Technological Advances in Software Engineering

A. N. Habermann
Computer Science Department and
Software Engineering Institute
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

A major challenge for software engineering today is to improve the software production process. Nowadays, most software systems are handcrafted, while software project management is primarily based on tenuous conventions. Software engineering faces the challenge of replacing the conventional mode of operation by computer-based technology. This theme underlies the Software Engineering Institute that the DoD has established at Carnegie-Mellon University. Among the contributors to software development technology are ideas, such as object-oriented programming, hardware improvements related to personal workstations, and programming environments that provide integrated sets of tools for software development and project management. Facilities and tools are by themselves not sufficient to achieve an order of magnitude improvement in the software production process. Future directions in software engineering must emphasize a constructive approach to the design of reusable software and to automatic generation of programs. We will briefly explore the promising technology that can be used to implement these ideas.

1 Introduction

A major challenge for the software engineering field is to bring about a radical improvement in the software production process which is plagued by low quality and inflexibility of its products and serious overruns in terms of both cost and time. A decade ago, when software engineering first emerged as a separate sub-discipline, the initial focus was more on controlling the production process than on achieving a radical improvement by changing the process. The result of this control view has been a number of substantial activities in areas such as measuring system performance and programmer productivity and developing techniques for program testing and symbolic debugging. Today's practice is still largely dominated by this control view and its ensuing analytic approach to improving software production.

Although useful for better understanding of the software production process and suitable for finding gradual improvements, analytic tools of the kind mentioned above are generally not adequate to achieve an order of magnitude improvement in the software production process. A radical improvement requires a constructive approach that changes the process itself instead of improving on existing practices. The purpose of this paper is to review the current events in software engineering that support this constructive approach and to explore future developments that may lead to a substantial improvement of the software production process.

Current events that relate to software production are rooted in the short but dynamic history of software engineering. A brief analysis of the history is followed by a discussion of one of the major events in the area at the present time, which is the mechanization of the discipline into support systems and tools that assist programmers in the application of software engineering techniques. The observation that tools and support systems are by themselves not sufficient to bring about a radical improvement in the software production process leads to an outlook on the future. The major ideas and concepts for achieving the desired radical improvement in software production are reusability and automation. The last part of this paper is dedicated to a discussion of these subjects and shows how they can be applied in practice. It is the author's belief that the software production process can be improved substantially if we can steer the development of software engineering in the direction of reusability and automation.

2 The Nature of Software Engineering

Engineering is the creation of mechanisms or objects that facilitate the achievement of a goal. Civil engineers build bridges for people to get to the other side, electrical engineers build radios for the purpose of broadcasting news and musical entertainment, programmers write database systems for people to store and retrieve information. The Oxford dictionary of the American Language stresses the fact that engineering is the application of scientific knowledge and the control of power to achieve the intended goal. The adjective "scientific" seems unnecessarily restrictive because experience and transfer of know-how are substantial factors in engineering without necessarily being scientific. The Romans, for instance, were able to build excellent bridges without the knowledge of Newton's laws of classical mechanics.

An interesting aspect of engineering is that the goal of an engineering endeavor may be to facilitate achieving some other goal. This is the idea of a tool. The mechanical engineer, for instance, may design a floating crane that is used by the civil engineer to build his bridge. The goal of the mechanical engineer is not a particular bridge, but the process of building bridges. The object he creates is not the beam that spans the river, but the tool that enables the civil engineer to put that beam in place.

The task of the software engineer resembles that of the mechanical engineer in the example above. The subject matter of

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1986 ACM-0-89791-177-6/86/0002/0029 $00.75

software engineering is the means and methods that are applied to the creation of software and not the substance of the software product. A peculiar aspect of software engineering, however, is the fact that these means and methods are largely expressed in terms of software. Translated back to the example above, this state of affairs would correspond to a situation in which the mechanical engineer would choose to provide another bridge, instead of a floating crane, as the tool that would enable the civil engineer to build his bridge. The result is a recursive relationship where the tool to achieve the desired goal (the bridge that replaces the floating crane) is of the same nature as the ultimate goal (the bridge to be built by the civil engineer). An interesting consequence of this recursive relationship between tool and goal is that the means and methods invented for achieving the goal (the creation of software) are also applicable to the tools designed for achieving that goal (the software that supports the development of software).

Engineering skills can often be judged by the quality of the resulting product. This is true for all forms of engineering, be it mechanical, electrical or software engineering. Examples of high and poor quality are known to all of us. What to say, for instance, of a text formatting program that allows the title of a section to go on the last line of a page while the text starts on the next page. Another example is the coin box of my car that is not wide enough for my hand but too deep for my thumb. It is not so easy to move a coin forward with one or two fingers and then catch it with your thumb. A particularly nasty example of poor engineering used to be in the electronic mail system at CMU when it was first introduced. The sender of a message was not notified of a misspelling in the address part until after invoking the send command. Instead of returning the undeliverable message, the mail system discarded the message and forced the sender to retype the entire text. Another example of questionable engineering is the operating system that does not let you get out when you type "logoff", but reacts with an error message that says, "Type logout to log off".

