delim $$ 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 — 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 a general model of software development environments (SDEs). Our model consists of three components: policies, mechanisms and structures. Once we have defined this general model of software development environments, there are various points of view from which we might classify environments. We might, for example, classify the SDEs according to their coverage of the software life cycle; or classify them according to the kinds of tools that they provide, contrasting those that provide a kernel set with those that provide an extended set; etc. 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 software development 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 [5] 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*. For example, Howden [15] considers SDEs for medium and large systems only from the standpoint of capitalization and richness of the toolset. Thus, the main focus of this paper — and, indeed, of our research — is the problem of scale. We introduce a classification of SDEs in terms of a sociological metaphor that emphasizes this problem of scale and provides insight into the environmental requirements for projects of different sizes. This metaphor suggests four classes of models: individual, family, city and state. The individual and family classes are the current state of the art but are inadequate for building very large systems. We argue that the city model is adequate but that very 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. In section 2, we present our model of software development environments, discuss the individual components and their interrelationships, and illustrate various distinctions that we make with environments from the literature. We do not describe any of these environments, since this article is not intended to be a survey, but accessible references are provided. In section 3, we classify SDEs into the four classes suggested by our metaphor, characterize these classes, present a basic model for each class, and categorize a wide variety of existing environments into the individual, family and city classes (we know of no examples of the state class). Finally, in section 4 we summarize the contributions of our model and classification scheme. We confine our discussion in the sections below primarily to those environments concerned with the problems of implementing, testing, and maintaining software systems — that is, those environments that are concerned about 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 with this model and categorized according to our classification scheme presented in section 3. Our general model of software development environments consists of three interrelated components: policies, mechanisms and structures. General SDE Model = ( { Policies }, { Mechanisms }, { Structures } ) Policies are the rules, guidelines and strategies imposed on the programmer* We use the term *programmer* in a generic sense to include any user of the environment and use the two terms interchangeably. by the environment; mechanisms are the visible and underlying tools and tool fragments; and structures are the underlying objects and object aggregates on which mechanisms operate. 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. Policies are the requirements imposed on the user of the environment during the software development process. These rules and strategies are often hard-coded into the environment by the mechanisms and structures. 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 always compiling the modules before linking them. A different strategy is possible for execution preparation tools that provide dynamic linking and, hence, a different policy: for example, Multics' segmentation scheme [34] allows externally referenced names to be resolved at run-time. In most cases, the design of the tools and the supporting structures define or impose the policies. But policies need not be hard-wired — that is, implicit in the mechanisms and structures. A few architectures allow the explicit specification of policies. For example, Osterweil's process programming [35, 56] provides the ability to program the desired policies with respect to the various mechanisms and structures available; Darwin's law-governed systems [30] consist of declaratively defined rules restricting the interactions of programmers and tools. These approaches provide a way of explicitly imposing policies on the programmers independently of the mechanisms and structures. An orthogonal distinction is whether the policies are supported or enforced. If a policy is *supported*, then the mechanisms and structures provide a means of satisfying that policy. For example, suppose that 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 tools and structures to build systems in other ways as well. If a