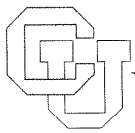
A Process-Object Centered View of Software Environment Architecture *

Leon Osterweil

CU-CS-332-86



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* The authors gratefully acknowledge the support of the National Science Foundation grant #DCR-8745444 in cooperation with the Defense Advanced Research Project Agency and the IBM Corporation.

A PROCESS-OBJECT CENTERED VIEW OF SOFTWARE ENVIRONMENT ARCHITECTURE

Leon Osterweil

CU-CS-332-86 March 1988

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

The authors gratefully acknowledge the support of the National Science Foundation grant #DCR-8745444 in cooperation with the Defense Advanced Research Project Agency and the IBM Corporation.

1. Introduction.

The essential purpose of a software environment is to provide strong, complete and readily accessible support for such key software processes as development and maintenance. The basis of such support must be a diverse and powerful set of functional capabilities supplied by what has previously been referred to as "software tools". Increasingly, however, it is becoming clear that the most challenging part of creating an effective software environment is not the creation of the software tools themselves, but rather the effective integration of those tools and presentation of their capabilities to the user.

Further, it is becoming quite clear that a software environment is very much like all other complex pieces of software in that it must be capable of growing, adapting and changing to meet the ever-changing needs of its users. Indeed, as software environments are a relatively new type of software product, it must be expected that our understanding of the requirements for this type of product is at a crude and primitive level. This makes it all the more important to be sure that the architectures which we develop for these systems be as flexible as possible. It also suggests that it behooves us to seek general--and probably new--paradigms and conceptual frameworks with which to organize and help understand the problems for which software environments are purported to be the solution.

In this paper we advance the notion that the key software processes of development and maintenance can be effectively modelled as an unexpectedly complex and intertwined collection of subprocesses supporting and communicating with each other through the exchange of software products. We further suggest that both these products and the processes themselves can profitably be viewed as software objects.

This leads to an architectural view of software development and maintenance which is elegantly uniform and encompassing, while also being quite amenable to extension and modification across a broad spectrum. Thus we believe that it a promising framework within which to study and understand the nature of software processes. We also believe that it suggests an architecture for a family of powerful, robust and flexible software environments which are good candidates for active partnership in experimental research which should be successful in leading to a fuller understanding and explication of software processes and the environment mechanisms capable of effectively supporting them.

2. Background and Related Work.

We believe that the more successful software environment efforts in the past

have been most successful where they have attempted to convey to users the sense that the essence of their work was the creation of information in the form of software objects and products. In that such efforts have concealed the presence and actions of tools they have succeeded. In place of confronting users with the need to understand and manage tools, they have required users to understand and manage data objects and structures necessary to the creation of software end-products. Certainly in the process of managing software objects and products it has been necessary to manage their transformation as well. Here the need for tools arises, but here the most successful environment efforts have attempted to reduce the user view of the needed tools to the most simplified minimum--arriving at the view that tools are best thought of as functions or operators.

Some examples of environments which foster and support this view are Toolpack/IST [Osterweil 83, ClemOst 86], Smalltalk [Goldberg 84] and Interlisp [TeitMas 81]. In all cases the user of the environment is encouraged to think of his or her job as being the creation or alteration of software objects through functional transformations which either create new objects from scratch or transform older objects. These environments are viewed as being predominantly interactive, with user direction coming essentially one command at a time. In this we believe that these older environments have missed an opportunity to be more successful and to link up with another fruitful and active line of research.

We believe it is important to recognize that the user of a software environment does not issue random commands one at a time to an environment, but rather attempts to follow an orderly procedure for fashioning more highly developed products out of collections of software piece-parts. If one were to capture the stream of commands issued to a software environment such as Toolpack/IST or Smalltalk, one would presumably have the equivalent of an execution history trace taken through a procedure intended to create software product(s) in an orderly way. At present such procedures are kept informally in the minds of software practitioners. It is significant to observe, however, that increasingly researchers are attempting to realize and formalize these procedures. The most visible attack on this problem is work aimed at studying "The Software Process" [SPW1, SPW2]. But other research efforts [JeffTPA 81] have also been aimed at studying and formalizing such activities as software design.

Our hypothesis is that the best way of understanding how software is developed and maintained is to view these activities as coordinated sets of processes, rather than as a monolithic Process, and to think of these processes as being definable and expressible in actual well-defined procedural, algorithmic code. This view urges us strongly in the direction of viewing the processes used to effect software development and maintenance as pieces of software themselves. This suggests that software processes should be viewed as products which have

to be developed and maintained in just the way that more classical software products also have to be developed and maintained.

This in turn suggests that there may be a great deal to be gained by extending the scope of what can be considered to be a software object to include software processes. Among the most important gains seems to be the possibility that the tools and procedures applicable to development and maintenance of classical software objects might also prove to be applicable and effective in developing and maintaining themselves.

Tempering this sanguine view of software processes as objects is the realization that software processes can also be activated and caused to operate on other software objects. Thus software processes seem to have a dual nature--passive objects upon which functional tools can operate and active operators capable of application to passive objects. This observation does not invalidate our view of environment architecture, but rather makes it clear that the architecture must recognize this dual nature of software processes and must, in fact, be a positive aid in effectively managing the duality.

The reward for doing this effectively is the widening of the scope of a software environment to encompass the processes which drive the application of tools to the development and management of software products. This widening should be a significant help to the user in carrying out the necessarily complex software processes. It should also enable precise visibility into the nature and functioning of software processes, thereby expediting the experimental evaluation and rapid evolution of software processes and the tools and aids by which they can most effectively be supported.

