Software Engineering

Roger S. Pressman, Ph.D.

As software engineering moves into 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 The innocence and enthusiasm of its early years have been replaced by more reasonable expectations (and even a healthy cynicism) fostered by years of experience.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 midlife 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. SOFTWARE ENGINEERING—A LAYERED TECHNOLOGY

Although hundreds of authors have developed personal definitions of software engineering, a definition proposed by Fritz Bauer [2] 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 do we "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. Any engineering approach (including software engineering) must rest on an organizational commitment to quality (Figure 1). Total Quality Management, Six Sigma, 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* [3] 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, work products (models, documents, data, reports, forms, etc.) 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 engineering and analysis, design, program construction, testing, and software 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.

Portions of this paper (including the figures) were adapted from *Software Engineering: A Practitioner's Approach*, 6th ed. [1] and are used with the permission of McGraw-Hill.

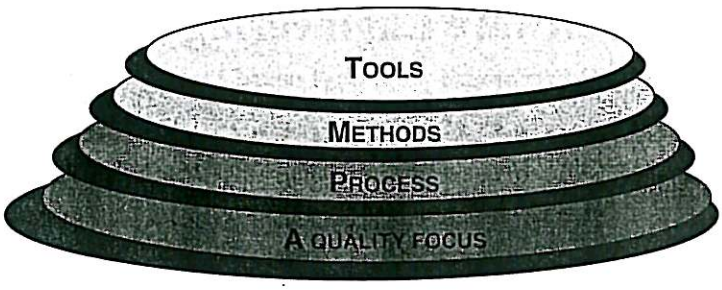

Figure 1 Software engineering layers.

2. A PROCESS FRAMEWORK

A process framework establishes the foundation for a complete software process by identifying a small number of *framework activities* that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of *umbrella activities* that are applicable across the entire software process.

Each framework activity (Figure 2) is populated by a set of *software engineering actions*—a collection of related tasks that produces a major software engineering work product (e.g., design is a software engineering action). Each action is populated with individual *work tasks* that accomplish some part of the work implied by the action.

The following *generic process framework* (used as a basis for the description of process models) is applicable to the vast majority of software projects:

- **Communication.** This framework activity involves heavy communication and collaboration with the customer (and other stakeholders[1]) and encompasses requirements gathering and other related activities.
- **Planning.** This activity establishes a plan for the software engineering work that follows. It describes the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule. It also addresses a team's approach to quality assurance and configuration management.
- **Modeling.** This activity encompasses the creation of models that enable the developer and the customer to better understand software requirements and the design that will achieve those requirements.
- **Construction.** This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the design or code.
- **Deployment.** The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the product and provides feedback based on the evaluation.

These five generic framework activities can be used during the development of small, simple programs; the creation of large Web applications; and for the engineering of large, complex computer-based systems. The details of the software process will be quite different in each case, but the framework activities remain the same.

The common process framework is complemented by a number of *umbrella activities*. Typical activities in this category include the following:

- **Software project tracking and control** allows the software team to assess progress against the project plan and take necessary action to maintain schedule.
- **Risk management** assesses risks that may affect the outcome of the project or the quality of the product.
- **Software quality assurance** defines and conducts the activities required to ensure software quality.
- **Formal technical reviews** assess software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.
- **Measurement** defines and collects process, project, and product measures that assist the team in delivering software that meets customer's needs and can be used in conjunction with all other framework and umbrella activities.

[1] A *stakeholder* is anyone who has a stake in the successful outcome of the project—business managers, end users, software engineers, support people, and so on. Rob Thomsett jokes that "a stakeholder is a person holding a large and sharp stake. . . . If you don't look after your stakeholders, you know where the stake will end up."

