SOFTWARE PROCESSES ARE SOFTWARE TOO

Leon Osterweil

University of Colorado Boulder, Colorado USA

1. The Nature of Process.

The major theme of this meeting is the exploration of the importance of .ul process as a vehicle for improving both the quality of software products and the the way in which we develop and evolve them. In beginning this exploration it seems important to spend at least a short time examining the nature of process and convincing ourselves that this is indeed a promising vehicle.

We shall take as our elementary notion of a process that it is a systematic approach to the creation of a product or the accomplishment of some task. We observe that this characterization describes the notion of process commonly used in operating systems-- namely that a process is a computational task executing on a single computing device. Our characterization is much broader, however, describing any mechanism used to carry out work or achieve a goal in an orderly way. Our processes need not even be executable on a computer.

It is important for us to recognize that the notion of process is a pervasive one in the realm of human activities and that humans seem particularly adept at creating and carrying out processes. Knuth [Knuth 69] has observed that following recipes for food preparation is an example of carrying out what we now characterize as a process. Similarly it is not difficult to see that following assembly instructions in building toys or modular furniture is carrying out a process. Following office procedures or pursuing the steps of a manufacturing activity are more widely understood to be the pursuit of orderly process.

The latter examples serve to illustrate an important point--namely that there is a key difference between a process and a process description. While a process is a vehicle for doing a job, a process description is a specification of how the job is to be done. Thus cookbook recipes are process descriptions while the carrying out of the recipes are processes. Office procedure manuals are process descriptions, while getting a specific office task done is a process. Similarly instructions for how to drive from one location to another are process descriptions, while doing the actual navigation and piloting is a process. From the point of view of a computer scientist the difference can be seen to be the difference between a type or class and an instance of that type or class. The process description defines a class or set of objects related to each other by virtue of the fact that they are all activities which follow the dictated behavior. We shall have reason to return to this point later in this presentation.

For now we should return to our consideration of the intuitive notion of process and study the important ramifications of the observations that 1) this notion is widespread and 2) exploitation of it is done very effectively by humans. Processes are used to effect generalized, indirect problem solving. The essence of the process exploitation paradigm seems to be that humans solve problems by creating process descriptions and then instantiating processes to solve individual problems. Rather than repetitively and directly solving individual instances of problems, humans prefer to create generalized solution specifications and make them available for instantiation (often by others) to solve individual problems directly.

One significant danger in this approach is that the process itself is a dynamic entity and the process description is a static entity. Further, the static process description is often constructed so as to specify a very wide and diverse collection of dynamic processes. This leaves open the distinct possibility that the process description may allow for process instances which do not perform "correctly." Dijkstra makes this observation in his famous letter on the GOTO statement, [Dijkstra 69] observing that computer programs are static entities and are thus easier for human minds to comprehend, while program executions are dynamic and far harder to comprehend and reason about effectively. Dijkstra's point was important then and no less significant now. Processes are hard to comprehend and reason about, while process descriptions, as static objects, are far easier to comprehend. Finally it is important to also endorse Dijkstra's conclusion that our reasoning about process descriptions is increasingly useful in understanding processes as the descriptions are increasingly transparent descriptions of all processes which might be instantiated.

In view of all of these dangers and difficulties it is surprising that humans embark upon the indirect process description/instantiation/execution approach to problem solving so frequently. It is even more startling to observe that this approach is successful and effective so often. This suggests that humans have an innate facility for indirect problem solving through process specification. It is precisely this innate ability which should be able to propel us to become far more systematic and effective in the development and evolution of computer software. What currently stands most directly in our way is our failure--to date--to understand our most central and difficult problems in terms of the process description/instantiation/execution paradigm.

2. Computer Software Processes.

It has become increasingly popular to characterize software development as a manufacturing activity in which the final delivered software should be considered to be a manufactured product. Efforts to consider software development analogous to other manufacturing activities such as auto assembly or parts fabrication have considerable intuitive appeal, although they have not been entirely satisfactory. From our perspective it seems clear that the set of activities carried out in order to effect the development or evolution of a software product should be considered to be a process. There seems to be a clear analogy between this sort of process and the process by which airplane wings are built, gourmet meals are prepared and autos are assembled.

The analogy seems to be relatively weaker in that 1) the product produced by a software process is intangible and invisible and 2) there seems to be no tangible process description from which our software processes are instantiated.

The former difficulty has been observed and discussed before. It is certainly true that we are hindered in our work by the fact that we cannot see our product and by the fact that we are neither guided nor constrained by the laws of physics, biology or chemistry in creating it and reasoning about it. Our product is a pure information product, being a structure of information and relations upon that information. Curiously enough, people in other disciplines emulate our situation, fashioning abstract models of their more concrete problems and systems in order to render them more analyzable and manipulable. The difference here is that their information structures are created to represent key aspects of their more concrete realities, while our information structures are our final products and our realities. They can use the five human senses to grasp and reason about their product--we cannot use these same senses to help us with our product. Accordingly we have difficulty reasoning about software products, and difficulty in thinking of them as real and tangible.

The latter difficulty is the main subject of this presentation--namely the paradoxical lack of process descriptions which are appropriate for the specification of the processes which we use in developing and evolving software. The paradox here is that elaborate and careful descriptions of our processes would seem to be most appropriate in view of the fact that our products are so difficult to grasp and so large and complex. Contemporary software products would have to be considered very large and complex by any objective measure. Our large systems may contain millions of lines of source code, but also contain volumes of documentation, enormous quantities of design information, specifications, testcases and testing results, as well as user manuals, maintenance manuals, and so forth. Further each of these software objects must have been integrated and correlated within themselves and with each other in surprisingly intricate ways. Our early attempts to define and characterize even relatively simple software products have lead to the conclusion that, while we were expecting these products to be large and complex, they are far larger and more complex than we had expected.

