Programming Environments for Reusability

A. N. Habermann
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Programming Environments are designed to support the development of programs and systems. Some environments emphasize the production process, others focus on the product. Traditional commercial environments rely on topdown design and stepwise refinement. More recently, environments have reached the market that offer integrated sets of tools in support of system development. Tool integration and automation have been the main issues of research for some time. Many interesting ideas that make use of program generators and Artificial Intelligence techniques have been demonstrated in prototype environments. It is interesting to speculate which type of environment will be best suited to support emerging techniques such as reusability.

1. Introduction

The main objectives of Software Engineering are: improving the quality of software systems and increasing the efficacy of the software production process. This dual goal is pursued by developing a variety of facilities consisting of methods, tools and techniques that make programmers more productive and that make systems more reliable, faster and friendlier.

Large software producers are accustomed to measuring programmers' productivity in terms of the number of lines of code a programmer writes in a month or in a year. Practice shows that this number is on the order of 2,000 lines of debugged code per year [2]. Since there is little hope that productivity can be improved simply by having programmers write faster, the dominant approach so far has been to focus on correctness. This approach forces programmers to spend most of their time on testing, debugging and modifying programs. Therefore, a noticeable increase in productivity may result if the time spent on these activities can be substantially reduced.

There are two popular methods of reducing the debugging and program modification effort, both based on the observation that it is better to prevent errors than try to correct them afterwards. The first and most widely practiced method is to institute tight control over the production process. This method amounts to enforcing rules that standardize the way programmers specify, test and document their programs. This method has as a side effect that additional positions are created for system analysts, who oversee the software production process. The second method is called the "clean room" approach [19], which emphasizes the importance of verification as part of program and system design [4].

An alternative approach, very different from controlling the production process or introducing a programming methodology, is to reduce the programming effort simply by writing less code. There is a general belief that many programs for standard computations and input/output processing are unnecessarily rewritten from scratch each time a new project is started. This may be avoided if we can develop standard techniques for practicing "reusability". The purpose of writing reusable programs is to avoid this writing from scratch and enable programmers to generate programs by instantiating reusable components.

When reusability becomes a technique, it will change our attitude to programming. It will shift the emphasis from "do it yourself" to "see what is available". For most programmers, programming will primarily be the task of composing systems out of existing program components. Programming environments will play an important role in the process. They will help programmers with constructing concrete programs from reusable components. It is thus interesting to see how well existing programming environments are able to support reusability.

This work was supported in part by the United States Army, Software Technology Development Division of CECOM COMM/ADP, Fort Monmouth, NJ, and in part by ZTI-SOF of Siemens Corporation, Munich, Germany.

The purpose of this paper is to examine which type of programming environment is suitable for supporting reusability. In the next section we discuss a variety of techniques that lead to reusable programs. A review of the various types of programming environments that support software development follows in Section 3. In the last section we then briefly discuss which of these environments is most likely to be suitable for supporting reusability.

2. Reusability

There are some techniques in use that encourage the reuse of programs. However, for a variety of reasons, most components of software systems are not reusable. After reviewing the standard techniques and the obstacles to reuse, we will discuss some emerging techniques that may make a larger fraction of system software reusable.

2.1. Standard Techniques

The most common forms of reusability practiced today are *code sharing* and *program libraries*. For instance, substantial pieces of code are shared among users through the use of a common operating system. Very few programmers feel a need to write their own file I/O routines. Most users are happy to use procedures, both in their programs and at the system command level, that are provided by the operating system interface. The code for these procedures is shared in a direct sense because no user gets a private copy of the code.

Another form of code sharing takes place in the design of a system family that builds on a common kernel and a collection of parameterized facilities that can be adapted to the particular requirements of an individual family member. The family concept is practical for systems that have many elementary functions in common. The family approach is very natural to system designs that make use of generic tools. Several projects that generate structure-oriented programming environments have taken this approach [12, 13, 23].

The use of program libraries is another widely-practiced form of reusability. Well-known are the libraries of mathematical routines that have been perfected over the last twenty years. Note that these programs do not exist in a single version, but allow the user to select from a variety of implementations. This choice makes it possible for a programmer to find the version that suits the particular characteristic of his input. A matrix inversion program, for instance, can be chosen according to the special characteristics of the input matrix such as triangular shape or sparseness.