These ideas are more fully developed and more carefully presented in [Osterweil 86]. The interested reader is encouraged to explore the details and ramifications of these ideas in that paper. For the present paper, however, we proceed by indicating the implications that these ideas have for the architecture of a software environment predicated upon the desirability of supporting this view through the application of tools.

3. Process-Object Environment Architecture.

If we liken an environment's tool capabilities to the primitive capabilities furnished by a computer's underlying hardware, then the essential job of the environment can be likened to the job of a programming language, which must help users utilize these capabilities effectively and also encourage thinking and working at a higher, cleaner, and more concise level of abstraction. This analogy is well drawn and useful. Both environments and programming languages

enable the composition and configuration of functional capabilities, and must store and tranfer information in order to do so. Both must exchange information with external media and with users as well. More recently it has become clear that both must also concern themselves with supporting the cooperative activities of many users.

Further, just as programming languages have been built to support the creation of processes for manipulating data (classical application programs), so we believe that the goal of an environment should likewise be to support the creation of processes for manipulating software. Examples of such processes are software development and software maintenance. Some very high level and very primitive examples of such processes can be found in [Osterweil 86].

Thus, a software environment must support the creation, assembly and manipulation of software objects and it must provide facilities for defining the way in which its tools operate upon these objects. In addition, however, the environment must also provide capabilities for orchestrating the application of these capabilities as processes.

Viewing an environment as a device for fashioning and executing precise descriptions of key software processes helps us to establish some basic architectural features. Consider, for example, software development--the most obvious software process. Development effects the creation of a software product, which should be viewed as a complex structure of such software objects as source text, executables, testcases, design criteria and documentation.

Hence development is the process of creating and aggregating software objects with tools. We believe that this and all other processes should be rigorously and precisely specified. Following the programming language metaphor we are led to consider the utility of typing. As shall be demonstrated, we have determined that superior process rigor and precision can be advanced by considering all software objects to be instances of types and all software tools to be operators whose operands are these type instances.

One important implication of this view is that software development can take place only after operand types have been carefully defined, tool functions have been rigorously expressed, and the development procedure itself has previously been considered, and defined. Clearly quite a bit of work must take place before the development process can begin. This work, moreover, must itself be precise and disciplined as its product is a precise and disciplined development process. In fact, it seems clear to us that this work should itself be defined as a process--namely a software process development process, or PDP.

Thus our belief that a software environment shares most of the goals and

methods of a modern programming language and our belief that software processes can and should be expressed rigorously in algorithmic code have led us to conclude that in order to assure that a software development process (DP for short) produces highest quality products, the DP should first be thought of as itself the product of a PDP, the process aimed at the creation of the DP and the structures in terms of which it is defined. Shortly we shall see that this notion is useful in helping to conceptualize, control and organize the diverse activities supported by a software environment.

Consideration of the nature of software maintenance is also useful in this regard. The goal of software maintenance is to alter a software product. So far we have discussed two very different products--the DP and the more conventional software products which the DP produces. Maintenance may be aimed at altering either or both. Alteration of the convential software product is more straightforward, and we shall refer to it as product maintenance. It can be viewed as a simple case of a more difficult and significant type of maintenance, which we shall refer to as process maintenance. Process maintenance entails altering the DP, analyzing the software products which had been produced by the original DP, and altering these software products to make them consistent with the new DP.

An important example of process maintenance is the integration of a new tool into an environment. The new tool is presumably to be integrated because it is capable of creating objects of a new type or because it is more efficient in creating objects of existing types. Consequently, the development process description must also be changed to indicate where and how the new tool is to be used.

This example is particularly important because of our belief that environments must be extensible in order both to accomodate evolving tools and techniques, and to support experimentation and evaluation which must drive and direct such evolution. Thus software environments must be extensible, incorporating effective and orderly procedures for the insertion of new tools and types. The foregoing discussion indicates that procedures are not insignificant, suggesting the desirability of carefully defining them and carrying them out by means of still another type of process--namely a process maintenance process (PMP).

Thus we have introduced three separate but closely related types of processes which must be supported by an environment--development processes (DP's), process development processes (PDP's) and process maintenance processes (PMP's). In order to be sufficiently powerful and effective, a software environment must recognize the legitimacy and separate characteristics of all three, must support the careful definition of all three, and must offer tool and procedural support for all three. Indeed, we believe that effective software maintenance within a rigorous framework cannot be carried out without

acknowledging and supporting all three.

We observe in passing that these three processes are by no means the only processes which a software environment must support. Some other processes, and considerable further discussion of the three we have already mentioned can be found in [Osterweil 86].

It is important not to leave the reader with a pessimistic view that software development cannot begin until a bewilderingly complex web of processes has been built from scratch. Instead, we must hasten to point out that it is likely that these different types of processes may share significant amounts of substructure and tool support and that, as all are objects, many might well be expected to have been developed by alteration or adaptation of other, preexisting processes.

3.1. Objects, Types, Tools and Processes.

We now proceed to more specific discussions of objects, types, tools and processes--the key architectural components of a software environment. We recall that an environment is best understood as a system for supporting the rigorous definition and subsequent interpretive execution of software process descriptions, where these descriptions are expressed by means of a language which supports the rigorous definition of objects, types and operators. Relatively little will be said about the flow of control facilities of the language. Preliminary work indicates, however, that the language will have to contain extensive flow of control and proceduring mechanisms and will also have to support structures for concurrency and task synchronization. This point is addressed in some detail in [Osterweil 86].

