HUMAN SYSTEMS ISSUES IN SOFTWARE ENGINEERING

H. Paul Zeiger

ABSTRACT

The architecture, design, and construction of computer software is a human activity. It is intensive in conception, imagination, description, and communication. As such, it is probably the most psychologically oriented of the engineering disciplines. This paper is devoted first to illuminating the salient features of this human activity from the point of view of Descriptive Psychology, with emphasis on the problems peculiar to software engineering. It is devoted secondly to promoting the use of Descriptive Psychology as a tool within the discipline of software engineering to cope with the formidable descriptive tasks encountered there.

This paper is intended for two audiences. The first audience consists of persons with some familiarity with Descriptive Psychology, who have had some contact with computers, and who have a modicum of curiosity about what goes on behind the closed doors of the shops where

Advances in Descriptive Psychology, Volume 5, pages 105-121.
Editors: Anthony O. Putman and Keith E. Davis.
Copyright © 1990 Descriptive Psychology Press.
All rights of reproduction in any form reserved.
ISBN: 0-9625661-0-1.

computer software is produced. For these readers I hope to show, as in a National Geographic article, that what goes on in that alien culture is not so alien after all, that the practices there are driven primarily by human aspirations and limitations, and only secondarily by the strange properties of the computer itself. The second audience consists of professional software engineers who are interested in anything that will make their engineering efforts go more smoothly. For these readers I hope to show that perspectives from Descriptive Psychology can shed light on certain puzzling aspects of their work, and can point the way toward improved methodologies for software design and construction.

WHY DESCRIPTIVE PSYCHOLOGY?

When a civil engineer designs a bridge, the constraints imposed by his clientele on the performance of the bridge can be expressed in relatively simple physical terms: span, width, load-carrying capacity, and the like. When the bridge is built, the work plan is constrained by similar physical parameters: where abutments can be placed, how big a piece can be lifted into place at once, etc. When a software engineer designs a program, say a word processor, the desires of his clientele are normally that the program support them in some mental task whose characteristics are specified (usually with difficulty) in conceptual or behavioral terms: it must be easy to learn, and it must smoothly support changing your mind about what you wish to write. When the program is built, the work plan is constrained by how much one programmer can accurately visualize, by how reliably members of the work team can communicate complex agreements about who does what, etc. In these respects software engineering is the most psychologically oriented of the engineering disciplines.

Moreover, as we shall show in greater detail below, all the mainstream tasks of the software engineer are descriptive tasks. Formally speaking, anything that can be represented in a calculational system (Ossorio, 1971/1978) can be programmed. From a more practical point of view, the main challenge in getting a program right is to get its desired behavior represented in some formalism: *any* formalism. For this purpose, the particular formalisms of Descriptive Psychology (Ossorio, 1971/1978, 1969/1981, 1970/1981, 1979/1981) are very well-suited. For example, in the word processor example above, an excellent starting point for the design would be a paradigm case formulation (including lots of detail) of the process of an author writing an article; a useful model for the programmers to use when communicating with one another would be an object or configuration (Ossorio, 1971/1978) description of the text being worked on. Descriptive Psychology is especially valuable in such applications because it is not tied to any particular programming language. Software engineers tend to suffer from their own special case of the Whorfian Hypothesis: One who programs in Fortran thinks in Fortran, and thereby overlooks important features of the task at hand that Fortran is poor at representing. Anything we can do to break this kind of set is to the good.

THE CONTEXT FOR PRODUCT DEVELOPMENT

This section is devoted to an external view of the creation of products. It represents the external world of the creators of software, and thereby the principal constraints on what will work among their possible actions. The principal constituents of this world are elaborated below.

The Target Community

For every product we build, there is a community that it is going out into. This community is characterized by its members, statuses, concepts, locutions, practices, and world (Putman, 1981).

Among the statuses are those involving actual use of existing and proposed software products (data entry clerk), and those for which the competence required for the status will be to some degree embodied in a new product (accountant). Sometimes these come together in the same person, as when a graphic designer uses a desk top publishing system. The members of the target community expect certain statuses (e.g., Receivables Clerk) to be filled by humans and others (e.g., corporate data utility) to be filled by machines. Some products, given the current popularity of Artificial Intelligence, will fall on the borderline of this distinction, sometimes feeling like persons, sometimes like machines.

