instance, from a first-level decision regarding the simplicity of file names, we can see the relation of several other decisions concerning the exact rules governing name formation. The relation between these decisions and those governing the structure of the directory are not so apparent, however.

Metacode permitted us to represent the design in as much completeness as we wanted at the logical level. For example, in the representation of the OPEN function, there was no problem in showing all the conditions and operations of interest. Since we could use programming-language statements, we could have made it fully complete.

Taken overall, the two forms did not provide a very complete representation as we progressed through the stages of the design. The lack of structure representation posed a serious problem as we moved from decision statements to metacode. Thus, our rating of their combined effect is low.

Accuracy (9/7/8)

Decision statements are straightforward, although in some cases there is ambiguity. For example, the decisions on file space management do not point us to a definite implementation. The accuracy of the decision statements is just the accuracy of natural language, improved somewhat by the sparseness of the format and the absence of extraneous verbiage.

The metacode we used introduced some ambiguity, especially in data structures. This, however, was due to our failure to describe data structures at the metacode level, not to the representational form itself. However, the use of English expressions permits one to be sloppy, sometimes resulting in inaccurate statements.

Overall we found the representation to be accurate. Accuracy tends to be a localized property and thus to be additive. No particular accuracy problems were noted in going from decision statements to metacode.

Consistency (5/7/5)

Neither form made it easy to achieve a consistent representation. Decisions were often made at widely differing levels of detail even when an attempt was made to limit to them to a particular level. The representation helped only minimally to organize decisions by level or by area. This is especially noticeable in the refined decision statements of Stage 3.

Metacode is somewhat better. The use of the same control structures imposes a certain amount of consistency on the descriptions. Standardization in the terms and operations used would further increase the consistency of metacode descriptions. This is illustrated in the case study where set phrases such as "get owner's directory" are used in several different functions.

The difference between the two forms and the "distance" between them is basic enough that it really doesn't make sense to address seriously the issue of overall consistency.

Ease of Use (8/10/7)

We found both representations easy to use (i.e., generate), although decision statements were not as easy as metacode. This appears to be partly due to the experience of programmers/designers in working with a proramming language representation. It is also partly due to the fact that it is difficult to choose the decisions to be made and that they typically apply to large segments of the design. Generation of representations in both, of course, is generally quite easy since the syntax is not complicated. The mismatch again detracts from the overall performance.

Cost to Use (10/10/10)

Neither form requires great effort to use or maintain and thus both are cost-effective.

3.2.5 Discussion Of Representational Effectiveness -

A fundamental property of any representation is its ability to represent items of interest. An equation cannot represent our feelings toward a situation while poetry cannot represent (precisely) the flow of fluid in a pipe.

Beyond the simple binary question of capability of a representation, is the more difficult question of relative ability. A decision statement may represent the structure of a system, at least implicitly, but how well does it do this? Even more difficult is the case where we have two competing representations: for example, does a flow chart or metacode better represent the control structure of a system?

In Table 2 we have tried to provide ~~provided~~ a set of subjective discriminations. For seven information elements (decisions, relations, etc.) that we used in our design, we have indicated, on a scale of 0 to 10, how well they could be represented in each of the two representational forms we are analyzing. The different information elements were not equally distributed throughout the design, of course; but, we have evaluated the effectiveness of both representational forms for all information elements in order to consider their relative merits.

Decisions (10/4)*

Decisions are easily represented by decision statements, by

* (10/4) are the ratings given decision statements and metacode respectively on their ability to represent decisions.

definition! While it is clear that metacode (indeed, any representation) encodes a set of decisions, they are not so easily seen. Further, while it is easy to represent a local decision in metacode (e.g. "the test for file ownership will be performed after the test for existence of the file") the representation of a global decision (e.g. "file ownership will always be checked before performing any operation") will be distributed across a large amount of metacode, but would be represented in a single decision statement.

Relations (3/10)

Relations between parts of the system are not easily representable by decision statements. We typically would have to write out a decision statement for each possible relation. Further, the mode of usage of decision statements (deciding the form or function of some part of a system) does not encourage expressing relations except in cases where that relationship is of major importance.

In metacode, the relationship between parts is expressed textually, by simply naming the object (through subroutine calls and use of variables). This makes it easy to express relationships, although they may not be as visible as in a structure chart.