Yet another form of reuse that has great potential is the use of generic units that define the operation on a collection of objects independent of the types of these objects. This idea is supported in Ada[1] by generic packages that accept type parameters, and in functional languages by the concept of polymorphic types. It is peculiar that the idea of a variety of different implementations for a single set of specifications (as provided for math routines) has apparently not occurred to the designers of these languages.

2.2. Obstacles

There are several reasons why it is so hard to reuse programs. Three of the main reasons are:

- programs are hard to read
- programs are strongly context dependent
- information about programs is hard to find.

Although language designers often adopt readability as one of their design goals, the fact is that for most languages it is extremely difficult to derive the meaning of a program from its text. There may be some hope for Prolog programs, because here the code is practically the same as the specification. The situation is worse for functional languages and practically hopeless for imperative languages such as FORTRAN, Pascal, C or Ada. The main problem with these languages is, of course, that the program text describes *how* the computation is to take place, but does not describe *why*. Programs become hard to modify without proper specification and documentation. This characteristic makes it hard to reuse programs.

It is surprisingly difficult to port programs from one environment to another. Standardizing on a particular operating system such as UNIX[2] is generally believed to alleviate the problems of context dependencies of programs. Although it may be true that it is extremely hard to port code from UNIX to an IBM operating system for instance, even porting code from one UNIX to another is by no means trivial. Most UNIX systems suffer from local peculiarities, enhancements and restrictions. As a case in point, it took more than half a year at CMU to convert all software from Berkeley-UNIX 4.2 to the upwards compatible version 4.3.

Many systems have grown so large that it becomes very hard for most users to find out what is available. Few systems provide effective browsing facilities and hierarchical help or plain straightforward explanations. Discouraged by the steep learning curve, users are often inclined to make do with a bare minimum of essential

[1]Ada is a registered trademark of the US Government, AJPO.

[2]UNIX is a registered trademark of AT&T's Bell Laboratories, NJ.

facilities. Increasing the user's skills is more likely to occur by word of mouth than by reading the documentation. This is altogether a serious obstacle to reusing programs.

2.3. Reusability through Abstraction

The problem of understanding programs can be solved by representing reusable programs at a higher level of abstraction than is possible in an existing programming language. Such an abstract representation is routinely used in textbooks on algorithms. Using a higher level of abstraction has the advantage that the logic of an algorithm is conveyed without binding control flow and data representations too early in the design process. However, if the abstract representation is far removed from a standard program representation, translation into such a standard representation becomes a substantial effort.

A programming language is not the suitable vehicle for representing the logic of an algorithm. Consider, for example, the doubly linked list structure presented in [15]. The Pascal or Ada programmer has no trouble following the explanation in Knuth's book, because these languages lead the programmer immediately down the path of pointers. Even the qualification "doubly linked" strongly suggests an implementation where each element has a left and a right pointer such that

$$\textbf{if } \mathrm{right}(x) = y \textbf{ then } \mathrm{left}(y) = x.$$

Suppose one wants to implement this structure in a functional language such as Miranda [29]. Functional languages do not provide pointers. One must therefore first determine the essential property of doubly linked lists before one can find a suitable implementation in such a language. The essential property is, of course, that one can walk a doubly linked list in both directions. One can move from each node either to its left neighbor or to its right neighbor.

Once this property is understood, the representation in a functional language becomes even simpler than in pointer languages. Standard operations allow one to access the first or last element of a list. Removing the first or the last element leaves respectively the tail or the head. Another standard operation is "insert" which prepends a new element to a list. If the current element is the first element of the list, a left move and a right move are respectively accomplished by

insert(last,head) and insert(first,tail).

It is clear that programming languages are far too suggestive as to what solution one should adopt. For the given example, imperative languages promote the pointer implementation, while functional languages provide a list implementation that suits the problem well. Although both are able to implement the bidirectional structure, neither one is able to express its essential property explicitly. It is interesting to observe that the attempt of describing an implementation in a different programming language naturally leads first to describing the essential property of a problem independent of language and then to deriving a solution as an implementation of those properties. This shows that it is natural to look for a more abstract description, at a higher level than a programming language, in which the logic of algorithms and procedures can be described.