In view of the fact that our products are so large and complex, and in view of the compounding effect of the intangibility and invisibility of these products it is all the more surprising that we generally go about the job of developing and evolving them by employing processes which are rarely if ever guided by appropriately sharp, detailed and understood process descriptions. It is even more astounding that we are generally rather successful with these processes. This suggests that our innate process description/instantiation/execution capabilities are powerful indeed. They seem to be sufficiently powerful that we are able to improvise effective process descriptions on the fly, maintain them in our heads, modify them as necessary and guide others in their effective execution. Just think how much we might be able to achieve were we able to externalize these process descriptions, express them rigorously, promulgate them publicly, store them archivally, and exploit computing power to help us in guiding and measuring their execution.

This talk suggests a specific approach to doing this. We suggest that it is important to create software process descriptions to guide our key software processes, that these descriptions be made as rigorous as possible and that the processes then become guides for effective application of computing power in support of the execution of processes instantiated from these descriptions.

Our specific approach is to suggest that contemporary "programming" techniques and formalisms be used to express software process descriptions.

3. Software Process Programming.

By now many will doubtlessly be observing that the idea of describing software processes is not new and has been done before. Certainly software "program plans," "development plans," even "configuration control plans," are examples of externalized software process descriptions. Further, industrial processes are routinely expressed by means of such formalisms as Pert charts and flow diagrams. Office procedures are often described in written procedures manuals. In view of this it is reasonable to question what new is being proposed here and to ask why we believe that we should expect dramatic improvement in the way in which we develop and evolve software in view of the fact that we are emulating techniques which have not been completely successful in other disciplines.

Our optimism springs from the fact that we have evolved powerful and disciplined approaches to handling intangibles as products, where other disciplines have not. We have dealt with the fact that our products are procedures for creating and managing invisible information structures. While our solutions are far from perfect, they are, at least, direct attempts to manage the invisible and intangible. By contrast the efforts of other disciplines seem less direct, less powerful and less effective. As computer scientists we have long ago embraced the use of rigorously defined languages as vehicles for describing complex and intricate information creation and management processes. We have evolved systems and procedures for studying these process descriptions, for analyzing them, and for applying them to the solution of specific problems. This whole realm of activities is what we have come to loosely describe as "programming." As computer scien-

tists we have developed programming languages, have had some success in formally describing them and their effects, have encoded enormously complex descriptions of information manipulation processes, and have had success in instantiating such processes to carry out such impressive jobs as guiding the landing of people on the moon, switching myriads of telephone calls in a second and navigating airplanes and rockets at high speeds. In doing all this, we as a community have developed considerable instincts, talents and tools for treating processes as tangibles. It seems most natural and most promising to harness these instincts, talents and tools to the job of describing software development and evolution processes just as we approach classical "programming."

Our suggestion is that we describe software processes by "programming" them much as we "program" computer applications. We refer to the activity of expressing software process descriptions with the aid of programming techniques as *process programming*, and suggest that this activity ought to be at the center of what software engineering is all about. In succeeding remarks we shall attempt to briefly outline why we have this opinion.

4. A Rudimentary, but Instructive, Example.

In this section we will present a very small example of a process program and use it to illustrate some of the benefits of the process programming approach. Before giving the example it seems worthwhile to make some basic observations about the strong analogy between a classical application program and a process program.

Perhaps the most basic observation is that both programs should be viewed as descriptions of processes whose goal is the creation or modification of a complex information aggregate. Each is used by instantiating the description into a specific product template, binding specific data values to it, and then allowing the process to transform those values into software objects and aggregate those objects into a final software product. In the case of an application program we generally think of the size of the input data objects as being relatively small--perhaps single numbers or short character strings--although we are all familiar with application programs in which the input data values are large and complex data aggregates. The inputs to software process programs will generally be of this latter kind, typically being source code statements or procedures, design information, test cases, or blocks of documentation. In both cases, however, the input data is examined, stored, and transformed into output software objects and products.

Further, in both cases the problem at hand, be it the solution of an application data processing problem or the conduct of a software process, is effected by instantiation of the process description, binding of values to the instance and evolution of problem specific solution elements. In the case of the application program, the embodiment of the problem solution description is executable code. In the process programming case, the problem solution is the process program itself. Later we shall discuss the sense in which we believe that process programs should be considered to be executable. For now, however, we can think of them as being expressed in a comfortable high level language.

In the following example, the process programming language is taken to be Pascal-like. The reader should not be misled into thinking that such a language is the preferred medium in which to express process programs. Research into appropriate process programming language paradigms is needed. We are convinced, for example, that rule-based languages will be effective vehicles for expressing certain software process descriptions, while algorithmic languages may be preferable in other areas. Thus, the following example is offered only in the spirit of lending definition and specificity to this discussion.

