Next Generation Software Environments: Principles, Problems, and Research Directions

Richard N. Taylor, Deborah A. Baker‡, Frank C. Belz†,
Barry W. Boehm†, Lori A. Clarke*, David A. Fisher‡,
Leon Osterweil**, Richard W. Selby, Jack C. Wileden*,
Alexander L. Wolf*, Michal Young

Department of Information and Computer Science
University of California, Irvine[1]

*Department of Computer and Information Science
University of Massachusetts, Amherst[2]

**Department of Computer Science
University of Colorado, Boulder[3]

†TRW
Redondo Beach, California

‡Incremental Systems Corporation
Pittsburgh, Pennsylvania[4]

July 15, 1987

[1]This work was supported in part by National Science Foundation grants DCR-8451421, DCR-8502558, and DCR-8521398; and Hughes Aircraft.

[2]This work was supported in part by the following grants: Rome Air Development Center, F30602-86-C0006; National Science Foundation grants DCR-8404217 and DCR-8408143; and Control Data Corporation, No.84M103.

[3]Work supported by National Science Foundation grants MCS-8302644 and DCR-8403341, the Department of Energy grant #1537612, and The American Telephone and Telegraph Company.

[4]This work was supported in part by the Department of Defense, Advanced Research Projects Agency, Order No. 5057, monitored by the Department of the Navy, Space and Naval Warfare Systems Command, under contract N00039-85-C-0126. Approved for Public Release, Distribution Unlimited.

Document Citation Numbers

University of California, Irvine
Department of Information and Computer Science
Technical Report Number 87-16

University of Colorado, Boulder
Department of Computer Science
Technical Report Number CU–CS–370–87

University of Massachusetts, Amherst
Department of Computer and Information Science
Technical Report Number 87-63

Incremental Systems Corporation
Technical Report Number 87–7–1

Arcadia Document Number
UCI-87-10

Contents

The past decade has seen a burgeoning of research and development in software environments. Conferences have been devoted to the topic of practical environments, journal papers produced, and commercial systems sold. Given all the activity, one might expect a great deal of consensus on issues, approaches, and techniques. This is not the case, however. Indeed, the term "environment" is still used in a variety of conflicting ways. Nevertheless substantial progress has been made and we are at least nearing consensus on many critical issues.

The purpose of this paper is to characterize environments, describe several important principles that have emerged in the last decade or so, note current open problems, and describe some approaches to these problems, with particular emphasis on the activities of one large-scale research program, the Arcadia project. Consideration is also given to two related topics: empirical evaluation and technology transition. That is, how can environments and their constituents be evaluated, and how can new developments be moved effectively into the production sector?

1 A Characterization of Software Environments

A common thread that runs through the literature on software environments is that the purpose of environments is to *support the user* in some software development or maintenance activity. Sometimes this activity is highly constrained and well defined, such as constructing syntactically correct source code. Other environments have broader scope, but are highly restrictive in the order of events that are permitted. Still other environments are simply collections of tools and data management facilities that are believed to be helpful in a broad arena of software evolution activities.

Thoughtful consideration of this notion of "supporting the user" yields some important insights. First, if the activities that an environment supports are not precisely and unambiguously described, it is difficult for potential users to assess whether their needs will be met. Second, the facilities provided by an environment may be so loosely structured that they *could* support a variety of activities, but if all structuring and composition is solely the end user's responsibility, for which no automated support is provided, then undue burden is placed upon the user. It is likely that such an environment will be used to support only the simplest and smallest activities. Third, clearly specifying and supporting a specific activity may not be enough. Change to virtually any software development or maintenance activity is inevitable. Users will wish to incorporate new tools and development methodologies. If the environment's structure is closely entwined with the original activity, then accommodating change may be difficult and costly, or not possible at all.

In our estimation, therefore, a useful environment will

- support clearly and precisely defined activities,

- mechanize the structuring and composition of support functions, and
- accommodate changes and personal preferences.

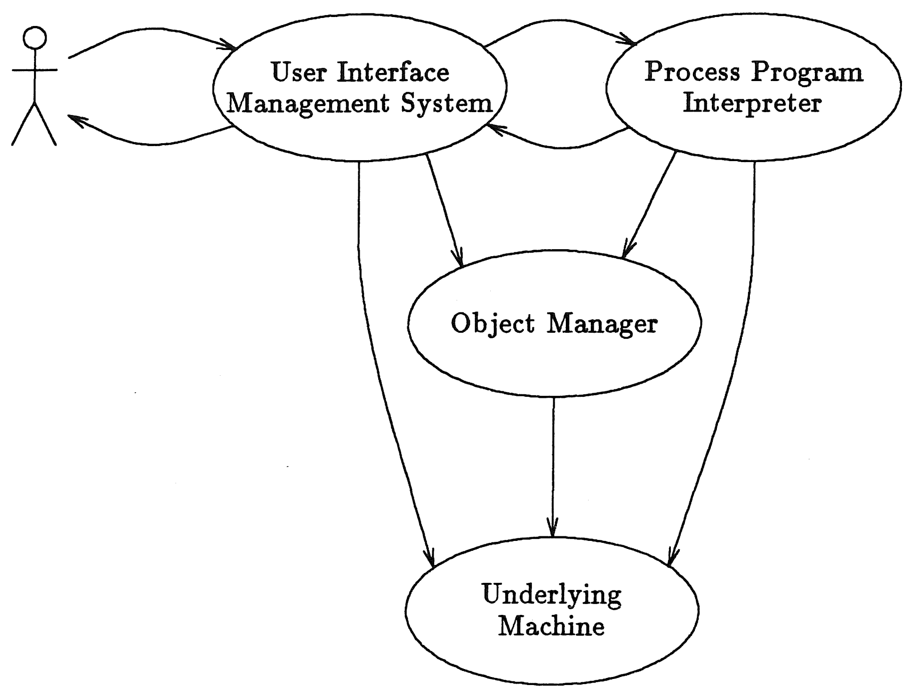
Such notions of usefulness could be applied to any software system, however, and environments must be distinguished from large, multi-option tools if the term is to have any useful meaning. Significantly we believe that this distinction cannot be made for some early so-called programming environments. We therefore offer the following definitions of next generation software environments.

An environment consists of two parts: a fixed part and a variant part. The aspects of an environment that are prone to change, such as the tool set, are encapsulated in the variant part. The unchanging mechanisms necessary to ensure the integrity, extensibility, flexibility, and integration of the environment are in the fixed part. We believe this division is a critical separation of concerns. In contrast, environments like *Interlisp* [TM81] and *Smalltalk* [GR83] make no distinction between fixed and variant part, or even between the environment and the software developed within the environment (to the point where the software can run nowhere except within the environment — they are inextricably bound).

More specifically, the variant part consists of the evolving set of data objects (such as specifications, programs, and test data) along with rigorous, detailed descriptions of software development or maintenance activities, which we term "process programs" [Ost86] [Ost87]. These activities are defined in terms of individual specific operators, which correspond to the classical notion of tools. These operators can themselves be modeled as process programs (if not actually implemented as such), and are correspondingly in the variant part too.

The fixed part, which can also be termed the *environment infrastructure*, consists of all the mechanisms necessary for the automated interpretation of process programs. Specifically, it consists of a language for writing process programs, an agent that enables interpretation of process programs, including mechanisms for managing persistent objects, and facilities for providing interfaces to the human user. The fixed part also encapsulates assumptions regarding its underlying machine. That is, it may make use of an operating system, a storage manager, and so forth. These assumptions are also components of the infrastructure. The components of the infrastructure that are active entities are shown in Figure 1.

The user may initiate action by making a request for support of some activity. The request is passed along, through the user interface management system, to the process program interpreter. Assuming that the request is well formed, the interpreter initiates a series of actions, executing the process specified by the user. This execution will involve invoking operators upon operands — both of which are managed by the "object manager" component. Examples of operators include lexers, parsers, code generators, debuggers, test data generators, and specification and design language processors. Examples of operands include source code, executable



$\longrightarrow \equiv$ "makes a request upon"

Figure 1: The active entities of an environment infrastructure with the "makes a request upon" relationship shown between them. The variant parts of an environment are not shown.

modules, test data, specifications, designs, management data, symbol tables, various internal representations of programs, designs, system generation directives (e.g., make files), and intermediate analysis results.

Of course many of the operators will themselves be processes that cause additional actions to occur. Note that the entities belonging to the variant part of the environment, such as tools and data objects, are *managed* by this part of the infrastructure — they are called into action at appropriate times and do their work — but they are not themselves components of the infrastructure.

There are, of course, operations in a software process that can only be performed by people. In essence, the mundane aspects of processes are automated in this view of environments, while the creative aspects are performed by creative agents — people. Accordingly the interpreter may issue a request for an operation to be performed by the user, passing the request through the user interface management system.

The user interface management system may directly request services of the object manager for storage of windows and depictions of objects. Finally, the underlying machine provides services for each of the other three automated components of the infrastructure.

In short, the infrastructure is a virtual machine for the interpretation of process programs.

Clearly we are burying many of the critical and interesting technical issues inside the process interpretation system and the object manager. The subsequent sections of this paper, which are organized around the entities shown in Figure 1, will clarify and elucidate the ideas suggested here. The notions of software processes, process programming languages, and the interpretation of process programs are considered first, in Section 2, as they are central to our view of environments. These are followed by discussions of object management in Section 3, the user interface management system in Section 4, the underlying machine in Section 5, and then the two related topics of empirical evaluation and technology transition in Sections 6 and 7.

2 Software Process Definition and Interpretation

2.1 Software Processes

Perhaps the most striking feature of the environment architecture described here is that it empowers users to rigorously specify their software products and product types *and* rigorously and explicitly specify alterable process programs to guide in the development and maintenance of these products. Previous environment architectures have exploited only primitive notions of explicitly specified products and processes. They have supported relatively fixed processes and products, often specified only implicitly. Moreover, the user's freedom to specify the process supported or the type of product produced by the environment was generally sharply restricted.

For instance, some previous environments have been aimed at supporting the development and maintenance of software specifications and designs. Systems such as *PAISLey* [ZS86], *RSL/REVS* [BBD77], *SARA* [EFRV86], Data Flow Diagram Designs [War86], Jackson System Development [Cam86], and *USE* [WPSK86], are examples of such previous environments. While certainly valuable for their intended purposes, each of these systems provides support for the creation of a relatively narrow range of software objects by relatively restrictive and inflexible processes. Specifically, they guide users to the development of design or specification objects in a particular fixed discipline and format, which is usually pictorial or graphical. For example, *RSL/REVS* is organized to strongly aid users in creating, analyzing, and maintaining designs as hierarchies of graph structures that are heavily annotated. In such environments, the exact structure of the objects and their pictorial representations vary from one system to another. In some cases the user is able to tailor and adapt these software object types. Invariably, however, these adaptations can be made only within a narrow range. For example, users of such environments may be able to select the specific fields to be incorporated into a design node, but only from a given fixed list of fields and types.