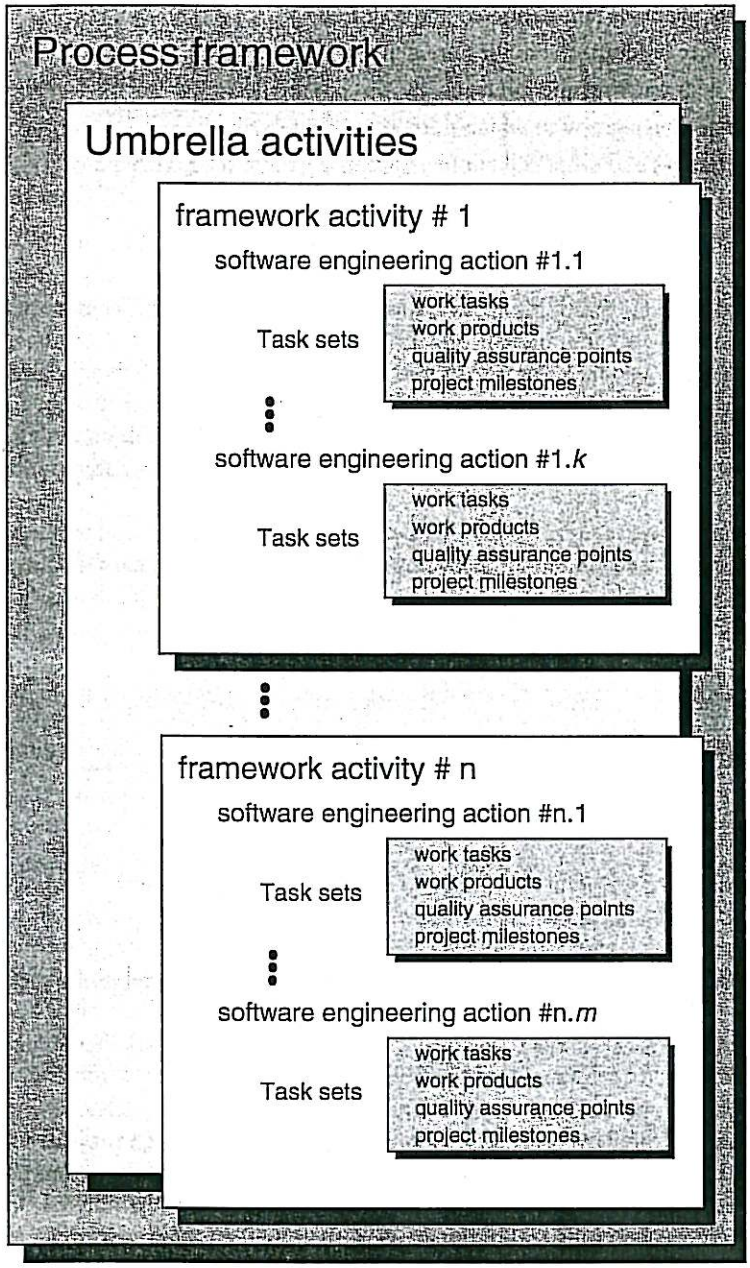

Figure 2 A software process framework.

- **Software configuration management** manages the effects of change throughout the software process.
- **Reusability management** defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.
- **Work product preparation and production** encompasses the activities required to create all work products such as models, documents, logs, forms, and lists.

All process models can be characterized within the process framework shown in Figure 2. Intelligent application of any software process model must recognize that adaptation (to the problem, to the project, to the people doing the work, and to the organizational culture) is essential for success. But process models do differ in fundamental ways:

- The overall flow of activities and tasks and the interdependencies among activities and tasks
- The degree to which work tasks are defined within each framework activity

- The degree to which work products are identified and required
- The manner in which quality assurance activities are applied
- The manner in which project tracking and control activities are applied
- The overall degree of detail and rigor with which the process is described
- The degree to which customers and other stakeholders are involved in the process
- The level of autonomy given to the software project team
- The degree to which team organization and roles are prescribed

Process models that stress detailed definition, identification, and application of process activities and tasks have been applied within the software engineering community for the past 30 years. When these *prescriptive process models* are applied, the intent is to improve system quality, to make projects more manageable, to make delivery dates and costs more predictable, and to guide teams of software engineers as they perform the work required to build a system. Unfortunately, there are times when these objectives are not achieved. If prescriptive models are applied dogmatically and without adaptation, they can increase the level of bureaucracy associated with building computer-based systems and inadvertently create difficulty for developers and customers.

Process models that emphasize project "agility" and follow a set of principles that lead to a more informal (but, proponents argue, no less effective) approach to software process have been proposed in recent years. These process models are generally characterized as *agile* because they emphasize maneuverability and adaptability. They are appropriate for many types of software projects and are particularly useful when Web applications are engineered.

Which software process philosophy is best? This question has spawned emotional debate among software engineers. It is important to note, however, that both process philosophies have a common goal—to create high quality software that meets the customer's needs.

3. SOFTWARE PROCESS MODELS

Software engineering incorporates a development strategy that encompasses the process, methods, and tools layers described earlier. 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 work products that are required. An overview of important process models is presented in the sections that follow.

3.1. Prescriptive Models

Prescriptive software process models prescribe a set of process elements: framework activities, software engineering actions, tasks, work products, and quality assurance and change control mechanisms for each project. Each process model also prescribes a workflow, that is, the manner in which the process elements are interrelated to one another.

All prescriptive process models accommodate the generic framework activities that have been described earlier, but each applies a different emphasis to these activities and defines a workflow that invokes each framework activity (as well as software engineering actions and tasks) in a different manner.

3.1.1. The Waterfall Model

There are times when the requirements for a problem are reasonably well understood, when work flows from communication through deployment in a reasonably linear fashion. This situation is sometimes encountered when well-defined adaptations or enhancements to an existing system must be made (e.g., an adaptation to accounting software that has been mandated because of changes to government regulations). It may also occur in a limited number of new development efforts, but only when requirements are well defined and reasonably stable.

The *waterfall model,* sometimes called the *classic life cycle* (Figure 3) suggests a systematic, sequential approach[2] to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating with on-going support of the completed software.

The waterfall model is the oldest paradigm for software engineering. However, over the past two decades, criticism of this

[2]Although the original waterfall model proposed by Winston Royce [4] made provision for "feedback loops," the vast majority of organizations that apply this process model treat it as if were strictly linear.

