Chapter 1

Software Engineering Life Cycle Processes

1. Introduction to IEEE/EIA Standard 12207.0-1996

IEEE/EIA Standard 12207.0-1996 establishes a common framework for software life cycle processes. This standard contains processes, activities, and tasks that are to be applied during the acquisition of a system that contains software, a stand-alone software product, and software service during the supply, development, operation, and maintenance phases of software products.

The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) published ISO/IEC 12207, *Information Technology -- Software Life Cycle Processes*, in August 1995. IEEE/EIA Standard 12207.0 is an American version of the international standard. IEEE/EIA Standard 12207.0 consists of the clarifications, additions, and changes accepted by the Institute of Electrical and Electronics Engineers (IEEE) and the Electronic Industries Association (EIA) during a joint project by the two organizations. IEEE/EIA Standard 12207.0 contains concepts and guidelines to foster better understanding and application of the standard. Thus, this standard provides industry a basis for software practices that is usable for both national and international businesses.

IEEE/EIA Standard 12207.0 may be used to:

- Acquire, supply, develop, operate, and maintain software.
- Support the above functions in the form of quality assurance, configuration management, joint reviews, audits, verification, validation, problem resolution, and documentation.
- Manage and improve the organization's processes and personnel.
- Establish software management and engineering environments based upon the life cycle processes as adapted and tailored to serve business needs.
- Foster improved understanding between customers and vendors and among the parties involved in the life cycle of a software product.
- Facilitate world trade in software.

IEEE/EIA Standard 12207-1996 is partitioned into three parts:

- IEEE/EIA Standard 12207.0-1996, *Standard for Information Technology -- Software Life Cycle Processes*: Contains ISO/IEC 12207 in its original form and six additional annexes (E through J): Basic concepts; Compliance; Life cycle process objectives; Life cycle data objectives; Relationships; and Errata. A unique IEEE/EIA foreword is included.
- IEEE/EIA Standard 12207.1-1997, Guide for ISO/IEC 12207, *Standard for Information Technology -- Software Lifecycle Processes -- Life Cycle Data*: Provides additional guidance on recording life cycle data.

- IEEE/EIA P12207.2-1997, Guide for ISO/IEC 12207-1997, *Standard for Information Technology -- Software Life Cycle Processes -- Implementation Considerations*: Provides additions, alternatives, and clarifications to ISO/IEC 12207's life cycle processes as derived from U.S. practices.

2. Application of IEEE/EIA Standard 12207-1996

This section lists the software life cycle processes that can be employed to acquire, supply, develop, operate, and maintain software products. IEEE/EIA Standard 12207 groups the activities that may be performed during the life cycle of software into five primary processes, eight supporting processes and four organizational processes. One of the primary processes, *development*, is emphasized in this tutorial set.

2.1 Development process

The *development process* contains the activities and tasks of the developer. The process contains the activities for requirements analysis, design, coding, integration, testing, and installation and acceptance related to software products. It may contain system related activities if stipulated in the contract. The developer performs or supports the activities in this process in accordance with the contract. The developer manages the development process at the project level following the *management process,* which is instantiated in this process. The developer also establishes an infrastructure under the process following the *infrastructure process*, tailors the process for the project, and manages the process at the organizational level following the *improvement* process and the *training* process.

This process consists of the following 12 activities:

1. Process implementation
2. System requirements analysis
3. System architectural design
4. Software requirements analysis
5. Software architectural design
6. Software detailed design
7. Software coding and testing
8. Software integration
9. Software qualification testing
10. System integration
11. System qualification testing
12. Software installation
13. Software acceptance support.

This tutorial only covers seven of the above activities: system requirements analysis and design, software requirements, software architecture and detailed design, implementation (coding), testing and the maintenance process.8 are listed below

1. *System requirements analysis.* Describes the functions and capabilities of the system; business, organizational and user requirements; safety, security, human-factors engineering (ergonomics), interface, operations, and maintenance requirements; also includes design constraints and system measurements.

2. *System architectural design.* Identifies items of hardware, software, and manual-operations. It shall be ensured that all system requirements are allocated among the items. Hardware configuration items, software configuration items, and manual operations are subsequently identified from these items.

3. *Software requirements analysis.* Establishes and documents software requirements including functional, performance, external interfaces, design constraints, and quality characteristics.

4. *Software architectural design.* Transforms the requirements for the software item into an architecture that describes its top-level structure and identifies the software components. All requirements for the software item are to be allocated to their software components and further refined to facilitate detailed design.

5. *Software detailed design.* Develops a detailed design for each software component of the software item. The software components are refined into lower levels containing software units that can be coded, compiled, and tested. All software requirements are allocated from the software components to software units.

6. *Software coding and unit testing.* Coding and unit testing of each software configuration item.

7. *Software integration and testing.* Integration of software units and software components into the software system. Perform qualification testing to ensure that the final system meets the software requirements.

8. *Software maintenance.* Modification of a software product to correct faults, to improve performance or other attributes, or to adapt the product to a changing environment.

2.2 Supporting life cycle processes

The supporting life cycle processes consist of eight processes. A supporting process supports another process as an integral part with a distinct purpose and contributes to the success and quality of the software project. A supporting process is employed and executed as needed, by another process. The supporting processes are:

1. *Documentation process.* Defines the activities for recording the information produced by a life cycle process.

2. *Configuration management process.* Defines the configuration management activities.

3. *Quality assurance process.* Defines the activities for objectively assuring that the software products and processes are in conformance with their specified requirements and adhere to their established plans. *Joint reviews, audits, verification,* and *validation* may be used as techniques of *quality assurance.*

4. *Verification process.* Defines the activities (for the acquirer, the supplier, or an independent party) for verifying the software products and services in varying depth, depending on the software project.

5. *Validation process.* Defines the activities (for the acquirer, the supplier, or an independent party) for validating the software products of the software project.

6. *Joint review process.* Defines the activities for evaluating the status and products of an activity. This process may be employed by any two parties, where one party (reviewing party) reviews another party (reviewed party) in a joint forum.

7. *Audit process.* Defines the activities for determining compliance with the requirements, plans, and contract. This process may be employed by any two parties, where one party (auditing party) audits the software products or activities of another party (audited party).

8. *Problem resolution process.* Defines a process for analyzing and removing the problems (including non-conformances), regardless of nature or source, that are discovered during the execution of development, operation, maintenance, or other processes.

2.3 Organizational life cycle processes

The organizational life cycle processes consist of four processes. They are employed by an organization to establish and implement an underlying structure made up of associated life cycle processes and personnel, and continuously improve the structure and processes. They are typically employed outside the realm of specific projects and contracts; however, lessons from such projects and contracts contribute to the improvement of the organization. The organizational processes are:

1. *Management process.* Defines the basic activities of the management, including project management related to the execution of a life cycle process.

2. *Infrastructure process.* Defines the basic activities for establishing the underlying structure of a life cycle process.

3. *Improvement process.* Defines the basic activities that an organization (that is, acquirer, supplier, developer, operator, maintainer, or the manager of another process) performs for establishing, measuring, controlling, and improving its life cycle process.

4. *Training process.* Defines the activities for providing adequately trained personnel.

3. Introduction to Tutorial

This tutorial (Volume 1 of the set) has been partitioned into chapters to support the IEEE Certificate for Software Development Professionals (CSDP) examination and as a software engineering textbook and reference guide. Chapter 1 provides an overview of the history and current state of software engineering. Chapter 2 to covers several subject areas from the CSDP exam specifications, including professionalism and software law. Chapters 3 through 5 and Chapters 7 through 8 discuss the major development processes. Chapter 6 surveys *software development strategies and methods.* Chapter 9 discusses maintenance (again in support of the CSDP exam).

Each chapter begins with an introduction to the subject and the supporting papers and standards. Each introduction incorporates the appropriate clauses from IEEE/EIA Standard 12207-1997. The appropriate IEEE Software Engineering Standards are contained within every chapter. Two additional Computer Society Press Tutorials contain appropriate standards.

- The standards for the management process can be found in a separate Computer Society Press tutorial, *Software Engineering Project Management*, 2nd edition. Be sure to obtain the latest printing from 2001. The three standards included in the *Project Management* tutorial are:
  - IEEE Standard 1058-1998, IEEE *Standard for Software Project Management Plans* (draft)
  - IEEE Standard 1062-1998, *IEEE Recommended Practice for Software Acquisition*
  - IEEE Standard 1540-2001, *IEEE Standard for Software Life Cycle Processes — Risk Management*
- The requirements standards are included in a separate Computer Society Press tutorial, *Software Requirements Engineering*, 2nd edition. The three standards included in the *Requirements* tutorial are:
  - IEEE Standard 830-1998, *Recommended Practice for Software Requirement Specifications*
  - IEEE Standard 1233-1998, *Guide to Preparing System Requirements Specifications*
  - IEEE Standard 1362-1998, *Standard for Information Technology – System Definition – Concept of Operations*

4. Articles

The first paper (letter) credits Professor Friedrich L. Bauer with coining the phrase "software engineering." In 1967, the NATO Science Committee organized a conference to discuss the problems of building large-scale software systems. The conference was to be held in Garmisch, Germany, in 1968. At a pre-conference meeting, Professor Bauer of Munich Technical University, proposed that the conference be called "Software Engineering" as a means of attracting attention to the conference. This short paper (letter), written by Professor Bauer of Munich and edited by Professor Andrew D. McGettrick of University of Strathclyde, Glasgow, Scotland, was the foreword from an earlier IEEE Tutorial, *Software Engineering -- A European Perspective* [1], explaining the history behind the term. It is therefore included in this tutorial to give full credit to Professor Bauer.

The centerpiece of this chapter is an original paper from an earlier Software Engineering tutorial [2], written by the well-known author and consultant, Roger Pressman. As indicated in the *Preface*, one of the problems of software engineering is the shortage of basic papers. Once something is described, practitioners and academics apparently move onto research or to the finer points of argument, abandoning the need to occasionally update the fundamentals. Dr. Pressman undertook the task of updating the basic papers on software engineering to the current state of practice. Pressman discusses technical and management aspects of software engineering. He surveys existing high-level models of the software development process (linear sequential,

prototyping, incremental, evolutionary and formal) and discusses management of people, the software project, and the software process. He discusses quality assurance and configuration management as being equally as important as technical and management issues. He reviews some of the principles and methods that form the foundation of the current practice of software engineering, and concludes with a prediction that three issues — reuse, re-engineering, and a new generation of tools -- will dominate software engineering for the next ten years.