Compilers are also notorious for the poor quality of error analysis and error reporting. Many compilers get confused after the detection of the first error and produce long lists of spurious messages because of failing to distinguish internally between a correct and an erroneous program state. An error detection that makes sense in the correct state may be irrelevant in the erroneous state in which preceding errors were found.

Particularly frustrating are cryptic messages for which no further information is obtainable. When I switched to a new operating system environment, the system told me that I was using an outdated version and that I lacked the benefit of some substantial system improvements. It did not tell me, however, how I could get access to this improved version, and obvious procedures such as editing my login-file did not work. Another frustrating example was my first encounter with an Ada compiler that claimed to provide a friendly user interface. Part of the program I had written was:

```
with text_io; use text_io;
package body HELLO is

    procedure sayhello is
        print ("hello there"); newline;
    end sayhello;

end HELLO;
```

The compiler indicated the print line as erroneous, listed an obscure number of five digits and announced that the result type did not match the type of a library subprogram. It seems rather difficult to derive the actual mistake, which is the spelling of *new_line as newline, from the text of the given error message.*

For software, there are some general criteria that distinguish a good design from a poor one and some specific criteria that depend on the specific nature of the software product. To begin with the latter, a banking system requires absolute accuracy of the numbers it moves around, while an electronic mailsystem can tolerate an occasional misspelling in the text of a message and, if it is smart, even in the address part of a message. Another example is the rigid response time requirement for a realtime system versus the tolerance of modest compile time delays for a timesharing system.

Software engineering is particularly concerned with the general criteria that determine the quality of a design and of the resulting software product. Criteria frequently discussed in the literature are:

Correctness	the correspondence of specification, design and implementation
Reliability	the ability to reproduce a result
Performance	the ability to respond within tolerable time limits without excessive demands on storage capacity
Adaptability	the ability to modify software to take advantage of hardware improvements or to respond to changing application requirements
Extensibility	the ability to extend the functionality of a system
Friendliness	the ability to interact with the user in terms of understandable messages while not requiring irrelevant precision of user input
Reusability	the use of parts of a system in the design and implementation of another system
Fault-Tolerance	the protection of information integrity against hardware or power failure
Robustness	the protection of information integrity against unintentional user mistakes and malicious user acts
Privacy/Security	the protection of information against unauthorized access and against the effects of modification in someone else's data.

The most important principle developed in the fields of software engineering and programming languages is that of modularity based on data encapsulation and data abstraction. The modularity concept allows us to control the interface to objects of a particular type by showing the specifications of basic operations that apply to these objects while hiding the particular implementation. Data encapsulation restricts all access to objects to the basic operations defined in the interface while data abstraction hides the internal object structure in addition to hiding the implementation of these basic operations. This form of modularity localizes the effect of implementation modifications which greatly enhances the quality of the software product along many of the criteria listed above.

The purpose of this section has been to present a global image of software engineering and its major concerns. This presentation sets the tone for a brief characterization of software engineering's history and for a discussion of its current highlights and its future development.

3 The Evolution of Software Engineering

The foundation for software engineering was laid in the sixties with the invention and formulation of basic concepts in programming languages and operating systems. The design of FORTRAN which introduced the concept of procedural abstraction was soon followed by the design of Algol60 which introduced a wealth of new concepts including data types, parameter evaluation

modes, recursive procedures, static and dynamic scopes, dynamic data objects and a formal description of language syntax. Later in the decade, SIMULA67 introduced the concept of object-oriented programming through classes and subclasses, while Algol68 and Pascal introduced user defined data types, reference variables and disjunctive type structures. Much of the engineering during this period was concerned with the optimization of parsing and code generation and with the efficient use of hardware resources in timesharing operating systems.

Around 1970, the focus of attention shifted from basic concepts in languages and systems and their implementation to the construction of systems out of program modules. Programmers became more ambitious and wanted to construct systems that were hard to express in a single program. At this point in time, the need arose for programming-in-the-large which concerns itself with program interface specifications, the modification process of program modules in the context of an evolving system, and the interaction between programmers in the context of a software production project. This development had the effect that software engineering shifted its focus from the construction of individual programs to the process that controls the creation of software systems.

The transition from pure programming-in-the-small to the more ambitious programming-in-the-large is viewed by many as the actual birth of software engineering. The distinction between these two forms of programming was clearly stated for the first time in a seminal paper by DeRemer and Kron [DK76]. Some of the most important initial results of software engineering were the modularity concept and Parnas' hiding principle [Pa72]. Other constructive work in software engineering of that period included the design of system version control and configuration management mechanisms. In addition, a substantial effort was put into measurements of performance and productivity as well as into models for controlling the software life cycle which includes the production process from inception and specification to implementation and successive releases. The waterfall model is the best known among the various models proposed for life cycle management [Le80].