Assuming that we will succeed in describing reusable components by abstract schemata, we are facing two issues: how programmers will find out about the existence of a reusable component and how programmers can transform such a component into a concrete program that can be compiled or interpreted by existing language systems. Good browsing capabilities and hierarchical help facilities are needed for programmers to find out what is available. For each reusable component a clear explanation is needed to describe its purpose and describe all the available options and design decisions a user must make. Such choices are to be expressed by attaching parameters and attributes to reusable components.

The description of a reusable component and all its options may be fairly elaborate. The bidirectional structure, for instance, might allow three different implementations that respectively use lists, arrays or heap variables. For each of these the memory size may be set in advance or not. The description would mention that each element has a predecessor and a successor except for the first and last elements. The description may provide the option of defining the successor of the last element to be that element itself, to be the first element or to be undefined. The reusable component may also leave open the choice of whether structure elements must be of a uniform type or can be of heterogeneous types.

A general design of the way reusable components can be described is beyond the scope of this paper. However, the example shows that such a description is likely to be fairly elaborate and that it may require quite a bit of work to construct a concrete program out of a reusable component. This type of work can be facilitated greatly by a programming environment that provides the tools for program transformations and for browsing through the descriptions and options of reusable components.

3. Programming Environments

Programming Environment research and development have gone in various directions. Environments can be partitioned into five categories depending on their premises and goals. The five categories are:

- Structure-oriented environments
- Language environments
- Toolkit environments
- Software development methodologies
- System development assistants.

The main characteristics of each of these environments are briefly discussed below.

3.1. Structure-Oriented Environments

Examples of structure-oriented environments are Mentor [6], Gandalf [12], the Cornell Program Synthesizer [27], Pecan [22], PSG [13] and many others. A major objective of these environments is to eliminate the textual interface between user and system. The important idea is to have the user operate directly on data structures and structured data objects in the programming environment. This approach has the significant advantage that user-system dialogue takes place in terms of typed objects with well-defined operations. Having this knowledge, the system is able to check and assure that the user's actions make sense.

Another characteristic of structure-oriented environments is the use of a concurrent model for environments. The traditional model, which is still most common today, is sequential. Information is not shared by a collection of tools, but explicitly passed from one tool to the other. UNIX pipes are a typical example of this model. In the concurrent model, on the other hand, tools operate on a common database which enables them to share information.

The sequential model has the advantage that new tools can easily be added, because tools communicate exclusively through input/output connections. However, the model has two major drawbacks: lack of uniformity of tool design and usage, and potential inefficiency caused by parsing and transforming input for each tool. The concurrent model is exactly the opposite of the sequential model. A tool designer using the concurrent model has the integration of tools foremost in mind and designs the interaction of tools with users and with other tools as a facility of the programming environment. In the concurrent model, tools need not parse and transform input, because information is shared in the common database. On the other hand, the concurrent model has the drawback that extending an environment with new tools may be hard because the existing data formats are fixed and cannot easily be modified to include the information needed by the new tool. Recent work on *views* [10] seems to offer a promising approach to solving this problem. The essence of the solution is to define for each tool the particular view it has on the data and to construct the data formats by merging these views.

Another common characteristic of this category of environments is the generic approach. If the environment consists of typed objects instead of text files, one wants to express environment-specific information in these objects. This object orientation leads to a desire to generate families of environments, each designed to provide support for a specific task. As a result, the question arises how this multitude of specialized environments can be generated within a reasonable period of time. Since the size of each environment is on the order of a language compiler, one cannot expect to be successful with this approach if each environment must be hand-coded. That would mean a production effort of two or more man-years per environment.

To solve the generation problem, program generators have been written that translate declarative environment descriptions produced by the environment designer into programs and tables. These declarative descriptions are much shorter than programs written in a programming language such as C or Ada. Moreover, special purpose environments have been generated that alleviate the designer's task of producing these descriptions. The generic approach has not only served the purpose of generating a variety of environments, but has also proven to be extremely valuable in making it easy to modify and enhance existing environments.