4.1. The Example.

Here is an example of a process program which describes a very straightforward type of process for testing application software. As such it is offered as an example of how to describe one small but important part of a larger software development process.

```
Function All_Fn_Perf_OK(executable, tests);
  declare executable executable_code,
        tests testset,
        case, numcases integer,
        result derived_result;
--Note that executable_code, testset, and derived_result
--are all types which must be defined, but are not defined
--here.
  All_Fn_Perf_OK := True;
  For case := 1 to numcases
--This is the heart of the testing process, specifying
--the iterative execution of the testcases in a
--testset array and the comparison of the results with
--expected behavior.
     derive (executable,
          tests[case].input_data,
          result)
     if ~resultOK (result,
          testcase[case].req_output)
        then All_Fn_Perf_OK := False;
          exit;
--Note that the process specified here mandates that
--testing is aborted as soon as any test execution does
--not meet expectations. This is an arbitrary choice
--which has been made by the process programmer who
--designed this testing procedure.
     end loop;
  end All_Fn_Perf_OK;
```

While this process program is quite short it exemplifies some important aspects of process programs. First it should be noted that it highlights the key aspects of the testing process, while hiding lower level details. As such it is a reasonable example of the use of modularity, and of its application to the hard and important task of conveying software process information clearly. In this case we see that the testing process is

an iterative loop which ends when a testcase fails. Further we see that the essence of testing is the evolution of test results and the evaluation of them. Details of how the results are evolved and how they are evaluated are left for elaboration in lower level process program procedures. It turns out that these details depend upon the fine-structure of objects of types testset and derived_result. It is important in itself that such entities as testsets, testcases, and derived results are treated as typed objects, requiring rigorous specification, and dealt with as operands to key operators in the process.

We may be interested in evaluating test results either for functional correctness or for adequate performance, or both. We may wish to determine acceptable functionality by comparing two simple numbers, by comparing two information aggregates, or by applying some complex function to the derived result and the expected result. The process program we have just shown is equally descriptive of all of these types of processes. It can be specialized to meet any specific testing need by the proper elaboration of the types and procedures which it uses.

For example, the following procedure specifies that a testing result is to be considered acceptable if and only if 1) the the functional result computed and the functional result needed cause the function "fcnOK" to evaluate to True, and 2) the observed execution time for the testcase is less than or equal to the execution time needed.

In order to understand this procedure and the way in which it elaborates upon All_Fn_Perf_OK, it is important to have the following type descriptions. It should be assumed that these definitions have been made in some outer scope of the process program in such a way that they are visible and accessible to the procedures we are specifying here.

```
declare testset array of
  testcase[1..numcases];
--
declare testcase record of
  (input_data real,
   req_output record of
    (fcn_tol predicate,
     time_tol predicate,
     fcn real,
     time real));
--
declare derived_result record of
  (fcn_output real,
   observ_timing real);
```

We can now specify key lower level process program procedures.

```
Function resultOK (result, needed);
  declare result derived_result,
        needed req_output;
  if fcnOK (result.fcn_output,
         needed);
 --Did the test compute an acceptable functional result
        OR
    (result.observ_timing >
     needed.time)
--Did the test run fast enough
      then resultOK := False;
      else resultOK := True;
  endif;
  end resultOK;
--
--
Function derive (pgm, test, result);
  Declare pgm executable_code,
        test testcase,
        result derived_result;
--start by resetting the testing timer to zero
  reset_clock
--derive the functional output result and set it in the
--right field of the derived_result object
  result.fcn_output := execute (pgm, test.input_data);
--stop the testing timer and store the execution time
--of the testcase in the right field of the
--derived_result object
  result.observ_timing := read_clock;
  end derive;
--
--
Function fcnOK (result, needed);
  Declare result derived_result,
        needed req_output;
--NB  It is assumed here that the process programming
--language allows for the inclusion of a function which
--has been specified as the value of a typed object
--and for the application of that function as part
--of the execution of a process program procedure.
  fcnOK := needed.fcn_tol (needed.fcn, result.fcn_output);
  end fcnOK;
```

In the interest of saving space, this example is left incomplete. Some lower level procedures and some types have not been completely specified. We believe that this lack of completeness should not interfere with a basic understanding of process programming. We hope that the reader sees that these algorithmic expressions can be used to effectively capture and convey details of a software development procedure clearly, but in a phased, gradual way which is familiar to software engineers. It is not important that the particular details of this example either agree or disagree with the reader's experiences or opinions on how testing should be carried out, but rather that the reader understand that the process programming vehicle has been used to unequivocally describe a specific procedure in terms that software practi-

tioners should have little trouble understanding and reasoning about. Similarly it is believed that this mechanism could be used to clearly and unequivocally express whatever other testing procedure a software practitioner might wish to specify. In particular, if a given process program is not deemed suitable, then it becomes an appropriate centerpiece for sharp, focused discussions of how it should be modified to make it more suitable.

Further, this example is intended to suggest that the specified testing process might well be imbedded in a higher level process which describes the broader development contexts within which testing is to be carried out. In general we see no limits on the level at which software processes might be described by process programs. Process programs have been used to express entire software lifecycle models and have been elaborated down to very low level details. When process programs are written at a high level, but are not elaborated to low levels of detail, they become general prescriptions for how software work is to be done. When process programs describe lower levels of detail they become tools for substantive discussions of exact procedure. Each type of activity in its own way seems useful in assuring effective software development.

We believe it is significant that the example is longer and more complex than might be expected. What is described above is a rather straightforward testing loop, yet there are surprisingly many details to be specified. There is an unexpectedly large hierarchy of types needed to clearly and accurately express exactly what this testing process entails. This suggests to us that the processes which we intuitively carry out are more complex than might be expected and explains why it is often so difficult to explain them to others, so easy to overlook unexpected consequences, and so hard to estimate the effort needed to carry them out. These observations lead us to some important conclusions about the potential benefits of process programming.