An alternative approach to controlling the complexity of large software systems is taken by the founders of a programming methodology. Their activities give rise to the concept of structured programming [Di76] and to various approaches to program verification. Structured programming is in fact a philosophy based on the limitations of human beings in dealing with the substance of programs. It builds on our strengths (rather than our weaknesses) by promoting the utilization of three of our abilities in dealing with algorithms: enumeration, induction and abstraction. Enumeration allows us to distinguish between an overseeable number of cases; induction allows us to make use of iteration and recursion; abstraction allows us to ignore details at proper moments and to reduce complexity by viewing collections of objects as atomic units.

Program verification has been put on a solid basis in the last decade. The axiomatic approach is particularly suitable for proving the correctness of programs based on their control structure. Algebraic verification is particularly well suited for demonstrating the completeness and consistency of a collection of operations defined for an encapsulated data structure. The method of a denotational description of the semantics is particularly suitable for showing the consistency of a language design and for expressing the meaning and interpretation of language constructs.

Although program verification is well understood, a major drawback of the state of the art is our inability to apply the various methods to large systems. The attempts in that direction have resulted in some interesting interactive verification systems [Go75, Lu79] that can handle small to medium size programs but not large systems consisting of many components that are not always collectively available.

The state of the art in program verification at the end of the last decade was one of the causes for another change in the direction of software engineering leading to the exploration of software development tools and environments. Two other causes were the analytic approach to improving the software production process and the labor-intensive implementation of life cycle support. The analytic approach blocked further progress because of the tacit assumption that the software production process was basically well organized and needed only further local optimization. The labor-intensive approach to life cycle support puts system development and project management entirely in the hands of people with little or no support from software technology. In the next section, we discuss the resulting events of the present that are characterized by a mechanization of life cycle support into integrated programming environments.

4 Programming environments

A programming environment is a software system that supports the development and maintenance of software products. The term "programming environment" does not refer so much to the activity of writing programs, but more to the manipulation of programs for the purpose of system generation, configuration and version control, project management and documentation. Although the term "system development environment" is actually more appropriate in this context, we will stay with tradition and stick to the widely used term "programming environment" to denote systems that support the entire spectrum of activities involving software production. The goal is for programming environments to support the entire life cycle and not just the programming fraction of the cycle.

Traditional programming environments lack some properties that seem very desirable in modern programming environments. These properties are tool integration and uniformity of the user interface. Tools are integrated when they possess common knowledge that can be applied in each tool. This common knowledge often takes the form of shared data formats or of information stored in a common database. An example of tool integration is the combination of editor, compiler and debugger that all operate on a common syntax tree. The editor shares syntactic knowledge with the compiler and is able to enforce the syntax rules while a program is being written. The debugger shares program structure knowledge with the complier and is able to translate problems back into source representation through the common database. Tool integration is of great help to create environments that are more specifically task-oriented than the traditional general-purpose environments which are still most common today.

Tool integration is almost totally lacking in the traditional programming environment. Tools such as the text editor, the compiler, the linking-loader and the debugger share at best some knowledge of the underlying file system. No information is shared, however, about data formats or data values and no information is exchanged through a common database. All communication between traditional tools takes place through input/output, while correctness of the representation is entirely in the eye of the beholder (not in the tool!). A text file is a Pascal program, for instance, because the author believes it is one, not because the text editor checked that it really is.

Uniformity of the interface is obtained by using the same command formats and parameter conventions for all tools. In modern programming environments, uniformity of the interface is obtained by default through the general editing environment that controls all interactions between user and programming

environment. This arrangement has the additional advantage that the user may not always have to know which tool is being applied. Traditional environments often do not provide a uniform interface. Users have to remember for each tool a particular command syntax, a parameter convention and the interpretation of various switches. In contrast to the traditional environment, uniformity of the interface matches well with the image of a task-oriented programming environment that provides a collection of cooperating tools designed to assist a user in various complementary subtasks of a project.

Programming environments can be categorized by the basic philosophy underlying their design. We distinguish four categories. A first category consists of the language extension environments. The design of these environments starts out with a particular programming language. Making a programming language the cornerstone of your design leads naturally to environments that emphasize programming-in-the-small, but don't support programming-in-the-large. Examples of language environments that have that characteristic are Interlisp [Te78], Smalltalk [Go83] and Gnome (for Pascal) [Ga84]. Several designers of language extension environments realized soon enough that dealing with modular interfaces, version control, configuration management , etc., is often more intricate than writing programs. Since programming languages provide little or no support for dealing with system-building issues, the natural step is to extend the language environment with a collection of tools for programming-in-the-large. Examples of language environments of this kind are Cedar (for Mesa) [Sw85], Lillith (for Modula2) [Wi81], Toolpack (for FORTRAN) [Os83], APSE (for Ada) [Bu80] and the Gandalf Prototype (for C) [No85].