To recapitulate, the three major characteristics of the environments in this category are:

- direct interaction with structured objects, eliminating the traditional textual interface
- the concurrent environment model which facilitates sharing of information among tools
- the generic approach which facilitates the creation and incremental modification of families of environments.

3.2. Language Environments

Language environments exist for almost all important programming languages. Examples are Interlisp for Lisp [28], Smalltalk [11], Cedar for Mesa [26], Lilith for Modula 2 [32], Toolpack for FORTRAN [20], Rational-

Ada [1] and MacGnome for Pascal [3]. They have in common that the environment is centered around a particular programming language and that all programming is done in that language.

The motivation for designing language environments is that programming languages provide facilities for writing individual programs (known as programming-in-the-small), but fail to address system building issues (known as programming-in-the-large). Even Ada, with its constructs for describing intermodular dependencies, went only halfway and limits its facilities to static descriptions of systems. For instance, Ada does not provide constructs for the typical programming-in-the-large facilities that enable programmers to describe alternative implementations or successive versions of the code. Designers of language environments recognized the lack of support for programming-in-the-large in their environments which initially provided support only for language translation, debugging and execution. Extension with tools for programming-in-the-large enables a programmer to stay within the language of choice and build modular systems in its environment. Most of the language environments make use of an existing database or file system for implementing the system building support tools.

The Rational-Ada environment goes the furthest in providing support not only for system building, but also for project management (known as programming-in-the-many). The additional facilities needed for project management address the team aspect of software projects. These facilities support the propagation of information about the project status and enforce some elementary rules of behavior among programmers of a team. For example, the environment can enforce a check-in, check-out policy that prevents programmers from overwriting each others modifications. Most language environments, however, provide support for system building, but not for project management. Interlisp and Smalltalk have this characteristic and are therefore basically single-user environments.

In contrast to structure-oriented environments, all language environments are monolithic and handcrafted. The designers of language environments have not pursued the idea of system families. The handcrafted approach has the advantage that support facilities can be made to run very efficiently, but has the drawback that the environment is hard to modify and that tools and policies are difficult to replace. The language environments are built in the tradition of language compilers. The main characteristics of language environments are:

- single language environment with extensions for building systems
- good support for a single user, little support for project management
- environments are handcrafted, provide efficient tools, are hard to modify.

3.3. Toolkit Environments

Some examples of toolkit environments are PWB [5], DSEE [16], PCTE [9] and CAIS [18]. The Programmers Workbench is an extension of UNIX; DSEE is a system for programming-in-the-large on Apollo workstations; PCTE is the modified version of UNIX designed by a number of European companies under the auspices of the Esprit program; CAIS is the DoD-STARS design for an Ada environment. In contrast to the language environments, the design of toolkit environments starts with programming-in-the-large. These environments provide support for system building with an emphasis on system configuration management and version control. The tools provided by toolkit environments are language independent. The tools are primarily extensions of the operating system, its file system and its user interface.

Toolkit environments extend the operating system interface with the concept of typed objects and with tools. The concept of typed objects is added by all toolkit environments, except PWB, and replaces the notion of text file. However, the range of typed objects is usually fixed and very limited. Typical types supported in the environment are "document", "source", "object code", etc. Users and environment designers cannot add their own types. The tools added to the environment make use of the object types and hide the untyped character of the underlying file system.

The approach taken by the designers of these environments is that the programmer's task can be greatly facilitated by providing tools that help a programmer with the most frequently occurring subtasks and with those aspects of system development control that are tedious for programmers to do, but easy for machines. An example is DSEE which provides elaborate facilities for system version control and for keeping track of the modification process. Neither CAIS nor PCTE provides much support for system building beyond the introduction of typed objects. But all environments in this category are built with the intent that the user is completely in charge and decides when to apply one of the available tools.

In contrast to the language environments, toolkit environments are in principle multi-language environments. Adding a new programming language to the environment requires only little change. However, similar to language environments, toolkit environments are monolithic and handcrafted. The idea of system families plays no role in the design and tools are made to perform well by fine tuning the environment by hand. Also, little support is provided for project management. PWB, CAIS and PCTE provide virtually none, while DSEE provides good support for propagation of information, and some for enforcing project development rules.