3.2. Object Management.

We have argued that tools should be viewed as functional tranformers acting upon objects in much the same way that machine language instructions act upon their operands. In both cases control and discipline result from allowing operands to be accessed only by a small selected set of accessing primitives and basic manipulative functions (which can, of course, be composed to produce higher level functionality).

This is how hardware systems implement primitive types, such as integer and real. Modern programming languages provide the user with additional types which are not implemented in hardware, but are offered to the user in a similarly restricted and disciplined fashion. Some languages also offer the user facilities for creating new types--abstract data types--in which use and access are

restricted and disciplined in the same way.

Data types in a process description language will be defined in this way also. Instances of a type will only be created, accessed and manipulated through a small set of previously defined functional primitives. Tools will be highly modular, generally being composed out of a number of smaller tool fragments. These tool fragments will all access a needed object only through an accessing primitive for the type of which the object is an instance. This will enable tool fragments to share, manipulate, and elaborate upon objects created by other tool fragments while also incorporating the sort of discipline in object access that will insure the integrity, modifiablility and extensibility of the environment.

Unfortunately current computer systems do not assure that persistent objects will always be accessed only in such a disciplined way. Such discipline can be assured within any single execution of a program written in a language such as Ada (TM). After termination of the execution of that program, however, persistent objects are generally made accessible and alterable with ordinary utility tools. Data base management systems often enforce such disciplined access control, but usually only for a very limited class of types (e.g., integers, reals, and character strings) and structures (e.g., relations). Moreover, they present a view of storage as one monolithic unit (e.g., the data base) with one set of general purpose operations.

Consequently, we have concluded that a software environment which implements the architectural view we are suggesting will have to incorporate its own object management subsystem for providing the sort of disciplined object access required to assure integrity, flexibility and extensibility.

3.2.1. Objects.

The objects which seem to require management in a software environment range widely in size and character. Small objects, such as tokens and graph edges, require management, but so do large objects such as entire programs or diagnostic output objects. Objects may be as diverse as text, object code, test data, symbol tables or bitmap display frames. One organizing premise which seems most helpful is to require that each object be considered to be an instance of a type. This by itself does not seem to provide enough structure and organization to support effective object management, however.

In addition, it seems important to recognize that each object is also generally related to a number of other objects in the store by a potentially large variety of types of relations. These interobject relations should be stored explicitly as objects in order to be of maximum utility. Some examples of types of

interobject relations are: versioning, consistency, derivation and hierarchy.

Hierarchy relations enable users to, for example, model the inherent structure of systems being developed or maintained. Hierarchical structures should be implemented in such a way as to allow objects to be grouped, and to allow such groups to be objects which can then be included in other groups. It should be possible for these groups to overlap, making it possible for a user to include an object in several hierarchical structures. This inclusion should be logical rather than physical, however, to eliminate duplication of storage or difficulties in properly reflecting updates of shared objects. Our experience with a similar capability in the Toolpack project [Osterweil 83, ClemOst 86] indicates that this capability encourages users to think and operate at a higher level of abstraction.

The types, of which individual objects are instances, should also form a hierarchy. Types should be structured by a subtype relation similar to that in Smalltalk 80, in which subtypes inherit accessing primitives from their supertypes. Thus, for example, the type "flowgraph" might well have "set/use annotated flowgraph" as a subtype, in which case "set/use annotated flowgraph" objects will be manipulable by all of the accessing primitives applicable to objects of type "flowgraph." This should simplify the implementation, exploitation and conceptualization of both types and type instances (objects) in environments of the sort we are suggesting here.

It is worthwhile to point out that the objects of given type may also be organized hierarchically, and that their hierarchy may not bear any relation to the hierarchical structure which includes their types.

It is particularly important that the object store incorporate explicit consistency relations among its objects as consistency is the property whose establishment is the goal of testing and evaluation. Testing and evaluation seek to study the presence or absence of errors, and errors are essentially inconsistencies among software objects. Most often an error is an inconsistency between a specification of intent and a solution specification (eg. see [Osterweil 82, Osterweil 81]). Thus testing and evaluation tools can appropriately be viewed as operators aimed at either establishing or disproving consistency relations. It seems most logical and effective for these relations to be explicit.

The value of explicitly storing the derivation relation among objects will be addressed shortly.

It is important to point out that types and the type hierarchy itself should all be treated as objects. This observation is useful in further explaining our earlier comments on maintenance. As observed earlier, it is vital that an

environment support the alteration or addition of new types to a DP. As types are to be defined as clusters of accessing primitives and are organized into a hierarchy, type alteration or addition entails potentially complex transformations to these clusters and hierarchy. That is why we believe that such alteration or addition is best thought of and carried out as a well-defined algorithmic procedure (a PMP). Certainly a PMP must be defined in terms of operations on objects, types and type structures are the objects.

This observation makes it clear that an environment cannot be viewed as a single process, but rather as an interconnected collection of processes. An environment must support a PDP by enabling the definition and manipulation of types and type structures as objects. These objects can then be frozen and perpetuated for use by the DP. Within a DP, however, it is generally desirable that these types not be considered objects, but rather collections of executable code procedures. Within a DP the integrity of a type definition can be maintained by stripping away any primitives capable of accessing the type definition primitives. This would imply that the type (ie. its collection of accessing primitives) could only be changed by a PMP within which the type is an object by virtue of the inclusion of primitives capable of accessing the type's accessing primitives. Such an approach offers the advantages of 1)control over the how a type is defined, 2) discipline in how type instances must be accessed and altered, but 3) a controlled and careful way in which type definitions can be altered.