Among the concepts of the target community are many that the software itself will have to understand to some degree (font and typeface, for a desktop publisher), and more that the builders of the software will have to understand (graphic design department, paste-up). If the software is to be at all conversational in its interactions, it will have to have built-in many of the locutions of the target community. The practices of the target community work one way before the introduction of the software, and a different way after. For example, tasks formerly done by a draftsman may be absorbed by the graphic designer when a desktop publishing system is introduced. New statuses, for both persons and machines, may have been introduced, together with new or modified practices, as when a business moves from paper to computer-based accounting. The software builders have to appreciate both sets of practices (before and after), and the software itself has to embody those practices in which it participates directly. For example, a computer-based accounting system will prepare the monthly financial statements; it thereby embodies (an alternative version of) the practice engaged in by persons under the paper accounting system. Anyone who installs an accounting system without a clear understanding of both before and after versions of the practice, in the firm under consideration, is asking for trouble.

The Creators of Software

Into the above community come the vendors of software. They are outsiders to the target community, and are viewed as vendors of products, services, or solutions. None of these views, by itself, fits the case. If software is a product, it's a product with a big service component; if a service, it carries with it many objects to be left behind; and in any case it had better provide solutions to what the client sees as problems. The attitude of the target community toward the vendor's software is usually driven by cost/benefit considerations: up-front cost plus cost of maintenance and enhancements plus personnel training cost plus cost of time of experts and consultants, versus faster delivery of needed knowledge, smoother organizational functioning, fewer mistakes, and greater productivity. Usually these benefits are intangible and difficult to estimate, while at least some of the costs are concrete. Any techniques that support more detailed organizational analysis (task analysis, means-end descriptions; see Ossorio, 1971/1978) are called for in examining these cost/benefit issues.

The vendor needs to address the cost/benefit issues, make the benefits more concrete and estimatable, and above all generate accurate images of what the system can and cannot do. In this he is in the role of an actor's impresario, trying to make clear the potential contributions of an absent party. At the same time he has to weigh his promises against the cost of delivering on them—costs that can vary wildly as a result of apparently small changes in system capabilities.

NEEDED ACCOMPLISHMENTS

Somebody on the vendor's work team needs to accomplish the following tasks:

1. Locate High Opportunity Target Communities. Over the last decade, opportunities for successful software (and hardware) introduction into communities has been generally overestimated. (The so-called home computer market is the classic example.) Sometimes this has been due to naive optimism on the part of technologists looking for a place to hawk their wares. More often it has been due to an unwillingness or

inability to grasp the real practices and values of the target community, and to do an adequate cost/benefit analysis from the perspective of that community.

2. Conceive Products Appropriate to Them. Similarly, we have had embarrassingly many examples of both hardware and software that were inappropriate to their target communities, for example, the original Macintosh in the business community (not powerful enough), the Amiga in the current home market (too expensive), all kinds of self-help software (benefits too vague). Disaster after disaster has been obvious in hindsight, yet none were foreseen, at least by the product builders. The effect of all this harsh experience upon the practices of the creators of software has been that now only the firms with very deep pockets will build a product "on spec". Smaller companies stick to building products so desperately needed that the client is willing to put money up front. This practice has at least the salutary effect of providing the opportunity for checking the appropriateness of a software product continuously throughout its development. There will be more below on how this can be done and on the possible role of Descriptive Psychology in the process.

3. Create Business Relationships with Them. At present it seems inconceivable that a successful software product could be developed without the close involvement of members of the target community throughout the entire development process. For the reasons why, see the next section.

