Models of Software Development Environments

Dewayne E. Perry
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

Gail E. Kaiser
Columbia University
Department of Computer Science
500 West 120th Street
New York, NY 10027

January 1991

Abstract

We present a general model of software development environments that consists of three components: structures, mechanisms and policies. The advantage of this model is that it distinguishes intuitively those aspects of an environment that are useful in comparing and contrasting software development environments. Our initial application of the model is to characterize four classes of environments by means of a sociological metaphor based on scale: the individual, the family, the city and the state models. The utility of the IFCS taxonomy is that it delineates the important classes of interactions among software developers and exposes the ways in which current software development environments inadequately support the development of large systems. We demonstrate the generality of our model by also applying it to a previously published taxonomy that categorizes environments according to how they relate to four historical trends: language-centered, structure-oriented, toolkit and method-based environments.

Index Terms: Programming Environments, Software Development Environments, Models, Taxonomies, Policies, Scale, Coordination, Cooperation.

1. Introduction

A model is useful primarily for the insight it provides about particular instances and collections of instances. By abstracting away non-essential details that often differ in trivial ways from instance to instance and by generalizing the essential details into the components of the model, we derive a tool for evaluating and classifying these instances, perhaps in ways that we had not thought of before we constructed our model. It is with this purpose in mind — classification and evaluation — that we introduce the SMP model, which consists of three components: structures, mechanisms and policies. The advantage of this model is that it distinguishes intuitively those aspects that are useful in comparing and contrasting software development environments (SDEs), particularly with respect to which of the components is most emphasized.

Given this general model of software development environments, there are various points of view from which we might taxonomize environments. We might, for example, classify the SDEs according to their coverage of the software life cycle, or classify them according to the viewpoint of a particular role, such as environment builder, tool builder or environment user. Each of these classifications yields useful comparisons and insights.

Another important point of view, which we have not seen in the literature, is a classification of SDEs relative to the problems of *scale*: what is required of environments for projects of different sizes taking into account the numbers of people and the length of the project as well as the size and complexity of the system. Note that the distinction between programming-in-the-small and programming-in-the-large [10] has some intimations of the problems of scale. However, this distinction is basically one of single-unit versus multiple-unit systems and captures only a small part of this problem. We build software systems that range from small to very large, and will be able to build even larger systems as hardware gets cheaper and more powerful. What has not been sufficiently considered is the effect of the scale of systems on the tools needed to build them[1].

Therefore, this paper has a dual focus: first, we present the SMP model; and second, we explore its application to the classification of SDEs. We propose a sociological metaphor that emphasizes this problem of scale and provides insight into the environmental requirements for supporting the technical development phases of different sized projects. This metaphor suggests four classes of models: individual, family, city and state. We call this the IFCS taxonomy. Environments classified as individual and family models represent the current state of the art but are inadequate for building very large systems. We argue that the city model is adequate but that relatively little attention has been given to this class. Further, we argue that future research and development should address the city model and the proposed state model.

The SMP model, however, is not specific to this application. To demonstrate generality, we also consider a second taxonomy, Dart *et al.* [8], based on historical trends representing language-centered, structure-oriented, toolkit and method-based environments. We show how these classes of environments can be characterized in terms of our model.

---

1. For example, Howden [26] considers SDEs for medium and large systems only from the standpoint of capitalization and richness of the toolset.

In section 2, we present our SMP model of software development environments, discuss the individual components and their interrelationships, and illustrate various distinctions that we make with environments from the literature. Since this paper is not intended to be a survey, we do not describe any of these environments in depth. In section 3, we classify SDEs into the IFCS taxonomy suggested by our metaphor, characterize these classes, present a basic model for each class in terms of structures, mechanisms and policies, and categorize a wide variety of existing environments into the individual, family and city classes (we know of no examples of the state class). We discuss the trend-oriented taxonomy in section 4. Finally, in section 5 we summarize the contributions of the SMP model and the IFCS classification scheme, discuss possible applications of the SMP model, and suggest some open issues in environmental support for building large systems.

2. A General Model of Software Development Environments

Our general model of software development environments consists of three interrelated components: structures, mechanisms and policies.

General SDE Model = ({ Structures }, { Mechanisms }, { Policies })

- Structures are the underlying objects and object aggregates on which the mechanisms operate;
- mechanisms are the visible and underlying tools and tool fragments; and
- policies are the rules, guidelines and strategies imposed on the programmer[2] by the environment.

We appeal to a variety of sources as justification for our three-component model. First, separating policies from mechanisms is a well-known strategy in operating system kernels such as Hydra [89]. This separation of the two notions enables the user of an operating system to fine-tune the desired policies by means of supplied mechanisms. More recently, this separation has been re-emphasized in research on enactable process models (for example, consider Osterweil's process programming [52, 78] and the discussion on policies at the 5th International Software Process Workshop [58]). Second, the importance of structure and its effect on both policies and mechanisms has not been sufficiently appreciated until recently (for example, consider Wileden, Clarke and Wolf's [87] discussion of the interaction between structures and problems of change, and various discussions of object management issues [4, 49, 55, 68, 78]).

In general, these three components are strongly interrelated: decisions about one component may have serious implications for the other two components and place severe limitations on them. We discuss each of these components of the model, illustrate them with examples from the SDE literature, and discuss their interdependencies.

---

2. We use the term *programmer* in a generic sense to include any user of the environment and use the two terms interchangeably.

2.1 Structures

Structures are those objects or object aggregates that represent the software under development. In the simplest (and chronologically, earliest) incarnation, the basic structure is the file system (as in UNIX™ [38] , for example). The trend, however, is towards more complex and comprehensive structures, representing additional information beyond the basic software artifacts. One reason for complex objects is found in integrated environments, particularly those centered around a syntax-directed editor [15, 64]. These SDEs share a sophisticated internal representation such as an abstract syntax tree [13] or a graph structure [39, 74, 88] to gain efficiency in each tool (because, for example, each tool does not need to reparse the textual form, but uses the intermediate, shared representation). The disadvantage of this approach is that it is difficult to integrate additional tools into the environment, particularly if the structure provided does not integrate well the mechanisms and their intended policies. Garlan's tool views [18] provide a partial[3] solution: a structure and a mechanism for generating the underlying common structure consistent with all the requirements of the different tools in the SDE.

Another reason for this trend is to maintain more information about software objects to support more comprehensive mechanisms and policies. For example, the use of project databases has been a topic of considerable interest in the recent past [4, 49]. The basic structure currently generating a large amount of interest is the *objectbase* [23, 55, 68]; it is hoped that this approach will solve the deficiencies of file systems and conventional databases.

These basic structures are the foundation for building more complex and more comprehensive higher-order structures. For example, Inscape [56, 57] maintains a complex semantic interconnection structure among the system objects to provide comprehensive semantic analysis and version control mechanisms and policies about semantic consistency, completeness and compatibility among the system objects. Smile's [31] experimental database is a higher-order organization of basic structures that supports mechanisms and policies for managing changes to existing systems. The Project Master Data Base (PMDB) [54] provides an entity-relationship-attribute model [7] to represent, for example, problem reporting, evaluation and tracking processes. CMS's Modification Request Tracking System [69] builds a structure that is intertwined with Source Code Control System's [67] version management database (which in turn is built on top of the UNIX file system); it coordinates change requests with the actual changes in the system. Horwitz [24] combines the attribute grammar and relational database models into a single integrated structure, as a proposed extension to the Synthesizer Generator [63][4]. Finally, Apollo's Domain Software Engineering Environment (DSEE) [41] provides a comprehensive set of structures for coordinating the building and evolving of software systems; these structures support configuration control, planning and programmer interactions.

---

3. We say *partial* in the sense that Garlan's views do not help at all if the environment and its tools already exist independently of Garlan's mechanisms and new tools need to be added. It is a *full* solution in the sense that if one develops the entire environment with Garlan's views, then adding a new tool requires the addition of the view needed by that tool to the original set, unmerging and then remerging the integrated structure.
4. Note this should be distinguished from the implementation of a syntax-directed editor in terms of a relational database management system, as investigated by Linton [44].

In general, structures tend to impose limitations on the kinds of policies that can be supported and enforced by SDEs. Simple structures such as files provide a useful communication medium between tools but limit the kinds of policies that can be supported. The more complex structures required by integrated environments such as Gandalf [50] enable more sophisticated policies, but make it harder to integrate new mechanisms and policies into the environment. Higher-order structures such as Infuse's hierarchy of experimental databases [36] and Cosmos's versions, configurations and histories [83] make it possible to enforce policies that govern the interactions of large groups of programmers, but do not allow the policy maker the ability to define his or her own policies.