3.2.2. Persistence.

An environment must also support persistence--the ability to save objects beyond the end of execution of any of the tool or processes that manipulate it. Nevertheless, the environment must assure that all accesses to that object are only through the object's accessing functions. Thus, an object should be allowed to persist only if its accessing primitives persist as well. Thus an instance of an object can not be separated from the defined operations on that object.

Returning to our previous discussion of types, we note that a type object does not persist from a PDP through to a DP, because it is no longer an object during the DP. The type object does persist from a defining PDP through to an altering PMP, because during the PMP it is possible to change the type definition. This necessitates the persistence of all of the type object's accessing primitives and the persistence of the type object as an object.

For the most part, tools should be unaware of the persistence of the objects that they manipulate. Thus tools should access objects without knowing how or

when they were created. Objects produced by intermediate tool fragments, in response to user requests should be saved by the environment's object manager according to some algorithm for evaluating potential for future reuse. These persistent intermediate objects should then be available for reuse in responding to a subsequent user request, thereby expediting the environment's response. This notion was advanced and implemented in the Toolpack project through the Odin integration system [Clemm 86, ClemOst 86].

3.3. Tool Management.

Tool management capabilities complement object management capabilities. This is appropriate and logical because tools are viewed as functional tranducers defined on objects, and objects are instances of types defined in terms of functional primitives. Thus in this section we stress the relationships among tools and objects as a primary focal point of our discussion.

3.3.1. Tool Structure.

An environment must be a vehicle for providing extensive and growing tool capabilities. These capabilities should be furnished primarily by collections of small tools--tool fragments-- rather than by a few, large, monolithic tools. For example, the task of prettyprinting should be carried out by a collection of tool fragments including a lexical analyzer, a parser, and a formatter-- operating in concert. More sophisticated prettyprinting functions require the invocation of a static semantic analyzer. An instrumented test execution requires upwards of a dozen tool fragments, operating at times in sequence and at times in parallel.

One of the main advantages of composing larger tools out of smaller fragments is that, if the tool fragments are well chosen, they will prove to be usable as components of a variety of larger tools, thereby enabling the creation of these larger tools at lower cost. For example, both the prettyprinter and dynamic instrumentation tool just mentioned require lexical analysis and parsing in order to begin their work. Both tools incorporate these fragments, thereby saving the creators of these tools the effort of having to recreate these functional capabilities.

The tool fragment architecture also leads to the materialization of intermediate objects which, if made persistent, can, as noted above, lead to efficiency through reuse. We now see that an environment can support both reuse of tool fragments in the creation of tools during a PDP and reuse of intermediate software objects during a corresponding DP.

Another benefit of the tool fragment architecture is that it encourages toolmakers to think in terms of good modular decomposition for their tools and good organization for the objects which their tools use. Thus, we believe that the writer of a prettyprinter, for example, will create a better tool because correspondingly more time can be spent on the problem of prettyprinting, and no time need be wasted on creating a parser. In addition, because the writer of the prettyprinter accesses the output of a proven parser, the prettyprinter is likely to be better for its reliance on a more robust and thorough parser than would likely have been created from scratch. In short, we contend that the prettyprinter will be a better tool because it will be based conceptually upon such data and software constructs as lexical tokens, parse trees, and symbol tables and will be based physically upon major bodies of robust proven code.

3.3.2. Tool Structure and Relationships.

In previous sections we have described numerous benefits of treating tools as transformers of instances of types. We noted that this discipline makes it possible to prevent the application of tools to inappropriate objects; that it supports the definition of tool invocation sequences which rigorously stipulate potentially reusable intermediate products of such sequences. We shall also see shortly that it also contributes to version control, demand-driven (lazy) rederivation of modified objects, and a useful form of inferencing that can be exploited by planning tools.

In this section we see that it is also the basis for definition of a structure which relates tools-- the Type Dependency Graph (TDG). A TDG is a structure which contains as its nodes all of the types known to the environment's object manager, and has as its edges annotated designations of the various tools. Thus, a TDG summarizes the relationships between tools and objects. Specifically, a tool may only operate on objects that are instances of the types found at the origin of an edge in the TDG bearing that tool's designation. Similarly, the only tools that can produce objects of a given type are those whose designation appears on an edge that terminates in a TDG node corresponding to that type.

A TDG induces derivation relationships on the actual objects in the environment object store. This type of relation among objects was alluded to earlier. Objects, being instances of the types found in a TDG, are related by dependency relationships corresponding to those defined by (some subset of) the TDG edges. These relationships among objects then may be viewed as an Object Dependency Graph (ODG), created and maintained by interaction of the tool and object managers. In particular, newly created objects are said to depend for their creation upon the existence of their predecessors, in which case the object

manager records this dependency relation by logically making the new objects descendants of their predecessors in the Object Dependency Graph (ODG).

Each object can be viewed as the root of a subtree of the ODG which we refer to as the object's own Dependency Tree. To be most useful and effective, this subtree should be doubly linked--upwards and downwards. The uplinks of the tree can be followed in order to determine all of the ancestor objects which were used, either directly or indirectly, to create the object; the downlinks of the Dependency Tree indicate which objects depend upon the object for their creation.