Summarizing, we find that the main characteristics of toolkit environments are:

- the environment extends the operating system interface primarily with tools for programming-in-the-large
- text files are replaced by a fixed set of object types
- environments are handcrafted, provide efficient tools, and are hard to modify.

3.4. Software Development Methodologies

Examples of well-known software development methodologies are Jackson's JSD [14], Softech's SADT [24], Yourdon's SA/SD [33] and many similar systems. The emphasis of these methodologies is not on tools or languages, but on support of stepwise system development. The common idea of these methodologies is that software systems can be developed in an orderly fashion if programmers follow a fixed sequence of steps and obey certain rules.

Software design methodologies are based on specific models of the software development process. SADT, for instance, uses the notions of actigram and datagram to describe system components, mainly in terms of actions that involve input, tools, database information and output. The result of adopting a specific model for the development process is that tools in the environment become much more cohesive than in the toolkit approach. Tools are designed as a close-knit collection of functions, each function contributing in a specific way to supporting the methodology.

The dominant concepts in software development methodologies are top-down design and stepwise refinement. Although these concepts can be supported by other environments, they do not play as strong a role as in software development methodologies. However, a much more significant distinction between software development methodologies and the other environments is the fundamental concepts ignored by the methodologies and emphasized by the others. The major concepts that play no role in software development methodologies are abstract datatypes, and in particular data encapsulation, subsystems and version control. The reason that the methodologies ignore these concepts may be in their operational approach to the development process. The subtasks of programmers are primarily described in terms of actions in which information hiding and data dependencies play no significant role.

Software design methodologies were first designed without support of a programming environment. More recently, the large software producers have adopted these methodologies and have built the necessary environments to support them. In contrast to the other types of environments, the software development methodologies provide a complete guide for the development process, while the other environments leave the use of the tools entirely in the hands of the users. A toolkit environment, for instance, does not tell its user when and in which order tools must be used.

The major characteristics of software methodologies are:

- complete guidance through the top-down software development process
- a coherent set of tools that support stepwise development
- no usage of data encapsulation, subsystems and version control.

3.5. System Development Assistants

Examples of system development assistants are ISTAR by Imperial Software [7], ERGO [25] and SEAR [17] at CMU, and various AI systems among which the Programmers Apprentice at MIT [30] and REFINE by Kestrel [31] are developed further than most. The common theme of these environments is to provide user assistance and have the environment complement the user's work.

The development assistant environments resemble the structure-oriented environments and the methodologies in providing a coherent set of tools that assists the user in performing a specific task. However, the system development philosophy promoted by these environments varies widely and does not particularly emphasize top-down design or stepwise refinement. Another major difference with the methodologies is that the development assistants make extensive use of data structures and data dependencies. A difference with the toolkit environments is that the environment performs some subtasks automatically, particularly clerical tasks which are not performed well by the user. Development assistants participate more actively in the work and interact with the user in a manner that one might expect of a human assistant.

Development assistants share with structure-oriented environments the objective of providing support for specialized tasks. In contrast to toolkit environments, development assistants typically build their set of tools on top of a system kernel that provides common facilities to tools and monitors the application of tools in the user environment. However, development assistants resemble language environments and tooklit environments in their monolithic character and in their handcrafted implementation. The idea of sharing facilities and code in a system family has not yet been adopted by the designers of development assistants. Also, the handcrafted approach contributes to runtime efficiency, but makes it hard to enhance and modify an environment.

The development assistants vary widely in objective. ISTAR is a software development environment in the traditional sense, but it achieves its goal in a novel way by having its users express their tasks in terms of contracts and subcontracts. The Programmers' Apprentice and REFINE are examples of knowledge-based systems. SEAR is a specialized environment for building expert systems. ERGO is an ambitious environment for the derivation of programs through a formal representation of the relationship between programs. ISTAR and REFINE are available as commercial products, while the other three have been developed in the academic environment.

In summary, the main characteristics of system development assistants are:

- the environment complements the user's work and performs subtasks automatically
- the environment applies domain knowledge
- environments are handcrafted, provide efficient tools, and are hard to modify.