policy is *enforced*, then not only is it supported, but it is not possible to do it any other way within the environment. We call this *direct enforcement* when the environment explicitly forces the programmer to follow the policy. A slightly different kind of enforcement is that of *indirect enforcement*: policy decisions are made outside the environment either by management or by convention but once made they are supported but not enforced by the environment. For example, management decides that all systems are to be generated only from modules resident within the Source Code Control System (SCCS) [47]. The environment supports configuration management with SCCS; however, it is the management decision that forces the programmers to control their modules within SCCS. 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 second as *higher-order* policies. For example, 'all projects will be done in Ada' is a higher-order policy. 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; others may be hidden from the user and function as lower-level support mechanisms. For example, UNIX [23] provides tools directly to the user, while Smile [17] hides these tools beneath a facade that provides the programmer with higher-level mechanisms (that in turn invoke individual UNIX tools). Policies are encoded in mechanisms in one of two ways: either *explicitly* by policy makers for a particular project, or *implicitly* by the toolsmiths for a particular environment. In the first case, mechanisms such as shell scripts [16], CLF's [61] or Marvel's rules [18] enable the policy maker to define explicitly, as process programs, the policies to be supported by the system. Whether these policies can also be enforced depends on how well the mechanisms restrict the programmer in using the environment. In the second case, the examples from the preceding section (illustrating hard-wired policies) exemplify implicit encoding. In most SDEs, policies are implicitly encoded in the mechanisms. There are good historical reasons for this situation: we must work out particular instances before we can generalize. Particular mechanisms and structures must first be built that implicitly encode policies 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. Structures are those objects or object aggregates on which the mechanisms operate. In the simplest (and chronologically, earliest) incarnation, the basic structures — the objects with which we build systems — are files (as in UNIX, for example). The trend, however, is towards more complex and comprehensive objects as the basic structures. One reason for complex basic structures is found in integrated environments, particularly those centered around a syntax-directed editor [9, 44]. These SDEs share a complex internal representation such as an abstract syntax tree [7] or a graph structure [24, 53, 62] 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 support well the mechanisms and their intended policies. Garlan's tool views [11] provide a partial* 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 that one adds the view needed by that tool to the original set and generates the newly integrated structure. 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 [2, 32]. The basic structure currently generating a large amount of interest is the *objectbase* [14, 38, 48] — it is hoped that this approach will solve the deficiencies of files and databases. These basic structures are the foundation for building more complex and more comprehensive higher-order structures. For example, Inscape [39, 40] 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 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) [37] provides an entity-relationship-attribute model [4] to represent, for example, problem reporting, evaluation and tracking processes. CMS's Modification Request Tracking System [49] builds a structure that is intertwined with SCCS's configuration 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. Finally, Apollo's Domain Software Engineering Environment (DSEE) [26] provides a comprehensive set of structures for coordinating the building and evolving of software systems; these structures support, for example, configuration control, planning and programmer interactions. 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 [33] 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 [21] and Cosmos's versions, configurations and histories [60] 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. 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. 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: move right 0.25i A: box ht .01i wid 5.5i "•" at A.w + (.5i,0) "Individual" at A.w + (.5i, .25i) "•" at A.w + (2i,0) "Family" at A.w + (2i, .25i) "•" at A.w + (3.5i,0) "City" at A.w + (3.5i, .25i) "•" at A.w + (5i,0) "State" at A.w + (5i, .25i) 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; 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 city models, and nothing is known about SDEs that support the state model, we make as few distinctions as possible. The purpose of this paper is to draw attention to these two representative classes — that is, 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 in terms of scale from the others. 