Dependency trees have already been exploited in certain version control systems such as RCS [Tichy 83]. In such systems, all descendants are created by the action of a single tool, generally a text editor. It is assumed that there is a single root version, and that this version has a tree of descendants, each of which can be built by successive applications of the tool. One value of this scheme that it helps make clear the potential range of the impact of changes to any one version.

We are suggesting the use of a similar dependency structure within the object store, but this structure is developed, not by successive applications of a single tool, but by applications of any legal sequence of tools. Thus, a user may use the environment to create a Dependency Tree for versions of source code as in RCS, or may create a complex tree in which objects of various types, the results of different sequences of tool applications, are all stored in a single Dependency Tree.

Such an environment must also incorporate algorithms for determining when objects at lower levels of a Dependency Tree have become obsolete because of alterations or deletions of objects higher up in the tree. Whenever an object at a higher level of the tree is changed, the object manager must recognize that all descendants of that object are to be viewed with suspicion. The environment need not take immediate steps to recreate these objects, however. Instead, it can employ a demand-driven reconstruction strategy such as that implemented in Odin [Clemm 86, ClemOst 86], under which new versions of an object are not created until they have been requested either directly or indirectly by the user. At that time, all the objects reachable by traversing uplinks from the object in the Dependency Tree are examined. If the object was, in fact, created from ancestor objects which have subsequently been altered, the object manager must begin the process of recomputing it from the current version of the ancestor objects. Objects between the altered ancestors and the desired object must be reconstructed. They must also be compared to their previous versions. If, at any point, the reconstructed objects match their previous versions, the reconstruction process can stop and the equivalence of old and new versions of

objects lower in the tree relied upon.

3.3.3. Planning Tool Activations.

In general we have learned that the functional capabilities which users require and expect from tools tend to become so complex that it is difficult to determine in advance the order in which tool fragments will have to be invoked in order to derive a requested object from existing objects. For example, a data flow analyzer (eg. [FosdOst 76]) must analyze the compilation units of a program in two passes, where the order of analysis during the second pass is computed during the first pass. In this case it is impossible to predetermine the exact order of invocation of tool fragments on the separate software objects. An environment must support the synthesis of such tools. One way in which this can be done is by means of a planning tool fragment whose job is to dynamically create tool invocation sequences that are tailored and adjusted in accordance with the current state of the object store and preprogrammed conditions. Planner tools have been used in Odin to create tool invocation sequences in which tools are scheduled for future invocation and in which software objects are taken and produced in unexpected or changed orders. In addition, planners have been used to schedule the invocation of other planners at projected future critical points.

Finally it is important to point out that an environment must support the notion of active as well as passive tools. Active tools commence execution without direct invocation by users. They carry out their activities by acccessing objects according to plans designed in advance, probably by software engineers during the PDP. These active tools must be invokable by such control mechanisms as timers or daemons, whose job is to detect relevant changes in the object store.

This capability seems to be a particularly useful one to a PMP. Specifically, we expect that a maintenance process would have to incorporate a task consisting of or containing an active tool whose job would be to periodically, and on its own initiative, recompute certain prespecified consistency relations to see if changes had altered the value of the relations from true to false. If so, then the need for changes to other objects is indicated.

3.3.4. Tools as Objects.

In the environment architecture which we are proposing here tools and tool fragments are objects in precisely the same sense in which types are objects. Similarly, the Type Dependency Graph (TDG) is an object in the same sense in

which a type structure is an object. Tools, tool fragments and TDG's are objects during a PDP, but are not objects during a DP. Thus, tools, tool fragments, and the TDG are key components and agents of any DP, but they have no accessing primitives associated with them and cannot be altered by or during a DP. On the other hand all are objects during a PDP and can therefore be altered. Of course, as with alterations to types, such alterations need to be communicated to, and their effects assimilated by, affected DP's. This is one of the jobs of a PMP.

The delineation of these three processes and identification of the objects upon which they operate has served to sharpen our appreciation of the functional roles of the various users of a software environment. For example, the internal composition of tools should be of limited interest to those using the tools as part of a DP, but must necessarily be a primary concern of tool developers as part of a PDP Thus tool developers are expected to develop and alter tools as part of a PDP. As part of that process they should be able to use tool and object managers to access tool specification objects, current tool fragment objects, and development object types to assist them in the creation (or modification) of new tool objects, and the alteration of TDG's to accomodate such additions and alterations.

Once tools or types have been changed, a PMP must evaluate the changes made and propagate their effects carefully and efficiently. For example, if an existing object type is modified in such a way that there are no changes in the set of accessing primitives that define the type, then a PMP should not need to make any consequent changes to a DP using that type and incorporating instances of that type.

On the other hand, if any of the accessing primitives defining that type are changed, then a new type is thereby created. This necessarily affects all tool fragments which either created or used objects of that type, and all DP's using the type and such tool fragments. Changes to tools, TDG's and impacted processes must be determined and made as part of a PMP. Clearly careful attention must be paid to this process and to the creation of new tools which might be of particular value in supporting the process. In particular, it may be desirable to create a tool that can transform objects of the old type to objects of the new, and incorporate a corresponding edge in the TDG.

4. Future Directions of This Research.

We are convinced that the foregoing ideas can form the core of the architecture of a very powerful, flexible and extensible software environment. Further we

believe that these ideas bring a great deal of unity and focus to the efforts of software process researchers, software environment researchers and architects, and programming language researchers. We have already embarked upon a broad and vigorous program of research aimed at exploring the ramifications of these ideas, sharpening and refining them, and experimentally evaluating them. In this section we briefly present some of the key issues that we see as the prominent ones in this research.