5. Advantages of Process Programming.

In general we believe that the greatest advantage offered by process programming is that it provides a vehicle for the materialization of the processes by which we develop and evolve software. As noted above, it is startling to realize that we develop and evolve so large, complex, and intangible an object as a large software system without the aid of suitably visible, detailed and formal descriptions of how to proceed. In that process programs are such descriptions they offer an enormous advantage. Through a process program the manager of a project can communicate to workers, customers and other managers just what steps are to be taken in order to achieve product development or evolution goals. Workers, in particular, can benefit from process programs in that reading them should indicate the way in which work is to be coordinated and the way in which each individual's contribution is to fit with others' contributions.

Finally, in materializing software process descriptions it becomes possible to reuse them. At present key software process information is locked in the heads of software managers. It can be reused only when these individuals instantiate it and apply it to the execution of a specific software process. When these individuals are promoted, resign or die their software process knowledge disappears. Others who have studied their work may have anecdotal views of the underlying process descriptions, but these descriptions themselves vanish with the individual who conceived them. Obviously such process knowledge is a valuable commodity and ought to be preserved and passed on. Materializing it is a critical necessity.

The preceding discussion simply emphasizes that any vehicle for capturing software process knowledge is far better than no vehicle at all. As observed earlier, however, Pert charts and procedures manuals are such vehicles. We believe that process programming is superior to these other approaches, however, in that it enables far more complete and rigorous description of software processes. By defining a process programming language in which data objects, data aggregates and procedural details can be captured and expressed to arbitrary levels of detail we make it possible to express software processes with greater clarity and precision than has previously been possible. Further, as the process descriptions are to be expressed in a programming language, both the act of creating the descriptions and the act of reading and interpreting them should be comfortable for software professionals.

Thus process programs written in a suitable process programming language should be expected to be particularly effective media for communicating arbitrarily large and complex software processes between software professionals. This should not be surprising, as all languages are supposed to be media for communication. Going further, the fact that software process programs are to be expressed in a computer programming language suggests that this language should also be an effective medium for communicating these process descriptions to computers as well. Specifically, we proposed that another important reason for writing software process programs is so that they can be automatically analyzed, compiled and interpreted by computers.

Some readers must certainly have observed that our previous example, replete as it was with type definitions, declarations and non-trivial algorithmic procedures, was likely to contain errors. As it was written in a language with definable syntax and semantics, however, the possibility of parsing it, semantically analyzing it and conveying diagnostic information back to the writer and readers is quite a real one. Clearly as process descriptions grow to contain more detail, and the programming language in which they are written matures to have more semantic content, the process programmer can expect to receive more diagnostic feedback.

We see no reason why process programs written in such a rigorously defined language might not be compiled and executed. There would seem to be no reason why the testing process program presented in the previous section could not be executed on a computer once the lowest level procedures are defined. We have written a number of process programs which have been elaborated sufficiently far to suggest that

they could be directly executed on a computer. Other process programs, which have not been elaborated down to suitable levels of detail, can still be reasonably thought of as being interpretable, but only if the lowest level procedures are to be "executed" either by software tools or by humans.

This observation suggests that process programs can be thought of as vehicles for indicating how the actions of software tools might be integrated with human activities to support software processes. Thus the process programming viewpoint leads to a novel approach to designing software environments. In this approach, software objects are thought of as variables--or instances of types. Software tools are thought of as operators which transform software objects. Humans are accorded certain well defined roles in creating and transforming objects too. The specification of what they do, when they do it, and how they coordinate with each other and with their tools is embodied in the process program, and is thus orchestrated by the process programmer. It is important to stress that the process programmer, by leaving certain high level tasks unelaborated, thereby cedes to the human software practitioner correspondingly wide latitude in carrying out those tasks. Thus interpretable process programs do not necessarily unduly constrain or regiment humans. The level of control and direction provided to humans is under the control of the process programmer, who exercises this control by providing or withholding details of the tasks assigned.

Further, this seems a suitable time to repeat our earlier observation that we cannot currently be sure of the linguistic paradigm in which software process programs ought to be expressed. This question must be considered to be a research topic. Our example process program was algorithmic. Thus, humans interpreting parts of it would be guided by procedural instructions. Process programs written in a rule-based language would be guided by prescriptive rules and would presumably feel more free and unconstrained.

6. Processes as Software.

The foregoing discussion indicates why we believe that software process descriptions should be considered to be software. They are created in order to describe the way in which specific information products are to be systematically developed or modified. In this respect they are no different than a payroll program or a matrix inversion program. Further, our early work indicates that process programs can be written in compilable, interpretable programming languages which might bear a striking similarity to the languages in which conventional applications programs are written.

We believe that the primary difference between the process programs which we are suggesting and conventional programs is that process programs represent programming in a non-traditional application domain. This domain is distinguished first by the fact that its objects are perhaps larger, less well defined and more poorly understood than those in traditional application areas. Second, and more important, the product in this application area is not simply passive data, but includes yet another process definition (ie. an executable program for a traditional application area). Thus process programming is a more indirect activity. Its goal is the creation of a process description which guides the specification and development of an object which in turn guides the specification and development of another object which solves problems for end users. This doubly indirect process must be considered to be particularly tricky and error prone. It seems, however, to be the essence of software engineering. It is remarkable that we do it as well as we do. It seems self-evident that we could do it far better if process program software was materialized and made as tangible as the application software whose creation it guides.