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. Historically, 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*. 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 nano-management issues. Individual Model = ( { *tool-induced policies** } , { *implementation tools* } , { *single structure* } ) These We use *italics* for general descriptions of the components and normal typeface for specific components. 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 [60]; language-oriented environments by the Cornell Program Synthesizer [44]; and the transformational environments by Refine [52]. The toolkit environments are, historically, the archetype of the individual model. 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 [10]). 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. The notation ‘‘. . .’’ at the end of the list indicates that additional facilities may be available. Toolkit Model = ( { *tools-induced policies*, *script-encoded policies*, ...** } , { editors, compilers, linker/loaders, debuggers, ... } , { file system } ) 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 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 noted for their extreme flexibility and 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 = ( { *virtually no restrictive policies* } , { interpreter, *underlying support tools* } , { intermediate representation } ) 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 [42], or generated from a formal specification, such as by the Synthesizer Generator [43]. Language-Oriented Model = ( { error prevention, early error detection and notification, ... }, { editor, compiler, debugger, ... } , { decorated syntax tree } ) Transformational environments typically support a wide-spectrum language (such as V [52]) that includes a range of object and control structures from abstract to concrete. Programs are initially written in a 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 [27] and PDS [3]). Programmer's apprentices, such as KBEmacs [45, 46], are a variation of this policy in that the programmer can switch between the transformational approach and interpretive approach at any time. Transformational Model = ( { transformational construction, ... }, { interpreter, transformational engine, ... }, { intermediate representation/decorated syntax tree, ... } ) 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. 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). 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. The notation ''. . .'' at the beginning of the list indicates that we include the policies, mechanisms and structures from the previous level. Family Model = ( { ...*, *coordination policies* } , { ..., *coordination mechanisms* } , { ..., *special-purpose databases* } ) 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, 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** 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. 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 [58]; the integrated environments by Smile; the distributed environments by Cedar [55]; 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 [6]). 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) [31]. Extended Toolkit Model = ( { ..., support version/configuration control } , { ..., version/configuration management } , { ..., compressed versions, version trees } ) 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 Gandalf Prototype (GP) [12], to procedures in Smile, to entire modules in Toolpack [36] and $R sup n$ [6]. This model's structure is typically a special-purpose database, perhaps most prominent in SMS's [50] use of an object-oriented file system, although generic architectures such as CLF 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 [13, 19]. Integrated Model = ( { ..., enforced version control, enforced consistency } , { ..., version description languages, consistency checking tools } , { ..., special-purpose database } ) 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 [20] is a multi-user, language-oriented environment that depends on a distributed, static semantic analysis algorithm to guarantee consistency among module interfaces; NSE provides most of facilities of DSEE [25], and adds multi-user integration environments [1] that require extensions to Sun's network file system to support loop-back mount structures. Distributed Model = ( { ... } , { ..., *network mechanisms* } , { ..., distributed *objects* } ) The project management model is orthogonal to the progression from the extended toolkit model to the distributed model. 