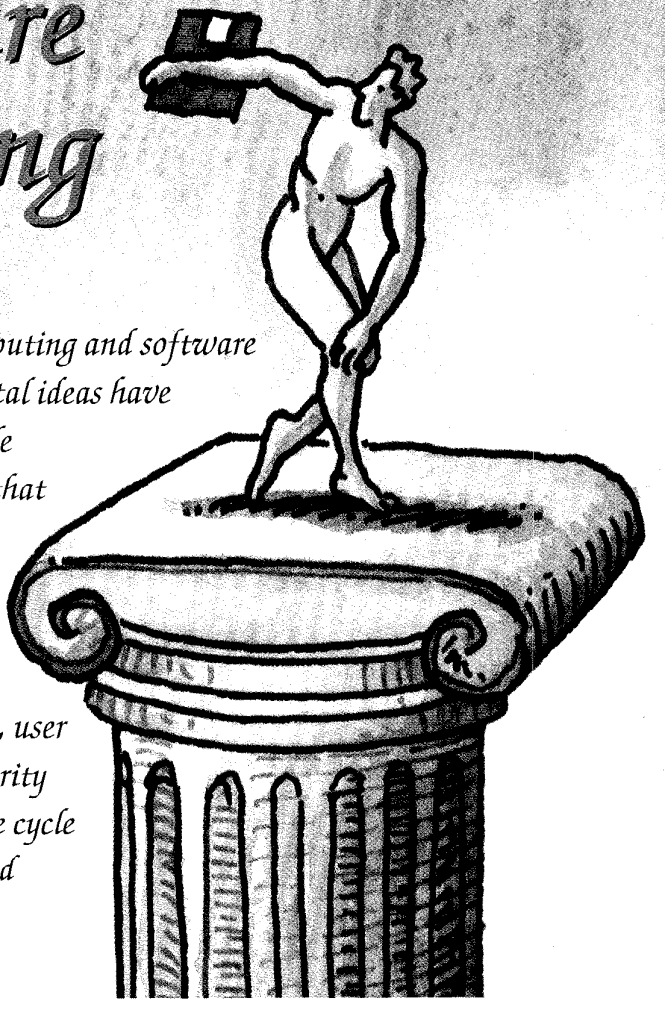
Toward a Discipline of Software Engineering

ANTHONY I. WASSERMAN

Despite rapid changes in computing and software development, some fundamental ideas have remained constant. This article describes eight such concepts that together constitute a viable foundation for a software engineering discipline: abstraction, analysis and design methods and notations, user interface prototyping, modularity and architecture, software life cycle and process, reuse, metrics, and automated support.



The nature and complexity of software have changed significantly in the last 20 years. In the 1970s, applications ran on a single processor, produced alphanumeric output, and received their input from a linear source. For many years, lasting through the 1980s, software development organizations typically followed a waterfall model of software development. Most software architectures were either transformation-based, where input was successively transformed into output, or transaction-based, where input such as a command line determined selection of a particular function.[1] Networking and remote access were rare.

0740-7459/96/$05.00 © 1996 IEEE

Today's applications are far more complex, typically having a graphical user interface and a client-server architecture. They frequently run on two or more processors, under different operating systems, and on geographically distributed machines. Such applications rely heavily on layers of systems software for windowing, network connectivity, messaging, security, and database or object management. Traditional architectures and development approaches no longer work well for many of today's applications.

It is impractical to do analysis and design independently, particularly when following a spiral model of development.

Seven key factors are driving the fundamental changes we are seeing:

- the criticality of time-to-market for commercial products;
- shifts in the economics of computing, including lower hardware costs and relatively higher development and maintenance costs;
- the availability of powerful desktop computing;
- extensive local and wide-area networking, particularly the emergence of the World Wide Web as a publishing and distribution medium;
- growing availability and acceptance of object technology;
- the WIMP (windows, icons, menus, pointers) style of graphical user interfaces; and
- the continuing unpredictability of the waterfall model of software development.