4. Capture the Needed Knowledge and Competence. Persons acquire knowledge by observation and contemplation, and competence by practice and experience (Ossorio, 1969/1981, p. 32). Software acquires both, as of this writing, by its builder's building them in. Some of the researchers in Artificial Intelligence are trying to change this, but the fruits of these labors seem unlikely to hit the commercial market for several years. There is, however, an important element common to the acquisition of competence by both persons and software: *Actual participation (perhaps with restricted status) in the practices of the target community is necessary in order to generate the feedback needed to get the behavior right.* For persons, this is done by apprenticeship, with the apprentice learning on his own as he goes. For software, it is done by testing mockups and prototypes in the field environment, with the "learning" accomplished by the programmers who build the discovered corrections into the behavior of the software. It makes sense to say that by these means the software acquires the knowledge and competence needed to do its job right.

But it takes a major acquisition of knowledge and competence just to get a product to the point where it is eligible for even a limited field trial. The key ingredient in capturing this knowledge and competence from the target community is again participation, but this time it is participation of the builders and the clients in each other's communities. For example, the software vendor needs to create activities in which both system builders and members of the target community participate. These can include observation of the target community, interviews, and walk-throughs of practices; shows about proposed system behavior, walk-throughs of designs, and critiquing of mockups, diagrams, specifications, and prototypes. Often, it will (and should) feel to the target community like bringing a new person on board.

5. Embody Them in Working Products. The goal is to come up with a working product that meets the needs of the target community, is appreciated by that community, and is reliable and maintainable; and do so within the limitations of allocated resources. In addition, software, like a human worker, has to grow and change with the job, and it should be possible for this to happen at reasonable incremental costs. Experience has shown all this to constitute a very challenging undertaking; many products fall by the wayside. (Indeed, my preference in software for my personal use is for programs that have survived for at least a year in a community of at least 10,000 active users.)

To build even a minimally satisfactory product calls for the accurate communication of large volumes of detail among all team members. Experience has shown that it is essential that this detail be written down; the opportunities for misunderstanding in oral communication are just too great. English (or any other natural language) often fails for this communication due to too much ambiguity or too little expressive power. For this reason one often finds a host of special formalisms in use for internal communication of design choices: data flow diagrams (Yourdan & Constantine, 1979), decision tables, HIPO Charts; social practice descriptions (Putman & Jeffrey, 1985); various kinds of tables, structured English, etc. Each of these formalisms can be viewed as a special case of one of the descriptive formats proposed by Ossorio (Ossorio, 1971/1978), so that Descriptive Psychology provides a kind of metalanguage into which we can fit each of these special communication tools. Doing this placement gives us a valuable perspective from which to create new communicational tools or enhance the old ones.

Ultimately, the entire behavior of the system must be specified in some machine-readable form. This specification may use any of the above-mentioned languages, actual compilable code, sets of rules for an expert system shell, private little languages, or what have you. But, as of this writing, whatever is going to be in the behavioral repertoire of the software, we have to put there explicitly by means of these descriptions. And, given the difficulty of getting these descriptions right, they must all be subjected to several levels of checking, usually by some combination of independent review of the descriptions themselves with testing of the resulting software in action. The sheer mass of this descriptive task constitutes the most serious obstacle to successful completion of most software projects.

6. Introduce the Products into the Target Community. Some years ago an advertisement appeared in one of the computer trade journals showing several crates sitting forlornly on an otherwise empty loading dock. The caption read: "Some computer systems aren't delivered, they're abandoned". Today, vendors of both hardware and software are sensitive to the problems of getting a new system into effective use, but these problems remain thorny. To stretch an analogy used above, it is like bringing a new person on board, but the new person is infinitely more helpless that any human being to push for getting himself wisely used. Further elaboration of this challenging task will have to wait until I garner more experience with it myself.

ECONOMIC CONSTRAINTS

All six of the above accomplishments cost money to do. The last three of them (capture needed knowledge and competence, embody them in working products, and introduce the products into the target community) are legitimately chargeable to software development contracts. The first three (locate high opportunity target communities, conceive products appropriate to these communities, and create business relationships with them) must come out of the software vendor's profits. The chargeable accomplishments are usually done against a budget, either an internal one or an external fixed-price contract. Thus it is crucial to be able accurately to estimate their cost up front. It is more important to have a reliable upper bound on the project cost than to be able to predict the actual cost. This is because there are so many things that software might do, which look attractive on the surface, and which should not be done because the development costs outweigh the benefits. A software vendor unable to filter these situations out is doomed immediately. (In situations involving advanced development, estimation of costs within a 50% tolerance is frequently impossible. Clients and contractors have come up with many imaginative project-staging and risk-sharing arrangements for dealing with such situations.)