A second category of programming environments is the group of life cycle support systems. These systems focus primarily on system version control and/or project management. Emphasis of these systems is on documentation of the specifications, of the modifications and of the development history. Features frequently added are automatic recompilation, access control and propagation of changes. Most of the systems in this category are built as an extension of an available file or database system. Some examples of such environments are CADES [Sn80], PWB [Do77] and DSEE [Le84].

A third category of programming environments is the class of task-oriented environments. In this category, emphasis is on the integration of tools to assist the users in performing a specific task. The idea of integration is tightly connected to the useful idea of putting knowledge about the task to be performed in the tools and in the programming environment. The origin of this category of environments is in the concept of syntax-directed editors which later evolved into structure editors. The first system of this kind is the Emily system [Ha71]. The idea of tool integration, starting with editors, was later extended to other tools such as interpreters, debuggers and documentation support. A major breakthrough in this area was the partial automation of generating these environments. This addition is so important because the desire to build task-specific environments creates the need for a large number of slightly different programming environments. The task orientation makes sense only if specific environments can be generated fairly easily. It would not work if every task-oriented environment had to be constructed from scratch and took an amount of time comparable to that of writing a handcrafted compiler. Some well-known environments in this category are the Program Synthesizer [Te81, RT84], the Gandalf System [Ha83, No85], Mentor [Do80, Ka82], POE [Fi84], SYNED [Ga83] and PECAN [Re84].

A fourth category is formed by environments that support a particular system design methodology. These environments provide support tools for designing software according to certain rules that are based on a software development philosophy. Popular methodologies primarily used in industry are those by Jackson [Ja75] and Yourdon [Yo75]. An environment based on a specification methodology is HOS [HZ83].

The first category is distinct from the other three in that each member of that category provides a single language environment and supports only that specific language. Another typical characteristic of this category is that its members are by and large single user environments in contrast to the environments in the other categories that are more oriented towards team work. A common characteristic of the first, second and fourth categories is that their products are all handcrafted and are therefore fairly hard to modify. It is relatively difficult to adapt these environments to the specific wishes of their users. Task-oriented environments, which form the third category, escape this limitation by the generic approach that early on has been recognized by their designers as being crucial to satisfying the need for constructing many variations of a task-oriented environment. An additional advantage of the generic approach is that a particular software design philosophy does not have to be hardwired into the environment as it has to in the environments of all three other categories. Further discussion of the generic approach follows when the topics of reusability and automation are addressed.

Programming environments for software development serve three main purposes:

- to support the programming task
- to control the system construction task
- to assist in project management

Environments of the programming language extension category initially provide most of their support for the programming task which corresponds to programming-in-the-small. In terms of programmers and program modules, one might characterize the programming task as one-to-one: the individual programmer is working on a single program at a time.

Environments of the life cycle category tend to stress the system construction task, leaving the support of the programming task to the traditional tools such as text editor, compiler, debugger and file system. The system construction task is commonly viewed as programming-in-the-large. This is the one-to-many situation where an individual programmer assembles a version of his program module, with modules written by other programmers, into a system version.

Project management adds a new dimension to the software development process, often labeled "programming-in-the-many". The purpose of project management is to control the interaction of programmers and their rights to access and modify programs. Project management facilities enforce design and development rules, but also provide information on current status, development history and future goals. Project management is poorly supported by most programming environments. The environments of the software development and methodology categories may support project status information but are usually not designed to enforce a set of coherent project management rules. This form of control is generally left to costly human labor. The category of task-oriented environment is the most promising with respect to supporting project management because of its generic approach which allows the implementation of a variety of policies that can serve specific needs. The task of project management puts us in the many-to-many situation where the main issue is to control the actions of

programmers who work together on program modules in various states of completion.

The work on programming environments is a constructive response to the challenge of solving the software production problem. The development started out with environments that provided useful tools to support various programming tasks. The current state of the art of generating programming environments is able to handle the syntactic issues of structure, formatting and representation, and also the semantic issues of static consistency, runtime support and dynamic modifications [Re82, Ka85].

We are at a stage where environments can be built that have more the character of an assistant than of a tool box. These environments provide an integrated set of task-oriented tools that complement the work of the people involved in software production. Another useful task these assistant-type environments can perform is to enforce desirable project management rules and maintain a development history. The future is in environments that show intelligent behavior. In addition to maintaining correctness of a software product during all stages of its development, such environments are designed to make value judgements about the quality of the product and its components. These future environments will display knowledge about the software production process similar to what has already been accomplished today by designers of special purpose expert systems.

5 The Software Engineering Institute

Reports published in the literature and presented at conferences show that there are serious problems in the software production process which have a negative effect on the resulting software product. Part of the problem is caused by the difficulty of changing the process to make use of modern technology. Although new equipment and new tools exist, it is often hard to change established methods and hard to integrate new methods into an existing organization.