Any of these factors alone would have a significant effect; taken together, they have completely transformed the software development process in most organizations.

Despite the rapid changes in computing technology and software development, some fundamental concepts of software engineering have remained constant. This article delineates eight of these key ideas, treating each one individually while recognizing close interrelationships among them. None of these ideas is revolutionary or particularly exceptional. Taken together, they sum up many of the advances in software engineering and thereby serve as the heart of a set of best practices that software development organizations can use.

ABSTRACTION

Abstraction is a fundamental technique for understanding and solving problems.

Abstraction is a common intellectual technique for managing the understanding of complex items. It allows us to concentrate on a problem at some generalized level without regard to irrelevant low-level details. It also allows us to work with concepts and terms that are familiar in the problem environment without having to transform them to an unfamiliar structure.

Abstraction hierarchies are natural and very common. The term "house" represents a category, or *class*; it hides considerable detail. When such details interest us, we treat them as *attributes*: dimensions, number of bathrooms, type of frame, and so on. For example, at a high level of abstraction, a house is a type of building; it shares some properties (building dimensions, for example) with other types of buildings, such as a supermarket. Other attributes are unique to houses. A house is also a type of residence, sharing properties with other types of residences (apartment, for instance). Depending on the problem being addressed, the abstract class "house" may inherit properties from either the class "residence" or the class "building," or both. Even with values for these attributes, abstraction hides still lower levels of implementation detail, such as plumbing and electricity.

Notions of abstraction are pervasive in software engineering and other aspects of computing. In object-oriented development, for example, abstraction provides a clean definition of the operations (methods) for a class, as well as specifying the parameters needed to instantiate a class from a generic class. Abstraction is also the central concept of information hiding,[2] which lets software developers focus on the appropriate level of detail concerning a software component. Application programming interfaces (APIs)—as found in network services, foundation classes, and component libraries—are excellent examples of information hiding.

ANALYSIS AND DESIGN METHODS AND NOTATIONS

Analysis and design methods and notations are basic tools for communication in an engineering discipline.

The cognitive leap from understanding a problem to implementing a system is too great to proceed without including analysis and design models as key deliverables in the development process. Many analysis and design methods and notations have been created to serve as a basis for problem or system understanding.

Analysis considers the problem space, addressing the structure of the problem, including the logical structure

of objects in the real world. The goal of requirements analysis is to provide an unambiguous and comprehensible specification from which a system can be designed and built. Design, by contrast, deals with the structure of the system that implements a solution to the problem. For example, design must deal with system constraints such as interfaces to existing systems and components, as well as overall system architecture.

There is also an important distinction between the notation (representation) and the method used. The notation, whether textual, graphical, or mixed, is used for communicating analysis and design information to others, and often has a well-defined semantics. The method defines a set of steps to be followed in deriving the analysis or design model expressed in this notation. To the user—a programmer or quality assurance engineer, for example—all that is relevant is the representation itself, since the process of creating that representation is hidden.

Universality. An important aspect of established engineering disciplines is a universally used set of unambiguous representations for modeling artifacts. Electrical engineers use block diagrams and schematic diagrams; several other engineering fields depend on blueprints. All practicing engineers in those fields, and other professionals who work with them, use and understand these notations.

Software engineering has no equivalent. Instead, there are hundreds of different notations, many with numerous variants. Of these, only a few—state transition diagrams, entity-relationship models, dataflow diagrams, and structure charts—are *widely* known and used. The various approaches to analysis and design disagree about which modeling primitives and notations are appropriate for representing analysis and design information.

In practice, software developers sometimes implement systems without producing or validating any analysis and design model at all. Even more often, they change an implementation without properly updating the existing analysis or design model to maintain consistency. Four decades of painful experience have shown that developers cannot easily go back and forth between the problem domain and the implementation domain unless the problem at hand is relatively small.