Capturing the needed knowledge and competence presents at least two economic problems: On the client side it is intangible (how much does it cost to have this key person pestered by knowledge engineers?), and on the vendor side it is expensive. It is expensive because system implementation eventually requires machine-readable descriptions of system behavior, and somebody has to span the cultural gap between the target community and some community in which machine-readable descriptions are a core element. It doesn't matter too much whether we call this somebody a systems analyst, a knowledge engineer, or a programmer; and it doesn't matter too much what language the machine-readable descriptions are in (any of the ones mentioned so far are among the possibilities). The crucial parameter is the width of the cultural gap. (Different communities implies different members, statuses, concepts, practices, locutions, or world; see Putman, 1981). The main things that need to be carried between communities for system description are concepts, practices, and locutions. We understand fairly well how to train people to translate practices and locutions (although this is far from a trivial skill to acquire); translating concepts is far less well-understood and therefore rarer. In most situations today, the width of the cultural gap is large enough that those who can successfully work across it are rare and expensive. Furthermore the task itself is very exacting, so that the systems analyst, knowledge engineer, or programmer not only has to be competent at spanning the large cultural gap, but also at managing large masses of detail, constantly checking correctness, and frequently correcting large, messy descriptions without wrecking them—not to mention dealing patiently and creatively with all kinds of feedback from both communities: The clients hate this feature; the machine won't execute that one.

The problem of spanning this cultural gap provides the field for a host of currently attempted technological advances:

Formal Design Languages

Formal design languages constitute a compromise between English and machine-readable code (see Yourdon and Constantine, 1979, for several of the examples mentioned above). The creators of these languages attempt to remove the ambiguity of English by restricting the vocabulary and by providing formal procedures for defining new terms. They attempt to enhance the expressive power by adding new syntactic constructions, graphical or textual, to facilitate the construction of large, articulated, descriptions. Thus they allow us to bridge part way towards the terrible discipline of the fully unambiguous, implementable, description without taking the whole jump at once.

Rapid Prototyping Tools

Rapid prototyping tools are software construction aids that assist us in building preliminary systems having limited functionality, fast. They often force us to bridge further toward the machine than does a formal design language, but pay off with something that actually runs, albeit below the standards of the envisaged product. Such a prototype is often a necessary intermediate step, not merely for technical reasons, but in order to provide clients with something they can see, feel, and most importantly compare with their visions of how the product will behave. Just as a musician may be able to look at a score and hear the music it represents, an analyst may be able to look at a design language description and feel the behavior of the product described. Few clients are blessed with this gift; therefore the prototype is an important early step in generating design feedback.

Expert System Shells

Expert system shells offer quick construction of systems for which the principal concepts needed by the system itself are if-then-else rules. They promise us finished products after spanning a cultural gap that is an order of magnitude smaller than we are used to, if only the desired application is amenable to description in the rule-oriented language provided. As with other descriptive languages, there is a trade-off between breadth of application and ease of use: narrow-scope tools like M-1 and ExpertEase are easy to learn and use; broad-scope tools like S-1 and Kee call for specialized competence comparable to that of a programmer.

For the professional software builder, it is easy to become jaded about the continual procession of new software tools, each promising to make software productivity equal at last to the current challenges. There is an economic equilibrium: Software productivity has improved a lot in the last decade, and every advance is immediately absorbed by the new class of applications that has just become feasible using the new tools. The advance is never as great as its inventors hoped; still, there is always a new class of applications just around the corner that would become feasible if only we could make some part of the system building task (usually the knowledge capture) another order of magnitude cheaper.

Embodying knowledge and competence in working products also presents economic problems. Since complete, consistent, detailed descriptions are so expensive, the key challenge here is to throw away as little as possible along the way. That is, build technological capital in the form of re-usable descriptions that allow us easily to carry the bulk of past problem solutions into the future without having to re-solve those problems. (There will always be plenty of new ones!) The main tool here (in addition to all those already mentioned) is a library of descriptions, preferably machine-readable, some purchased, some locally built, that constitute the technological capital of the firm.

