Overview of Software Development Environments

Susan A. Dart, Robert J. Ellison, Peter H. Feiler, and A. Nico Habermann
Edited by Peter Fritzson

Overview

1 Introduction

Environment refers to the collection of hardware and software tools a system developer uses to build software systems. As technology improves and user expectations grow, an environment's functionality tends to change. Over the last 20 years the set of software tools available to developers has expanded considerably.

We can illustrate this change by observing some distinctions in the terminology. *Programming environment* and *software development environment* are often used synonymously, but here we will make a distinction between the two. By "programming environment" we mean an environment that supports only the coding phase of the software development cycle—that is, programming-in-the-small tasks such as editing and compiling. By *software development environmen*t we mean an environment that augments or automates the activities comprising the software development cycle, including programming-in-the-large tasks such as configuration management and programming-in-the-many tasks such as project and team management. We also mean an environment that supports largescale, long-term maintenance of software.

The evolution of environments also demands that we distinguish basic operating system facilities—fundamental services such as memory, data, and multipleprogram management—from the enhanced functionality that characterizes state-of-the-art environments. This enhanced functionality is typically achieved through tools such as browsers, window managers, configuration managers, and task managers. In a sense, environments have evolved in concert with the software engineering community's understanding of the tasks involved in the development of software systems.

To better understand the technological trends that have produced state-of-the-art environments, we here present a taxonomy of these trends. We cite examples of research and commercial systems within each class. We intend the taxonomy to show the description of the trends and to suggest where more work needs to be done.

The taxonomy comprises four categories, each representing trends having a significant impact on environments—on their tools, user interfaces, and architectures. The four categories are:

- Language-centered environments

These are built around one language, thereby providing a tool set suited to that language. These environments are highly interactive and offer limited support for programming-in-the-large.

- Structure-oriented environments

These incorporate techniques that allow the user to manipulate structures directly. The language independence of the techniques led to the notion of generators for environments.

- Toolkit environments

These provide a collection of tools that includes language-independent support for programming-in-the-large tasks such as configuration management and version control. There is little, if any, environment-defined control and management of tool usage.

- Method-based environments

These incorporate support for a broad range of activities in the software development process, including tasks such as team and project management (programming-in-the-many). These environments also incorporate tools for particular specification and design methods

We could discuss the trends from several perspectives. For example, we could take a tool builder's perspective, focusing on techniques for tool integration. We could take an expert system builder's perspective, focusing on the automation of the software development process by means of a programmer's assistant that uses knowledge-based concepts. However, we discuss the trends from the user's perspective; that is, we examine how the trends affect the user's perception of, and interaction with, an environment.

User requirements for environments cover a broad spectrum. The functionality of environments includes support for a single user for programming-in-the-small, coordination and management of multiple users for programming-in-the-large, and management of the software development cycle. The nature of the user interface is of considerable importance. Undoubtedly, the user of an environment needs to be able to customize it, either by tailoring or extending a particular tool or by creating specialized tools via generation facilities. To support this, the environment must be implemented so as to allow tools to be easily integrated into it. The user also needs facilities to support incremental development of software to aid prototyping. In essence, the user requires support for the entire software development cycle—from specification through coding to maintenance—including the ability to trace information across phases. This spectrum of needs is addressed across all the categories of four taxonomy, though no single category deals with them all. We do not attempt to survey all existing environments nor do we provide detailed descriptions or evaluations of

them. Neither do we advocate any particular environment. In fact, because users have varying levels of expertise, different application requirements, and different hardware, no single environment can satisfy all users.

The significance of our taxonomy is in its clarification of trends rather than in categorization of particular environments. A particular environment may fit into a number of categories. These categories do not represent competing viewpoints; instead, they represent areas of effort that have provided fertile feedback and inspired further research and development.

Language-centered Environments

2 Language-centered Environments

Language-centered environments are those in which the operating system and tool set are specially built to support a particular language. Examples of language-centered environments are Interlisp for the Lisp language, Cedar for Mesa / Cedar, Smalltalk for Smalltalk, and the Rational Environment for Ada. Initial implementations of Lisp environments emerged in the late 1960s. Researchers at Xerox worked concurrently on the Cedar, Smalltalk, and Interlisp environments in the mid-1970s. The Lisp activities culminated in the early 1980s with the definition of the Common Lisp language. The Rational Environment emerged in the early 1980s.

Lisp environments were the most influential in the development of techniques suited for language-centered environments. They contributed the notion of an exploratory style of programming and demonstrated the benefits of making semantic information available to the user. Subsequent environments for imperative languages extended these notions toward supporting programming-in-the-large to meet large-scale software development requirements.

2.1 Exploratory Style

Environments in the language-centered category encourage an exploratory style of programming to aid rapid production of software. The development environment and the runtime environment are the same. Code can be developed, executed, tested, debugged, and changed quickly; small code changes can be made executable in a matter of seconds. Programs can be built interactively in increments, allowing the user to experiment with software prototypes. Implementation techniques for these environments result in a coupling between the application program and the environment. This coupling makes all the facilities of the environment available to the user building an application.