4. Environments for Reuse

In this section we explore first what is needed for further development of the reusability idea. Many of these needs happen to be things that must be supported well by a programming environment. In the second part of this section we come back to the programming environment taxonomy and see which type of environment can best support reusability.

4.1. Development of Reusability

In Section 2 we reviewed the various techniques that can help produce reusable programs. The techniques discussed were:

- code sharing
- program libraries
- generic program modules
- program generators
- program schemata.

The first two are widely practiced today, while the third is gaining in popularity, particularly through the use of the Ada language. Program generators have been used for specific purposes (such as parsers and structure-oriented programming environments) and can be particularly useful in advancing the cause of reusability by supporting the design of system or program families.

We argued that for reusability to become a more useful technique, algorithms must be described at a higher level of abstraction than is currently possible in a programming language. Such an abstract description must be augmented by parameters and attributes that give the user a choice of implementation, resource utilization, object representation, etc. We use, somewhat reluctantly, the term "program schema" for such an abstract description. The use of this term should not be confused with the use of that term of fifteen years ago.

The design of program schemata is undoubtedly the least explored area of reusability. Although the other forms of reuse are still growing (which is encouraging!), research and development of a technique of reuse through program schemata are just starting. It is therefore natural to ask where to start and in which direction to go.

Language design should not be the first step. We should resist the computer scientist's propensity for attacking a problem by designing a programming language. There are two reasons why such an approach is not appropriate. First, designing a language for describing program schemata will focus our attention on form and not on substance. It is premature to make representation the main issue, when we do not have a clear picture of *what* should be described in a program schema. Secondly, a particular notation should not be chosen until various camps have explored the issue and are able to debate the matter of representation based on research results. This is a lesson learned from the Ada environment design. The Ada language is based on extensive research in language design and programming methodology. The APSE environment for Ada, however, was proposed before research and experimentation had taken place. The result has been a chaotic process of extensions, while the commerical market has largely ignored the Ada environment design altogether and designed its own.

The issues that seem most important with respect to program schemata for reuse are:

- description and documentation
- browsing capabilities

- object editing
- transformation tools
- tool control
- persistent data storage.

The first item on the list refers to the content of a program schema describing a reusable component. A schema must describe precisely and concisely what the component does and how it can be used. A useful example for documentation style may be the kind of explanation one finds in textbooks on the design of algorithms. It is likely that one also wants a mechanism that records the usage of a reusable component: where it is used, which options were used, etc.

The importance of a browsing capability cannot be emphasized strongly enough. A collection of reusable components will be practically worthless if the user must read through long lists of acronyms or cross reference lists. The collection must be accessible by topic, by keyword, by name and must also be traversible in alphabetic order or in random order. It should not be necessary for the user to enter names precisely. The environment must be flexible enough to match names that resemble the input. Wildcards should also be used extensively.

The concept of object-oriented programming has shown its usefulness. An important aspect of object-oriented programming is that the environment deals with typed objects. This approach has the great advantage that the environment is able to maintain consistency and check that operations applied to objects maintain consistency. In an environment consisting of typed objects many mistakes can be avoided by having the user edit these objects directly.

The primary task of an environment supporting reusability is to provide tools that help the user transform a reusable component into a finished program. The environment can contribute two other things, namely tool support and storage facilities. The advantage of designing a collection of tools together is that common facilities such as data structure access procedures can be provided by the environment for all tools to use. It is then not necessary for each tool to define and implement its own support. An important part of general tool support is access to storage. When the user is transforming reusable components into finished programs, the intermediate and final results of his work need to be preserved in part for future elaboration. The environment ought to provide the storage medium in which not only the objects themselves, but also their relationships can be stored.

4.2. Suitable Environments

Almost all types of programming environments discussed in Section 3 are suitable for supporting the current practice of reuse through shared code and program libraries. With respect to libraries, there may be some advantage in working with a language environment or a toolkit environment. The toolkit environments may even have a slight advantage over the language environments, because they often support a multi-language environment in which various language systems can make use of each others libraries. The structure-oriented environments and the system development assistants are somewhat less suitable, because their approach usually requires a specific effort to integrate existing code from elsewhere. The methodology environments handle program libraries as well as the toolkit environments because of their file system or database orientation.