Similarly, structures impose limitations and requirements on mechanisms. For example, textual files make sharing of data very easy among different tools, but impose the need for parsing in each tool. Moreover, structures limit the mechanisms' processing ability: decorated abstract-syntax trees provide much more amenable structures for semantic analysis than text structures do.

One fact should be clear: we have not yet reached a level of maturity in our SDEs with respect to structures. There is still a feeling of exploration about the kinds of structures that are needed. Indeed, there is the same feeling of exploration about the policies that can or should be supported by an SDE, particularly for those SDEs that are concerned with large-scale projects.

2.2 Mechanisms

Mechanisms are the languages, tools and tool fragments that operate on the structures provided by the environment and that implement, together with structures, the policies supported and enforced by the environment. Some of these mechanisms are visible to the programmers while others may be hidden and function as lower-level support mechanisms. For example, UNIX provides tools directly to the user, while Smile hides these tools beneath a facade that provides the programmer with higher-level mechanisms (that in turn invoke individual UNIX tools).

Analogous to the growing sophistication in structures, there is an increasing sophistication in mechanisms. As mechanisms become better understood and more standardized, they tend to become embedded into environment and tool generation systems such as the Synthesizer Generator and Yacc [29]. Further, improved insight into the interactions between mechanisms and structures has led to capabilities for deeper analyses of the software artifacts and their intra- and interrelationships. Consider, for example, dataflow analysis for change management [70], program-based testing [85] and merging of non-interfering versions [25].

It is difficult to develop mechanisms entirely divorced from policies. In particular, most mechanisms implicitly enforce some policies — that is, the policies are hard-wired into the mechanisms. For example, static linker/loaders generally require all externally referenced names to be defined in the set of object modules that are to be linked together. This requirement, together with the requirement that only linked/loaded objects may be executed, induces a policy of having to create and compile all of the modules before linking them. A different strategy is possible for execution preparation tools that provide dynamic rather than static linking and, hence, induce a different policy: for example, Multics' segmentation scheme [51] allows externally referenced names to be resolved at run-time.

It is also possible for mechanisms to support a range of policies through explicit encodings. Mechanisms such as shell scripts [30], Osterweil's process programming [52, 78], Darwin's [47], CLF's [86] and Marvel's [32] rules, and other approaches to enactable process models [82, 58] enable the policy maker to define requirements and restrictions on the interactions among programmers, among programmer and tools, and among tools. How well these explicit policies can be enforced depends on how well the mechanisms constrain the programmer in using the environment.

Of the two approaches, hard-wired policies are currently more common. There are good historical reasons for this situation: we first must work out particular instances before we can generalize them. Particular mechanisms and structures that implicitly encode policies must be built first in order to reach a sufficient understanding of the important issues. Once we have reached this level of maturity, we can then separate the specification of policies from mechanisms and structures.

2.3 Policies

Policies are the requirements imposed on the user of the environment during the software development process. As discussed in the previous sections, these rules and strategies are often hard-coded into the environment by the structures and mechanisms. In most cases, the design of the tools and the supporting structures define or impose the policies.

There are two important distinctions to consider about policies: first, whether policies are *supported* or *unsupported* by the environment; and second, whether policies are *enforced* by the environment or enforced by consensus. If a policy is supported, then the mechanisms and structures provide a means of satisfying that policy. For example, if top-down development is a supported policy, we would expect to find tools and structures that enable the programmer to build the system in a top-down fashion. By implication, we would also expect to find (other, or perhaps, the same) tools and structures to build systems in other ways as well.

If a policy is *enforced* by the environment, then not only is it supported, but it is impossible to violate that policy within the environment. Enforcement can also be by consensus — that is, a policy decision can be made outside the environment either by management or by convention, but once made it is supported but not enforced by the environment. For example, suppose that management decides that all released systems are to be generated only from modules resident within SCCS. The environment supports configuration management with SCCS; however, it is the management decision that forces the programmers to control their modules within SCCS. This does not mean that one cannot build a system outside SCCS (because one can). It does mean, however, that if one wishes to build a releasable system, then it must be built from components in SCCS. A policy might be determined by consensus but not supported by the environment. For example, a minimal development environment might require a physical signout sheet to keep track of which programmer is working on which modules.

There is a further distinction to be made between those policies that apply to mechanisms and structures and those that apply to other policies. We refer to the latter as *higher-order* policies. For example, 'all projects will be done in Ada' is a higher-order policy. In general, higher order policies are those policies concerned with the creation, instantiation, evolution, dissolution, measurement, administration and management of policies about environmental and process policies. The recognition of higher-order policies, and their impact on environments, appears to be in its infancy.

Thus, analogous to the trend towards more sophisticated structures and mechanisms, there is a trend towards more global and comprehensive policies that are explicitly described. Moreover, we are beginning to see the presence of higher-order policies, that is, policies about policies.

3. SDEs in the Context of Scale

We present a classification of SDEs from the viewpoint of scale: how the problems of size — primarily the numbers of programmers, but by implication the size of the system as well — affect the requirements of an SDE that supports the development of those systems. The classification is based on a sociological metaphor that is suggestive of the distinctions with respect to the problems of scale. From a continuum of possible models, we distinguish the following four classes of SDE models:

Basically, each class incorporates those classes to its left: a family is a collection of individuals, a city is a collection of families and individuals, and a state is a collection of cities, families and individuals. Our metaphor also suggests that there may be further distinctions to be made to the right of the family model — for example, neighborhoods, villages, etc. However, as relatively little is known about SDEs that support a city model, and nothing is known about SDEs that support a state model, we make as few distinctions as possible. One of the purposes of this paper is to draw attention to these two representative classes: the city and the state.

We present two orthogonal characterizations for each class. The first emphasizes what we consider to be the key aspect that distinguishes it from the others in terms of scale. These aspects are:

- *construction* for the individual class of models;
- *coordination* for the family class;
- *cooperation* for the city class; and
- *commonality* for the state class.

The second characterization emphasizes the relationships among the components of the model with respect to that particular class of models. In the case of the first three characterizations, we report what we consider to be historical trends; in the last case, we posit the dominating component on the basis of the trends in the first three classes. Thus,

- mechanisms dominate in the individual class;
- structures dominate in the family class;
- policies dominate in the city class; and
- higher-order policies dominate in the state class.

For each class of models we present a description of the class and support our characterizations with example SDEs. For convenience in the discussion below, we use the term *model* instead of *class of models*.

3.1 The Individual Model

The individual model of software development environments represents those environments that supply the minimum set of implementation tools needed to build software. These environments are often referred to as *programming environments*. The mechanisms provided are the tools of program construction: editors, compilers, linker/loaders and debuggers. These environments typically provide a single structure that is shared among mechanisms. For example, the structure may be simple, such as a file, or complex, such as a decorated syntax tree. The policies are typically *laissez faire* about methodological issues and hard-wired for low-level management issues.

Individual Model =
(
{ *single structure*[5] } ,
{ *implementation tools* } ,
{ *tool-induced policies* }
)

These environments are dominated by issues of software *construction*. This orientation has led to an emphasis on the tools of construction — that is, the mechanisms — with policies and structures assuming secondary importance. The policies are induced by the mechanisms — that is, hard-wired — while the structures are dictated by the requirement of making the tools work together.

We discuss four groups of environments that are instantiations of the individual model: toolkit environments, interpretive environments, language-oriented environments, and transformational environments. The toolkit environments are exemplified by UNIX; the interpretive environments by Interlisp [79]; the language-oriented environments by the Synthesizer Generator [63]; and the transformational environments by Refine [73].

The toolkit environments are, historically, the archetype of the individual model. In general, these environments are not particularly distinguishable from general purpose operating systems (such as UNIX and VMS [11]) The mechanisms communicate with each other by a simple structure, the file system. Policies take the form of conventions for organizing structures (as for example in UNIX, the bin, include, lib and src directories) and for ordering the sequence of development and construction (as exemplified by Make [17] — for example, .o files depend on .c files, etc.). These policies are very weak and concerned with the minutiae of program construction. However, shell scripts provide the administrator with a convenient, but not very extensive, mechanism for integrating tools and providing support for policies beyond those encoded in the tools.