We suggest that diverse areas of software engineering research can be significantly advanced by establishing a language in which software engineering processes can be effectively expressed, by expressing various key processes in that language, by creating compilation and runtime support systems for that language, and by then attempting to use these systems to provide effective automated support for the execution of these key software processes. Finally, we suggest that all of this is best pursued by creating and experimenting with an actual software environment predicated upon these ideas. Work on such a prototype environment is currently being begun. This environment is called Arcadia [Arcadia 86].

A central focus to this research is to to devise and study various algorithms for expressing software engineering processes. We have already begun to create algorithms for software development, software product maintenance, software process maintenance, software product evaluation and software process evaluation. As we have proceeded with the iterative refinement of these algorithms we have begun to learn much about the nature of these processes, their relations to each other and to other processes such as reuse. Thus we expect that continuation of this activity will lead to more important insights, and, eventually to sound process algorithms.

Our intention is to arrive at acceptable algorithms for describing development, maintenance, evaluation and testing, and reuse. Having devised these algorithms we propose to solidify our notions of superior language paradigms and constructs for expressing them, and to solidify our notions of tool and environment architecture by examining the runtime structures and procedures needed to effectively support execution of these algorithms.

4.1. Design of the Software Engineering Language Itself.

While exploratory development of process algorithms seems more central to the pursuit of this research, development of a language system in which such

algorithms might be encoded also strikes us as important. As this area is somewhat better established, it seems, moreover, easier to categorize and organize the way in which this research might be pursued.

In this section we attempt to indicate how an orderly attack on the problem of creating a software engineering language might be organized.

4.1.1. The Language Paradigm.

The first task in defining this language will be to decide which linguistic paradigm is most suitable. Our early investigations have built upon a bias towards the use of an algorithmic language. This bias is at least largely based upon our belief that software engineers will be drawn from the ranks of software practitioners, and are most likely to be trained in programming in algorithmic, sequential (possibly parallel) programming languages. Thus it is most attractive to suggest that, as software engineers, they program in a language with a similar, and therefore comfortable, philosophy and paradigm. Making such an important decision based solely on the grounds of tradition and convenience is imprudent, however. Thus we have attempted to seek deeper justification for this prediliction.

Our second basis for believing that a parallel, sequential algorithmic language is best suited for programming software engineering is that software development and maintenance are to be carried out by human practitioners, and humans seem to us to be psychologically most self assured in thinking about their plans and actions in terms of discrete, sequential steps and activities. Thus, a programming language which will be use to describe, regulate and control their activities in building software should seem most natural and comfortable if it expresses their activities in similarly discrete and sequential steps.

Although our current predisposition is towards more traditional and comfortable algorithmic languages, we recognize the need to consider non-traditional languages as well. We have, accordingly, pondered a variety of non-algorithmic language paradigms. One interesting paradigm, for example, is that of a process control language. As previously described, software development and maintenance might well be viewed as a real-time processes involving the synchronization of the activities of people and mechanized aids. Borrowing from the idiom of process control software might thus be highly effective. From this perspective, activities such as code and design creation would correspond to synthesis (input) processes, and consistency checking activities would correspond to analysis processes. The software development program would then be a software system which incorporated a variety of such synthesis and analysis processes as asynchronous tasks, which, nevertheless communicated broadly

among themselves, and paused at programmed intervals and events to synchronize and evaluate progress and consistency. Thus, control process software should be examined carefully as a model which might be worth emulating. Simulation languages capable of supporting the programming of such processes (eg. Simula and Simscript) are also worth examining.

Other potentially useful paradigms include object-oriented languages, functional or applicative languages and database language approaches. The appeal of an object oriented language approach is that it would clearly support and encourage the view of a software product as a collection of objects of diverse types. It seems clear that the designer of a language for software engineering should borrow heavily from object oriented language mechanisms for defining object types and operations on such types. In fact this strategy has been adopted by some KBSE researchers. Their work centers on the creation of a large and intricate knowledge structure which captures and correctly interrelates all information pertaining to the software being developed or maintained, and using that knowledge effectively in support of these processes. This is important to us, because, as observed earlier, our SPS can very reasonably be viewed as a knowledge structure, although it seems that the SPS and SP's we envision would probably be structures of much larger objects than are envisioned as the constituents of KBSE's.

We are still not persuaded that we should adopt an object-oriented language paradigm for our work, however, because of our relatively greater emphasis on the algorithms needed to develop and maintain the knowledge and information structures which we agree are central. We remain convinced that software engineers and practitioners do maintain a strongly algorithmic view of what they do, but that they have been thwarted in effectively exploiting it due to the lack of an adequate expressive device. Object oriented languages do not seem to us to sufficiently encourage the attention to algorithmic expression which seems urgently needed at this stage of exploration of the algorithmic nature of software engineering processes.

The appeal of a functional language is that it could support and encourage the view that software processes (eg. development and maintenance) are essentially the evaluation of large functions which are computed by the evaluation of a complex substructure of smaller functions. This view is appealing in that the software product which is the focus of these software processes is a complex composite of smaller objects and interrelations. Thus, it seems quite useful to describe processes on this product as the processes of creating its subcomponents and evaluating needed interrelations.