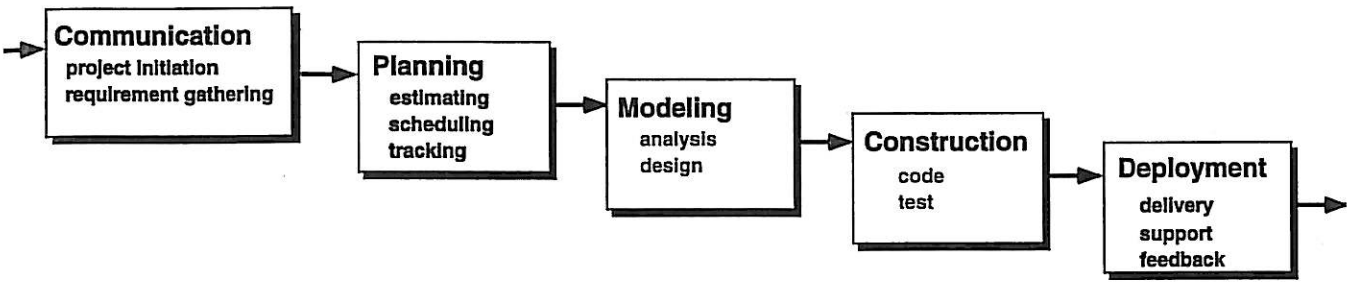

Figure 3 The waterfall model

process model has caused even ardent supporters to question its efficacy [5]. Among the problems that are sometimes encountered when the waterfall 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 waterfall 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.

Today, software work is fast-paced and subject to a never-ending stream of changes (to features, functions, and information content). The waterfall model is often inappropriate for such work.

3.1.2. Incremental Process Models

There are many situations in which initial software requirements are reasonably well defined but the overall scope of the development effort precludes a purely linear process. In addition, there may be a compelling need to provide limited software functionality to users quickly and then refine and expand on that functionality in later software releases. In such cases, a process model that is designed to produce the software in increments is chosen.

The *incremental model* combines elements of the waterfall model applied repetitively in an iterative fashion. The incremental model (Figure 4) applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software. For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing and document production functions in the first increment; more sophisticated editing, and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment may incorporate the prototyping paradigm discussed in Section 3.1.3.

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 evaluation). 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 software increment, until the complete product is produced.

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.

3.1.3 Evolutionary Process Models

Software, like all complex systems, evolves over a period of time. Business and product requirements often change as development proceeds, making a straight-line path to an end product unrealistic. Tight market deadlines may make completion of a

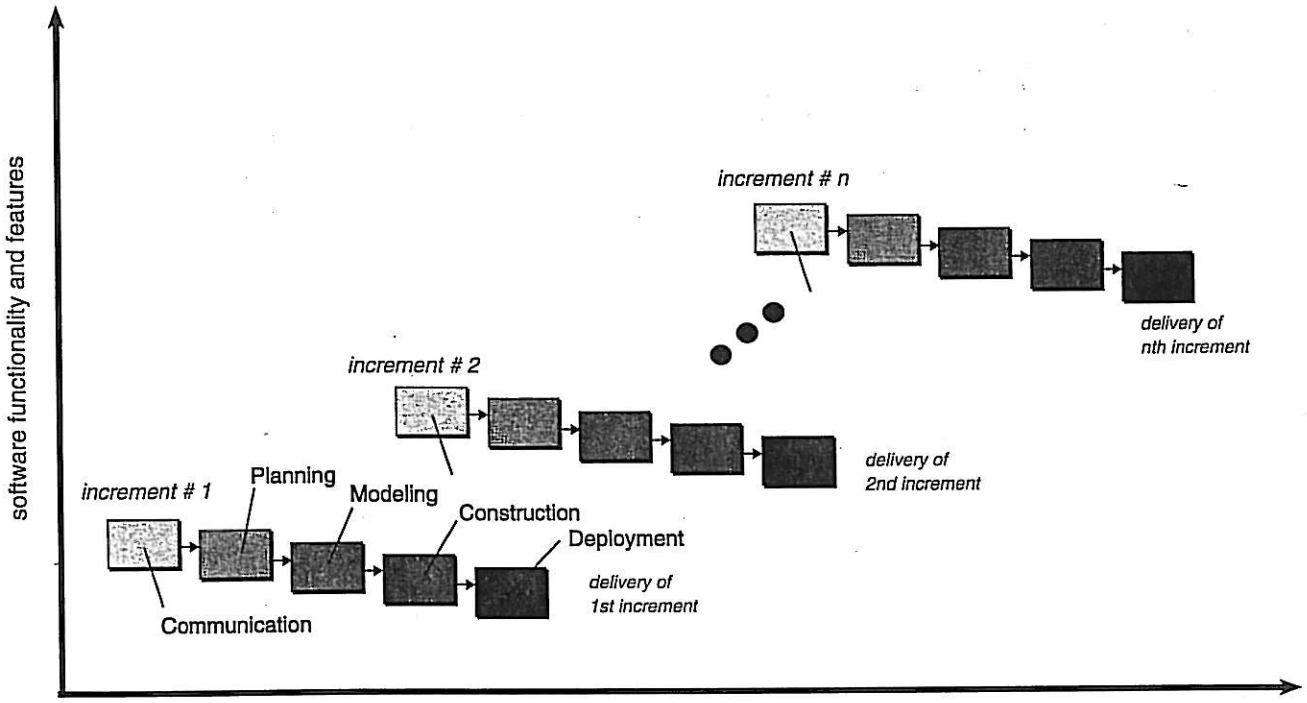

Figure 4 The incremental model.

comprehensive software product impossible, but a limited version may have to be introduced to meet competitive or business pressure. A set of core product or system requirements may be well understood, but the details of product or system extensions may have yet to have been defined. In these and similar situations, software engineers need a process model that has been explicitly designed to accommodate a product that evolves over time.

