Industrial Strength Software Development Environments

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

Abstract

''Industrial strength'' software development environments (SDEs) must provide facilities to address three essential properties of multi-developer software systems: evolution, complexity and scale. It is my contention that in order to be an industrial strength SDE, an environment must support at least a *city* model environment. Moreover, SDEs must include such high level policies as supporting cooperation and communication, supporting a unified process and product, providing multiple means of expression, supporting inter- and intraconnections, managing the change process and managing the multiplicity of versions. Underlying the implementation of these policies are three themes: developer-machine symbiosis in which the machine becomes an active partner in system development and evolution; formalization of the development/evolution process so that we can reason about the process and automate its support; and finally, formalization of the software product so that we can reason about it and automate its support.

Invited Keynote Presentation
for the
Software Engineering Track
11th World Computer Congress IFIP '89
August 28 - September 1, 1989
San Francisco CA

1. Introduction

A few caveats are in order before getting to the heart of this paper. I am not concerned about describing the details of current environments that I consider to be ‘‘industrial strength’’ — you will see below that I consider none to be available today — or about how to build them. Rather, this is a visionary discussion about the problems software development environments (SDEs) must solve, and the facilities they must provide, in order for them to be considered ‘‘industrial strength’’.

The problem I am concerned about is the automation and support of the technical development of software systems by multiple developers. For this reason, I make a distinction between programming environments (PEs), software development environments, and software engineering environments (SEEs). Programming environments are those sets of tools used by an individual programmer to construct his or her program. I consider PEs only indirectly because the tools provided by them are included in software development environments. I do not address the problems of software engineering environments (which include the tools provided by SDEs) primarily to limit the scope of this paper. The discussion of SDEs exposes what I consider to be important issues that are equally applicable to SEEs, and, in some cases, compounded by the problems addressed in SEEs.

In the sections that follow, I first delineate what I consider to be some of the essential problems in multi-person development of software systems and then discuss each problem individually. Next I describe the implications of these problems for SDEs and consider what policies must be supported by SDEs to solve them. I conclude by considering some underlying themes that are intertwined in these solutions, themes that are necessary for ‘‘industrial strength’’ SDEs.

2. Some Essential problems

Brooks in ‘‘No Silver Bullet: . . .’’ [1] distinguishes between essential and accidental properties of building software systems. I would like to discuss several interrelated problems that are an essential part of the problem in the multi-person development of software systems:

- evolution,
- complexity, and
- scale.

These problems of evolution, complexity and scale are, in various ways, interdependent. For example, the problems of complexity are exacerbated by the multiplicity of interactions that accompany increased scale; the problems of complexity are heightened by the fact that the system may evolve in several directions at once; the problems of scale are made more difficult by complex and evolving systems; and the problems of evolution are compounded by complex systems changed by many people simultaneously.

2.1 Evolution

Software changes in a variety of ways: errors are corrected; data structures and algorithms are changed to improve space or time performance; machine-dependent parts are changed to port a system to new hardware; extensions are made to provide new features; and parts are used and reused to create new applications. Further, all these activities may occur concurrently with multiple-release, multiple-machine systems. (For seminal work on system evolution, see Lehman and Belady [10]).

The notion of ‘‘getting it right the first time’’, while perhaps laudable in reducing the iterations of fixing bugs and improving performance, has little to do with the problems of portation, iterative enhancement, or modification for new uses. This homily exhibits a lack of recognition that an important, essential feature of software systems is that they evolve.

More important still, evolution is an essential fact at the micro-level of software development as well the macro-level. Requirements evolve from vague beginnings, through iterations with the customer, to a firm set of (one hopes) clear specifications. Architectures evolve from the constraints imposed by the requirements and from various other considerations. Designs evolve, beginning with requirements and architectural constraints, into detailed descriptions. And, code itself evolves as the system is built and tested. Thus, the common wisdom that development is followed by maintenance is not supported by the reality of evolving systems. What is traditionally claimed to be a part of the maintenance process (in all its variety) occurs much earlier in the development process.

