Software Engineering Tools and Environments: A Roadmap

William Harrison **Harold Ossher** **Peri Tarr**

IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598, USA
+1 914 784 7975
{harrisn,ossher,tarr}@watson.ibm.com

ABSTRACT

Tools and environments to aid developers in producing software have existed, in one form or another, since the early days of computer programming. They are becoming increasingly crucial as the demand for software increases, time-to-market decreases, and diversity and complexity grow beyond anything imagined a few decades ago. In this paper, we briefly review some of the history of tools and environments in software engineering, and then discuss some key challenges that we believe the field faces over the next decade.

Keywords

Tools, programming support environments, software engineering environments, process-centered software engineering environments, integration, separation of concerns.

1. INTRODUCTION

All software engineers use tools, and they have done so since the days of the first assemblers. Some people use stand-alone tools, while others use integrated collections of tools, called *environments*. Over time, the number and variety of tools has grown tremendously. They range from traditional tools like editors, compilers and debuggers, to tools that aid in requirements gathering, design, building GUIs, generating queries, defining messages, architecting systems and connecting components, testing, version control and configuration management, administering databases, reengineering, reverse engineering, analysis, program visualization, and metrics gathering, to full-scale, process-centered software engineering environments that cover the entire lifecycle, or at least significant portions of it. Indeed, modern software engineering cannot be accomplished without reasonable tool support.

The role of computers, their power, and their variety, are increasing at a dramatic pace. Competition is keen throughout the computer industry, and time to market often determines success. There is, therefore, mounting pressure to produce software quickly and at reasonable cost. This usually involves some mix of writing new software and finding, adapting and integrating existing software. Tool and environment support can have a dramatic effect on how quickly this can be done, on how much it will cost, and on the quality of the result. They often determine whether it can be done at all, within realistic economic and other constraints, such as safety and reliability.

Software engineering tools and environments are therefore becoming increasingly important enablers, as the demands for software, and its complexity, grow beyond anything that was imagined at the inception of this field just a few decades ago.

This paper does not attempt to survey this vast field in full, even less to predict the myriad paths of its evolution. Instead, it presents a personal view of some of the key issues and themes that we believe to be especially important.

The rest of this paper is organized as follows. Section 2 reviews some of the history of tools and environments in software engineering and identifies some key, recurring themes that have influenced the field in the past and are likely to continue to do so in the future. In Section 3, we describe some critical issues that we believe the software engineering domain currently faces and indicate some requirements on solutions. Section 4 presents some solution approaches to the problems raised in Section 3 and identifies current technologies that we believe may serve as enablers for these approaches. Section 5 describes some challenges presented by new application domains and indi-

cates how these domains further constrain solutions. Finally, Section 6 presents some conclusions.

2. A BRIEF HISTORY

Tools and "environments" have existed, in one form or another, to aid developers in producing software since compilers and editors became standard offerings with operating systems. The earliest "environments" were little more than small collections of stand-alone tools that could be used in a loosely coordinated fashion to help accomplish software engineering goals. Unix [36] is a good example of such an environment. It provides tools, like editors, compilers, debuggers, and other utilities (e.g., awk, sed, grep, find), whose inputs and outputs can be interconnected loosely, at the developer's discretion, via pipes and redirection. This interconnection is possible because the tools all accept and generate data in simple, standardized formats.

Early "environments," while quite useful, did not provide any real means of integrating tools, coordinating their executions, or automating common tasks—they simply provided tools, and they required developers to employ appropriate usage conventions that would permit the developers to coordinate the use of the tools. The earliest vehicles for automating the flow of control among tools, like Make [21] and Odin [15], were suited to a batch-style of operation. They provide the ability for developers to describe how tools and data relate, and *which* tools should be used to process changed data. For example, Make allows developers to indicate that a change to a particular set of source code files should result in recompilation. The use of the compiler may then trigger the execution of a test suite, and so on. The addition of tool integration mechanisms significantly enhanced the usability of collections of stand-alone tools, but the resulting environments were still loose federations of tools, and they suffered from all the usual problems associated with loose integration.

The first significant efforts in producing tightly integrated development environments were those in the area of programming support environments (PSEs). As their name suggests, PSEs are collections of tools that support *coding* activities. Earlier PSEs, like Pan [4], Gandalf [28], and the Cornell Synthesizer Generator [54], typically provided one or more compilers, language-sensitive editors (such as syntax-directed editors), and debuggers, and sometimes other tools as well (e.g., testing or documentation utilities). These tools were integrated tightly so that the activities of one tool were reflected appropriately in other tools. For example, changes made in the editor might be reflected immediately in the compiler and in the debugger. PSEs also generally included some support for task automation. For example, when a developer asked to run a program in the debugger, the PSE might first determine whether all compiled modules were up to date and recompile accordingly. The early PSEs introduced the use of inter-tool events to coordinate their operation and remove the batch-like control of *when* tools should be used to process data. They provided a message server with which tools could register their interest in particular events that other tools sent to the server. Upon receiving an event notification, the server would identify those tools that had expressed interest in the event and would forward it to those tools. In later PSEs, like Field [49], this *event-based* view of *control integration* allowed, for example, a compiler to coordinate with separately running editors and debuggers by sending events such as the location of syntax errors. The editors and debuggers, having registered their interest in such events, would be notified of the errors, and could highlight them.

PSEs comprised tightly integrated collections of tools, and as such, they were able to overcome many of the problems associated with earlier, loosely integrated environments. They were, and continue to be, extremely useful tools, and they are still used by the majority of developers today. Their major limitation, however, is that they support only one software engineering activity and its artifact—implementation and code, respectively—and exclude all other major activities and their artifacts, such as requirements engineering, specification, design, testing, and analysis. Even the earliest software lifecycle models (e.g., the waterfall model [51]) noted, of course, that all software engineering activities are interconnected, as are the artifacts produced by those activities. Thus, one activity may affect another, and changes to one artifact may necessitate changes to other, related artifacts, to ensure that the artifacts remain mutually consistent. Although even early environments provided some tools that promoted the development of artifacts other than code, these tools were not integrated with other tools, and the relationships between their artifacts and those of other tools could not be represented or used to automate tasks, such as change propagation. This was, essentially, the same situation as existed in the very earliest "environments," which contained collections of stand-alone tools that developers could choose to use in a coordinated fashion, but which did not, themselves, provide any assistance in doing so. The identification of the need for integrated support for software engineering activities throughout the software lifecycle represents the genesis of *software engineering environments* (SEE).

SEEs, like their PSE antecedents, are integrated collections of tools that facilitate software engineering activities. However, SEEs extend well beyond programming support tools, supporting software engineering *across the software lifecycle.* For example, RPDE^3 [43], a prototype for integrating activities from design through testing, emphasized the use of a natural human interface to promote development-by-elaboration, permitting different "perspectives" to be taken on an underlying web of design and development information. The Arcadia SEE [53] included tools for requirements specification, high- and low-level design, analy-

sis, and testing. These tools facilitated the development and analysis of different software artifacts. Moreover, the Arcadia environment supported *traceability* across the artifacts by permitting the description of interrelationships across different pieces of different artifacts; thus, for example, developers could indicate which design modules addressed particular requirements, and which code units implemented particular design modules. This kind of traceability greatly facilitates comprehension and change identification and propagation—a crucial improvement over traditional PSEs. In the area of environments and tools, SEEs represent, perhaps, the most significant area of research over the past decade; they are to the full software engineering lifecycle what PSEs were to coding.