The doubly indirect nature of software process programming is illustrated in the accompanying set of figures. Figure 1 shows an end-user's view of how an application program is used to solve a problem by creating information as a product. Here we see that the user's problem is solved through the instantiation of a process description (an executable application program), the binding of that instance to specific data which the user supplies, and the execution of a process which is defined by the process description (the application program). The effect of this process is to create information products, some of which are temporary and internal to the process and some of which are externalized and aggregated into the user's final product. Control of how this is done resides with the process which has been instantiated. It is worthwhile to observe that the process description which has been instantiated for this user is presumably available for instantiation for the benefit of other users. In such a case, these other instances are bound to different input information and carry out somewhat different processes to create different products. The end-user may be only dimly aware of the way in which the process description came into existence or the way in which it works.

Figure 2 shows a larger context in which the situation depicted in Figure 1 operates. In Figure 2 we see that the process description which has been instantiated for the benefit of the end-user is actually a part of a larger information aggregate, and that this information aggregate is the domain of an individual whom we refer to as a software practitioner. We refer to this larger information aggregate as a Software Product and see that it is the product which the software practitioner is responsible for creating. The software product contains such information objects as specifications, designs, documentation, testcases, test results, source code, object code, and executable code (the end-user's process description). Some of these objects are received as input directly from the software practitioner (just as some of the end-user's information product is received directly from the end-user). Additionally, however, much of the rest of the Software Product is derived by the action of a process (just as the end-user's process derives some information).

In both of these cases humans effect the creation of a product which is an information aggregate. The end user's product is derived through a process which is instantiated from a description created by someone else. Figure 2 does not make it clear how the software practitioner's product is derived.