Further, the evolution at each point in the process causes iteration back to previous levels (or phases). The neat, tidy separation of phases found in the waterfall model [20], while useful at a high level of abstraction, poorly reflects the reality of development. The evolutionary aspects of development — and hence, redevelopment and reuse — cross both the micro- and macro-levels of the development process as well as the software system.

Hence, what is needed in ‘‘industrial strength’’ SDEs is support for an evolutionary paradigm that is radically different from paradigms that are supported today.

2.2 Complexity

Brooks states that ‘‘Software entities are more complex for their size than perhaps any other human construct . . . ’’ Wulf, London and Shaw [28] make an even stronger claim: ‘‘large programs, even not-so-large programs, are among the most complex creations of the human mind’’. This complexity arises from several distinct sources.

Part of the complexity arises from the difficulty in comprehending the intricacies of algorithms and data structures, particularly after they have been heavily optimized. Understanding these creations is often like understanding long and convoluted philosophical arguments: you often have to work through them carefully each time you encounter them in order to convince yourself that they are indeed correct (if in fact they are). It is not an uncommon occurrence for the creator of a piece of software to have difficulty reconstructing the intricate lines of reasoning that led to its current state. The

difficulty for someone other than the creator is magnified accordingly. When this intricate software evolves, it is often an exceedingly difficult and error prone process, even for the original creator.

Part of the complexity arises from the sheer wealth, or volume, of detail. This is where the problems of complexity and scale interact most directly. Compounding this sense of complexity is the fact that a single person is incapable of understanding a large system. Whatever handles we had on combatting the problems of innate intricacy do not apply here. One of the reasons for our lack of ability to comprehend large systems, is (as Brooks states) the fact that we do not construct large software systems by replicating a small set of building components as we do when constructing buildings or computers; instead, we construct large software systems by multiplying the number of distinct components. The difficulty in understanding the wealth of detail is compounded because the number of interactions among components grows non-linearly with the number of components. These non-linearly growing interactions make it extremely difficult to keep separate, but interacting and interlocking, pieces of the system consistent with each other.

In both cases above, part of the complexity arises because a large amount of the information needed to understand the intricacies and the wealth of detail is only implicit in the representations that we use. We have syntactic representations that reveal only a part of the immense detail that is actually present. Attached to these syntactic representations — the names, the structures, language statements, groups of names and statements — are meanings that are not visibly expressed.

These sources of complexity are compounded by the iterative and concurrent nature of building and evolving software systems: the intricate details change, the wealth of detail changes, and the underlying meaning and intent changes; often these changes occur without our noticing them. It is clear that we need tools in our SDEs that help us manage the intricacy and wealth of detail and their interactions, as well as tools that expose the underlying meanings of the details.

2.3 Scale

The problems of scale manifest themselves in two dimensions: the size of the software and the number of people building that software. We have already alluded to the problems of scale in the software itself. These problems are compounded when the number of people increases as well. Not only does the number of interactions among components of the system multiply, but the number of interactions among the people multiplies as well.

The thought of having only the most productive developers work on large systems, and thus reduce the problems of programming-in-the-many to programming-in-the-few, is a seductive one. Having the best people possible is clearly a goal that all projects strive for. However, the reality is that there are not enough of the best people to go around. Thus we must consider that one aspect of the problem of scale is the use of ordinary developers as a significant contributors in building systems.

Another strategy for reducing the numbers of programmers is that of making them more productive — for example by using higher-level languages, by reusing previously written

code, etc. Obviously, this is a laudable strategy. However, we usually use these productivity measures to shorten development time or to build larger systems instead of reducing staff; that is, we seem to be in the chronic state of trying to build rocks we can't lift.

Thus, the problems of scale must be faced and provided for in an industrial strength SDE. Facilities must be provided for managing, reducing, and enhancing the interactions among developers as well as making them individually more productive.

3. Implications for SDEs

These three essential properties of developing software systems have a number of implications for the kinds of features that an industrial strength SDE must have. In order to discuss these features in a uniform way, I will use a general model of SDEs introduced by Perry and Kaiser [16]; the model defines an SDE to consist of three interrelated components: a set of *policies*, a set of *mechanisms* and a set of *structures*. Policies are rules, guidelines and strategies imposed on the developer by the environment; mechanisms are the visible and underlying tools and tool fragments; and structures are the underlying objects and object aggregates on which the mechanisms operate. Mechanisms and structures together support the desired policies.