---

5. We use *italics* for general descriptions of the components and normal typeface for specific components.

Toolkit Model =

(

{ file system } ,

{ editors, compilers, linker/loaders, debuggers, ...[6] } ,

{ *tools-induced policies*, *script-encoded policies*, ... }

)

Interpretive environments are also an early incarnation of the individual model. They consist of an integrated set of tools that center around an interpreter for a single language such as Lisp[7] or Smalltalk. The language and the environment are not really separable: the language is the interface to the user and the interpreter the tool that the user interacts with. The structure shared by the various tools is an internal representation of the program, possibly with some accompanying information such as property lists. These environments are often considered to be ‘‘exploratory programming’’ environments, and as such are noted for their extreme flexibility. Because of this desired flexibility, there are virtually no policies enforced (or, for that matter, supported). Thus, in contrast to the toolkit approach where the tools induce certain policies that force the programmer into certain modes of operation, programmers can essentially do as they please in the construction of their software.

Interpretive Model =

(

{ intermediate representation } ,

{ interpreter, *underlying support tools* } ,

{ *virtually no restrictive policies* }

)

Language-oriented environments are a blend of the toolkit and interpretive models. They provide program construction tools integrated by a complex structure — a decorated syntax tree. Whereas the tools in the toolkit environments are batch in nature and the tools in the interpretive environments are interactive, the tools in language-oriented environments are incremental in nature — that is, the language-oriented tools try to maintain consistency between the input and output at the grain of editing commands. A single policy permeates the tools in this model: early error detection and notification. These environments might be primarily hand-coded, as in Garden [62], or generated from a formal specification, such as by the Synthesizer Generator.

---

6. The notation ‘‘. . .’’ at the end of the list indicates that additional facilities may be available.

7. Some of the extended Lisps, such as Interlisp [79], are also well-integrated with the underlying operating system and include the file system as part of it’s structure.

Language-Oriented Model =

(

{ decorated syntax tree } ,

{ editor, compiler, debugger, ... } ,

{ error prevention, early error detection and notification, ... }

)

Transformational environments typically support a wide-spectrum language (such as V [73]) that includes a range of data and control structures from abstract to concrete. Programs are initially written in an abstract form and modified by a sequence of transformations into an efficient, concrete form. The mechanisms are the transformations themselves and the machinery for applying them. The structure is typically a cross between the intermediate representation of the interpretive model and the decorated syntax tree of the language-oriented model. As in the language-oriented environments, a single policy defines the style of the environment: the transformational approach to constructing programs (as, for example, in Ergo [42] and PDS [6]). Programmer's apprentices, such as KBEmacs [65, 66], are a variation of this policy in that the programmer can switch between the transformational approach and interpretive approach at any time.

Transformational Model =

(

{ intermediate representation/decorated syntax tree, ... } ,

{ interpreter, transformational engine, ... } ,

{ transformational construction, ... }

)

We have discussed four different groups of individual models and cited a few of the many environments that are examples of these different models. Most research environments and many commercial environments are instances of these individual models.

3.2 The Family Model

The family model of software development environments represents those environments that supply, in addition to a set of program construction tools as found in an individual model, facilities that support the interactions of a small group of programmers (under, say, 10 people). The analogy to the family for this model is that the members of the family work autonomously, for the most part, but trust the others to act in a reasonable way; there are only a few rules that need to be enforced to maintain smooth interactions among the members of the family. These rules, or policies, distinguish the individual from the family model of environments: in the individual model, no rules are needed because there is no interaction; in the family model, some rules are needed to regulate certain critical interactions among the programmers.

Family Model =

(

{ ...[8], *special-purpose databases* } ,

{ ..., *coordination mechanisms* } ,

{ ..., *coordination policies* }

)

The characteristic that distinguishes the family model from the individual model is that of *enforced coordination*. The environment provides a means of orchestrating the interactions of the programmers, with the goal that information and effort is neither lost nor duplicated as a result of the simultaneous activities of the programmers. The structures of the individual model do not provide the necessary (but weak form of) concurrency control. Because the individual model's structures are not rich enough to coordinate simultaneous activities,[9] more complex structures are required. These complex structures dominate the design of the environment, where in the individual model the mechanisms dominate; the mechanisms and policies in the family model are adapted to the structures.

We discuss four groups[10] of environments that are instantiations of the family model: extended toolkit environments, integrated environments, distributed environments, and project management environments. The extended toolkit environments are exemplified by UNIX together with either SCCS or RCS [81], the integrated environments by Smile, the distributed environments by Cedar [77], and the project management environments by CMS.

The extended toolkit model directly extends the individual toolkit model by adding a version control structure and configuration control mechanisms (see, for example, UNIX PWB [38]). Programmer coordination is supported with these structures and mechanisms; enforced coordination is supplied by a management decision to generate systems only from, for example, SCCS or RCS databases. Thus, this kind of family environment provides individual programmers a great deal of freedom, with coordination supported only at points of deposit into the version control database. The basic mechanisms for program construction from the individual toolkit model are retained. However, these tools must be adapted to the family model structure — for example, Make must be modified to work with RCS or SCCS. Alternatively, the tools may be constructed in conjunction with a database — e.g., the Ada program support environments (APSEs) [48].

---

8. The notation ''. . .'' at the beginning of the list indicates that we include the structures, mechanisms and policies from the models found to the left of the current one on the continuum.
9. This is not quite true: some individual environments support simultaneous activities in multiple windows. However, we know of no individual environment that solves this induced concurrency control problem. The result is an environment rich enough to support concurrent activity but not rich enough to coordinate it.
10. These groupings are not necessarily mutually exclusive. In particular, either distributed or project management aspects can be mixed and matched with either extended toolkit or integrated environments.

Extended Toolkit Model =

(

{ ..., compressed versions, version trees } ,

{ ..., version/configuration management } ,

{ ..., support version/configuration control }

)

The integrated model extends by analogy the individual language-oriented model, where the consistency policy permeates the tools. Here consistency is maintained among the component modules in addition to within a module. As in the individual model, the mechanisms determine consistency incrementally, although the grain size ranges from the syntax tree nodes of the Mercury [35], to procedures in Smile, to entire modules in Toolpack [53] and R^n [12]. This model's structure is typically a special-purpose database, perhaps most prominent in SMS's [71] use of an object-oriented file system, although generic architectures such as CLF's Worlds [86], and Marvel populate their objectbases with respect to a data model. The structures vary in their support from simple backup versions to both parallel and sequential versions [22, 33].

Integrated Model =

(

{ ..., special-purpose database } ,

{ ..., version description languages, consistency checking tools } ,

{ ..., enforced version control, enforced consistency }

)

The distributed model is orthogonal to the extended toolkit and integrated models. We note, however, that a distributed-toolkit model is not particularly interesting, whereas instances of the distributed-integrated model enable useful work to continue because of their reliability and availability structures. In this latter case, the distributed model expands the integrated model across a number of machines connected by a local area network. Additional structures are required to support reliability and high availability as machines and network links fail. For example, Mercury is a multi-user, language-oriented environment that depends on a distributed, static semantic analysis algorithm to guarantee consistency among module interfaces; Mercury supports a cache consistency protocol that maintains global attribute consistency to the degree possible given network partitions. Sun Microsystem's Network Software Environment (NSE) provides most of facilities of DSEE [40], and adds multi-user integration environments [2] that require extensions to Sun's network file system (NSF) to support loop-back mount structures.

Distributed Model =

(

{ ..., distributed *objects* } ,

{ ..., *network mechanisms* } ,

{ ..., high reliability and availability }

)

As in the preceding case, the project management model is orthogonal to the extended toolkit and integrated models. Analogously, the project management model is most often tied to the integrated model

because of the additional leverage available from that merger in terms of enforced policies. Thus, these environments provide additional support for coordinating changes by assigning tasks to individual programmers. In DSEE, structures and mechanisms are provided for assigning and completing tasks that may be composed of subtasks and activities. CMS adds a modification request (MR) tracking system on top of SCCS in which individual programmers are assigned particular change requests and the changes are associated with particular sets of SCCS versions. It is in this class of models that we find hints of things to come in the city model discussed below. However, project management is primarily an added layer rather than an integral part of the family model.

Project Management Model =
(
{ ..., *activity coordination structures* } ,
{ ..., *activity coordination mechanisms* } ,
{ ..., support activity coordination }
)

The family model represents the current state of the art in SDEs. In general, it is an individual model extended with mechanisms and structures to provide a small degree of enforced coordination among the programmers. The policies are generally *laissez faire* with respect to most activities; enforcement of coordination is generally centered around version control and configuration management. The most elaborate instance of the family model with respect to mechanisms is CLF; the most elaborate with respect to structures is NSE.

