Chapter 1. Introduction to IEEE Std. 1517—Software Reuse Processes

1.1 Reuse is Boring!

Software reuse is the process of building or assembling software applications and systems from previously developed software parts designed for reuse. Software reuse is practiced to save time and money, and to improve quality.

Although a potentially powerful technology, reuse has never been counted among the most interesting software topics. In truth, most software professionals consider reuse downright boring. After all, who would find a worn-out subject as old as programming itself interesting? For years, we have been hearing about the benefits reuse offers, but have yet to see them realized in practice. Even when the popularity of object-oriented development (OOD) brought the notion of reuse to the forefront, the software community was disappointed because less object reuse was achieved than expected. Reuse, it seems, is not an automatic byproduct of OOD.

1.2 The Next Form of Reuse

In the future, however, reuse is likely to elicit a very different reaction. The next form of reuse will be the key enabler of the world trade of software via the World Wide Web. Reuse via the Web has already captured the imagination of the software industry and business community at large. After all, who would not be interested in the primary enabler of an emerging multi-billion dollar industry that promises to turn the traditional software development approach upside down?

The next form of reuse centers on components and component-based development. Figure 1-1 shows the projected growth for the component industry over the next few years.[1] Components are expected to be the primary driver of the dramatic changes about to take place in software development.

Components lie at the very heart of the future vision of computing. Corporations expect that they soon will be running their businesses using Web-enabled, enterprise business applications composed from predefined, reusable, and replaceable components distributed over networks. Although part of the application may run on a client, part on the middle-tier, and another part on a backend database server, its comprising components—written in different languages and supplied from multiple sources—will work together to perform the application's services.

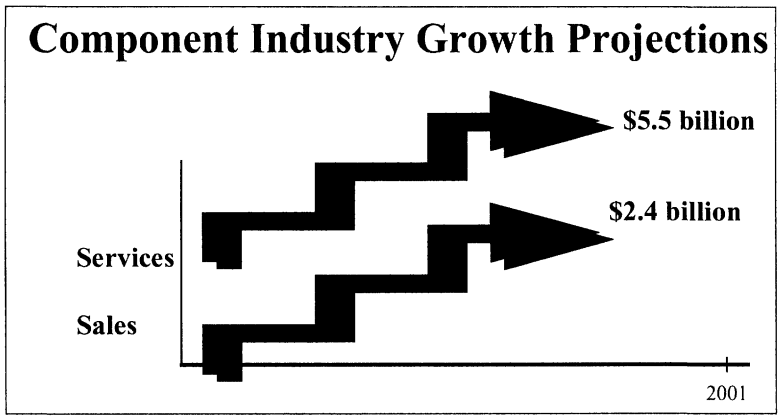

Figure 1-1. Between 1998 and 2001, component sales are predicted to increase from $1.1 billion to $2.4 billion, and related services from $2.2 billion to $5.5 billion.

Component-based applications offer the advantages of being both easily customized to meet current business needs and easily modified to meet changing business needs over time. Also, they leverage a corporation's investment in its legacy systems by containing valuable existing functionality wrapped into reusable components. Thus, component-based applications are likely to be composed of an interacting mixture of pre-developed components that preserve the business' core functionality and new components that take advantage of the newest technologies, such as the Internet. Today, examples of components include objects written in languages such as Smalltalk, C++, and Java, and other software parts such as Active X controls and design frameworks.

1.3 Components

A *component* may be thought of as an independent module that provides information about what it does and how to use it through a public interface, while hiding its inner workings. The interface identifies the component, its behaviors, and interaction mechanisms.

The idea of components is not new. Fundamental to the component concept are the predecessor software engineering concepts of program modularization, structured programming, and information hiding, which were introduced in the 1960s by Edgar Dijkstra, David Parnes, and others.

Although as of yet there is no industry consensus on the definition of a component, there is some agreement on the properties that a component is expected to have. For instance, a component should provide a set of common functionality that may be used as a self-contained building block in the construction of software applications. In addition, a component should have a well-defined interface, hide implementation details, fit into an architecture, be easily replaced, and have the

ability to inter-operate with other components. Finally, a component should be reusable in many different software applications and systems.

Szyperski's definition of a component emphasizes the importance of the interface and context specification of a component as well as its independent nature:

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*[2]

Table 1-1 is a list of components compiled from several recent software publication articles on the topic. The components listed include examples of small-grained, desktop-bound visual components (e.g., GUI widgets), as well as large-grained components that capture a complete business function (e.g., a shipping component that includes order entry and shipment functionality).