In the discussion that follows, I concentrate on the policies that are important in an industrial strength SDE; I also indicate what kinds of mechanisms and structures might be needed to support those policies.

Perry and Kaiser further delineated four classes of models according to a sociological metaphor that emphasizes the effects of scale on SDEs. These classes are points on a continuum of possible models, but they are selected for their suggestiveness about the different kinds of problems found in projects of various sizes. These four classes are the *individual*, the *family*, the *city* and the *state* models (where each larger class incorporates the smaller). The individual model represents a set of tools to be used by the individual developer who is working in isolation and concentrating on the *construction* of programs. The family models represents those tools that, in addition, facilitate the interactions of small groups of programmers working together. These tools assume that the members of the project act in reasonable ways and that only a minimal amount of *coordination* is needed (usually supplied by version management facilities). The city model represents those environments that support larger sized groups (say 20 or more) where the degrees of freedom allowed in the family model would result in anarchy. A richer set of policies is needed for supporting the *cooperation* needed in this model of environments. The state model supports a (possibly heterogeneous) set of SDEs (consisting of one or more of the other models) and administers a uniform set of policies across these environments — that is, a state model emphasizes a *commonality* across environments.

It is my contention that industrial strength SDEs must support at least a city model environment to address the problems of evolution, complexity and scale. Moreover, the following high level policies are necessary in order to address these problems:

- support cooperation and communication,

- support a unified process,
- provide a unified product,
- provide multiple means of expression,
- support interconnections and intraconnections,
- manage the change process, and
- manage the multiplicity of versions.

I will discuss each of these in turn by first considering the current support in SDEs for these policies, and then indicating some of the subpolicies, with their attendant mechanisms and structures, needed to support these high-level policies. As we shall see, the support of these individual policies interact in many interesting ways.

3.1 Cooperation and Communication

The SDEs at the current state of the art can support the individual and family model environments. Obviously, no cooperation or communication mechanisms are needed for the individual environments. Only relatively simple mechanisms are needed in a family situation where both cooperation and communication function at an informal level. For example, SCCS [19] or RCS [23] provide minimal coordination among developers in UNIX™ environments; electronic mail is used to some advantage within DSEE [10] to support communication among project developers about various important project events.

However, a much richer set of policies and their supporting mechanisms and structures are needed to provide an industrial strength environment. Because the interactions of larger groups of people are more complex, the rules and guidelines must be correspondingly more comprehensive as well. So far, very little work has been done in providing automated support for cooperation policies or in providing ways of managing communication among developers.

Indications of what these policies might be are provided by projects such as Infuse [13, 8] and IStar [4]. Infuse supports policies for both enforced cooperation, by means of hierarchical experimental databases which define the boundaries and the form of enforced cooperation, and voluntary cooperation, by means of workspaces which enable developers to form arbitrary alliances for cooperative purposes. IStar provides a contractual model which defines the boundaries of interactions between contractor and contractee in developing pieces of a system. The interactions are formalized as to what is required in the contract and how that contract is to be satisfied. Cooperation is circumscribed by these contracts.

In both Infuse and IStar, the policies are ‘‘hard-wired’’ into the environments. That is, the models of cooperation are pre-determined and supported (or enforced) by the environment. At a relatively low level, it is impossible to implement mechanisms and structures without implying some specific low level policy. However, there is a higher level of policy making that must be provided to the individual system development methodologists: they need a means of specifying the particular policies of cooperation

and communication appropriate to their particular process. I will consider this problem in further detail in the ‘‘underlying threads’’ section.

While some work has been done in supporting cooperation, very little has been done in facilitating communication beyond the use of electronic mail. This is an area that needs innovative exploration.

Equally important to supporting and making communication more effective is finding ways to reduce the amount of communication that must take place — an important way of lessening the multiplicative effect of scale. As formalization of the process provides a means of codifying the policies of cooperation, so formalizing the product provides a means of focusing some important aspects of communication among the developers. The formal product becomes the medium of communication and effectively reduces a many-to-many communication problem to a many-to-one communication problem. Such an approach focuses communication in a beneficial direction while reducing the amount of interpersonal communication that currently exists.