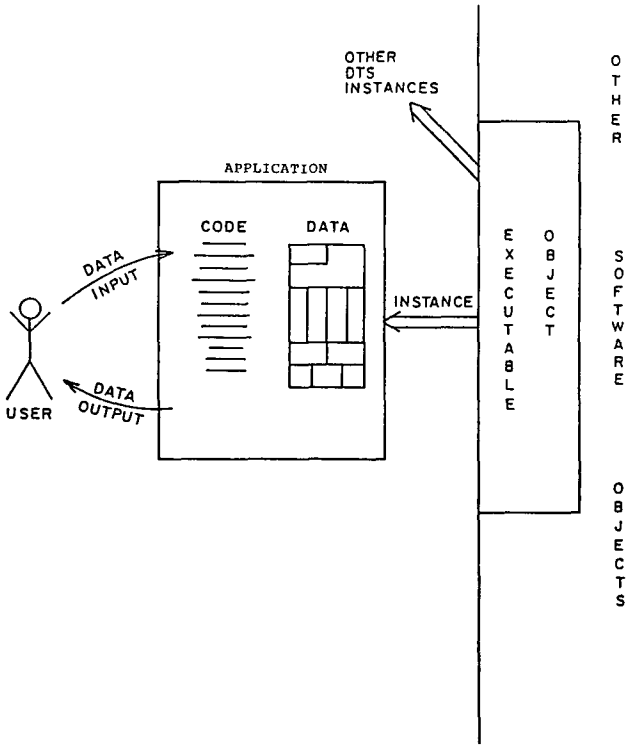

Figure 1

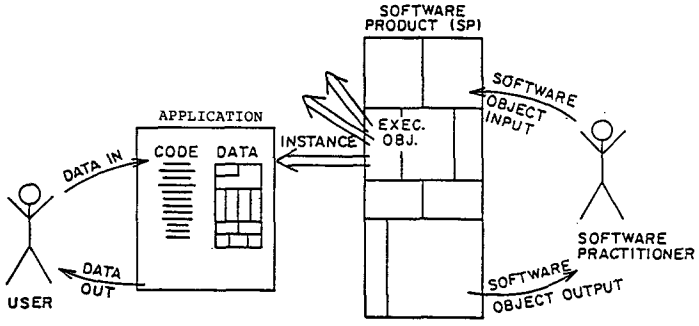

Figure 2

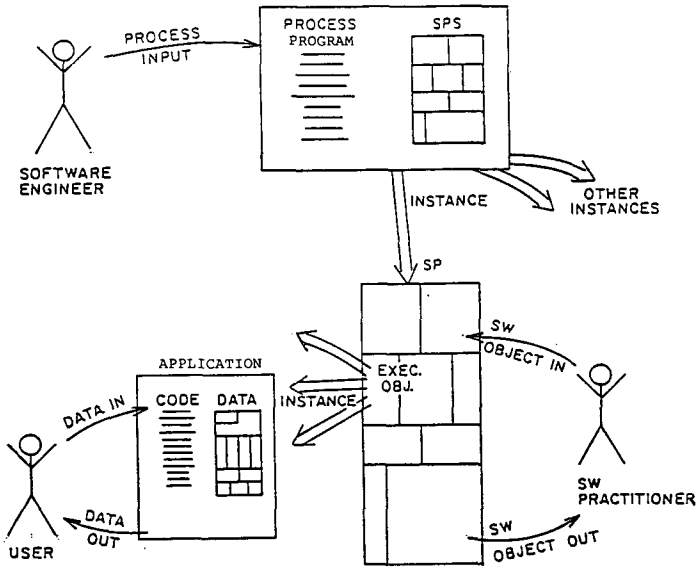

Figure 3

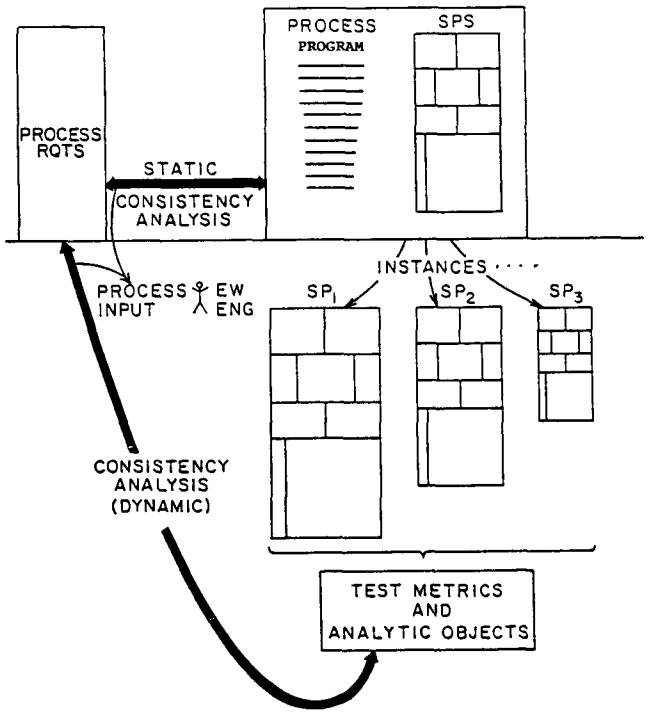

Figure 4

Figure 3 suggests how we believe this should be done. In Figure 3 we indicate that the Software Product should be viewed as an instance of a Software Product Structure (which we can think of as a very large template) and that the evaluation of the instance should be thought of as having been guided by the execution of a process program. We suggest that, just as the end-user may have only a dim understanding of the process which guides the creation of the end-user information product or where it came from, the software practitioner may also have only a dim understanding of the process program or where it came from. In each case the user is to be prompted from needed information, while the process accepts that information, uses it to derive more information, assembles that information, and produces the user's final information product. Certainly, at least for now, we expect that the level of involvement of the software practitioner in the process of creating the Software Product will be much higher than the usual level at which end-users are involved in the creation of their information products. This need for more involvement will result from the inability or reticence of the software process programmer to fully elaborate software process programs down to the level of directly executable code.

In that Figure 3 strongly suggests the underlying similarity of applications programs and process programs, it thereby strongly invites the question of whether process programs might themselves be considered to be instances of some higher level process description. In Figure 4 we indicate our belief that this is the case. Because software process programs are programs in what we now see to be the classical sense of this term, we should expect that they are best thought of as being only a part of a larger information aggregate. This information aggregate contains such other software objects as requirements specifications (for the process description itself), design information (which is used to guide the creation of the process description), and test cases (projects which were guided by processes instantiated by the process description).

There are a number of satisfying implications of the perspective that process program descriptions are themselves developed software. One such implication is that it is reasonable for there to be many different process descriptions that might be used to create similar software products. As process descriptions are developed to meet process requirements which may be expected to vary, clearly the design and implementation of these process descriptions should also be expected to vary. Thus it should be clear that there is no "ideal software process description." Software process descriptions should be created by software development processes which should take due regard of specialized requirements and constraints. In this sense software process programming is a true programming discipline. We believe that expert software managers have intuitively understood this for quite some time and have become adept at tailoring their processes to their needs. They do not seem to have appreciated their activities as programming activities, but it seems likely that the realization will be something of a relief.

Another satisfying implication of the view that software process descriptions emerge as part of a development lifecycle is that software process descriptions should undergo testing and evaluation. In fact this evaluation is carried out as the process description guides the evolution of individual software products. Successful process descriptions tend to get used again. Unsuccessful ones tend to be changed or discarded. The weakness of testing as the sole mechanism for software evaluation has been remarked upon before [Osterweil 81]. An integrated approach to evaluation incorporating static analysis and verification is clearly indicated. Once software process descriptions have been materialized as code it becomes quite reasonable to consider such other forms of evaluation. The result should be software processes which should be far more trustworthy.

The previous paragraph sketches some of the characteristics of what we might describe as a "process development process." We have begun to examine what a description of such a process might be like. It is also reasonable to consider "process evolution processes" as well. We have begun to consider what the nature of a software process evolution process description might be as well. These considerations have shed light upon the nature of what has been dubbed "maintenance."

There has been some fear that these considerations might lead to an ever-rising hierarchy of processes which produce processes, which produce processes which eventually actually produce end-user software. This fear seems groundless, as process programs and end-user programs do not seem to be essentially different. This leads to optimism that processes which we employ to develop and evolve end-user software products should also be applicable to the development and evolution of process program descriptions. This premise is a very important one and is being explored.

We have concluded that the notion of software process programming is a generalization, or extension to a new domain, of classical applications programming. The disciplines, techniques and tools of classical applications programming apply to process programming and can rapidly lead to the effective materialization of the descriptions of the processes which we use and their rapid improvement, maturation and archival storage for reuse.

7. Future Directions.

It should be clear that we believe that important contributions to software engineering can be made by pursuing the notion of process programming in any of a number of directions. In this section we indicate some of these directions, summarizing early work which has been done and continuations which seem indicated.