In addition, these design and specification support environments attempt to lead the user through specific procedural processes that are intended to expedite the creation of designs and improve the chance that the resulting designs are well formed and in compliance with the guidelines of the design or specification methodology being supported. Accordingly, such environments are often either indifferent or overtly hostile towards attempts to create design objects of new or different types, or to follow development procedures that have been devised by the user. From our perspective, these environments contain "hard coded" object specifications and processes (which they effectively support). They are not hospitable to user attempts to make significant alterations to such processes or design object specifications.

There are also other environments whose goals are aimed more at supporting the development of code. Environments such as *Interlisp* [TM81], *Arcturus* [ST84], and *Cedar* [Tei85] integrate facilities to support the creation of code in specific languages. They support such common activities as editing, parsing, debugging, and documentation. These environments assume that user activities can be uniformly and smoothly integrated by viewing them as examination and transformation of a single uniform representation of one product — code or a representation of the code. In Interlisp all software products, as well as the procedures and tools used to create them, are considered to be instances of lists. In Arcturus, software products and the commands used to manipulate them are all instances of Ada code. As long as the user's activities are effectively modeled in these ways, these environments provide strong support. As the user seeks to model software products and processes as objects of different types, support from these environments falters and becomes awkward.

Similarly, intelligent editors such as the *Cornell Program Synthesizer* [TR81], *Integral-C* [Ros86], *Gandalf* [HN86], and *Mentor* [DHKL84] are all effective in integrating user activities, but only over a restricted range. Here, the integration rationale is that user activities all revolve around a parsed representation of code in a specific language. Experience has shown that this representation supports many user activities more effectively than a textual representation of the code. Shifting focus from text to an internal representation, however, does not solve the problems posed by restricting users from being able to create and manipulate software objects of types of their own creation using explicit processes of their own creation. Structure editors implicitly assume that users are concerned with a few types of objects.

In all of these cases the effectiveness of the support tools is drastically reduced when the process that the user wishes to carry out is not anticipated by the environment. In the case of code synthesis using environments such as *Mentor* or *Interlisp*, when the user attempts to execute process steps operating on object types not related to code, such as tests, support is weak. In the case of a design environment, when the user attempts to stray beyond the supported methodology, or attempts to carry out such processes as coding or testing, support similarly is weak. The objective here is not to criticize specific systems, but to point out that the value of any *environment* is closely related to its abilities to support all the user's activities.

Support for only limited, pre-determined processes is particularly disturbing in view of the observation that there is currently little consensus about what constitutes adequate software products and effective software processes, and that products and processes must therefore be expected to vary from user to user, from location to location, and from time to time. The most effective process architecture for a spread-sheet application, for example, will be different from the most effective process architecture for developing a complex command-control-communications system. Similarly, project schedule, budget, personnel, reliability, or portability constraints will strongly condition the most effective choice and sequencing of major process activities. Just as the programming of a software product is more effective when preceded by product requirements analysis, architecture definition, and design activities, so will be the programming of one's software lifecycle process.

Many observers believe that progress towards understanding what constitutes adequate products and effective processes can only follow from experimentation with alternatives. We believe that the best way to facilitate such experimentation is to enable users to easily yet rigorously describe software products and processes in ways that are convenient and effective and to support the rapid interpretation of processes in terms of software tools and procedures. From our perspective it is clear that this is tantamount to the creation of environments in which the variant part — i.e., the product specifications, process descriptions and set of operators —

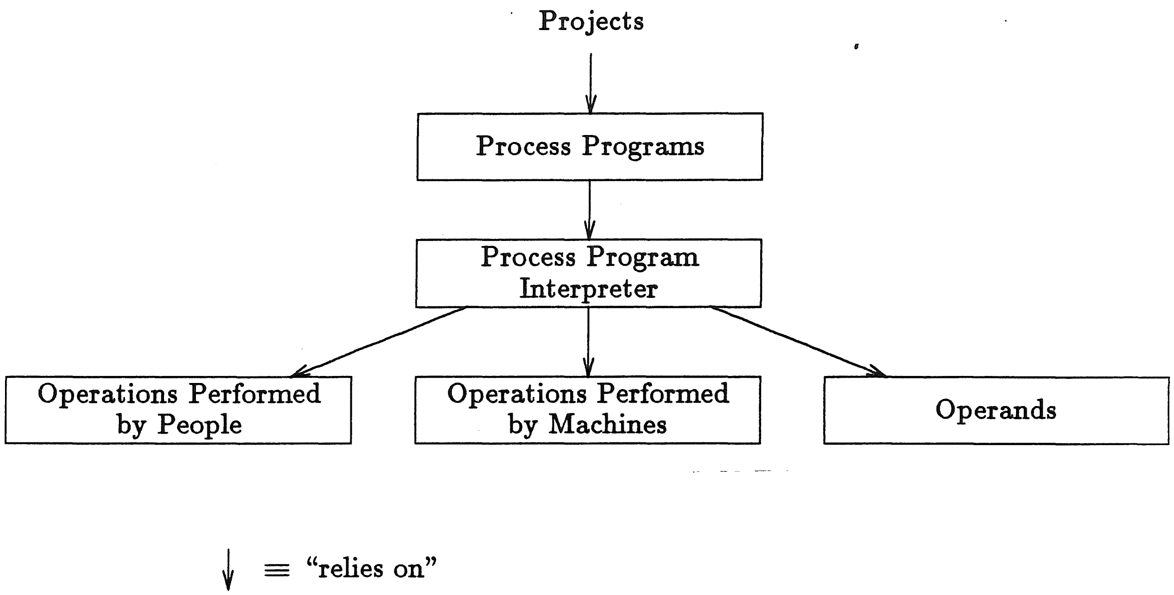


Figure 2: The relationship of process programs to the projects they support and to the mechanisms that implement them.

is specifiable by the user, and in which the environment exploits this specification to fashion user support, utilizing components from the fixed part.

2.2 Process Programming Languages

In this section we discuss some key details of how process programs should be used to enable users to flexibly specify how they wish to have machine resources applied to the support of their activities. Figure 2 shows the relationship of process programs to the projects they support and the operations and operands that implement them. We see that projects are to be directly supported by specifications of how they are to be carried out, where these specifications are to be captured by process programs. Process programs rely upon a process program interpreter to carry out their commands. This interpreter is responsible for translating the specifications embodied in the process program into operations on operands. As indicated in Figure 2, some of the operators upon which the interpreter relies are executed by computing devices, but others are executed by humans. An important aspect of process programming is that it is a vehicle for indicating which activities are to be carried out by humans and how these activities are to be coordinated with activities carried out by computing devices. We believe that one of the most important objectives of software engineering is to orchestrate the way in which humans, support software systems, and machines are to be coordinated to isolate and specify problems, to attack their solution and to determine the degree to which these activities have been successful or need to be modified. These activities are neither completely mechanical and automatable, nor completely spontaneous and indefinable. Rather they must be a careful blend of these approaches. We believe that this blend can best be specified and communicated by expressing it in a concrete form — namely the process program.

Early Precedents and Lessons All software projects have as their goal the creation and/or modification of software products. They work towards this goal by carrying out software operations on software operands. Thus, at the most basic level, process programs must be viewed as vehicles for specifying the coordination of such operations. It is worthwhile to observe that even operating system control languages can be viewed as primitive process programming languages. Language processors and system facilities are legitimate operators and the files managed by the operating system's file system are the operands. The user issues commands to the operating system and it effects the requested operations. Thus, command files or scripts are primitive process programs, using the operating system command language as the process programming language.

It is important to observe that these primitive process programs are used to indicate the ways in which human operations are to be coordinated with software operations. For example, users employ operators incorporated into tools, such as browsers, to help them carry out (human) selection operations. Selected objects are then used as operands to subsequent software operators, such as edit and compile. As another example, users often carry out standard sequences of operations at certain fixed times during software projects. They may invoke scripts to automatically compile new code, or automatically check consistency of new code with support libraries. These scripts orchestrate the interaction between machine operations (e.g., compiling and checking) and human operations (e.g., creating code). In this way, the scripts are small but good examples of process programs.

Scripts have also been used to automatically create new objects and maintain certain types of consistency between new and existing objects. For example, scripts are used to automatically recompile source code when support libraries have been changed, or to recreate executables when source code has been changed. The *Make* system in *Unix* [Fel79] is an example of a capability whose goal is to facilitate the creation of powerful scripts of just this sort, through the use of a terse and precise notation. Clearly this notation is a process programming language, albeit a primitive one.

Current Issues and Needs Operating system command languages and *Make* can be used to write process programs, but they lack the power needed to effectively program large and complex processes. One of their most basic and significant deficiencies is their lack of facilities for defining software objects as instances of types. Our ideas about the need for an object typing facility, and the way in which it should be provided, are elaborated more fully in the next section. Suffice it to say here that typing offers a powerful vehicle for organizing not just the basic objects to be managed in a software project, but also for defining and organizing the operators — both human and machine executable — to be employed by the project[1].

Going further, we believe the next basic capability that a process programming language must support is specification of the order in which operators are to be applied to operands. Many operating system command languages incorporate some flow of control operators, but these are generally quite primitive, often consisting only of basic looping and alternation constructs. In fact it seems that paucity of control flow expressive power may well be the weakest aspect of most operating system command languages.

Interestingly, other early attempts at rigorously expressing software process have focused directly on these aspects. Most notable among these attempts have been efforts to use diagrammatic representations to depict the major features of large-scale software processes. In this work, principal software processes were represented by boxes, and flow of control relations among them were represented by arrows. The "Waterfall Model" of software development relied upon this device in an early attempt to represent an overall software development process [Roy70]. Almost immediately, software process modelers attempted to use these pictorial representations to also show other relations such as data flow or process hierarchy. Even later work attempted to simultaneously show diagrammatic representations of many key relations among a variety of types of software objects. In order to do this, data objects were differentiated from process objects by making distinctions between the shapes of the boxes representing them. Distinctions among relations were made by defining different shapes of arrowheads and different colors and shapes of arrows to represent these different relations. Some examples of such more advanced diagrammatic process representations are ETVX boxes [RRJC85], SADT diagrams [RJ77] and Software Development Graphs [Bjo87].