Pressman's latest book is *Software Engineering: A Practitioner's Approach,* 5th edition (McGraw-Hill, 2000).

The last paper in this chapter was written by Buxton, entitled "Software Engineering -- 20 Years On and 20 Years Back." This paper provides another historical perspective of the origin and use of the term "software engineering." Professor Buxton is in a unique position to define the past history of software engineering, as he was one of the main reporters and documenters of the second NATO software engineering conference in Rome in 1969.

---

1. Thayer, R. H., and A. D. McGettrick, (eds.), *Software Engineering —- A European Perspective,* IEEE Computer Society Press, Los Alamitos, CA, 1993.

2. Dorfman, M., and R.H. Thayer (eds.), *Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, 1997.

The Origin of Software Engineering

Dear Dr. Richard Thayer:

In reply to your question about the origin of the term "Software Engineering," I submit the following story.

In the mid-1960s, there was increasing concern in scientific quarters of the Western world that the tempestuous development of computer hardware was not matched by appropriate progress in software. The software situation looked more to be turbulent. Operating systems had just been the latest rage, but they showed unexpected weaknesses. The uneasiness had been lined out in the NATO Science Committee by its US representative, Dr. I.I. Rabi, the Nobel laureate and famous, as well as influential, physicist. In 1967, the Science Committee set up the Study Group on Computer Science, with members from several countries, to analyze the situation. The German authorities nominated me for this team. The study group was given the task of "assessing the entire field of computer science," with particular elaboration on the Science Committee's consideration of "organizing a conference and, perhaps, at a later date,... setting up ... an International Institute of Computer Science."

The study group, concentrating its deliberations on actions that would merit an international rather than a national effort, discussed all sorts of promising scientific projects. However, it was rather inconclusive on the relation of these themes to the critical observations mentioned above, which had guided the Science Committee. Perhaps not all members of the study group had been properly informed about the rationale of its existence. In a sudden mood of anger, I made the remark, "The whole trouble comes from the fact that there is so much tinkering with software. It is not made in a clean fabrication process," and when I found out that this remark was shocking to some of my scientific colleagues, I elaborated the idea with the provocative saying, "What we need is software *engineering."*

This remark had the effect that the expression "software engineering," which seemed to some to be a contradiction in terms, stuck in the minds of the members of the group. In the end, the study group recommended in late 1967 the holding of a Working Conference on Software Engineering, and I was made chairman. I had not only the task of organizing the meeting (which was held from October 7 to October 10, 1968, in Garmisch, Germany), but I had to set up a scientific program for a subject that was suddenly defined by my provocative remark. I enjoyed the help of my cochairmen, L. Bolliet from France, and HJ. Helms from Denmark, and in particular the invaluable support of the program committee members, AJ. Perlis and B. Randall in the section on design, P. Naur and J.N. Buxton in the section on production, and K. Samuelson, B. Galler, and D. Gries in the section on service. Among the 50 or so participants, E.W. Dijkstra was dominant. He actually made not only cynical remarks like "the dissemination of error-loaded software is frightening" and "it is not clear that the people who

manufacture software are to be blamed. I think manufacturers deserve better, more understanding users." He also said already at this early date, "Whether the, correctness of a piece of software can be guaranteed or not depends greatly on the structure of the thing made," and he had very fittingly named his paper "Complexity Controlled by Hierarchical Ordering of Function and Variability," introducing a theme that followed his life the next 20 years. Some of his words have become proverbs in computing, like "testing is a very inefficient way of convincing oneself of the correctness of a program."

With the wide distribution of the reports on the Garmisch conference and on a follow-up conference in Rome, from October 27 to 31,1969, it emerged that not only the phrase *software engineering,* but also the idea behind this, became fashionable. Chairs were created, institutes were established (although the one which the NATO Science Committee had proposed did not come about because of reluctance on the part of Great Britain to have it organized' on the European continent), and a great number of conferences were held. The present volume shows clearly how much progress has been made in the intervening years.

The editors deserve particular thanks for paying so much attention to a tutorial. In choosing the material, they have tried to highlight a number of software engineering initiatives whose origin is European. In particular, the more formal approach to software engineering is evident, and they leave included some material that is not readily available, elsewhere. The tutorial nature of the papers is intended to offer readers an easy introduction to the topics and indeed to the attempts that have been made in recent years to provide them with the *tools,* both in a handcraft and intellectual sense, that allow them now *to* call themselves honestly software *engineers.*

O. Prof. Dr. Friedrich L. Bauer,
Professor Emeritus
Institut für Informatik der Technischen Universität München,
Postfach 20 24 20,
D-8000 München 2, Germany

Software Engineering

Roger S. Pressman, Ph.D.

As software engineering approaches its fourth decade, it suffers from many of the strengths and some of the frailties that are experienced by humans of the same age. The innocence and enthusiasm of its early years have been replaced by more reasonable expectations (and even a healthy cynicism) fostered by years of experience. Software engineering approaches its mid-life with many accomplishments already achieved, but with significant work yet to do.

The intent of this paper is to provide a survey of the current state of software engineering and to suggest the likely course of the aging process. Key software engineering activities are identified, issues are presented, and future directions are considered. There will be no attempt to present an in-depth discussion of specific software engineering topics. That is the job of other papers presented in this book.

1.0 Software Engineering—Layered Technology[1]

Although hundreds of authors have developed personal definitions of software engineering, a definition proposed by Fritz Bauer [1] at the seminal conference on the subject still serves as a basis for discussion:

> [Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

Almost every reader will be tempted to add to this definition. It says little about the technical aspects of software quality; it does not directly address the need for customer satisfaction or timely product delivery; it omits mention of the importance of measurement and metrics; it does not state the importance of a mature process. And yet, Bauer's definition provides us with a baseline. What are the "sound engineering principles" that can be applied to computer software development? How to "economically" build software so that it is "reliable"? What is required to create computer programs that work "efficiently" on not one but many different "real machines"? These are the questions that continue to challenge software engineers.

Software engineering is a layered technology. Referring to Figure 1, any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total quality management and similar philosophies foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more mature approaches to software engineering. The bedrock that supports software engineering is a quality focus.

The foundation for software engineering is the process layer. Software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework for a set of *key process areas* [2] that must be established for effective delivery of software engineering technology. The key process areas form the basis for management control of software projects, and establish the context in which technical methods are applied, deliverables (models, documents, data reports, forms, and so on) are produced, milestones are established, quality is ensured, and change is properly managed.

Software engineering methods provide the technical "how to's" for building software. Methods encompass a broad array of tasks that include: requirements analysis, design, program construction, testing, and maintenance. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities, and other descriptive techniques.

Software engineering tools provide automated or semiautomated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering (CASE), is established. CASE combines software, hardware, and a software engineering database (a repository containing important information about analysis, design, program construction, and testing) to create a software engineering environment that is analogous to CAD/CAE (computer-aided design/engineering) for hardware.

1 Portions of this paper have been adapted from *A Manager's Guide to Software Engineering* [19] and *Software Engineering: A Practitioner's Approach* (McGraw-Hill, fourth edition, 1997) and are used with permission.

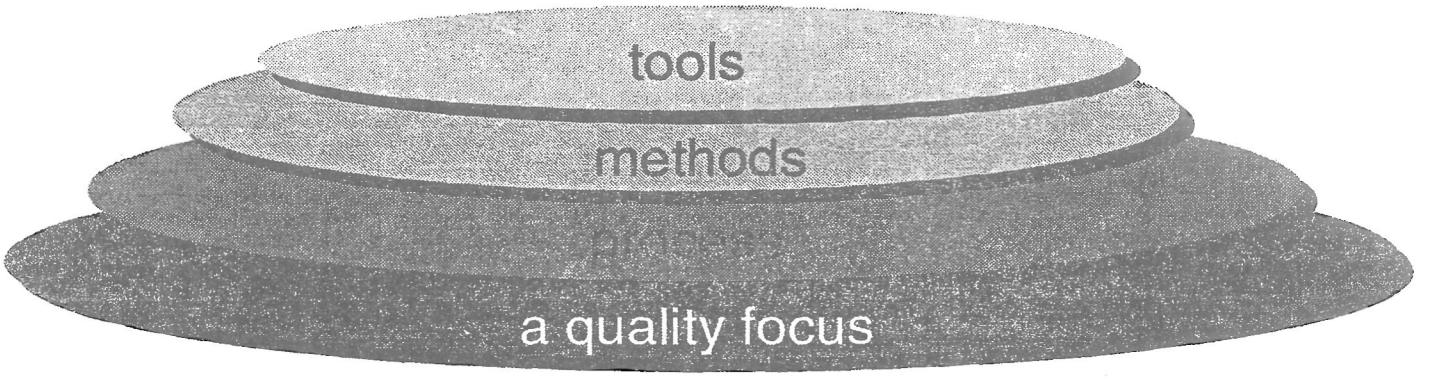

Figure 1. Software engineering layers

2.0 Software Engineering Process Models

Software engineering incorporates a development strategy that encompasses the process, methods, and tools layers described above. This strategy is often referred to as a *process model* or a *software engineering paradigm*. A process model for software engineering is chosen based on the nature of the project and application, the methods and tools to be used, and the controls and deliverables that are required. Four classes of process models have been widely discussed (and debated). A brief overview of each is presented in the sections that follow.