3.3 The City Model

As the size of a project grows to, say, more than 20 people, the interactions among these people increase both in number and in complexity. Although families allow a great deal of freedom, much larger populations, such as cities, require complicated systems of rules and regulations with their attendant restrictions on individual freedom. The freedom appropriate in small groups produces anarchy when allowed to the same degree in larger groups. It is precisely this problem of scale and this complexity of interactions that leads us to introduce the city model.

City Model =
(
{ ..., *structures for cooperation* } ,
{ ..., *cooperation mechanisms* },
{ ..., *cooperation policies* }
)

The notion of enforced coordination of the family model is insufficient when applied to the scale represented by the city model. Consider the following analogy. On a farm, very few rules are needed to govern the use of the farm vehicles while within the confines of the farm. A minimal set of rules govern who uses which vehicles and how they are to be used — basically, how the farm workers coordinate with each other on use of the vehicles. Further, these rules can be determined in *real time* — that is, they can be adjusted as various needs arise or change. However, that set of rules and mode of rule determination is

inadequate to govern the interactions of cars and trucks in an average city: chaos would result without a more complex set of rules and mechanisms that enforce the cooperation among people and vehicles. An alteration of these rules necessarily has serious consequences because they affect a much larger population (consider the problem when Europe changed from driving on the left to driving on the right side of the road). Thus, *enforced cooperation* is the primary characteristic of the city model.

It is our contention that family model environments are currently being used where city model environments are needed, and that the family model is not appropriate for the task. Because the family model does not support or enforce an appropriate set of policies to handle the problems incurred by an increase in scale, an externally defined set of methodologies and management techniques is established for each project in an attempt to stave off the anarchy that can easily occur. These methodologies and management techniques work with varying degrees of success, depending on how well they enforce the necessary cooperation among programmers.

Little work has been done on environments that implement a city model — that is, that enforce cooperation among programmers. We discuss two such environments, Infuse[11] [59] and ISTAR [14, 76], and mention briefly a third approach by Shy, Taylor and Osterweil [72] that appears to hold some promise. Infuse focuses on the technical management of the change process in large systems, whereas ISTAR focuses on project management issues. In both cases, the concern for policies of enforced cooperation dominate the design and implementation: in Infuse, the policy of enforced cooperation while making a concerted set of changes by many programmers has led to the exploration of various structures and mechanisms; in ISTAR, the contractual model and the policies embodied in that model dominate the search for project management structures and mechanisms.

The primary concern of Infuse is the technical management of evolution in large systems — that is, what kinds of structures, mechanisms and policies are needed to determine the implications and extent of changes made by large numbers of programmers. Infuse generalizes Smile's experimental databases into a hierarchy of experimental databases, which serves as the encompassing structure for enforcing Infuse's policies about programmer interaction. These cooperation policies are enforced among programmers in several ways [36].

- Infuse automatically partitions [46] the set of modules involved in a concerted set of changes into a hierarchy of experimental databases on the basis of the strength of their interconnectivity (this metric is used as an approximation to the oracle that tells which modules will be affected by which interface[12] changes). This partitioning forms the basis for enforcing cooperation: each experimental database

---

11. Infuse is the joint work of the authors that began initially as the change management component of the Inscape Environment (which explores the use of formal interface specifications and of a semantic interconnection model in the construction and evolution of software systems). The management issues of how to support a large number of programmers are sufficiently orthogonal to the semantic concerns of Inscape to be applicable in a much wider context (for example, to environments and tools supporting a syntactic interconnection model) and to be treated independently. For this reason, Infuse has been implemented separately (initially at Columbia University, then — for the most part — reimplemented and extended at AT&T Bell Laboratories).

12. We mean ''interface'' changes in the broadest semantic sense — that is, we mean any change in external, visible behavior of a module, not just the easily detectable syntactic changes.

proscribes the limits of interaction (however, see the discussion of workspaces below).

- At the leaves of the hierarchy are singleton experimental databases where the actual changes take place. When the changes to a module are self-consistent, it may be deposited into its parent database. At each non-leaf database, the effects of changes are determined with respect to the components in that partition — that is, analysis determines the *local consistency* of the modules within the database and ignores any inconsistencies that might arise relative to modules outside the database. Only when the modules within a partition are locally consistent may the database be deposited into its parent. This iterative process continues until the entire system is consistent.

- When changes conflict, the experimental database provides the forum for negotiating and resolving those conflicts. Currently, there are no formal facilities for this negotiation, but only the framework within which the negotiation and resolution can take place. Once the conflicts have been resolved, the database is repartitioned and the change process repeats for that (sub-) partitioning.

- Because the partitioning algorithm is only an approximation of the optimal oracle, Infuse provides an escape mechanism, the *workspace*, in which programmers may voluntarily cooperate to forestall expensive inconsistencies that would normally be discovered at the top of the hierarchy where it is the most expensive to repartition and correct.

Thus the rules for interaction are encoded in the mechanisms, with the hierarchy providing the supporting structure.[13]

```
Infuse Model =
        (
          { ..., hierarchy of experimental databases } ,
          { ..., automatic partitioning,
                  local consistency analysis,
                  database deposit,
                  local integration testing,
                  ... } ,
          { ..., enforced and voluntary cooperation }
        )
```

Whereas Infuse is concerned with the technical problems of managing system evolution, ISTAR is primarily concerned with managerial problems. ISTAR is an integrated project support environment (IPSE) [43] and seeks to provide an environment for managing the cooperation of large groups of people producing a large system. To this end, it embodies and implements a *contract model* of system development. ISTAR does not directly provide tools for system construction but instead supports

13. We are also investigating the utility of this structure for cooperation in integration testing [37]. With a notion of local integration testing analogous to our notion of local consistency checking, we are able to assist the integration of changes further by providing facilities for test harness construction, and integration and regression testing within this framework. A side-effect of this work was the exposure of special problems in adequately testing object-oriented programs [60].

"plugging in" various kinds of workbenches. The contract model dictates the allowable interactions among component programmers [76].

- The client specifies the required deliverables — that is, the products to be produced by the contractor. Further, the client specifies the terms of satisfaction for the deliverables — that is, the specific validation tests for the products.
- The contractor provides periodic reporting about the status of the project and the state of the product being developed. Clients are thus able to monitor the progress of their contracts.
- ISTAR provides support for amending the contracts as the project develops. Thus, the contract evolution is analogous to product evolution.
- A contract database provides the underlying structure for this environment. This structure is hierarchical, with each subcontract getting its own contract database.

Thus the interactions between the clients and the contractors are proscribed by the underlying model and the mechanisms in the environment enforce those rules of interaction. The exact interaction of tools in the construction of the components of the system is left unspecified, but the means of contracting for components of a system are enforced by the environment.

ISTAR Model =
(
{ ..., contract data base } ,
{ ..., *contract support tools* } ,
{ ..., contract model }
)

The research focus of the Arcadia Environment [78] has been that of providing a framework for building environments — that is, the emphasis has been on the underlying and supporting components that are needed in a wide variety of different environments — rather than building a specific environment. As such, it is neutral with respect to our IFCS classification. However, Shy, Taylor and Osterweil provides a philosophical model for the Arcadia framework that is rather suggestive of how Arcadia could be applied to the problems of scale. Their corporation model defines roles for the users of a system and a framework for defining policies concerning the interactions among these roles, among the roles and environmental mechanisms, and among the various mechanisms in the environment (in this case, within the framework of the Arcadia environment). The Arcadia framework together with the corporation model should be sufficient to build an interesting city model SDE.

3.4 The State Model

Pursuing our sociological metaphor leads to the consideration of a state model. Certainly the notion of a state as a collection of cities is suggestive of a company with a collection of projects. Just as individuals and families may function differently in the context of a city from the way they would as autonomous units, so too may cities function somewhat differently in the context of a state.

There are, we think, intimations of this model in the following kinds of strategies:

- developing a new standard tool for building common user interfaces, such as X windows;
- standardizing on a small number of languages, such as Ada [1], Common Lisp [75] or C++, for all projects;
- establishing a uniform development environment such as UNIX, CAIS [48], or PCTE [80] for all projects; and
- establishing a common methodology or set of standards to be used on all its projects.

It is easy to understand the rationale behind these decisions: reduction in cost and improvement in productivity. If there is a uniform environment used by several projects, programmers may move freely between projects without incurring the cost of learning a new environment. Further, reuse of various kinds is possible: tools may be distributed with little difficulty; code may be reused; design and requirements may be reused; etc. It should be noted that this implies a relatively high level of process maturity[14] [27] as well technical maturity.