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. Project Management Model = ( { ..., support activity coordination } , { ..., *activity coordination mechanisms* } . { ..., *activity coordination structures* } ) 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. 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 the complexity of interactions that leads us to introduce the city model. City Model = ( { ..., *cooperation policies* } , { ..., *cooperation mechanisms* }, { ..., *structures for cooperation* } ) 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 of the people and vehicles. The alteration of 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 the family model is currently being used where we need a city model, 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, we generally apply a set of methodologies and management techniques that 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* Infuse is 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. [41] and ISTAR [8]. 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 policies, mechanisms and structures 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 policies enforce cooperation among programmers in several ways [21]. Infuse automatically partitions [29] 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 measure is used as an approximation to the oracle that tells which modules will be affected by which changes). This partitioning forms the basis for enforcing cooperation: each experimental database 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. 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 for it. 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 at the top of the hierarchy. Thus the rules for interaction are encoded in the mechanisms, with the hierarchy providing the supporting structure.* We are also investigating the utility of this structure for cooperation in integration testing [22]. 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. Infuse Model = ( { ..., enforced and voluntary cooperation } , { ..., automatic partitioning, local consistency analysis, database deposit, local integration testing, ... } , { ..., hierarchy of experimental databases } ) 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) [28] 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 "plugging in" various kinds of workbenches. The contract model dictates the allowable interactions among component programmers [54]. 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 structure can change in the same ways that the products themselves can change. A contract database provides the underlying structure for this environment. 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 model } , { ..., *contract support tools* } , { ..., contract data base } ) Pursuing our 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.* Shy, Taylor and Osterweil [51] propose a corporation model that provides a comprehensive framework in which to build not only a city model SDE, but a state model SDE as well. There are, we think, intimations of this model in the following: the Department of Defense standardizing on a small number of languages, such as Ada and Common Lisp, for all its projects; a company trying to establish a uniform development environment such as UNIX and C/C++ for all its projects; a project attempting to develop a new standard tool, such as X windows; a company establishing a common methodology and 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. State Model = ( { ..., *commonality policies* } , { ..., *supporting mechanisms* } , { ..., *supporting structures* } ) 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* See various position papers and discussions in the 4th International Software Process Workshop [59]. suggests the following general description. Provide a generic model with its attendant tools and supporting structures for software development to be used throughout a particular company. Instantiate the model for each project, tailoring each instance dynamically to the particular needs of the individual project. Manage the differences between the various instances to support movement between projects. Thus, while little is known about the state model, it appears to be a useful and fruitful area for investigation. Ideally, scaling up from one class to the next would be a matter of adding structures and mechanisms 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 policies, mechanisms and structures. 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. 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 policies, mechanisms, and structures required for the higher levels of environments are, in some sense, more basic and must be designed in rather than just added on. 