2.1 Linear, Sequential Models

Figure 2 illustrates the *linear sequential* model for software engineering. Sometimes called the "classic life cycle" or the "waterfall model," the linear sequential model demands a systematic, sequential approach to software development that begins at the system level and progresses through analysis, design, coding, testing, and maintenance. The linear sequential model is the oldest and the most widely used paradigm for software engineering. However, criticism of the paradigm has caused even active supporters to question its efficacy. Among the problems that are sometimes encountered when the linear sequential model is applied are:

1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.
2. It is often difficult for the customer to state all requirements explicitly. The linear sequential model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.
3. The customer must have patience. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

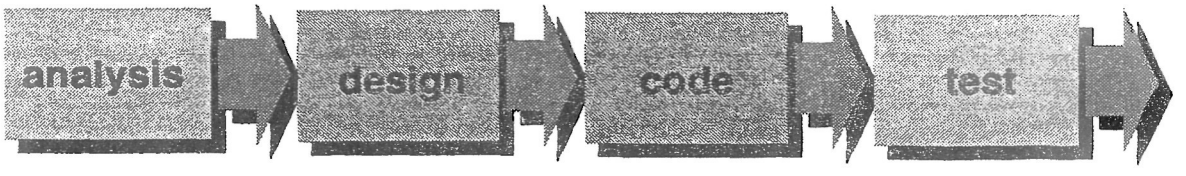

Figure 2. The linear, sequential paradigm

2.2 Prototyping

Often, a customer defines a set of general objectives for software, but does not identify detailed input, processing, or output requirements. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human-machine interaction should take. In these, and many other situations, a prototyping paradigm may offer the best approach.

The prototyping paradigm (Figure 3) begins with requirements gathering. Developer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A "quick design" then occurs. The quick design focuses on a representation of those aspects of the software that will be visible to the customer/user (for example, input approaches and output formats). The quick design leads to the construction of a prototype. The prototype is evaluated by the customer/user and is used to refine requirements for the software to be developed. Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.

Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to make use of existing program fragments or applies tools (report generators, and window managers, for instance) that enable working programs to be generated quickly.

Both customers and developers like the prototyping paradigm. Users get a feel for the actual system and developers get to build something immediately. Yet, prototyping can also be problematic for the following reasons:

1. The customer sees what appears to be a working version of the software, unaware that the prototype is held together "with chewing gum and baling wire" or that in the rush to get it working we haven't considered overall software quality or long-term maintainability. When informed that the product must be rebuilt, the customer cries foul and demands that "a few fixes" be applied to make the prototype a working product. Too often, software development management relents.
2. The developer often makes implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, the developer may become familiar with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

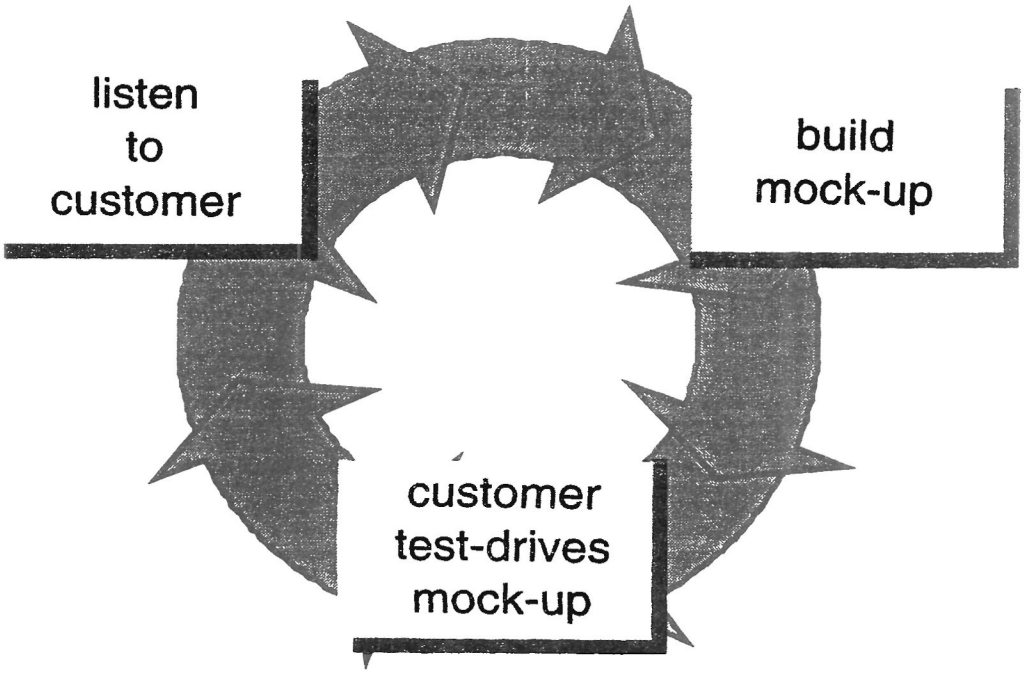

Figure 3. The prototyping paradigm

2.3 Incremental Model

Although problems can occur, prototyping is an effective paradigm for software engineering. The key is to define the rules of the game at the beginning; that is, the customer and developer must both agree that the prototype is built to serve as a mechanism for defining requirements. It is then discarded (at least in part) and the actual software is engineered with an eye toward quality and maintainability.

When an incremental model is used, the first increment is often a *core product*. That is, basic requirements are addressed, but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed review). As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.

The incremental process model, like prototyping (Section 2.2) and evolutionary approaches (Section 2.4), is iterative in nature. However, the incremental model focuses on the delivery of an operational product with each increment. Early increments are "stripped down" versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.

Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment. In addition, increments can be planned to manage technical risks. For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain. It might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end users without inordinate delay.

2.4 Evolutionary Models

The *evolutionary* paradigm, also called the *spiral model* [3] couples the iterative nature of prototyping with the controlled and systematic aspects of the linear model. Using the evolutionary paradigm, software is developed in a series of incremental releases. During early iterations, the incremental release might be a prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

Figure 4 depicts a typical evolutionary model. Each pass around the spiral moves through six task regions:

- **customer communication**—tasks required to establish effective communication between developer and customer
- **planning**—tasks required to define resources, time lines and other project-related information
- **risk assessment**—tasks required to assess both technical and management risks
- **engineering**—tasks required to build one or more representations of the application
- **construction and release**—tasks required to construct, test, install, and provide user support (for example, documentation and training)
- **customer evaluation**—tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering stage and implemented during the installation stage.

Each region is populated by a series of tasks adapted to the characteristics of the project to be undertaken.

The spiral model is a realistic approach to the development of large scale systems and software. It uses an "evolutionary" approach [4] to software engineering, enabling the developer and customer to understand and react to risks at each evolutionary level. It uses prototyping as a risk reduction mechanism, but more importantly, it enables the developer to apply the prototyping approach at any stage in the evolution of the product. It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world. The spiral model demands a direct consideration of technical risks at all stages of the project, and if properly applied, should reduce risks before they become problematic.

But like other paradigms, the spiral model is not a panacea. It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable. It demands considerable risk assessment expertise, and relies on this expertise for success. If a major risk is not discovered, problems will undoubtedly occur. Finally, the model itself is relatively new and has not been used as widely as the linear sequential or prototyping paradigms. It will take a number of years before efficacy of this important new paradigm can be determined with absolute certainty.

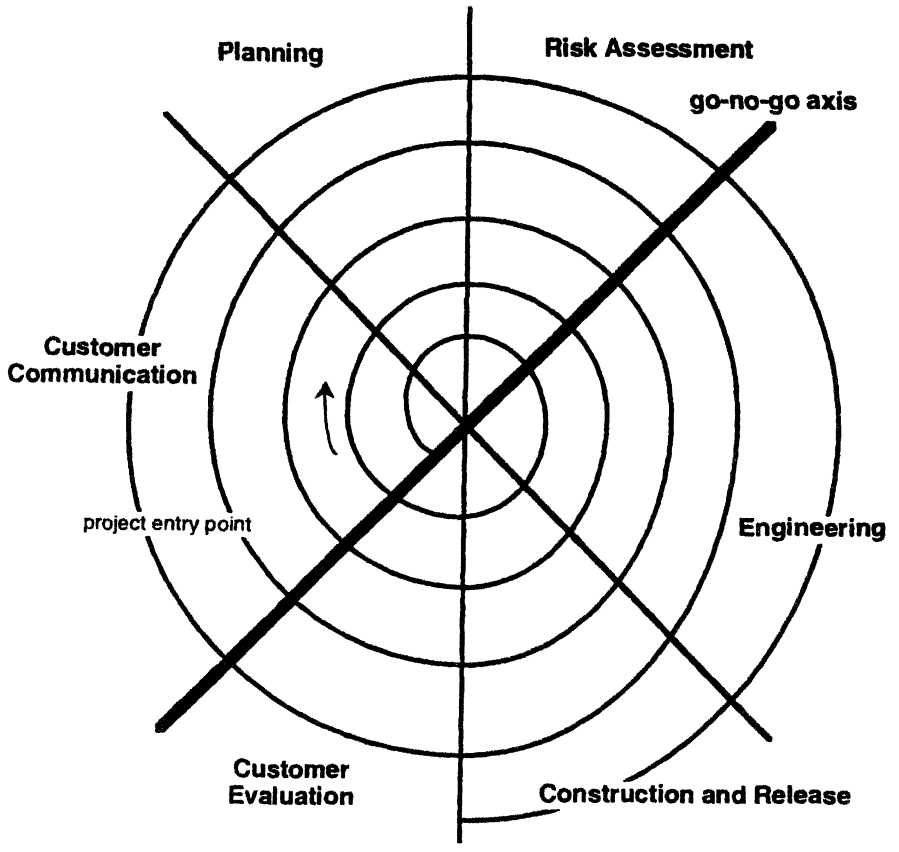

Figure 4. The evolutionary model

2.5 The Formal Methods Model

The formal methods paradigm encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable a software engineer to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation on this approach, called cleanroom software engineering [5, 6], is currently applied by a limited number of companies.

When formal methods are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms. Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily—not through ad hoc review, but through the application of mathematical analysis. When formal methods are used during design, they serve as a basis for program verification and therefore enable the software engineer to discover and correct errors that might otherwise go undetected.

Although not yet a mainstream approach, the formal methods model offers the promise of defect-free software. Yet, concern about its applicability in a business environment has been voiced:

1. The development of formal models is currently quite time-consuming and expensive.
2. Because few software developers have the necessary background to apply formal methods, extensive training is required.
3. It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

These concerns notwithstanding, it is likely that the formal methods approach will gain adherents among software developers who must build safety-critical software (such as aircraft avionics and medical devices) and among developers that would suffer severe economic hardship should software errors occur.

3.0 The Management Spectrum

Effective software project management focuses on the three P's: *people, problem, and process*. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavor will never have success in project management. A manager

who fails to encourage comprehensive customer communication early in the evolution of a project risks building an elegant solution for the wrong problem. Finally, the manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum.