HUMAN SYSTEMS CONSTRAINTS

When there are tasks that are done expensively and poorly, it is usually because the work is on the ragged edge of the abilities of persons to do it. Several of these situations in the creation of software have been suggested by the economic issues mentioned above. For example, bridging disparate cultures is difficult for many; so is managing large amounts of detail in an environment in which small slips make big messes. It is useful to enumerate the powers, and dispositions (Ossorio 1970/1981) needed for various system building tasks, along with significant behavioral limitations that will prevent persons from accomplishing these tasks. For example, we have so far mentioned the following abilities as needed at some point in the task: to capture the concepts, practices, and locutions of an alien community, to communicate them effectively to other team members, to effectively manage large masses of detail, and to cope with computing machinery. To complete this enumeration is beyond the scope of this article, but should be done as preparation for an analysis of how the entire task of software construction can be better partitioned among the various players. At present the jobs tend to fall into two classes: jobs calling for a combination of abilities that is absurdly rare (systems analyst, knowledge engineer), and jobs described by Fred Brooks's (1975) dictum: "Anyone smart enough to do . . . right is too smart to do it for long" (project librarian, software test specialist). We need to critique our classification of jobs and responsibilities with an eye toward getting the required competences lined up with what we can reasonably expect to find in one person.

There is a related class of relevant human systems constraints having to do with communication. Persons whose attention is focussed on creating or maintaining large, complex descriptions tend to forget to communicate to others those actions that impact the others' use of those large complex descriptions. As such a person, I can testify that it is tempting to dive into such a description and regard it as my whole world. Coming up for air and emerging into the broader reality is an uncomfortable task often put off as long as possible.

METHODOLOGY

Overview

When a product has been completed, it includes many descriptions of the behavior of the product. Some are brief and glossy, for marketing use, as you might receive unsolicited in the mail. Others are more detailed, for the persons who are going to use the product, like the user's manuals for WordStar or PCDOS. Still others go into the inner workings of the product, for those who will maintain it, for a typical microcomputer product, these are from 3 to 10 times the volume of the user's manual. Finally, there are the descriptions that are machine readable, for compilation or interpretation by the computer as it runs the product. For a typical microcomputer word processor or spread sheet, these descriptions total from 10,000 to 100,000 lines of code in some programming language. To gain enough familiarity to modify such a product might take an experienced programmer several weeks of study.

Note that the sum total of all these descriptions *is* the product: There is nothing else to add. They should all be consistent with each other. Each should be complete, from the perspective of the community it is for. And, of course, the behavior described by each should be appropriate to the needs of the target community. Each represents the same product, redescribed from the point of view of a different community or status within a community. The language of each differs, subtly or spectacularly, from that of the others.

Each of the above descriptions has to be created by members of the software vendor's staff, perhaps with machine aid. In addition these staff members will typically produce many more descriptions, partial (including demos), incorrect, supporting (scaffolding), even subsidiary products for internal use. Some of these descriptions may be, like those mentioned above, consistent and complete descriptions of the whole product from the perspective of some special community or status within the vendor's development shop. Work planning for product development consists of deciding which descriptions are to be produced when, by whom, and how they will depend on descriptions produced earlier. A key planning question is how we apportion the work to take advantage of the different strengths and weaknesses of the team members—especially the most oddball team member, the computer.

Most of the descriptions are notoriously error-prone. Much of the development time is devoted to checking their correctness by whatever means work. More key planning questions relate to this testing: How is each description to be tested? Which can be used to test each other?

The rest of this paper is prescriptive. It consists of suggestions and rules of thumb gleaned from several years of experience. Although they are all part of the folk wisdom of software engineering, the perspective of Descriptive Psychology has thrown a fresh light on each of them.

1. About the Descriptions Generated. Every description has to be readable to somebody; the fewer that are only readable to arcane specialists, the better.