Some earlier SEE-related research and development focused predominantly on the production of tools to aid activities at stages of the software lifecycle other than coding, and included "hooks" to enable connection with other tools or artifacts. The "Interactive Design Environment" [58] was a well-known move toward emphasizing design while retaining a connection to its realization in code. The dominant theme in early SEE research was the integration of tools across the software lifecycle, thus broadening the domain across which tools can be applied.

Because program code is generally expressed as programming language text, PSE's could rely upon widely-accepted standards for the data they shared. The broadening of scope to other development concerns forced attention to be paid to support for *data integration*: sharing data and integrating tools with respect to the data they share. One common theme was *repository-based integration*, an integration model that posited a common model for the shared information and provided support for its storage and management of concurrent and secure access. PCTE [12] is a well-known exemplar of this approach. Unfortunately, the models existing at the time could not reconcile the need for a commonly available view with the need for rapid changes of view by individual tools.

The fact that rapid development dominated the tool scene, coupled with the hope that distributed objects would solve the data integration problem, has hampered progress on SEEs. But two important lines of research have emerged from the SEE domain in the last decade, and each addresses a major theme of great current importance in software engineering. *Multi-view software environments* facilitate the development of one or more artifacts when different developers may have different views of the artifacts they are developing, and of the processes by which they are developing them. Thus, for example, in the Viewpoints system [41], two developers collaborating on a requirements specification may each see different models of the requirements, represented in different formalisms, and may each use different process models. The environment uses descriptions of intra- and inter-artifact consistency to maintain consistency across the different representations, or to manage inconsistency as it arises. As another example, in Statemate [29] and Rational Rose [48], a single developer may be able to work with multiple views of a given piece of software—for example, a control-flow representation and a data-flow representation, or a class view and an object collaboration view—simultaneously, and be able to update one view and have the other updated automatically. Multi-view environments facilitate the definition and integration or coordination of different, simultaneous views of software artifacts.

The second major line of SEE research was initiated by Osterweil's landmark paper [46], which posited the need for semi-automated support for the software *process* [22], in addition to tool support for artifact development. Osterweil hypothesized that the software engineering process should itself be treated as a piece of software—one that undergoes a similar lifecycle, including requirements specification, design, implementation, testing, analysis, etc. This hypothesis has had a profound effect on SEE research. It gave rise to *process-centered software engineering environments* (PSEE), which integrate tool support for software *artifact* development with support for the modeling and execution of the software engineering *processes* that produce those artifacts. The explicit representation of processes, their products, and their interactions, is the foundation on which modern integrated development environments, like the Rational environment, are built. In providing more powerful ways of describing and implementing software engineering processes, PSEEs have also provided a powerful means of integrating processes and tools, and (partially) automating tasks. Some important examples of PSEE research include Adele [8], ALF [13], Arcadia [53], EPOS [16], GOODSTEP [19], Marvel [35], Merlin [34], MELMAC [17], OIKOS [2], Oz [9], PCTE [12], and SPADE [5].

3. KEY CHALLENGES

Some common themes are apparent throughout the history of software engineering tools and environments research. Not surprisingly, perhaps the most ubiquitous is the theme of *integration*. As we have seen, the need for integration—of tools, processes, artifacts, and views—has been a driving force for many of the major changes in direction and new lines of research. Integration has been accomplished using a veritable plethora of approaches—ASCII text, shared repository, standardized representations and interfaces, transformation and adaptation, event-based integration, frameworks, etc.—and tools and platforms that facilitate them.

Along with work on integration and other themes like software process support and multi-view environments, a vast collection of tools have been prototyped or marketed, covering the tremendous range of software engineering activi-

ties mentioned in Section 1. Some of these have been developed in the context of integrated environments, some can cooperate loosely with some others, many are freestanding.

Unfortunately, the state-of-the-practice in software engineering has not advanced nearly as much as this impressive set of tools and integration mechanisms would imply. Why not?

A key reason is that each tool or environment is still highly specific to some context. It might require the software it manipulates to be written in a particular language, or to be represented using a particular kind of intermediate form or program database. It might run only on particular hardware, under a particular operating system, or using a particular compiler, environment or integration platform. It might require the presence of a variety of other tools or infrastructure components that the developer cannot or does not wish to use. It might not contain the "hooks" needed to enable it to work properly with other software the developer must use.

The *concepts* embodied in such tools are often applicable to other contexts, but the tools themselves are not. Developers frequently see tools they would like to use, or capabilities they would like to have within their own environments, only to discover that a context or packaging mismatch precludes or greatly complicates the use of those tools [24]. Tool developers are constantly plagued by the problem of making their tools available in multiple contexts, to achieve wider use, rather than being able to devote their efforts to innovation.

A major challenge for the tools and environments community is, therefore, to find ways to build and integrate tools so that they, or capabilities within them, can be easily adapted for use in new contexts. It is unlikely that contexts will become sufficiently standardized to enable "plug-and-play;" history demonstrates that standardization cannot be relied upon as a complete solution, because, among other reasons, new domains are identified as needing standards only well after developers in more than one group have already built software for them. Instead, adaptation and integration are key capabilities. Tools play a critical role in enabling flexible adaptation and integration, so this becomes a double challenge for the tools community:

- Developing and using tool architectures and other approaches that facilitate adaptation and integration.
- Building tools to assist with the adaptation and integration process.

3.1 Permanently Malleable Software

This problem of separate islands of context-specific software actually exists throughout the software world. As software has become more and more pervasive and its life expectancy has increased, it has been subject to greater pressures to *integrate* and *interact* with other pieces of software, and to *evolve* and *adapt* to uses in all manner of new and unanticipated contexts, both technological (e.g., new hardware, operating systems, software configurations, standards) and sociological (e.g., new domains, business practices, processes and regulations, users). Unfortunately, as the ongoing software crisis attests, software fundamentally cannot meet these challenges. Evolution, adaptation and integration are costly and difficult, if they can be performed at all. As a result, more and more people are using software that fails to do what they want, that does things they do not want, that does not integrate well with their hardware, other software, or into their business processes, and that becomes ever more brittle and unreliable over time. Essentially, software is like a square peg being forced into round, triangular, octagonal, and other kinds of ever-changing (and rapidly changing) holes into which it does not, and cannot readily be made to, fit.

This problem arises because it is impossible to predict, even with the most careful analysis, how business practices, business processes, people, and technology will co-evolve over time. As people use software, and as their businesses, domains, and available technologies change, they find different uses for the software, and they impose new requirements on it. In rapidly-changing technology fields, it is little wonder that it is impossible to anticipate all uses to which a piece of software will be put in the future, and in what contexts it will be used.

At present, however, the technology available to aid in software integration and evolution depends fundamentally on the ability to *anticipate* and *pre-plan* for change: designers anticipate likely evolution scenarios and variations, and build in "open points" to make them straightforward to accomplish, using techniques such as frameworks [59] and design patterns [23]. When significant changes arise that were not anticipated, reengineering is usually needed. Despite reengineering tools [40], this remains an exceedingly costly and error-prone process.