Language-centered environments use language-specific implementation techniques. To support the rapid production of software for research prototypes, Interlisp uses a large virtual memory space, for example. The tools are engineered as a monolithic system in one address space, and the application program is embedded in the same address space. Thus, the environ-

ment does not need to context-switch between tools and the application program. Execution of the application program can be halted, and change can be made to source that are then dynamically linked into the executable image. Such use of memory created the need for garbage collection of unused, expired objects.

The embedding of the application in environment code allows the application writer to use all the facilities in the environment when constructing an application. Since the environment and the application program share the same language, the application has all the features of the environment available to it as building blocks. For example, InterLisp is a ”residential system” in which the Lisp program resides in the runtime system as a data structure [1]. As a result, the user is able to quickly prototype an application by reusing code available as part of the environment. In the same manner, the user is able to extend the environment with tools to satisfy specific needs. This approach is particularly evident in Smalltalk, in which the user can create objects and operations that default to or inherit the properties of other objects.

Since the close coupling of application and environment results in a lack of clear separation between the host environment and the target runtime environment, in cases such as cross-machine development special efforts have to be made to deliver an application program without the full development environment.

2.2 Semantic Information

Language-centered environments support the exploratory, interactive mode of programming by recording and making available semantic information. Language-centered environments maintain syntactic and semantic information about programs in a particular program representation format. For example, the Rational Environment uses a DIANA format to represent Ada programs. It creates this structure when parsing program text and attaches semantic properties to the structure. Semantic information is typically symboltable information such as information about the definition and use of variables and procedures and information about types. By making this information available through such tools as browsers, the environment helps the user understand the status and structure of code under development. Browsing involves navigating through the set of program objects (functions or modules, for example) and making queries about the objects and their relationships. Interlisp's Masterscope was one of the first browsers to make semantic information available to the user during program development.

Browsers have been accepted as fundamental powerful tools for exploratory program development. They also have the potential of being very effective during program maintenance. Maintainers are usually not the original developers of a program and often can depend only on the source code as the up-to-date documentation. Before making changes to a large, unfamiliar program, maintainers usually spend considerable time understanding the program structure and the interconnection of its components. Browsers help maintainers determine the scope of a change by allowing them to in-

teractively examine the program structure and ask which components may be affected by a change.

2.3 Programming-in-the-large

High-level programming languages do not adequately support the activities involved in constructing large systems. For example, Ada provides packages to support modularization, but it does not permit alternative implementations or successive versions of code to be attached to a package specification. For that reason, language-centered environments have added facilities to support programming-in-the-large.

Techniques for controlling and managing multiple versions of modules among multiple users have also been developed. These techniques inspired more formal definitions of version control and configuration management. For example, Cedar pioneered a paradigm for system modeling. It allows the user to define a blueprint—that is, a system model. The model is a description of the modules that make up a program. Given the model, the environment can maintain a history of the user's selection of various versions in forming a program.The environment can also determine the recompilation needs of a program and can recomile modules to maintain consistency among them.

The Rational Environment supports multiple users. It provides facilities for building and maintaining versions of groups of modules (subsystems). It can enforce a check-in / check-out procedure that prevents programmers from overwriting each other's modifications. It also controls access to program components.

2.4 Observations

Language-centered environments give the user a one-language universe of discourse. These environments are well-suited to the coding phase of the software development cycle. They provide incremental compilation or interpretation techniques to help reduce the impact of small code changes during maintenance. The exploratory style of programming they support helps the user experiment with software prototypes. Tools such as browsers not only are extremely helpful to the user during exploratory program development but can be quite effective for maintenance of large software systems.

Because of the specialized techniques used to implement them, these environments generally do not support multiple languages and, in some cases, do not facilitate the porting of application programs. Also, language-centered environments can become too large for one person to comprehend and extend.

The environments for imperative languages support programming-in-the-large facilities such as version control. But they do not currently support programming-in-the-many tasks such as project management nor do they provide support for development tasks other than coding. It is not clear whether such environments can scale up their facilities to fulfill these requirements, but they will probably form one component of future environ-

ments that will support the entire software life cycle. The specialized, handcrafted nature of these environments makes it difficult to adapt them to phases other than coding.

Developers of commercial software systems are trying to refine their implementation techniques to improve performance. They are building language-centered environments for imperative languages such as C and Modula-2 and are attempting to scale up these environments to support the design phase and to incorporate some project management techniques. The research community is applying language-centered techniques to languages such as Prolog and to specification languages. Commercial application builders so far have used language-centered environments mainly for developing prototype systems.