The list of components illustrates how the industry's notion of components has evolved over the last decade. At first, the term *component* typically was used to refer to compound documents (e.g., OpenDoc components and Microsoft's Object Linking and Embedding (OLE) components), and then also to refer to binary and source code components (e.g., Active X Controls and CORBA components). Later, due to the market's drive to expand component technology beyond the desktop to the server, the concept of components was enlarged to include more abstract and larger-grained components, such as design templates, frameworks, and even application packages. In contrast to earlier components, most of which were visual and client-based, these types of components (e.g., Enterprise JavaBeans, SAP AG's R/3 Package, IBM's San Francisco Project, VISIX Software's Galaxy, Template Software's SNAP) are often non-visual, backend, and server-based. They implement infrastructure services such as event notification, backend services such as transaction processing, and business functions such as shipping.

1.3.1 Beyond Objects

Today, components are seen as a step beyond objects. Like objects, components use contractually specified interfaces to implement the requirement of plug-compatibility with other components. But they go beyond objects by better addressing the requirement for replaceability and by enabling the reuse of software parts that are larger-grained and at higher levels of abstraction.

A noted shortcoming of objects is that they do not scale up well when used to build large, complex application systems because they are too fine-grained and at too low an abstraction level. Another shortcoming is that system modification, maintenance, and testing can be difficult because of inheritance and behavior overriding. Replacement of an object with a new object that implements changes to the business may impact all other objects that inherit properties of the replaced object and thus may lead to extensive re-testing.

In contrast to objects, components can be large-grained as well as small-grained software parts. Large-grained components (enterprise components) encapsulate major chunks of an application's functionality in an independent, reusable, and easily replaceable unit. Because of their size, fewer enterprise components are needed to construct an application than objects. This solves the scaling problem with fine-grained components where integration and assembly become very tedious because of the number of parts needed to construct a large application.

Enterprise components maximize the potential benefits that reuse can deliver by making reuse practical in large-scale system projects. Also, because they are independent units that encapsulate a complete function, enterprise components also solve the replaceability problem. When changes in the business occur, the old component can be easily pulled out and replaced with a new component that satisfies the new business requirements with little or no impact on other parts of the system.

Table 1-1. Examples of Components

- Calendars: On screen widget
- Button for a user screen: List Box, Dialog Box
- Order entry process (full application)
- CICS transaction call
- Customer service screen
- Customer management component
- Validation component that interacts with multiple DB2 tables
- Customer management component
- Workflow component
- Shipping component (order entry and shipment specification)
- Charge amount
- Business Associates
- Container type
- Unit of measure
- Event notification (infrastructure service)
- User interface controls
- Inter-application communication protocols
- Customer component
- Invoice component
- Order component
- Security
- General ledger
- Asset depreciation component
- Letter-of-credit
- Manufacturing work-in-process module
- Inventory tracking application

1.4 Component-Based Development

So strong is the software community's belief in components, that it may be simply a matter of time before component-based development becomes the dominant software development approach. Component industry projections are that 61 percent of all new applications will be developed using components by the year 2001.[3]

Component-based development (CBD) is the latest embodiment of reuse. It is an assembly approach to software development in which software applications are constructed by means of assembling components. Some of these components may be predefined components housed in libraries or supplied by other sources such as external vendors, some may be harvested from existing systems and applications, and others may be developed anew for the project at hand. CBD may also be described as an architecture-driven software development approach, where an *architecture* is a generic structure or high-level design that is intended to be used to build a set of related software products or systems. The architecture provides a framework for assembling the components into a software application.

The strategy underlying CBD is to use predefined software components and architectures to eliminate redesigning and rebuilding the same software structures over and over again. Because of this underlying strategy, not only does the CBD approach imply reuse, it demands reuse to deliver any significant gains in software productivity, quality, and development speed. Since reuse lies at the very heart of the CBD approach, this development method must be guided by the principles of reuse.

1.4.1 Benefits of Component-Based Development

It has long been recognized that reuse is a powerful technology that potentially can deliver tremendous benefits. Exploiting component-based development can provide very significant software productivity, quality, and cost improvements to an organization. Table 1-2 lists the benefits of component-based development, which are really just a repeat of the benefits that have always been promised by reuse.