Evolutionary models are iterative. They are characterized in a manner that enables software engineers to develop increasingly more complete versions of the software.

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.

Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models noted in this paper. Regardless of the manner in which it is applied, the prototyping paradigm assists the software engineer and the customer to better understand what is to be built when requirements are fuzzy.

The prototyping paradigm (Figure 5) begins with communication. The software engineer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas in which further definition is mandatory. A prototyping iteration is planned quickly and modeling (in the form of a "quick design") occurs. The quick design focuses on a representation of those aspects of the software that will be visible to the customer or end user (e.g., human interface layout or output display formats). The quick design leads to the construction of a prototype. The prototype is evaluated by various stakeholders and feedback is used to refine requirements for the software to be developed. Iteration occurs as the prototype is tuned to satisfy the needs of the stakeholders, while at the same time enabling the developer to better understand what needs to be done.

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," and unaware that in the rush to get it working overall software quality or long-term maintainability have not been considered. 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, management relents.

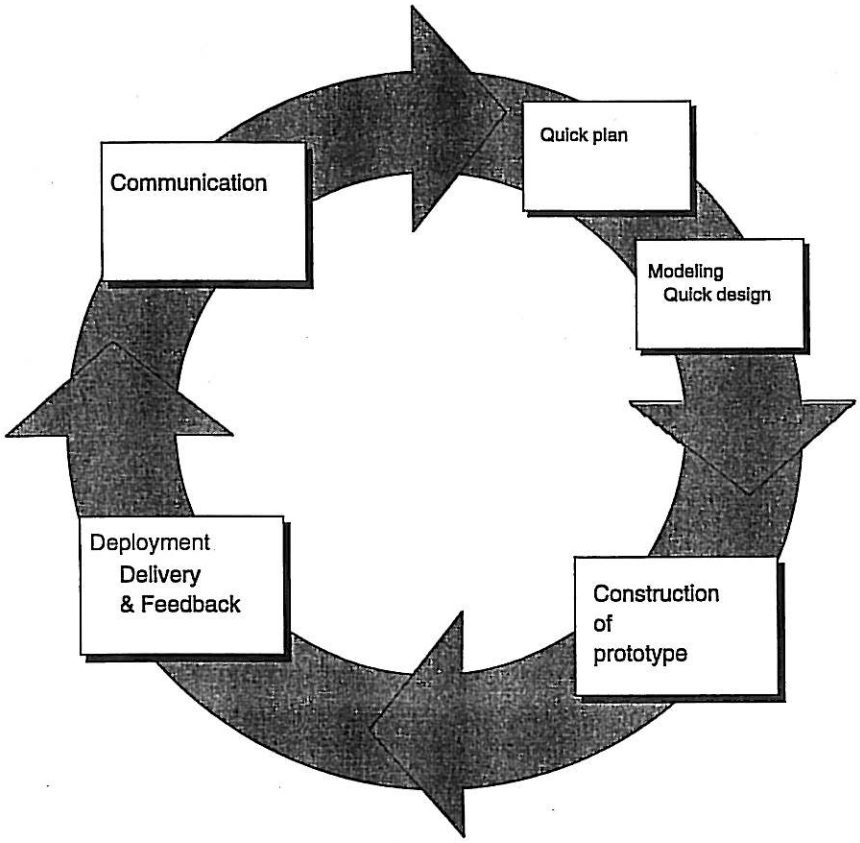

Figure 5 The prototyping paradigm.

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 or an inefficient algorithm may be implemented 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.

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.

The Spiral Model. The spiral model, originally proposed by Boehm [6], is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more complete versions of the software.

Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

A spiral model is divided into a set of framework activities defined by the software engineering team. For illustrative purposes, we use the generic framework activities discussed earlier.[3] Each of the framework activities represents one segment of the spiral path illustrated in Figure 6. As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center. Risk is considered as each revolution is

[3]The spiral model discussed in this section is a variation on the model proposed by Boehm. For further information on the original spiral model, see [6]. More recent discussion of Boehm's spiral model can be found in [7].