Structure-oriented Environments

3 Structure-oriented Environments

The initial motivation for structure-oriented environments was to give the user an interactive tool—a syntax-directed editor—for entering programs in terms of language constructs. This capability was extended to provide single-user programming environments that support interactive semantic analysis, program execution, and debugging. The editor is the central component of such environments; it is the interface through which the user interacts and through which all structures are manipulated. Efforts were continued to support programming-in-the-small and programming-in-the-large and to support structures such as history logs and access control lists. Thus the term "syntax-directed" has gradually been replaced by "structure-oriented.

Structure-oriented environments have made several contributions to environment technology—they have provided direct manipulation of program structures, multiple views of programs generated from the same program structure, incremental checking of static semantics and semantic information accessible to the user, and most important, the ability to formally describe the syntax and static semantics of a language from which an instance of a structure editor can be generated.

3.1 Manipulation of Structure and Text

Structure-oriented environments support the concept of direct structure manipulation. The user interacts directly with program constructs and avoids the tedium of remembering the details of the syntax. While program text is displayed on the screen, the user directly modifies the underlying structure. Early environments such as Emily used parse trees as program structures. Most current environments use abstract syntax trees, which were introduced in the Mentor environment.

Structure-oriented environments take several approaches to the manipulation of structures. One involves purely structural editing; it can be viewed

as primarily template-driven editing. An example of this approach is the Aloe editor in the Gandalf project. Aloe provides editing operations only on structural elements and does not permit the user to construct syntactically incorrect programs. To overcome the difficulties in entering and modifying language expressions, some environments such as the Cornell Program Synthesizer represent expressions as text.

Another approach is mixed-mode operation, which is being used in several commercial structure-oriented environments. For example, in the Rational Environment the user can operate on the textual representation and on the structure. The user enters program fragments as text and asks the environment to complete the processing as far as possible. Using incremental parsing techniques [2], the environment converts the text fragment into a program structure. The user can edit the program using commands applied to both the program structure and the program text. The environment keeps the two representations consistently updated.

3.2 Multiple Views of Program Structures.

Structure-oriented environments generate the textual representation of a program from its structure. Thus, different representations can be generated from the same structure. This property allows users of structure-oriented environments to view programs at different levels of abstraction and detail. Browsing of large program structures is provided by showing views of different levels of detail in different display windows. The user can select a program component in one window and have it displayed in more detail in another. This capability can be found in research environments such as the Gandalf prototype and in commercial products such as the Rational Environment. Moreover, research prototypes such as the Pecan environment have demonstrated that it is feasible to produce graphical representations from program structures.

3.3 Semantics and Incremental Processing

After the first syntax-directed editors were built, it was quickly recognized that enforcing correct syntax is only one way of supporting the programmer. Analysis of static semantics was added to editors. The semantic analyzer processes the program structure and decorates it with semantic information. The user can access the semantic information—such as the definition of identifiers and the locations of their use—and type information through the editor. For example, as the user programs a procedure call, the editor can display the specification of the procedure and, as soon as the procedure name has been entered, provide a template for the procedure parameters.

A structure-oriented environment is an interactive tool. Therefore, it should not only give the user immediate feedback on syntax errors but also report static semantic errors before the user moves on to edit other parts of the program. This means that the environment must track the user's changes to the program (that is, know its structure) and reanalyze only those parts that are affected, doing this upon explicit user request or when the user leaves the modified program unit. Proven compiler technology, such as attribute grammars, has been successfully extended to support such incremental

processing. Environments such as LOIPE demonstrate that the notion of incremental processing can also be applied to code generation and linking. There is a trade-off between processing as small a unit as possible and the complexity of the algorithm for doing so. Different types of processing can be done at different levels of granularity. For example, syntactic correctness can be enforced at the language construct level while static semantics is checked at the program unit level (at the level of a procedure, for example) and code is generated at the module level.

3.4 Generation of Structure-oriented Environments

One of the major contributions of structure-oriented environments is the ability to support manipulation of program structures in a language-independent manner. The environment developer achieves this by encapsulating the syntactic and semantic properties of a language in a grammar. Given this declarative description of the language, generation tools can automatically produce instances of structure-oriented environments. This is more efficient than building an environment from scratch. Proven compiler technology such as parser generation has paved the way for progress in this area. The Aloe editor introduced the capability of describing the syntax of a language. The Cornell Synthesizer Generator is a well-known example of supporting the description of semantic properties in terms of attribute grammars, and of providing incremental analysis algorithms as part of the generated structure-oriented environment instance.

The language-specific information can be kept in a form that can be loaded into the language-independent environment kernel at runtime, permitting one instance of the structure-oriented environment to understand several program structures simultaneously. The environment can be adapted and tailored through changes in the declarative description.

3.5 Observations

Structure-oriented environments support direct manipulation and multiple textual views of program structures. Research has shown how static semantic information can be attached to program structures and made available to the user. Algorithms that can incrementally analyze this information have been developed. Instances of structure-oriented environments can be generated automatically from descriptions of the language to be supported. These descriptions specify both syntactic and static semantic information in a declarative manner.