1.4.2 Delivering Quality

One of the most important benefits that reuse delivers is quality. Among all the powerful software technologies available today, software reuse is the best way to accelerate the production of high quality software. What sets reuse apart is its ability to provide the benefits of faster, better, and cheaper without compromise. With the exception of reuse, all other software technologies require a trade-off of possible benefits (e.g., faster software development at the expense of software quality). According to the Gartner Group's findings, reuse is the only technology that allows a company to simultaneously address software cost, time-to-market, flexibility, and quality.[6] Reuse enables a company to achieve both higher quality systems and lower costs, hence gaining a competitive advantage in the marketplace.

Table 1-2. Benefits of Component-Based Development[4,5]

- Deploy critical software applications more quickly
- Simplify large-scale software development
- Encapsulate business services into reusable application logic
- Shorten software development cycles
- Reduce the amount of new code to write
- Allow software applications to share functionality
- Make software applications more adaptable; easier to change
- Decrease software complexity
- Increase software reliability and overall quality
- Increase software productivity by reducing costs

1.4.3 Exploiting Reuse Benefits

Not only are there tremendous benefits to be gained from reuse, but also there are tremendous opportunities to employ reuse in software projects. Analysis of software applications and systems has shown that they are composed of similar parts. In general, it is reasonable to expect that 60 to 70 percent of a software application's functionality is similar to the functionality in other software applications, that 40 to 60 percent of its code is reusable in other software applications, and that 60 percent of its design is reusable in other software applications.[7] Therefore, the majority of almost any software application can be assembled from predefined components, provided those components were designed to be "plug compatible."

Components are the software industry's latest attempt to capitalize on this similarity. Backed by advances in technologies and tools, components offer the best chance yet to achieve a significant level of reuse in industrial-strength application development projects.

1.5 Components and Standards

However, there is one catch. *Standards.* The success of the components industry is totally dependent on standards. For example, it is obvious that interoperability standards are a basic necessity. Interoperability standards are necessary to be able to assemble components from different sources into working applications and systems.

Components are expected to communicate with one another and to use each other to provide their services or functionality. A standard component interoperability model assures components written in different languages, located in different places, and running on different platforms and on different machines or address spaces can be used together, sharing data and capabilities.

There currently are three *de facto* standards for component specification, interoperability, and distributed computing:

1. Object Management Group's Common Object Request Broker Architecture/Internet Interoperability Protocol (OMG's CORBA/IIOP): for CORBA components that are written in different languages such as C++, Visual Basic, and Java and run on multi-

ple distributed platforms

2. Microsoft's Component Object Model/Distributed Component Object Model (COM/DCOM): for ActiveX controls that can be built in different languages such as C++, Smalltalk and Java, and run on a Windows environment
3. Java/JavaBeans/Enterprise JavaBeans: for JavaBeans that are built in Java and run on all environments that support the Java virtual machine

IIOP is the CORBA message protocol used to provide component communication over the Internet or an Intranet. Remote Method Invocation (RMI) is used to enable distributed Java components to communicate with one another.

The fact that there are three such interoperability models rather than one is not considered a problem by companies attempting to implement component technology because the three models are quite similar and are bridged by tools.[8]

1.6 Process Standards

Although interoperability standards ensure that components will fit together, they are not enough to ensure the success of a global components industry. Software developers also need process standards that detail how to identify, analyze, design, implement, test, deploy, maintain, and evolve high-quality components and component-based applications.

Process standards serve two important functions to enable the world trade of software components:

1. Establishment of a common understanding of the software process between software producers and software consumers
2. Assurance of the quality of software components and component-based applications

1.6.1 Improving Communication

First, consider the issue of understanding between software producers and software consumers. In this context, a software producer refers to a software developer or vendor who provides software products (e.g., software systems, applications, or components) to a software consumer (e.g., user or software developer who uses the software product to build a new software product). Software producers and software consumers often do not speak the same language. When they enter into a relationship where one agrees to provide software to the other, it is very difficult to communicate product and project requirements unless the process to be used is well understood by all parties involved. The world trade of software complicates this problem because software producers and software consumers may not know of one another. Also, they may be members of different organizations and can be located in different places around the world.

A standard can foster an improved understanding between software consumers, software producers, and everyone else involved in the life cycle of software products, regardless of these complications. As an example, consider the user (or manager) who wants a CBD approach employed to develop a software product. Having a standard that specifies what is required in a software life

cycle model to enable CBD can clarify for the user (or manager) and the developer what is entailed in such an approach. The standard specification of what is required in the life cycle model can even be used as the basis for an informal or legally binding contractual agreement between the parties.