Compiling or interpreting descriptions is desirable. For example, programs that have to format text on a page often interpret a little language for describing the formatting: Center this line, change the right margin, etc. A program that needs to know the address of the Secretary of State of each of the 50 states would normally keep this information in a table. In each case we have a small descriptive language with narrowly prescribed formats that is read and written by both persons and programs.

Special viewing tools short of full compilation or interpretation may be needed. The best examples of these are for dealing with the complex descriptions we call computer program source listings: cross reference generators and formatters that capitalize and indent automatically according to the structure of the text.

All of these descriptions may be considered cases of some Descriptive Psychology format: The page formatting language turns the text to be printed into a kind of process description. The table is a state of affairs description consisting of a number of element-individual pairs. The programming language source listing is process description with object, process, and state of affairs constituents. It is useful to take this perspective when confronted with a particularly opaque description; sometimes it can be thereby be reorganized into something better.

For descriptions that are to be written in an actual programming language, use the highest level programming language for which you have an adequate implementation. When applied to programming languages, "higher level" means "permitting a narrower cultural gap between the program text and a description in the language of the target community". (Another way of saying this is that the basic concepts of the programming language are closer to the relevant concepts of the target community.) Consequently, the higher the level, the greater the economies across the board in construction, checking, and maintenance.

2. Make up Languages Appropriate to the Task. Descriptive Psychology provides a rich stock of descriptive formats (units for objects, processes, events, and states of affairs; task and means-end descriptions, etc.). There is an infinite range of possibilities for how these formats, or variations on them, might be used in a given practical descriptive task. The job calls for experience, imagination, and wisdom. Descriptive Psychology is a more a metalanguage than a language; it provides boundary conditions on what forms of expression make sense, and some hints as to what might work well in certain situations, but the detailed representation of each real world situation (including the form of the representation) is up to the person who needs to describe that situation for computer use. This is most obvious (and most difficult) at the point

where the bottom levels of human-oriented description meet the top levels of machine-oriented description.

3. Anchor at the Top. Descriptions of large, complicated systems are made comprehensible (sometimes) by the copious use of part-whole descriptions. Unless some very strong reason calls for an exception, always *describe the whole before its parts.* Such a description provides the context for each part described, enhances readability, and reduces the likelihood of inconsistency. Note that each part described may be an object, process, event or state of affairs (not to mention the derivative computer science concepts: procedure, data structure, message, agent, module, etc.).

Once you have become familiar with the purpose of a computer system that is under design, it is very easy to neglect to write down the topmost levels of its description. This tendency must be avoided, both for the sake of future maintainers of the system and for the sake of your own future elaboration of the details: The top levels of the design contain boundary conditions on what will work at the lower levels that are amazingly easy to forget.

The top levels are best written in a language very different from most programming languages. A good starting point is a community description: What members (software-implemented agents) will there be? what statuses? what practices? what concepts are needed to carry out these practices? what messages sent and received? what is the logical form of these messages? A good exercise for systems analysts is to critique data flow diagrams (Yourdon & Constantine, 1979) from this point of view. What do they cover? What do they omit?

4. Keep Everything Visible (to anyone who cares). The product is composed entirely of descriptions. Each description is written in the language of some community. Some of these communities are closely tied to the computer and some are not. Complexity is everywhere. The most important guideline I can think of is this: *describe each complexity in the language of the community it belongs to.* Lawyers have honed their language to deal with legal complexities, managers with business complexities, and accountants with financial complexities. These complexities are difficult enough to write down in a language that was designed for them. Writing them down in computerist's language is certain disaster: then for anyone to critique or maintain them, he has to be fluent in the concepts of both communities. The guideline above maximizes the visibility of the product's design; everyone who has an interest in a certain aspect of the system has at least a chance of reading the descriptions that pertain to that aspect of the system.

Of course, we must eventually have machine-readable descriptions, so if the descriptions discussed above need to be translated by hand into machine-readable form, we have accomplished little. The answer is to machine-translate these descriptions into something that is more harmonious with the computer. Put another way, we make these descriptions readable not only to some target community of persons, but to one or more computer programs designed specifically to handle this class of descriptions.