The reliance on pre-planning for change is problematic for two reasons. First, as noted above, it is impossible to anticipate all future needs, and changes that were not anticipated remain difficult or impossible to accomplish. Second, many current pressures militate against good up-front planning. Traditional models of software development (notably, the spiral [10] and waterfall [51] models) involve careful requirements analysis and design prior to implementation. This may be appropriate in more traditional contexts, such as the development of large, high-quality systems over a significant period of time, or the engineering of safety-critical software [38], but in many areas of business and research today there is great pressure to be first to market or first to publish, even at the cost of reduced quality. Products that appear first often capture the market, and higher-quality alternatives released later are often too late. This puts great

pressure on developers to shorten the software lifecycle in favor of quick release of a product that does enough to capture interest and a significant share of the market. Initial development is fast and focused, without spending time on planning for future changes that probably couldn't be anticipated adequately anyway. The changes come later and are often difficult, requiring some amount of reengineering to introduce needed open points. Clearly, the state-of-the-art approach to evolution—careful pre-planning—does not itself satisfy current and evolving needs.

In a nutshell, software is currently like *clay*, and it needs to become like *gold*. Clay is soft and malleable initially, but then it hardens. After that, bumps can be added to it, but it cannot be changed or reshaped without breakage. Attempts to force a hardened clay peg into a hole of a different shape are likely to lead to breakage. Gold, on the other hand, remains malleable for life. It can be reshaped as needed, and will assume the shape of a hole into which it is hammered.

We introduce the term *morphogenic software*[1] to refer to software that is malleable for life: sufficiently adaptable to allow context mismatch to be overcome with acceptable effort, repeatedly, as new, unanticipated contexts arise. In other words, it can be reshaped to fit into new holes as needed.

Morphogenic software extends the goals of integration, discussed in Section 2. Traditional integration solutions say, in effect, "integration can be achieved by building software using this particular integration approach (e.g., event mechanism, repository, or process specification approach)." This means that, when writing some software, one does not have to anticipate what other specific software it will have to integrate with, but one *does* have to commit to a particular context (the choice of integration approach). Adaptation across contexts is still difficult or impossible. Morphogenic software goes a step further by requiring a commitment to integration, but to not any particular approach.

In other words, the challenge is to find ways to write software, and tools to manipulate software, that will facilitate both rapid initial development and later adaptation and integration in new contexts, without up-front knowledge of what those contexts will be. This combines and extends some key challenges of the past, including integration, adaptability, reuse, portability, and retargetable systems, in the context of the modern software development milieu.

3.2 Separation and Integration of Concerns

A major barrier to portable software and morphogenic software is inadequate separation of concerns. The software implementing a desired capability may depend deeply on other software that is highly specific to use in a particular context. Worse yet, the parts of software artifacts that embody the capability may well be tangled or coupled with context-specific parts, or parts pertaining to other capabilities, so that one cannot even identify a module or collection of modules devoted solely to the capability itself. This makes evolution, adaptation or reuse of the capability difficult or impossible.

Separation of concerns has been a guiding principle of software engineering for decades [47]. Why, then, do these difficulties remain? We believe that the primary reason is a problem we have termed the *tyranny of the dominant decomposition* [52]. Software started out being represented on linear media, and despite advances in many fields, such as graphics and visualization, hypertext and other linked structures, and databases, it is still mostly treated as such. Programs are typically linear sequences of characters, and modules are collections of contiguous characters. This linear structure implies that a body of software can be decomposed in only one way, just as a typical document can be divided into sections and subsections only one way at a time. This one decomposition is dominant, and often excludes any other form of decomposition. For example, functional programs can only be decomposed by function, and object-oriented programs by class.

Traditional application of separation of concerns therefore involves examination of different separation criteria, such as functional versus data decomposition, and choosing one. In other words, which tyrant is best? The answer is that different tyrants are better in different situations [25]. Different concerns are important for different development activities at different stages of the software lifecycle, and for different people playing different roles in a software process. As a result, different decompositions—perhaps several simultaneously, as seen in the work on multi-view SEEs—are appropriate at different times. In more recent research, therefore, the focus has shifted to eliminating the tyrant, at least in part, by allowing separation according to more than one kind of concern at a time (e.g., [1][3][25][37][39][52]).

To realize the benefits of separation of concerns fully, it must be possible to achieve separation according to arbitrary kinds of concerns. These include, but are by no means limited to, function, data type or class, feature [56], aspect (e.g., distribution or persistence) [37], role [3], variant, viewpoint [41] and unit of change. Some concerns are identified early in the lifecycle, and guide initial develop-

[1] The term "morphogenic" is derived from "morphogenesis," which, in biology, refers to the "process of shape formation, which is responsible for producing the complex shapes of adults from the simple ball of cells that derives from division of the fertilized egg" [On-Line Medical Dictionary]. The parallels between the changes an organism undergoes during development—subject constantly to environmental and evolutionary pressures—and those a piece of software undergoes during the course of development, are striking.

ment; others first become apparent later in the lifecycle. Many concerns span multiple software artifacts and even multiple phases of the lifecycle. Full support for separation of concerns, liberated from the tyranny of the dominant decomposition, must allow the developer engaged in some development activity to focus on and work with whatever concerns of whatever kinds are appropriate for that particular activity, irrespective of the particular concerns used to decompose the software when it was first written.

It is appealing to think of many concerns as being independent or "orthogonal," but this is rarely the case in practice. Persistence and distribution are a common example: while they seem to be orthogonal capabilities that can exist separately or together, their implementations usually interact, and the details of each depends on the presence or absence of the other. It is essential to be able to achieve useful separation of overlapping and interacting concerns, identifying the points of interaction and maintaining the appropriate relationships across these concerns as they evolve.

Separation of concerns is clearly of limited use if the concerns that have been separated cannot be integrated; as Jackson notes, "having divided to conquer, we must reunite to rule" [32]. Thus, any approach to separation of concerns must have a corresponding integration approach, to permit the synthesis of software that deals with multiple concerns. Functional languages, for example, use nesting and function call as the means of building software from separate functions. Integration mechanisms within SEEs can be seen in this light, with the individual tools as the concerns being integrated[2]. Integration support must, therefore, grow in sophistication along with means of separating concerns.

We use the term *multi-dimensional separation of concerns* to refer to separation of concerns with the properties noted above [45]. We believe that, just as traditional separation of concerns has been key to software engineering in the past, multi-dimensional separation of concerns will be key to software engineering in future, enabling such critical activities as understanding, evolution, adaptation, reuse and morphogenesis. The challenge is to create mechanisms, methodologies, tools and environments that support it and leverage it.

4. MEETING THE CHALLENGES

We believe that addressing the challenges noted in Section 3 will be among the key thrusts in software engineering over the next decade. In this section we outline some of the approaches and specific requirements we foresee in the area of tools and environments.

[2] Each tool, in turn, usually consists of subsidiary concerns, such as functions or classes, integrated using programming language mechanisms.

4.1 Deployment of Commercial Technologies

Recent developments in commercial software development provide technologies that begin to address the integration challenges discussed above. These advances allow the basic functional characteristics of some software to be expressed without fixing decisions that depend on its context. We refer to software whose structure embodies the flexibility to allow it to be "grown" into many differently-shaped derivatives to fit different environmental concerns as *stem-software*, a limited kind of morphogenic software. This characteristic, and the increasingly widespread acceptance of these commercial technologies, make them a good foundation. If focused effort is put into enhancing and applying them suitably, they can be expected to grow towards more complete solutions.

4.1.1 Data Integration and XML

Adoption of XML (the eXtensible Markup Language) [60] by future software development tools may break a logjam that inhibits data sharing between software tools.[3]

Solutions to data integration issues can be characterized as *unidirectional* or *omnidirectional* sharing solutions. Unidirectional sharing feeds the data from one tool to another, as in Unix pipes, while omnidirectional sharing feeds information back-and-forth among several tools.

