A NEW PARADIGM FOR SOFTWARE AND ITS DEVELOPMENT

Anthony O. Putman and H. Joel Jeffrey

ABSTRACT

This paper presents a new paradigm for computer software and its development. It includes a new concept of software, a new methodology, and a radically different end product. The paradigm is to treat the software as a person engaged in the social practices of a Community. The social practice description, an extension of the basic process unit, is used to completely describe all that the software does and how it does it, until reaching an action that can be done with a small, easily written program comparable to a single skill. We have written an executive program, which selects and carries out the appropriate version of a social practice, using the description of the practice. The executive program works for any set of social practice descriptions; it is not rewritten for new software. In the traditional paradigm, one produces requirements and design and then writes the software. In the new paradigm, the requirements and design (in social practice description form), plus the skill programs, *are* the software. The new paradigm has been successfully used in two applications. It appears to be much more effective in building software, and particularly well suited for producing programs that engage in specifically human practices, such as understanding natural language and analyzing real world knowledge.

Advances in Descriptive Psychology, Volume 4, pages 119–138.
Editors: Keith E. Davis and Thomas O. Mitchell.

ISBN: 0-89232-358-2.

The development of computer software has become an enterprise of very substantial scale and increasing importance in today's world. Computers have permeated every aspect of our lives in ways that were quite literally unthinkable just 30 years ago; the explosive and continuing advances that have been made in computer technology in this brief span of time constitutes a technological achievement that may well have no parallel in recorded history. It is widely recognized, however, that the software required to utilize these computers to best advantage has not kept pace with the hardware; indeed, a "software gap" of enormous and growing proportions is widely acknowledged to exist.

A close inspection of the state of the art reveals a further disparity between the development of hardware and software. The design and production of computer hardware has gone through several generations of development since the original ENIAC. Today's computers are enormously faster, smaller, more reliable, and cheaper to build than their predecessors of 30 years ago. Such advances are reflections of the enormously more sophisticated *design* and *production* methods of today's hardware engineers.

The design and production of software has of course not remained static. The development and widespread utilization of high level languages was a substantial advance over programming in machine language. More modern languages (e.g., Pascal) represent a further advance. Recent methods of design, such as Yourdon data flow methodology (Yourdon & Constantine, 1979), stepwise refinement, or Jackson design methodology (Jackson, 1975) have in many cases provided a marked improvement (Bergland, 1981). Further, a good deal of recent work in Computer Science has attempted to improve software production by allowing the programmer to express what is to be done in a form somewhat closer to ordinary language and, in some cases, by preventing the programmer from writing code that does not make sense in terms of the real world objects and actions the code represents (Brodie & Zilles, 1981).

In spite of these advances, software production stands in marked contrast to hardware production with respect to productivity gains. While hardware costs have been reduced exponentially over the past three decades, software costs over the same period have been reduced at best a few percent (Infotech, 1982). Whereas few, if any, software practitioners would care to give up modern software languages and techniques, equally few would claim that there has been progress comparable to that of hardware, or that there is any work extant which shows such promise. Software production today remains an extraordinarily difficult, complex task, highly resistant to the concerted efforts of a large number of talented researchers and practitioners.

We suggest that closing the software gap requires a fundamentally new paradigm for computer software and its development. This paper presents what we believe to be such a paradigm. It includes a new concept of what software is, a new methodology for developing software, and a radically different end product (which, nonetheless, runs on and controls computers as software today does). Developing software within this paradigm seems to hold promise of substantially reducing the time and effort to produce software. As will be discussed later in this paper, there are sound logical grounds for this claim. We acknowledge from the beginning, however, that such a claim can only be verified through substantial experience and actual practice. We hope that this paper will provide sufficient motivation and knowledge to enable interested software developers to accumulate such experience.

THE TRADITIONAL PARADIGM

The usual concept of software is that of a system (in the technical language of Descriptive Psychology, a Configuration; See Ossorio, 1971/1978b, pp. 54–56), with its own internal structure and logic, which interfaces with other systems or users (which have their own logic in turn). Each piece of software has its own logic, or sense of internal coherence, which is what makes it *that* piece of software and not any other. This fact, while seldom formally represented in accounts of software, is nonetheless one of the central distinguishing features of software and is widely, if informally, acknowledged throughout the industry. (It is further a fact known to every programmer, and decried by every programming manager, that each piece of software's logic is extremely seductive.)