Inevitably these efforts are limited by the fact that there are arbitrarily many valid relations among the large number of software objects required to adequately describe software processes, and that different users may at different times wish to study various combinations of them. Creating one single diagram containing all of these relations is hardly a solution, as such a diagram is so complicated as to confound all understanding. Creation of a single internal representation capturing all of the complex relations in a software process, and then relying upon tools to draw specific diagrams ("views") upon request, seems to be a plausible solution to

[1]In either case, the semantics of operations can be formally defined using, for example, pre- and post- conditions. These conditions, in turn, utilize the accessing primitives that participate in defining the object types.

this dilemma. We believe that a process program, written in a suitable language, is the appropriate device for representing this.

Thus, we see that software practitioners have used operating system command languages as primitive programming languages to program micro-level processes. These languages are too primitive to support effective expression of large and complex software processes. At the same time, modelers have attempted to portray these large and complex processes with diagram systems that have been unable to clearly and *precisely* express all of the facets and details of true software processes. They have been thwarted in their attempts for the same reason — namely the lack of a language that is suitably expressive. Thus two diverse and important currents both point towards the same basic problem — the need to define a process programming language.

Characteristics of a Process Programming Language We have already noted that software process programming requires the ability to define a wide variety of software object types, and that this is best supported by powerful data typing and relationship mechanisms. (The issues here are addressed in depth in Section 3.) Moreover, support for controlling the procedural flow inherent in software processes must also be provided. Our early attempts to construct process programs for realistic software processes have convinced us that the range of control flow operators required is quite broad. Clearly iteration, alternation, selection, and procedure invocation are required in order to accurately portray the way in which software processes are carried out. In addition such control flow capabilities as parallel execution and exception handling seem essential.

Further consideration of how to enable specification of flow of control raises the deeper issue of whether an imperative model is the most appropriate linguistic paradigm to use in process programming. Although many aspects of many kinds of software process seem to be inherently procedural and algorithmic, there are other software activities that defy simple algorithmic description and suggest that the declarative paradigm is much more appropriate. Design creation is an example of such an activity.

In design creation the goal is to create a design specification. Often (e.g., in the case of the Software Cost Reduction methodology [PC86]), it is quite possible to specify the goal object — namely a complex structure of carefully prescribed design elements — but it is not clear how to give complete procedural details on how to construct it. In such cases it is often reasonable to create rules that guide and constrain activities, such as the selection of good candidates for design elaboration, or that can intelligently raise issues about apparent inconsistencies among design elements. Thus some aspects of design seem to be rule-based. Other aspects, such as the orderly elaboration of details of design elements and their correlation with each other, are much more procedural. This suggests that a process programming language might ideally be a language that combines procedural and rule-base paradigms.

Furthermore, our early work indicates that an important aspect of software processes is that they often create other processes that are executed later on. For example, test planning is a process whose goal is the creation of another process that is to be executed at some future point in the execution of the software development process. During test planning, the test plan is created as a software object. This may entail such subactivities as development of test cases, encoding of algorithmic strategies for the systematic execution of the test cases, and development of procedures for capturing test results. Much later in the development process, after code has been developed, this test plan object must be "executed." This entails treating the test plan object as a process, rather than an operand. This passive/active nature of some software processes points to the desirability of a language such as Lisp in which code and data are freely interchangeable.

In the Arcadia project we are experimenting with software process programming languages. In our earliest efforts we are coding process modules in a variety of language paradigms, attempting to arrive at a more precise set of requirements for this language.

3 Object Management

An environment user's primary objective is to create and/or maintain a *software product*. No matter what process program might be used in creating and maintaining it, a software product will be a very complex and highly interrelated collection of objects. Those objects will be of widely different kinds, ranging from source code and executable modules to documentation and test plans. Each kind of object will have an associated set of applicable operations, but operations applicable to one kind of object will generally not be appropriate for use with other kinds. This suggests that an environment's fixed part should provide support for managing typed objects and a rich set of relationships among them.

As Figure 1 indicates, the object management system will be a major component of the Arcadia environment infrastructure. It will be responsible for managing objects in two distinct classes: the *components* of the software products being produced by users of the environment, and the *tools and information structures* that constitute the environment itself. From the process programming perspective, the former can be viewed as the (input and output) data manipulated by a process program while the latter are the operators and internal data structures of the process program. (As previously noted, an object can move back and forth between categories during its lifetime.)

The object management system will provide the underlying mechanism on which the data management capabilities of a process programming language and a process program interpreter can be constructed. A particular process programming language might present its users exactly the same object management capabilities that the environment's object management system provides, as an assembly language presents its users exactly the same data types provided by the underlying machine. It seems equally likely, however, that a process programming language might offer a different view of objects than that provided by the environment's object manager. In either case, the properties of the object management system will influence the data management aspects of an environment's process programming languages.

Most environment builders have had to rely on a traditional file or database system for storing the objects associated with their environment. It is our belief, however, that a much richer set of capabilities for controlling object creation, access and organization is essential to an environment. In particular, a suitably powerful object management system will enhance the environment's support for change, its integration, its support for software reuse and its support for cooperative work by multiple developers.

Work on environments during the last decade has revealed four important areas of concern that must be addressed by an object management system. These are:

- type systems;
- relationship sytems;
- object persistence; and
- concurrent and distributed object management.

Each poses interesting problems. The capabilities sought in each of these areas and the problems we foresee are discussed below.

Type systems As indicated above, we view a type system as the primary mechanism for describing and maintaining objects. The object manager should be able to enforce the type system, hiding the internal structure of typed objects behind well-defined interfaces and strictly controlling the operations that can be performed on those objects. If all objects are instances of abstract data types, it is easier to share objects or to change their implementations. Thus, basing the object management system on a typing system that fully supports data abstraction will contribute to environment flexibility and software reuse.

Current approaches to object management in environments fall far short of providing full support for typed objects. Typically the components of a product are treated simply as files [Fel79] and tools are viewed as operators applicable to the contents of those files. Usually in such systems, only a predetermined and limited number of different kinds of components (e.g., source file, object file) and operations (e.g., compiler, linker) are available. The *Odin* subsystem of *Toolpack* [CO86] improved on this simple view by using file name extensions as a weak form of typing mechanism for files. It also allowed users to define which tools could operate on or produce files of various types. The *System Modeller*, developed as part of the *Cedar* system [LS83a] used the term "object" for referring to the files containing product components, but did not treat the objects as instances of abstract types. The Common APSE Interface Set (CAIS) [CAI85] defines a system model with three kinds of nodes—file, structural, and process—but does not treat those nodes as typed objects. While clearly improving on the simple use of files, all of these systems provide only partial support for typed objects. Meanwhile, work on support for typed objects within the traditional database community [SR86,CDF*86,ZW86], while encouraging, is still in its primitive stages and far from providing the flexibility and power needed for object management in a software development environment [Ber87]. Recent work on rich type systems, particularly in the context of object-oriented languages [Mey86], is also encouraging, but also still in its infancy. No consensus has yet emerged on a desirable and appropriate set of features for such a type system.

Thus, one major object management research issue is: What kind of type system is needed to describe the objects populating a software development environment? The type system needs to be flexible and powerful enough to capture the relevant properties of environment objects. Tools, processes, and perhaps even types themselves need to be treated as typed objects. Once the capabilities of the type system are clearly delineated, suitable mechanisms for realizing those capabilities must be found. While there are many intriguing proposals for type mechanisms, it is not clear which of these (e.g., single vs. multiple inheritance, specification vs. representation inheritance, generics, static vs. dynamic binding) form a compatible set providing the capabilities needed for environments.

Relationship systems Closely related to the ability to precisely define and maintain the typed objects in the environment is the ability to capture and maintain the relationships among those objects. Much environment work in the last ten years has focused on mechanisms for describing, reasoning about, or exploiting relationships among objects. Examples of relationships include those connecting various versions of a module, or those between the modules constituting a configuration, or those between a module and all the others that it calls, or those joining activities in a work breakdown structure. Examples of tools that reason about or exploit relationships among objects include revision control systems [Tic82], automated system building tools [Fel79], call graph analyzers, and work activity management systems [GLB*83].

Explicitly indicating the relationships among an environment's tools and information structures should make it easier to modify the environment since the effect of changes can be determined. Moreover, capabilities that rely on relationships, such as inference and derivation, will enhance environment integration by allowing users to interact with the environment at a high level, leaving the intermediate steps to be automatically determined. Generic relationship capabilities will also enhance integration by providing a uniform set of capabilities across different kinds of relationships.

A weakness in previous work is that there has been no systematic treatment of the numerous and complex relationships that exist among environment objects. The CAIS notions of primary and secondary relationships (also found in the node structure of the *ALS* [Tha82]), *Odin*'s derivation graphs, and the system models of *Cedar* represent important starting points. The concept of configuration threads found in *DSEE* [LRPC84] and the relationship capabilities for module interconnection languages provided by *INTERCOL* [Tic79] are additional examples of partial treatments specialized to one class of relationships.

Thus, another important object management research question is: What are suitable primitives and constructors for defining the relationships needed in environments? It is not clear whether the diverse relationships needed in software development can be captured in a single model or not. Moreover, how should the relationship structure and the type system interact? Associated with the relationship system is a set of capabilities, such as consistency checking, derivation tracking, and inferencing. Work needs to be done on identifying these capabilities and in exploring how generic such capabilities can be. For example, can generic consistency checking tools applicable to the relationship structure subsume the specialized consistency analyses associated with interface control or configuration management? Another important concern is when and how such capabilities are initiated. Some must be requested by the environment user, either directly or via an executing process. Others can be more effective if triggered by resulting events. Thus, support for "active" objects or daemons that are triggered by process or user-specified events in the environment is needed.

Object persistence The object manager must be able to preserve the components of software products and the constituents of software environments for arbitrary periods of time. Moreover, it should preserve both the structure and the restrictions on how these objects can be manipulated that are imposed by the type system. Under such a scheme, the traditional distinction between primary and secondary storage representations of objects is hidden within the typed object abstraction. This can free both environment users and environment builders from concern about distinctions between internal and external representations of objects and conversions between those representations. Thus, the object manager should support *persistence*, enabling objects to continue to exist beyond the lifetime of any of the tools that manipulate them and preserving the integrity of their types and relationships to other objects.

Current approaches to persistence, based on files or databases, require explicit action by the tools. Using a file system, a tool must take responsibility for converting the internal form of an object to an acceptable (e.g., linear) external form and, when needed, converting it back. There are few restrictions to assure that the type of an object is not violated (e.g., that its contents are not altered using an editor while it resides in the file) or changed (e.g., that a stack is not read back as an array). Using a database system, the tool must make calls on the database to explicitly store and retrieve information. Current databases provide support for only a limited number of types, so once again the tool must provide the conversion algorithms and there is no guarantee of type integrity. There has been some interesting work on merging database support into programming languages [ABC*83,CLF86,OSD86], although implemented prototypes have been very restrictive about the supported types [ABC*83] or the underlying program model [CLF86].