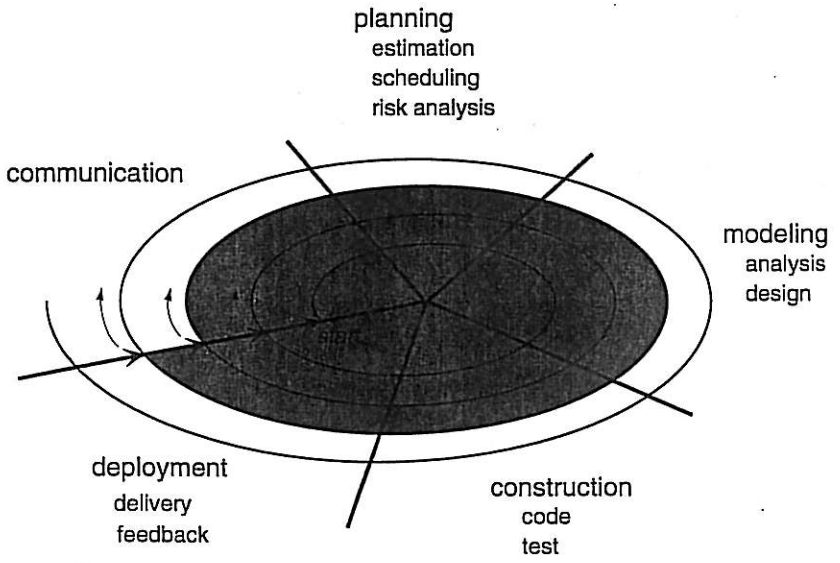

Figure 6 A typical spiral model.

made. *Anchor point milestones*—a combination of work products and conditions that are attained along the path of the spiral—are noted for each evolutionary pass.

The first circuit around the spiral might result in the development of a product specification and subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manager adjusts the planned number of iterations required to complete the software.

The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level. The spiral model uses prototyping as a risk reduction mechanism but, more important, 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 uncovered and managed, problems will undoubtedly occur.

3.1.4. Specialized Process Models

Special process models take on many of the characteristics of one or more of the prescriptive models presented in the preceding sections. However, these models tend to be applied when a specialized or narrowly defined software engineering approach is chosen.[4] A brief overview of a number of specialized models is presented in this section.

Component-Based Development. Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, can be used when software is to be built. These components provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built.

The *component-based development* model incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software. However, the component-based development model composes applications from prepackaged software components.

[4]In some cases, these specialized process models might better be characterized as a collection of techniques or a "methodology" for accomplishing a specific software development goal. However, they do imply a process.

The Formal Methods Model. The formal methods model encompasses a set of activities that lead to formal mathematical specification of computer software. Formal methods [8, 9] 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* [10, 11], is used by some software development organizations.

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 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:

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

These concerns notwithstanding, the formal methods approach has gained adherents among software developers who must build safety-critical software (e.g., developers of aircraft avionics and medical devices) and among developers who would suffer severe economic hardship should software errors occur.

Aspect-Oriented Software Development. Regardless of the software process that is chosen, the builders of complex software invariably implement a set of localized features, functions, and information content. These localized software characteristics are modeled as components (e.g., object-oriented classes) and then constructed within the context of a system architecture. As modern computer-based systems become more sophisticated (and complex), certain "concerns"—customer required properties or areas of technical interest—span the entire architecture. Some concerns are high-level properties of a system (e.g., security, fault tolerance). Other concerns affect functions (e.g., the application of business rules), wereas others are systemic (e.g., task synchronization or memory management).

When concerns cut across multiple system functions, features, and information, they are often referred to as *crosscutting concerns. Aspectual requirements* define those crosscutting concerns that have impact across the software architecture. *Aspect-oriented software development* (AOSD), often referred to as *aspect-oriented programming* (AOP), is a relatively new software engineering paradigm that provides a process and methodological approach for defining, specifying, designing, and constructing *aspects*—"mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern." [12] For additional information, the interested reader should see [12–14].

3.1.5. The Unified Process

The Unified Process (UP)[5] [15] is an attempt to draw on the best features and characteristics of other prescriptive software process models, but characterize them in a way that implements many of the best principles of agile software development (Section 3.2). The UP[5] recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system (the use case[6]). It emphasizes the important role of software architecture and "helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse" [15]. It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

Figure 7 depicts the "phases" of the UP and relates them to the generic activities that have been discussed in Section 2.0.

The *inception phase* of the UP encompasses both customer communication and planning activities. By collaborating with the customer and end users, business requirements for the software are identified, a rough architecture for the system is proposed, and a plan for the iterative, incremental nature of the ensuing project is developed.

The *elaboration phase* encompasses the customer communication and modeling activities. Elaboration refines and expands the preliminary use cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software—the use-case model, the analysis model, the design model, the implementation

[5]The Unified process is generally used in conjunction with the Unified Modeling Language (UML), a widely used notation for analysis and design modeling.
[6]A *use case* is a text narrative or template that describes a system function or feature from the user's point of view. A use case is written by the user and serves as a basis for the creation of a more comprehensive analysis model.

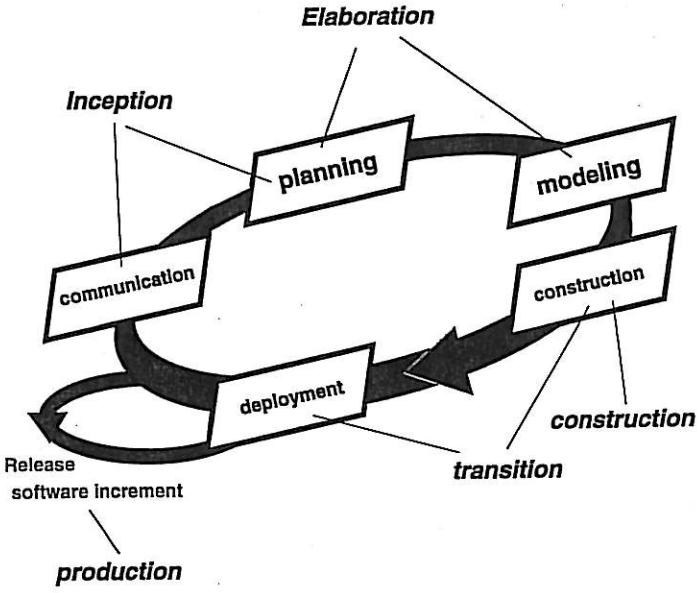

Figure 7 The Unified Process (UP).