1.6.2 Assuring Quality

Second, quality has always been an important software issue. The world trade of software makes quality an even more important software issue. The business of global reuse of components via the World Wide Web cannot succeed unless the quality of components can be assured. Component consumers will demand quality assurance from component producers. Interface standards are not enough to guarantee component quality.

Historically, the software industry has followed the Deming school of total quality management which espouses using a better quality process to produce a better quality product. Empirical evidence has shown that the quality of the software life cycle process used directly affects the quality of the software it produces.[9,10] In other words, software quality follows from software process quality. Process standards that capture the best-known software practices are considered the best means available to assure the quality of the software life cycle process and, hence, the quality of the software product. Like any software user, component consumers will demand that component producers use software processes and practices that result in high quality software components and component-based applications. The components industry must rely on process standards and product quality standards as important means to assure the quality of components and component-based applications.

1.7 IEEE Std. 1517—Reuse Processes

IEEE Std. 1517—Standard for Information Technology—Software Life Cycle Processes—Reuse Processes is the process standard for reuse and for CBD.[11] It is a requirements specification for practicing reuse and CBD on an enterprise-wide basis. IEEE Std. 1517 identifies the processes involved in practicing software reuse and describes, at a high level, how the processes operate and interact during the software life cycle. Also, it defines reuse terminology. Because IEEE Std. 1517 addresses both the development and maintenance of software with the use of predefined reusable software parts and the development and maintenance of reusable software parts, it is applicable to CBD. In this standard, reusable software parts are called *assets*.[12]

1.7.1 Purpose of IEEE Std. 1517

In general, the purpose of IEEE Std. 1517 is to provide the basis for the incorporation of reuse into the software life cycle. More specifically, its purposes are to:

- Establish a framework for practicing reuse within the software life cycle
- Specify the minimum set of processes, activities, and tasks to enable the practice of reuse when developing or maintaining software applications and systems

- Define the input and output deliverables required and produced by these reuse processes
- Explain how to integrate reuse processes, activities, and tasks into the ISO/IEC & IEEE/EIA 12207 Standard software life cycle framework
- Improve and clarify communication between software producers and software consumers regarding reuse processes and reuse terminology
- Promote and control the practice of software reuse to develop and maintain software products

It is important to emphasize that this standard does not define one specific, rigid software life cycle that must be adhered to. Instead, IEEE Std. 1517 is more correctly viewed as simply a specification of the minimum requirements that a software life cycle must meet to include reuse. In other words, it provides the software community with a clear and explicit specification as to what is required in a software life cycle to accomplish the practice of reuse and to enable CBD.

Software developers will shy away from a standard that dictates one specific process and rightly so, because this is not a one-size-fits-all situation. It is a well-known fact that different software projects and different organizations have very different software life cycle requirements and preferences. However, going to the other extreme where there is no reuse process standard whatsoever leaves the software industry without any guidance as to what is needed to practice reuse or any means to evaluate and choose among software products and service providers that claim to enable and support reuse. The result of having no clear idea of what is needed to practice reuse is a history of little reuse.

Because this standard is a requirements specification rather than an implementation of reuse processes, IEEE Std. 1517 is open and can be used with virtually any software life cycle offered by a vendor or developed in-house by an organization. Organizations and developers who do not want a specific set of reuse processes forced upon them have nothing to fear and much to gain from IEEE Std. 1517.

1.7.2 Uses of IEEE Std. 1517

Like all IEEE standards, the use of this standard is voluntary. An individual organization may choose to adopt IEEE Std. 1517 as a means of improving its software processes. For example, an organization that desires to use the software reuse practices that have been deemed by consensus to represent the "best of breed" may choose to adopt this standard.