The Ada language environment is particularly suitable for reuse through generic components, because these are directly supported in the language. Other language environments could support a similar mechanism if the module concept of their language were extended to include types and functions as parameters. The other environment types, with the exception of the methodologies, could support generic units in a similar manner as a part of their programming-in-the-large facilities. It is difficult to see how methodology environments easily can support generic units since neither data types nor programming-in-the-large are provided.

Program generators are used extensively for creating structure-oriented environments, but their use is not particularly promoted in the created user environments. There seems to be no particular reason to assume that one environment type is more suitable to support program generators than any other. Program generators are application programs that can be placed in any program library or file system.

Greater difference exists with respect to the support of program schemata. Here it seems that the structure-oriented environments and the system development assistants have a distinct advantage over the others. These environments provide the best support for typed objects and for tool control. They are able to assist the user with object editing, can maintain consistency and can also provide the browsing capabilities. The toolkit environments can be used with a little more difficulty, but will require a lot of discipline on the part of the user.

It seems that the language environments and the methodologies are the least suitable for supporting program schemata. In language environments it is not possible to represent program schemata at a sufficiently abstract level. It might be possible to do this if the designer is willing to go outside of the language, which he probably has done already to add system version control to the environment. The methdology environments are the least suitable for supporting program schemata. In contrast to language environments, the description is not the problem, nor is storage, but all other issues are. Methodology environments lack support for typed objects, while things such as browsing, tool sets and tool support are foreign to their methodology approach.

Conclusion

Current practice of program reuse is primarily based on code sharing and program libraries. Generic units and program generators are emerging as additional reusability techniques. Reusability may become significantly more important if we can find ways to express reusable components at a higher level of abstraction than is possible in a programming language. Such a description must not bind the implementation too early and must capture the logic of an algorithm or procedure. This form of reusability needs the support of a programming environment that uses typed objects, and provides transformation tools and extensive browsing capabilities.

While all types of environments discussed in this paper are able to support the reusability techniques currently practiced, the development of program schemata is better supported by the object-orientation of the structure-oriented environments and by the development assistants. Both types are characterized by their support of integrated tool sets that operate on collections of typed objects. The toolkit environments can probably provide a similar functionality, but much will depend on disciplined behavior on the part of the user. The language environments are limited in what they can do to support abstract program schemata because this approach requires going outside of the chosen programming language. The methodology environments are probably the least suitable for supporting further development in reusability because of their lack of support for abstract datatypes and integrated tool sets.

References

[1] Archer, J. E., "The Design of the Rational Environment", in *Springer Verlag's Lecture Notes in Computer Science, Proc. CRAI Intn'l. Conf. on Software Factories and Ada, Capri, Italy,* May 1986.

[2] Boehm, B. W., *"Software Engineering Economics"*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.

[3] Chandok, R. et al., "Structure Editing-Based Programming Environments: The Gnome Approach", in *Proc. Natn'l. Computer Conf. AFIPS,* July 1985.

[4] Dijkstra, E. W., *"A Discipline of Programming"*, Prentice-Hall, Series in Automatic Computation, pp. 16 - 23, Englewood Cliffs, NJ, 1976.

[5] Dolotta, T. A. et al., "UNIX Time-Sharing System: The Programmer's Workbench", in *Interactive Programming Environments,* D. R. Barstow, Ed. MacGraw Hill, NY, 1984.

[6] Donzeau-Gouge, V. et al., "Practical Applications of a Syntax-Directed Program Manipulation Environment", in *Proc. 7th Intn'l. Conf. on Software Engineering,* Orlando, Fla., March 1984.

[7] Dowson, M., "ISTAR: An Integrated Project Support Environment", in *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* Palo Alto, Calif., Dec. 1986.

[8] Floyd, Ch., "Eine Untersuchung von Software-Entwicklungsmethoden", in *Proc. German Chapter of ACM,* **18**, Munchen, published by Teubner Verlag, Stuttgart, April 1984.