Constraints (8/3)

Constraints are quite easy to express using decision statements. For example, we might state as a decision "only the owner of a file may alter its protection code". Constraints on the programs themselves can be easily expressed in some instances (e.g. "any program that accesses a file must provide an accurate error return indicating the results of the access") but in other instances may be difficult (e.g. we might want to express that some data could be used globally and other local data).

In metacode, constraints can only be expressed implicitly (by not doing something or doing it in a particular way -- in either case, we have to infer the constraint) or in the form of declarations about data structures.

Open Questions (0/3)

During any design activity we form questions that are not immediately answered; that is, they are left "open." We must represent these open questions so that we can come back to them at a later stage and determine whether we then have sufficient information to answer them. As we have defined decision statements, they cannot represent open questions. However, a trivial expansion of the representational form to include decisions about the design process would remedy this.

Metacode also is not well-suited in general to representing open questions. It is, however, on a local basis since when a statement is made in metacode that is not operational (e.g. "get the next valid input") it is implicitly a statement of an open question that must be answered (e.g. "how are we to determine what the next valid input is?")

Requirements (0/0)

Requirements are very similar to open questions, but we have differentiated them to indicate that requirements are imposed from outside the design process while open questions arise because of the design actions being taken. (Clearly, an open question may turn out to be a requirement that we failed to state in the first place; so, the distinction between them should not be carried to any great length). Again, decision statements, by definition, are not able to state requirements -- that must be done using some other (although perhaps syntactically similar) representation. In this case, metacode also is not able to express requirements per se, although one might use metacode to express a process-oriented requirement.

Consideration of both these information elements -- open questions and requirements -- points up a very important issue often overlooked in design. We must talk about the design processs itself, not just the final product. We uncover questions to which we must find answers, discover relationships between constraints, make decisions about the allocation of our design resources and so on. This information, just as much as the information that goes to make up the final design, must be represented so that we can work with it in an orderly manner.

Functions (10/8)

The last two elements we consider are the basic elements that make up any design: functions -- expressions of what something is to do, and structures -- the objects that will provide the needed functions. Functions are easily representable by decision statements and metacode, although in metacode the extent of a function is more restricted in scope. For example, we can state in a decision statement that "all routines in the file system will provide complete error checking of all inputs", while in metacode this same functional description would have to be placed in each routine by showing a call on an error checking routine.

Structures (2/10)

In the case of structures -- that is, programs and data -- decision statements are not very useful for describing the programs themselves. We can state that a certain structure is to be used (e.g. "the directory lookup will use a linear search algorithm") and describe its properties (e.g. "the linear search will compare only the first 6 characters of a file name.") but it is difficult to express the exact structure of the program in question (e.g. consider describing the complete control flow of the directory lookup module using decision statements).

Metacode, on the other hand, is ideal for describing program structures in detail. The metacode we used had no special provisions for describing data, a deficiency which could be easily remedied.

3.3 IMPACT OF REPRESENTATION ON THE DESIGN PROCESS

Thus far we have analyzed our representation along several partial dimensions. We now wish to look at how the representations affected the design process overall. For our purposes here, we consider the design process as being composed of two activities: elaboration and review.

3.3.1 Effect On Elaboration -

We must differentiate between elaboration between stages using the *same* form and between stages using *different* forms. Between stages using decision statements, the form of the representation clearly made elaboration much easier: We simply took each decision and considered all the more detailed decisions implied by it. We did not need to spend much effort deciding what problem to work on next. The decision statements organized the design activities into discrete, rather local areas of concern. Further, they made clear the actual steps necessary to elaborate the design: take whatever decisions were implied or enabled by the higher-level decision being refined.

As noted above, difficulty was encountered in going from decision statements to metacode. The result was the "mush" of Stage 4. The primary problem seems to have been that the decision statements do not do a good job of representing the structure of the developing system, either in indicating the clumping of functions into discrete packages (modules) or by indicating the interconnections between parts.