Alternatively, an organization may decide to adopt this standard as a way to assert that its software development process conforms to the best-known software reuse practices. Some reasons to conform to this standard include:

- Improve the software life cycle processes used to develop and maintain software applications and systems
- Adopt the best software reuse practices
- Adopt a CBD approach
- Improve the quality of software applications and systems developed

- Decrease the costs of developing and maintaining software applications and systems
- Decrease the time required to develop and maintain software applications and systems
- Understand software reuse terminology
- Increase competitive advantage of software applications and systems
- Extend a software life cycle model to include reuse processes, activities, and tasks

1.7.3 Application of IEEE Std. 1517

IEEE Std. 1517 is written for both managers and technical personnel that are involved in acquiring, supplying, or developing software applications and systems or reusable assets. IEEE Std. 1517 applies to:

1. the acquisition of software and software services
2. the acquisition of assets
3. the supply, development, operation, and maintenance of software applications and systems using a CBD approach
4. the supply, development, management, and maintenance of assets
5. the establishment of a systematic reuse program and components strategy at the organization or enterprise level

1.8 Placement of IEEE Std. 1517 in the IEEE Software Engineering Standards Collection

As Table 1-3 shows, software engineering standards have been produced by several organizations.[13] At the time of writing, IEEE Std. 1517 is one of more than 300 software engineering standards that have been developed and maintained by more than 50 different standards organizations. The IEEE Computer Society is responsible for the creation and maintenance of approximately 50 standards. IEEE Std. 1517 is a member of the IEEE Computer Society Software Engineering Standards Committee (SESC) collection of standards. IEEE Std. 1517 has been designed to fit into the SESC standards collection as a practice standard; i.e., a description of the best software reuse practices. It provides the reuse processes requirements to be met by a software life cycle model that claims reuse support.

1.8.1 SESC

The IEEE Software Engineering Standards Committee (SESC) was formed in 1976.[14] Its mission is to plan, improve, and coordinate a collection of software engineering standards whose function is to prescribe the norms of software engineering practice.

> *Software engineering is: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).*[15]

Since its formation, the number of SESC software engineering standards has grown from one to nearly 50.[16] Software engineering standards cover the various software life cycle processes such as design, testing, and maintenance; the various aspects of software quality such as quality assurance, performance monitoring, and reliability; and supporting functions such as training, project management, and configuration management.

To better manage and make standards easier to use, the SESC is reorganizing its standards collection. The reorganization is based on an architecture comprised of five classes of objects: customer, process, agent, resource, and product.[17] This object-oriented view represents software engineering as being performed by a project that consists of a set of agents. The agent interacts with the customer and uses resources to perform a process that produces a product.

Table 1-3. Examples of Organizations that Produce Software Engineering Standards

ORGANIZATION	ORGANIZATION'S NAME
US National Standards	
ASTM	American Society for Testing and Materials
EIA	Electronic Industries Association
ANS	American National Standard
IEEE	Institute of Electrical and Electronics Engineers
US Government Standards	
NIST	National Institute of Standards and Technology
DoD	Department of Defense
NASA	National Aeronautics and Space Administration
International Standards	
ISO	International Organization for Standardization
IEC	International Electrotechnical Commission

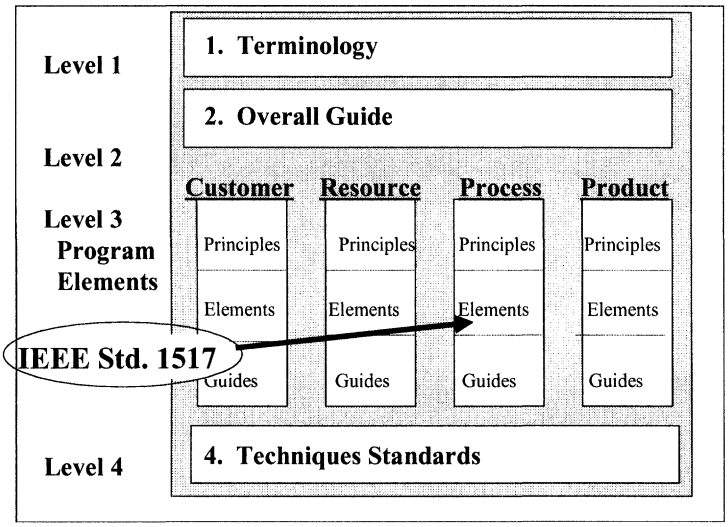

Figure 1-2. The SESC Standards Architecture

1.8.2 SESC Standards Architecture

The SESC Standards Architecture is depicted in Figure 1-2.[18] The first level of the SESC standards architecture contains software engineering terminology standards and the taxonomy of the standards.

The second level contains an overall guide to the SESC collection of standards. It describes the relationships among the SESC standards.