Industrial strength SDEs will provide policies, mechanisms and structures to supply various forms of cooperation and communication.

3.2 Unified Process

Currently, tool support in SDEs focuses on the coding, building and testing phases of the system life-cycle. We have elaborate policies, with their attendant mechanisms and structures, at very low levels of detail for these parts of the development process.

The earlier parts of the cycle — requirements, architecture, and design — are supported only in an ad hoc way, usually by documentation tools. While we are now beginning to see design tools on the market, due primarily to the advent of graphic terminals, we still have a long way to go before these tools are integrated into the life-cycle supported by integrated SDEs.

It is extremely tempting to think of the transition from one phase of the life-cycle to another as that of refinement. To some extent, the process of moving from requirements to architecture to design to code is one of refinement. Certainly, each phase adds more constraints on the solution and thus reduces the solution space. However, there are discontinuities encountered in the process as well as non-linear relationships between the components at one level and those of the next. For example, moving from an expression of *what* must be provided to *how* it is provided is both discontinuous and non-linear. Certainly non-functional requirements have these problems; I claim that functional requirements also have them.

Thus, what we need to support the entire life-cycle are policies and their supporting mechanisms and structures that are endemic to each activity and that support all the activities needed in the development of software systems.

3.3 Unified Product

As current process support in SDEs is centered around coding, so current product support in the development process is centered around the code. While we all recognize that there is more to the actual product that just the code, we still tend to emphasize this

specific part of the product to the detriment of the other parts. Requirements merely prime the pump; architecture and design get us to the point where we can at last begin to produce code; and so on. Code is the only thing that is kept up-to-date, obviously because it has to be. The other parts of the product are generally disregarded once they have served their initial function.

This approach to the system product has to change because the costs of doing business this way are too high. Unifying the various products of the development process into a multi-faceted product serves a number of very important functions. Of course, the unified process should have a policy that ascertains the consistency among the components and guarantees their currency. First, an integrated, timely product of this kind is of inestimable value in reducing the discovery costs that are incurred every time a new person becomes part of the project or when changes and additions are made to the product. Second, this unified product serves as the basis for negotiating changes to the product *at the appropriate level*, as well as assessing the costs and effects of those changes. Third, we have already mentioned the use of formality in reducing the paths of communication; a unified product provides similar advantages.

The problems of evolution, complexity and scale are currently compounded by the inhibiting underbrush (cf. Brooks [1]) generated with inaccurate, inadequate, incorrect and out-of-date components of the development product. An industrial strength environment will remove that underbrush.

3.4 Multiple-Expression

It is implicit in the typical mono-lingual environments and cultures that exist today that insufficient consideration (or, as I suspect in most cases, no consideration) is given to the effect of language on the development software systems. Samuel Johnson's maxim ''language is the dress of thought'' [7] catches some of the insight about language and expression. However, I think that Carlyle [3] captures the essence of the importance of our means of expression when he states ''language is called the garment of thought, rather it should be called the flesh garment, the body, of thought''. The means of expression should reveal the body that we intend to express; all too often it obscures our intent instead because of its inappropriateness.

The fact that we use a language like FORTRAN to do character processing or LISP to do numeric crunching shows how little we pay attention to the tools of expression that we have at our disposal. We should use the best means of expression for the particular problem. At a minimum, this argues for a multi-lingual SDE in which we can build multi-lingual systems.

The preceding observation about the inappropriate use of tools is concerned only about our coding language. How much more apt are these considerations for the various other parts of the software product. We may need a variety of means of expression for each of these parts. For example, requirements must provide both the contractor (customer, or user) and the implementers with their views about what the system is supposed to do. The means of expression for one are very different from those for the other. The users need the requirement expressed in non-technical terms, perhaps augmented by animation or by usable prototypes; the implementers need clear, unambiguous, precise and accurate

technical descriptions of the capabilities of the system to be built expressed in system building terms, not user's terms. These two distinct views obviously must be consistent with each other. Further, the problems encountered in finding the appropriate expression of these views may be compounded by the need for more than one mode of expression (as we argued for above in the actual coding of different problems). The appropriateness of a mode of expression may vary from one phase to the next: at one point the most useful expression may be textual rather than graphical; it may be formal rather than informal; or static rather than dynamic.