3.1 People

The cultivation of motivated, highly skilled software people has been discussed since the 1960s [see 7, 8, 9]. The Software Engineering Institute has sponsored a *people-management maturity model* "to enhance the readiness of software organizations to undertake increasingly complex applications by helping to attract, grow, motivate, deploy, and retain the talent needed to improve their software development capability." [10]

The people-management maturity model defines the following key practice areas for software people: recruiting, selection, performance management, training, compensation, career development, organization, and team and culture development. Organizations that achieve high levels of maturity in the people-management area have a higher likelihood of implementing effective software engineering practices.

3.2 The Problem

Before a project can be planned, objectives and scope should be established, alternative solutions should be considered, and technical and management constraints should be identified. Without this information, it is impossible to develop reasonable cost estimates, a realistic breakdown of project tasks, or a manageable project schedule that is a meaningful indicator of progress.

The software developer and customer must meet to define project objectives and scope. In many cases, this activity occurs as part of structured customer communication process such as *joint application design* [11, 12]. Joint application design (JAD) is an activity that occurs in five phases: project definition, research, preparation, the JAD meeting, and document preparation. The intent of each phase is to develop information that helps better define the problem to be solved or the product to be built.

3.3 The Process

A software process (see discussion of process models in Section 2.0) can be characterized as shown in Figure 5. A few framework activities apply to all software projects, regardless of their size or complexity. A number of *task sets*—tasks, milestones, deliverables, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities—such as software quality assurance, software configuration management, and measurement—overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

In recent years, there has been a significant emphasis on process "maturity." [2] The Software Engineering Institute (SEI) has developed a comprehensive assessment model predicated on a set of software engineering capabilities that should be present as organizations reach different levels of process maturity. To determine an organization's current state of process maturity, the SEI uses an assessment questionnaire and a five-point grading scheme. The grading scheme determines compliance with a capability maturity model [2] that defines key activities required at different levels of process maturity. The SEI approach provides a measure of the global effectiveness of a company's software engineering practices and establishes five process maturity levels that are defined in the following manner:

Level 1: Initial—The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort.

Level 2: Repeatable—Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

Level 3: Defined—The software process for both management and engineering activities is documented, standardized, and integrated into an organization-wide software process. All projects use a documented and approved version of the organization's process for developing and maintaining software. This level includes all characteristics defined for level 2.

Level 4: Managed—Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled using detailed measures. This level includes all characteristics defined for level 3.

Level 5: Optimizing—Continuous process improvement is enabled by quantitative feedback from the process and from testing innovative ideas and technologies. This level includes all characteristics defined for level 4.

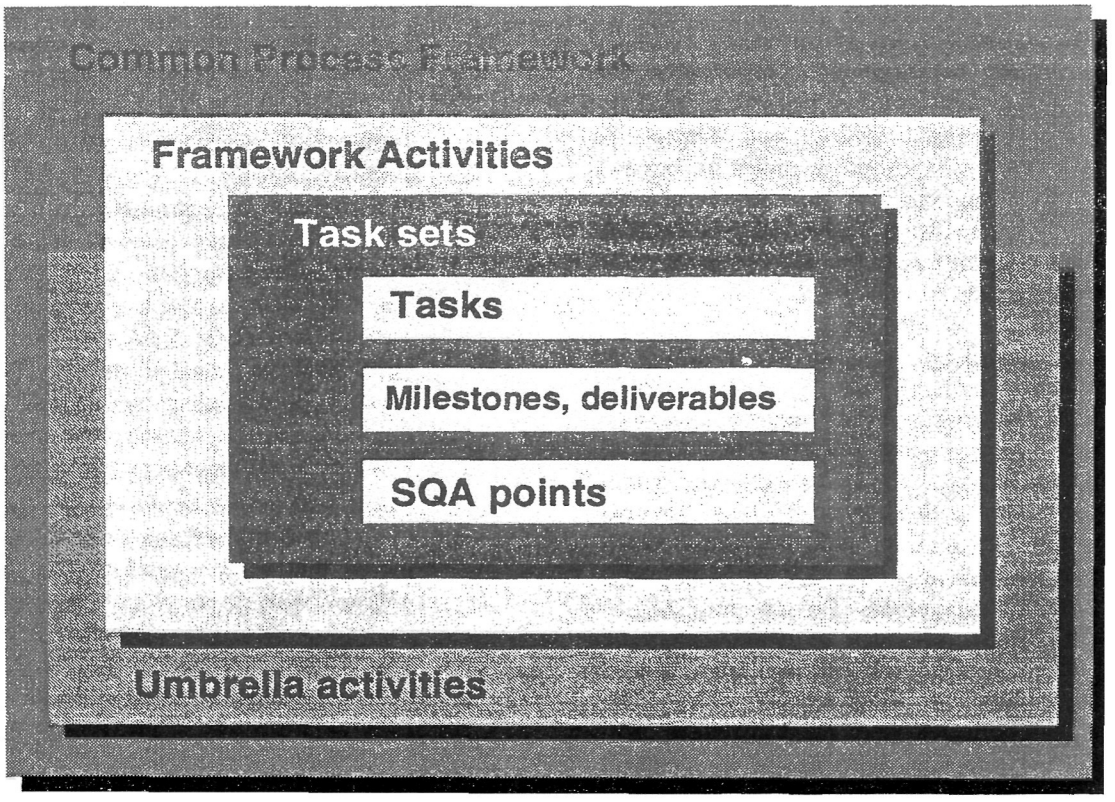

Figure 5. A common process framework

The five levels defined by the SEI are derived as a consequence of evaluating responses to the SEI assessment questionnaire that is based on the CMM. The results of the questionnaire are distilled to a single numerical grade that helps indicate an organization's process maturity.

The SEI has associated key process areas (KPAs) with each maturity level. The KPAs describe those software engineering functions (for example, software project planning and requirements management) that must be present to satisfy good practice at a particular level. Each KPA is described by identifying the following characteristics:

- *goals*—the overall objectives that the KPA must achieve
- *commitments*—requirements (imposed on the organization) that must be met to achieve the goals, provide proof of intent to comply with the goals
- *abilities*—those things that must be in place (organizationally and technically) that will enable the organization to meet the commitments
- *activities*—the specific tasks that are required to achieve the KPA function
- *methods for monitoring implementation*—the manner in which the activities are monitored as they are put into place
- *methods for verifying implementation*—the manner in which proper practice for the KPA can be verified.

Eighteen KPAs (each defined using the structure noted above) are defined across the maturity model and are mapped into different levels of process maturity.

Each KPA is defined by a set of *key practices* that contribute to satisfying its goals. The key practices are policies, procedures, and activities that must occur before a key process area has been fully instituted. The SEI defines *key indicators* as "those key practices or components of key practices that offer the greatest insight into whether the goals of a key process area have been achieved." Assessment questions are designed to probe for the existence (or lack thereof) of a key indicator.

4.0 Software Project Management

Software project management encompasses the following activities: measurement, project estimating, risk analysis, scheduling, tracking, and control. A comprehensive discussion of these topics is beyond the scope of this paper, but a brief overview of each topic will enable the reader to understand the breadth of management activities required for a mature software engineering organizations.

4.1 Measurement and Metrics

To be most effective, software metrics should be collected for both the process and the product. Process-oriented metrics [14, 15] can be collected during the process and after it has been completed. Process metrics collected during the project focus on the efficacy of quality assurance activities, change management, and project management. Process metrics collected after a project has been completed examine quality and productivity. Process measures are normalized using either lines of code or function points [13], so that data collected from many different projects can be compared and analyzed in a consistent manner. Product metrics measure technical characteristics of the software that provide an indication of software quality [15, 16, 17, 18]. Measures can be applied to models created during analysis and design activities, the source code, and testing data. The mechanics of measurement and the specific measures to be collected are beyond the scope of this paper.

4.2 Project Estimating

Scheduling and budgets are often dictated by business issues. The role of estimating within the software process often serves as a "sanity check" on the predefined deadlines and budgets that have been established by management. (Ideally, the software engineering organization should be intimately involved in establishing deadlines and budgets, but this is not a perfect or fair world.)

All software project estimation techniques require that the project have a bounded scope, and all rely on a high level functional decomposition of the project and an assessment of project difficulty and complexity. There are three broad classes of estimation techniques [19] for software projects:

- **Effort estimation techniques.** The project manager creates a matrix in which the left-hand column contains a list of major system functions derived using functional decomposition applied to project scope. The top row contains a list of major software engineering tasks derived from the common process framework. The manager (with the assistance of technical staff) estimates the effort required to accomplish each task for each function.
- **Size-Oriented Estimation.** A list of major system functions is derived using functional decomposition applied to project scope. The "size" of each function is estimated using either lines of code (LOC) or function points (FP). Average productivity data (for instance, function points per person month) for similar functions or projects are used to generate an estimate of effort required for each function.
- **Empirical Models.** Using the results of a large population of past projects, an empirical model that relates product size (in LOC or FP) to effort is developed using a statistical technique such as regression analysis. The product size for the work to be done is estimated and the empirical model is used to generate projected effort (for example, [20]).

In addition to the above techniques, a software project manager can develop estimates by analogy. This is done by examining similar past projects then projecting effort and duration recorded for these projects to the current situation.

4.3 Risk Analysis

Almost five centuries have passed since Machiavelli said: "I think it may be true that fortune is the ruler of half our actions, but that she allows the other half to be governed by us... [fortune] is like an impetuous river... but men can make provision against it by dykes and banks." Fortune (we call it risk) is in the back of every software project manager's mind, and that is often where it stays. And as a result, risk is never adequately addressed. When bad things happen, the manager and the project team are unprepared.

In order to "make provision against it," a software project team must conduct risk analysis explicitly. Risk analysis [21, 22, 23] is actually a series of steps that enable the software team to perform risk identification, risk assessment, risk prioritization, and risk management. The goals of these activities are: (1) to identify those risks that have a high likelihood of occurrence, (2) to assess the consequence (impact) of each risk should it occur, and (3) to develop a plan for mitigating the risks when possible, monitoring factors that may indicate their arrival, and developing a set of contingency plans should they occur.

4.4 Scheduling

The process definition and project management activities that have been discussed above feed the scheduling activity. The common process framework provides a work breakdown structure for scheduling. Available human resources, coupled with effort estimates and risk analysis, provide the task interdependencies, parallelism, and time lines that are used in constructing a project schedule.