State Model =
(
{ ..., *supporting structures* } ,
{ ..., *supporting mechanisms* } ,
{ ..., *commonality policies* }
)

In this model, the concern for commonality, for standards, is dominant. This policy of commonality tends to induce policies in the specific projects (that is, in their city model environments). Thus, the policies of the state model are higher-order policies because they have this quality of inducing policies, rather than particular structures and mechanisms.

While one can imagine the existence of instances of this model (and there are certainly many cases where it is needed), we do not know of one. Our intuition[15] suggests the following general description.

- Provide a generic process model with its attendant tools and supporting structures for software development to be used throughout a particular organization.
- Instantiate the model for each project, tailoring each instance dynamically to the particular needs of the individual project.
- Manage the changes in all the instances and the differences between the various instances to support movement of resources (in a generic sense) between projects.

---

14. Humphrey lists 5 levels of process maturity: initial, repeatable, defined, managed, and finally, optimized.
15. See various position papers and discussions in the 4th International Software Process Workshop: Representing and Enacting the Software Process [82].

While little is known about the state model, it appears to be a useful and fruitful area for investigation. Research on enactable[16] process models appears to hold considerable promise. The important question is to what extent can the modeling of the product development and evolution process be applied to modeling the process development and evolution process. For example, is Shy, Taylor, and Osterweil's corporation model, together with the Arcadia Framework (specifically, the process management facilities), sufficient to provide a state model environment? This question applies equally well to other process modeling environments.

3.5 Scaling Up from One Class to the Next

Ideally, scaling up from one class to the next would be a matter of adding structures, mechanisms and policies on top of an existing environment. In at least one case this has been done without too much difficulty: scaling up from the individual toolkit model to the family extended toolkit model. This example involves only a small increment in policy.

It is extremely attractive to think of the higher-level models as using the lower-level models as components upon which to establish new structures, mechanisms and policies. Unfortunately, there are several difficulties. First, there is the problem of the tightness of coupling between structures and mechanisms. Even in scaling up from the toolkit to the extended toolkit environments, retrofitting of old tools to new structures is necessary. This raises the fundamental question of whether it is more profitable to retrofit changes into the system or to reconstruct the entire environment from scratch. For example, environment generators assume a common kernel that is optimized for a specific model, and often a particular group within the model. Consequently, they are difficult to scale up. Mercury scales up the Synthesizer Generator by extensive modifications to its common kernel rather than by adding something to coordinate generated editors. Infuse provides another example: since it is a direct generalization of Smile, we initially attempted to extend Smile's implementation. This strategy failed and the current implementation is completely independent of Smile. NSE, on the other hand, did succeed in scaling up UNIX to support optimistic concurrency control, but this required modifications to the operating system's kernel.

Second, problems arise from the lack of structures and mechanisms in the base-level environment suitable for the next level. For example, multi-user interpretive environments are extremely rare. Further, this lack of suitable structures and mechanisms is particularly important in moving from the family model to the city model where enforcement is a much more serious issue. Building security measures on top of a permissive environment (such as UNIX) is particularly difficult; it is too easy to subvert the enforcement mechanisms.

Last, there is the problem of how well the granularity of the structures and the mechanisms of one level lend themselves to supporting the next level. For example, the file system in the toolkit approach is easily adapted to the extended toolkit. However, some of the higher-level structure of the extensions is embedded, by convention, within the lower-level structure, as in SCCS where version information is embedded by an SCCS directive within the source files.

---

16. The term ''enactable'' is used in the software process research community as a neutral term for ''executable'' because it implies very little about how the executability is created.

Note that most of our examples illustrating scaling difficulties are, of necessity, from the individual to the family model. Since this increment is much smaller than from family to city, we can expect greater obstructions in scaling from the family to the city model.

The fundamental source of these various problems lies in the fact that the structures, mechanisms and policies required for the higher levels of environments are, in some sense, more basic and must be designed in rather than just added on.

4. SDEs in the Context of Historical Trends

To show the generality of our model, we apply it to an existing taxonomy presented in Dart *et al.* [8]. SDEs are grouped as follows: language-centered environments, structure-oriented environments, toolkit environments, and method-based environments. These categories represent trends and historical lineage, but are not necessarily orthogonal to each other. Unlike the preceding discussion, the categories are not based on a unifying metaphor but instead on perceived trends.

We provide a characterization of each category in terms of the SMP (structures, mechanisms, and policies) model, discuss briefly the SDEs in each category, and relate them to our IFCS (individual, family, city and state) taxonomy.

4.1 Language-Centered Environments

Language-centered environments are those that support a single programming language with structures and mechanisms appropriate to that language. These environments tend to be interactive, although the standard Algol-like, system building languages traditionally have had batch interfaces. The mechanisms in the former tend to be more integrated than in the latter. In either case, there is little support for programming-in-the-large.

LCE Model =
(
{ *single structure* } ,
{ *language-specific construction tools* } ,
{ *tool-induced construction policies* }
)

These environments tend to provide a single structure. This structure may be language independent, or may be a structure that is implied by the language. Mechanisms tend to be typical construction tools such as editors, compilers, debuggers, browsers, and system generation tools. The policies tend to be typically implicit, low-level construction policies induced by the mechanisms and structures.

Examples of LCEs given by Dart *et al.* include Cedar, CommonLisp, Interlisp, Smalltalk [20], and Ada. Of these, all are interactive environments with the exception of most of the Ada environments. An interactive LCE for Ada is Rational's [3] Ada environment.

Using our scale-oriented taxonomy, these environments tend to fall into our *individual* model class, and generally into the *interpretive* model subclass.

4.2 Structure-Oriented Environments

Structure-oriented environments are those that replace the typical editor and compiler tools with a syntax-directed editor in which the underlying structure is created interactively either by language commands or incremental parsing [19]. The techniques used in SOEs have been generalized to provide structure-oriented environment generators for certain classes of languages.

SOE Model =
(
{ abstract syntax trees } ,
{ *generation-specific tools* } ,
{ enforced, incremental, fine-grained construction policies }
)

These environments provide language-independent structures, typically abstract syntax trees. What varies among the environments is the structure for representing the semantics associated with the syntax and the mechanisms for determining semantic correctness. For example, some environments use attribute grammars, others use action routines. Generally, mechanisms include editors for grammars (representing the abstract syntax), unparsers (representing the concrete syntax), and environment generators. The specific form of these mechanisms depends on the form of the semantic representation. Environmentally enforced policies are those appropriate at the statement level: prevention of syntax errors and early detection of semantic errors (specifically, type errors).

Examples of these type of environments include Mentor [13], the Synthesizer Generator, Pecan [61], and Gandalf.

As in the previous case, these environments tend to fall into our *individual* model class. However, in this case, we grouped them in the *language-oriented* model subclass.

4.3 Toolkit Environments

Toolkit environments provide a collection of language-specific as well as language-independent tools. Besides the usual construction tools, there may be tools for configuration management and version control. There are, however, very few restrictions on tool usage.

TKE Model =
(
{ file system/object-management system } ,
{ *assorted construction tools* } ,
{ *laissez faire* }
)

These environments provide a file system or object management system as the primary structure. There are a wide variety of mechanisms that range from merely a collection of tools, through a set of loosely integrated tools, to a set of tightly integrated tools. There is generally no policy enforcement with regard to how the tools are to interact with each other or how they are to be used.

Examples[17] of TKEs include UNIX/PWB, VMS VAXset CMS [11], PCTE, CAIS, DSEE, Arcadia, and PCTE+ [5].

We note that the TKE model is not identical to our *toolkit* subclass of the *individual* model. The TKE model spans a spectrum of environments that is somewhat analogous to our underlying metaphor. It includes our *toolkit* and *extended toolkit* subclasses, but does not quite extend to our *city* model.[18] Instead, the continuum is along the line of file management, object management, and process management — that is, how rich the underlying management facilities are. For example, UNIX/PWB and VMS VAXset CMS provide file management systems; PCTE and CAIS provide object management; DSEE provides activity management (a simple form of process management); and Arcadia and PCTE+ provide both object management and process management.

The spectrum provided by TKEs is richer than we indicated in our IFCS metaphor. That is partly because we considered such toolkit environments as Arcadia, PCTE and PCTE+ to be frameworks from which one might build environments to satisfy different IFCS requirements.

4.4 Method-Based Environments

Method-based environments support either a wide range of software process activities or focus on a particular specification and design method. Where LCE and SOE environments support primarily programming-in-the-small, TKE and MBE environments support primarily programming-in-the-large, with some supporting also programming-in-the-many.