The need for multiple means of expression for each part of the product is complicated by the fact that each part requires its own mode of expression that may be distinct from that of the others. The means of expression for requirements are very different from that of architecture, especially that part of requirements that must be understood by the customer or user. The same relation holds true for expressing architecture and design.

One further comment about our means of expression. What Wittgenstein states in *Tractatus Logico-Philosophicus* [25] is applicable here: ''the limits of my language mean the limits of my world''. Equally important to expressivity is the limitation of that expressiveness. We want to rule out ambiguous, unclear expression as much as possible. While we cannot rule out poor or ambiguous thought, we can try to make all ill-thought expressions more readily apparent or perceivable as such. The use of formal languages for each of the components in a software product is a fruitful way of bringing this about.

Intuitively, an industrial strength SDE providing appropriate and limited means of expression should have a beneficial affect on all aspects of complexity.

3.5 Interconnections and Intraconnections

Providing multiple-expressions for the various components in a unified software product and supporting them in an SDE are necessary, but multiple means of expression by themselves are not sufficient to assist adequately in the iterative, evolutionary software development process. The various components in the product need to be interconnected — the dependencies between successive components need to made explicit. Further, each component in the product needs to be intraconnected — the dependencies within each component need to be made explicit.

In my paper, ''Software Interconnection Models'' [15], I distinguished three models of software interconnections: the *unit* model, the *syntactic* model, and the *semantic* model. The unit model is very useful in capturing relatively gross dependencies among various units in a component — for example, between the source files needed to build a system. Useful tools abound that make very good use of this large-grained kind of information, particularly that support the coding, building and testing parts of the process. The syntactic model is even more useful in capturing dependencies because it concentrates on the syntactic objects that we build our systems with — the model provides a finer grain of dependency. Various analysis tools (such as data-flow analysis tools, static-semantics analysis tools, etc.) produce very useful results based on these syntactic dependencies. The primary problem with both of these models is that there is no information about why these dependencies exists. It was for this purpose that I introduced the semantic interconnection model: to attach semantic information (expressing design intent) to

interface objects, and to use that semantic information expressed in those interface objects to capture the implementers intent. The semantic interconnection model uses unit, syntactic and predicate dependencies to express the interconnections among interface and implementation objects in software systems.

The primary advantage offered by these kinds of inter- and intraconnections is automation. The connections can be made automatically, and various kinds of analyses can use the connections to provide better understanding about the relationships between and among the components of a software system.

Unfortunately, where we have informal expressions, these kinds of techniques work only poorly, if at all. Here interactive techniques that make explicit the connections among informal components are needed. For example, Potts and Bruns in ‘‘Recording the Reasons for Design Decisions’’ [19] use a hypertext mechanism to formalize the connections between informal components in design documentation, such as design decisions and justifications, alternate decisions and their rationales.

Thus, an industrial strength SDE must provide tools to manage both the formal and informal inter- and intraconnections among the various parts of the software product.

3.6 The Change Process

The problems and interrelatedness of evolution, complexity and scale are most visible when making changes to systems. As we have mentioned earlier, this change process begins very early in the development process. The besetting problem in managing changes stems from the fact that implications of changes usually extend in many directions at once: within the current level of the system (for example, at the design level), at the lower levels (for example, in the implementation) and at higher levels (architecture and requirements). Certainly changes have effects on those parts of the system that are dependent on the part being changed. Changing the design of a data structure has a significant effect on all those parts of the design that depend on that structure, and on all parts of the implementation that depend on it. Moreover, changes may have upward effects as well. Some design changes have a profound effect on architecture; some architectural changes may have serious effects on the requirements or the satisfaction of those requirements.

What we need for industrial strength SDEs are tools that assist developers in making changes, and that take an active, symbiotic part in the change process. The automation of assistance in the change process is one of the primary benefits that accrues from having a unified software product with both formal and informal inter- and intraconnections. These connections can be used to aid in determining the implications, both upward and downward (or forward and backward, depending on your orientation) from the level in which the change is made.