model, and the deployment model. In some cases, elaboration creates an "executable architectural baseline" [16] that represents a "first cut" executable system.[7]

The *construction phase* of the UP is identical to the construction activity defined for the generic software process. Using the architectural model as input, the construction phase develops or acquires the software components that will make each use case operational for end users.

The *transition phase* of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment activity. Software is given to end users for beta testing[8] and user feedback reports both defects and necessary changes. In addition, the software team creates the necessary support information (e.g., user manuals, troubleshooting guides, installation procedures) that is required for the release. At the conclusion of the transition phase, the software increment becomes a usable software release.

The *production phase* of the UP coincides with the delivery and feedback activity of the generic process. During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.

3.2. Agile Software Development

In 2001, Kent Beck and 16 other noted software developers, writers, and consultants [17] (referred to as the "Agile Alliance") signed the "Manifesto for Agile Software Development." It stated:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
> *Individuals and interactions* over processes and tools
> *Working software* over comprehensive documentation
> *Customer collaboration* over contract negotiation
> *Responding to change* over following a plan
> That is, while there is value in the items on the right, we value the items on the left more.

A manifesto is normally associated with an emerging political movement, one that attacks the old guard and suggests revolutionary change (hopefully for the better). In some ways, that is exactly what agile development is all about.

---

[7] It is important to note that the architectural baseline is not a prototype in that it is not thrown away. Rather, the baseline is fleshed out during the next UP phase.

[8] *Beta testing* is a controlled testing activity in which the software is used by actual end users with the intent of uncovering defects and deficiencies. A formal defect/deficiency reporting scheme is established and the software team assesses feedback.

Although the underlying ideas that guide agile development have been with us for many years, it has only been during the past decade that these ideas have crystallized into a "movement." In essence, agile[9] methods were developed in an effort to overcome perceived and actual weaknesses in conventional (prescriptive) software engineering. Agile development can provide important benefits, but it is not applicable to all projects, all products, all people, and all situations. It is also *not* antithetical to solid software engineering practice and can be applied as an overriding philosophy for all software work.

Any agile software process is characterized in a manner that addresses a number of key assumptions [18] about the majority of software projects:

1. It is difficult to predict in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds.
2. For many types of software, design and construction are interleaved. That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design.
3. Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

Given these three assumptions, an important question arises: How do we create a process that can manage *unpredictability*? The answer, as we have already noted, lies in process adaptability (to rapidly changing project and technical conditions). An agile process, therefore, must be *adaptable*.

But continual adaptation without forward progress accomplishes little. Therefore, an agile software process must adapt *incrementally*. To accomplish incremental adaptation, an agile team requires customer feedback (so that the appropriate adaptations can be made). An effective catalyst for customer feedback is an operational prototype or a portion of an operational system. Hence, an *incremental development strategy* (Section 3.1.2) should be instituted. *Software increments* (executable prototypes or a portions of an operational system) must be delivered in short time periods so that adaptation keeps pace with change (unpredictability). This iterative approach enables the customer to evaluate the software increment regularly, provide necessary feedback to the software team, and influence the process adaptations that are made to accommodate the feedback.

The most widely discussed agile process model is *Extreme Programming* (XP).[10] XP uses an object-oriented approach as its preferred development paradigm and encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing. Figure 8 illustrates the XP process and notes some of the key ideas and tasks that are associated with each framework activity. Key XP activities are summarized in the paragraphs that follow.

Planning. The planning activity begins with the creation of a set of "stories" (also called *user stories*) that describe required features and functionality for software to be built. Each *story* is written by the customer and is placed on an index card. The customer assigns a *value* (i.e., a priority) to the story based on the overall business value of the feature or function.[11] Members of the XP team then assess each story and assign a *cost*—measured in development weeks—to it. Customers and the XP team work together to decide how to group stories into the next release (the next software increment) to be developed by the XP team.

As development work proceeds, the customer can add stories, change the value of an existing story, split stories, or eliminate them. The XP team then reconsiders all remaining releases and modified its plans accordingly.

Design. XP design rigorously follows the KIS (keep it simple) principle. A simple design is always preferred over a more complex representation. In addition, the design provides implementation guidance for a story as it is written—nothing less, nothing more. The design of extra functionality (because the developer assumes it will be required later) is discouraged.[12]

If a difficult design problem is encountered as part of the design of a story, XP recommends the immediate creation of an operational prototype of that portion of the design. Called a *spike solution,* the design prototype is implemented and evaluated. The intent is to lower risk when true implementation starts and to validate the original estimates for the story containing the design problem.