4.5 Tracking and Control

Project tracking and control is most effective when it becomes an integral part of software engineering work. A well-defined process framework should provide a set of milestones that can be used for project tracking. Control focuses on two major issues: quality and change.

To control quality, a software project team must establish effective techniques for software quality assurance, and to control change, the team should establish a software configuration management framework.

5.0 Software Quality Assurance

In his landmark book on quality, Philip Crosby [24] states:

> The problem of quality management is not what people don't know about it. The problem is what they think they do know...
>
> In this regard, quality has much in common with sex. Everybody is for it. (Under certain conditions, of course.) Everyone feels they understand it. (Even though they wouldn't want to explain it.) Everyone thinks execution is only a matter of following natural inclinations. (After all, we do get along somehow.) And, of course, most people feel that problems in these areas are caused by other people. (If only they would take the time to do things right.)

There have been many definitions of software quality proposed in the literature. For our purposes, software quality is defined as: *Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.*

There is little question that the above definition could be modified or extended. In fact, the precise definition of software quality could be debated endlessly. But the definition stated above does serve to emphasize three important points:

1. Software requirements are the foundation from which quality is assessed. Lack of conformance to requirements is lack of quality.
2. A mature software process model defines a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.
3. There is a set of implicit requirements that often go unmentioned (for example, the desire for good maintainability). If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

Almost two decades ago, McCall and Cavano [25, 26] defined a set of quality factors that were a first step toward the development of metrics for software quality. These factors assessed software from three distinct points of view: (1) product operation (using it); (2) product revision (changing it), and (3) product transition (modifying it to work in a different environment, that is., "porting" it). These factors include:

- *Correctness*. The extent to which a program satisfies its specification and fulfills the customer's mission objectives.
- *Reliability*. The extent to which a program can be expected to perform its intended function with required precision.
- *Efficiency*. The amount of computing resources and code required by a program to perform its function.
- *Integrity*. Extent to which access to software or data by unauthorized persons can be controlled.
- *Usability*. Effort require to learn, operate, prepare input, and interpret output of a program.
- *Maintainability*. Effort require to locate and fix an error in a program. [Might be better termed "correctability"].
- *Flexibility*. Effort required to modify an operational program.
- *Testability*. Effort required to test a program to insure that it performs its intended function.
- *Portability*. Effort required to transfer the program from one hardware and/or software system environment to another.
- *Reusability*. Extent to which a program [or parts of a program] can be reused in other

applications—related to the packaging and scope of the functions that the program performs.

- *Interoperability.* Effort required to couple one system to another.

The intriguing thing about these factors is how little they have changed in almost 20 years. Computing technology and program architectures have undergone a sea change, but the characteristics that define high-quality software appear to be invariant. The implication: An organization that adopts factors such as those described above will build software today that will exhibit high quality well into the first few decades of the twenty-first century. More importantly, this will occur regardless of the massive changes in computing technologies that are sure to come over that period of time.

Software quality is designed into a product or system. It is not imposed after the fact. For this reason, *software quality assurance* (SQA) actually begins with the set of technical methods and tools that help the analyst to achieve a high-quality specification and the designer to develop a high-quality design.

Once a specification (or prototype) and design have been created, each must be assessed for quality. The central activity that accomplishes quality assessment is the formal technical review. The *formal technical review* (FTR)—conducted as a *walk-through* or an *inspection* [27]—is a stylized meeting conducted by technical staff with the sole purpose of uncovering quality problems. In many situations, formal technical reviews have been found to be as effective as testing in uncovering defects in software [28].

Software testing combines a multistep strategy with a series of test case design methods that help ensure effective error detection. Many software developers use software testing as a quality assurance "safety net." That is, developers assume that thorough testing will uncover most errors, thereby mitigating the need for other SQA activities. Unfortunately, testing, even when performed well, is not as effective as we might like for all classes of errors. A much better strategy is to find and correct errors (using FTRs) before getting to testing.

The degree to which formal *standards and procedures* are applied to the software engineering process varies from company to company. In many cases, standards are dictated by customers or regulatory mandate. In other situations standards are self-imposed. An assessment of compliance to standards may be conducted by software developers as part of a formal technical review, or in situations where independent verification of compliance is required, the SQA group may conduct its own audit.

A major threat to software quality comes from a seemingly benign source: changes. Every change to software has the potential for introducing error or creating side effects that propagate errors. The change control process contributes directly to software quality by formalizing requests for change, evaluating the nature of change, and controlling the impact of change. Change control is applied during software development and later, during the software maintenance phase.

Measurement is an activity that is integral to any engineering discipline. An important objective of SQA is to track software quality and assess the impact of methodological and procedural changes on improved software quality. To accomplish this, *software metrics* must be collected.

Record keeping and recording for software quality assurance provide procedures for the collection and dissemination of SQA information. The results of reviews, audits, change control, testing, and other SQA activities must become part of the historical record for a project and should be disseminated to development staff on a need-to-know basis. For example, the results of each formal technical review for a procedural design are recorded and can be placed in a "folder" that contains all technical and SQA information about a module.

6.0 Software Configuration Management

Change is inevitable when computer software is built. And change increases the level of confusion among software engineers who are working on a project. Confusion arises when changes are not analyzed before they are made, recorded before they are implemented, reported to those who should be aware that they have occurred, or controlled in a manner that will improve quality and reduce error. Babich [29] discusses this when he states:

> The art of coordinating software development to minimize... confusion is called *configuration management.* Configuration management is the art of identifying, organizing, and controlling modifications to the software being built by a programming team. The goal is to maximize productivity by minimizing mistakes.

Software configuration management (SCM) is an umbrella activity that is applied throughout the software engineering process. Because change can occur at any time, SCM activities are developed to (1) identify change, (2) control change, (3) ensure that change is being properly implemented and (4) report change to others who may have an interest.

A primary goal of software engineering is to improve the ease with which changes can be accommodated and reduce the amount of effort expended when changes must be made.

7.0 The Technical Spectrum

There was a time—some people still call it "the good old days"—when a skilled programmer created a program like an artist creates a painting: she just sat down and started. Pressman and Herron [30] draw other parallels when they write:

> At one time or another, almost everyone laments the passing of the good old days. We miss the simplicity, the personal touch, the emphasis on quality that were the trademarks of a craft. Carpenters reminisce about the days when houses were built with mahogany and oak, and beams were set without nails. Engineers still talk about an earlier era when one person did all the design (and did it right) and then went down to the shop floor and built the thing. In those days, people did good work and stood behind it.
>
> How far back do we have to travel to reach the good old days? Both carpentry and engineering have a history that is well over 2,000 years old. The disciplined way in which work is conducted, the standards that guide each task, the step by step approach that is applied, have all evolved through centuries of experience. *Software engineering* has a much shorter history.

During its short history, the creation of computer programs has evolved from an art form, to a craft, to an engineering discipline. As the evolution took place, the free-form style of the artist was replaced by the disciplined methods of an engineer. To be honest, we lose something when a transition like this is made. There's a certain freedom in art that can't be replicated in engineering. But we gain much, much more than we lose.

As the journey from art to engineering occurred, basic principles that guided our approach to software problem analysis, design and testing slowly evolved. And at the same time, methods were developed that embodied these principles and made software engineering tasks more systematic. Some of these "hot, new" methods flashed to the surface for a few years, only to disappear into oblivion, but others have stood the test of time to become part of the technology of software development.

In this section we discuss the basic principles that support the software engineering methods and provide an overview of some of the methods that have already "stood the test of time" and others that are likely to do so.

7.1 Software Engineering Methods—The Landscape

All engineering disciplines encompass four major activities: (1) the definition of the problem to be solved, (2) the design of a solution that will meet the customer's needs; (2) the construction of the solution, and (4) the testing of the implemented solution to uncover latent errors and provide an indication that customer requirements have been achieved. Software engineering offers a variety of different methods to achieve these activities. In fact, the methods landscape can be partitioned into three different regions:

- conventional software engineering methods
- object-oriented approaches
- formal methods

Each of these regions is populated by a variety of methods that have spawned their own culture, not to mention a sometimes confusing array of notation and heuristics. Luckily, all of the regions are unified by a set of overriding principles that lead to a single objective: to create high quality computer software.

Conventional software engineering methods view software as an information transform and approach each problem using an input-process-output viewpoint. Object-oriented approaches consider each problem as a set of classes and work to create a solution by implementing a set of communicating objects that are instantiated from these classes. Formal methods describe the problem in mathematical terms, enabling rigorous evaluation of completeness, consistency, and correctness.

Like competing geographical regions on the world map, the regions of the software engineering methods map do not always exist peacefully. Some inhabitants of a particular region cannot resist religious warfare. Like most religious warriors, they become consumed by dogma and often do more harm that good. The regions of the software engineering methods landscape can and should coexist peacefully, and tedious debates over which method is best seem to miss the point. Any method, if properly applied within the context of a solid set of software engineering principles, will lead to higher quality software than an undisciplined approach.

7.2 Problem Definition

A problem cannot be fully defined and bounded until it is communicated. For this reason, the first step in any software engineering project is customer communication. Techniques for customer communication [11, 12] were discussed earlier in this paper. In essence, the developer and the customer must develop an effective mechanism for defining and negotiating the basic requirements for the software project. Once this has been accomplished, requirements analysis begins. Two options are available at this stage: (1) the creation of a prototype that will assist the developer and the customer in better understanding the system to be build, and/or (2) the creation of a detailed set of analysis models that describe the data, function, and behavior for the system.

7.2.1 Analysis Principles

Today, analysis modeling can be accomplished by applying one of several different methods that populate the three regions of the software engineering methods landscape. All methods, however, conform to a set of analysis principles [31]:

1. **The data domain of the problem must be modeled.** To accomplish this, the analyst must define the data objects (entities) that are visible to the user of the software and the relationships that exist between the data objects. The content of each data object (the object's attributes) must also be defined.
2. **The functional domain of the problem must be modeled.** Software functions transform the data objects of the system and can be modeled as a hierarchy (conventional methods), as services to classes within a system (the object-oriented view), or as a succinct set of mathematical expressions (the formal view).
3. **The behavior of the system must be represented.** All computer-based systems respond to external events and change their state of operation as a consequence. Behavioral modeling indicates the externally observable states of operation of a system and how transition occurs between these states.
4. **Models of data, function, and behavior must be partitioned.** All engineering problem-solving is a process of elaboration. The problem (and the models described above) are first represented at a high level of abstraction. As problem definition progresses, detail is refined and the level of abstraction is reduced. This activity is called partitioning.
5. **The overriding trend in analysis is from essence toward implementation.** As the process of elaboration progresses, the statement of the problem moves from a representation of the essence of the solution toward implementation-specific detail. This progression leads us from analysis toward design.