5. Minimize Redundancy. Redundancy in descriptions of a software product is a major liability. It bloats the descriptions, inhibits their readability (and implementability) and, worst of all, opens the door for inconsistencies between the various redundant parts. On the other band, the removal of redundancy is absolutely secondary to the objectives of the preceding paragraph; readability to the concerned communities comes first. This means that whenever two descriptions of the "same thing" are needed for two different communities, we need to either pick one and derive the other from it (preferably by machine), or create a third and derive both of them from it. The simplest example is a table of numbers: It needs to be in text form to be created, critiqued, and revised by persons, yet eventually gets represented using tbe internal representation of numbers in the computer. Often we find ourselves building separate programs to do this translation; tbey are compilers for data.

6. Build a Dictionary of Concepts. The total of all the concepts used in all the descriptions that constitute a product make up a language and world private to the product (and the development team). For example, the documentation for a word processor often contains specialized terms like "cursor", "text buffer", "insert mode", and "edit session". If we continued the search for specialized terms throughout all the descriptions making up the word processor right down to code, we would typically come up witb 500 to 1500 words and phrases. As obnoxious as such a jargon is, it is inevitable, and the best we can do is to compile a dictionary for it. That way we at least reduce the chances that two team members (or one at different times) are using slightly (or wildly) different versions of wbat they thought was the same concept. In the jargon of the systems analyst, tbe "data dictionary" is an approximation to this dictionary. To design effective formats for such a dictionary that allow it to be more comprehensive than current data dictionaries is a major potential application of Descriptive Psychology to software engineering.

7. Isolate the Tough Descriptive Nuts; Start on Them Early. Early in the descriptive task for a product, there often surface situations tbat cause butterflies in the stomach of the experienced analyst. Some description may appear to require a billion cbaracters of text to write down, or some object may require conceptual elaboration in two entirely

different directions for two of the communities involved, or some needed part of a description may appear to be subject to combinatorial explosion. These tough descriptive nuts may constitute real show stoppers, or merely points where unusual skill will be needed to come up with the right kind of description. In either case, lavish attention early from the most experienced brains on the project is called for. The worst thing that can happen is to pick an inadequate descriptive methodology for one of the nuts and try to bull your way through, only to have the project bog down because of it 10,000 lines of code later.

8. Test Concurrently with Every Step of Development. As more and more descriptions are constructed and hooked together, more and more of the behavior of the desired product becomes manifest. For every deviation of this behavior from that desired, we must find the bug and repair it. To do so appears to take an amount of effort that increases faster than linearly in the size of the portion of the product so far constructed. To avoid major wastage of time (testing typically consumes at least as much time as building), we must organize incremental testing along with incremental building so that the troubleshooting time does not explode as the product nears completion. I find this a most challenging facet of the art of software engineering.

The preceding paragraph was concerned with the effort necessary to find bugs. There is an analogous situation with respect to the effort necessary to fix them. Each bug fix can be visualized as backtracking in the construction process to the point where an error was made, then rebuilding forward without making the error. The
cost increases with the distance you have to backtrack: Early-stage design errors that surface late in the construction process are particularly costly. With long experience, project managers develop an instinct for how to organize the work to hold down the length of the likely backtracks. It would be nice to have some kind of theory of this phenomenon.

9. Use Persons Far From Your Own Community as Testers. A good aeronautical engineer can almost feel the stresses in a plane in which he or she is flying. The same is true for software engineers, and they seem to have an instinctive disinclination to break that which they have built. Therefore we need persons with a different mental set to serve as testers.

10. Make a Visual Mockup First. Each software product has its own distinct feel. Some feel like flexible and powerful machine tools in the hands of an expert, like a radial arm saw used by a cabinet maker (the Unix shell is a possible example). Others have the comfortable feel of a familiar household appliance, always responding appropriately to a few simple commands (the PFS series of products was designed to be this

way). Still others give the illusion of a rather limited person who is holding a conversation with you (e.g., the infamous Eliza, or the dialog boxes used by many Macintosh programs). It is essential that this feel be harmonious with the desired role of the product in the target community, and this must be assured up front, for the feel influences everything about the internal design. It is becoming fashionable among software developers to check this out via one or more mockups that simulate the product's behavior in a very restricted range of scenarios (Dan Bricklin's Demo Program is often used for this).