Rethinking the way large software systems are produced and maintained has been of great concern to the Office of the Under Secretary of Defense for Research and Engineering (OUSDRE). This office has undertaken a major effort to alleviate software-related problems in the military directly, by standardization of proven tools and techniques, and indirectly, by stimulating research and development of software engineering techniques. OUSDRE has established a special program for this purpose, known as the "Software Initiative". This program has three major components: the Ada Joint Program Office (AJPO), the STARS program (Software Technology for Adaptable, Reliable Systems, not to be confused with the Strategic Defense Initiative, commonly known as Starwars) and the Software Engineering Institute. The task of the AJPO is to promote the use of the new Ada[TM1] language and encourage the development of Ada related tools and standards. The STARS program consists of six task forces that prepare requests for proposals in a variety of areas such as software engineering environments, development methodologies, business applications, software metrics, etc.

The Software Engineering Institute (SEI) was established at Carnegie-Mellon University in December, 1984. The Institute is one of four institutes in the university structure and has the status of a college. It is planned to grow to 250 technical people over a period of five years. As of November, 1985, the Institute has just over seventy employees.

The purpose of the SEI is to accelerate technology transition in order to improve the software production process and its resulting products [Ba85]. The plans call for close interaction with the DoD and its software suppliers on the one hand and with the research laboratories in industry and academia on the other hand. The main task of the SEI is to make existing advanced technology ready for transition into the user environment. The SEI plans to do this through building prototype and demonstration systems and by providing training services. An industrial affiliates program has been established for the purpose of sharing the SEI's expertise with DoD contractors and organizing joint projects. It is the intention that industry will turn the SEI's prototypes into useful, marketable products.

The SEI has produced a one and five year plan that describes the spectrum of activities in the software engineering field that is of immediate relevance to its mission. The plans call for six areas of interest addressing both the software production process and the resulting product. The six areas are:

Technology Identification and Assessment
: find promising advanced technology that is ready for transition

The Nature of the Transition Process
: find ways to make new technology attractive and viable for the production environment

Education
: design, with universities and industry, software engineering course material at the master degree level

Reusability and Automation
: build software as a variation of existing software rather than from scratch

System Construction and Evolution
: integrated environments that support the software development and maintenance process

Reasoning about Software
: quality and performance control of the software product.

Work in these areas is organized in the form of projects that concentrate on topics such as the evaluation of existing Ada environments, the legal issues of software licensing and general software development tools. Intermediate results of the projects are presented and demonstrated regularly at open house meetings to which representatives of DoD contractors, government agencies and universities are invited. One of the projects takes the form of a series of workshops in which a gradually increasing number of attendees discuss the issues of software production. A summary of these discussions will be available as an SEI tech report in the summer of 1986.

6 Reusability and Automation

Programming environments, software metrics and software engineering methodologies are helpful, but not enough to bring about an order of magnitude improvement in the quality of our software products and in the predictability of the production cost and effort. Constructive and analytic techniques are both valuable and should be further pursued to facilitate the software production process. However, one must expect no more than a gradual improvement from applying these techniques, because none of them necessarily changes the software production process itself.

It seems that a major problem in software production is the fact that most software is written from scratch. The reasons why this is common practice are threefold.

- programs are hard to read

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

- programs are strongly tied to their context
- information on existence of programs is often hard to get.

The fact that the meaning of a program is hard to derive from the source code gives programmers the feeling that one might as well write the program from scratch instead of trying to understand the designer's reasoning behind an existing program text. Although descriptive documentation is of some help, its two major drawbacks are its separation from the source text and its lack of rules that guarantee uniformity and completeness. Formal specifications are in fact far more helpful for an accurate description of what a program does, but are generally even harder to understand than the source text. A possible solution to this problem is to agree on a functional description of programs that does not describe in detail what a program does, but describes the data structures it uses, the input values it accepts and the output values it produces [He78].

After going through the effort of understanding someone else's program, good intentions are often rewarded with disappointment because of the program's dependency on the runtime environment. Even if a program is designed to run on a popular operating system such as UNIX[TM2], the programmer who wants to make use of it in his environment will discover that the program does not run because of incompatible peripheral equipment or local operating system extensions. Context dependencies are often hard to detect because documentation on these matters is rarely provided. A programming language such as Ada may alleviate this problem because of its precise description of package dependencies. Explicit description of a program's dependency on other programs is strongly recommended over implicit dependencies that are generated by deep nesting of scopes and by an excessive use of global objects.

It is often very hard to find out which programs written by other people can be used again. Many programs are designed as system modules and are buried deep down in a system description. Names of program modules often make little sense outside of the system context, while the purpose of a program usually is described in relation to the modules it interacts with instead of in terms of its own independent functionality. The result is that a design always seems unnecessarily complicated to an outsider. This phenomenon causes the outsider to think that he could have done a better job than the designer of the existing program. This lack of confidence in your fellow programmer is a major cause of unnecessary duplication of effort.