XP encourages *refactoring*—a construction technique that is also a design technique. Fowler [18] describes refactoring in the following manner:

> Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves the internal structure. It is a disciplined way to clean up code [and modify/sim-

[9]Agile methods are sometimes referred to as *light methods* or *lean methods*.

[10]It is important to note that many other agile process models have been proposed. Among them are Adaptive Software Development [19], DSDM [20], Scrum [21], Crystal [22], and Feature Driven Development [23].

[11]The value of a story may also be dependent on the presence of another story.

[12]These design guidelines should be followed in every software engineering method, although there are times when sophisticated design notation and terminology may get in the way of simplicity.

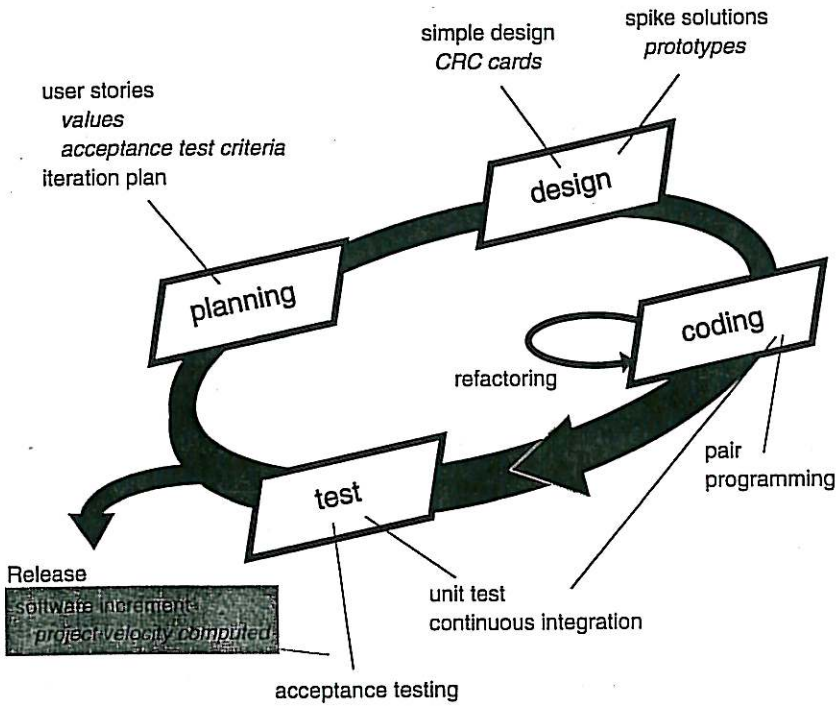

Figure 8 The Extreme Programming process.

plify the internal design] that minimizes the chances of introducing bugs. In essence, when you refactor you are improving the design of the code after it has been written.

In XP, design is viewed as a transient artifact that can and should be continually modified as construction proceeds. The intent of refactoring is to control these modifications by suggesting small design changes that "can radically improve the design" [18]. It should be noted, however, that effort required for refactoring can grow dramatically as the size of an application grows.

A central notion in XP is that design occurs both before *and after* coding commences. Refactoring means that design occurs continuously as the system is constructed. In fact, the construction activity itself will provide the XP team with guidance on how to improve the design.

Coding. XP recommends that after stories are developed and preliminary design work is done, the team should not move to code, but rather develop a series of unit tests that will exercise each of the stories that are to be included in the current release (software increment).[13] Once the unit test has been created, the developer is better able to focus on what must be implemented to pass the unit test. Nothing extraneous it added (KIS). Once the code is complete, it can be unit tested immediately, thereby providing instantaneous feedback to the developers.

A key concept during the coding activity (and one of the most talked about aspects of XP) is *pair programming.* XP recommends that two people work together at one computer workstation to create code for a story. This provides a mechanism for real-time problem solving (two heads are often better than one) and real-time quality assurance.

Testing. We have already noted that the creation of unit test[14] before coding commences is a key element of the XP approach. The unit tests that are created should be implemented using a framework that enables them to be automated (hence, they can be executed easily and repeatedly). This encourages a regression testing strategy whenever code is modified (which is often, given the XP refactoring philosophy).

XP *acceptance tests,* also called *customer tests,* are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer. Acceptance tests are derived from user stories that have been implemented as part of a software release.

[13]This approach is analogous to knowing the exam questions before you begin to study. It makes studying much easier by focusing attention only on the questions that will be asked.

[14]Unit testing focuses on an individual software component, exercising the component's interface, data structures, and functionality in an effort to uncover errors that are local to the component.

4. THE MANAGEMENT SPECTRUM

Effective software project management focuses on the three Ps: *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 stakeholder 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.

4.1. People

The cultivation of motivated, highly skilled software people has been discussed since the 1960s (e.g., [25–27]). The Software Engineering Institute (SEI) 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" [28].

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.

4.2. 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 estimates of the cost, a realistic breakdown of project tasks, or a manageable project schedule that provides a meaningful indication 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* [29, 30]. 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.

4.3. The Process

The Software Engineering Institute has developed a comprehensive process meta-model that is predicated on a set of system and software engineering capabilities that should be present as organizations reach different levels of process capability and maturity. To achieve these capabilities, the SEI contends that an organization should develop a process model that conforms to *The Capability Maturity Model Integration* (CMMI) guidelines [31].