Thus, an important issue that must be addressed by the object manager is: How should persistence be provided for arbitrarily complex, typed objects? To permit maximum flexibility in the creation of objects and their relationships, the persistence of an object should be a property orthogonal to all other object properties. It is not clear how persistence should be recognized in a program (e.g., declared as part of the type or explicitly requested with the instantiation of an object) or how invisible persistence can be (e.g., no need to explicitly "commit" or "linearize" objects). Supporting a rich type system and providing an invisible line between memory and secondary storage raise challenging problems.

Concurrent and distributed object management To allow multiple users to work conveniently on the same software development project requires support for concurrent and distributed object management. Assuming a network of workstations, different members of a development project may simultaneously be invoking the same or different tools to operate on one or more of the same objects. Thus, the object manager must be able to mediate concurrent usage of objects and to maintain consistency of both the objects and their relationships. Ideally, the object manager should make the distributed nature of the object base and the concurrent access to its objects invisible to users and tools in the environment.

A variety of approaches for handling distribution and concurrency have emerged from programming language [ALR83] [Hoa78] [LS83b] and file system and database research [HM85] [WPE*83] [SHN*85]. Unfortunately no single model for dealing with these issues is universally accepted within one of these domains, let alone for objects that move between them. Moreover, some of the more popular approaches are ill-suited for use in an environment object management system. Locking schemes, for example, typically apply to entire objects and do not permit concurrent access to disjoint subsets of an object's components, which may be a frequent occurrence in an environment. Transactions schemes generally presuppose relatively short duration transactions, while a software developer's transactions (e.g., update a source program) may last for days or weeks. The rollback approach to conflict resolution is also of questionable value in an environment.

Thus, one of the major problems facing object management is: What are appropriate constructs for expressing distribution and concurrency constraints and what underlying mechanisms must be provided to support these constraints? It is not clear what storage management primitives need to be provided to adequately capture the distribution and concurrency needs of an environment. As with types, relationships, and support for persistence, the appropriate descriptive notations must be developed as well. Also, where should the desired concurrent/distributed behavior be described — in the tools that create the actual instances of the objects, in the abstract data types that define the objects, or in the process programs that describe how the objects are to co-exist within the environment?

Arcadia Approaches As indicated above, much work has previously been done on problems related to object management. That work, however, has generally been directed toward solving individual problems, leading in some cases to incompatible solutions, and has not yet resulted in consensus on the appropriateness of those solutions. Moreover, much of the work has been oriented toward domains other than software development.

The approach to developing an object management system that is being taken in the Arcadia project is therefore one of synthesis and extension. In particular, we are initially looking to programming language technology for guidance in the design of a type system and the expression of distribution and concurrency constraints, and initially looking to database technology for guidance in the design of mechanisms for persistence, relationships, change, and distribution. It is clear that some new solutions are still required to satisfy the special needs of software development environments. To sharpen our understanding of these needs, we are examining process programs for a wide variety of activities, examining a wide variety of tools that would make use of the object management system, and reflecting on our experience building *Odin* and *Keystone* [CHOW85], which can be viewed as primitive object management systems. We are also developing formal models, as we have previously done for module interface relationships [WCW86], for describing and evaluating the various capabilities intended for inclusion into the object management system.

One design that we are considering provides a functional layering of the desired capabilities. At the lowest level are facilities for such things as storage management, concurrency control, and transaction management. The next level supports the basic concept of types, essentially defining the type system provided by the object manager. Above that are primitives for realizing object relationships. The capabilities for revision control, partitioning of objects into libraries, and the like, appear at the highest level. All of this together provides the basis on which type systems for process programming languages can be implemented.

We intend to build successively more sophisticated prototypes of the object management system. This activity will be facilitated by the recent trend in database research toward the development of *database toolkits* [SZR86,Ber87,Spe87,CDF*86]. These toolkits provide basic, low-level capabilities such as storage management, concurrency control, and transaction management. The idea is to provide a foundation upon which to build higher-level capabilities, such as those for typing and relationships. The obvious benefit of using such a foundation is the ability to experiment with alternative higher-level structures without having to construct instances of the lower-level facilities for each such alternative. These toolkits are intended to be "general purpose" and we intend to experiment with prototypes of the toolkits to ensure that our particular needs can be satisfied.

Until database toolkits become available, we are building prototypes that examine particular aspects of object management. Three significant examples of this are a relationship management system, an application generator called *Graphite* [CWW86], and a storage system for our internal representation of programs, Iris (see Section 7). The relationship management system is intended as a vehicle for exploring the suitability of various automated constraint-satisfaction and inferencing techniques in the domain of process programming. In particular, it provides a general framework for specifying goals in terms of relationships over objects, and mechanisms, such as backward and forward chaining, for reasoning about the satisfaction of those goals. *Graphite* is being used to investigate issues in the specification of types, insulating tools from changes to those types, and hiding details of how instances of those types are made persistent. The class of types that *Graphite* focuses on is attributed graphs, since it is clear that this particular class is important in a software development environment. For instance, one of our uses for attributed graphs is to internally represent programs. The third example is also concerned with attributed graphs since the purpose of the storage system prototype is to investigate issues in the persistence of such graphs. In particular, we are using the storage system prototype to study techniques for achieving efficient access to subsets of attributes by exploiting locality of reference.

4 Interface with the Human User

The user interface management system is the third major component of an environment's fixed part. We consider it here, discussing first the characteristics of good interfaces that an environment should exhibit. Some outstanding problems are then noted. The remainder of the section addresses various specific approaches to the design of user interface management systems, including separating tool functionality from interaction properties, the use of abstract depictions in managing the display, and techniques to aid in achieving uniformity.

Characteristics of good interfaces Broad consensus exists on the qualities which distingish good user interfaces for software environments. *Uniformity* (or *consistency*) reduces the difficulty of learning new activities and moving between activities. The *direct manipulation* interaction paradigm, using graphics and pointing devices, increases the communication bandwidth between tool and user. *Permissive* (or *non-preemptive*) interfaces allow the user to interleave activities in a natural way.

Uniformity reduces the number of details a human user must remember, and increases skill transfer between activities. A uniform interface makes the same set of operations available everywhere they make sense, and allows the user to specify an operation in the same manner wherever it is available. Interpreter-based programming environments made significant early progress toward uniformity by unifying the command language and programming language of the environment. More recently, editor-based programming environments have provided a uniform set of commands for manipulating program source code, blurring the distinction between editing, compiling, and debugging. Limited progress has been made in providing a uniform interface across a wider variety of activities, mostly by imposing informal standards (like the *Macintosh* user interface guidelines [Ins85]) and providing a toolkit of reusable components (scrollbars, menus, and the like).

Direct manipulation, or more precisely the illusion of directly manipulating a set of objects, requires a rich visual representation of state. This visual representation unburdens the users' short-term memory, replacing recall tasks with easier recognition tasks. (Menus serve a similar purpose with respect to remembering commands). Objects are referred to with a pointing device and through implicit pointing (e.g., cursor position.) Changes in the representation provide immediate confirmation of user actions. The basic principles of direct manipulation are applicable to character displays, but modern bitmapped workstations are capable of richer visual representations of state. Pioneering work in the application of graphics to programming and software engineering include the *Incense* debugging system [Mye83], the *Balsa* algorithm animation system [BS84], and the *Pecan* programming environment [Rei85].

Permissiveness is an essential aspect of direct manipulation, too seldom achieved in current systems. A permissive interface allows the user to choose the next action, arbitrarily interleaving interactions with each object depicted on the screen. The converse of permissiveness is *preemption*. A preemptive interface imposes an order on user actions. The prompt/input paradigm of gathering input is a classic example of preemption.

Window systems are primarily a means of limiting preemption. Windows grafted onto a conventional system in the form of multiple virtual terminals provide a minimal degree of permissiveness, sufficient for the user to temporarily escape from the control of a single application. The multiple views of *Pecan* [Rei85] and the *Pi* debugger [Car86] hint at the richer interaction possible when each tool may coordinate several threads of control.

Outstanding problems Uniformity becomes both more important and harder to maintain as the scope of an environment grows. A large, extensible environment will contain tools contributed by a diverse community of developers and users. Both the toolset and interaction techniques can be expected to evolve during the lifetime of the environment. A critical problem, then, is decoupling the human interface from tools so that each may evolve independently. Providing a set of reusable components is helpful, but may not be enough by itself. The *SunView* facilities [Sun86], for instance, encourage similar visual appearance across tools, but they are not much help in establishing consistent interpretations of mouse and keyboard actions within windows managed by tools. The interface between interaction and tool functionality (in the application domain) is the most troublesome interface in modularizing interactive graphics programs. Because graphics toolkits deal entirely with the graphical domain, they do not help clean up this interface.

The problem becomes apparent when one notes that other tools, as well as human users, may use a tool component. A good human interface is generally not a good tool interface. An all-purpose interface, like Unix character streams, is unlikely to be satisfactory in either role. Thus, in current Unix-based systems, the set of tool-usable tools is quite disjoint from the set of interactive tools. It is difficult, for instance, to use a screen-oriented editor or a spreadsheet program as part of a pipe or shell script.

Techniques and approaches User interface management systems (UIMS) is an active area of research, outside the context of software environments research as well as within it. The following paragraphs discuss current approaches to separating application functionality from interaction facilities, managing the display, and establishing a uniform interface to all the functions supported by an environment. The design of the *Chiron* user interface subsystem of *Arcadia* is briefly presented as an example of a system that brings together several threads from current user interface research.

Separating functionality and interaction. Several current approaches to direct manipulation interfaces carefully separate the application domain (or *model*) from the presentation domain (or *view*). A tool manipulates objects in an application domain. An encapsulated tool component (sometimes called the *controller*) maintains consistency between objects in the two domains, so that the presentation domain accurately reflects the state of application objects and the application domain properly responds to user activity in the presentation domain. We see this as a key separation.

In "editor" environments supporting a narrow set of objects and functions, a tool component may map the central application data structure (typically a parse tree) to the presentation domain. Separation of concerns between application domain and presentation domain is achieved, but at the cost of requiring all environment facilities to operate on the shared data structure rather than a variety of data structures suited to different applications. Environments of wide scope require a more flexible scheme.

The *Incense* debugger [Mye83] introduced the notion of *artists* to maintain the depictions of each particular type of application data. Each artist encapsulates information about a particular application data type, and how it should be represented. Since this information is encapsulated in individual artists, outside of any shared user interface infrastructure, tools are not forced to share a common representation or data model for application objects. This is important, but raises the question of associating artists with types.