Structure-oriented environments have become mature enough that they are becoming available in commercial products. Environment builders can generate instances of structure editors for different languages with little effort. In many cases, the capabilities of these generated editors are restricted to syntactic checking. Semantic analyzers are often constructed manually.

Structure-oriented environments have been accepted primarily as teaching aids. Universities have been using them to teach introductory programming courses. Despite their availability, they have found little acceptance in industry. Structure editors are being used to support only the coding phase and are being viewed as tools for programming-in-the-small. So far, little

empirical data has been collected to indicate whether structure-oriented environments actually increase productivity.

Initial attempts to scale up structure-oriented environments to support programming-in-the-large and programming-in-the-many have encountered difficulties. Techniques currently used in many structure-oriented environments have shortcomings in terms of providing efficient, persistent storage for large structures and in coordinating concurrent access to the structures for multiple users or tools. Furthermore, for different tools to be integrated in a structure-oriented environment, they must either be adapted to understand a common structural representation or there must be mechanisms for consistent updating of structures through multiple views.

Tookit Environments

4 Toolkit environments

Toolkit environments consist of a collection of small tools and are intended primarily to support the coding phase of the software development cycle. They provide little environment-defined control or management over the ways in which the tools are applied. The toolkit approach starts with the operating system and adds coding tools such as a compiler, editor, assembler, linker, and debugger, as well as tools to support large-scale software development tasks such as version control and configuration management. The toolkit approach was motivated by the need to be language-independent while supporting programming-in-the-large facilities. The toolkit approach uses simple data modeling to aid the extensibility and portability of tools. Examples of commercial toolkit systems are the Unix Programmer's Workbench (Unix/PWB), the DEC VMS VAX-set, and the Apollo Domain Software Engineering Environment (DSEE). Examples of prototype toolkit environments are the Portable Common Tool Environment (PCTE) and the Common APSE Interface Set (CAIS).

4.1 Extensibility and Portability

Unix is an operating system that has encouraged extensions. It has a very simple data model for tool interaction and persistent data storage: an ASCII byte stream. This uniform model enables the user to tailor the Unix environment by adding new tools or modifying existing ones. Tools interact via ASCII files or communication channels called pipes, which encourages reuse of existing tools and tool fragments. Each tool or tool fragment must parse the text stream to extract a structured representation of the data; that is, no structure or semantic information is recorded with the data. This results in tools with few incremental processing capabilities. The Unix/PWB places few, if any, restrictions on when and how tools can be used. Such a model gives the user considerable latitude but provides little support in terms of consistently and automatically managing user activities. The simplicity of the tools and their interactions makes them portable across similar environments.

The simple ASCII model lacks support for typing stored data and for uniformly describing and processing the structure of data. It provides very limited typing of objects other than the use of file name extensions. Developments are in progress to extend file systems to include some of this support. Apollo's Extensible Stream package provides a mechanism that supports typed files and permits the introduction of user-defined file types. DSEE has facilities embedded in its file system to provide a version control mechanism that is transparent to the environment user and the tools. Environments such as PCTE and CAIS support persistent, typed objects with an extensible set of object attributes, while the Arcadia research project addresses tool integration and extensibility as well as object management.

4.2 Operating System Extensions

To provide more environment-defined control facilities while retaining tailorability features, commercial environment builders such as Atherton Technology are placing higher-level interfaces on top of the normal operating system user command interface. Such environments allow the user to work within the context of the high-level functions of the environment, such as those for project management, but also to jump into the native operating system command level, such as that for Unix or VMS, when needed. These higher-level "shells" try to place more controls on tool usage in toolkit environments. They can also provide a uniform interface independent of the underlying operating system. Not only can the user be shielded from the operating system itself but he or she can have transparent access to distributed computing facilities. Similarly, the user can add new tools to the toolkit environment without needing to have a knowledge of the underlying hardware and operating system.

4.3 Programming-in-the-large

Toolkit environments provide programming-in-the-large tools that are independent of a particular programming language. These tools try to ease the programmer's managing of code by providing mechanisms for recording version numbers for source code. Some file systems append a version number to a file name and increment it each time the file is rewritten. The Unix / PWB and VMS VAXset provide explicit tools-the SCCS (Source Code Control System) and the CMS (Code Management System), respectively-to perform version control. Such tools simply record versions and coordinate access to them; the user must decide how to use the version information. For configuration management, systems such as OSEE build a consistent software system as described and requested by the user.

4.4 Observations

Toolkit environments use operating system facilities to "glue" tools into a collection. The intent is to provide a language-independent environment that supports multiple languages with appropriate tools. Such environments allow a high degree of tailoring but provide few environment-defined management or control techniques for using the collection of tools. The user must establish management policies to ensure that tools will be used correctly. Although very popular because of the tailorability and port-

ability of their tools, toolkit environments do not greatly assist the maintenance of large software systems.