Consider, for example, the following part of a design (a close paraphrase of an actual design document written by one of the authors):

> The translator scans the source file, recognizing a "DICT" or "dom" statement. When either is encountered, the information for the item is stored in an internal table, of the form
>
> name dict-flag dom-flag
>
> When the structure items are encountered, the translator checks each item name in the table, issuing an error message if the "dict-flag" or "dom-flag" is false.

Constituent processes mentioned in this design are "scanning the source file," "storing the information for an item in an internal table," "looking up a name in the table," and "issuing an error message." The internal table is an object component; it has constituents "name," "dict-flag," and "dom-flag."

The technical device in Descriptive Psychology for representing such a combination of objects and processes and their relations is the state of affairs unit (Ossorio, 1978c). It is the division into immediate constituent objects and processes, and their relations, that define a given configuration; further, the choices made in making these divisions are what cumulatively generate the logic of the software.

Thus, when designing software within the traditional paradigm, the fundamental questions are (a) what will the internal logic of the system be, and (b) how will the system interface with other users and/or systems. In answering these questions, the traditional software designer basically is concerned with inputs, operations, and outputs. In other words, the designer views the software as a mechanistic, causal-deterministic system (or, to be less precise but more clear, a machine).

(At this point, we expect that many readers will object: ''But what's wrong with that? That's exactly what a computer is—a mechanistic, causal deterministic system, or if you prefer, a machine.'' We do not view this traditional paradigm as unreasonable or wrong: we merely view it as having certain problems and limitations that the new one does not.)

The traditional view of the social practices known as ''software development'' can be broken down into three stages: (a) develop the requirements for the software; (b) design the software; and (c) implement the software (Horowitz, 1975; Jensen & Tonies, 1979). It is widely agreed in the computer software field that a unified approach to these three stages is needed, such that the outcome of the requirement stage is immediately useful in the design stage and the outcome of the design stage is immediately useful in the implementation stage (Brodie & Zilles, 1981; Infotech, 1982). Such a unified approach may exist in theory, but it is virtually never seen in practice. There have been many attempts at the unified approach, including requirement techniques and languages, design languages, and myriads of implementation languages, all of varying degrees of usefulness and complexity. Approaches include formalism (Gries, 1981; Wulf, Hilfinger, & Flon, 1981), predicate calculus (Kowalski, 1979), and many program design methodologies (Bergland, 1981). The field is in fact quite broad and active today (Brodie & Zilles, 1981; Infotech, 1982). None, however, has succeeded at being the unified approach; indeed, the lack of such success can be taken as both a fundamental shortcoming of the current paradigm of software development and a standard of adequacy of any new paradigm which claims to replace it.

We suggest that the root of the loose connection among the tasks of software development (requirements, design, implementation) lies in the traditional concept of what software is. Specifically, it lies in there being a division between the internal logic of the system and its interface with other systems. In writing requirements, one is specifying what the system will do, from the outside. In designing, one is dividing the system into pieces and defining their interactions. The classic phrasing, constantly encountered in computer science literature and textbooks, is "Requirements state *what* the system will do; design states *how* it will do it."

The fragment of translator design at the beginning of this section is representative; it tells how the translator will do a certain task. The requirements for that task are:

> The translator will check that each field in the structure has a "DICT" and a "dom" statement preceding the structure itself, and issue an error message for any field that does not.

One can, and typically does, repeat this breakdown, until one reaches a small enough piece that one can write the code for the piece straightforwardly. At that point, one can "implement" or "code" the software—that is, produce the actual code, in a language the computer can process, to carry out the task given in the requirements.

Using the technical language of Descriptive Psychology, we can see that in the case of requirements one is defining (or describing) social practices; in design one is, at best, giving the social practices of a different community, and more typically is giving state of affairs or basic object unit descriptions (Ossorio, 1978c). Implementation is even more divorced from the other stages, as it consists in giving purely performative descriptions, with no framework within which to state the relationship between these descriptions and the social practices described in the requirements and design.

It is this lack of a complete, coherent framework that is critical. One can, and in fact often does, discuss the relation between the requirements, design, and code, but the language (the locutions, concepts, and behaviors) for discussing each are distinct, with no common ground (other than ordinary language). The result is that developing requirements is a separate enterprise from developing designs, and each is separate from the third enterprise, coding, in which one actually produces the software.

In a nutshell, we can characterize the traditional concept of software as drawing its boundary around the system being built. Having made that initial move, one then defines, or refines, one's descriptions of the parts of the software and their interactions, and the interfaces with other systems or users. It is precisely this initial move that the new paradigm makes differently; having made a different initial move, what follows is not merely different, but in some cases radically so.