Multiple inheritance in a type system provides a powerful mechanism for associating artists with application data types. The *annotation* mechanism of *Loops* [SBK86] [SB86] can be used to trigger an artist whenever an application object is manipulated. Lisp object systems [BKK*86,Moo86,BDG*87] provide a similar capability with method-mixing in multiple inheritance. Conceptually, an artist is "wrapped around" an existing data type, as illustrated in Figure 3, so that the old interface (available operations) shows through the new.

Managing the display. Current approaches to user interfaces generally interpose an intermediate level of representation between application objects and their concrete depiction on the screen (Figure 4). This *abstract depiction* serves several purposes. It is generally more convenient for an artist to manipulate a structured description of a display than a lower-level representation, especially if the structure of the abstract depiction reflects the structure of the depicted object. Also, manipulating a portion of the abstract depiction can result in efficient incremental update of the display, provided the rendering agent is able to determine which portions of the concrete depiction may be affected by the change.

More importantly, an abstract depiction can be used as a basis for input correlation, relating an input action (e.g., mouse click) with a particular application object. Window systems provide input correlation down to the window level, but not within windows. This is sufficient for menus and scrollbars, which can be designed so that each choice lies in its own window, but not sufficient for general diagrammatic depictions of software objects. An abstract depiction level managed by the environment can perform input correlation to the level of individual picture

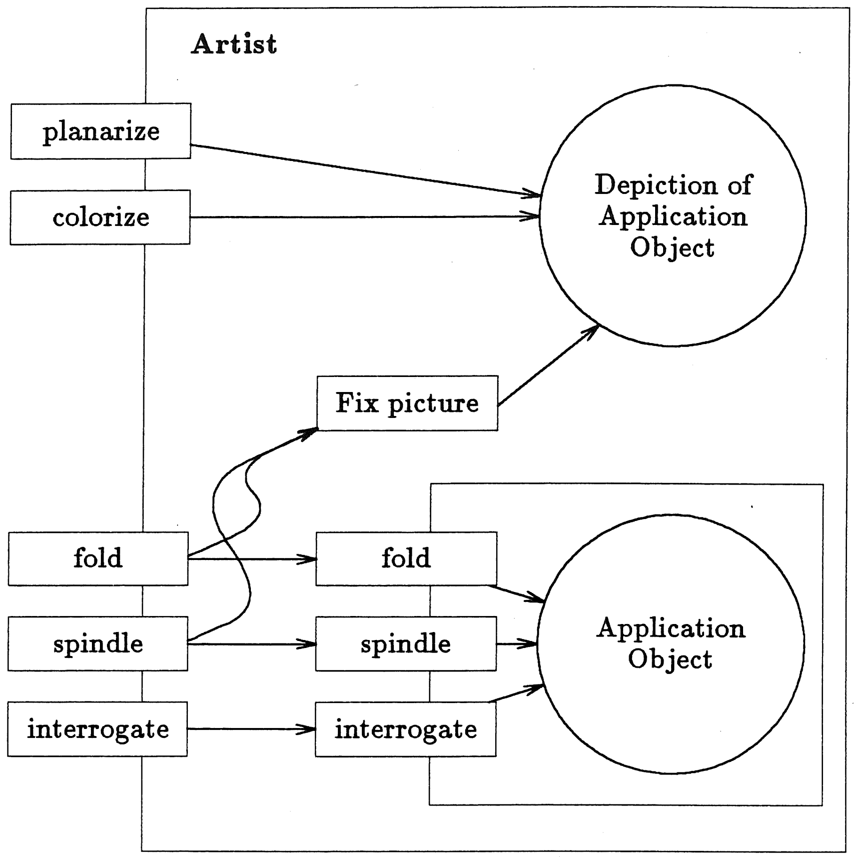


Figure 3: An artist is logically "wrapped around" an abstract data type. The application (or *model*) object is encapsulated in an abstract data type, with visible operations `fold`, `spindle`, and `interrogate`. Each of these operations "show through" the artist, in the sense that the artist exports operations with identical signatures and semantics, except that the presentation object (or *view*) is updated as a side effect. Operations which do not change the application object (`interrogate`, in this diagram) are simply re-exported without change; additional operations on the presentation object (`planarize`, `colorize`) may be added.

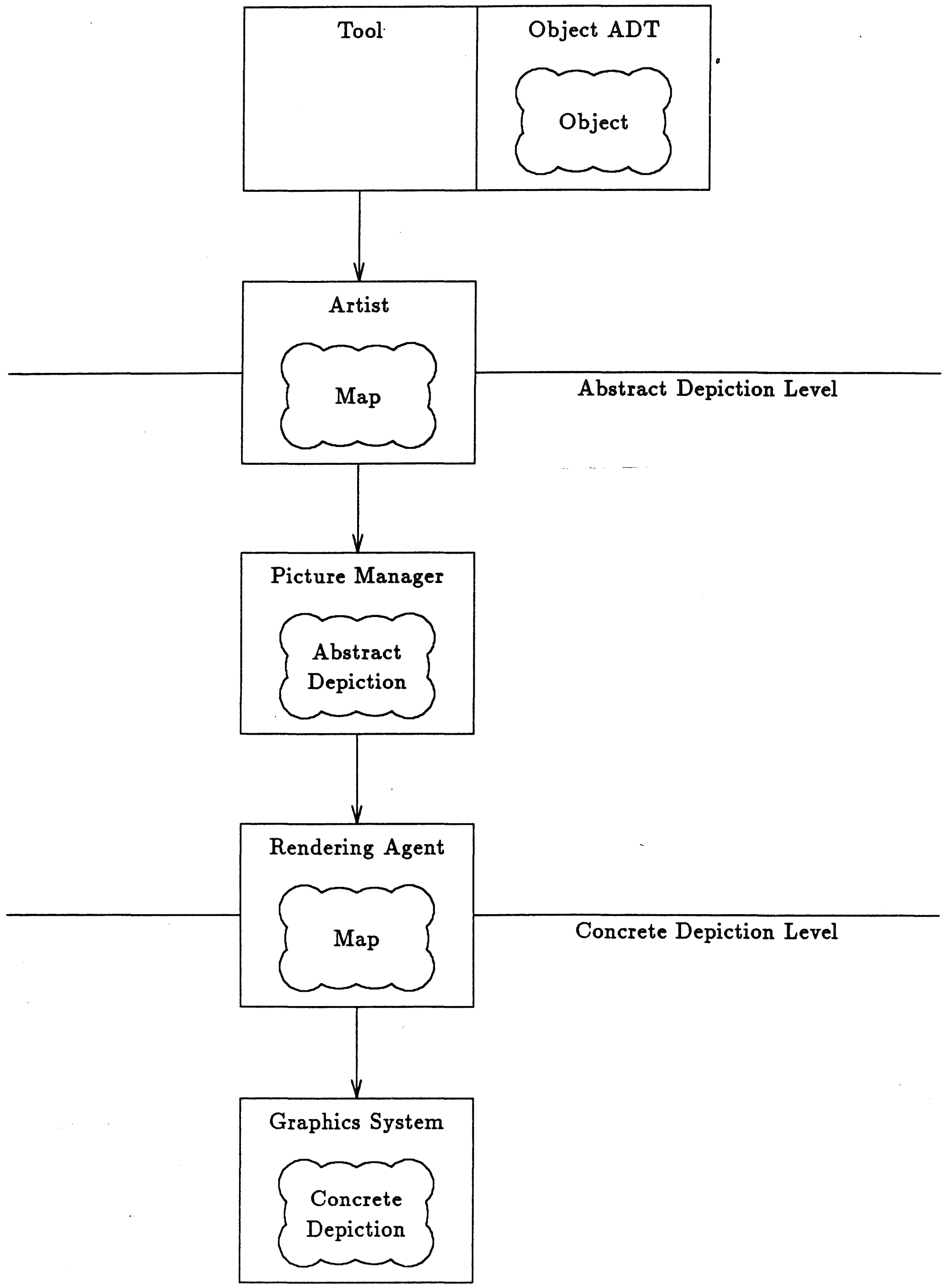


Figure 4: The most difficult part of interactive graphics programming is maintaining the association between application objects and their depictions. When a structured intermediate representation (an *abstract depiction*) is interposed between application objects and their depictions, this task can be considerably simplified. The system can maintain the relationship between the abstract and concrete depictions, and associate input events with particular components of the abstract depiction. The artist that created that particular component can then be notified; it need only maintain the relationship between high-level graphical objects and the application objects they depict.

elements.

Approaches to uniformity. Centralized interpretation of low-level input can be used to achieve a basic level of uniformity. For instance, if the lexeme *select* is bound to a single click of the leftmost mouse button, then the application will receive the event *select,* rather than a raw key click, when the button is pressed. Binding of lexemes to raw events should always be under control of the user, rather than the tool builder. Techniques adequate for administering this level of interpretation are well known (e.g., the TIP tables of *Cedar*). Central administration can also guarantee consistent interpretation of a small set of "global" commands, for instance, terminating a tool. Anyone who has attempted to kill an unfamiliar Unix program with keyboard incantations will appreciate the importance of such guarantees.

Reusable components are a complementary approach to promoting uniformity. Application-independent components, such as scrollbars, are already in common use. Clean encapsulation of interaction facilities makes it feasible to provide reusable components for data abstractions in a particular application domain (e.g., Petri nets), as well. Since artists are associated with abstract data types, the path of least resistance for tool developers is to reuse an artist for all interactive tools dealing with a particular data abstraction.

Arcadia approach to user interface. The *Chiron* user interface subsystem of *Arcadia* is characterized by artists bound to abstract data types through a type inheritance mechanism, a simple diagram-oriented abstract depiction, concurrency between and within tools, and support for uniformity across tools.

Since abstract data types are key to modularizing tool fragments in *Arcadia*, *Chiron* uses type inheritance to bind artists to objects. An artist inherits application functionality and adds new state (a depiction) and new operations (e.g., `planarize`, `colorize`). It also manages side effects to the new state from existing operations (i.e., updating the display when the object changes). *Chiron* provides a diagram-oriented $2\frac{1}{2}$D hierarchical display model, including nested and overlapping windows. Artists manipulate this ***abstract depiction.*** *Chiron* maps it into the *concrete depiction,* typically a bitmap.

Chiron emphasizes concurrency between and within tools. Each depicted object may have its thread of control, and each may independently maintain its depiction and react to user actions. In addition, a rendering agent maintains the concrete depiction concurrently with manipulations of the abstract depiction (subject to interlocks on the latter), and input proceeds concurrently with output. Additional detail on *Chiron* can be found in [YTT87].