Omitting these analysis and design activities or doing an incomplete job on them is widely recognized as a principal cause of development errors, high maintenance costs, and failure to build systems that meet user requirements. Without usable and precise requirements specifications and design models, acceptance testing and quality assurance become nearly impossible, and documentation will likely be inaccurate.

Interdependence of analysis and design. Although analysis and design activities have different goals and address different requirements, it is impractical to perform these tasks independently, particularly when following a spiral model of development. Analysis methods frequently yield a problem structuring that can serve as the basis for a system architecture. Conversely, design issues may bring new requirements to light, often involving handling of exceptional conditions, causing bidirectional feedback between analysis and design.

Along the same lines, the entire development process can be more efficient if the analysis step takes advantage of knowledge about existing software assets. Rather than starting with a "blank slate," as assumed in most texts on analysis methods, analysts often work within the context of an application development framework, an existing enterprise data model, an extensive set of existing analysis models, and/or an existing system that is being modernized.

In theory, it should be possible to develop a method-independent software design representation that provides the appropriate syntax and semantics for describing the design. Various software design methods could then be used to create designs represented with this notation. If software engineering is to mature as a discipline, standard specification and design notations will have to be developed, along with methods for problem decomposition and architectural design that produce those notations.

USER INTERFACE PROTOTYPING

Prototyping of the user interface is the most effective way to elicit user requirements and to improve usability of applications.

Over the past few years, iterative prototyping of graphical user interfaces has become a central component of rapid application development. This step is supported by a wide variety of GUI builders and user interface management systems, including both stand-alone and RAD tools.

If software engineering is to mature as a discipline, standard specification and design notations have to be developed.

User interface prototyping is traditionally associated with the develop-

ment of interactive information systems. However, such prototyping also plays an important role in many embedded systems: copy machines, automated teller machines, and modern aircraft all contain software with an important user interface component. Accordingly, user interface design has become an essential part of software engineering.

Of all the various qualities of software design, none has proven over time to be more significant than modularity.

User Software Engineering was the first development method that combined iterative prototyping of the user interface, a systematic approach to software development, and a set of supporting tools.[3,4] USE introduced a three-part architecture for such systems, separating the user interface from system operations and the database, which allowed interface design to be treated independently. (This same architectural structure is now common in client-server systems.) Experience with such prototyping approaches has shown that the user interface is a key determinant of system usability and user satisfaction and that users working with the prototype were better able to contribute to the analysis process.

Over the past decade, GUIs have become nearly universal. It quickly became clear that the skills needed for designing good user interfaces are quite different from those needed for other aspects of software development. The needed expertise is likely to increase as new forms of interactivity, such as speech and gesturing, become more common. Many software product teams now include user interface experts and conduct extensive usability tests on their interfaces at an early stage of the development process. Applications for simulation, visualization, and charting are user-interface intensive, with half or more of the finished code related to the GUI. The same is true for games, where video, sound, and 3D graphics are a key part of the interactive experience.

Repeated prototyping of the user interface is a natural step when following a spiral approach to application development. For large projects, though, it is important to treat this prototyping activity as part of the analysis and design process, not as a shortcut to rapid system deployment.

MODULARITY AND ARCHITECTURE

Software architectures play a major role in determining system quality and maintainability.

The overall architecture of a software system has long been recognized as important to its quality (or lack thereof). An architecture's building blocks include units that carry out the system's behavior, connections that show how these units transmit or share information, and connections that show how the various program operations are activated, either sequentially or concurrently.

A well-structured system follows basic principles of software design. It should exhibit a logical and modular structure and support information hiding. Design decisions should be isolated from one another in a way that minimizes the impact of changes in those decisions. Each unit should have a published interface.

Of all the various qualities of software design, none has proven over time to be more significant than modularity. A system can be structured into a collection of modules, where each module has a well-defined interface to other modules, carries out a well-defined function, hides a design decision, and can be independently tested and verified. This structuring approach allows separation of concerns, which localizes design issues.