THE NEW PARADIGM

The paradigm we are presenting is a new concept of software and its development. It includes both a language and a methodology. It is a coherent language such that requirement specification leads directly to, and is part of, design, which leads directly to, and is part of, implementation. The methodology is a fully worked out, implementable method for developing software within this new paradigm. The advantages of working within this new paradigm stem from having software development to be one enterprise rather than several separate ones, as is now the case.

We have seen that the traditional paradigm results from drawing the boundary around the software itself. When this is done, the result is what might appropriately be termed a "technical system." The new paradigm results from drawing the boundary in a different place: around the community in which the software has a place. This represents a direct acknowledgement that software is *used,* either by people or other software. There are people and other software objects that communicate and interact with the new software: The new paradigm draws the boundary to include the software, all the people who interact with it, and all other pieces of software that it communicates and interacts with. In other words, the new paradigm is to treat the software as a Person in a Community of other Persons.[1]

"Person" here is the technical concept of one who engages in the Practices of a Community. We do not mean to indicate that the software is to simulate human thinking or feeling or that it has attitudes, interests, and so on, but rather that the software engages in social practices and that the description of what the software does is contained in Social Practice descriptions. Social Practice descriptions thus become the language for specifying the software; as it turns out, they also serve as the language for designing and implementing the software.

Whereas the usual view of software yields a technical system, this one yields a *human* system.

The fundamental question for software production becomes, "What is this (software) person doing in this community, and how is it doing it?" rather than any question of internal state, memory contents, etc. As we shall see, the form of the answer to this question is the Social Practice description, as it is for a person of the more usual sort.

It may be useful to note an analogy between the concept of software we are introducing and attempts to define a person in psychology in general and Descriptive Psychology in particular. Defining a person has traditionally been done by referring to some part or structure that was considered to be essentially human, the possession of which defined the possessor as human. This is analogous to what we have noted as the traditional approach to software, in which what defines a particular piece of software is its internal structure and logic. In Descriptive Psychology, on the other hand, we note the (logical) fact that what makes an object a person is the place it has in the practices of the human community. Specifically, a person is someone whose behavior is paradigmatically Deliberate Action (Ossorio, 1978a). We are introducing a parallel distinction for software: What defines a piece of software is what it does, i.e., the practices it engages in, not its internal structure.

To sum up, the new paradigm consists of viewing software as part of a human system, and acting on that concept with a set of technically useful concepts and practices, the outcome of which is a new form of software. Let us now examine these technical elaborations and their pragmatic implications.

A UNIFYING LANGUAGE FOR SOFTWARE DEVELOPMENT

In the new paradigm, the basic move is to ask, "What practices does this software have a place in?" (a more technical rendering of, "What does it do?"). Asking this question, per se, is hardly new; it is the very question that leads to traditional requirements, design, and implementation, especially when coupled with its natural counterpart, "How does it do it?"

What is new is the form of the answer to the question: Social Practice descriptions, as defined in Descriptive Psychology (Ossorio, 1981), in a technically elaborated format which is adequate for representing all of the facts about a Social Practice. The capability of giving this sort of answer, in a technically usable format, is the linchpin of the new paradigm.

Social Practices are, fundamentally, what people (human or software) *do*. A Social Practice is "a pattern of actions engaged in by one or more persons" (Ossorio, 1981). Everyday examples include (a) writing a paper for a technical journal, (b) dining, (c) negotiating, (d) writing information to a temporary file, (e) translating a program from a high-level language to machine code, and (f) finding the outgoing line for an incoming telephone call.

Examples (d), (e) and (f) illustrate two points. First, the actor need not be a human (an obvious point, but one which in ordinary usage we tend to pass over). Practice (d) could appropriately be seen as being done by a human or piece of software; practice (e) (commonly known as compiling) could be done by a human but is virtually always done by a compiler; practice (f), historically, used to be done by human telephone operators and now is almost always done by an electronic switching machine controlled by software.

The second point is that differing Practices constitute one of the aspects by which one distinguishes one Community from another, and that the concept of Social Practice is inextricably linked with the larger concept of Community (Putman, 1981). Just as not every person plays chess or runs marathons, not every person compiles programs or hooks telephone circuits together. Pragmatically, the persons who communicate and interact with the software to be built form the Members of the Community. This is the anchoring point in using the paradigm, for asking what Practices a piece of software has a place in, is to make use of what is called technically a Part Description (Ossorio, 1966); it is elliptic for, ''What Practices of which Communities does this software have a place in?'' Note that it is not at all uncommon, especially for software persons, for one to have one place in the Practices of one Community and another place in the Practices of another Community, with no larger community subsuming them both.