Data sharing among tools has been hampered by a feast-or-famine approach to interchange standards. Some are extremely low-level standards that fail to encode structure, like files of ASCII text. Others are extremely high-level, like CDIF models of particular domains. These have the disadvantage that they cover narrow domains and do not deal well with information that protrudes into more than one domain, and that there are few tools to assist with their construction. With one major class of exceptions[4], therefore, software tool prototype development has ignored interchange issues.

XML promises to provide a mid-level standard that can enable the creation of more flexibly-deployed and reusable software tools. However, this can be achieved only if the software community carefully picks among the XML claims to find more reality-based solutions.

[3] Another article in this volume discusses the importance of XML to software engineering on the Internet [11].

[4] The programming languages themselves provide the major class of exceptions. The need to interchange programs between tools and humans has dictated an agreed-upon, high-level structural interchange format, the language's syntax. The need to convert back-and-forth between internal formats and this interchange format has forced the existence a tool suite for parsing and unparsing which then simplifies adaptation to the interchange format.

Rather than expecting XML to solve data integration problems by enabling a new generation of high-level standards, we should look to the fact that the adoption of XML files rather than ASCII files as an interchange format allows content-model mismatches to be dealt with more expeditiously, through the use of transformation tools. Such tools can examine definitions of different XML models (e.g., DTDs or XML Schemas), and, with user assistance, detect related elements, reconcile differences, and generate transformers (e.g., in XSL). These transformers can then be used to reshape the XML produced by one tool for consumption by another. The use of XML as a common substrate has the significant advantage that it separates the technological issue of reshaping content both from the details of specific tools and from the burden of adapting to a wide variety of basic representations at the same time.

The commercial adoption of XML has provided tools that parse, unparse, and transform XML representations. The availability of parsing and unparsing tools reduces the cost of adoption to the point where all new tools should deal with their one-way information flows (e.g., their input and output, except, of course, communication with users through user interfaces) in XML format. This brings the tools closer to being stem-software since their derivative forms have the opportunity of reshaping the XML as needed in different environments.

Technologies for solving the problems of reshaping also enable the occasional application of XML to omnidirectional sharing situations.

Short of the repository-based approaches discussed in the next section, omnidirectional sharing can be accomplished with one-way sharing mechanisms by keeping tools' inputs and outputs as stored documents that can be used or created by other tools. These documents constitute a form of coarse-grained repository for tools without a common data model. Use of the repository is augmented by transformation tools and notification mechanisms that allow it to reshape information to fit other tools' needs and to awaken them on demand. The same considerations that make XML an important approach to unidirectional sharing therefore facilitate this model for omnidirectional sharing. Omnidirectional sharing through a reshaping and notification coordinator is in use in at least one advanced system available commercially today. Adapting unidirectional technologies for omnidirectional use requires more careful use of XML than is common, because of the need to preserve identity in round-tripping. For example, a program element identified to the developer by its name should generally (but not always) be identified in related XML elements by a unique identifier because the name could be changed by a renaming not seen by all the tools.

4.1.2 Data Integration and Enterprise Java Beans

XML is appropriate for very coarse-grained integration characterized by loose coupling and low bandwidth. Tighter coupling, or requirements for higher bandwidth, requires the use of a shared repository.

Traditional designs for repositories have required that participating tools use a common data model. To remove this constraint, morphogenic software needs to contain an isolation layer to insulate the tool's model from the repository model. To date, software tool developers have therefore eschewed morphogenic software in favor of rapid development.

Confronted with the same need faced by SEE designers to deploy software with different internal models against a common repository of information, the commercial software community has promoted and adopted the Enterprise Java Beans™ (EJB) technology [55].

Enterprise Java Beans provide a style for Java programs that separates the concerns of an object's algorithms ("the business logic") from its data storage and access concerns. The software is written in a stem form and is subject to maturing through a process called *deployment.* Deployment provides implementations for the operations that the stem uses to access its data, and for points at which calls to other objects may be intercepted. These implementations support the tailored choices made regarding data storage, verify authority for access, and participate in concurrency and transaction management protocols. The implementations may be created either by developers directly or by tools that support deployment. However, as is the case with XML, the current state of EJB technology suffers from the fact that it was developed without regard to the performance issues of interest in much tool work, and that much of its support relies on an underlying presumption that information is located by untyped keys rather than typed pointers. So, while one might hope to advocate that all new tools should deal with their omnidirectional information flows (e.g. their persistent storage) by writing them as EJB's, this requires the availability of a free, lightweight deployment tool for simple situations and the development of extensions that support more pointer-like keys.

4.1.3 Control Integration and Java Messaging Services

Event-based message delivery is at the heart of many control-flow integration backplanes, and that fact is now affecting commercial software infrastructures as well as software development infrastructures. Capitalizing on the Java success, a messaging model called Java Messaging Services (JMS) [33] has been put forward. JMS has several severe limitations that inhibit its use as a promoter of morphogenic software.

Morphogenic software emphasizes the separation from stem-software of unrelated concerns. JMS, however, re-

quires the sender of a message to be cognizant of those elements of the contents that are used to determine who should receive it. The sender must explicitly include these as properties of the message in spite of the fact that they are contained within the message as well. In addition, both the sender and receiver of a message must be concerned with the categorization of messages into *topics* that are used to describe the flow of the messages. Both of these concerns are clearly separate from the functional concerns of the sending and receiving objects.

These characteristics argue for the use of a stronger delivery model: message brokering.

4.1.4 Control Integration and Message Brokering

The message broker model is a generalized delivery model supporting the routing of messages, from the sender to one or more receivers. The delivery criteria are treated as a separate concern from the processing concerns of either. The message broker extracts properties from the message and uses those properties in the light of delivery rules to determine the recipients. As with the EJB technology, this process is characterized in the deployment of software, rather than in its development in stem form. Instead of being internal to a tool, the process of matching receivers to senders and consumers to providers is performed when the software is placed with other software elements by providing dispatch or delivery rules that may depend on message contents.

While both JMS and most commercial message brokers emphasize event-based messaging, that model is not suitable for most software. The bulk of the software is written, in programming languages like Java™ or COBOL, to expect the response to a request for service (a method or subroutine call) to be sufficiently forthcoming that no provision need be made for delay or transaction boundary crossing. There is, however, no good reason why the same model for separating delivery concerns from processing concerns should not be applied to the normal request/response idiom. In making this application, we should be careful that software elements make no more than the necessary presumptions about the way they will mature in deployment. For example, function consumers (senders or callers) should not insist on delivery to single servers or to servers who are interested only in the fact that the request was made.

With respect to this area of integration, new tools should interface with the components they call and that call them through open delivery mechanisms. The use of open delivery mechanisms separates the delivery specification, e.g. that delivery will look at and only at the class of one particular parameter, from the tools. This allows delivery to components other than the ones the tool builder envisaged. The ideal is to use a language implementation that allows brokering of all calls, such as a Java interpreter or C++ compiler that is open with respect to dispatch semantics. Although uncomplicated in the light of existing multiple-dispatch technologies, such interpreters have yet to be written. They have the advantage that they allow tool builders to code calls to other components in the same manner as calls to internal functions.