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 to large-scale systems, we developed our model of software development environments and introduced our metaphorical classification scheme. We summarize our contributions as follows: Our general model distinguishes precisely those aspects of an environment that are useful in evaluating SDEs: policies, mechanisms and structures. Our 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 explain why these two models are ill-suited for the development of large systems. We show that the city model introduces the qualitative differences in the policies, structures and mechanisms 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 delineation of the state model. 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. An earlier version 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. Professor Kaiser is supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from AT&T, Citibank, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

Evan W. Adams, Masahiro Honda, Terrence C. Miller. ‘‘Object Management in a CASE Environment’’, Pittsburgh PA, May 15-18. pp 154-163. Philip A. Bernstein. ‘‘Database System Support for Software Engineering’’, Monterey CA, March 1987. pp 166-178. Thomas E. Cheatham, Jr., Glenn H. Holloway and Judy A. Townley. ‘‘Program Refinement by Transformation’’, San Diego CA, March 1981. pp 430-437. P. Chen. ‘‘The Entity-Relationship Model — Toward a Unified View of Data’’, 1:1 (March 1976). pp 9-36. Frank DeRemer and Hans H. Kron. ‘‘Programming-in-the-Large Versus Programming-in-the-Small’’, SE-2:2 (June 1976). pp 80-86. T.A. Dolotta, R.C. Haight and J.R. Mashey. ‘‘The Programmer’s Workbench’’, 57:6-2 (July-August 1978). pp 2177-2200. V. Donzeau-Gouge, G. Kahn, B. Lang, and B. Melese. ‘‘Documents Structure and Modularity in Mentor’’. 23-25 April 1984, Pittsburgh PA. *SIGSOFT Software Engineering Notes*, 9:3 (May 1984). *SIGPlan Notices*, 19:5 (May 1984). pp 141-148. Mark Dowson. ‘‘Integrated Project Support with IStar’’, November 1987. pp 6-15. Peter H. Feiler and Raul Medina-Mora. ‘‘An Incremental Programming Environment’’, SE-7:5 (September 1981). pp 472-482. S.I. Feldman. ‘‘Make — A Program for Maintaining Computer Programs’’, 9:4 (April 1979). pp 255-265. David Garlan. ‘‘Views for Tools in Integrated Environments’’, Reidar Conradi, Tor M. Didriksen and Dag H. Wanvik, editors. Lecture Notes in Computer Science, 244. Berlin: Springer-Verlag, 1986. pp 314-343. A.N. Habermann and D. Notkin. ‘‘Gandalf: Software Development Environments’’, SE-12:12 (December 1986). pp 1117-1127. A. Nico Habermann and Dewayne E. Perry. ‘‘System Composition and Version Control for Ada’’, H. Huenke, editor. New York: North-Holland, 1981. pp 331-343. Mark F. Hornick and Stanley B. Zdonik. ‘‘A Shared, Segmented Memory System for an Object-Oriented Database’’, 5:1 (January 1987), pp 70-95. William E. Howden. ‘‘Contemporary Software Development Environments’’, 25:5 (May 1982). pp 318-329. William Joy. ‘‘An Introduction to the C shell’’, 1986. pp USD:4. Gail E. Kaiser and Peter H. Feiler. ‘‘Intelligent Assistance without Artificial Intelligence’’, February 1987, San Francisco, CA. pp 236-241. Gail E. Kaiser, Peter H. Feiler and Steven S. Popovich. ‘‘Intelligent Assistance for Software Development and Maintenance’’, May 1988. pp 40-49. Gail E. Kaiser and A. Nico Habermann. ‘‘An Environment for System Version Control’’, San Francisco CA, February 1983. pp 415-420. Gail E. Kaiser, Simon M. Kaplan and Josephine Micallef. ‘‘Multiuser, Distributed Language-Based Environments’’, November 1987. pp 58-67. Gail E. Kaiser and Dewayne E. Perry. ‘‘Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution’’, Austin TX, September 1987. pp 108-114. Gail E. Kaiser, Dewayne E. Perry and William M. Schell. ‘‘Infuse: Fusing Integration Test Management with Change Management’’. 18-22 September 1989, Orlando FL. Brian W. Kernighan and John R. Mashey. ‘‘The UNIX Programming Environment’’, 12:4 (April 1981). pp 25-34. David Alex Lamb. ‘‘IDL: Sharing Intermediate Representations’’, 9:3 (July 1987). pp 297-318. David B. Leblang and Robert P. Chase, Jr.. ‘‘Parallel Software Configuration Management in a Network Environment’’, November 1987. pp 28-35. David B. Leblang and Robert P. Chase, Jr.. ‘‘Computer-Aided Software Engineering in a Distributed Workstation Environment’’, Pittsburgh PA, April 1984. *SIGSOFT Software Engineering Notes*, 9:3 (May 1984). *SIGPLAN Notices*, 19:5 (May 1984). pp 104-112. Peter Lee, Frank Pfenning, Gene Rollins, and William Scherlis. ‘‘The Ergo Support System: An Integrated Set Tools for Prototyping Integrated Environments’’. 28-30 November 1988, Boston MA. *SIGSOFT Software Engineering Notes*, 13:5 (November 1988). *SIGPlan Notices*, 24:2 (February 1989). pp 25-34. M. M. Lehman and W. M. Turski. ‘‘Essential Properties of IPSEs’’, 12:1 (January 1987). pp 52-55. Yoelle S. Maarek and Gail E. Kaiser. ‘‘Change Management for Very Large Software Systems’’, March 1988, Scottsdale AZ. pp 280-285. Naftaly Minsky and David Rozenshtein. ‘‘Software Development Environment for Law-Governed Systems’’. 