The nucleus of a Social Practice description is the specification of the Intentional Action parameters of the Practice, to wit:

W	the name of the State of Affairs desired
K	the distinctions one must make to engage in this Practice
KH	the skills one must have to engage in this Practice
P	the performance (i.e., observable episode) one engages in, in engaging in this Practice
A	the achievement, or outcome of engaging in this Practice
PC	personal characteristics that make a difference in this Practice
S	the Significance of this Practice; the larger Practice one is engaging in, by engaging in this Practice

A major portion of giving a Social Practice description is specifying the process aspect, the Performance. To do that, we have the basic process unit (BPU), as defined in Ossorio (1978c, pp. 41–51) and elaborated in Jeffrey and Putman (1983). The BPU, consisting of Stages, Elements, Individuals, Eligibilities, Contingencies, and Versions, codifies all of the process aspects of a Practice and the structure of which participants in the Practice may engage in each action in which way.

The use of the BPU as a notational device for representing, in a technically useful form, the information about the Performance of a Practice is discussed at some length in our report on the MENTOR project (Jeffrey & Putman, 1983). In order to use Social Practice descriptions technically, one must have a comparable representation for the remaining aspects of the Practice: the skills, knowledge, available performances, and the connections of the Practice to larger contexts. That representation format which had been developed, is the unifying language needed.

With this language, one answers the question, ''What practice does this software engage in?'' In any but the most trivial of Communities, a tremendous amount of detailed information must be given to describe the Practices at a technically useful level of detail. The Social Practice description (SPD) format allows one to specify that information, in a way that the logical connections between the items of data are preserved. (For example, the objects involved in a Practice, together with the information on which actual historical individuals may serve as each, is specified as part of the description of the Practice.) Further, one may specify the Practice at any level of detail desired, again preserving the appropriate logical connections between practices, subpractices, sub-subpractices, etc.

Social Practice Descriptions and Program Logic

Paradigmatically, one specifies what an ordinary human person is doing by reference to Social Practices. When one has given Social Practice descriptions for the Practices that some person engages in, one has specified everything that that person does, and all of the ways of doing each of the things, all of the conditions under which any possible optional things will be done, and which actual objects will fill which roles. This description is complete (at that level of detail), since the BPU is a codification of all of the facts about a process (Ossorio, 1978b) and the Social Practice is a specification of the action (Ossorio, 1978a).

In other words, all of the logic of what to do at any point, and what to use in doing it, is captured in the SPDs.

Programs, on the other hand, are traditionally viewed as nothing more than a set of instructions to be executed in some order. Each actual set of instructions executed is accomplishing a version of a process (or processes). The process is the Performance parameter of some Practice. In order to ensure that the execution sequences of the program correspond exactly to the Versions of the Practice, some of the statements in the program have the logical task of capturing the logical constraints of what may follow what, what to do under various circumstances, what items to use in doing the action, etc.—that is, the logical constraints represented in the Contingencies, Elements, Individuals, Eligibilities, and Versions of the basic process unit. In traditional software development, a great deal of effort and care goes into ensuring that only those sequences of instructions corresponding to the Version of the Practice appropriate to the circumstances will be executed.

Here is an example. The following function, written in the C language (except for the line numbers at the extreme left), searches an existing list of numbers for a new one. If it finds it, it returns the location of the item in the list; if not, it returns the value -1.

```
1                    char items[ 100 ] [ 20 ];
2                    int Nitems;
3
4 find( item )
5 char item[ ];
```

```
 6 {
 7             int i;
 8             int place;
 9
10             place = -1;
11             for( i = 0;
12                     i < Nitems;
13                             i++ ){
14                 if( strcmp( item, items[ i ] ) == 0 ){
15                         place = i;
16                         break;
17                 }
18             }
19             return( place );
20 }
```

(For expository purpose, we have written this program in a form that is correct but not compact.)

If the variable "Nitems" has the value 3, the possible sequences of statements that could be executed are:

```
10, 11, 12, 14, 15, 16, 19
10, 11, 12, 14, 13, 12, 14, 15, 16, 19
10, 11, 12, 14, 13, 12, 14, 13, 12, 14, 15, 16, 19
10, 11, 12, 14, 13, 12, 14, 13, 14, 13, 12, 19
```

corresponding to finding the item in the first, second, or third list position in the list, or not finding the item in the list at all.