5 Capabilities of the Underlying Machine

All aspects of environments described above must ultimately rest on some set of underlying machines. Whether an environment can successfully and readily be implemented atop a particular set of machines depends on how closely the needs of the former can be matched with the capabilities of the latter. One issue, for example, is the mapping between the environment's notion of execution (of tools, for example) and the model of execution supported by the underlying machine. Another similar example is the mapping between the environment's object management mechanism and the storage structures (both primary and secondary) of the underlying machine. The mapping of the user interface management system to an underlying window system interface was briefly discussed in the preceding section. Here we limit our attention to supporting parallelism within an environment and supporting a distributed environment.

In our estimation, the underlying machine must provide good features for exploiting and controlling parallelism. Early operating systems, such as *Unix*, provided this capability via the notion of operating system process. An important objective of their definition is to protect users from one another — providing firewalls. While firewalls are certainly necessary, always binding protection to the concept of parallelism prevents effective sharing of data, tool integration, and exploitation of true hardware parallelism. It imposes a sequential view of tools and programming on developers.

We see a critical need for operating systems and underlying hardware to provide multiple threads of control within a single virtual address space. This capability, sometimes called "lightweight processes" [SZBH86], allows truly concurrent tools to operate on shared objects. This can be exploited effectively in the interface to the human user and in many tool designs. It is also necessary for good data collection in support of the evaluation of environments, a topic discussed in the following section. This is because data collection, regarding the performance of a tool or development process, can occur silently and unobtrusively, not degrading the performance of the activity being monitored. Operating systems should also provide asynchronous I/O primitives, to avoid restricting parallelism within a tool.

Turning now to the topic of distributed environments, we believe that future environments must be designed to support multi-person teams of developers utilizing a local-area network. Moreover, comprehensive, large, industrial-quality environments should not make any assumptions regarding the physical proximity of the developers. In particular, wide-area persistent object management facilities must be provided, enabling cross-country sharing and co-development of objects.

One of the most promising developments in operating systems, which offers the potential for providing many of the capabilities just described, is the *Mach* operating system being developed at CMU [Ras86]. The Arcadia Project is currently evaluating *Mach* for use as its underlying operating system.

6 Measurement and Evaluation of Environments

Software environments are intended to reduce the cost of software development and maintenance and to increase the quality of the resulting software. It is not enough to just propose software environments or build prototypes, however. The effectiveness of software environments needs to be measured and evaluated. Environments incorporate a diverse set of components, such as user interface facilities and analysis tools, support numerous kinds of objects, and apply to a wide range of problem domains. Hence the evaluation criteria, or software metrics, need to be tailored to cost and quality indicators for the particular environment components, objects, and application areas. Moreover, software environments need to be evaluated in a multiplicity of situations: developers with different expertise levels, different software error profiles, and so forth.

Though the need for measurement and evaluation of software environments is apparent, there are several pertinent open problems. One problem relates to how software environments should be evaluated: approaches have ranged from "single observation" studies [WHBK86] to more systematic approaches [Sel85]. Another problem is that there has been no unifying system to support the processes of specifying, collecting, and analyzing software metrics. Yet another problem is whether evaluation mechanisms should be incorporated into the software environment architecture (such as is being done in *Arcadia*) or supported in a stand-alone system (e.g., *TAME* [BR87]).

Approach to measurement and evaluation Software environments contain a range of components embodying innovative technologies. We believe the impact of these technologies should be assessed by conducting a series of controlled, empirical studies. The intent of such studies is to characterize the usefulness of particular tools and to evaluate the effectiveness of a software environment as a whole. When properly carried out, the series of studies can capture several factors that may affect an environment and its components, such as expertise of programmers using the environment. The studies should focus on evaluation criteria that are customized to the purpose of an individual environment. Evaluation criteria may include the degree to which an environment supports rapid change of large software systems, extensibility of the tool set, alternate lifecycle models, programming-in-the-large, reuse of previous work-products, object persistency, customization of the user interface, and transparency from the underlying operating system and machine architecture. Depending on the evaluation criteria, a series of empirical studies may incorporate in-depth, small group studies, and/or large scale experiments. In order for the results of the studies to be representative of large populations of potential

- An environment should support metrics that address a multiplicity of software quality, cost, and productivity factors.
- An environment should support metrics that enable the measurement and evaluation of a wide variety of software objects and software processes.
- An environment should support metrics that capture information related to the usage of human resources.
- An environment should support metrics that capture information related to the changes and errors in software objects and processes.
- An environment should support metrics that capture information from both the static and dynamic analyses of software objects and processes.
- An environment should support metrics that apply to the multiple levels of inter-human communication and organization.
- An environment should support metrics that apply to the multiple levels of organization of software tools and methodologies.
- An environment should support the data validation of collected metrics.
- An environment should support the collection of both individual metrics and characteristic metric sets.
- An environment should enable the definition of new metrics in terms of algebraic combinations of metrics currently collected.
- An environment should support the rapid analysis of and feedback from collected metrics.
- An environment should support a natural interface between itself and statistical and graphical packages.

Table 1: Sample guidelines for incorporating metrics into software environments.

environment users, the studies need to use subjects with a wide variation of expertise, ranging from novices through highly experienced professionals. The use of sensitive statistical techniques, such as within-subjects, fractional factorial designs, takes into account both large variations in human performance, such as the 10:1 differential noted in [Cur83], and interactions among the factors being compared.

Drawing from earlier work in evaluating software technologies, we have developed several guidelines pertaining to the purposes, types, and scopes of metrics that are desirable [Sel87]. Some of the guidelines are shown in Table 1. The guidelines may be viewed as a first step toward articulating the measurement capabilities needed by software researchers and developers. They are intended to structure the process of integrating measurement into environments, delineate the measurement issues that affect environment builders, and be reusable across multiple environment projects.

Application to Arcadia The approach in the Arcadia project is for a series of empirical experiments to characterize the environment's overall effectiveness. The studies will employ several evaluation criteria, such as those listed above, and numerous scenarios regarding software creation and manipulation processes, software objects, and software developers. In the controlled studies, we intend to apply our experience gained from conducting empirical evaluations in related areas [Sel86] [SBB87]. These studies will provide both feedback to the developers during production of the environment and valuable information for Arcadia users.

The Arcadia environment architecture will support the specification, automated collection, and automated analysis of software metrics, in recognition of the need mentioned above. We are applying the guidelines of Table 1. We have investigated various approaches for determining which metrics to collect and have focused on the concept of characteristic metric sets. A characteristic metric set is defined as a concise collection of metrics that capture distinct cost and quality factors [BS85]. The desirable properties of metrics include their being objective, automatable, and transparently calculable.

We believe the most effective way to achieve these properties is to build the metric collection mechanisms into the environment's infrastructure. Our approach is to define a characteristic set of metrics for the environment as a whole and a characteristic set for each software object type. (Recall that all software objects in the environment are instances of types, including those objects that house software process descriptions). The software metrics in the set are customized to meet the individual cost, quality, and productivity indicators of a particular object type. The metrics are viewed as accessing primitives to the types and may be inherited from other types. Daemons with programmable firing criteria are the vehicles for calculating the metrics; they aid in achieving the transparent calculation of the metrics. The mechanism here is very similar to the annotation concept described in Figure 3.

7 Development and Tech Transfer

Several essential principles of technology transfer have emerged in the past few years. Among the most important are that (a) the introduction of new technology tends to raise uncertainty in the organizations that depend upon it, (b) the degree of uncertainty is generally proportional to the extent to which the new technology affects the members of the organizational structure, and (c) difficulty in achieving effective technology transfer is proportional to the degree of unresolved uncertainty.

Software support environments affect most or all the members of projects, so to the extent that an approach to such environments is revolutionary, putting that approach into widespread use represents a challenge. Acceptance of a dramatically new approach to environment technology can be enhanced by resolving or mitigating the uncertainties; there are several ways to do this.

First among these is to guarantee that the environment really provides appropriate service to the project members throughout their period of dependence on it. To achieve appropriate service mandates some general properties of quality software, such as robustness, adaptability, user-friendliness, and adequate support. In addition, it requires focussing on the high priority needs of the projects, such as achieving software development schedule compression and matching the user context. This in turn requires that the developers of the new environment technology obtain practical feedback on its effectiveness before the projects adopt it. One important technique to achieve early feedback is to build the environment in stages and to use the early versions of the environment in the process of building later stages. This is a special case of iterative development that many environment developers find extremely valuable. It must be augmented by techniques to achieve feedback from a more representative sample of potential users. This may include, for example, prototyping and incremental development in which such representative users exercise the early increments. In some cases, users' participation may be extended to include contributing to the specification and design of the environment capabilities.

Acceptance of new technology requires more than a quality product, however. It also requires the perception of quality. There are many ways to establish that perception. A first step includes persuasion through such techniques as effective demonstrations, reasoned arguments, and macro- and micro-economic analyses supported by empirical measurements. These relatively indirect means must eventually give way to the direct means of experience. Testimony from satisfied users is a most powerful way to strengthen an emerging perception of quality. Ultimately, convincing persuasion can only be achieved by the personal experience of using the environment directly.

The accurate perception of quality is still not sufficient; certain "entry barriers" to acceptance must be minimized. For example, some new technologies, although they provide long term benefits or support difficult tasks very well, may make short term or simple tasks more complex and expensive. This can be a fatal entry barrier. So can comparatively high initial costs or very limited availability. Broad acceptance generally requires simple mechanisms for simple tasks, low entry cost, and broad availability when compared to alternate available technology.

Finally, the new technology may not be accepted even where there is an accurate perception of quality and the entry barriers have been minimized. Frequently there is need for a *champion* within the organization to guide the acceptance of the new technology through the maze of organizational roadblocks.

The principles above suggest a number of ways in which technology transfer can fail to take place.

An environment may simply not be good enough; it may fail to support the most important needs of its targeted user community or it may be maladapted to the context in which that community must work. It may improve the support of some users, but degrade the support for others. Unless the environment is a commercial product, it is unusual that provision for feedback in the design and development phase extends to representative users. In fact, in some cases, even the environment developers fail to use the environment in their own development efforts. The environment may not be sufficiently robust, user-friendly, or well supported.

The benefits of an effective new technology may never be accepted. In many cases, new environment technology is not pushed beyond a minimal stage of visibility and awareness. In particular, very few empirical studies have been conducted to evaluate software environments and to characterize their usefulness in a variety of problem domains. The failure to provide convincing economic arguments based on such data has certainly doomed many attempts to involve particular user communities with environments beyond the demonstration stage. Thus, effective environments may not be acknowledged as such in many organizations.