The current generation of toolkit environments uses a fairly mature technology. Research on extensions to operating systems is continuing. Extensions include a better data model to support persistent storage and distributed data access, and uniform operating-system-independent user interfaces.

Method-based Environments

5 Method-based Environments

Method-based environments each support a particular method for developing software. The methods fall into two broad classes: development methods for particular phases in the software development cycle, and methods for managing the development process. Development methods are those used by individual developers in phases such as requirements analysis, system specification, and design. Methods for managing the process are those that support orderly development of a software system via product management procedures for consistent evolution of the product by a number of developers, and via models for organizing and managing people and activities.

5.1 Support for Development Methods

Development methods address various steps in the software development cycle. They include but are not limited to methods for specification, design, validation and verification, and reuse. Different methods exhibit various degrees of formality: a method may be informal, as in written text; semiformal, as in textual and graphical descriptions with limited checking facilities; or formal, with an underlying theoretical model against which a description can be verified. Examples of semiformal methods for specification and design are SREM, IORL, CORE, SADT, SDL, PSL / PSA, variations of data flow diagrams and control flow diagrams, and entity-relationship (ER) diagrams. Examples of more formal methods for specification are Petri nets, state machines, and specification languages such as GiST, Refine, VDM, and Anna. While several of the semiformal methods have been practiced to some degree since the mid- 1970s, formal methods are less used. In many cases, they were initially practiced with paper and pencil.

Originally, tools for development methods were provided on mainframes and textual notations or special graphics terminals were used. Examples of such systems are TAGS and DCDS. The availability of computers to a larger number of developers and the advent of affordable graphical facilities on personal computers and workstations have encouraged the development of a large number of commercial tools— especially for semiformal methods of schematic design—often called computer-aided software engineering, or CASE, tools. Examples of such products are Index Technology's Excelerator, Nastec's CASE 2000, Cadre's Teamwork, and Interactive Development Environments' Software Through Pictures. Such tools support individual us-

ers in drawing and updating graphical designs interactively, help organize the design into a hierarchy of abstraction levels, and perform certain consistency checks. Users can invoke analyzers such as level checkers that check for consistent use of names and for connections between levels of the design. Cross-reference information is derived by the analyzer (in most cases not incrementally) and is stored in data dictionaries that can be queried interactively.

Many CASE vendors have realized that it is desirable for their tool to support more than one design method and for customers to adapt the tool to their own methods. Recent releases of commercial tools allow users to define their own graphical symbols for the graphical editor and to write and interface their own analyzers. In some cases, instances of a tool can be generated for different methods. Integration of different methods is limited to instances of graphical editors and analyzers that can be invoked from the same system menu. For methods that differ only in the shape of symbols, such as Merise diagrams and ER-Chen diagrams, a tool can display the design with either set of symbols. All instances of the tool store the cross-reference information in the same data dictionary.

5.2 Support for Managing the Development Process

Managing software development consists of managing the product under development as well as managing the process for developing and maintaining the product. Support of product management includes facilities for version, configuration, and release management along with procedures and standards for performing these tasks consistently. Support for managing the development process includes facilities for project management (project planning and control), task management (helping developers organize and track their tasks), communication management (knowing and controlling the communication patterns in the project organization), and process modeling.

Initially, tools to support particular management functions such as scheduling, cost estimation, or change request control were built in isolation. Recently, research has tried to take a more global approach to understanding and formalizing the software development process and its management. This is evidenced in the literature [3-5]. Research prototypes of environments such as IPSE 2.5, Genesis, and PMA investigate the feasibility of supporting various aspects of encoding and supporting a process model. ISTAR, a commercial development, centers the environment around a particular model—the contract model—rather than around a particular tool. The environment supports planning and management of tasks and management of products as they are being built by teams of developers. It also provides facilities to integrate specification, design, coding, testing, and documentation tools. One of the challenges for such environments is to support process models and management styles without disturbing those in place at a particular organization. The practicality of such environments has yet to be shown.

5.3 Observations

Method-based environments support particular development methods and the management of the development process. A number of tools for single users that support semiformal graphical development methods have become available. Instead of encoding particular methods in these tools, their developers have engineered them to be more general purpose. Instances of design tools can be created through a tailoring and generation process. Generally, isolated tools are available to support version and configuration management and project management.

The distance between existing tools and those of an ideal software development environment is great. Progress must occur in several areas and these are the target of long-term research activities. Research efforts such as MCC's Leonardo project are investigating the feasibility of bringing the exploratory style of language-centered environments to specification and design environments. This requires a better understanding of the specification and design process and the development of formal methods that appropriately capture information and decisions. A supportive environment must capture and reason about the semantics embedded in the method, and must process information incrementally to assist the designer in exploring design alternatives. A better formal understanding of the derivation of efficient implementations from a specification permits more automation of this process. Research in formal and executable specifications, program transformations, and program synthesis is making progress, but solutions can be expected initially only in particular application domains. Examples of such research efforts are the Prospectra project sponsored by Esprit and projects at USC / ISI [6] and Kestrel Institute [7]. Finally, an environment that helps manage the development process requires research into formal models for capturing the process, reasoning about it, and supporting the dynamics of its execution.