The Stage-Options and Contingencies of a BPU description of this process are:

- NAME: Find finds the place of an item in a list

  Stages:

  1. Find searches for the new item in the list
     Option 1: Find finds the new item in the list
     Option 2: Find discovers that the new item is not in the list
  2. Find tells the caller the position of the new item in the list
  3. Find tells the caller that the new item is not in the list

  Contingencies:

  1. Stage 2 only if Stage 1-Option 1
  2. Stage 3 only if Stage 1-Option 2

In the example program, Lines 15 and 19 comprise the Performance of the Practice named in Stage 2; lines 10 and 19 comprise the Performance of the Practice of Stage 3; lines 11, 12, 13, and 18 control the repetition of Stages to produce the appropriate Version of Stage 1-Option 1; line 16 (which stops the repetition produced by the "for" statement on line 11) is an explicit example of a statement selecting a Version: when the item matches the list item (a Contingency), line 19 is to be done next.

Thus, the execution of a program may be seen as having two logical functions:

- Selecting the Version of the Practice to be done;
- Carrying out the selected Version.

In the new paradigm, we are capturing all of the logic of what the (software) person is to do in Social Practice descriptions; therefore, the code itself need contain none of it.

(At this point one may suspect that a program within the new paradigm is going to be substantially different from a traditional program. We shall see later that this suspicion is correct.)

Producing Software with Social Practice Descriptions

To produce software, one must design and implement it—that is, produce a program that can be executed by a computer. The key conceptual move here has been to note that Social Practice descriptions can be used for this purpose. We have noted that making technical use of this approach requires a technically elaborated format, or language, for giving those Social Practice descriptions, just as the BPU makes possible technical applications of the concept of a Process (Jeffrey & Putman, 1983).

The Social Practice representation format has been developed and is in use. Known as DIAMOND[2], it is essentially the extension to Social Practices of the BPU. DIAMOND serves as the specification, design, and implementation language for software. To specify a piece of software, one specifies the Practices within which it has a place, giving SPDs in the DIAMOND format. These Practices include, as we have noted before, all the persons, human and otherwise, who interact or communicate with this software, i.e., who engage in any Practice with it.

Having specified completely the Practices the software is to engage in, one then elaborates the specifications. This means simply breaking down each SPD into successively more detailed descriptions. Since one begins with the SPDs from the first stage, and adds to them further descriptions in the same format, the specification stage is in fact immediately useful in the design stage, and design is the same enterprise, involving the same concepts and the same practice as specification.

The breakdown of each description into more and more detailed SPDs continues until a point is reached at which one has a Performance that cannot be meaningfully broken down into behaviors. (Beyond this point, further breakdown would be giving movement descriptions, such as "First I moved my right arm six inches forward, then I moved my thumb and forefinger one inch apart.") The BPU can be used to continue the breakdown if there is a point in doing so. The choice to stop the breakdown process is almost entirely a pragmatic one (with perhaps some esthetic component). One stops when one can simply and straightforwardly write a program that does the Practice. Until this point, there is a meaningful answer to, "How does it do X?"; the SPD gives that answer. At this point, there is no "how"—that is, there is no answer in terms of "It does A, B, and C, and those things, in that order, are a version of X." Rather, the software person has a program such that the execution of the program is an instance of the Performance of X. Such a program may appropriately be viewed as a skill of the software person being constructed. Just as when discussing the behavior of the more usual sort of persons, there is a point beyond which one says, "There is no how; he just *knows how* to do it," so with software persons at such a point one says, "It simply *knows how* to do that; there is no other how."

The "find" program above is an example of a program at the bottom level of detail. A piece of software that included the Practice of finding an item in a list of items as a Stage of another Practice would find the item by executing "find." While one could give a BPU breakdown of finding an item in a list of items (as the Stage-Option breakdown illustrates), one would not ordinarily do so.

It may seem somewhat arbitrary to say that "specification" stops after the first description step. In fact, one might well raise the question of why we are distinguishing separate tasks here at all. This is a reasonable question, because one "stage" flows naturally into the next. There is no difference, logically, between the two. It would not be surprising if the specification-vs.-design distinction were to wither away in the future.

WHAT DOES SOFTWARE DONE THIS WAY LOOK LIKE?

It is commonplace, when one paradigm replaces another, for ordinary, everyday objects to change quite substantially, even to the degree that they may appear to have disappeared entirely, to be replaced with something entirely different but with the same name. (What does your digital quartz "watch" have in common with your grandfather's pocket "watch," other than both being used to tell the time?) We have seen that this is a genuinely new paradigm. It is not surprising, therefore, that the software produced is quite different.

Social Practice Descriptions and Program Logic Revisited

We have seen that the execution of a traditional program can be seen as having two (logically) separate tasks: selection of the Version of the Practice, and the carrying out of the selected Version. In traditional software these two tasks are very closely intertwined, with statements that are part of selecting a Version juxtaposed with those that are part of doing that version, and whose execution may come before or after those of the other type. Indeed, it is not uncommon to have statements that are doing both tasks. In the new paradigm, these tasks are accomplished in a different way.