MBE Model =
(
{ *textual, graphical or intermediate representations* } ,
{ *method-supportive tools* } ,
{ *method-specific policies* }
)

As there are a large variety of environments in this class, the structures provided vary with the kinds of methods supported. For example, if the methods are based on a graphical interface, then graphical structures are provided; if the methods are based on a textual interface, then textual or intermediate representations may be provided. Again, the tools are methods specific: textual or graphical editors, depending on whether they support textual or visual languages (although some methods may have both textual and visual support); analysis tools such as syntax checkers, type checkers, state machine analyzers and theorem provers; presentation tools; and in some cases, code generation tools. Where the previous environments concentrated primarily on mechanisms and structures for construction and generation (and only provided policies implicitly), in MBEs we have environments that are inherently policy based. The

17. Note that some of these examples, such as CAIS, PCTE, PCTE+ and Arcadia, are frameworks rather than environments. The characterization as environments is that of Dart *et al.*.

18. There is the potential for city models, but there are, as yet, no instances.

policies supported and enforced are those specific to the method. These policies determine the activities and their ordering, and the tools and their interactions, to a much larger degree than in the other three classes of environments.

From the large number of MBEs cited in Dart *et al.*, we mention only a few: Gist [16], Refine, Anna [45], VDM [28], PROSPECTRA [9], ISTAR, and Software Through Pictures [84].

One might roughly divide these methods-based environments into various methods-based classes: transformational, formal methods, phase-specific methods, and project management methods. The MBEs Gist and Refine form what we characterized as the *transformational* subclass of the individual model. ISTAR we characterized as a *city* model. The extent to which the remaining MBEs support individual, family or city models depends on the extent to which they support or enforce interactions among a small or large number of users. Virtually all are individual or family model environments and most are individual model environments.

5. Contributions

There are two important contributions in this paper: the SMP model and the IFCS taxonomy. We summarize each of these below and indicate where further applications are possible or where further work is needed.

5.1 The SMP Model

Our general model distinguishes intuitively those aspects of environments that are useful in evaluating SDEs: structures, mechanisms and policies. We illustrated its utility by using it to characterize two different taxonomies. However, these two exercises each focused on rather narrow views of certain kinds of environments. We believe that the model is useful beyond these particular uses.

There are a variety of perspectives that one might choose to consider the effects of either scale or historical trends. Dart *et al.* chose to consider environments from the user's perspective. In our scale-oriented taxonomy, we have chosen instead to take an architectural perspective. Other perspectives that one might include are those of tool builders, system builders, process specifiers, and project managers. We believe that our model is an appropriate tool for any of these perspectives.

In the preceding discussions we confined our attention primarily to those environments addressing the problems of implementing, testing, and maintaining software systems — that is, those environments that are concerned with the technical phases of the software development process. We believe that environments that concentrate on the full life-cycle and project management issues also could be described. Some intimations of this approach are to be found in the historical-trends taxonomy.

There are other kinds of environments than just those that support programming and the building of software systems. For example, CAD/CAM environments provide support analogous to the environments discussed here. However, the domain is quite different and the use of formal methods is much more prevalent than in current SDEs. This use of formal specifications results in sophisticated structures and mechanisms coupled with detailed domain-specific policies. It would also be interesting to survey CAD/CAM environment to determine whether there are classifications similar to our IFCS taxonomy or

whether other metaphors are needed to differentiate the problems of scale.

5.2 The IFCS Taxonomy

A substantial part of our research focuses on the problems of large-scale systems. This led us to consider why existing environments are inadequate for solving these problems. In order to characterize and compare environments with respect to their suitability for large-scale systems, we introduced our metaphorical IFCS classification scheme. We summarize our contributions as follows:

- Our IFCS taxonomy delineates four important classes of interaction among software developers with respect to the problems of scale.
- The individual and family models represent the current state of the art in SDEs. We argue informally that these two models are ill-suited for the development of large systems.
- We show that the city model introduces qualitative differences in the structures, mechanisms and policies required for very large software development projects.
- We propose a state model, which is in need of further clarification, understanding and implementation.

We conclude that there is a pressing need for research in both the technical and managerial aspects of city model environments and in the elaboration of the state model.

Acknowledgments

For their careful readings and critical comments on earlier versions of this paper: Dave Belanger, Craig Cleveland, Narain Gehani, Warren Montgomery, Peggy Quinn, and Maria Thompson. For their helpful comments and discussions: Bob Balzer, Lori Clarke, Larry Druffel, Bob Ellison, Peter Feiler, Nico Habermann, Jack Wileden, and Alex Wolf. We would also like to thank the anonymous referees, particularly the one who suggested the name ‘‘IFCS’’ for our sociological taxonomy. An extended abstract of the paper appeared in the *Proceedings of the 10th International Conference on Software Engineering*, Raffles City, Singapore, 11-15 April 1988, pp 60-68.

Kaiser is supported by National Science Foundation grants CDA-8920080, CCR-8858029 and CCR-8802741, by grants from AT&T, BNR, Citicorp, DEC, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

References

[1] *Ada Programming Language,* New York: American National Standards Institute, 1983.

[2] Evan W. Adams, Masahiro Honda, Terrence C. Miller. ‘‘Object Management in a CASE Environment’’, *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh PA, May 15-18. pp 154-163.

[3] J. E. Archer, Jr., and M. T. Devlin. ‘‘Rational’s Experience Using Ada for Very Large Systems’’, *Proceedings of the First International Conference on the Ada Programming Language Applications for the Nasa Space Station,* NASA, June 1986. pp B2.5.1-B2.5.12.

[4] Philip A. Bernstein. ‘‘Database System Support for Software Engineering’’, *Proceedings of the 9th International Conference on Software Engineering*, Monterey CA, March 1987. pp 166-178.

[5] Gerard Boudier, Fernando Gallo, Regis Minot, and Ian Thomas. ‘‘An Overview of PCTE and PCTE+,’’ *ACM SIGSOFT ’88: Third Symposium on Software Development Environments,* 28-30 November 1988, Boston MA. pp 248-257. *SIGSOFT Software Engineering Notes,* 13:5 (November 1988). *SIGPLAN Notices,* 24:2 (February 1989).

[6] Thomas E. Cheatham, Jr., Glenn H. Holloway and Judy A. Townley. ‘‘Program Refinement by Transformation’’, *Proceedings of 5th International Conference on Software Engineering*, San Diego CA, March 1981. pp 430-437.

[7] P. Chen. ‘‘The Entity-Relationship Model — Toward a Unified View of Data’’, *ACM Transactions on Database Systems*, 1:1 (March 1976). pp 9-36.

[8] Susan A. Dart, Robert J. Ellison, Peter H. Feiler, and A. Nico Habermann. ‘‘Software Development Environments’’, *Computer,* 20:11 (November 1987). pp 18-28.

[9] P. De la Cruz, B. Krieg-Bruekner, and A. Perez Riesco. ‘‘From Algebraic Specifications to Correct Ada Programs: the Esprit Project PROSPECTRA,’’ *Ada: The Design Choice. Proceedings of the Ada-Europe International Conference, Madrid, 13-15 June 1989,* Angel Alverez, editor. Cambridge: Cambridge University Press, 1989.

[10] Frank DeRemer and Hans H. Kron. ‘‘Programming-in-the-Large Versus Programming-in-the-Small’’, *IEEE Transactions on Software Engineering*, SE-2:2 (June 1976). pp 80-86.

[11] Digital Equipment Corp. *User’s Introduction to VAX DEC/CMS.* Maynard, Mass: Digital Equipment Corp., 1984.

[12] Keith D. Cooper, Ken Kennedy, and Linda Torczon. ‘‘The Impact of Interprocedure Analysis and Optimization in the R^n Environment,’’ *ACM Transactions on Programming Languages and Systems,* 8:4 (October 1986). pp 491-523.

[13] V. Donzeau-Gouge, G. Kahn, B. Lang, and B. Melese. ‘‘Documents Structure and Modularity in Mentor’’. *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* 23-25 April 1984, Pittsburgh PA. pp 141-148. *SIGSOFT Software Engineering Notes,* 9:3 (May 1984). *SIGPLAN Notices,* 19:5 (May 1984).

[14] Mark Dowson. ‘‘Integrated Project Support with IStar’’, *IEEE Software,* 4:6 (November 1987). pp 6-15.

[15] Peter H. Feiler and Raul Medina-Mora. ‘‘An Incremental Programming Environment’’, *IEEE Transactions on Software Engineering*, SE-7:5 (September 1981). pp 472-482.