Mix and match. Through architectural design, a system can be divided into units. There are several partitioning approaches:

- modular decomposition—based primarily on assigning functions to modules;
- data-oriented decomposition—based on external data structures;
- event-oriented decomposition—based on events that the system must handle;
- outside-in design—based on user inputs to the system; and
- object-oriented design—based on classes of objects and their interrelationships.

One or more of these approaches can be used on the same system. For example, a client-server architecture may separate a system's user interface component from its database access component. The user-interface component can be partitioned further using event-oriented decomposition, outside-in design, and/or object-oriented design by selecting from a set of classes that provides the needed user-interface services.

These approaches often complement one another. For example, data-oriented design is a useful first step in approaching an object-oriented design. A skilled designer often knows several of these design approaches and can use the most appropriate one in a specific situation.

Design patterns. The notion of design patterns was introduced by Christopher Alexander in the context of architecture.[5]

According to Gamma and colleagues,[6]

> *"A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities."*

Furthermore, a design pattern describes appropriate conditions for its use in terms of other design constraints and tradeoffs.

Design patterns are different from class libraries and application development frameworks. Class libraries typically present a set of generally applicable classes, fully coded and documented, that can be used across a wide variety of applications. Application development frameworks provide a set of classes specialized for a particular type of application, such as an interactive information system (GUI interface to a relational database). Design patterns fall somewhere in the middle, and are being used to build reusable classes and frameworks for specific application domains.

Growing standardization. Although the notion of standard software architectures goes back at least to Structured Design,[1] advances in client-server software and object technology have raised great interest in this topic. In fact, the rapidly increasing use of standard architectures is a tangible result of 20 years of research and development on software engineering design. In particular, standardized designs and design patterns can provide the architecture for an object-oriented software system or subsystem, as well as provide an efficient way to incorporate reuse into software design. The development of component libraries, patterns, and frameworks indicates progress toward codification of design experience, and provides developers with valuable reuse capabilities. The architecture of certain types of systems is virtually predetermined, either because it is inherent in a set of development tools or because there is no justification to deviate from the mechanisms provided in a framework.

For example, client-server architectures are rapidly becoming standardized, both in terms of the application distribution and the layers of middleware used to enable communication between application parts.

Such standard approaches help reduce the effort required to build client-server applications. RAD tools such as Visual Basic and PowerBuilder are used to build graphical interfaces to relational databases, and a palette of graphical icons is used to design the user-interface elements (data fields, check boxes, and so on). The user interface is connected to the database through a standard API, such as Open Database Connectivity. The client is an executable Microsoft Windows application connected to a server where the DBMS runs. Both the application architecture and much of its content rely on predefined frameworks and component libraries.

Distributed object management also illustrates the power of standard architectures and components. Corba (Common Object Request Broker Architecture) contains an interface definition language for specifying object interfaces, and a messaging mechanism that lets objects in a heterogeneous environment request services from one another. The application architecture is built around an object request broker, which handles requests and responses. Corba is aimed at providing a standard style of application development that encourages the independent development of object services and the standardization of interface definitions.

LIFE CYCLE AND PROCESS

Having some *defined and manageable process for software development is much better than not having one at all.*

This statement may seem weak or insufficient. It may also seem that the notion of software process belongs at the beginning of the list of fundamental concepts. However, many high-quality software systems have been developed with little in the way of an organized and disciplined development process. Moreover, the great variations among application types and organizational cultures make it impossible to be prescriptive about the process itself. Thus, it appears that the software process is not as fundamental to software engineering as are abstraction and modularization.

Different types of software need differing degrees of development infrastructure. Figure 1 shows the contrast between enterprise-wide applications, for which controlled development is needed, and individual and department-level applications, for which RAD methods may suffice.

Great variations among application types and organizational cultures make it impossible to be prescriptive of the process.