The Inscape Environment [18] is a research experiment whose main concern is building tools to support the construction and evolution of large systems. The tools are integrated around the constructive use of formal module interface specifications with the environment automating the semantic interconnections provided by those specifications and their use in implementations. Inscape thus makes use of unit, syntactic, and semantic

interconnections to assist in the change process. As the system is constructed, the environment automatically (or interactively where automation is not possible) records the semantic dependencies of implementations on the interface objects as well as the semantic dependencies on local objects. As a module implementation is built, Inscape propagates an interface that is derived from the implementation. The propagated interface can be compared against the specified interface to see how well the implementation satisfies the specification. The first major benefit from this approach is an automated link between the interface specifications (the interface design) and the implementations using and generating those interfaces. The second benefit, and I think the more important one, is that the implications of changes can be determined by the environment by using the semantic interconnections established during system construction. If a change is made to an interface specification, Inscape checks the various places where that part of the interface is used to determine the effects of that change on the implementation (for example, whether there was an effect at all, whether code may be removed, or whether new code must be added) and reports it to the user. Conversely, if a change is made in the implementation, Inscape determines how that affects both the implementation and the enclosing interface. Thus, we have in Inscape an example of using the syntactic and semantic inter- and intraconnections to provide assistance in understanding the effects of change and automatically propagating the effects of those changes.

Another important benefit of an approach such as Inscape's semantic interconnections is the explicitness of semantic dependencies. In the discussion on complexity above, I mentioned that one of the contributing factors was the lack of visibility of important detail. Semantic dependencies lurk beneath the visible detail of syntactic dependencies. Here those details are made explicit and visible and we are thus able to take them into consideration and reason about them as part of the evolution process.

An industrial strength SDE will provide automated assistance in managing the change process as a necessary part of managing the problems of evolution, complexity and scale and reducing their effects.

3.7 The Multiplicity of Versions

At the beginning of a system's development, there is little problem in keeping track of the various pieces of the system. The basic relationship between versions — that of derivation — is sufficient to manage the system components. But as changes are made, enhancements are installed, new uses are incurred, and multiple products are developed in parallel, the problems in keeping track of the versions for different parts of the system become significant. The basic relationship of derivation is still important because it provides a history mechanism, but it is not sufficient to describe the more complex relationships that result from these various strands of use and reuse.

Of the high-level policies that we have discussed, this one of managing a multiplicity of versions has the most support in current SDEs, at least as far as managing the various versions of source code are concerned. We have mechanisms and structures that provide the basic relationship of derivation in two forms: the relationships of revision and variant (that is, sequential and parallel versions). While these two relationships express some of

the intuition about the different kinds of relationships that exist among versions, they are not sufficient to support one very important aspect of version and configuration management: what happens when you use one version instead of another. The integrating policy needed here is ‘‘support the concept of substitutability in building composed versions of systems’’.

To accomplish this policy, we must be able to formalize the kinds of relationships that occur between individual versions and among groups of versions, both independent of their use and in the context of their use. In current mechanisms, the various kinds of relationships among versions are determined in an ad hoc way — usually by developer fiat. With the formalization of various products in the system, we can provide formalization of these relationships so that they can be automatically determined.

Moreover, current mechanisms tend to concentrate on relationships of versions independent of their use and ignore contextual relationships. Note that there is some consideration of their use in the emphasis on resource dependence: Tichy [23] has a notion of ‘‘upward compatibility’’ that depends upon the resources provided as well as those depended upon. However, this concept falls short of what is needed in the tools to manage the building of systems from components. Invariant [15], the version management part of Inscape, defines several notions that are of importance here [15]. The first is that of version *equivalence*. Clearly anything that is equivalent is substitutable. The second is that of version *compatibility*. Typically, only part of the behavior of interface objects are used in a particular implementation. If one substitutes objects that supply that required behavior, even if they are not equivalent, then they can be considered compatible. Resource dependency can also be analyzed in such a way to define version resource compatibility. Well-formed system compositions [5, 6] enable us to determine whether all the resources are supplied when substituting one version for another that is compatible in the sense that it supplies the required behavior, but unknown as to whether it is resource compatible.