7.2.2 Analysis Methods

A discussion of the notation and heuristics of even the most popular analysis methods is beyond the scope of this paper. The problem is further compounded by the three different regions of the methods landscape and the local issues specific to each. Therefore, all that we can hope to accomplish in this section is to note similarities among the different methods and regions:

- All analysis methods provide a notation for describing data objects and the relationships that exist between them.
- All analysis methods couple function and data and provide a way for understanding how function operates on data.
- All analysis methods enable an analyst to represent behavior at a system level, and in some cases, at a more localized level.
- All analysis methods support a partitioning approach that leads to increasingly more detailed (and implementation-specific models).
- All analysis methods establish a foundation from which design begins, and some provide representations that can be directly mapped into design.

For further information on analysis methods in each of the three regions noted above, the reader should review work by Yourdon [32], Booch [33], and Spivey [34].

7.3 Design

M.A. Jackson [35] once said: "The beginning of wisdom for a computer programmer [software engineer] is to recognize the difference between getting a program to work, and getting it *right*." Software design is a set of basic principles and a pyramid of modeling methods that provide the necessary framework for "getting it right."

7.3.1 Design Principles

Like analysis modeling, software design has

spawned a collection of methods that populate the conventional, object-oriented, and formal regions that were discussed earlier. Each method espouses its own notation and heuristics for accomplishing design, but all rely on a set of fundamental principles [31] that are outlined in the paragraphs that follow:

1. **Data and the algorithms that manipulate data should be created as a set of interrelated abstractions.** By creating data and procedural abstractions, the designer models software components that have characteristics leading to high quality. An abstraction is self-contained; it generally implements one well-constrained data structure or algorithm; it can be accessed using a simple interface; the details of its internal operation need not be known for it to be used effectively; it is inherently reusable.
2. **The internal design detail of data structures and algorithms should be hidden from other software components that make use of the data structures and algorithms.** Information hiding [36] suggests that modules be "characterized by design decisions that (each) hides from all others." Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information that is necessary to achieve software function. The use of information hiding as a design criterion for modular systems provides greatest benefits when modifications are required during testing and later, during software maintenance. Because most data and procedures are hidden from other parts of the software, inadvertent errors (and resultant side effects) introduced during modification are less likely to propagate to other locations within the software.
3. **Modules should exhibit independence.** That is, they should be loosely coupled to each other and to the external environment and should exhibit functional cohesion. Software with *effective modularity*, that is, independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified (consider ramifications when development is conducted by a team). Independent modules are easier to maintain (and test) because secondary effects caused by design/code modification are limited; error propagation is reduced; and reusable modules are possible.
4. **Algorithms should be designed using a constrained set of logical constructs.** This design approach, widely know as *structured programming* [37], was proposed to limit the procedural design of software to a small number of predictable operations. The use of the structured programming constructs (sequence, conditional, and loops) reduces program complexity and thereby enhances readability, testability, and maintainability. The use of a limited number of logical constructs also contributes to a human understanding process that psychologists call *chunking*. To understand this process, consider the way in which you are reading this page. You do not read individual letters; but rather, recognize patterns or chunks of letters that form words or phrases. The structured constructs are logical chunks that allow a reader to recognize procedural elements of a module, rather than reading the design or code line by line. Understanding is enhanced when readily recognizable logical forms are encountered.

7.3.2 The Design Pyramid

Like analysis, a discussion of even the most popular design methods is beyond the scope of this paper. Our discussion here will focus on a set of design activities that should occur regardless of the method that is used.

Software design should be accomplished by following a set of design activities as illustrated in Figure 6. *Data design* translates the data model created during analysis into data structures that meet the needs of the problem. *Architectural design* differs in intent depending upon the designer's viewpoint. Conventional design creates hierarchical software architectures, while object-oriented design views architecture as the message network that enables objects to communicate. *Interface design* creates implementation models for the human-computer interface, the external system interfaces that enable different applications to interoperate, and the internal interfaces that enable program data to be communicated among software components. Finally, *procedural design* is conducted as algorithms are created to implement the processing requirements of program components.

Like the pyramid depicted in Figure 6, design should be a stable object. Yet, many software developers do design by taking the pyramid and standing it

on its point. That is, design begins with the creation of procedural detail, and as a result, interface, architectural, and data design just happen. This approach, common among people who insist upon coding the program with no explicit design activity, invariably leads to low-quality software that is difficult to test, challenging to extend, and frustrating to maintain. For a stable, high-quality product, the design approach must also be stable. The design pyramid provides the degree of stability necessary for good design.

7.4 Program Construction

The glory years of third-generation programming languages are rapidly coming to a close. Fourth-generation techniques, graphical programming methods, component-based software construction, and a variety of other approaches have already captured a significant percentage of all software construction activities, and there is little debate that their penetration will grow.

And yet, some members of the software engineering community continue to debate "the best programming language." Although entertaining, such debates are a waste of time. The problems that we continue to encounter in the creation of high-quality computer-based systems have relatively little to do with the means of construction. Rather, the challenges that face us can only be solved through better or innovative approaches to analysis and design, more comprehensive SQA techniques, and more effective and efficient testing. It is for this reason that construction is not emphasized in this paper.

7.5 Software Testing

Glen Myers [38] states three rules that can serve well as testing objectives:

1. Testing is a process of executing a program with the intent of finding an error.
2. A good test case is one that has a high probability of finding an as-yet-undiscovered error.
3. A successful test is one that uncovers an as-yet-undiscovered error.

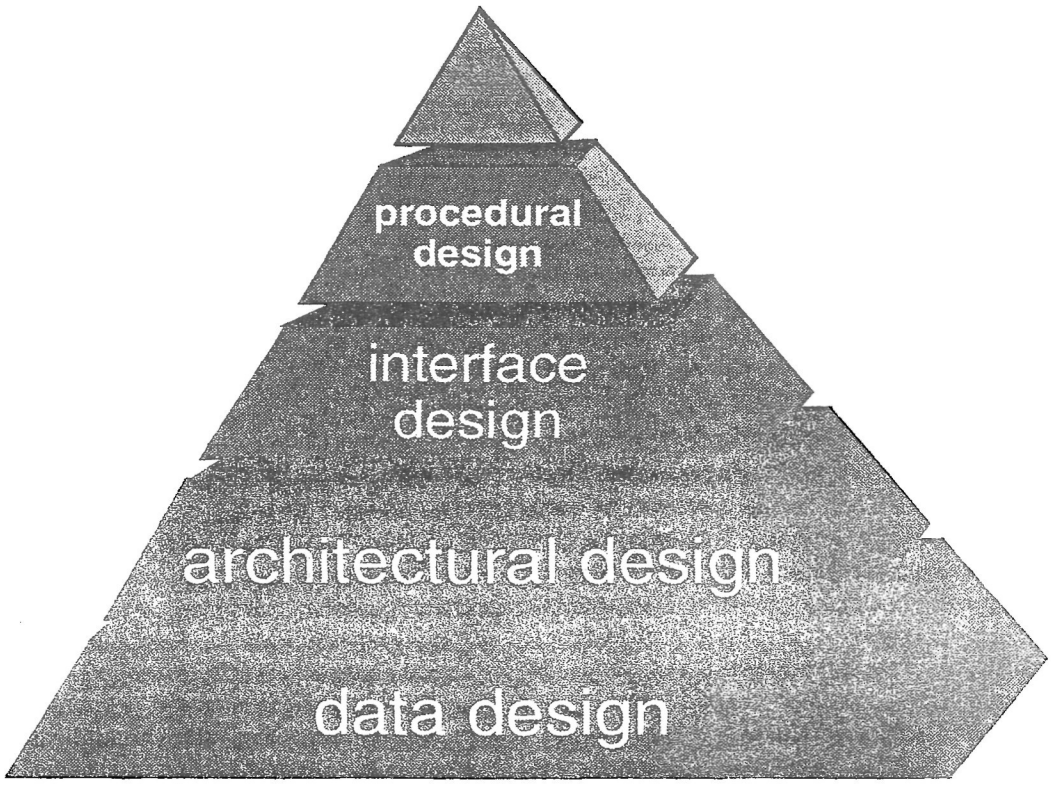

Figure 6. The design pyramid

These objectives imply a dramatic change in viewpoint. They move counter to the commonly held view that a successful test is one in which no errors are found. Our objective is to design tests that systematically uncover different classes of errors and to do so with a minimum of time and effort.

If testing is conducted successfully (according to the objective stated above), it will uncover errors in the software. As a secondary benefit, testing demonstrates that software functions appear to be working according to specification, that performance requirements appear to have been met. In addition, data collected as testing is conducted provides a good indication of software reliability and some indication of software quality as a whole. But there is one thing that testing cannot do: testing cannot show the absence of defects, it can only show that software defects are present. It is important to keep this (rather gloomy) statement in mind as testing is being conducted.

7.5.1 Strategy

A strategy for software testing integrates software test-case design techniques into a well-planned series of steps that result in the successful construction of software. It defines a template for software testing—a set of steps into which we can place specific test-case design techniques and testing methods.

A number of software testing strategies have been proposed in the literature. All provide the software developer with a template for testing, and all have the following generic characteristics:

- Testing begins at the module level and works incrementally "outward" toward the integration of the entire computer-based system.
- Different testing techniques are appropriate at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

A strategy for software testing must accommodate low-level tests that are necessary to verify that a small source code segment has been correctly implemented, intermediate-level tests designed to uncover errors in the interfaces between modules, and high-level tests that validate major system functions against customer requirements. A strategy must provide guidance for the practitioner and a set of milestones for the manager. Because the steps of the test strategy occur at a time when deadline pressure begins to rise, progress must be measurable and problems must surface as early as possible.

7.5.2 Tactics

The design of tests for software and other engineered products can be as challenging as the initial design of the product itself. Recalling the objectives of testing, we must design tests that have the highest likelihood of finding the most errors with a minimum of time and effort.