Suppose we have a set of Social Practice descriptions, as discussed above. We can conceive of a program, an Executive, which operates as follows:

1. Knowing which Practice it is to carry out, the Executive selects, from the Version list in the description, the next Version. Since the Versions are a list of all of the possible ways this Practice can take place (Ossorio, 1978c), we are guaranteed that if this Practice can be done at all, that way of doing it will appear in the Versions (subject of course to limitations on the knowledge of the person who gave the SPD).
2. Using the Eligibilities, the Executive verifies that there is an Individual to instantiate each Element that appears in the Version.
3. The Executive verifies that any attributional constraints in the Contingencies are satisfied, checking the status of the state of affairs whose name appears in the contingency.
4. The outcome of these steps is a Version of the Practice that is appropriate to the persons (human or otherwise) and their eligibilities in the Practice, and the facts as they currently stand. The Executive now examines each Stage-Option in the Version, and either finds a Version of Stage-Option Practice, via the same steps (1) to (3), or notes that it has a program that is the skill for carrying out the Practice. It then carries out the Version, executing each program that comprises the skill by which it engages in each Practice.

Such an Executive program would, essentially, embody the logic of acting on Social Practice descriptions. It would operate independently of any particular Practice, engaging in any Practice by finding out how (using an SPD) or by having the relevant Know-how, a program.

Such an Executive is not merely conceivable. It has been written, and it works. It has been tested in an actual organizational setting, with SPDs describing the Practices of the organization, down to the level of issuing commands (including the actual commands themselves) to other software. The first version was the MENTOR program (Jeffrey & Putman, 1983). Further work has extended this version, particularly in the areas of adding skills for checking the status of states of affairs and for carrying out Practices. (In the tradition of "Boy Friday" and "Girl Friday," the second version has been christened "Thing Friday", nicknamed "Friday.")

Having been written once, there is no need to write a new executive for a new piece of software; for the new piece of software, one needs the Social Practice descriptions and the skills.

New Paradigm Software

Software produced within the new paradigm consists of:

1. The Social Practice descriptions that describe all of the practices this software person is to engage in. Since the descriptions include the information in the BPU for specifying the Performance parameter of the Practice, all of the information pertaining to how to do each Practice is contained in these descriptions.
2. Programs that provide the software person's skills, including the means for acquiring necessary knowledge (e.g., assigning appropriate status to the states of affairs involved).
3. The executive program, which does a major portion of the software's work, but which is content-free. (All of the content is in the SPD data base.)

An Example

Let us take another look at our translator example. Suppose one has a language, "R," which includes all of the forms of the C language as well as certain others: (a) a statement of the form "RID n" (where "n" is an integer greater than zero), (b) a statement of the form

```
RD {
    data item 1
    date item 2
           ...
    data item n
}
```

and (c) a statement of the form

```
REL name {
        name 1   number 1
        name 2   number 2
                   ...
        name k   number k
    }
```

Further, each element of the REL structure is required to have a "DICT" and "dom" declaration, preceding the REL structure. Programs in the R language are to be translated into programs in C, as follows: C statements remain unchanged; "RID n" is translated to "#define *name* n"; "REL *name* {" is translated to "struct *name* {". (This example, as before, is a slight modification of actual software written by one of the authors.)

The translator consists of the Social Practice descriptions and the skill programs (plus the executive, which does not change from one software person to the next), as follows (although we will only give the Stage-Options of certain Practices and one of the skills):

The requirements:

- Name: The translator translates an R program into a C program

  Stages:

  1. The translator reads a program in the R language
  2. The translator produces a program in the C language
  3. The translator produces a data file containing the information in the RD statement

The design (of Stage 2):

- Name: The translator produces a program in the C language

  Stages:

  1. The translator prints a C statement unchanged
  2. The translator produces a "#define *name*" statement
  3. The translator produces a "struct *name*" statement
  4. The translator checks that each element in the REL structure has a "DICT" and "dom" declaration

- Name: The translator checks that an element in the REL structure has a "DICT" and "dom" declaration

  Stages:

  1. The translator makes a list of all elements encountered before the REL statement
  2. The translator checks each element in the REL structure against the list of elements

- Name: The translator checks each element in the REL structure against the list of elements

  Stages:

  1. The translator reads the element name from the source file line
  2. The translator looks up the element name in the list *by* itemno = find (name);
  3. The translator issues an error message for an element

Stage 2 is at the bottom level of detail. The italicized "by" in its Name indicates a specialized form of behavior description, a procedure description, in which one gives a different name to the Performance parameter. The way the translator does this Stage is by executing "find," the skill for engaging in this Practice.