User Interfaces

6 User Interfaces

Major developments in hardware technology such as bitmapped displays and mouse devices have made novel user interfaces possible. It seems quite natural to use a pointing device as the user interface for object-oriented languages such as Smalltalk. The Smalltalk language-centered environment replaces the textual user interface—command lines—with pop-up menus, a window manager, and a pointing device, thereby providing a very "friendly" interface. On-line tutoring is provided by means of tools such as "explain" and "example" facilities and through on-line documentation.

Cedar is also recognized for its user interface, which uses techniques similar to those of Smalltalk and incorporates icon management. Cedar provides a level of abstraction by defining an imaging model to handle complex graphics on different hardware. This model defines graphic objects by their semantics rather than by their representation. The implementation then maps these graphic objects onto the hardware constraints. Cedar dem-

onstrates how a nontextual interface can be abstracted from the operating system functions.

6.1 Structure-oriented Environments

In structure-oriented environments, the user interface is ruled by the semantics of the language being edited. The Aloe editor of the Gandalf project is a prime example of a pure structure-oriented editor. For example, when the user gives the command if, the editor constructs the if node in the structure, displays the template for the if construct in the program, and places the cursor at the condition of the construct. The syntax is enforced because the user can only apply the if command when the placement of the construct is syntactically correct. The user moves the cursor according to the structure. The cursor is shown on the screen as a highlight of the textual area that represents the substructure to which the cursor refers. For example, moving the cursor up from the condition of an if construct highlights the text of the whole if construct. To modify existing programs, editing operations such as cut and paste and structural transformation operations are provided. This allows the user to change, delete, or move syntactic program fragments rather than single characters or lines of text. Transformation operations allow the user to convert, for example an *if* statement into a *while* loop or to nest a sequence of statements into a for loop. Despite these editing operations, using a purely structural editor to modify expressions is awkward. Therefore, some structure editors (the Cornell Program Synthesizer, for example) treat expressions as text rather than as structure—that is, they allow the expressions to be entered and edited as text and then they parse them.

6.2 The Rational Environment

The Rational ADA Environment works in a mixed mode that allows the user to edit a program both structurally and textually. After the user enters a program fragment in text form the environment constructs a structure representing the text entered and places the cursor at the first place to be completed. For example, asking the editor to complete the "if" results in the construction of the if statement presented. The user can move the cursor and edit according to the structure (for example, move to the next statement or delete a statement) and according to the text (for example, move down a line or delete the rest of the line).

6.3 Elision

A technique known as elision displays a marker such as "..." (three dots) or a label Instead of the actual program. Elision markers are used to compact source code below a particular nesting level. For example, when the user places the cursor at the first statement of a procedure, only the specification and not the body of the procedure is seen. Constructs such as for while, or if may be similarly compacted. However, when the user moves the cursor to the vicinity of an elision marker, the text is automatically expanded.

6.4 Browsing

Browsing is provided in a similar manner. In one window, the user sees a high-level view of a program, for example the a list of all Ada package names in a program library. The details of the packages are hidden. The user can select a package and open it for viewing. This results in a textual representation of the selected component in more detail in a separate window. The user may see, for example, a list of procedure specifications contained in the package. As before, the user can select one of the structural elements and examine it in more detail or modify it.

6.5 Method-based environments

Method-based environments consisting of CASE tools have powerful graphics editing facilities. Interfaces appear similar to the menu-based ones for language-centered environments. The difference lies in the manipulation of graphical symbols based on the syntactical constraints of particular programming technique. Symbols and links can be created, deleted, and changed. This is done by positioning the cursor with a pointing device and selecting a particular item from the available menu.

Conclusions

7 Conclusions

The four types of environment in our taxonomy represent the major technical directions that software development environments have taken. We have discussed the work in each area primarily from the user environments are interactive, incremental environments suited to an exploratory style of programming. Structure-oriented environments generalize and formalize these techniques and provide generators. Both types of environments show the advantages of making semantic information available. Toolkit environments stress user tailorability and the reuse of a fairly generic tool set, while method-based environments concentrate on providing support for development activities such as requirements analysis and design, and management activities such as guiding a team of programmers during the development process. While it would seem straightforward to merge these capabilities to achieve a highly interactive, tailorable, multiple-user, full-life-cycle environment, the implementation strategies used for each type of environment cannot be easily combined to produce such a result. We conclude by examining what we consider to be the primary issues when we combine or extend the advantages offered by each category.

7.1 Data Management

Language-centered and structure-oriented environments can incrementally maintain executable objects and static semantic information about program units. This functionality is very dependent on the underlying programming-language foundations. There are formal means to define the language units, the constraints on those objects, and the relationships between them. Such information is used to maintain a consistent state after a modification

(such as the state after a modification changing of a variable name or the deleting of a statement).