7.1. Software Process Studies.

We believe that great strides can be made in understanding software processes once they have been materialized. Materializing them in the form of process programs offers great hope because such programs are rigorous and

comfortable for both readers and writers. Discussion and evaluation of such software processes as testing [Howden 81] and design [JeffTPA 81] have been going on for quite some time. In addition debates about software development lifecycle models have been proceeding for well over a decade. We believe that these discussions have not been nearly as substantive and effective as they could and should be, largely because they have not been carried out in a rigorous and agreed-to form of discourse. Adoption of the idiom of process programming has the potential to precipitate rapid and significant advances in understanding the software lifecycle and the various of the subprocesses which comprise it, as process programming supplies a natural idiom in which to carry out these discussions.

At the University of Colorado we have begun to develop a variety of process programs and have found that these activities have almost invariably led to interesting insights into the processes under study. Much of this work has been done by graduate students who have had only a brief exposure to the idea of process programming. Their success in producing cogent and compelling process programs leads one to believe that this technique is rather easily motivated and rather easy to embark upon. These early experiences suggest that our instincts and experiences as programmers tend to lead us to worthwhile questions when they are applied to programming software processes.

We have written and compared process program definitions describing testing and evaluation. We believe that such process program definitions are best viewed as rigorized test plans. As such they elucidate both the test planning and the testing processes which we carry out. We have also written software requirements as elaborate data specifications. This exercise suggests some interesting ways in which requirements might be captured more rigorously. Again, our instincts and experiences seem to tend to guide us to interesting and worthwhile questions, suggesting new formalisms which seem to have much promise. We have also tried to capture some software lifecycle models as process program definitions. Glaring weaknesses in such well-known models as the "Waterfall" manifest themselves in the form of trivial and incomplete code.

Experimental process program description writing should continue. It seems certain to shed important light on the nature of our software processes. It also holds promise of providing a vehicle for sharply accelerated progress towards consensus on some processes, providing this formalism is adopted widely as a medium of discourse.

7.2. Process Programming Language Studies.

It is clear that the rapid progress which we have just described cannot be achieved until and unless there is some consensus about a language in which software processes and software products can be defined. In our earliest work we attempted to use simple language constructs in familiar linguistic paradigms. While quickly indicating the power of the process programming approach, these efforts also quickly served to show that powerful linguistic primitives are essential if process programs are to be sufficiently precise and powerful.

Software products can only be described as very complex data aggregates. Thus powerful type definition and data aggregation mechanisms must be included in any process programming language. A full range of control flow mechanisms also seems needed if the language is to be capable of conventional algorithmic expressions. Alternation and looping are clearly crucial parts of familiar software processes. We were surprised to observe, moreover, that these processes could not be expressed effectively without the use of concurrency specifications. On reflection this should be no surprise. Human processes are clearly highly concurrent, thus they can best be expressed using concurrency primitives. In addition, it seems that the scoping and accessing rules for a process programming language must be sophisticated. Our early work has shown that classical hierarchical scoping rules are not adequate to describe the complex ways in which software subprocesses must communicate with each other. Rigid message-passing mechanisms have also been shown to be unequal to the stringent information sharing requirements of actual software processes.

All of these early results indicate that significant research is needed to produce the definition of a suitable language. As observed earlier, this research cannot be restricted to an examination only of algorithmic languages. We have produced some very provocative process programs using informal languages which borrowed freely from the rule-based, object-oriented and applicative linguistic paradigms. It seems likely that only a mixed paradigm or wide spectrum language will prove adequate for comfortably expressing software process programs.

7.3. Software Environment Architecture Research.

As noted earlier, one of the more exciting consequences of this work is that it has suggested a novel architecture for a software environment. We now believe that a software environment is best viewed as a vehicle for the specification of process programs, and for their compilation and interpretation. In such an environment tools would be treated as operators, or lowest level software processes, whose jobs were defined in terms of the need to create and transform software objects which would be treated as instances of types. These types would correspond to different types of intermediate and final software products. Humans would participate in executing such software processes by serving as the execution devices for subprocesses which were not elaborated to sufficient levels of detail to enable interpretation by either tools or the host computing system execution environment.

One of the more powerful suggestions of this work is that software processes can themselves be treated as software objects by an environment. We believe that development and evolution processes can be produced in such a way that they specify how both application programs and process pro-

grams are created and maintained. As software processes are software objects, and our environment architecture treats all software objects as instances of types, this suggests that software processes might well be organized by some sort of type hierarchy. We have begun pursuing the notion that various software processes can be organized into such a hierarchy and can, perhaps, be characterized by the operational characteristics that they inherit and/or mix in. We propose to explore these hypotheses by experimentally building and using a process programming environment. Arcadia is a project to create just such an environment [Arcadia 86]. The first prototype environment, to be called Arcadia-1, will provide a testbed for the evaluation of many of our environment architectural ideas, as well as a vehicle for experimenting with process programming.

7.4. Software Metrics.

One of the most gratifying aspects of our exploration of the notion of process programming is that it has led to insights and research directions which were not expected outgrowths of the work. Once such outgrowth has been some promising ideas about software metrics. It seems clear that in materializing software process descriptions we are creating a natural subject for measurement. If a software process such as testing or development is to be thought of as the execution of a process program, it seems reasonable to measure the size of the product in terms of the size of the objects declared within the process program, and to measure the degree of completion of the process in terms of the position of the execution pointer in the process program code. Static analysis of process programs could likewise lead to promising measures of the complexity of the corresponding software processes.

We make no claims that such software metrics are necessarily superior to existing metrics, but we do suggest that comparison of such new metrics to more classical ones warrants research attention.