[9] Gallo, F. et al., "The Object Management System of PCTE as a Software Engineering Database Management System", in *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* Palo Alto, Calif., Dec. 1986.

[10] Garlan, D., "Views for Tools in Integrated Environments", in *Springer Verlag's Lecture Notes in Computer Science, Proc. Workshop on Advanced Programming Environments, Trondheim,* Aug. 1986.

[11] Goldberg, A., *"Smalltalk-80: The Interactive Programming Environment"*, Addison & Wesley, Reading, Mass., 1984.

[12] Habermann, A. N. and D. E. Notkin, "Gandalf: Software Development Environments", in *IEEE Transactions on Software Engineering,* **12**, *12,* Dec. 1986.

[13] Henhapl, W. et al., "PSG: A Programming System Generator", *German Chapter of the ACM,* **18**, *Munchen,* published by Teubner, Stuttgart, 1984.

[14] Jackson, M., *"Principles of Program Design"*, Academic Press, 1975.

[15] Knuth, D. E., *"The Art of Programming", Vol. 1, p. 278,* Addison-Wesley, Reading, Mass., 1973.

[16] Leblang, D. B. and R. P. Chase, Jr., "Computer-Aided Software Engineering in a Distributed Workstation Environment", in *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* Pittsburgh, Pa., Apr. 1984.

[17] McDermott, J., "Making Expert Systems Explicit", in *Proc. IFIP 10th World Computer Congress, Dublin, Ireland*, published by North Holland/Elsevier, Sept. 1986.

[18] Military Standard Common APSE Interface Set, proposed MIL-STD- CAIS, Ada Joint Program Office, Wash., DC, Jan 1985.

[19] Mills, H. D. and R. C. Linger, "Data Structured Programming: Program Design without Arrays and Pointers", in *IEEE Trans. Software Engineering*, 12, *2*, Feb. 1986.

[20] Osterweil, L. J., "Toolpack - An Experimental Software Development Environment Research Project", in *IEEE Trans. Software Engineering*, 9, *11*, Nov 1983.

[21] "*Readings in Artificial Intelligence and Software Engineering*", edited by C. Rich and R. C. Waters, published by Morgan Kaufman Publishers, Inc., Los Altos, Calif., 1986.

[22] Reiss, S. P., "PECAN: Program Development Systems that Support Multiple Views", in *Proc. 7th Intn'l. Conf. Software Engineering*, Seattle, Wash., Mar. 1985.

[23] Reps, T. and R. Teitelbaum, "The Synthesizer Generator", in *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, Pa., Apr. 1984.

[24] Ross, D. T. and K. E. Schoman, Jr., "Structured Analysis for Requirements Definition", in *IEEE Trans. Software Engineering*, 3, *1*, Jan. 1977.

[25] Scherlis, W. L. and D. Scott, "First Steps Towards Inferential Programming", in *Proc. IFIP Congress 83*, Paris, Sept. 1983.

[26] Sweet, R. E., "The Mesa Programming Environment", in *Proc. ACM/SIGPLAN Symposium on Language Issues in Programming Environments*, Jul. 1985.

[27] Teitelbaum, R. and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", in *Comm. ACM*, 24, *9*, Sept. 1981.

[28] Teitelman, W. and L. Masinter, "The Interlisp Programming Environment", in *Computer*, 14, *pp. 25 - 34*, April 1981.

[29] Turner, D., "An Overview of Miranda", in *SIGPLAN Notices*, 21, *12*, Dec. 1986.

[30] Waters, R. C., "The Programmer's Apprentice: Knowledge-Based Program Editing", in *IEEE Trans. Software Engineering*, 8, *1*, Jan. 1982.

[31] Westfold, S., "Artificial Intelligence in Software Production", in *Springer Verlag's Lecture Notes in Computer Science, Proc. CRAI Intn'l. Conf. on Software Factories and Ada, Capri, Italy*, May 1986.

[32] Wirth, N., "Lilith: A Personal Computer for the Software Engineer", in *Proc. 5th Intn'l. Conf. Software Engineering, San Diego, Calif.*, March 1981.

[33] Yourdon, E. and L. Constantine, "*Structured Design*", Prentice Hall, Englewood Cliffs, Calif., 1979.