Over the past two decades a rich variety of test-case design methods have evolved for software. These methods provide the developer with a systematic approach to testing. More importantly, methods provide a mechanism that can help to ensure the completeness of tests and provide the highest likelihood for uncovering errors in software.

Any engineered product (and most other things) can be tested in one of two ways: (1) knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational; (2) knowing the internal workings of the product, tests can be conducted to ensure that "all gears mesh"; that is, internal operation performs according to specification and all internal components have been adequately exercised. The first test approach is called *black-box testing* and the second, *white-box testing* [38].

When computer software is considered, black-box testing alludes to tests that are conducted at the software interface. Although they are designed to uncover errors, black-box tests are also used to demonstrate that software functions are operational; that input is properly accepted, and output is correctly produced; that the integrity of external information (such as data files) is maintained. A black-box test examines some aspect of a system with little regard for the internal logical structure of the software.

White-box testing of software is predicated on close examination of procedural detail. Logical paths through the software are tested by providing test cases that exercise specific sets of conditions and/or loops. The status of the program may be examined at various points to determine if the expected or asserted status corresponds to the actual status.

8.0 The Road Ahead & The Three R's

Software is a child of the latter half of the twentieth century—a baby boomer. And like its human counterpart, software has accomplished much while at the same time leaving much to be accomplished. It appears that the economic and business environment of the next ten years will be dramatically different than anything that baby boomers have yet experienced.

Staff downsizing, the threat of outsourcing, and the demands of customers who won't take "slow" for an answer require significant changes in our approach to software engineering and a major reevaluation of our strategies for handling hundreds of thousands of existing systems [39].

Although many existing technologies will mature over the next decade, and new technologies will emerge, it's likely that three existing software engineering issues—I call them the three R's—will dominate the software engineering scene.

8.1 Reuse

We must build computer software faster. This simple statement is a manifestation of a business environment in which competition is vicious, product life cycles are shrinking, and time to market often defines the success of a business. The challenge of faster development is compounded by shrinking human resources and an increasing demand for improved software quality.

To meet this challenge, software must be constructed from reusable components. The concept of software reuse is not new, nor is a delineation of its major technical and management challenges [40]. Yet without reuse, there is little hope of building software in time frames that shrink from years to months.

It is likely that two regions of the methods landscape may merge as greater emphasis is placed on reuse. Object-oriented development can lead to the design and implementation of inherently reusable program components, but to meet the challenge, these components must be demonstrably defect free. It may be that formal methods will play a role in the development of components that are proven correct prior to their entry in a component library. Like integrated circuits in hardware design, these "formally" developed components can be used with a fair degree of assurance by other software designers.

If technology problems associated with reuse are overcome (and this is likely), management and cultural challenges remain. Who will have responsibility for creating reusable components? Who will manage them once they are created? Who will bear the additional costs of developing reusable components? What incentives will be provided for software engineers to use them? How will revenues be generated from reuse? What are the risks associated with creating a reuse culture? How will developers of reusable components be compensated? How will legal issues such as liability and copyright protection be addressed? These and many other questions remain to be answered. And yet, component reuse is our best hope for meeting the software challenges of the early part of the twenty-first century.

8.2 Reengineering

Almost every business relies on the day-to-day operation of an aging software plant. Major companies spend as much as 70 percent or more of their software budget on the care and feeding of legacy systems. Many of these systems were poorly designed more than decade ago and have been patched and pushed to their limits. The result is a software plant with aging, even decrepit systems that absorb increasingly large amounts of resource with little hope of abatement. The software plant must be rebuilt, and that demands a reengineering strategy.

Reengineering takes time; it costs significant amounts of money, and it absorbs resources that might be otherwise occupied on immediate concerns. For all of these reasons, reengineering is not accomplished in a few months or even a few years. Reengineering of information systems is an activity that will absorb software resources for many years.

A paradigm for reengineering includes the following steps:

- *inventory analysis*—creating a prioritized list of programs that are candidates for reengineering
- *document restructuring*—upgrading documentation to reflect the current workings of a program
- *code restructuring*—recoding selected portions of a program to reduce complexity, ready the code for future change, and improve understandability
- *data restructuring*—redesigning data structures to better accommodate current needs; redesign the algorithms that manipulate these data structures
- *reverse engineering*—examine software internals to determine how the system has been constructed
- *forward engineering*—using information obtained from reverse engineering, rebuild the application using modern software engineering practices and principles.

8.3 Retooling

To achieve the first two R's, we need a third R—a new generation of software tools. In retooling the software engineering process, we must remember the

mistakes of the 1980s and early 1990s. At that time, CASE tools were inserted into a process vacuum, and failed to meet expectations. Tools for the next ten years will address all aspects of the methods landscape. But they should emphasize reuse and reengineering.

9.0 Summary

As each of us in the software business looks to the future, a small set of questions is asked and re-asked. Will we continue to struggle to produce software that meets the needs of a new breed of customers? Will generation X software professionals repeat the mistakes of the generation that preceded them? Will software remain a bottleneck in the development of new generations of computer-based products and systems? The degree to which the industry embraces software engineering and works to instantiate it into the culture of software development will have a strong bearing on the final answers to these questions. And the answers to these questions will have a strong bearing on whether we should look to the future with anticipation or trepidation.

References

[1] Naur, P. and B. Randall (eds.), *Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee*, NATO, 1969.

[2] Paulk, M. et al., *Capability Maturity Model for Software*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1993.

[3] Boehm, B., "A Spiral Model for Software Development and Enhancement," *Computer*, Vol. 21, No. 5, May 1988, pp. 61–72.

[4] Gilb, T., *Principles of Software Engineering Management*, Addison-Wesley, Reading, Mass., 1988.

[5] Mills, H.D., M. Dyer, and R. Linger, "Cleanroom Software Engineering," *IEEE Software*, Sept. 1987, pp. 19–25.

[6] Dyer, M., *The Cleanroom Approach to Quality Software Development*, Wiley, New York, N.Y., 1992.

[7] Cougar, J. and R. Zawacki, *Managing and Motivating Computer Personnel*, Wiley, New York, N.Y., 1980.

[8] DeMarco, T. and T. Lister, *Peopleware*, Dorset House, 1987.

[9] Weinberg, G., *Understanding the Professional Programmer*, Dorset House, 1988.

[10] Curtis, B., "People Management Maturity Model," *Proc. Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1989, pp. 398–399.

[11] August, J.H., *Joint Application Design*, Prentice-Hall, Englewood Cliffs, N.J., 1991.

[12] Wood, J. and D. Silver, *Joint Application Design*, Wiley, New York, N.Y., 1989.

[13] Dreger, J.B., *Function Point Analysis*, Prentice-Hall, Englewood Cliffs, N.J., 1989.

[14] Hetzel, B., *Making Software Measurement Work*, QED Publishing, 1993.

[15] Jones, C., *Applied Software Measurement*, McGraw-Hill, New York, N.Y., 1991.

[16] Fenton, N.E., *Software Metrics*, Chapman & Hall, 1991.

[17] Zuse, H., *Software Complexity*, W. deGruyer & Co., Berlin, 1990.

[18] Lorenz, M. and J. Kidd, *Object-Oriented Software Metrics*, Prentice-Hall, Englewood Cliffs, N.J., 1994.

[19] Pressman, R.S., *A Manager's Guide to Software Engineering*, McGraw-Hill, New York, N.Y., 1993.

[20] Boehm, B., *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, N.J., 1981.

[21] Charette, R., *Application Strategies for Risk Analysis*, McGraw-Hill, New York, N.Y., 1990.

[22] Jones, C., *Assessment and Control of Software Risks*, Yourdon Press, 1993.

[24] Crosby, P., *Quality is Free*, McGraw-Hill, New York, N.Y., 1979.

[25] McCall, J., P. Richards, and G. Walters, "Factors in Software Quality," three volumes, NTIS AD-A049-014, 015, 055, Nov. 1977.

[26] Cavano, J.P. and J.A. McCall, "A Framework for the Measurement of Software Quality," *Proc. ACM Software Quality Assurance Workshop*, ACM Press, New York, N.Y., 1978, pp. 133–139.

[27] Freedman, D and G. Weinberg, *The Handbook of Walkthroughs, Inspections and Technical Reviews*, Dorset House, 1990.

[28] Gilb, T. and D. Graham, *Software Inspection*, Addison-Wesley, Reading, Mass., 1993.

[29] Babich, W., *Software Configuration Management*, Addison-Wesley, Reading, Mass., 1986.

[30] Pressman, R. and S. Herron, Software Shock, Dorset House, 1991.

[31] Pressman, R., *Software Engineering: A Practitioner's Approach*, 3rd ed., McGraw-Hill, New York, N.Y., 1992.

[32] Yourdon, E., *Modern Structured Analysis*, Yourdon Press, 1989.

[33] Booch, G., *Object-Oriented Analysis & Design*, Benjamin-Cummings, 1994.

[34] Spivey, M., *The Z Notation*, Prentice-Hall, Englewood Cliffs, N.J., 1992.

[35] Jackson, M., *Principles of Program Design*, Academic Press, New York, N.Y., 1975.

[36] Parnas, D.L., "On Criteria to be used in Decomposing Systems into Modules," *Comm. ACM*, Vol. 14, No. 1, Apr. 1972, pp. 221–227.

[37] Linger, R., H. Mills, and B. Witt, *Structured Programming*, Addison-Wesley, Reading, Mass., 1979.

[38] Myers, G., *The Art of Software Testing*, Wiley, New York, N.Y., 1979.

[38] Beizer, B., *Software Testing Techniques*, 2nd ed., VanNostrand Reinhold, 1990.

[39] Pressman, R., "Software According to Nicollo Machiavelli," *IEEE Software*, Jan. 1995, pp. 101–102.

[40] Tracz, W., *Software Reuse: Emerging Technology*, IEEE CS Press, Los Alamitos, Calif., 1988.

Software Engineering—20 Years On and 20 Years Back*

J. N. Buxton
Department of Computing, Kings College, London

This paper gives a personal view of the development of software engineering, starting from the NATO conferences some 20 years ago, looking at the current situation and at possible lines of future development. Software engineering is not presented as separate from computer science but as the engineering face of the same subject. It is proposed in the paper that significant future developments will come from new localized computing paradigms in specific application domains.