Many small- to medium-sized systems can be built by one or two individuals who act as analyst, designer, programmer, documenter, and quality

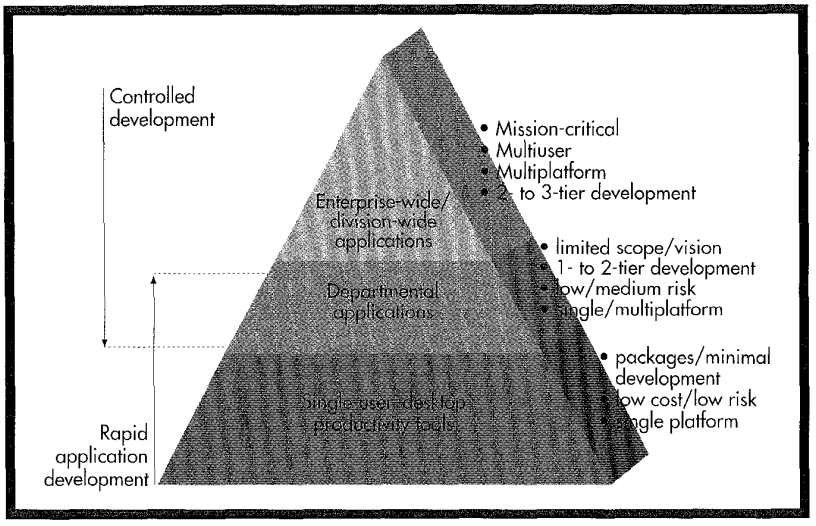


Figure 1. *Contrasting development approaches for different application scopes.*

assurance engineer. The tools for such a project may be no more than a text editor, a drawing tool, and either a programming environment, a RAD tool, or even a personal productivity tool. Such projects typically have no formally defined process and have little project management control other than a delivery date. The risk associated with project delays and errors is relatively small and usually confined to a small work group.

Larger, more complex systems are likely to involve numerous developers and other stakeholders (customers, end users, and so on) over months or years. Such systems are frequently critical to the operation of their organizations or as part of products. The tools needed for these projects go well beyond those needed for smaller projects and typically include analysis and design tools, project and process management tools, test case generators, debuggers, configuration management and version control systems, as well as other tools specialized for the type of application.

Successful development of larger systems requires a well-defined process supported by effective management oversight and application development tools. These requirements are imposed by the greater risk associated with failures and delays in the development of mission-critical systems. The nature of the process itself varies greatly among organizations and even among different groups in an organization that is building different types and sizes of applications, often using different technologies.

Closely linked to the software development process is the product being produced by the process—typically a set of executable programs with libraries, examples, documentation, and installation procedures. Managing the process includes maintaining control (configuration management) over the items being used and created. Each task comprising the software process uses, creates, and/or modifies items such as requirements, specifications, external documentation, test cases, architectural and detailed design, source code, object code, and test drivers, as well as class libraries and items reused from other projects. All of these items may also have versions to be distinguished, tracked, and saved.

The Software Engineering Institute has been a particularly effective catalyst for process improvement, encouraging organizations to evaluate their process maturity and create software process groups to improve it. Achieving a rating of SEI Level 3 (defined, repeatable process) is an important goal for many organizations.

REUSE

Reuse of existing software development assets is an essential part of any software development process.

Software reuse is a long-standing notion. For example, the first operating systems had application programs that provided common services to all programs, thereby eliminating the need for each programmer to program them. Similarly, one of the attractions of Fortran (dating to 1957) was its support for standard input and output streams through a runtime library, as well as its built-in mathematical functions. Beyond those features, numerical mathematicians developed reusable libraries of mathematical functions.

These basic concepts of reusable functionality are now taken for granted, and have been significantly extended. Operating systems have large libraries of system services and published APIs that allow programs to access those services. Programming environments include not only built-in function libraries, but also class libraries.

A broad notion of reuse, however, goes well beyond APIs and class libraries to encompass many types of artifacts associated with a software development project. Since most APIs and class libraries are independent of a particular application domain, software development organizations must also look at domains specific to their own needs and practices. Once found, these items should be treated as corporate assets; the investment that went into creating them must be recognized. Such items include not only source and object code, but also design patterns, document templates, test scripts, user interface layouts, and software process definitions.