4.1.5 Control Integration and Composition Engines

Separate specification of deployment characteristics from stem characteristics requires languages and conceptual models for these specifications. Software composition engines, like GenVoca [7] and the subject-oriented programming compositor [44], employ directives for composing components that are separate from the components themselves, treating all the software components as stem-form software. Compositors like these raise to consciousness the need for investigation of tools that describe rules for how messages flow, how software is combined, how needs are matched with provisions, and how consistency-checking is performed. If the software is to become capable of morphogenesis, these issues cannot be left as the province of individual programming languages, but must be treated as concerns separate from the function of the individual pieces of software.

4.2 Management of Concerns

Support for full multi-dimensional separation of concerns, or for variants or subsets of it that allow separation according to more than one criterion at a time, requires tool and environment support, and suggests the need for new methodologies and environments that best leverage it. Some tools have begun to appear (e.g., [6], [27], [37], [39], [45]), but many more are needed, and research into appropriate methodologies and processes will be required before truly effective environment support can be provided. Given the growing awareness of the problem of the tyranny of the dominant decomposition and the interest in providing mechanisms that break it, we expect real progress on such tools over the next decade.

At present, separation of concerns is predominantly an early lifecycle activity; it occurs during requirements engineering, design, and coding. Once chosen, the concern structure becomes relatively fixed; most changes require system refactoring and rearchitecting. In actuality, however, new concerns become relevant throughout the software lifecycle. Thus, tool and environment support will be needed to facilitate separation of concerns as an activity that occurs on an ongoing basis, as new concerns are identified or become relevant.

Separation of concerns itself is really a process that involves several activities—*identification*, *encapsulation*, and *integration* of concerns—all of which require tool support. Some key challenges in these areas are described below.

Identification of concerns: Identification of concerns is the process of determining the set of relevant concerns, the

units of software that address them, and their interrelationships. Identification can occur either "up-front" or "in retrospect." When developers identify concerns up-front, they explicitly design one or more software artifacts so as to encode the concern structure in the software structure. For example, a requirements engineer may be concerned about *features* of the software, and therefore organize the requirements by feature concerns; similarly, a Java™ programmer is concerned about *classes*, and so separates the implementation into a set of class concerns. Many types of interrelationships among concerns are common, and these are typically defined as part of the up-front design of the concerns.

New concerns may become important as the software lifecycle progresses. Thus, developers may wish to identify new concerns in retrospect, indicating how fragments of existing software artifacts address the new concerns, and how the new concerns relate to each other and to the existing concerns. For example, the Java™ programmer may need to talk to the requirements engineer; doing so, s/he must be able to discuss the system features. To identify features as new concerns within the implementation code, the developer must identify all of the classes, methods, and member variables that address (i.e., are part of) each of the features; for example, all `displayObject`() methods are part of the "print" feature.

Tool support for identification of concerns must permit both up-front and in-retrospect identification. Identification may be done *extensionally* or *intentionally*, and finding convenient and effective ways to identify concerns, particularly intentionally, is an important research challenge in this area. The ability to identify concerns that span multiple artifacts is also very important. For example, a developer whose concern is the "print feature" may care about *all* of the artifact fragments associated with that feature: the requirements specification, the design elements, the implementation code, the test cases, etc. The ability to identify these "cross-lifecycle" concerns promotes traceability and round-tripping, both of which continue to be crucial.

Identification of concerns and their interrelationships must be done *consistently*, in such a way as to ensure that stated semantic properties are maintained. As a simple example, if the method `displayObject`() references a member variable `_display` in a given class, then if `displayObject`() is designated as addressing the "print" feature (as well as the "Object" class concern), `_display` must also be part of the "print" feature. In general, ensuring that concerns are consistent and that they satisfy their specifications is an important issue—particularly so in the presence of overlapping (non-orthogonal) concerns.

Visualization of software in terms of different kinds of concerns that have been identified is a critical capability. For identification of concerns to result in reduced complexity, it must be possible for developers to focus on certain specific concerns while ignoring others. It must also be possible for developers to see and understand how concerns relate and interact, to assess and control impact of change and to facilitate extraction. Significant research is needed to determine how best to structure multi-dimensional concern structures to promote comprehension, how developers will need to navigate through them, and how to visualize them.

Encapsulation of concerns: Identification of concerns indicates how concerns impact software, but it does not itself ensure that the concerns are encapsulated. Modularization mechanisms are needed to enforce encapsulation of concerns. These mechanisms may be part of a given formalism, as classes are in object-oriented formalisms, but it may also be the case that a particular concern cannot be modularized in a given formalism When a concern spans multiple formalisms or when a particular kind of concern cannot be modularized in a given formalism (as features cannot be encapsulated in Java™) extra-formalism modularization mechanisms are required. Additional research will be required to define both new artifact formalisms that better support identification and encapsulation of multiple dimensions of concerns, and extra-formalism modularization mechanisms that can be used to achieve additional decompositions along other dimensions.

Integration of concerns: Any separation of concerns mechanism must include powerful integration mechanisms, to permit the selective integration of the separate concerns. Integration, a key theme throughout the history of software engineering, requires much tool and environment support. It requires the definition of compositor [44] or weaver [37] tools, which combine understanding of the underlying artifact formalisms with semantic sophistication, to synthesize integrated concerns that satisfy stated specifications and interrelationships. They also need to perform "matching" and "reconciliation," examining the (interfaces of the) concerns to be integrated, and interacting with the developer to match up corresponding elements and reconcile differences between them.

Summary: Fully achieving the goal of multi-dimensional separation of concerns will require both new research and the synthesis of existing research in a variety of software engineering and other areas. These include software architecture [26], refactoring and reengineering, component-based software engineering, software analysis, software specification, methodologies, programming (and other) languages, HCI, and visualization.

4.3 Environments for Morphogenic Software

Morphogenic software is an extremely ambitious goal, and it will have to be realized incrementally. It involves research on the evolutionary pressures facing people, organizations, and their software, and on how the people, organizations and software must respond to those pressures as a

unit. As such, it is an inherently multi-disciplinary area of research, with software engineering tools and environments playing a key role.

Support for separation and integration of concerns is critical to achieving the goal of morphogenic software. In particular, the ability to identify portions of software that realize a given concern, to extract it as a first-class component and then later integrate it into a new context, are fundamental requirements for morphogenesis. This section explores some other kinds of tool and environment support needed to facilitate the engineering of morphogenic software.

Managing Dependencies and Interactions. It has been accepted for a long time that one of the key reasons why software is too rigid to be readily adaptable is that it contains many hidden assumptions, dependencies and interactions. Perhaps these were understood by the original developers, but they are usually not specified or even documented, and are therefore not understood for long. Even small changes can therefore have unexpected effects, which can be difficult to detect and correct. It has long been hoped that tools would help significantly with these problems, but only recently have sufficient advances been made for this to be realistic. Tools can encourage isolation in a natural way, as discussed in Section 4.2. They can ease the capture of useful information at the time of initial development and evolution; the challenge here is to do so with minimum intrusion, or the immediate market pressures will render this unacceptable. They can discover useful information in retrospect, on demand. This might involve analysis, both static and dynamic, to detect dependencies and interactions, and perhaps even assumptions [20]. They can present visualizations of the software to manifest and help developers to understand assumptions, dependencies and interactions.

Adaptation. Even software that has been successfully separated and extracted from the context in which it operates cannot usually be used as-is in new contexts. It must be adapted to the new needs before it can be integrated with other software in the new context. Traditional adaptation mechanisms are inadequate to the task; to the extent adaptation is done at all at present, it is done largely by hand Work on packaging mismatch is beginning to address some of these issues [18]. Comprehensive tool support is sorely needed.