11. Implement the More Visible Parts Before the Less Visible. Although description proceeds most naturally top down, from the whole to the parts, the parts can be implemented in many possible orders. The order of implementation should at least do justice to the extreme error-proneness of the descriptive task: Of all the parts you might implement at a given point in time, implement first the one that is the most visible; that way you get the most opportunity for testing and inevitable revision. The preceding paragraph is an example: The product's feel is its most visible part. If two parts are equally visible, implement first the one that is more error-prone. If, on the other hand, you implement something invisible, is faults will remain hidden, lulling you into a false sense of how much has been finished. This advice dovetails with the discussion above about testing concurrently with every step of development.

12. Don't be Afraid to Use a Rich Array of Descriptive Methods and a Correspondingly Rich Array of Software Tools for Dealing with Them. Fifteen years ago, almost all the descriptions making up a business-oriented software product were of two forms: tables and Cobol code. Today it is more common to have many forms: tables, knowledge bases, interpreted descriptions of processes, relational models, social practice descriptions, and even different programming languages for different parts of the product. Each of these descriptive forms might have associated with it a compiler, an interpreter, an editor, a critic, or a formatter. We must be careful not to be overwhelmed by the task of building these tools, yet it is often economical to build them because they can be used across a range of similar products.

13. Cast the Most Volatile Parts in the Most Pliable Medium. Tables are "soft" (easily changed); code is "hard"; the other descriptive forms fall somewhere between. Some descriptions get revised with every bug fix or product enhancement while others remain stable for years. Obviously, we want to make the medium fit the function: soft media for volatile functions; hard media for nonvolatile functions. This apparently simple objective is astoundingly hard to achieve in practice.

EPILOGUE

I have tried here to follow my own advice and give an exposition anchored at the top and elaborated down to a useful level of detail. I wish that it contained more concrete, useful, rules. I am happy, though, with the number of guidelines and rules of thumb included, and particularly pleased with the perspectives on software engineering that I have demonstrated by using them. I hope that some of the questions raised here may provide others with fruitful research topics.

ACKNOWLEDGMENTS

I wish to thank those who have participated with me in discussions, sometimes heated, about the subjects in this paper, especially P. G. Ossorio and Lowell Schneider. I also thank those who have helped to generate the concrete experience, sometimes harsh, that motivated those discussions, especially Terry Schmid, Cecelia Jacobs, and Stephanie Allen. And I especially thank those who have participated in both the above statuses: Joe Jeffrey and Anthony O. Putman.

Author's address: 710 Hawthorn Ave., Boulder, CO, 80304

REFERENCES

Brooks, F. B., Jr. (1975). *The mythical man-month*. Reading, MA: Addison-Wesley.

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

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

Ossorio, P. G. (1970/1981). Outline of Descriptive Psychology for personality theory and clinical applications. In K. E. Davis (ed.), *Advances in Descriptive Psychology* (Vol. 1, pp. 57-81). Greenwich, CT: JAI Press, 1981. (Originally published in 1970 as LRI Report No. 4d. Whittier, CA and Boulder: Linguistic Research Institute.)

Ossorio, P. G. (1979/1981). Conceptual-notational devices. In K. W. Davis (Ed.), *Advances in Descriptive Psychology* (Vol. 1, pp. 81-104). Greenwich, CT: JAI Press, 1981. (Originally published in 1979 as LRI Report No. 22. Boulder: Linguistic Research Institute.)

Putman, A. O. (1981). Communities. In K. E. Davis (Ed.), *Advances in Descriptive Psychology* (Vol. 1, pp. 195-209). Greenwich, CT: JAI Press.

Putman, A. O., & Jeffrey, H. J. (1975). A new paradigm for software and its development. In K. E. Davis & T. O. Mitchell (Eds.), *Advances in Descriptive Psychology* (Vol. 4 pp. 119-138). Greenwich, CT: JAI Press.

Yourdan, E. & Constantine, L. L. (1979). *Structured design*. Englewood Cliffs, NJ: Prentice-Hall.