7.5. Implications for Software Reuse.

Another unexpected outcome of this line of inquiry has been a somewhat different understanding of the nature of software reuse and the problems in effectively achieving this important goal. We believe that the goal of software reuse is the successful integration into an evolving software product of software object(s) which were developed as part of a different development process. That being the case, reuse entails the careful assimilation of the reused objects into the new product. Given that software products are notorious for their intricate interconnectivity, it seems evident that only a complex process will suffice for assuring that the reused objects are properly interwoven into their new context, and properly evaluated in that new context. Thus effective reuse can only be achieved through the execution of a suitable process which should be defined by means of a suitable reuse process program. It is interesting to note that managers often observe that reuse "must be planned for." From our perspective that means that reuse processes must have been previously programmed. Further, these programs must be executed only at the proper points in the larger process of developing the reusing software. We conjecture that software reuse can only be expected to be a realistic prospect when the structure of the reused software closely matches the structure of the reusing software, and when the process by which the reused software was developed closely matches the process by which the reusing software is being developed.

Finally, it seems important to repeat our observation that perhaps the most important benefit of process programming is that it offers the hope that software processes themselves can be reused. We believe that effective software process descriptions are one of the most valuable resources which we as a society have. The realization that these resources are never effectively preserved beyond the working lifetime of the people who execute them is a sobering realization. We look forward to the prospect that process programs which have been shown to be effective can one day be captured rigorously and completely and made part of libraries of reusable software process programs. Such reusable process programs would then become the modular parts out of which new process definitions could be fashioned or adapted. We expect that early process programs will be produced from scratch by software engineering researchers, but that in the future process programs will be customized by working engineers out of standard process programs.

8. Conclusion.

In this paper we have suggested that the notion of a "process program "--namely an object which has been created by a development process, and which is itself a software process description--should become a key focus of software engineering research and practice. We believe that the essence of software engineering is the study of effective ways of developing process programs and of maintaining their effectiveness in the face of the need to make changes.

The main suggestions presented here revolve around the notion that process programs must be defined in a precise, powerful and rigorous formalism, and that once this has been done, the key activities of development and evolution of both process programs themselves and applications programs can and should be carried out in a more or less uniform way.

This strongly suggests the importance of devising a process programming language and a software environment capable of compiling and interpreting process programs written in that language. Such an environment would become a vehicle for the organization of tools for facilitating development and maintenance of both the specified process, and the process program itself. It would also provide a much needed mechanism for providing substantive support for software measurement and management.

We are convinced that vigorous research directed towards 1) the creation of a process programming language, 2) the construction of a compilation and interpretation system for programs written in it and 3) the use of these tools in the careful description of key software processes will surely be of enormous value in hastening the maturation of software engineer-

ing as a discipline.

9. Acknowledgments.

The author gratefully acknowledges that this work was made possible by the generous support of the National Science Foundation, through Grant #DCR 1537610, the US Department of Energy through Grant #1537612, and The American Telephone and Telegraph Company. In addition the author wishes to thank Professor John Buxton of King's College, University of London, Professor Manny Lehman of Imperial College, University of London, and Dr. Brian Ford of Nag, Ltd., Oxford, England, for their help, encouragement and numerous challenging conversations during Academic Year 1985-86, while the author formulated these ideas while on leave of absence in England. In addition numerous stimulating conversations and interchanges with Stu Feldman, Dick Taylor, Bob Balzer, Geoff Clemm, Lori Clarke, Dennis Heimbigner, Stan Sutton, Shehab Gamelel-Din, Brigham Bell, Steve Krane, Steve Squires, Bill Scherlis, Frank Belz and Barry Boehm also helped to shape these ideas significantly. Finally the author thanks the students in Computer Science 582, Fall 1986, for their patience in trying to understand process programming and their energy and enthusiasm in producing an impressive base of process programs.

REFERENCES

[Arcadia 86] R.N. Taylor, L.A. Clarke, L.J. Osterweil, R.W. Selby, J.C. Wileden, A. Wolf and M. Young, Arcadia: A Software Development Environment Research Project, ACM/IEEE Symposium on Ada Tools and Environments, Miami, Florida, April 1986.

[BoehmMU 75] B. Boehm, R. McClean, D. Urfrig, "Some Experiments with Automated Aids to the Design of Large Scale Reliable Software," *IEEE Trans. on Software Eng.*, SE-1, pp. 125-133 (1975).

[Dijkstra 68] Dijkstra, Edsger W., "Go To Statement Considered Harmful," *CACM* 11 pp. 147-148 (March 1968).

[Howden 85] Howden, W.E., "The Theory and Practice of Functional Testing," *IEEE Software,* 2 pp. 6-17 (Sept. 1985).

[JeffTPA 81] R.Jeffries, A.Turner, P.Polson, M.Atwood, "The Processes Involved in Designing Software," in *Cognitive Skills and Their Acquisition* (Anderson, ed.) Lawrence Erlbaum, Hillsdale, NJ, 1981.

[Knuth 68] Knuth, Donald E., *The Art of Computer Programming, V.1--Fundamental Algorithms* Addison Wesley, Reading, MA 1968.

[Osterweil 81] L. J. Osterweil, "Using Data Flow Tools in Software Engineering," in *Program Flow Analysis: Theory and Application,* (Muchnick and Jones, eds.) Prentice-Hall Englewood Cliffs, N.J., 1981.

[SPW1 84] Proceedings of Software Process Workshop, Runnymede, England, February 1984.

[SPW2 85] Proceedings of Second Software Process Workshop, Coto de Caza, CA, March 1985.