The details differ widely depending on how much of the base software is visible and modifiable. A *black-box* approach requires that the base software, and its interfaces (including details of data produced or consumed), be left alone. Tool support is therefore needed to compare actual and desired interfaces, to reconcile differences between them, and to produce adapters allowing the actual interfaces to work in the new context. A *white-box* approach allows modification of the base software. Tool support is needed to help determine what modification is necessary, and to isolate the modifications from the base software, using separation of concerns techniques (Section 4.2).

Correctness and Consistency Management. Assuming that one has adapted a portion of software and integrated it into a new context, how can one be confident that the result is correct and consistent? Tool support is needed to perform both checking of various sorts, and testing. Checking tools would need to be based on some specification of what is intended; if not a full specification, at least some properties that are to be checked, as is done with model checking [14]. They are also likely to require characterizations of the software components, and of the manner in which they are adapted and integrated. Since, given our assumptions about initial software development, we cannot expect detailed characterizations of software to be provided, tools are needed to construct them on demand, probably using a combination of program analysis and developer interaction. A key challenge is making this work in such a way that the benefits clearly outweigh the costs.

Summary. We have discussed some key tools needed to support morphogenic software. There are many more, such as tools to help locate software for adaptation and reuse, perhaps using such techniques as concept extraction, data mining and sophisticated search.

The vision of morphogenic software can thus provide a focus for research on a wide variety of tools, with the ultimate goal of an environment for development and evolution of morphogenic software. Long before this goal is reached, however, one can expect significant benefit from individual tools that deal even partially with some of the problems discussed above. As we have seen, some of these have already begun to appear.

5. NEW DOMAINS

In the prior section we discussed some key issues applying to important research directions for software engineering in general over the next decade. In this section we consider some emerging new domains that we believe will require non-traditional software engineering methodologies, tools, and environments. These new domains impose some extremely challenging requirements and constraints on the solutions, in addition to those discussed earlier, and we expect the tools and technologies that address these domains to expand the frontiers of modern software engineering.

5.1 Pervasive Computing

Not long after standard computers became sufficiently powerful that software engineers, for the most part, stopped worrying about saving a few bytes or cycles, the advent and expansion of pervasive computing has re-introduced the need for software engineering for constrained devices. In doing so, it has also introduced some new challenges, including mobility [50], far greater heterogeneity of devices,

constant flux in the set of devices available, the desire for the same applications to work on multiple devices, and the need for software integration across devices.

Most work to date on constrained devices has been in the area of embedded systems. Tools, running on standard computers, support development of software for specific devices. Developers use such tools to develop applications specifically for a particular device and operating system. Retargeting an application to a new device, and field updates, in which new or updated applications are installed on existing embedded devices, are both sizeable jobs.

The pervasive computing world is rather different. It is more a case of bringing common services to a multiplicity of devices than of developing custom applications for specific devices. There is strong economic pressure for a service provider to support a wide variety of devices, and not by rewriting the service for each of them; devices come and go with sufficient rapidity that custom coding is not usually viable. And yet, devices have significantly different characteristics, so the services must at least be customized. This creates a tension, and the need for software engineering approaches and tools to assist in addressing it.

The biggest problem with constrained devices is the limitation on computing resources imposed by constraints on power. The second biggest problem is usability. User interaction is always impoverished, and its details differ significantly from device to device. Supporting a service on a large number of devices by using only their common capabilities is not viable; it is essential to exploit the particular strengths of each device to get usability that is acceptable at all. This leads to the need for *transcoding* [31], and tools to support it.

Transcoding is the process of adapting content for presentation and interaction in different (constrained) environments. It might involve adaptation of legacy content, such as databases, through techniques like data mining and web page presentation. Better effects can be achieved with content structured with transcoding in mind, so that the structure, and perhaps additional information such as annotations, can be used in the transcoding process. Tool support is vital to present users with content and device models, and allow them to specify easily how the transcoding should be done. Variants of this support are needed in the development environment and also on the device itself, to allow end-user tailoring. Obviously, the tool running on the device will be much different than that running in the development environment, as it must itself address the issue of usability in the constrained setting.

Clearly, cross-development environments are critical in this domain, as in the case of embedded systems: compilers, debuggers, profiling and performance tuning tools, testing tools. A new twist is provided by the fact that support for multiple targets is critical, so that applications can be developed once and deployed, with customization but without major redevelopment, on many kinds of devices. The devices differ not just in hardware, but also in operating systems and application models. Applications are not typically portable across application models; this is a significant change of context, requiring excellent separation of concerns and morphogenesis.

5.2 e-Commerce

The burgeoning e-commerce domain is sufficiently different from traditional software domains that many software engineering issues must be rethought. There is far greater emphasis than before on COTS software and on black-box reuse, versus on writing and evolving large software systems owned by a single company. Indeed, the components being reused, and the data they use, are often not even on machines one owns or controls. There is no global view or control, but rather many largely independent systems that must interact. They all evolve, again without central control, and the evolution often involves interfaces, not just internal details. This evolution occurs dynamically, and the commerce must continue despite it. There are growing demands for continuous availability, even in the face of surges in usage, some of which have already been observed to be an order of magnitude larger than “normal” usage. Because of the exchange of money and sensitive information, security and reliability are often critical.

Many of these issues are not new to e-commerce, and have arisen in distributed systems, agent systems, highly reliable systems, software process modeling and execution, etc. However, the e-commerce domain brings them all together in a context of major and growing importance, and a context of unprecedented scale. E-commerce software is being developed, and rapidly, because this is one of the domains in which being early to market is important, yet without the benefit of a large body of appropriate software engineering research results or experience. There is a significant challenge for the software engineering research community to come up with architectures, models, methodologies, approaches and processes, and the tools and environments that embody them, in order to facilitate software engineering and improve software quality in this domain.

Another area where tools are desperately needed is management of scale. Presentation, visualization and management of complexity has long been an important theme in tool development, and one that has met with limited success. In the e-commerce arena, the numbers go way beyond anything such tools have dealt with before: e.g., on the order of 10^8 people on the Internet, and 10^{11} agents running on their behalf; even a single auction system running 2 million auctions simultaneously. Not all these interact at once, of course, but the interactions are unpredictable, and surges are common (and currently are often catastrophic). What

design and testing approaches and tools are needed for software with these properties?

Within many e-commerce applications, there is great variability in requirements for reliability, availability and security in different parts of the system. The properties are critical in some parts of a system, whereas high throughput is much more important in others. For example, users are quite tolerant (at least at present) of minor errors or occasional disconnects when browsing a catalogue, as long as they can usually do so quickly and conveniently, but when transferring money to purchase items, they demand reliability and security. What approaches and tool support are necessary to build systems in which these different tradeoffs co-exist, ensuring that the barriers are properly maintained?

Most e-commerce applications are an integration of underlying business rules and a user experience, which involves graphical and often multi-media presentation, and is critical to acceptance of the application. How can tools help developers to create the user experience, and to achieve this integration effectively, especially in the face of evolution of both aspects? This goes considerably beyond traditional user interfaces, and their separation from underlying functionality.