Elaboration between stages using metacode was also quite easy for the same reasons. (There was some metacode used in Stage 4 which was elaborated in Stage 5). While we had only one stage completely repressented in metacode, the next stage (not considered here) which generated assembly code from the metacode was extremely easy*. This appears to be

*Our experience in this respect confirms that reported by others [Caine and Gordon, 1975]. Coding went extremely fast and few logic errors were found. It was not uncommon for a single programmer to code up and check out (in stand-alone) tests up to 100 assembly language instructions per day. Unfortunately, we do not have available precise statistics on this stage of the project.

due to the fact that the representational forms (metacode and assembly language) are both programming languages in their structure. One of the most important things we learned after the fact is that we should have had another level using metacode, but somewhat higher than Stage 5, which could have then been refined; this would have gone a long way toward reducing the mush encountered in Stage 4.

3.3.2 Effect On Review -

When reviewing a design, one is basically just asking whether the right decisions have been made or not. Decision statements are a great aid to review since they clearly present the decisions that have been made. Their lack of completeness, especially in expressing the structure of the system limits their effectiveness, however. In other words, they make a part of the review process very easy, but hinder another part -- the review of structure.

The effect of metacode on review is also ambiguous. On the one hand, because metacode displays the logical structure of programs in a way that makes it easier to understand than a lower-level representation, program review is enhanced. On the other hand, at the level we used it, metacode incorporates a good deal of detail, much of it unrelated to the information which must be reviewed. This makes review more difficult. Obviously, if we had used metacode more consistently at a less-detailed level, some of this problem would have gone away. The poor mismatch between the two forms, made review of the design as it developed at that stage much more difficult than it should have been. We will treat this situation in more detail in later sections.

3.4 IMPACT OF THE DESIGN PROCESS ON THE REPRESENTATION

The impact in this direction is not as easy to analyze. For one thing, the representation was not a free variable -- we had chosen the representational forms before starting and did our best to stick with them.

If you look carefully at what we were doing in Stages 2 and 3, however, you will see that even if we had not pre-chosen decision statements for those stages, what we were doing would have forced us into their use (or something similar). At that point, the design process was ranging widely over a number of issues, trying to determine what should be done and how to do it. It was not a highly organized sequence of design actions and thus needed the loose representation offered by decision statements.

The "mush" of Stage 4 is clearly a result of the design process we were using at that point. We were trying to consolidate our earlier decisions into clumps that could be handled later as individual design problems in the form of modules. Although we set out trying to use metacode at that stage, we failed because the design actions we were taking did not conveniently fall into that form.

By Stage 5, we had consolidated the design into separable units corresponding to modules. Then, the representation afforded by metacode was the natural one to use. The only noticeable impact at that stage was that the metacode became more detailed than we had planned because the design activity itself was operating at that degree of specificity.

4. IMPROVEMENTS

In this part we will address the problem of how to improve design representations from three standpoints. First, we will summarize what we have learned about the needed forms of representation from the analysis above. Second, we will provide some practical suggestions for improving the use of the particular representations analyzed in this paper. Third, we will suggest some needed long-range developments.

It is well known that representations effect problem-solving performance and design problem-solving is no exception. The case study we have analyzed here is an example of a general situation -- the representations we have for software design are not sufficient. The issue of interest to a computer scientist is: "What are the characteristics of an effective design representation?" Needless to say, software engineers will be interested in the results of feasible answers to that question. The more immediate issue of interest to software engineers is: "What can be done to improve design representations like those analyzed in this paper?" We will provide some initial answers in the following sections.

4.1 CHARACTERISTICS OF AN EFFECTIVE DESIGN REPRESENTATION

The analysis above is not extensive enough, nor is the data rich enough, to permit a definitive answer to the question stated above. What we have learned from our analysis however is this:

Metacode by itself is not sufficient

Metacode is an excellent representation at the level of design where we used it. While more abstract statements make it useful at higher levels where many decisions have not yet been made, it is not appropriate for the highest levels where it is not yet known what programs there will be and many of the decisions being made are still at the functional level. Some representational form other than program representation is needed at this highest stage of formative design where the basic structure of the system and its components is being decided. These higher levels that we have called design may more appropriately be labeled specification. While better representations are being developed for specification of software at various levels [Bell and Bixler, 1976; Robinson, 1976] they may not be completely suitable for representing the tentative decisions and design activities that still must take place when developing more precise specifications.