An extremely important aspect about groups of components that is not addressed in current mechanisms, is that a number of components have a relationship as a whole. For example, a set of components might together embody a particular change and must be used together, even though individually they may appear to be equivalent versions of other variants. What often distinguishes this set of components from other collections of components is their dependence upon some shared resource or algorithm.

In order for an industrial-strength SDE to manage the multiple dimensions of the unified system product, we will need a sufficiently rich set of versions relationships. Current mechanisms must be extended to manage all components in the unified product, to provide the kinds of version relationships that are endemic to each component in the software product, and to support intercomponent relationships.

4. Conclusions — Underlying Themes

I have said very little about the actual mechanisms and structures needed for industrial strength SDEs. Instead I have concentrated on the policies and high level features that SDEs must have in order to be considered ‘‘industrial strength’’. However, there are

implications for these underlying mechanisms and structures — the policies and features discussed form the requirements to be satisfied — and in many cases, there are still open questions that must be solved. For example, the question of what kind of general underlying structure is needed to support an industrial strength SDE is still an open question. Current work is concentrated on ‘‘object bases’’ [22, 21]. Whatever its form, it must be able to support the wide variety of components that comprise a system, the multiple expressions of those objects, their inter- and intraconnections, and a wide variety of relationships.

In the preceding section, a number of underlying themes have emerged as basic to the variety of solutions needed to construct industrial strength SDEs. The most important of these are the formalizations of both the process and the product. Somewhat orthogonal to these two themes — and, in fact, enabled by them — is the increased participation of the environment in the construction and evolution of software systems — the environment working in symbiosis with the developers.

4.1 Developer— Machine Symbiosis

The basis for the increased synergy between developers and the environment is automated analysis (where possible, interactive analysis where it is not possible) and environmental management of the resulting dependencies and connections at unit, syntactic, and semantic levels. By means of this analytical detail, industrial strength SDEs will manage the process of evolution by determining the implications of changes according to the various levels of interdependencies. Further, the understanding incorporated in the environment of the various modes of expression will enable the industrial strength SDE to aid the programmer in handling the various aspects of complexity.

The projected results from this increased symbiosis are improved quality in the system and increased productivity in the developers. Specific advantages include error prevention, early error detection, and reduced discovery costs.

4.2 Formalization of the Process

One of the primary ingredients in supporting the policies discussed in the preceding section and in providing the increased symbiosis between the developers and the environment is the formalization of the software process. Formalization yields better understanding of the process and provides a means of reasoning about that process. This understanding and reasoning ability can then be incorporated into the environment and, thus, provide better support for cooperation and increased effectiveness of communication among developers, and support for the activities in creating and evolving systems. Finally, a better understanding of the process — that is, a formalization of the process — yields a better understanding of the product and aids our formalization of the product as well.

Modeling the software process with formal process models [26, 4, 5, 11] and process programs [12], and providing the underlying support to enact them is an important direction of current research. Using process models, we formally define the particular process that is appropriate to the product and to the project — we formally define the

policies that dictate how process activities are related, how developers interact with each other (that is, how they cooperate and communicate), how developers interact with their tools, and so forth.

Formalizing the process is a necessary first step towards industrial strength SDEs.

4.3 Formalization of the Product

Formalization of the product is of paramount importance in supporting a unified product, in enabling the environment to determine the various inter- and intraconnections, in supporting the environmental management of the evolution process, and in managing the multiplicity of versions that accrue as the system evolves.

Formalization of the product is fundamental in exposing the otherwise implicit complexity of software, reducing the increased communication as the scale of a project increases, and improving the quality of the communication that occurs. The formalized product serves (with the aid of the SDE) as the oracle about the product. Of course, this can only happen if all the components in the product are consistent with each other and current with respect to each other.

Thus, formalization of the product is another necessary step towards industrial strength SDEs.

Acknowledgements

Tom Wetmore, Anil Pal, Mark Dowson, and Winston Royce provided insightful readings of this paper.

References

[1] Frederick P. Brooks, Jr. ‘‘No Silver Bullet: Essence and Accidents of Software Engineering’’, *Computer*, 20:4 (April 1987), pages 10-20.