The problem of acquiring information about the existence of programs is solved in part by encouraging the potential user to try harder to find out what is available and how it is used. However, this is not a reasonable proposition without counting on substantial help from the original designer. The only way that one can realistically hope that people will try to reuse software is to demand that designers of original programs take reusability into account from the start. If reusability is adopted as an original design objective, one may expect a program documentation style to emerge that clearly explains a program's independent functionality, its intended use and its dependency on its context.

A programming environment can play an important role in making software reusable. It can provide facilities that allow users to browse through libraries that describe existing programs and their usage. The better programming environment will provide, in addition, an engineering environment that is used for transforming an existing program into one needed for a specific application, or for deriving a specific program from a general description.

Reuse of software is at this point in time our best hope for improving the software production process and its resulting product. It has the potential of reducing the cost and effort of the process and it has a good chance of increasing reliability through incremental modifications of programs of proven quality. However, the preceding discussion shows that the term reusability must not be given the narrow interpretation of reusing existing programs without change. In fact, reusability spans a spectrum of applications that each makes sense in a particular context.

Two direct applications of reusability are the use of program libraries and of shared code. The best example of reused program libraries is that of mathematical subroutines. The IEEE Society has done us a good service by standardizing a set of mathematical routines, including specifications of input/output precision. It would be extremely helpful if similar standard packages were designed and maintained for string processing, window management and namespace management. The Ada language made a useful contribution by including in the language a standard package for file handling and for text I/O.

Users of timesharing operating systems are very familiar with the idea of sharing code. Their programs routinely use common operating system facilities for file handling, input/output and memory management. It is in this context immaterial whether or not executable versions of code are shared. Even if programs each use their own copy of a common program, the fact that counts is that the utility program was not written by the user, but taken, as is, from an available pool.

Practice has shown that reusability through code sharing is greatly facilitated by eliminating the context dependency factor. This can be done in one of two ways: either by writing a program that is independent of its context (this is basically the Ada Language approach), or by having the various users work in the same context so that context dependencies are irrelevant. Although the latter seems to be a cop out, its usefulness has been firmly established by the success of the UNIX operating system. The reason why many companies are interested in standardizing on UNIX, as universities basically have done over the last decade, is to capitalize on the available software that runs on UNIX while avoiding the problems of having to translate and rewrite existing programs to run on different operating systems.

The Ada language provides another form of reusability through type abstraction [Ad83]. Generic packages can be written in Ada that specify the traversal and updating operations on data structures while leaving the element type unspecified. This facility supports the concept of reusability by allowing a programmer to define the details of a data structure once and for all, independent of what type of objects will be stored in that structure. A generic package for queues, for instance, can be instantiated for messages, for jobs, for arrival and departure schedules, etc.

Other practical forms of reusability are through specification and through common design. An example of the former is contained in the large volume of literature on data structures [e.g. Kn73] in which sorting algorithms take much of the limelight. The algorithms dealing with the manipulation of these structures are specified independent of a particular programming language, but in sufficient detail to be implemented in any language. Examples of common design are found in the literature on operating systems [Ha76] and compilers [Ah77]. Memory management and parsing techniques are the typical examples of common design that is applied in many operating systems and compilers. Although this form of reusability has the drawback of requiring implementation, it has the great

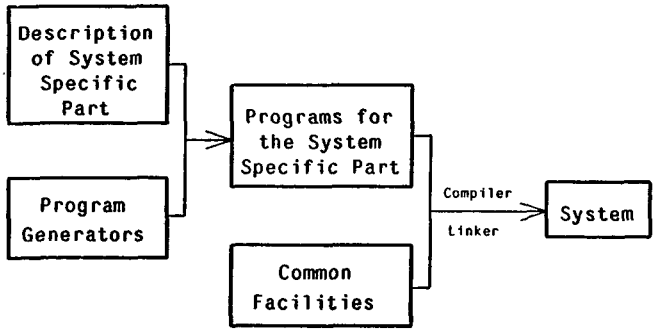
[2]UNIX is a trademark of Bell Laboratories

advantage of building on a conceptual basis that can be taught in the classroom.

Reusability can be greatly enhanced by automation of the program generation process. Automation in this context means using tools that translate a precise program specification into a program form for which a compiler or interpreter exists. The automation tools are commonly known as *generator* programs, or *generators* for short. The input of a generator is a program description and its output is a program in a programming language or in some other form that can be translated into machine code. The target of a generator might for example be intermediate code as generated by the front-end of a compiler. This intermediate code is then translated into machine code by the code generator part of a compiler [Ca80]. The idea of a generator was first proposed for compilers in the form of a compiler-compiler [Br62].

Reusability and automation can be very effective when applied to the design of a family of systems that have a large part in common. Examples of such families are database management systems, compilers and programming environments. The kind of facilities which members of a system family typically have in common are the facilities for database or file management, for input/output and for maintaining the user interface.