Software evolution in this large-scale, uncontrolled domain is especially challenging. As noted above, interactions among agents, systems or components are often unpredictable, and yet any of the interacting parties might evolve independently. Morphogenic software, and especially the consistency and inconsistency management parts of it, have an important role here, because, in a sense, context is constantly changing, and software must adapt in a consistent manner. In this domain, adaptation is even more challenging because it must often be dynamic, so as not to disrupt ongoing commerce. Since most e-commerce systems involve interaction with software one does not own or control, and often does not have installed on one's own machines, detection of and adaptation to interface changes becomes particularly important. How to engineer software where this is even possible, and providing tool support for doing so and for accomplishing the needed adaptation, are challenging areas of research.

In the past, the software engineering community has focussed primarily on development and evolution of large, complex, expensive, centrally-owned systems. This is still an important problem, but to a relatively small number of large companies. There is now a tremendous opportunity to focus on the software engineering challenges of e-commerce, helping to ensure that software quality and capabilities do meet the demanding needs of this domain.

6. CONCLUSIONS

Throughout the history of tools and environments in software engineering, we have seen some important issues and trends emerge. These include separation of concerns, integration and coordination, "plug and play" and support for multiple views. We have also seen numerous attempts to address these issues as they have manifested themselves in different contexts, each imposing unique requirements and constraints on solutions. These attempts have shaped the history of the field.

Many of the key challenges we see at present and in the future for software engineering tools and environments also pertain to these common themes. The modern software development milieu, however, renders them broader and deeper, with ever more complex requirements and constraints on solutions. Addressing them adequately will necessitate not only pushing the frontiers of the state of the art in these areas, but also reexamining and redefining many of the assumptions, methodologies, and technologies on which the field currently rests.

We expect to see several major trends emerge in the future:

- New methodologies, formalisms, and processes to address non-traditional software lifecycles, and the tool and environment support to facilitate them. With significant economic, business, and research pressures to be first to market or first to publish, especially in new domains like pervasive computing and e-commerce, fewer individuals are able to expend the resources on a traditional spiral lifecycle, even knowing what they are losing as a result of cutting corners. Further, with rapid development, rapidly changing requirements and platforms, and an extremely fast-moving and volatile field, it is questionable whether many new domains would actually benefit from the more costly traditional software lifecycle—it simply takes too long. Current methodologies and tools generally do not work well in such non-traditional contexts, and they do not "degrade" gracefully. Developing methodologies and models that do work in these contexts, and tool and environment support for them, is a critical challenge of the upcoming decade.
- Greater focus on, and methodology, formalism, tool and environment support for, separation of concerns and morphogenic software. The problem of separate islands of useful functionality that are seldom usable in the contexts in which they are needed, is becoming ever more acute. At the same time, advances in both separation of concerns research and in integration research and standards provide a glimmer of hope that functionality and context can be separated and integrated more effectively, leading towards context independence and hence greater adaptability.
- The adoption or adaptation of XML, Enterprise Java Beans and sophisticated message brokering for integration of both tools and commercial applications. We believe that these (or similar) technologies have a

chance of successful adoption, where prior standards have failed, because they are standards at a different level. They do not mandate specific models or require specific implementations of specific sets of services, like integration platforms of the past. They provide less function, but are more likely points of agreement and standardization, in much the same way that procedure call has been in the programming language world.

The 1990's saw a great deal of work on process-centered software engineering environments, and the development of many kinds of tools that facilitate different software engineering activities. We expect that the need for effective PSEEs and diverse tools will increase, particularly as the variety of challenging software domains and software engineering models—traditional and non-traditional—increase. Our hope is that research in these areas will occur along with improvements in adaptability and integration, so that the new tools and environments will not be yet more isolated islands, but integral parts of a growing whole.

ACKNOWLEDGEMENTS

We thank Stu Feldman and Michael Karasick for helpful discussions and valuable insights on e-commerce and pervasive computing, respectively, Stan Sutton for helpful comments on drafts, and Tappan Parikh for assistance with the literature search.

REFERENCES

[1] M.Aksit, L.Bergmans, S.Vural. "An object-oriented language-database integration model: The composition filters approach." In Proceedings ECOOP'92.

[2] V. Ambriola, P. Ciamcarini, and C. Montangero. "Software Process Enactment in Oikos." Proceedings of the Fourth ACM SIGSOFT Symposium on Practical Software Development Environments, December 1990.

[3] E.P. Andersen and T. Reenskaug. "System Design by Composing Structures of Interacting Objects." Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 1992.

[4] Robert A. Balance, Susan L. Graham and Michael L. Van deVanter. "The Pan Language-Based Editing System for Integrated Development Environments." Proceedings of SIGSOFT'88: Fourth Symposium on Software Development Environments, December 1990.

[5] Sergio Bandinelli, Alfonso Fuggetta, Carlo Ghezzi, and Luigi Lavazza. "SPADE: An Environment for Software Process Analysis, Design, and Enactment." In *Software Process Modelling and Technology* (A. Finkelstein, J. Kramer, and B. Nusibeh, editors), John Wiley & Sons Inc., 1994.

[6] Elisa L.A. Baniassad and Gail C. Murphy. "Conceptual Module Querying for Software Reengineering." In Proceedings of the International Conference on Software Engineering (ICSE 20), April 1998.

[7] Don S. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci and M. Sirkin. "The GenVoca Model of Software-System Generators." *IEEE Software*, September 1994.

[8] Noureddine Belkhatir, Jacky Estublier, and Melo L. Walcelio. "ADELE-TEMPO: An Environment to Support Process Modeling and Enaction." In *Software Process Modelling and Technology* (A. Finkelstein, J. Kramer, and B. Nusibeh, editors), John Wiley & Sons Inc., 1994.

[9] Israel Ben-Shaul and Gail Kaiser. "A Paradigm for Decentralized Process Modeling and its Realization in the Oz Environment." Proceedings of the 16th International Conference on Software Engineering, 1994.

[10] Barry W. Boehm. "A Spiral Model of Software Development and Enhancement." In *IEEE Computer*, vol. 21, no. 5, May 1988.

[11] Luca Bompani, Paolo Ciancarini and Fabio Vitali. *Software Engineering on the Internet: a roadmap*. In this volume.

[12] Gerard Boudier, Ferdinando Gallo, Regis Minot, and Ian Thomas. "An Overview of PCTE and PCTE+." Proceedings of SIGSOFT'88: Third Symposium on Software Development Environments, November 1988.

[13] Gerome Canals, Nacer Boudjlida, Jean-Claude Derniame, Cladue Godart, and Jaques Lonchamp. "ALF: A Framework for Building Process-Centred Software Engineering Environments." In *Software Process Modelling and Technology* (A. Finkelstein, J. Kramer, and B. Nusibeh, editors), John Wiley & Sons Inc., 1994.

[14] Edmund M. Clarke, David E. Long, and K.L. McMillan. "Compositional Model Checking." Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science, 1989.

[15] Geoffrey M. Clemm and Leon J. Osterweil. "A Mechanism for Environment Integration." In *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 1-25, January 1990.

[16] R. Conradi, E. Osjord, P.H. Westby, and C. Liu. "Initial Software Process Management in EPOS." In *Software Engineering Journal* (Special Issue on Software Process and its Support), vol. 6, September 1991.

[17] Wolfgang Deiters and Volker Gruhn. "Managing Software Processes in the Environment MELMAC." Proceedings of the Fourth Symposium on Software Development Environments, December 1990.

[18] Robert DeLine. “Avoiding Packaging Mismatch with Flexible Packaging.” In *Proc. International Conf. On Software Engineering*, 1999.