There appear to be significant problems in adopting a functional programming language approach, however. One is the need to at least partially linearize the

order of evaluation of functions and subfunctions. The problem here is that some of the functions are to be carried out by humans who have difficulty carrying out unbounded parallel activities, and because the other functions must be carried out on a bounded number of computing devices. Functional programming systems assume the responsibility for such linearization, and take this process out of the hands of the software engineer. This strikes us as being awkward, at least for the present, when efficient compilation of efficient object code for large and complex functional programs is very much a research topic. Further, we continue to believe that some aspects of at least some software development processes are more straightforwardly describable in terms of sequential algorithmic steps rather than composition and nesting of functions. Testing and consistency determination would seem to be in this category. Thus, perhaps it is most reasonable to design the software engineering language in such a way as to combine both procedural and functional programming capabilities in such a way as to exploit the strengths of each in supporting software engineering process description and control.

Finally, it seems certain that at least some of the notational and descriptive devices used in database languages are useful as means for describing the software product in a software engineering language. Thus we would expect to borrow at least some of the descriptive mechanisms of such languages. We are not as sanguine about the prospect of exploiting such languages as vehicles for expressing software processes. We are particularly skeptical about how well such languages and associated support systems would be able to support software process maintenance. Earlier we observed that this sort of maintenance entails alteration of the software product structure (the database schema) while retaining and reassimilating most, if not all, of its contents. We believe that extensible algorithmic language compilation systems currently provide the most useful paradigms for how to approach this problem.

Whatever the language paradigms used or merged to form the basis for the software engineering language, there will have to be important further research in establishing an effective semantic base within that language for support of software activities. Thus another important aspect of this research will be the determination of the built-in primitive data types and operators which the language should provide, as well as the appropriate linguistic and conceptual treatment of the relations which bind software objects together to form software products.

4.1.2. A Compilation System for the Software Engineering Language.

As a primary reason for creating a software engineering language is to use it to coordinate the work of software tools and their integration into a cogent

software environment, a prerequisite for such a language is that it be supported by an effective compiler for the language. This compiler will have to accept specifications of such objects as types, TDG's, and ODG's and transform them into object stores which will then be able to organize and structure software objects as they are created. The compiler will also have to accept development and maintenance process definitions and transform them into procedures which coordinate the work of human software workers with each other and with the activities of software tools which implement various automatically supported software operations.

Compilation issues can be divided broadly into three types--syntactic issues, semantic issues and code generation issues. The last two types of issues seem to be the most interesting.

4.1.2.1. Semantic Issues.

It is difficult to guess at which issues will pose the most difficulty in carrying out semantic analysis of the software engineering language, especially in view of the fact that not even the language paradigm has been selected. On the other hand, it does seem that a powerful and extensible type structure is essential, and this indicates that the semantic phase will have to be capable of potentially sophisticated type checking. In addition, the need for supporting extensions and alterations to the type structure of the language, while facilitating the large-scale retention of objects created under the previous type schema, indicate the need for a compilation system in which the type structure can be modified as an object and used to create a new semantic phase for the compiler. In addition, the compilation system must be capable of analyzing the differences between the old and new type structures to enable maximal reuse and retention of objects.

This last observation suggests a more precise interpretation of our earlier suggestion that the software environment which we are proposing might be useful in maintaining itself. We suggest that it is useful to consider the compiler for the software engineering language must be considered to be an object. Any single executable instance of the software engineering language compiler should be considered to be an operator in a PDP. The source text of the compiler, however, should be considered to be an object in a PMP, thereby enabling the PMP to alter the semantics of the language. Thus the semantics-alteration process we have described is a specific type of PMP.

4.1.2.2. Software Engineering Language Optimization Issues.

The task of the optimization phase of software engineering language compilation systems is to emit sequences of instructions to carry out and synchronize either human operations or computer based activities in such a way as to effect the algorithmic processes described by the coder. Language semantic definitions should assure that there is no doubt about how language operations are to be interpreted in terms of manual processes and mechanized tools. Further, control flow and synchronization operations will also require semantic definition in order to enable emission of effective object code.

Generation of efficient code is a more interesting problem. Two efficiency issues suggest themselves--one is the efficient storage of software objects and the second is effective reuse of intermediate software objects. The problem of achieving efficient storage of software objects is an important one, which seems to fall more in the province of runtime support systems, and will be discussed shortly. The problem of achieving effective reuse of intermediate objects is a central issue in compilation and also affects the philosophy of tool implementation. Reuse of intermediate objects is only possible if the operations specified by the user can be seen as being composed of lower level operations which create such intermediate objects. Thus, if all of the operations in the software engineering language are implemented as monoliths, there would seem to be correspondingly little opportunity for reuse of intermediate objects. On the other hand, if operations are generally implemented as concatenations or structures of lower level operators, which produce intermediate software objects, these then become ideal candidates for reuse in subsequent computations.

Thus, the desirability of optimizing the object code generated by the software engineering language compiler provides strong impetus for the implementation of functional tools as composites of smaller, lower level tools (called tool fragments in [Osterweil 83]). The strategy governing the way in which intermediate objects are selected for storage for potential reuse was a research issue in the context of [Osterweil 83]. Language statements were processed essentially interactively and there were no alternatives to statistical approaches to guide the strategy for saving intermediate objects. In this proposal, with our suggestion that software development and maintenance processes be captured in compilable code, it becomes clear that optimization algorithms and strategies much like those used in classical languages can and should be applied.

4.1.3. Runtime Support for the Software Engineering Language Processor.

It is clear that the software engineering language will require powerful runtime support subsystems in order to be the basis for effective execution of software

engineering processes. Two key areas of support immediately suggest themselves for early consideration-- object management and input/output.