The common facilities are basically used unchanged by all members of a system family. Slight differences may be expressed in parameters or system generation switches. The additional system-specific part, however, is what distinguishes one family member from another. Here, automation comes into play: program generators translate a designer's description that is written in a predefined formalism into a collection of compilable (or interpretable) programs and data. A scheme for generating systems is depicted below:



The designer describes the specific behavior of the target system by defining the objects that the user will be able to create, manipulate and delete in that target system. The designer's description consists of three parts:

the abstract syntax
: describing the logical structure of the objects in the target system and how they are composed

the concrete syntax
: describing the representation of objects in the target system in user-readable form

the runtime support
: describing the actions that must be taken at runtime for purposes such as checking semantics, resource allocation or project management.

The use of these descriptions is best illustrated by an example. Suppose the target system is an electronic mail system. Typical objects in a mailsystem are messages, mailboxes and bulletin boards. The common facilities provide general operations for creating, deleting and browsing through objects in the target system. Specific are the structures of messages, mailboxes and bulletin boards. A message, for instance, is composed of more primitive objects such as a date, an address and a text. The abstract syntax describes this logical structure of messages, as it also does for mailboxes and bulletin boards. The concrete syntax describes the output format of these objects in order that the user can read messages and see mailboxes and bulletin boards. The runtime support description defines how the specific objects use the common facilities such as memory or output windows, what updating is needed when objects are created or deleted and what kind of consistency rules or access control applies to the objects in the target system.

Reusability may even apply to the system-specific part. It is not uncommon that system family members provide just slightly different operations on the objects in the target system. This commonality can be captured in a library of specific facilities available to the designer for copying or slight modification. This idea has been successfully applied in the design of many task-oriented programming environments that were mentioned earlier.

Automation of the abstract and concrete syntax is not difficult to achieve since designing formalisms for the description of syntax has been well understood for more than two decades. Automation of the runtime support, however, is far more difficult because most of the runtime support consists of active procedures that play the role of watchdogs for the objects they are attached to. Writing runtime support in a traditional programming language is an acceptable alternative if automation is lacking, because it still requires only a small part of the code to be written from scratch. Of course, if automation of the runtime support is not achievable, the existence of a library of standard runtime support routines becomes all the more important.

7 Conclusion

The main objective of software engineering is to help produce high quality software systems within reasonable bounds of time and cost. The major factors that determine the quality of a software product in addition to its desired functionality are reliability, performance, flexibility and friendliness of the user interface. Software engineering is right now facing the challenge of solving the serious problems encountered in the software production process which lead to cost and time overruns and products that are lacking in many of the quality factors.

Tools most frequently used for improving the software production process are program measurement and software development support tools. Measurement tools are helpful in finding the bottlenecks in the existing software production methodology. Support tools are helpful in alleviating the task of the people involved the production process. Both kinds of tools help to make the production process more effective and more reliable. The original design of isolated support tools is gradually being replaced by integrated programming environments that behave more as intelligent assistants than as toolboxes.

Measurements and support tools are designed to correct flaws in an existing methodology, but do not address the more fundamental question of methodology itself. There is a general feeling that current practices are inadequate (and will become more so in the near future) to satisfy the growing demand for reliable software that is produced on time and within budget. The basic flaws of the current process are its labor-intensive approach to project

management and product development and its propensity for programming from scratch.

It seems that a significant improvement can be achieved if we can produce reusable software and automate the generation of new software. Success in the area of reusability may reduce the production of new software to a fraction of what is commonly written today, while automation has the potential of simplifying the production process with an additional gain in reliability.

Reusability has no chance of being successful unless taken into account as a major design objective from the start. A major obstacle to overcome is the problem of information dissemination. With current software production practices, it is extremely difficult to find out what is available and how things work. Time is ripe for a major effort to define the concept of reusable software precisely and to develop techniques for creating reusable software.

References

[Ad83] *Reference Manual for the Ada Programming Language.* United States Department of Defense, January 1983.

[Ah77] Aho, A. V. and J. Ullman. *Principles of Compiler Design.* Addison Wesley, 1977.

[Ba85] Barbacci, M. R.; A. N. Habermann; M. Shaw. The Software Engineering Institute: Bridging Practice and Potential. *IEEE Software [2], 6.* November 1975.

[Br62] Brooker, R. A. and D. Morris. A General Translation Program for Phrase Structure Languages. *Journal of the ACM 9*, pp. 1-10, 1962.

[Bu80] Buxton, J. N. *Requirements for Ada Programming Support Environments* (Stoneman). US Government, Department of Defense, February 1980.

[Ca80] Cattell, R. G. G. Automatic Derivations of Code Generators from Machine Descriptions. *Transactions on Programming Languages and Systems*, Vol. 2, 2, April 1980.

[Di76] Dijkstra, E. W. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, New York, 1976.

[Dk76] DeRemer, F. and H. Kron. Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Transactions Software Engineering [2], 2.* June 1976.