Any one familiar with software as it is traditionally done will note that this bears only slight resemblance to software as it has existed. The most direct parallel to traditional software is the skills. The Social Practice data base contains all of the logic of the practices the software engages in i.e., what it is to do. From a traditional perspective, new paradigm software appears to be a combination of requirements, several levels of design, and a collection of what are usually called "utility routines."

The central point of the new paradigm is not that one is proceeding in a top-down fashion, nor even that the language (Social Practice descriptions in a form comparable to the BPU for Process Descriptions) is different. Rather, it is:

> In the old paradigm, one produces requirements and design, and then writes the software. In the new paradigm, the requirements and design, plus the small skill programs, *are* the software

Those familiar with programming language and operating system research in computer science may note that the executive program and Social Practice description language can be seen, from that perspective, as the long-sought universal operating system and programming language, respectively. Such a perspective may shed some light on why these universals have been so difficult to achieve: They (logically) require an adequate conceptualization of action, an adequate language for representing actions, and an appropriate embodiment of the logic of acting on the action descriptions.

An Analogy

One can draw an analogy here which may be useful in understanding the relationship between traditional software and new-paradigm software. The executive program can be compared to a variable speed electric motor, which can supply the motive force for a great variety of machines simply by being positioned appropriately with the other equipment. With such a motor, an equipment designer need pay no attention to how to power a refrigerator, a watch, an elevator, etc. He need only make sure that the equipment being designed and built allows the motor, which he takes off the shelf, to be put in. Traditional software development is like equipment development in which one has to design and build a specialized motor for every application.

Of course, such a motor does not make equipment design and construction trivial. One still has to supply the proper connections for it and connect the drive shaft appropriately. Similarly, the SPD data base has to be produced, and the skill programs written, some of which will undoubtedly be difficult and sophisticated. It could turn out that doing this is as difficult and time consuming as traditional software production. But such a result would be quite surprising for, as we have seen, much of the work of the software is now being done by the executive program, which is not rewritten but simply "plugged in" to the new SPDs and skills that comprise a new piece of software. Further, the logic of what the software is doing is contained in a format, the Social Practice description, which was designed for the purpose of representing the actual logical structure of actions, and which has been demonstrated (Jeffrey & Putman, 1983) to be a powerful and compact representation methodology.

METHODOLOGY

To build software using the new paradigm, one produces a complete specification of the human system (the Community) of which it is a Member. Step by step, the way in which one does that is:

1. Identify all of the users of the software, human and otherwise. (This is specifying the Members of the Community.)

2. Specify the interactions the Members will have with the software. This includes specifying what they are doing, what the software is doing, and the role (or status) the software plays in these interactions. The specifications are given in action terms, not metaphor, abstraction, or purely nominal language (e.g., not "passes data to X").

For example, "A manager gets a copy of the Department budget" is the name of an action; it is intelligible as a Social Practice as are "An employee fills out an expense account statement," "the accounting system sends a copy of the statement to the budget maintenance program" or "the expense account verification program checks an expense account statement for errors." On the other hand, "The accounting system sends information to the budget maintenance program" is not intelligible as a Social Practice, much as "Gil told Will something" is not. Neither is "The expense account verification program processes an expense account form."

3. This task continues until the appropriate people (the software's designers, appropriate managers, users, etc.) agree that the specification is complete—i.e., that these are all of the Practices the software has a place in.

4. For each of the Practices, ask

- What knowledge (facts or discriminations of states of affairs) is required to engage in this Practice?
- What skills are required to engage in this Practice?
- What performance is needed to engage in this Practice?
- What actions does the software engage in to get each item of knowledge?

5. Given these knowledge and skills, can the Executive straightforwardly do the Practice, with (at most) a small, obvious piece of code? If not, then this Practice is decomposed into other, smaller Practices (with the SPD format), and the process repeated.

The outcome of this procedure is a set of Social Practice descriptions that, together with whatever additional skill programs are needed, constitute the new software.

NOW WHAT?

The obvious next step is to carry out the methodology given in the previous section with more pieces of real software. We have utilized this methodology twice in the afore-mentioned MENTOR and Friday projects to excellent effect; now it needs to be used for more traditional (software) applications. We believe it is clear that there is good reason to expect the new paradigm to be substantially more effective in building software, but this expectation must be tested.