28-30 November 1988, Boston MA. *SIGSOFT Software Engineering Notes*, 13:5 (November 1988). *SIGPlan Notices*, 24:2 (February 1989). pp 65-75. Robert Munck, Patricia Obendorf, Erhard Ploedereder, Richard Thall. ‘‘An Overview of DOD-STD-1838A (proposed), The Common APSE Interface Set, Revision A’’. 28-30 November 1988, Boston MA. *SIGSOFT Software Engineering Notes*, 13:5 (November 1988). *SIGPlan Notices*, 24:2 (February 1989). pp 235-248. John R. Nestor. ‘‘Toward a Persistent Object Base’’, Reidar Conradi, Tor M. Didriksen and Dag H. Wanvik, editors. Lecture Notes in Computer Science, 244. Berlin: Springer-Verlag, 1986. pp 372-394. David Notkin. ‘‘The GANDALF Project’’, 5:2 (May 1985). pp 91-105. Elliott I. Organick. Cambridge MA: The MIT Press, 1972. Leon Osterweil. ‘‘Software Processes Are Software Too’’, Monterey CA, March 1987. pp 2-13. L.J. Osterweil. ‘‘Toolpack — An Experimental Software Development Environment Research Project’’, SE-9:6 (November 1983). pp 673-685. Maria H. Penedo. ‘‘Prototyping a Project Master Database for Software Engineering Environments’’, Palo Alto CA, December 1986. *SIGPLAN Notices*, 22:1 (January 1987). pp 1-11. Maria Penedo, Erhard Ploedereder, Ian M. Thomas. ‘‘Object Management Issues for Software Engineering Environments (Workshop Report)’’ 28-30 November 1988, Boston MA. *SIGSOFT Software Engineering Notes,* 13:5 (November 1988). *SIGPlan Notices,* 24:2 (February 1989). pp 226-234. Dewayne E. Perry. ‘‘Software Interconnection Models’’, Monterey CA, March 1987. pp 61-69. Dewayne E. Perry. ‘‘The Inscape Environment’’, Pittsburgh PA, May 15-18. pp 2-12. Dewayne E. Perry and Gail E. Kaiser. ‘‘Infuse: A Tool for Automatically Managing and Coordinating Source Changes in Large Systems’’, St. Louis MO, February 1987. pp 292-299. Steven P. Reiss. ‘‘Working in the Garden Environment for Conceptual Programming’’, November 1987. pp 16-27. Thomas W. Reps and Tim Teitelbaum. Texts and Monographs in Computer Science. New York NY: Springer-Verlag, 1989. Thomas Reps. Cambridge MA: The MIT Press, 1984. Charles Rich and Richard C. Waters. ‘‘Automatic Programming: Myths and Prospects’’ 21:8 (August 1988), pp 40-51. Charles Rich and Richard C. Waters. ‘‘The Programmer’s Apprentice: A Research Overview’’, 21:11 (November 1988), pp 10-25. M. J. Rochkind. ‘‘The Source Code Control System’’, SE-1:4, (December 1975). pp 364-370. Lawrence A. Rowe and Sharon Wensel, editors. *Proceedings of the 1989 ACM SIGMOD Workshop on Software CAD Databases*, Napa CA, 27-28 February 1989. B. R. Rowland, R. E. Anderson, and P. S. McCabe. ‘‘The 3B20D Processor & DMERT Operating System: Software Development System’’, 62:1 part 2 (January 1983). pp 275-290. R. W. Schwanke, E. S. Cohen, R. Gluecker, W. M. Hasling, D. A. Soni, M. E. Wagner. ‘‘Configuration Management in BiiN SMS’’.

15-18 May 1989, Pittsburgh PA. pp 383-393. Izhar Shy, Richard Taylor, and Leon Osterweil. ‘‘A Metaphor and a Conceptual Framework for Software Development Environments’’, University of California, Irvine, Arcadia Document Number UCI-89-10. 14 July 1989. Douglas R. Smith, Gordon B. Kotik and Stephen J. Westfold. ‘‘Research on Knowledge-Based Software Environments at Kestrel Institute’’, SE-11:11 (November 1985). pp 1278-1295. Richard Snodgrass. Rockville MD: Computer Science Press, 1989. Vic Stenning. ‘‘An Introduction to ISTAR’’, Chapter 1 in Ian Sommerville, editor. IEE Computing Series 7. London: Peter Peregrinus Ltd., 1986. pp 1-22. Daniel Swinehart, Polle Zellweger, Richard Beach and Robert Hagmann. ‘‘A Structural View of the Cedar Programming Environment’’, 8:4 (October 1986) pp 419-490. 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’’. 28-30 November 1988, Boston MA. *SIGSOFT Software Engineering Notes,* 13:5 (November 1988). *SIGPlan Notices,* 24:2 (February 1989). pp 1-13. Warren Teitelman and Larry Masinter. ‘‘The Interlisp Programming Environment’’, 14:4 (April 1981). pp 25-34. Walter F. Tichy. ‘‘RCS — A System for Version Control’’, 15:7 (July 1985) pp 637-654. Colin Tully, editor. ‘‘Representing and Enacting the Software Process’’, 11-13 May 1988, Moretonhampstead, Devon, UK. 14:4 (June 1989). J. Walpole, G. S. Blair, J. Malik and J. R. Nicol. ‘‘A Unifying Model for Consistent Distributed Software Development Environments’’. 28-30 November 1988, Boston MA. *SIGSOFT Software Engineering Notes,* 13:5 (November 1988). *SIGPlan Notices,* 24:2 (February 1989). pp 183-190. David S. Wile and Dennis G. Allard. ‘‘Worlds: an Organising Structure for Object-Bases’’, 9-11 December 1986, Palo Alto CA. *SIGPlan Notices,* 22:1 (January 1987). pp 130-142. Alexander L. Wolf, Jack C. Wileden, Charles D. Fisher, and Peri L. Tarr. ‘‘PGraphite: An Experiment in Persistent Typed Object Management’’. 28-30 November 1988, Boston MA. *SIGSOFT Software Engineering Notes,* 13:5 (November 1988). *SIGPlan Notices,* 24:2 (February 1989). pp 130-142.