Approach to Development and Technology Transfer in Arcadia The approach to environment development and technology transfer within the Arcadia project spans several issues. The Arcadia consortium is developing running versions of its environment, dubbed *Arcadia-N*, with N being the version number. Once the *Arcadia-1* prototype version is available, we will develop future environment versions using *Arcadia-1*. This accomplishes several purposes: it allows first-hand insights into the benefits and limitations of the environment, it enables the use of Arcadia analysis tools on the environment itself, and it gives an example of a large system maintained by the environment. Since the initial analysis tools will analyze Ada programs, we are writing the environment itself in Ada. In order to facilitate wide distribution and portability of the environment, it will use the X window system [SG86] and run on commonly available graphics workstations, such as Sun's. We intend for the environment to encapsulate its dependencies on specific operating systems and underlying hardware, which is a natural compromise between too much emphasis on portability (e.g., early *Toolpack*) and not enough (e.g., *Cedar*). The environment will include tool-building tools to assist in the generation and customization of new tools.

The project goals encompass a wide range of concerns, such as extensibility, integration, and portability, but it is important to note that the primary focus is on the underlying software research issues highlighted throughout this paper. In particular, the consortium does not intend to deliver a production-quality version of the environment. Our plans call for a separate organization to build such a system and provide user support.

The commercial members of the Arcadia consortium, TRW and Incremental Systems Corporation, are taking the lead in our technology transfer effort. Our technology transfer plan spans all the principles mentioned earlier. We plan, for example, multi-phased empirical studies to evaluate the effectiveness of the environment and constituent tools in a variety of problem areas. One of the studies will examine the use of Arcadia in a large-scale software project, most likely taking place at TRW. We also plan to identify champions within development organizations to help catalyze the adoption of the environment. Industrial affiliates of the consortium may also contribute to the technology transfer role.

Two specific ways in which we are seeking to make the prototype environments useful to a wide community are through providing operators supporting Ada language processing and providing a process program incorporating the Spiral Model of software development. Each of these is discussed briefly below.

Operators for Ada Language Processing Initial releases of Arcadia environments will have operators in them that are of interest to the broad community of Ada software developers. Rather than describe the process programs Arcadia will support, we briefly describe here some of the operators and operand types being produced that have use in many such process programs.

A common internal representation for representing Ada programs is essential to this set of operators. It must be simple so that it is clearly understood by the designers of the various tools and components and so that it is not burdensome to use. By its very definition, a common internal form is used by various tools and is indeed a medium of communication (at least among the front end components) and cannot, therefore, include tool-specific information. Instead, there must be simple and efficient support for management of tool specific information. A well designed common internal form facilitates tool development and promotes efficiency within the environment.

The basic components for a "front-end" for Ada language processing are a lexical analyzer, a parser, and a semantic analyzer. Interfaces to each of these components must be carefully designed to allow substitution and reuse of components. It seems quite likely, for instance, that there will be at least two semantic analysis components: one for Ada and one for additional restrictions. The Ada version will include exactly those semantic restrictions imposed by the Ada language [ALR83]. The second can impose additional restrictions imposed by a particular project or organization, such as a ban on `goto` statements or required local exception handling for all locally defined exceptions.

Such a front end can be used alone — to process Ada text into an internal form — or in combination with other component sets. For instance, a print component can be added to make a pretty printer. The components that compose code generation, along with those for library management, linking and loading, and run time facilities can be added to make a compiler. Components for editing and holophrasting can be added to the print component and the front end components to make an incremental semantic editor[2]. An interpreter can be composed from previously defined components with the addition of components that perform semantic actions corresponding to operators in an Iris tree (the internal form used by this set of operators) and an interpreter driver. In one interpreter effort, the semantic actions can include compiled code, symbolic interpretation or interpretation of the Iris structure [EZ86]. Components for interpretation, loading and linking and debugging can be added to the editing components and the front end to make an interactive debugger. The possibilities are numerous. The same front end components will be used by tools of various types throughout the environment.

The reuse of the front end components leads to reuse of objects as well. Once the front end has been invoked on a particular piece of Ada source text, say for compilation, there is no reason for the front end to be re-invoked as part of an invocation of the pretty printer, interpreter, debugger, or other analysis tools that might exist. The objects that are reused include not only the Iris trees themselves, but also additional information that is generated by components, but which is not common enough to be part of Iris.

A Process Program for the Spiral Model Another aspect of our technology transfer effort concerns process programming. As described earlier, the Arcadia project is developing an environment in which support for the specification of software processes is facilitated through use of a process programming language. Early users of *Arcadia-1* will be provided with modular software process programs as well as tools for the modification of these process programs and addition of new ones. Our intent is to encourage early users to experiment with software processes and to precipitate consensus about the nature of effective software processes. In the Arcadia project, we have been investigating process architecture issues and approaches, including the conceptual framework of the Spiral Model of the software lifecycle [Boe85] [BB86]. The risk-driven nature of the Spiral Model allows a project to configure its process architecture around its major sources of risk. For example, a prototype-intensive process may be used to address user interface uncertainty risks; a design-to-cost or design-to-schedule process may be used to address risks of not meeting tight budget or schedule constraints. We have done some initial work in expressing the Spiral Model as a process program, and have incorporated spiral-model risk management concepts in the development plan for *Arcadia-1*. We plan to further elaborate the Spiral Model in the context of process programming, and

[2] An incremental semantic editor includes full semantic analysis (i.e. type checking and overload resolution) at each editing step.

to incorporate automated aids for software risk management into *Arcadia-1*.

8 Summary and Conclusion

The current flurry of activity in environments and in software process specification is exciting. A proper focus for environments — supporting the user's multiple, complex activities — is being reemphasized at a time when some pertinent subtechnologies are maturing. This paper has presented definitions that are useful in categorizing and assessing developments in environments, and has attempted to separate some key concerns. In so doing, a number of emerging principles and important open problems have been identified and some promising research directions described.

One key distinction is between an environment's fixed infrastructure and its variant part. As part of the infrastructure, a user interface management system provides communication between humans and executing software processes. These processes are described in a formal process programming language and are interpreted by a process program interpreter. Mundane, automatable activities are handled directly; creative activities are performed by creative agents: people. A key component of the automated interpreter is an object management subsystem, whose typing system, relationship system, persistence scheme, and facilities for distributed and concurrent object management, support the constructs of the process programming language. Having process programming as a key part of the concept makes the environment an active agent, rather than a purely reactive one.

In our estimation, progress on the various fronts of environment research is now tied to realistic prototype development, empirical evaluation, and technology transfer. Prototypes are needed to validate concepts, generate feedback, and provide demonstrations that new environment technologies are useful to large development teams tackling large development activities. To be fully convincing, and to generate as much insight as possible, realistic prototypes must be subjected to well-designed empirical evaluation. Carefully planned technology transfer activities are then needed to ensure that the sought-after benefits are fully realized.

The Arcadia consortium has been formed to do research in environment architectures. We are attempting to make major strides in the development of the fundamental technologies, develop prototypes, conduct careful empirical studies, and move the technology to industrial practice.

Acknowledgements

Arcadia has benefited from the ongoing encouragement of William L. Scherlis and Stephen L. Squires.

We gratefully acknowledge the contributions of participants from each institution of the consortium, who have been instrumental in shaping virtually every aspect of Arcadia. During the past two years key contributions have come from A. Brindle, D. Heimbigner, C. LeDoux, M. Penedo, I. Shy, S. Sykes, W. Tracz, and S. Zeil.

Additional contributions have been made by R. Adrion, G. Barbanis, M. Burdick, G. Clemm, R. Cowan, E. Epp, S. Gamalel-Din, C. Kelly, S. Krane, R. King, D. Luckham, T. Nguyen, D. Richardson, W. Rosenblatt, R. Schmalz, C. Snider, S. Sutton, P. Tarr, and D. Troup.

References

[ABC*83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, P.W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, 1983.

[ALR83] *Military Standard Ada Programming Language (ANSI/MIL-STD-1815A-1983).* American National Standards Institute, January 1983.

[BB86] Barry W. Boehm and Frank C. Belz. Reasoning about iteration: a cost-benefit approach. In Mark Dowson, editor, *Proceedings of the Third International Software Process Workshop*, pages 40–42, IEEE Computer Science Press, Breckinridge, Colorado, USA, November 1986.

[BBD77] Thomas E. Bell, David C. Bixler, and Margaret E. Dyer. An extendable approach to computer-aided software requirements engineering. *IEEE Transactions on Software Engineering*, SE-3(1):49–60, February 1977.

[BDG*87] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Kenneth Kahn, Sonya E. Keene, Gregor Kiczales, Larry Masinter, David A. Moon, Mark Stefik, and Daniel L. Weinreb. Common Lisp object system specification. April 1987. Draft document, X3J13 Committee for the Common Lisp Standards effort.

[Ber87] Philip A. Bernstein. Database system support for software engineering. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 166–178, IEEE Computer Society Press, Monterey, California, March 1987.

[Bjo87] Dines Bjorner. On the use of formal methods in software development. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 17–29, IEEE Computer Society Press, Monterey, California, March 1987.

[BKK*86] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: merging Lisp and object-oriented programming. In *OOPSLA '86: Object Oriented Programming Systems, Languages, and Applications*, pages 17–29, 1986. Appeared as *SIGPLAN Notices 21*(11).

[Boe85] Barry W. Boehm. A spiral model of software development and enhancement. In Jack C. Wileden and Mark Dowson, editors, *Proceedings of the Second International Software Process Workshop*, pages 22 – 42, IEEE Computer Science Press, Coto de Caza, Trabuco Canyon California, USA, March 1985.

[BR87] Victor R. Basili and H. Dieter Rombach. TAME: tailoring an Ada measurement environment. In *Proceedings of the Fifth National Conference on Ada Technology*, Arlington, VA, March 1987.

[BS84] Marc H. Brown and Robert Sedgewick. A system for algorithm animation. *Computer Graphics*, 18(3):177–186, July 1984.

[BS85] Victor R. Basili and Richard W. Selby. Calculation and use of an environment's characteristic software metric set. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 386–391, London, August 1985.

[CAI85] *Military Standard Common APSE Interface Set (CAIS), Proposed MIL-STD-CAIS.* Department of Defense, January 1985.

[Cam86] John R. Cameron. An overview of JSD. *IEEE Transactions on Software Engineering*, SE-12(2):222–240, February 1986.

[Car86] Thomas A. Cargill. The feel of pi. In *Winter 1986 USENIX Technical Conference*, pages 62–71, USENIX Association, Denver, Colorado, January 1986.

[CDF*86] Michael J. Carey, David J. DeWitt, Daniel Frank, Goetz Graefe, Joel E. Richardson, Eugene J. Shekita, and M. Muralikrishna. *The Architecture of the EXODUS Extensible DMBS: A Preliminary Report.* Technical Report CS-644, Computer Science Department, University of Wisconsin – Madison, Madison, May 1986.