[19] Wolfgang Emmerich, Petr Kroha, Wilhelm Schäfer. “Object-Oriented Database Management Systems for Construction of CASE Environments.” Proceedings of the Conference on Database and Expert Systems Applications (DEXA’93), 1993.

[20] Michael D. Ernst, Jake Cockrell, William G. Griswold and David Notkin. “Dynamically Discovering Likely Program Invariants to Support Program Evolution.” In *Proc. International Conf. On Software Engineering*, 1999.

[21] Stuart I. Feldman. “MAKE—A Program for Maintaining Computer Programs.” In *Software—Practice and Experience*, vol. 9, no. 4, pp. 255-265, April 1979.

[22] Alfonso Fugetta. *Software Process: a roadmap*. In this volume.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. “Design Patterns: Elements of Reusable Object-Oriented Software.” Addison-Wesley, 1994.

[24] David Garlan, Robert Allen and John Ockerbloom. “Architectural mismatch, or Why it’s hard to build systems out of existing parts.” In *Proc. International Conf. On Software Engineering*, 1995.

[25] David Garlan, Gail E. Kaiser and David Notkin. “Using Tool Abstraction to Compose Systems.” IEEE Computer 25(6), pages 30–38, June 1992.

[26] David Garlan and Mary Shaw. “Software Architecture: Perspectives on an Emerging Discipline.” Prentice-Hall, April 1996.

[27] William G. Griswold, Yoshikiyo Kato, Jimmy J. Yuan. “Aspect Browser: Tool Support for Managing Dispersed Aspects.” Position paper, OOPSLA ’99 Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems, http://www.cs.ubc.ca/-~murphy/multd-workshop-oopsla99.

[28] A. Nico Habermann and David Notkin. “Gandalf: Software Development Environments.” In *IEEE Transactions on Software Engineering*, vol. SE-12, no. 12, pp. 1117-1127, December 1986.

[29] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. “STATEMATE: A Working Environment for the Development of Complex Reactive Systems.” In *Transactions on Software Engineering*, vol. 16, no. 4, pp. 403-414, April 1990.

[30] William Harrison and Harold Ossher. “Subject-oriented programming (a critique of pure objects)." In Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA), September 1993.

[31] Masahiro Hori, Kohichi Ono , Goh Kondoh and Sandeep Singhal. “Authoring Tool for Web Content Transcoding.” In *Proc. Markup Technologies ’99*, Graphic Communications Association, Alexandria, VA, December 1999.

[32] M. Jackson. Some complexities in computer-based systems and their implications for system development. In *Proceedings of the International Conference on Computer Systems and Software Engineering*, pages 344–351, 1990.

[33] Java™ Message Service Documentation, Sun Microsystems, Inc. (http://java.sun.com/products/jms/docs.html)

[34] G. Junkermann, B. Peuschel, W. Schäfer, and S. Wolf. “MERLIN: Supporting Cooperation in Software Development Through a Knowledge-Based Environment.” In *Software Process Modelling and Technology* (A. Finkelstein, J. Kramer, and B. Nusibeh, editors), John Wiley & Sons Inc., 1994.

[35] Gail E. Kaiser, Peter H. Feiler, and Steven S. Popovich. “Intelligent Assistance for Software Development and Maintenance.” In *IEEE Software*, May 1988.

[36] B. W. Kernighan and J. R. Mashey. “The Unix Programming Environment.” In *IEEE Computer*, vol. 14, no. 4, pp. 12-24, April 1981.

[37] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. “Aspect-Oriented Programming.” In proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.

[38] Nancy G. Leveson and Peter R. Harvey. “Analyzing Software Safety.” In *IEEE Transactions on Software Engineering*, vol. 9, no. 5, September 1983.

[39] Mira Mezini and Karl Lieberherr, “Adaptive Plug-and-Play Components for Evolutionary Software Development.” In Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA), October 1998.

[40] Hausi A. Müller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley and Kenny Wong. *Reverse Engineering: a roadmap*. In this volume.

[41] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. “A Framework for Expressing the Relationships Between Multiple Views in Requirements Specifications.” In *Transactions on Software Engineering*, vol. 20, no. 10, pp. 260-773, October 1994.

[42] William J. Opdyke and Ralph E. Johnson. “Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems.” SOOPPA Conference Proceedings, ACM Press, September 1990.

[43] Harold Ossher and William Harrison. “Support for change in RPDE^3.” Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments, December 1990.

[44] Harold Ossher, Matthew Kaplan, Alexander Katz, William Harrison, and Vincent Kruskal. “Specifying Subject-Oriented Composition.” In *Theory and Practice of Object Systems*, vol. 2, no. 3, pp 179-202, 1996.

[45] Harold Ossher and Peri Tarr. “Multi-Dimensional Separation of Concerns and the Hyperspace Approach.” *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000. (To appear.)

[46] Leon J. Osterweil. “Software Processes are Software Too.” Proceedings of the 9th International Conference on Software Engineering, March 1987.

[47] David L. Parnas. “On the Criteria To Be Used in Decomposing Systems into Modules.” *Communications of the ACM*, vol. 15, no. 12, December 1972.

[48] Terry Quatrani and Grady Booch. *Visual Modeling with Rational Rose 2000 and UML*. Addison-Wesley, 1999.

[49] Steven P. Reiss. “Connecting Tools Using Message Passing in the FIELD Environment.” In *IEEE Software*, vol. 7, no. 4, July 1990.

[50] Gruia-Catalin Roman, Amy L. Murphy and Gian Pietro Picco. *Software Engineering for Mobility: a roadmap*. In this volume.

[51] W.W. Royce. “Managing the Development of Large Software Systems.” Proceedings IEEE WESCON, August 1970. (Reprinted in Proceedings Ninth International Conference on Software Engineering.)

[52] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. “N Degrees of Separation: Multi-Dimensional Separation of Concerns.” Proceedings of the 21st International Conference on Software Engineering, May 1999.

[53] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon J. Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michal Young. “Foundations for the Arcadia Environment Architecture.” Proceedings of SIGSOFT’88: Third Symposium on Software Development Environments, November 1988.

[54] T. Teitelbaum and T.R Reps. “The Cornell Program Synthesizer: A Syntax Directed Programming Environment.” In *Communications of the ACM*, vol. 24, no. 9, pp. 563-573, September 1981.

[55] Anne Thomas. “Enterprise JavaBeans™ Technology, Server Component Model for the Java™.” December 1998. (Prepared for Sun Microsystems, Inc.; http://java.sun.com/products/ejb/white_paper.html)

[56] C. R. Turner, A. Fuggetta, L. Lavazza and A. L. Wolf. Feature Engineering. In *Proceedings of the 9th International Workshop on Software Specification and Design*, 162–164, April, 1998.

[57] Robert J. Walker, Elisa L.A. Baniassad, and Gail C. Murphy. “An Initial Assessment of Aspect-oriented Programming." In Proceedings of the International Conference on Software Engineering (ICSE 21), May, 1999.

[58] A.I. Wasserman, P.A. Pircher, D.T. Shewmake, and M.L. Kersten. “Developing Interactive Information Systems with the User Software Engineering Methodology.” In *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, pp. 326-345, February 1986.

[59] Rebecca Wirfs-Brock and Ralph Johnson. “Current Research in Object-Oriented Design.” In *Communications of the ACM*, pp. 104-124, September 1990.

[60] Extensible Markup Language (XML) 1.0, W3C Recommendation 10 February 1998, Document REC-xml-19980210, W3C Publications (http://www.w3.org/TR/REC-html).