Effective reuse beyond the level of function and class libraries has proved to be more difficult than hoped. Many software artifacts built for a specific project need additional work before they can be of general use. This might entail generalizing components, standardizing documentation, and storing the artifacts in a controlled repository. The time pressure of projects rarely allows for this extra work. Furthermore, it often takes

several attempts at building similar systems before broadly reusable items can be identified and defined. For example, the inherent conflict between generality and specificity affects how many parameters are provided for the item. In short, it's hard to build high-quality reusable components.

Several key steps, including consciousness raising, are required for a successful reuse program. The most important step seems to be a dedicated organizational effort, with resources allocated specifically to the reuse program. (*IEEE Software*, Sept. 1994, covers many aspects of reuse.)

METRICS

Improvements in the software development process and system quality cannot be evaluated without an effective metrics effort.

Every organization wants to improve its software development process because there are tangible benefits associated with building better software. The notion of continuous process improvement is central to business process reengineering activities, and is a basic component of the SEI's Capability Maturity Model.

Desired process improvements include reduced development time, more accurate estimation of costs and schedules, reduction of reported problems in released software, better communication between users and analysts, increased use of analysis and design methods, increased reuse, better internal and external documentation, earlier and more accurate performance prediction, tighter control over intermediate and final products, tighter integration of quality assurance activities with development activities, and inclusion of usability engineering.

Unfortunately, it's impossible to measure progress toward such goals without a well-defined set of items to be measured and accurate measurements of current practice. There is broad agreement on the need for organizations to measure key aspects of their software development activities, as well as product quality, including usability.

In the area of quality, Capers Jones suggests measuring defect quantities in deliverables by product phase, efficiency of defect removal during reviews and inspections, defects delivered in released systems, defect severity, and customer/user satisfaction.[7,8] In the area of productivity, Jones suggests a variety of schedule, resource, and cost measures, along with functional metrics (function points) as a way to determine productivity.

Different types of metrics are associated with each software development task. In design, for example, one class of measurement is associated with defects, such as interface definition errors; another, with "goodness," or quality. This idea goes back to Structured Design, which introduced numerous metrics such as cohesion, coupling, fan-in, fan-out, module size, and scope of effect.

Metrics are well established in testing and quality assurance. Thoroughness of testing is given in terms of code coverage measures, such as branch coverage, loop coverage, and relational coverage. Metrics have also been thoroughly defined for cost estimation.[9,10]

TOOLS AND INTEGRATED ENVIRONMENTS

The application development environment and its tools should provide comprehensive and integrated support for the development process.

The software development process must be supported by appropriate software development tools. An effective development environment provides automated tools for as many tasks as possible.

Current development environments are diverse, ranging from single users on personal computers to large, geographically dispersed teams using a variety of machines to share development information. This diversity reflects the wide range of development processes and methods, the availability of different tools on different hardware platforms, and the vast range of applications being developed. While the specific tools used by an organization vary by application domain, implementation language, and operating system, the objective is to have an integrated set of tools that work together harmoniously to make the application development process more effective.

The services provided by a software engineering environment have been categorized into frameworks that serve as the basis for describing, comparing, and contrasting existing and proposed environments.[11,12] The environment infrastructure provides a set of services that typically cover object management, process management, communication, and operating systems.

Effective reuse beyond the level of function and class libraries has proved to be more difficult than hoped.

Such a model provides a product-neutral way to categorize tools and environments. Organizations seeking to acquire or develop comprehensive environments can look to these models as checklists, recognizing that it is nei-

ther economical, technically possible, nor organizationally feasible to acquire a compatible set of tools that provides all of these services.

Much work is needed before comprehensive multivendor, multiplatform tool integration can be achieved.

One important barrier in current software engineering environments is the difficulty of integrating tools that address different aspects of the development process. The general lack of standards for tools has slowed the creation of fully integrated environments. Although there are numerous open-systems standards for operating systems interfaces, network interconnections, and programming languages, few such standards exist at the level of design methods and tool integration.