Uniformity

A representational form capable of representing a design from the earliest formative stage down to exact specifications from which actual programs can be constructed easily and unambiguously would be a powerful tool. We tried decision statements for the less structured stages of our project where metacode was not sufficient. While it worked reasonably well, the gap between it and metacode caused problems that a uniform representation could have avoided. There seem to be three possibilities:

- Use metacode throughout since it works so well at the detailed end of the spectrum.
- Find a representation that works well at high levels which can be elaborated to a point detailed enough to permit easy generation of code.
- Find different representations for both ends of the spectrum that interface better than do programing language and natural language formats.

We have already indicated our doubts about the first alternative. The second alternative involves two things: 1) finding a representational form that fits two rather different situations -- unstructured general decisions and structured, program-specific decisions; 2) raising the level at which we can produce code unambiguously. It is important to note that the second thing has been done before, when higher-level languages were developed. Current efforts to rationalize large areas of program content (such as has been done in the case of mathematical and statistical packages) offer a good chance of this happening through the packaging of software components which can then be designated by language forms above the level of current high-level languages.

Uniformity requires something analogous to the graphical representations used so successfully by the designers of physical objects. Yet, even here, we should note that different forms (renderings, blueprints, wiring diagrams) are used for different stages and different aspects of the design.

Further, that an architect starts a rendering knowing what the primary functions and physical constraints are for the building being designed. This permits him or her to begin working immediately with the structure of the design (generated by a knowledge of the general structural type required for a given function)*. As we build a richer

*Creative architects devise new and interesting designs, of course, precisely by not following the old pattern. Instead, they search for new structures that supply the needed functions or even question whether the stated functions are really those that are needed. Creative design of this type is often needed, but in many other cases it is not. If a designer tries to be super-creative in a situation that does not need it, the customer pays needlessly.

repertory of software designs and as we learn to specify better in advance what a system is to do rather than figuring it out during design, we may be able to start software designs at a lower level than is often the case now.

As a final comment on alternative 2, we suspect it may be a graphical form of representation, suitably tailored to the linear-text format of programming languages. One of the main things missing from our use of decision statements was the ability to express relationships, something that a graphical form does well.

The third alternative is probably the most realistic. Most of the comments made in connection with alternative 2 apply here as well.

Conformity to the Design Process

A representation should not drive the design process unduly*. The representation should instead conform to the design process so that it captures the information being generated easily and in a natural manner. Any design representation must allow easy refinement of the design as

improperly
excessively

* However, we may choose a representation in some instances precisely because it will force us to use a desirable set of design techniques.

well facilitate backup to previous design states when that is necessary. It must also permit easy comparison to constraints.

All Those Good Things

Any representation should be clear, easy to use, complete, accurate and so on. To be useful a design representation must be understandable by humans. In particular, any new representation should be based on an understanding of how people design, choose problemss, propose solutions, and so on. This is not to say that the representation should not be machineable. Indeed, it should be so that the mechanical details can be handled easily Further, as our understanding of the design process increases then we will be able to increase our augmentation of the design proccess. For representations that are to be used primarily by humans, however, the emphasis must be on making them suitable for human use. We have assumed the reader is aware of these human factors considerations and, thus, have not stressed them here.

4.2 IMMEDIATE IMPROVEMENTS

While development of an improved representation may take some time, software designers are still faced with the problem of representing their designs. On the basis of our experience with the RTE and other projects and the analysis we have carried out, we make the following suggestions:

Use Metacode

It provided us a very good working medium and significantly facilitated coding. There are two extremes in the statements used in metacode -- completely standardized and completely up to the individual. Our metacode usage on the RTE project was probably too free-form. Overstandardization must be guarded against, however.

View the Problem Correctly

We started this paper by arguing that what one is doing is developing a representation, not just documentation. While this may seem a subtle distinction, it is clear that viewing one's task properly significantly affects success. Thus, we suggest that you keep in mind when planning a design project that you must develop a representation of the object, not just document your design.

Add to

Augment Decision Statements

Decision statements appear to be useful for capturing much of what is going on at high levels of design. As we discovered, however, it is necessary to augment them with structure charts and a certain amount of prose to capture the information missed by the decision statements.

Control Uniformity of Level

Concentrating decisions about particular structures and considering decisions at the same level at the same time helps to work out the relations between parts and to develop all that is needed for a given item.