[16] S. F. Fickas. ‘‘Automating the Transformational Development of Software,’’ *IEEE Transactions on Software Engineering,’’* SE-11:11 (November 1985). pp 1268-1277.

[17] S.I. Feldman. ‘‘Make — A Program for Maintaining Computer Programs’’, *Software — Practice & Experience*, 9:4 (April 1979). pp 255-265.

[18] David Garlan. ‘‘Views for Tools in Integrated Environments’’, *Advanced Programming Environments*, Reidar Conradi, Tor M. Didriksen and Dag H. Wanvik, editors. Lecture Notes in Computer Science, 244. Berlin: Springer-Verlag, 1986. pp 314-343.

[19] Carlo Ghezzi and Dino Mandrioli. ‘‘Augmenting Parsers to Support Incrementality’’, *Journal of the ACM*, 27:3 (July 1980). pp 564-579.

[20] Adele Goldberg. *Smalltalk-80. The Interactive Programming Environment.* Reading: Addison-Wesley, 1984.

[21] A.N. Habermann and D. Notkin. ‘‘Gandalf: Software Development Environments’’, *IEEE Transactions on Software Engineering*, SE-12:12 (December 1986). pp 1117-1127.

[22] A. Nico Habermann and Dewayne E. Perry. ‘‘System Composition and Version Control for Ada’’, *Software Engineering Environments*, H. Huenke, editor. New York: North-Holland, 1981. pp 331-343.

[23] Mark F. Hornick and Stanley B. Zdonik. ‘‘A Shared, Segmented Memory System for an Object-Oriented Database’’, *ACM Transactions on Office Automation Systems*, 5:1 (January 1987), pp 70-95.

[24] Susan Horwitz and Tim Teitelbaum. ‘‘Generating Editing Environments Based on Relations and Attributes’’, *ACM Transactions on Programming Languages and Systems*, 8:4 (October 1986). pp 577-608.

[25] Susan Horwitz, Jan Prins, and Thomas Reps. ‘‘Integrating Noninterfering Versions of Programs,’’ *ACM Transactions on Programming Languages and Systems*, 11:3 (July 1989). pp 345-387.

[26] William E. Howden. ‘‘Contemporary Software Development Environments’’, *Communications of the ACM*, 25:5 (May 1982). pp 318-329.

[27] Watts S. Humphrey. ‘‘Characterizing the Software Process’’, *IEEE Software*, 5:2 (March 1988). pp 73-79.

[28] Cliff B. Jones. *Systematic Software Development Using VDM.* Englewood Cliffs, NJ: Prentice-Hall International, 1986.

[29] Stephen C. Johnson. ‘‘Yacc: Yet Another Compiler-Compiler,’’ *UNIX Programmer’s Manual*, Seventh Edition, Volume 2B, January 1979.

[30] William Joy. ‘‘An Introduction to the C shell’’, *UNIX User’s Manual Supplementary Documents*, 1986. pp USD:4.

[31] Gail E. Kaiser and Peter H. Feiler. ‘‘Intelligent Assistance without Artificial Intelligence’’, *Thirty-Second IEEE Computer Society International Conference*, February 1987, San Francisco, CA. pp 236-241.

[32] Gail E. Kaiser, Peter H. Feiler and Steven S. Popovich. ‘‘Intelligent Assistance for Software Development and Maintenance’’, *IEEE Software*, 5:3 (May 1988). pp 40-49.

[33] Gail E. Kaiser and A. Nico Habermann. ‘‘An Environment for System Version Control’’, *Twenty-Sixth IEEE Computer Society International Conference*, San Francisco CA, February 1983. pp 415-420.

[34] Gail E. Kaiser and Simon M. Kaplan. ‘‘Reliability in Distributed Programming Environments,’’ *6th Symposium on Reliability in Distributed Software and Database Systems*, Williamsburg VA, March 1987, IEEE Computer Society. pp 45-55.

[35] Gail E. Kaiser, Simon M. Kaplan and Josephine Micallef. ‘‘Multiuser, Distributed Language-Based Environments’’, *IEEE Software*, 4:6 (November 1987). pp 58-67.

[36] Gail E. Kaiser and Dewayne E. Perry. ‘‘Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution’’, *Conference on Software Maintenance-1987*,

Austin TX, September 1987. pp 108-114.

[37] Gail E. Kaiser, Dewayne E. Perry and William M. Schell. ‘‘Infuse: Fusing Integration Test Management with Change Management’’. *Proceedings of COMPSAC ’89 — The 13th Annual International Computer Software and Applications Conference* 18-22 September 1989, Orlando FL. pp 552-558.

[38] Brian W. Kernighan and John R. Mashey. ‘‘The UNIX Programming Environment’’, *IEEE Computer*, 12:4 (April 1981). pp 25-34.

[39] David Alex Lamb. ‘‘IDL: Sharing Intermediate Representations’’, *ACM Transactions on Programming Languages and Systems*, 9:3 (July 1987). pp 297-318.

[40] David B. Leblang and Robert P. Chase, Jr.. ‘‘Parallel Software Configuration Management in a Network Environment’’, *IEEE Software*, 4:6 (November 1987). pp 28-35.

[41] David B. Leblang and Robert P. Chase, Jr.. ‘‘Computer-Aided Software Engineering in a Distributed Workstation Environment’’, *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh PA, April 1984. pp 104-112. *SIGSOFT Software Engineering Notes*, 9:3 (May 1984). *SIGPLAN Notices*, 19:5 (May 1984).

[42] Peter Lee, Frank Pfenning, Gene Rollins, and William Scherlis. ‘‘The Ergo Support System: An Integrated Set Tools for Prototyping Integrated Environments’’. *ACM SIGSOFT ’88: Third Symposium on Software Development Environments*, 28-30 November 1988, Boston MA. pp 25-34. *SIGSOFT Software Engineering Notes*, 13:5 (November 1988). *SIGPLAN Notices*, 24:2 (February 1989).

[43] M. M. Lehman and W. M. Turski. ‘‘Essential Properties of IPSEs’’, *Software Engineering Notes* 12:1 (January 1987). pp 52-55.

[44] Mark A. Linton. ‘‘Implementing Relational Views of Programs’’, *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh PA, April 1984. pp 132-140. *SIGSOFT Software Engineering Notes*, 9:3 (May 1984). *SIGPLAN Notices*, 19:5 (May 1984).

[45] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Bruekner, and Olaf Owe. *Anna. A Language for Annotating Ada Programs. Reference Manual.* Lecture Notes in Computer Science: 260. Berlin: Springer-Verlag, 1987.

[46] Yoelle S. Maarek and Gail E. Kaiser. ‘‘Change Management for Very Large Software Systems’’, *Seventh Annual International Phoenix Conference on Computers and Communications*, March 1988, Scottsdale AZ. pp 280-285.

[47] Naftaly Minsky and David Rozenshtein. ‘‘Software Development Environment for Law-Governed Systems’’. *ACM SIGSOFT ’88: Third Symposium on Software Development Environments*, 28-30 November 1988, Boston MA. pp 65-75. *SIGSOFT Software Engineering Notes*, 13:5 (November 1988). *SIGPLAN Notices*, 24:2 (February 1989).

[48] Robert Munck, Patricia Obendorf, Erhard Ploedereder, Richard Thall. ‘‘An Overview of DOD-STD-1838A (proposed), The Common APSE Interface Set, Revision A’’. *ACM SIGSOFT ’88: Third Symposium on Software Development Environments*, 28-30 November 1988, Boston MA. pp 235-248. *SIGSOFT Software Engineering Notes*, 13:5 (November 1988). *SIGPLAN Notices*, 24:2 (February 1989).

[49] John R. Nestor. ‘‘Toward a Persistent Object Base’’, *Advanced Programming Environments*, Reidar Conradi, Tor M. Didriksen and Dag H. Wanvik, editors. Lecture Notes in Computer Science, 244. Berlin: Springer-Verlag, 1986. pp 372-394.

[50] David Notkin. ‘‘The GANDALF Project’’, *The Journal of Systems and Software*, 5:2 (May 1985). pp 91-105.

[51] Elliott I. Organick. *The Multics System: An Examination of Its Structure*. Cambridge MA: The MIT Press, 1972.

[52] Leon Osterweil. ‘‘Software Processes Are Software Too’’, *Proceedings of the 9th International Conference on Software Engineering*, Monterey CA, March 1987. pp 2-13.