[2] Thomas Carlyle. *Sartor Resartus*, 1833-34. Chapter 11.

[3] Mark Dowson. ‘‘Integrated Project Support with IStar’’. *IEEE Software*, November 1987. pp 6-15.

[4] Mark Dowson, editor. *Proceedings of the 3rd International Software Process Workshop: Iteration in the Software Process*, Breckenridge CO, November 1986. IEEE Computer Society, 1987.

[5] A. Nico Habermann and Dewayne E. Perry. *Well Formed System Composition.* Carnegie-Mellon University, Technical Report CMU-CS-80-117. March 1980.

[6] A. Nico Habermann and Dewayne E. Perry. ‘‘System Composition and Version Control for Ada’’. Symposium on Software Engineering Environments. Bonn, West Germany. June 16-20, 1980. Published in *Software Engineering Environments*, edited by H. Huenke, North Holland, 1981, pp. 331-343.

[7] Samuel Johnson. *Lives of the English Poets*, ed. G. B. Hill, 1905. Volume I, ‘‘Cowley’’, p 58.

[8] 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.

[9] David B. LeBlang and Robert P. Chase. ‘‘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

[10] M. M. Lehman and L. A. Belady. *Program Evolution. Processes of Software Change*. APIC Studies in Data Processing No. 27. London: Academic Press, 1985.

[11] M. M. Lehman. ‘‘Process Models, Process Programs, Programming Support’’. *Proceedings of the 9th International Conference on Software Engineering*, Monterey CA, March 1987. pp 14-16.

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

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

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

[15] Dewayne E. Perry. ‘‘Version Control in the Inscape Environment.’’ *Proceedings of the 9th International Conference on Software Engineering*, Monterey CA, March 1987. pp 142-149.

[16] Dewayne E. Perry and Gail E. Kaiser. ‘‘Models of Software Development Environments.’’ *Proceedings of the 10th International Conference on Software Engineering*, Raffles City, Singapore, April 1988. pp 60-68.

[17] Dewayne E. Perry. ‘‘The Inscape Environment.’’ *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh PA, May 1989.

[18] Colin Potts and Glenn Bruns. ‘‘Recording the Reasons for Design Decisions’’. *Proceedings of the 10th International Conference on Software Engineering*, Raffles City, Singapore, April 1988. pp 418-427.

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

[20] Winston Royce. ‘‘Managing the Development of Large Software Systems’’. *IEEE WESCON Proceedings*, August 1970, pp 1-9. Reprinted in *Proceedings of the 9th International Conference on Software Engineering*, Monterey CA, March 1987, pp 328-338.

[21] Lawrence A. Rowe adn Sharon Wensel, editor. *Proceedings of the 1989 ACM SIGMOD Workshop on Software CAD Databases*, Napa CA, February 1989.

[22] Richard N. Taylor, e*et al.* ‘‘Foundations for the Arcadia Environment Architecture’’. *Proceedings of ACM SIGSOFT ’88: Third Symposium on Software Development Environments*, Boston Mass., November 1988.

[23] Walter F. Tichy. *Software Development Control Based on System Structure Description*. Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University, January 1980.

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

[25] Colin Tully, editor. *Proceedings of the 4th International Software Process Workshop: Representing and Enacting the Software Process*, Devon, England, May 1988. IEEE Computer Society, 1988.

[26] Jack C. Wileden and Mark Dowson, editors *Proceedings of the 2nd International Workshop on The Software Process and Software Environments*, March 1985, Coto De Caza, Trabuco Canyon, CA. *Software Engineering Notes* 11:4 (August 1986).

[27] Ludwig Wittgenstein. *Tractatus Logico-Philosophicus*, ed. A. J. Ayer, London: Routledge & Kegan Paul, 1961. Section 5.6.

[28] W. A. Wulf, R. L. London, and M. Shaw. ‘‘Abstraction and Verification in Alphard: Introduction to Language and Methodology’’, *IEEE Transactions on Software Engineering*, SE-2: 4 (December 1976), pages 253-265. Reprinted in M. Shaw, editor, *ALPHARD: Form and Content*. New York: Springer-Verlag, c1981. Pages 15-59.