Method-based environments do not yet support such incremental analysis and do not seem to have theoretical foundations as mature as those of language-centered and structure-oriented environments. The development methods they support often take an operational or control-flow approach to the development process. Most process models are more oriented to managing user activity rather than to managing objects. The object management architecture used by structure-oriented environments is not directly applicable to method-based environments. It is a research question as to whether it should or can be applied.

The kind of tool integration and incremental processing found in structure-oriented environments and in many language-centered environments depends on using a shared data store. The underlying representations for those environments, and for the tools found in the method-based environments, are very specialized. A generalization of a shared data store is a database management system. Commercial databases have been used to support configuration and project management information, but typically have not been used for tool-related data management. Generalization of the features that users like in individual tools in the language-centered and method-based environments requires a solution to the shared data storage problem. The file system extensions discussed earlier may offer a partial solution to this problem.

7.2 Programming-in-the-large/programming-in-the-many

As tools for supporting development methods become more readily available, they are being applied to larger projects. This requires scaling up a method—that is, incorporating into its notation and supporting tools techniques for managing large descriptions. Techniques for scaling up are well known in the programming language field and appear in both the toolkit and language-centered environments. They include partitioning mechanisms (such as modularization, data encapsulation, and interface checking) and management mechanisms (such as version control). These concepts are not used as extensively by the method-based environments, which concentrate on the operational aspects of the development process. The subtasks of the programmer are described primarily in terms of actions in which information hiding and data dependencies play no significant role. The issue here may be a simple one of maturity, since there is considerable commerical activity in this area

7.3 Tailorability

There are a variety of ways to achieve tailorability. Toolkit environments take advantage of generic tools defined on relatively unstructured data. In the other categories, the information maintained is much more complex. Thus, tailorability is more likely to be achieved by tool generators. For method-based environments, we need to consider the tailorability of individual tools (such as editors) for a specific design method, and the tailorability of project or configuration management policies. Structure-oriented environments provide generators for tools such as semantically

based editors. Generators are also appearing in method-based environments, where they have been used to build graphical editors for a variety of design methods. Tailoring the environment to a specific management policy or process development model is more difficult than tailoring a specific tool and will depend on better formalisms for describing the software process.

There has been progress in software development environments—their support of coding is particularly well understood. Environments are slowly improving, but combining technologies to better address the entire software life cycle remains an area for active research.

8 A Sampler of Software Development Environments

Listed here are the languages, methods, and environments discussed in the text. They are grouped into the four categories of our taxonomy; a citation to the literature appears for each.

A Sampler of Environments

8.1 Language oriented environments

Ada

Ada Programming Language, American National Standards Institute, New York,1983.

Cedar

D.C. Swinehart, P.T. Zellweger, and R.B. Hagmann, "The Structure of Cedar," *SIGPLAN Notices* (Proc. ACM SIGPlan Symp. Language Issues in Programming Environments), July 1985, pp. 230-244.

Common Lisp

G.L. Steele Jr., *Common Lisp—The Language*, Digital Press, Burlington, Mass.,1984.

DIANA

G. Goos et al., eds., *DIANA—An Intermediate Language for Ada* (*Lecture Notes in Computer Science*, Vol.161), Springer Verlag, Berlin,1983.

Interlisp

W. Teitelman and L. Masinter, "The Interlisp Programming Environment," in *Tutorial: Software Development Environments*, Computer Society Press, Los Alamitos, Calif.,1981, pp. 73-81.

ObjectMath

P. Fritzson, *ObjectMath—Object Oriented mathematical Modeling in Scientific Computing, Applied to Machine Elements Analysis*, Report LiTH-IDA-R-91-06, March 1991.
P. Fritzson, D. Fritzson, L. Viklund, J. Herber: "Transformation of Equation-

Based Real-World Models to Efficient Code, Applied to Machine Elements Geometry", In *Proc. of the 1:st National Swedish Symposium on Real-Time Systems*, Uppsala, August 19-20, 1991

The Rational Environment

J.E. Archer, Jr., and MT Devlin, "Rational's Experience Using Ada for Very Large Systems," *Proc. First Int'l Conl. Ada Programming Language Applications for the NASA Space Station*, NASA, June 1986, pp. B2.5.1-B2.5.12.

Smalltalk

A. Goldberg, "The Influence of an Object-oriented Language on the Programming Environment," in *Interective Programming Environments*, D.R. Barstow, H.E. Shrobe, and E. Sandewall, eds., McGraw-Hill, New York,1984, pp. 141-174.

8.2 Structure-oriented environments

Aloe

PH. Feiler and R. Medine-Mora, "An Incremental Programming Environment," *IEEE Trans. Software EngIneering*, Sept. 1981, pp. 472-482.