[Do77] Dolotta, T. A. and R. C. Haight. *PWB/UNIX--(Overview and Synopsis of Facilities)* Technical Report, Bell Laboratories, June 1977.

[Do80] Donzean Gouge, V. et al. *Programming Environments Based on Structure Editors: the Mentor Experience.* INRIA Rapports de Recherche, No. 26, July 1980.

[Fi84] Fischer, C. N. et al. The POE Language-Based Editor Project. *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984.

[Ga83] Gansner, E. R. et al. A Language-Based Editor for an Interactive Programming Environment. *Proceedings, IEEE-Compcon83*, San Francisco, Calif., February 1983.

[Ga84] Garlan, David and P. Miller. GNOME: An Introductory Programming Environment Based on a Family of Structure Editors. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984.

[Go75] Good, D. I.; R. L. London; W. W. Blesdoe. An Interactive Program Verification System. *Sigplan Notices*, June 1975.

[Go83] Goldberg, A. and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison & Wesley, Reading, Mass., 1983.

[Ha71] Hansen, W. J. *Creation of Hierarchic Text with a Computer Display.* Ph.D. Thesis, Stanford Univeristy, June 1971.

[Ha76] Habermann, A. N. *Introduction to Operating System Design.* Science Research Associates, Inc., Chicago, Palo Alto, Toronto, 1976.

[Ha83] Habermann, A. Nico and D. Notkin. The Gandalf Software Development Environment. *Proceedings of the Second International Symposium on Computation and Information*, Monterrey, Mexico, September 1983.

[He78] Heninger, J., D. L. Parnas et al. *Software Requirements for the A-7E Aircraft.* Naval Research Lab., Washington, D.C., Memo Rep. 3876, November 1978.

[HZ83] Hamilton, M. and S. S. Zeldin. The Functional Lifecycle Model and Its Automation: USE.IT. *Journal of Systems and Software*, Vol. 3, No. 1, March 1983.

[Ja75] Jackson, M. *Principles of Program Design.* Academic Press, 1975.

[Ka82] Kahn, G. et al. *Metal: A Formalism to Specify Formalisms.* Technical Report, INRIA (1982).

[Ka85] Kaiser, G. E. *Semantics for Structure Editing Environments* Ph.D. Dissertation, Carnegie-Mellon University, 1985.

[Kn73] Knuth, D.
The Art of Computer Programming, Volume III: Sorting and Searching.
Addison-Wesley, Reading, Mass., 1973.

[Le80] Lehman, M. M.
On Understanding Laws, Evolution and Conversation in the Large-Program Life Cycle.
The Journal of Systems and Software 1, 3 (1980).

[Le84] Leblang, D. B. and R. P. Chase, Jr.
Computer-Aided Software Engineering in a Distributed Workstation Environment.
Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. Pittsburgh, Pa., April 1984.

[Lu79] Luckham, D. C. et al.
The Stanford Pascal Verifier User Manual.
Stanford University, Stanford, Calif., March 1979.

[No85] Notkin, D. S.
The GANDALF Project.
The Journal of Systems and Software, [5], 2. May 1985.

[Os83] Osterweil, L. J.
Toolpack - An Experimental Software Development Environment Research Project.
IEEE Transactions on Software Engineering,
pp. 673-685, November 1983.

[Pa72] Parnas, D. L.
On Criteria to Be Used in Decomposing Systems into Modules.
CACM, December 1972.

[Re82] Reps, T. W.
Generating Language-Base Environments.
Ph.D. Dissertation, Cornell University, 1982.

[Re84] Reiss, S.
Graphical Program Development with PECAN Program Development Systems.
Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, April 1984.

[RT84] Reps, T. W. and R. Teitelbaum.
The Synthesizer Generator.
Proceedings ACM/SIGSOFT Software Engineering Symposium on Software Development Environments, Pittsburgh, Pa., April 1984.

[Sn80] Snowdon, R. A.
An Experienced-Based Assessment of Development Systems.
Software Development Tools, pp. 64-75,
Springer Verlag, Berlin, Heidelberg, New York, 1980.

[Sw85] Sweet, R. E.
The Mesa Programming Environment.
Proceedings, ACM SIGPLAN85 Symposium on Language Issues in Programming Environments, Seattle, Washington, 1985.

[Te78] Teitelman, W. et al.
The Interlisp Reference Manual.
Xerox Palo Alto Research Center, Palo Alto, Calif., 1978.

[Te81] Teitelbaum, T. and T. Reps.
The Cornell Program Synthesiser: A Syntax-Directed Programming Environment.
CACM, September 1981.

[Wi81] Wirth, N. and R. Ohran.
Lilith - A Personal Computer for Software Engineering.
Proceedings, 5th International Conference on Software Engineering, San Diego, Calif., March 1981.

[Yo75] Yourdon, E.
Techniques of Program Structure and Design.
Prentice-Hall, 1975.