Typically, software tools are produced by different vendors who each focus on a specific part of the overall problem. For example, different tools support architectural design and test case generation. Users want these tools to work together. Tool integration should produce complete environments that support the entire software development life cycle.

Tool integration issues fall into five categories:[13]

- Platform integration: the ability of tools to interoperate on a (potentially heterogeneous) network.
- Presentation integration: commonality of the user interface among tools.
- Process integration: linkage between tool usage and the software development process.
- Data integration: the ways in which tools share data.
- Control integration: the ability for tools to notify and initiate actions among one another.

The goal of data integration is to provide tools with a shared repository, where all tools share an extensible common data model and a public interface that supports object management services. This approach was first proposed for the 1980 US Defense Department report "Requirements for Ada Programming Support Environments" (nicknamed "Stoneman"). The first commercial environment to provide such a shared repository was IDE's Software through Pictures, circa 1985.[14] IBM's AD/Cycle concept, never fully implemented, was based on a repository manager and a common data model.

The notion of control integration originated with the Pecan and Field environments.[15,16] The first commercial version of such a mechanism was the Broadcast Message Server in Hewlett-Packard's Softbench.[17] When control integration is combined with data integration, events such as access to or changes to the shared repository can send messages to other tools (services), which can in turn trigger other repository activities or messages. This control integration mechanism is also found in Corba.

The factors surrounding tool integration are complex, and often involve both technical and business issues. The absence of standards means that most tool integration is done via "point-to-point" connections, with relatively little use of frameworks. In the absence of standards for repository models and control integration messages, much work is needed before comprehensive multivendor, multiplatform tool integration can be achieved.

Considerable progress has been made in the past few years toward applying these eight fundamental ideas and improving the software engineering process. For example, the widespread acceptance of object-oriented techniques and the demand for APIs draw from the fundamental ideas of abstraction and modularity. Many of the concepts underlying the Internet and the World Wide Web may be viewed as applications of the same ideas. Similarly, today's RAD tools came into existence because their creators recognized the value of reusable components and standard architectures and incorporated them into a visual tool. Java is particularly interesting because it facilitates large-scale reuse through its built-in class libraries and provides machine independence through a layer of abstraction (the Java virtual machine). Collectively, such tools help to reduce the time needed to build and to modify systems, as well as changing the associated software engineering process.

The premise that an engineering type of discipline should be applied to software design and development resulted in the term *software engineering* in 1967. Experience over the past three decades has shown that engineering can be applied successfully to the construction of complex software systems.

However, software engineering has a long way to go before it is widely recognized as an engineering discipline. First, there is only limited agreement on the skills needed by a software engineer, despite proposals dating from the 1970s[18] and ongoing committee efforts. Few software engineering curricula and degree programs exist, even though curriculum proposals also date from the 1970s.[19] Issues of certification, registration, and licensing of software engineers have been similarly discussed for two decades.

To be sure, there is a "chicken-and-

egg" aspect to these discussions. If a respected professional organization were to establish certification and/or registration criteria for software engineers, that would create a de facto set of required skills and practical experience. Such requirements would certainly lead to numerous academic programs for aspiring software engineers, as well as professional development programs required for ongoing certification or licensing.

Consistent use of software engineering practices will not emerge until the field is more mature. The maturation process requires consolidation of existing development methods and tool integration mechanisms, quantitative data on best practices, and widely accepted agreement on the intellectual content of the field of software engineering, supported by both academic and professional programs for practicing software developers. ◆

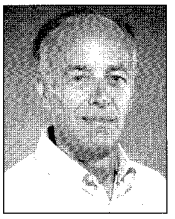
ACKNOWLEDGMENT

This paper is based on the Inaugural Stevens Lecture on Software Development Methods, presented by the author at the Seventh International Workshop on Computer-Aided Software Engineering (CASE '95) in Toronto, July 1995.