Assuming that the outcome of such tests is positive, what benefits might be expected to come from using this paradigm (other than economic ones)? We have two hypotheses:

First, it seems to us that this paradigm would make possible software that is larger and more complex than what can be written within the traditional paradigm. For example, it is not possible today to write a program of one billion lines of code (that works properly).

Second, the new paradigm seems to us particularly appropriate for programs that engage in specifically human practices. Natural language understanding, automatic fact analysis (Ossorio, 1978b), and any task in which understanding an actual human is paramount, such as counseling or psychotherapy, are examples. In this way, it may become possible to achieve a long-anticipated goal of computing: the creation of an artificial intelligence device that can appropriately be said to simulate the functions of a (non-software) person.

ACKNOWLEDGMENT

Portions of the material contained in this chapter were included in a paper presented by the second author at the conference of the Society for Descriptive Psychology in Boulder, Colorado, August, 1983. The title of the paper was, "A New Paradigm for Software Engineering." Authors' addresses: H. Joel Jeffrey, 606 S. Washington, Wheaton, IL 60187; Anthony O. Putman, Descriptive Systems, 1019 Baldwin Avenue, Ann Arbor, MI 48104.

NOTES

1. Here, and throughout this paper, we have used capitalization to indicate a technical term from Descriptive Psychology, in hopes of making the paper more accessible to those outside that field.
2. DIAMOND is a proprietary product of Descriptive Systems, Inc.

REFERENCES

Berland, G. D. (1981, October). A guided tour of program design methodologies. *Computer, 14* (4), 13–37.

Brodie, M. L., & Zilles, S. N. (Eds.) (1981, January). *Proceedings of the Workshop on Data Abstraction, Databases, and Conceptual Modeling, 16* (1, Special Issue). (Association for Computing Machinery SIGPAN [Special Interest Group on Programming Languages].)

Gries, D. (1981). *The science of programming*. New York: Springer-Verlag.

Horowitz, E. (Ed.) (1975). *Practical strategies for developing large software systems*. Reading, MA: Addison-Wesley.

Infotech State of the Art Report on Programming Technology (1982). Maidenhead, Berkshire, England: Pergamon Infotech Ltd.

Jackson, M. A. (1975). *Principles of program design*. New York: Academic Press.

Jensen, R. W., & Tonies, C. C. (1979). *Software engineering*. Englewood Cliffs, N.J.: Prentice-Hall.

Jeffrey, H. J., and Putman, A. O. (1983). The MENTOR project: Replicating the functioning of an organization. In K.E. Davis and R. Bergner (Eds.), *Advances in Descriptive Psychology* (Vol. 3, pp. 243–270) Greenwich, CT: JAI Press.

Kowalski, R. (1979). *Logic for problem solving* (Artifical Intelligence Series 7.) New York: Elsevier-North Holland.

Ossorio, P. G. (1966). *Persons* (LRI Report No. 3). Los Angeles, CA and Boulder, CO: Linguistic Research Institute.

Ossorio, P. G. (1978a). *Meaning and symbolism* (LRI Report No. 15). Whittier, CA and Boulder, CO: Linguistic Research Institute. (Originally published in 1969 as LRI Report No. 10. Boulder, CO: Linguistic Research Institute.)

Ossorio, P. G. (1978b). *State of affairs systems* (LRI Report No. 14). Whittier, CA and Boulder, CO: Linguistic Research Institute. (Originally published in 1971 as RADC-TR–71–102, Rome Air Development Center, Rome, New York.)

Ossorio, P. G. (1978c). *"What actually happens"*. Columbia: University of South Carolina Press. (Originally published in an earlier version in 1971 as LRI Report No. 10a. Whittier, CA and Boulder, CO: Linguistic Research Institute. Later listed as LRI Report No. 20.)

Ossorio, P. G. (1981). Notes on behavior description. In K. E. Davis (Ed.), *Advances in Descriptive Psychology* (Vol. 1, pp. 13–36). Greenwich, CT: JAI Press. (Originally published in 1969 as LRI Report No. 4b. Los Angeles and Boulder, CO: Linguistic Research Institute.)

Putman, A. O. (1981). *Communities.* In K. E. Davis (Ed.), *Advances in Descriptive Psychology* (Vol. 1, pp. 195–210). Greenwich, CT: JAI Press.

Wulf, W. A., Shaw, M., Hilfinger, P. N., & Flon, L. (1981). *Fundamental structures of computer science.* Reading, MA: Addison-Wesley.

Yourdon, E., & Constantine, L. L. (1979). *Structured design.* Englewood Cliffs, N.J.: Prentice-Hall.