[53] L.J. Osterweil. ‘‘Toolpack — An Experimental Software Development Environment Research Project’’, *IEEE Transactions on Software Engineering*, SE-9:6 (November 1983). pp 673-685.

[54] Maria H. Penedo. ‘‘Prototyping a Project Master Database for Software Engineering Environments’’, *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Palo Alto CA, December 1986. pp 1-11. *SIGPLAN Notices*, 22:1 (January 1987).

[55] Maria Penedo, Erhard Ploedereder, and Ian M. Thomas. ‘‘Object Management Issues for Software Engineering Environments (Workshop Report)’’ *ACM SIGSOFT ’88: Third Symposium on Software Development Environments*, 28-30 November 1988, Boston MA. *SIGSOFT Software Engineering Notes*, 13:5 (November 1988). *SIGPLAN Notices*, 24:2 (February 1989). pp 226-234.

[56] Dewayne E. Perry. ‘‘Software Interconnection Models’’, *Proceedings of the 9th International Conference on Software Engineering*, Monterey CA, March 1987. pp 61-69.

[57] Dewayne E. Perry. ‘‘The Inscape Environment’’, *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh PA, May 15-18. pp 2-12.

[58] Dewayne E. Perry, editor. *Proceedings of the Fifth International Software Process Workshop: Experience with Process Models*, October 1989, Kennebunkport, Maine. IEEE Computer Society Press.

[59] Dewayne E. Perry and Gail E. Kaiser. ‘‘Infuse: A Tool for Automatically Managing and Coordinating Source Changes in Large Systems’’, *ACM Fifteenth Annual Computer Science Conference*, St. Louis MO, February 1987. pp 292-299.

[60] Dewayne E. Perry and Gail E. Kaiser. ‘‘Adequate Testing and Object Oriented Programming’’ *Journal of Object-Oriented Programming*, 2:5 (January 1990). pp 13-19.

[61] Steven P. Reiss. ‘‘Graphical Program Development with PECAN Program Development System,’’ *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh PA, April 1984. pp 30-41. *SIGSOFT Software Engineering Notes*, 9:3 (May 1984). *SIGPLAN Notices*, 19:5 (May 1984).

[62] Steven P. Reiss. ‘‘Working in the Garden Environment for Conceptual Programming’’, *IEEE Software*, 4:6 (November 1987). pp 16-27.

[63] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator A System for Constructing Language-Based Editors*. Texts and Monographs in Computer Science. New York NY: Springer-Verlag, 1989.

[64] Thomas Reps. *Generating Language-Based Environments*. Cambridge MA: The MIT Press, 1984.

[65] Charles Rich and Richard C. Waters. ‘‘Automatic Programming: Myths and Prospects’’ *Computer*, 21:8 (August 1988), pp 40-51.

[66] Charles Rich and Richard C. Waters. ‘‘The Programmer’s Apprentice: A Research Overview’’, *Computer*, 21:11 (November 1988), pp 10-25.

[67] M. J. Rochkind. ‘‘The Source Code Control System’’, *IEEE Transactions on Software Engineering*, SE-1:4, (December 1975). pp 364-370.

[68] Lawrence A. Rowe. ‘‘Report on the 1989 Software CAD Databases Workshop,’’ *Information Processing 89. Proceedings of the IFIP 11th World Computer Congress.* San Francisco, CA. August 1989. Amsterdam: North Holland, 1989. pp 719-725.

[69] B. R. Rowland, R. E. Anderson, and P. S. McCabe. ‘‘The 3B20D Processor & DMERT Operating System: Software Development System’’, *The Bell System Technical Journal*, 62:1 part 2 (January 1983). pp 275-290.

[70] Barbara G. Ryder and Marvin C. Paull. ‘‘Incremental Data-Flow Analysis,’’ *ACM Transactions on Programming Languages and Systems*, 10:1 (January 1988). pp 1-50.

[71] R. W. Schwanke, E. S. Cohen, R. Gluecker, W. M. Hasling, D. A. Soni, and M. E. Wagner. ‘‘Configuration Management in BiiN SMS’’. *Proceedings of the 11th International Conference on Software Engineering,’’* 15-18 May 1989, Pittsburgh PA. pp 383-393.

[72] Izhar Shy, Richard Taylor, and Leon Osterweil. ‘‘A Metaphor and a Conceptual Framework for Software Development Environments’’, in *Software Engineering Environments: Proceedings of the International Workshop on Environments, Chinon, France, September 1989*, F. Long, editor. Berlin: Springer-Verlag, 1990. pp 77-97.

[73] Douglas R. Smith, Gordon B. Kotik and Stephen J. Westfold. ‘‘Research on Knowledge-Based Software Environments at Kestrel Institute’’, *IEEE Transactions on Software Engineering*, SE-11:11 (November 1985). pp 1278-1295.

[74] Richard Snodgrass. *The Interface Description Language* Rockville MD: Computer Science Press, 1989.

[75] Guy L. Steele, Jr. *Common Lisp — The Language.* Burlington, Mass: Digital Press, 1984.

[76] Vic Stenning. ‘‘An Introduction to ISTAR’’, Chapter 1 in *Software Engineering Environments*, Ian Sommerville, editor. IEE Computing Series 7. London: Peter Peregrinus Ltd., 1986. pp 1-22.

[77] 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 (October 1986) pp 419-490.

[78] R. N. Taylor, R. W. Selby, M. Young, F. C. Belz, L. A. Clarke, J. C. Wileden, L. Osterweil, A. L. Wolf. ‘‘Foundations for the Arcadia Environment Architecture’’. *ACM SIGSOFT ’88: Third Symposium on Software Development Environments*, 28-30 November 1988, Boston MA. pp 1-13. *SIGSOFT Software Engineering Notes*, 13:5 (November 1988). *SIGPLAN Notices*, 24:2 (February 1989).

[79] Warren Teitelman and Larry Masinter. ‘‘The Interlisp Programming Environment’’, *IEEE Computer*, 14:4 (April 1981). pp 25-34.

[80] Ian Thomas. ‘‘PCTE Interfaces: Supporting Tools in Software-Engineering Environments,’’ *IEEE Software*, 6:6 (November 1989). pp 15-23.

[81] Walter F. Tichy. ‘‘RCS — A System for Version Control’’, *Software — Practice and Experience*, 15:7 (July 1985) pp 637-654.

[82] Colin Tully, editor. ‘‘Representing and Enacting the Software Process’’, *Proceedings of the 4th International Software Process Workshop* 11-13 May 1988, Moretonhampstead, Devon, UK. *ACM SIGSOFT Software Engineering Notes* 14:4 (June 1989).

[83] J. Walpole, G. S. Blair, J. Malik and J. R. Nicol. ‘‘A Unifying Model for Consistent Distributed Software Development Environments’’. *ACM SIGSOFT ’88: Third Symposium on Software Development Environments*, 28-30 November 1988, Boston MA. pp 183-190. *SIGSOFT Software Engineering Notes*, 13:5 (November 1988). *SIGPLAN Notices*, 24:2 (February 1989).

[84] Anthony I. Wasserman and Peter A. Pircher. *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, 9-11 December 1986, Palo Alto CA. pp 131-142. *SIGPLAN Notices*, 22:1 (January 1987).

[85] Elaine J. Weyuker. ‘‘The Evaluation of Program-Based Software Test Data Adequacy Criteria’’, *Communications of the ACM* 31:6 (June 1988). pp 668-675.

[86] David S. Wile and Dennis G. Allard. ‘‘Worlds: an Organizing Structure for Object-Bases’’, *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, 9-11 December 1986, Palo Alto CA. pp 130-142. *SIGPLAN Notices*, 22:1 (January 1987).

[87] Jack C. Wileden, Lori A. Clarke, and Alexander L. Wolf. ‘‘A Comparative Evaluation of Object Definition Techniques for Large Prototype Systems’’ *ACM Transactions on Programming Languages and Systems*, 12:4 (October 1990).

[88] Alexander L. Wolf, Jack C. Wileden, Charles D. Fisher, and Peri L. Tarr. ‘‘PGraphite: An Experiment in Persistent Typed Object Management’’. *ACM SIGSOFT ’88: Third Symposium on Software Development Environments*, 28-30 November 1988, Boston MA. pp 130-142. *SIGSOFT Software Engineering Notes*, 13:5 (November 1988). *SIGPLAN Notices*, 24:2 (February 1989).

[89] William A. Wulf, Roy Levin, and Samuel P. Harbison. *HYDRA/C.mmp: An Experimental Computer System* New York: McGraw-Hill, 1981. pp 35-36.