[CHOW85] Geoffrey M. Clemm, Dennis Heimbigner, Leon Osterweil, and Lloyd G. Williams. Keystone: a federated software environment. In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, pages 80–88, June 1985.

[CLF86] *CLF Overview.* USC Information Sciences Institute, Marina del Rey, California, March 1986. Informal Report.

[CO86] Geoffrey M. Clemm and Leon J. Osterweil. *A Mechanism for Environment Integration.* Technical Report CU-CS-323-86, University of Colorado, Boulder, January 1986.

[Cur83] B. Curtis. Cognitive science of programming. In *Sixth Minnowbrook Workshop on Software Performance Evaluation*, Blue Mountain Lake, NY, July 19-22, 1983.

[CWW86] Lori A. Clarke, Jack C. Wileden, and Alexander L. Wolf. Graphite: a meta-tool for Ada environment development. In *Proceedings of the IEEE Computer Society Second International Conference on Ada Applications and Environments*, pages 81–90, IEEE Computer Society Press, Miami Beach, Florida, April 1986.

[DHKL84] Veronique Donzeau-Gouge, Gerard Huet, Gilles Kahn, and Bernard Lang. Programming environments based on structured editors: the Mentor experience. In *Interactive Programming Environments*, pages 128–140, McGraw-Hill Book Co., New York, 1984.

[EFRV86] Gerald Estrin, Robert S. Fenchel, Rami R. Razouk, and Mary K. Vernon. SARA (System ARchitects Apprentice): modeling, analysis, and simulation support for design of concurrent systems. *IEEE Transactions on Software Engineering*, SE-12(2):293–311, February 1986.

[EZ86] Edward C. Epp and Steven J. Zeil. *Ada Interpretation in a Tool-Fragment Environment.* Technical Report, Department of Computer and Information Science, University of Massachusetts at Amherst, 1986.

[Fel79] Stuart I. Feldman. Make—a program for maintaining computer programs. *Software — Practice & Experience*, 9(4):255–265, April 1979.

[GLB*83] Cordell Green, David Luckham, Robert Balzer, Thomas Cheatham, , and Charles Rich. *Report on a Knowledge-Based Software Assistant.* Technical Report, Kestrel Institute, June 1983.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language And Its Implementation.* Addison Wesley, 1983.

[HM85] Dennis Heimbigner and Dennis McLeod. A federated architecture for information management. *ACM Transactions on Office Information Systems*, 3(3):253–278, July 1985.

[HN86] A. Nico Habermann and David Notkin. Gandalf: software development environments. *IEEE Transactions on Software Engineering*, SE-12(12):1117–1127, December 1986.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Ins85] *Inside Macintosh*. Apple Computer, Inc., Cupertino, California, promotional edition, March 1985.

[LRPC84] David B. Leblang and Jr. Robert P. Chase. Computer-aided software engineering in a distributed workstation environment. *SIGPLAN Notices*, 19(5):104–112, May 1984. (Proceedings of the First ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments).

[LS83a] Butler W. Lampson and Eric E. Schmidt. Organizing software in a distributed environment. In *Proceedings of the ACM SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, pages 1–13, June 1983.

[LS83b] Barbara Liskov and Robert Scheifler. Guardians and actions: linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[Mey86] Norman Meyrowitz, editor. *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, ACM SIGPLAN, Portland, Oregon, September 1986. (Appeared as SIGPLAN Notices, 21(11), November 1986.).

[Moo86] David A. Moon. Object-oriented programming with flavors. In *OOPSLA '86: Object Oriented Programming Systems, Languages, and Applications*, pages 1–8, 1986. Appeared as *SIGPLAN Notices 21*(11).

[Mye83] Brad A. Myers. Incense: a system for displaying data structures. *Computer Graphics*, 17(3):115–125, July 1983.

[OSD86] Jack A. Orenstein, Sunil K. Sarin, and Umeshwar Dayal. *Managing Persistent Objects in Ada*. Technical Report CCA–86–03, Computer Corporation of America, Cambridge, Massachusetts, May 1986.

[Ost86] Leon Osterweil. *A Process-Object Centered View of Software Environment Architecture*. Technical Report CU-CS-332-86, University of Colorado, Boulder, May 1986.

[Ost87] Leon Osterweil. Software processes are software too. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 2–13, Monterey, California, March 1987.

[PC86] David Lorge Parnas and Paul C. Clements. A rational design process: how and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2):251–257, February 1986.

[Ras86] Richard F. Rashid. From RIG to Accent to Mach: the evolution of a network operating system. In *Proceedings of the Fall Joint Computer Conference*, pages 1128–1137, November 1986.

[Rei85] Steven P. Reiss. PECAN: program development systems that support multiple views. *IEEE Transactions on Software Engineering*, SE-11(3):276–285, 1985.

[RJ77] Douglas T. Ross and Kenneth E. Schoman Jr. Structured analysis for requirements definition. *IEEE Transactions on Software Engineering*, SE-3(1):6–15, February 1977.

[Ros86] Graham Ross. Integral C — a practical environment for c programming. In *Proceedings of the Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 42–48, December 1986.

[Roy70] Winston W. Royce. Managing the development of large software systems. In *Proceedings, IEEE WESCON*, pages 1–9, IEEE, August 1970. also reprinted in Proceedings 9th International Conference on Software Engineering, pp. 328–338.

[RRJC85] Ronald A. Radice, Norman K. Roth, Albert C. O'Hara Jr., and William A. Ciarfella. A programming process architecture. *IBM Systems Journal*, 24(2):79–90, 1985.

[SB86] Mark Stefik and Daniel Bobrow. Object-oriented programming: themes and variations. *AI Magazine*, 6(4):40–62, Winter 1986.

[SBB87] Richard W. Selby, Victor R. Basili, and F. T. Baker. CLEANROOM software development: an empirical evaluation. *IEEE Transactions on Software Engineering*, SE-13, 1987. (to appear).

[SBK86] Mark J. Stefik, Daniel G. Bobrow, and Kenneth M. Kahn. Integrating access-oriented programming into a multiparadigm environment. *IEEE Software*, 3(1):10–18, January 1986.

[Sel85] Richard W. Selby. *Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics.* Technical Report TR-1500, Ph.D. Dissertation, Department of Computer Science, University of Maryland, College Park, Maryland, 1985.

[Sel86] Richard W. Selby. Combining software testing strategies: an empirical evaluation. In *Proceedings of the Workshop on Software Testing*, pages 82–90, ACM/SIGSOFT and IEEE-CS Software Engineering Technical Committee, Banff, Canada, July 1986.

[Sel87] Richard W. Selby. Incorporating metrics into a software environment. In *Proceedings of the Fifth National Conference on Ada Technology*, Arlington, VA, March 1987.

[SG86] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, April 1986.

[SHN*85] M. Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector, and Michael J. West. The ITC distributed file system: principles and design. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 35–50, 1985.

[Spe87] Alfred Z. Spector. *Distributed Transaction Processing in the Camelot System.* Technical Report CMU–CS–87–100, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, January 1987.

[SR86] Michael Stonebraker and Lawrence A. Rowe. The design of POSTGRES. In *Proceedings of the ACM SIGMOD '86 International Conference on Management of Data*, pages 340–355, June 1986.

[ST84] Thomas A. Standish and Richard N. Taylor. Arcturus: a prototype advanced Ada programming environment. *Software Engineering Notes*, 9(3):57–64, May 1984. (Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments).

[Sun86] *SunView Programmer's Guide.* Sun Microsystems, Inc., Mountain View, California, February 1986.

[SZBH86] Daniel Swinehart, Polle Zellweger, Richard Beach, and Robert Hagmann. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–490, October 1986.

[SZR86] Andrea H. Skarra, Stanley B. Zdonik, and Stephen P. Reiss. An object server for an object-oriented database system. In *Proceedings of the Object-Oriented Database Systems Workshop*, pages 196–204, IEEE Computer Society Press, September 1986.

[Tei85] Warren Teitelman. A tour through Cedar. *IEEE Transactions on Software Engineering*, SE-11(3):285–302, 1985.

[Tha82] Richard M. Thall. The KAPSE for the Ada Language System. In *Proceedings of the AdaTEC Conference on Ada*, pages 31–47, ACM, October 1982.

[Tic79] Walter F. Tichy. Software development control based on module interconnection. In *Proceedings of the Fourth International Conference on Software Engineering*, pages 29–41, IEEE Computer Society Press, Munich, West Germany, September 1979.

[Tic82] Walter F. Tichy. Design, implementation, and evaluation of a revision control system. In *Proceedings of the Sixth International Conference on Software Engineering*, pages 58–67, Tokyo, Japan, September 1982.

[TM81] W. Teitelman and L. Masinter. The Interlisp programming environment. *Computer*, 14(4):25–33, April 1981.

[TR81] Tim Teitelbaum and Thomas Reps. The Cornell Pprogram Synthesizer: A syntax directed programming environment. *Communications of the ACM*, 24(9):563–573, September 1981.

[War86] Paul T. Ward. The transformation schema: an extension of the data flow diagram to represent control and timing. *IEEE Transactions on Software Engineering*, SE-12(2):198–210, February 1986.

[WCW86] Alexander L. Wolf, Lori A. Clarke, and Jack C. Wileden. A formal model for describing and evaluating visibility control mechanisms. In *Proceedings of the IEEE Computer Society 1986 International Conference on Computer Languages*, pages 182–189, IEEE Computer Society Press, Miami Beach, Florida, October 1986.

[WHBK86] N. H. Weiderman, A. N. Habermann, M. W. Borger, and M. H. Klein. A methodology for evaluating environments. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 199 – 207, December 1986.

[WPE*83] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Proceedings of the*

Ninth ACM Symposium on Operating Systems Principles, pages 49–70, 1983.

[WPSK86] Anthony I. Wasserman, Peter A. Pircher, David T. Shewmake, and Martin L. Kersten. Developing interactive information systems with the user software engineering methodology. *IEEE Transactions on Software Engineering*, SE-12(2):326–345, February 1986.

[YTT87] Michal Young, Richard N. Taylor, and Dennis B. Troup. *Software Environment Architectures and User Interface Facilities.* Technical Report Arcadia Document UCI-87-07, Department of Information and Computer Science, University of California, Irvine, May 1987.

[ZS86] Pamela Zave and William Schell. Salient features of an executable specification language and its environment. *IEEE Transactions on Software Engineering*, SE–12(2):312–325, February 1986.

[ZW86] Stanley B. Zdonik and Peter Wegner. Language and methodology for object-oriented database environments. In *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, pages 378–387, January 1986.