Synthesizer Generator

T. Reps and T. Teitelbaum, "The Synthesizer Generator," *SIGPLAN Notices* (Proc. ACM SIGSoft/SIGPLAN Software Engineering Symp. on Practical Software Development Environments), May 1984, pp. 42-48.
Also by the same authors:
The Synthesizer Generator— A System for Constructing Language-Based Editors, Springer Verlag, 1989.
The Synthesizer Generator Reference Manual, Springer Verlag, Third edition 1989.

DICE

P. Fritzson: *"Symbolic Debugging through Incremental Compilation in an Integrated Environment", J. Systems and Software* Vol. 3, 1983, pp. 285-294.

Emily

W.J. Hansen, "User Engineering Principles for Interactive Environments", in *Interactive Programming Environments*, D.R. Barstow, H.E. Shrobe, and E. Sandewall, eds., McGrawHIII, New York, 1984, pp. 217-231.

Gandalf

A.N. Habermann and D. Notkin, "Gandalf: Software Development Environments," *IEEE Trans. Soltware Engineering*, Dec.1988, pp. 1117-1127.

LOIPE

D. Notkin, "The Gandalf Project," *J. Systems and Software*, May 1985, pp. 81-105.

Mentor

V Donzeque-gouge et al., "Programming Environments Based on Structured Editors: The Mentor Experience," *In Interactive Programming Environ-*

ments, D.R. Barstow, H.E. Shrobe, and E. Sandewall, eds., McGraw-Hill, New York, 1984, pp. 12&140.

Mjølner/Orm
G. Hedin and B. Magnusson: "The Mjølner Environment - Direct Interaction with Abstractions", *Proc. of ECOOP'88*, Oslo, Norway, August 1988, (LNCS 322, Springer-Verlag).

Pecan
S.F Reiss, "Graphical Program Development with PECAN Program Development Systems," *SIGPLAN Notices* (Proc. ACM SIGSoft / SIGPlan Software Engineering Symp. on Practical Software Development Environments), May 1984, pp. 31-41.

8.3 Toolkit enviroments:

Apollo DSEE
D.B. Leblang and R.P Chase, Jr., "Computer-aided Software Engineering In a Distributed Workstation Environment," *SIGPLAN Notices* (Proc. ACM SIGSoft / SIGPlan Software Engineering Symp. on Practical Software Development Environments), May 1984, pp. 104-112.

Arcadia
R.N. Taylor et al., *Arcadia: A Software Development Environment Research Project*, tech. report, Univ. of California, Irvine, Mar. 1986.

CAIS
Military Standard Common APSE Interface Set, Proposed MIL-STD-CAIS, Ada Joint Program Office, Washington, D.C., Jan. 1985.

PCTE
F. Gallo, R. Minot, and I. Thomas, "The Object Management System of PCTE as a Software Engineering Database Management System," *SIGPLAN Notices* (Proc. Second ACM SIGSOFT / SIGPLAN Software Engineering Symp. on Practical Software Development Environments), Jan. 1987, pp. 12-15.

Unix/PWB
TA. Dolotta, R.C. Haight, and J.R. Mashey, "Unix Time-sharing System: The Programmer's Workbench," in *Interactive Programming Environments*, D.R. Barstow, H.E. Shrobe, and E. Sandewall, eds., McGraw-Hill, New York,194, pp. 353-369.

VMS VAXset CMS
User's Introduction to VAX DEC/CMS, Digital Equipment Corp., Maynard, Mass., 1984.

References

9 References

9.1 Main reference

Susan Dart, Robert Ellison, Peter Feiler, and Nico Habermann: "Software Development Environments", *Computer*, Nov 1987.

9.2 Other references

[1] E. Sandewall, "Programming in an Interactive Environment: The Lisp Experience," in *Interactive Programming Environments*, D.R. Barstow H.E. Shrobe, and E. Sandewall, eds., McGraw-Hill, New York, 1984, pp. 3l-80.

[2] C. Ghezzi and D. Mandrioli, "lncremental Parsing," *ACM Trans. Programming Languages and systems*, July 1979, pp. 58-70.

[3] M.M. Lehman and L.A. Belady, *Program Evolution—Processes of Software Change*, Academic Press, Orlando, Fla., 1985,

[4] M. Dawson, "Iteration in the Software Process: Review of the 3rd Int'l Software Process Workshop," in *Proc. 9:th Int'l Conf.on Software Engineering*, Computer Society Press, Los Alamitos, Calif., l987, pp. 36-4l.

[5] *Proc.9th Int'l Conf. on Software Engineering*, Computer Society Press, Los Alamitos, Calif.,1987.

[6] R. Balzer, "A 15 Year Perspective on AUtomatic Programming", *IEEE Trans. Software Engineering*, Nov. 1985, pp. 1257-1268.

[7] D.R. Smith, G.B. Kotik, and S.J. Westfold, "Research on Knowledge-based Software Environments at Kestrel Institute", *IEEE Trans. Software Engineering*, Nov. 1985, pp. 1278-1295.