The third level consists of four three-layer stacks (or the program elements). The four stacks contain standards regarding the customer, process, product, and resource objects of software engineering. Each stack is divided into three layers representing three types of standards:

- Layer 1: *Principles*—Principle or policy standards describe objectives for standards that belong to a stack
- Layer 2: *Elements*—Element standards provide high-level guidelines of software engineering activities that belong to a stack
- Layer 3: *Application Guides and Supplements*—Guides provide guidance on how to apply Element Standards that belong to a stack

The fourth level of the standards architecture contains the Techniques Standards. These standards describe software-engineering techniques that are broadly applicable (e.g., can be used by several life cycle processes or activities).

IEEE Std. 1517 is classified as a *process elements* standard. As shown in Figure 1-2, it belongs at Level 3 and in Layer 2 of the Process Stack in the SESC Standards Architecture.

1.9 Supplement to ISO 12207

IEEE Std. 1517 is a supplement to the ISO and IEEE 12207 Standard for Information Technology—Software Life Cycle Processes.[19] IEEE Std. 1517 identifies the reuse processes and explains how they are integrated into the ISO 12207 life cycle process framework. (Note that the ISO 12207 framework and the IEEE 12207 framework are the same. Therefore, when reference is made to the ISO 12207 framework, it refers to both versions of the standard.)

A 24 September 1997 meeting letter of the IEEE Software Engineering Standards Committee states: "The ISO 12207 framework is considered the strategic definition of the software life cycle model." (See Chapter 2, Figure 2-2 for the 12207 Life Cycle Framework.) However, the ISO 12207 Standard is not sufficient by itself to enable and support the practice of reuse because it does not explicitly define reuse process requirements for the software life cycle. IEEE Std. 1517 was created to make reuse an explicit part of the software life cycle. (See Chapter 3, Figure 3-1 for the 1517 Life Cycle Process Framework.)

The writers of IEEE Std. 1517 designed it as a supplement that is integrated into the ISO 12207 specification. Desiring to practice what they preached, the writers of IEEE Std. 1517 reused the life cycle framework and process descriptions from the ISO 12207 Standard, rather than recreate a new framework to show how to augment the software life cycle with reuse processes. (See Chapter 8, Figure 8-1 for the 1517 Framework and Processes.) This means that compliance with IEEE Std. 1517 also results in compliance with the ISO (and IEEE) 12207 Standard. This also means that it is impossible to understand or apply the IEEE Std. 1517 without also understanding the ISO 12207 Standard—both its framework and its contents. For this reason, Chapter 2 is devoted to a detailed explanation of the ISO 12207 Standard.

1.10 References

1. D. Kara, "Build vs. Buy: Maximizing the Potential of Components," *Component Strategies,* July 1998, pp. 22–35.
2. Szyperski, *Component Software: Beyond Object-Oriented Programming,* Addison Wesley Longman, 1998, P. 34.
3. B. Ambler and S. Venkat, "Large-Grained Components & Standards: Perfect Together," *Component Strategies,* Oct. 1998, pp. 32–46.
4. R. Levin, "Components on Track," *Information Week,* 5 June1998, pp. 93–98.
5. M. Buchheit and B. Hollunder, "Building and Assemblying Components," *Object Magazine,* Nov. 1997, pp. 62–64.
6. *Software Reuse Report,* Gartner Group, Stanford, Conn., 1995, p.7.
7. W. Tracz, *Tutorial: Software Reuse: Emerging Technology,* IEEE Computer Society Press, Los Alamitos, Calif., 1988.
8. N. Ward-Dutton, "Componentware Turns the Corner," *Application Development Trends,* July 1998, pp. 18–19.

9. P. Lawlis et al., "A Correlation Study of the CMM and Software Development Processes," *CrossTalk,* Sept. 1995, pp. 21–25.

10. *CMM Summary*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., SEI Web page: sei.cmu.edu.

11. *IEEE 1517, Standard for Information Technology—Software Life Cycle Processes—Reuse Processes*, IEEE, Piscataway, N.J., 1999.

12. Ibid.

13. J. Moore, *Software Engineering Standards*, IEEE Computer Society, 1998, pp. 267–276.

14. Ibid., 1998, pp. 34–57.

15. *IEEE Std. 610.12, Standard Glossary of Software Engineering Terminology*, IEEE, Piscataway, N.J., 1990.

16. J. Moore, *Software Engineering Standards*, pp.34–57.

17. Ibid., p. 23.

18. Ibid., p. 23.

19. *ISO/IEC 12207—1996 Standard for Information Technology—Software Life Cycle Processes*, International Organization for Standardization and International Electrotechnical Commission, Geneva, Switzerland, 1996.