20 YEARS BACK

The start of the development of software engineering as a subject in its own right, or perhaps more correctly as a new point of view on computing, is particularly associated with the NATO conferences in 1968 and 1969. The motivations behind the first of these meetings was the dawning realization that major software systems were being fuilt by groups consisting essentially of gifted amateurs. The endemic problems of the "software crisis"—software was late, over budget, and unreliable—already affected small and medium systems for well-coordinated applications and would clearly have even more serious consequences for the really big systems which were being planned.

People in the profession typically had scientific backgrounds in mathematics, the sciences, or electronics. In 1968 we had already achieved the first big breakthrough in the subject—the development of high-level programming languages—and the second was awaited. The time was ripe for the proposition which was floated by the NATO Science Committee, and by Professor Fritz Bauer of Munich, among others, that we should consider the subject as a branch of engineering; other professions built big systems of great complexity and by and large they called themselves "engineers," and perhaps we should consider whether we should do the same. Our big systems problems seemed primarily to be on the software side and the proposition was that it might well help to look at software development from the standpoint of the general principles and methods of engineering. And so, the first NATO Conference on Software Engineering was convened in Garmisch Parten-Kirchen in Bavaria.

It was an inspiring occasion. Some 50 people were invited from among the leaders in the field and from the first evening of the meeting it was clear that we shared deep concerns about the quality of software being built at the time. It was not just late and over budget; even more seriously we could see, and on that occasion we were able to share our anxieties about, the safety aspects of future systems.

The first meeting went some way to describing the problems—the next meeting was scheduled to follow after a year which was to be devoted to study of the techniques needed for its solution. We expected, of course, that a year spent in studying the application of engineering principles to software development would show us how to solve the problem. It turned out that after a year we had not solved it and the second meeting in Rome was rather an anticlimax. However, the software engineering idea was now launched: it has steadily gathered momentum over some 20 years. Some would no doubt say that it has become one of the great bandwagons of our time; however, much has been achieved and many central ideas have been developed: the need for management as well as technology, the software lifecycle, the use of toolsets, the application of quality assurance techniques, and so on.

So during some 20 years, while the business has expanded by many orders of magnitude, we have been able to keep our heads above water. We have put computers to use in most fields of human endeavor, we

Address correspondence to Professor J. N. Buxton, Department of Computing, Kings College London, Strand, London WC2R2LS, England.

*The paper is a written version of the keynote address at the International Conference on Software Engineering, Pittsburgh, May, 1989.

Reprinted from *J. Systems and Software*, Vol. 13, J.N. Buxton, "Software Engineering—20 Years On and 20 Years Back," pp. 153–155, 1990, with kind permission from Elsevier Science–NL, Sara Burgerhartstraat 25; 1055 KV Amsterdam, The Netherlands.

have demonstrated that systems of millions of lines of code taking hundreds of man-years can be built to acceptable standards, and our safety record, while not perfect, has so far apparently not been catastrophic. The early concepts of 20 years ago have been much developed: the lifecycle idea has undergone much refinement, the toolset approach has been transformed by combining unified toolsets with large project data bases, and quality control and assurance is now applied as in other engineering disciplines. We now appreciate more clearly that what we do in software development can well be seen as engineering, which must be underpinned by scientific and mathematical developments which produce laws of behavior for the materials from which we build: in other words, for computer programs.

THE PRESENT

So, where is software engineering today? In my view there is indeed a new subject of "computing" which we have to consider. It is separate from other disciplines and has features that are unique to the subject. The artifacts we build have the unique property of invisibility and furthermore, to quote David Parnas, the state space of their behavior is both large and irregular. In other words, we cannot "see" an executing program actually running; we can only observe its consequences. These frequently surprise us and in this branch of engineering we lack the existence of simple physical limits on the extent of erroneous behavior. If you build a bridge or an airplane you can see it and you can recognize the physical limitations on its behavior—this is not the case if you write a computer program.

The subject of computing has strong links and relationships to other disciplines. It is underpinned by discrete mathematics and formal logic in a way strongly analogous to the underpinning of more traditional branches of engineering by physics and continuous mathematics. We expect increasing help from relevant mathematics in determining laws of behavior for our systems and of course we rely on electronics for the provision of our hardware components. A computer is an electronic artifact and is treated as such when it is being built or when it fails; at other times we treat it as a black box and assume it runs our program perfectly.

At the heart of the subject, however, we have the study of software. We build complex multilayered programs which eventually implement applications for people—who in turn treat the software as a black box and assume it will service their application perfectly.

So, where and what is software engineering? I do not regard it as a spearate subject. Building software is perhaps the central technology in computing and much of what we call software engineering is, in my view, the face of computing which is turned toward applications. The subject of computing has three main aspects: computer science is the face turned to mathematics, from which we seek laws of behavior for programs; computer architecture is the face turned to the electronics, from which we build our computers; and software engineering is the face turned toward the users, whose applications we implement. I think the time has come to return to a unified view of our subject—software engineering is not something different from computer science or hardware design—it is a different aspect or specialty within the same general subject of computing.

20 YEARS ON

To attempt to answer the question, Where should we go next and what of the next 20 years? opens interesting areas of speculation. As software engineers, our concerns are particularly with the needs of the users and our aims are to satisfy these needs. Our techniques involve the preparation of computer programs and I propose to embark on some speculation based on a study of the levels of language inwhich these programs are written.

The traditional picture of the process of implementing an application is, in general, as follows. The user presents a problem, expressed in his own technical terminology or language, which could be that of commerce, nuclear physics, medicine or whatever, to somebody else who speaks a different language in the professional sense. This is the language of algorithms and this person devises an algorithmic solution to the specific user problem, i.e., a solution expressed in computational steps, sequences, iterations, choices. This person we call the "systems analyst" and indeed, in the historical model much used in the data processing field, this person passes the algorithms on to a "real programmer" who thinks and speaks in the codeof the basic computer hardware.

The first major advance in computing was the general introduction of higher level languages such as FORTRAN and COBOL—as in effect this eliminated from all but some specialized areas the need for the lower level of language, i.e., achine or assembly code programming. The separate roles of systems analyst and programmer have become blurred into the concept of the software engineer who indeed thinks in algorithms but expresses these directly himself in one of the fashionable languages of the day. This indeed is a breakthrough and has given us an order of magnitude advance, whether we measure it in terms of productivity, size of application we can tackle, or quality of result.

Of course we have made other detailed advances—we have refined our software engineering techniques, we have devised alternatives to algorithmic programming

in functional and rule-based systems, and we have done much else. But in general terms we have not succeeded in making another general advance across the board in the subject.

In some few areas, however, there have been real successes which have brought computing to orders of magnitude more people with applications. These have been in very specific application areas, two of which spring to mind—spreadsheets and word processors. In my view, study of these successes gives us most valuable clues as to the say ahead for computing applications.

The spreadsheet provides a good example for study. The generic problem of accountancy is the presentation of a set of figures which are perhaps very complexly related but which must be coherent and consistent. The purpose is to reveal a picture of the formal state of affairs of an enterprise (so far, of course, as the accountatn thinks it wise or necessary to reveal it). The traditional working language of the accountatn is expressed in rows and columns of figures on paper together with their relationships. Now, the computer-based spreadsheet system automates the piece of paper and gives it magical extra properties which maintain consistency of the figure under the relationships between them as specified by the accountant, while he adjusts the figures. In effect, the computer program automates the generic problem rather than any specific set of accounts and so enables the accountant to express his problem of the moment and to seek solutions in his own professional language. He need know nothing, for example, of algorithms, high-level languages, or von Neuman machines, and the intermediary stages of the systems analyst and programmer have both disappeared from his view.

The same general remarks can be applied to word processing. Here what the typist sees is in effect a combination of magic correcting typewriter and filing cabinet—and again the typist need know nothing of algorithms. In both these examples there has been conspicuous success in introduction. And there are others emerging, e.g., hypertext. And historically there is much in the thesis that relates to the so-called 4GLs and, even earlier, to simulation languages in the 1960s.

Let me return to the consideration of levels of language, and summarize the argument so far. I postulated a traditional model for complete applications in which a specific user problem, expressed in the language of the domain of application, underwent a two-stage translation: first into algorithms (or some language of similar level such as functional applications or Horn logic clauses) and second down into machine code. Our first breakthrough was to automate the lower of these stages by the introduction of high-level languages, primarily algorithmic. I now postulate that the second breakthrough will come in areas where we can automate the upper stage. Examples can already be found: very clearly in specific closed domains such as spreadsheets and word processors but also in more diffuse areas addressed by very high-level languages such as 4GLs and simulation generators.

Perhaps I should add a word here about object-orientedness, as this is the best known buzz word of today. I regard an object-oriented approach as a halfway house to the concept I am proposing. Objects indeed model those features of the real world readily modelable as classes of entities—that is why we invented the concept in the simulation languages of the early 1960s. However, the rules of behavior of the object are still expressed algorithmically and so an object-oriented system still embodies a general purpose language.

It is central to the argument to realize that spreadsheets and such can be used by people who do not readily think in terms of algorithms. The teaching of programming has demonstrated over many years that thinking in algorithms is a specific skill and most people have little ability in transposing problems from their own domains into algorithmic solutions. Attempts to bring the use of computers to all by teaching them programming do not work; providing a service to workers in specific domains directly in the language of that domain, however, does work and spectacularly so.

CONCLUSIONS

I come to the conclusion, therefore, that the most promising activity for the next 20 years is the search for more domains of applications in which the language used to express problems in the domain is closed, consistent, and logically based. Then we can put forward generic computer-based systems which enable users in the domain to express their problems in the language of the application and to be given solutions in their own terms. To use another buzz word of the day, I look for non-specific but localized paradigms for computing applications.

Of course this is not all that we might expect to do in the next 20 years. We can do much more in developing the underpinning technology in the intermediate levels between the user and the machine. Most of our work will still be devoted to the implementing of systems to deliver solutions to specific problems. But while we proceed with the day-to-day activities of software engineering or of the other faces of computing with which we may be concerned, wemight be wise to look out for opportunities to exploit new application domains where we can see ways to raise the "level of programming language" until it becomes the same as the professional language of that domain. Then, we will achieve another breakthrough.