An expanded version of this article will soon be published as a briefing by the IEEE Computer Society Press.

REFERENCES

1. E. Yourdon and L.L. Constantine, *Structured Design*, Prentice Hall, Englewood Cliffs, N.J., 1979.
2. D.L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Comm. ACM*, Dec. 1972, pp. 1,053-1,058.
3. A.I. Wasserman, "The User Software Engineering Methodology: An Overview," in *Information Systems Design Methodologies: A Comparative Review*, T.W. Olle, A.A. Verrijn-Stuart, and H.G. Sol, eds., North-Holland, Amsterdam, 1982, pp. 591-628.
4. A.I. Wasserman et al., "Developing Interactive Information Systems with the User Software Engineering Methodology," *IEEE Trans. Software Eng.*, Feb. 1986, pp. 326-345.
5. C. Alexander, *The Timeless Way of Building*, Oxford Univ. Press, New York, 1979.
6. E. Gamma et al., *Design Patterns: Elements of Object-Oriented Software Architecture*, Addison-Wesley, Reading, Mass., 1995.
7. C. Jones, *Applied Software Measurement*, McGraw-Hill, New York, 1991.
8. C. Jones, *Assessment and Control of Software Risks*, Prentice-Hall, Englewood Cliffs, N.J., 1994.
9. B.W. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, N.J., 1981.
10. B.W. Boehm et al., "The COCOMO 2.0 Software Cost Estimation Model—a Status Report," *Amer. Programmer*, July 1996, pp. 2-17.
11. European Computer Manufacturers' Association, *A Reference Model for Frameworks of Software Engineering Environments (Version 3)*, ECMA Report No. TR 55 (Version 3), 1993. Reprinted as *NIST Special Publication 500-211*, Nat'l Inst. of Standards and Technology, Gaithersburg, Md., 1993.
12. I. Thomas and B. Nejmeh, "Definitions of Tool Integration for Environments," *IEEE Software*, Mar. 1992, pp. 29-35.
13. A.I. Wasserman, "Tool Integration in Software Engineering Environments," in *Software Engineering Environments*, F. Long, ed., Springer-Verlag, Berlin, 1990, pp. 138-150.
14. A. Wasserman and P. Pircher, "A Graphical, Extensible Integrated Environment for Software Development," *Proc. ACM SIGSOFT/SIGPLAN Symp. Practical Software Development Environments*, *ACM SIGPLAN Notices*, Vol. 22, No. 1, Jan. 1987, pp. 131-142.
15. S.P. Reiss, "PECAN: Program Development Systems that Support Multiple Views," *Proc. 7th Int'l Conf. Software Eng.*, ACM, New York, 1984, pp. 324-333.
16. S.P. Reiss, "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software*, July 1990, pp. 57-66.
17. M. Cagan, "HP SoftBench: An Architecture for a New Generation of Software Tools," *Hewlett-Packard J.*, June 1990, pp. 36-47.
18. P. Freeman, A.I. Wasserman, and R.E. Fairley, "Essential Elements of Software Engineering Education," *Proc. 2nd Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1976, pp. 116-122.
19. P. Freeman and A.I. Wasserman, "A Proposed Curriculum for Software Engineering Education," *Proc. 3rd Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1978, pp. 56-62.



Anthony I. (Tony) Wasserman was founder and chairman of Interactive Development Environments, Inc., developer of the *Software through Pictures* CASE environment. He also served as IDE's president and chief executive officer from 1983 to 1993. Prior to starting IDE, he was a professor of medical information science at the University of California, San Francisco, and a lecturer in the Computer Science Division at the University of California, Berkeley. He is currently an independent consultant on a broad range of software development topics, focusing on methods and tools.

Wasserman received an AB in mathematics and physics from UC Berkeley and a PhD in computer sciences from the University of Wisconsin–Madison. He is a Fellow of the ACM and IEEE.

Address questions about this article to Wasserman at TWasserman@aol.com.