The need for a powerful object manager has been amply indicated by earlier sections of this paper. One of the central concepts in the approach we are suggesting here is that software development be thought of in terms of the need for creating, organizing and managing software objects. Clearly it is imperative to have effective ways in which to store them. The problems in doing so are badly compounded by our contention that objects are tightly interconnected to each other by such types of relations as hierarchy, derivation, and consistency. Clearly such simple organizational strategies as tree structures are woefully inadequate. We believe that relational database approaches have serious drawbacks as well. Some of these have been indicated earlier, and center on the dynamism of the structure of the object store.

We view the Odin system [ClemOst 86] as a prototype object management capability which incorporates a number of desirable features. From the perspective of this paper, we now understand that Odin actually incorporates some features of a software engineering language subset, a semantic analyzer modification and maintenance system subset, and an object manager. It seems clear that whatever object managment system is incorporated into the proposed software engineering language, it will have to have strong ties to the semantic analysis phase of the language compiler.

Input/output capabilities for the language are also quite interesting to ponder. Here we are inclined to view all processes which are carried out by humans as being input processes, and therefore in need of language assistance. Such assistance would have to range from simple text I/O support, through interactive editor support, to support for interaction with graphical and pictorial images. It seems essential that all of these interactions be implemented and supported in terms of basic language I/O primitives to assure a reasonably uniform user view of the software processing capabilities offered. Thus, whether the user were creating source code, design elements, test data sets or functional specifications, there would be a strong sense of uniformity of interaction with the software engineering language's features.

Output would have to offer a similarly uniform feel. The purpose of output capabilities would be to enable the user to see objects and relations in the emerging software product. Thus, we expect that it will be important for the user to be able to view a variety of objects, perhaps from a variety of perspectives, and to interact with those objects. This suggests that the I/O package will have to incorporate such capabilities as windowing, and menus. The use of color might well also prove to be of value.

Finally, it should be noted that the software engineer is also likely to need to view the process-objects which are being created and to get some insight into the processes which have been constructed. This interaction is different from the interaction needed by software practitioners. It corresponds more to the needs of a debugger of a program than to the needs of a user of that program. Thus, it is expected that the runtime system will also have to incorporate tools and capabilities for enabling the software engineer to study the structure and contents of the software process-object itself, in addition to the structure and contents of its individual component objects and relations. Here too, we are struck by the fact that these needs do not differ signifcantly from the needs of the software practitioners. This again suggests that the software engineering language may be adequate for the development and maintenance of programs for the development and maintenance of software process-objects.

4.2. The Arcadia Protype Environment Project.

As indicated above, many of these ideas are to be experimentally evaluated through a collaborative effort to build a protoype environment implementing many of these ideas. This prototype environment is to be called Arcadia. The Arcadia project involves researchers from the University of California at Irvine, the University of Massachusetts at Amherst, the University of Colorado at Boulder, TRW, Inc., Aerospace Corporation, and Incremental Systems Corp. More details of the Arcadia approach and directions can be found in [Arcadia 86].

5. Acknowledgments.

The ideas described here have been developed over a period of a few years. The author has profited considerably from many useful conversations with a number of colleagues. Numerous conversations with John Buxton, Dick Taylor, Bob Balzer, Lori Clarke and Geoff Clemm have been particularly useful in shaping these ideas.

In addition, the author wishes to express his gratitude to the National Science Foundation, and the US Department of Energy for their support of this work through grants numbered, DCR-8403341 and DE-FG02-84ER13283 respectively.

6. REFERENCES.

[Arcadia 86] R.Taylor, et.al., "Arcadia: A Software Development Environment Research Project," Univ. of Calif., Irvine, Dept. of Info. and Comp. Sci, Tech. Rpt., April 1986.

[Clemm 86] G.M. Clemm, "The Odin System: An Object Manager for Extensible Software Environments," Univ. of Colo. Dept. of Comp. Sci. Ph.D. Thesis Boulder, CO (1986).

[ClemmOst 86] G.M. Clemm and L.J. Osterweil, "The Odin Environment Integration Mechanism," Univ. of Colo. Dept. of Comp. Sci. Tech. Rpt. #CU-CS-323-86 (May 1986).

[FosdOst 76] L.D. Fosdick and L.J. Osterweil, "Data Flow Analysis in Software Reliability," ACM Computing Surveys, 8 pp. 305-330 (Sept. 1976).

[Goldberg 84] A. Goldberg, "Smalltalk-80: The Interactive Programming Environment," Addison-Wesley, Reading, Mass, 1984.

[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.

[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.

[Osterweil 82] L.J. Osterweil, "A Strategy for Integrating Program Program Testing and Analysis," in Program Testing, (Chandrasekaran and Radicchi, eds.) North Holland, pp. 187-229, (1982).

[Osterweil 83] L.J. Osterweil, "Toolpack--An Experimental Software Development Environment Research Project," IEEE Trans. on Software Eng., SE-9,

pp. 673-685 (November 1983).

[Osterweil 86] L.J. Osterweil, "Software Process Interpretation and Software Environments," Univ. of Colo. Dept. of Comp. Sci. Tech Rpt. #CU-CS-324 (May 1986).

[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.

[TeitMas 81] W.Teitelman and L.Masinter, "The Interlisp Programming Environment," Computer, 14 pp. 25-33 (April 1981).

[Tichy 83] W. Tichy, "Design, implementation and evaluation of a revision control system," Proc. 6th Int. Conf. on Software Engineering, Tokyo, (Sept. 1982) pp.58-67.