4.3 NEEDED LONG-RANGE DEVELOPMENTS

Our study of this case and our other investidations in the area of software design make it clear to us that there are a number of long-range developments that are needed. Among these are the three general areas outlined below:

Increased Attention to Representation

We feel that the importance of representation has been underrated, or at least unrecognized, in the area of software design. Indeed, it has only been in the past few years that a genuine recognition has taken place of the role that programming languages play. We now recognize (at least an increasing nummber of people do) that our choice of language structures may play an important role in areas of program development other than just simply speed of coding.

As the focus of much of the effort of creating software systems moves up into the design area, we must learn to pay attention to the representation that is used from the earliest stages of a project. Just as mathematics is an indispensible representational tool to the engineer in other areas, we must develop appropriate representational forms for use in all stages of software development.

Multiple Representation

One of the clearest lessons we have learned is that in general a single representation is not sufficient for the range of tasks encountered in creating software. The obvious implication is that we must develop representations for different parts of the task and for different purposes. Further, we believe that as our understanding of reliability, protection, modularity and other software qualities improves, we will find that we must have representations that emphasize these aspects of a system (just as a plumbing map serves a specialized function for an architect).

Not only must we develop specialized representations, but we develop representations and techniques that tie together into a coherent set the more specialized forms. But, we must be sure to view this as a long-range task. We should not try to develop perfect representations immediately, but rather do the best we can, try the results, and gradually evolve something better.

Increased Experimentation and Analysis

Subjective analyses and *ad hoc* development of representations such as ours have their place, but it is clear that in many areas our progress will be greatly enhanced by more rigorous analysis and experimentation. Techniques exist for evaluating the human factors aspects of technology in other areas that can be adapted for use here. While such techniques do not guarantee better results, and, indeed, can lead to worse results if improperly applied, our current approaches to analyzing, understanding, and improving our technological tools are woefully inadequate. Improvement in this area is essential for the long-term success of our endeavors.

5.0 CONCLUSION

In this paper and its companion [Freeman, 1976a] we have presented and analyzed the representation used in the design of a particular system. Our sole purpose has been to learn from this experience what we could about the form and use of design representation and to present our analysis of the situation so that others can learn from it, too.

We have presented our technical conclusions in several places above. The only thing we would add here is that it is our belief that there is a strong need for other studies that add to our knowledge of the tools and techniques of software creation.

6.0 ACKNOWLEDGEMENTS

I am indebted to the members of the Szamok Software Development Department for their patience in learning and using the representational forms discussed above: Zoltan Szekely, Andras Kelen, Sandor Majoros, Sandor Zold, and Mary Koos-Hutas. Comments from several colleagues and the typing assitance of Shirley Rasmussen are also gratefully acknowledged.

7.0 REFERENCES

1. Thomas E. Bell and David C. Bixler. "A Flow Oriented Requirements Statement Language," Proc. 1976 MRI Symposium, held at Brooklyn Polytechnic Institute, April, 1976.

2. Stephen H. Caine and E. Kent Gordon. "PDL - A Tool for Software Design." Proc. 1975 NCC.

3. F. DeRemer and H. Kron. "PRogramming-in-the-Large versus Programming-in-the-Small," Proc. 1975 International Conference on Reliable Software, IEEE Press.

4. Peter Freeman. "Toward Improved Review of Software Designs," Proc. 1975 NCC, AFIPS Press.

5. Peter Freeman. "Software Design Representation: A Case Study of the RTE Project," TR 80, ICS Dept. U C Irvine, 1976.

6. Peter Freeman. "Software Reliability and Design: A Survey," Proc. 1976 Design Automation Conference, IEEE Press.

7. Henry C. Lucas. Toward Creative System Design, Columbia University Press, 1974.

8. Clement C. McGowan and John Kelly. Top-Down Structured Programming Techniques, Petrocelli/Charter, 1975.

9. Lawrence J. Peters and Leonard L. Tripp. "Design Representation Schemes," Proc. 1976 MRI Symposium, held at Brooklyn Polytechnic Institute, April 1976.

10. Larry Robinson. "Specification Techniques for Software Reliability," Proc. 1976 Design Automation Conference, IEEE Press.

11. Jean Sammet. Programming Languages: History and Fundamentals. Prentice-Hall, 1971.

12. W.P. Stevens, G.J. Myers, and L.L. Constantine. "Structured Design," *IBM Systems Journal* 13,2, 1974.

13. Edward Yourdon. *Techniques of Program Structure and Design*. Prentice-Hall, 1975.