The CMMI represents a process meta-model in two different ways: (1) as a "continuous" model and (2) as a "staged" model. The continuous CMMI meta-model assesses different process areas (e.g., project planning requirements management, measurement and analysis, configuration management) against specific goals and practices and rates each process area on a five-point scale.

The CMMI defines each process area in terms of "specific goals" and the "specific practices" required to achieve these goals. *Specific goals* establish the characteristics that must exist if the activities implied by a process area are to be effective. *Specific practices* refine a goal into a set of process-related activities.

The staged CMMI model defines the same process areas, goals, and practices as the continuous model. The primary difference is that the staged model defines five maturity levels, rather than five capability levels. To achieve a maturity level, the specific goals and practices associated with a set of process areas must be achieved.

5. 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 mature software engineering organizations.

5.1. Measurement and Metrics

To be most effective, software metrics should be collected for both the process and the product. *Process-oriented metrics* [32, 33] can be collected during the process and after it has been completed. Process metrics collected during the software process 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 [34], 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 [34–37]. 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.

5.2. 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 [1] for software projects:

1. **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.
2. **Size-Oriented Estimation** creates a list of major system functions 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 (e.g., function points per person month) for similar functions or projects are used to generate an estimate of effort required for each function.
3. **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 (see, e.g., [38]).

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

5.3. 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 [39–41] is actually a set 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.

5.4. 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 timelines that are used in constructing a project schedule.

5.5. 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.

6. SOFTWARE QUALITY ASSURANCE

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. If fact, a definitive 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 goes unmentioned (e.g., the desire for good maintainability). If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

More than three decades ago, McCall and Cavano [42, 43] defined a set of quality factors that are a first step toward the development of metrics for software quality. These factors assess 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, i.e., "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. The extent to which access to software or data by unauthorized persons can be controlled.
Usability. The effort require to learn, operate, prepare input, and interpret output of a program.
Maintainability. The effort require to locate and fix an error in a program. (Might be better termed "correctability.")
Flexibility. The effort required to modify an operational program.
Testability. The effort required to test a program to insure that it performs its intended function.
Portability. The effort required to transfer the program from one hardware and/or software system environment to another.
Reusability. The 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. The effort required to couple one system to another.

The intriguing thing about these factors is how little they have changed in over 30 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 is that 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* (FTR). Conducted as a *walkthrough* or an *inspection* [44], the FTR 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 [45].

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 object of SQA is to track software quality and assess the impact of methodological and procedural changes on 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.

7. 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 [46] 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 process. Because change can occur at any time, SCM activities are developed to (1) identify changes, (2) control changes, (3) ensure that changes are being properly implemented, and (4) report changes 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.

8. 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 [47] 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 relatively 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 cannot 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.

8.1. 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, (3) the construction of 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:

1. Conventional software engineering methods
2. Object-oriented approaches
3. 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.

8.2. 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 [29, 30] have been discussed earlier in this paper. In essence, software engineers 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 built, and/or (2) the creation of a detailed set of analysis models that describe the data, function, and behavior for the system.

8.2.1. Analysis Principles

Today, analysis modeling can be accomplished by applying one of a number of different methods that populate the three regions of the software-engineering methods landscape. Yet, all methods conform to a set of analysis principles [1]:

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 objects 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.

8.2.2. 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 that are 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 to understand 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

There are many different ways to look at the requirements for a computer-based system. Some software people argue that it is best to select one mode of representation (e.g., the use case) and apply it to the exclusion of all other modes. Other practitioners believe that it is worthwhile to use a number of different modes of representation to depict the analysis model. Different modes of representation force the software team to consider requirements from different viewpoints, an approach that has a higher probability of uncovering omissions, inconsistencies, and ambiguity.

The work product that evolves as a consequence of the application of an analysis method is the *analysis model.* The analysis model contains four generic elements:

1. **Scenario-based elements.** The system is described from the user's point of view, using a scenario-based approach. Scenario-based elements of the analysis model are often the first part of the analysis model that is developed. As such, they serve as input for the creation of other modeling elements.
2. **Class-based elements.** Each usage scenario implies a set of "objects" that are manipulated as an actor interacts with the system. These objects are categorized into classes—collections of things that have similar attributes and common behaviors.
3. **Behavioral elements.** The behavior of a computer-based system can have a profound effect on the design that is chosen and the implementation approach that is applied. Therefore, the analysis model must provide modeling elements that depict how software responds to specific events within its operating domain.
4. **Flow-oriented elements.** Information is transformed as it flows through a computer-based system. The system accepts input in a variety of forms, applies functions to transform it, and produces output in a variety of forms.

8.3. Design

M. A. Jackson [48] 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."

8.3.1. 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]:

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 that lead to high quality. An abstraction is self-contained. It generally implements one well-constrained data structure or algorithm and can be accessed using a simple interface. The details of its internal operation need not be known for it to be used effectively and 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 [49] 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 share 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 the 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,*—independent modules—is easier to develop because function may be compartmentalized and interfaces are simplified (consider the 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 [50], 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.

8.3.2. 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 that are illustrated in Figure 9. *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, whereas 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 to software components. Finally, *component-level design,* also called *procedural design,* is conducted as algorithms are created to implement the processing requirements of program components.

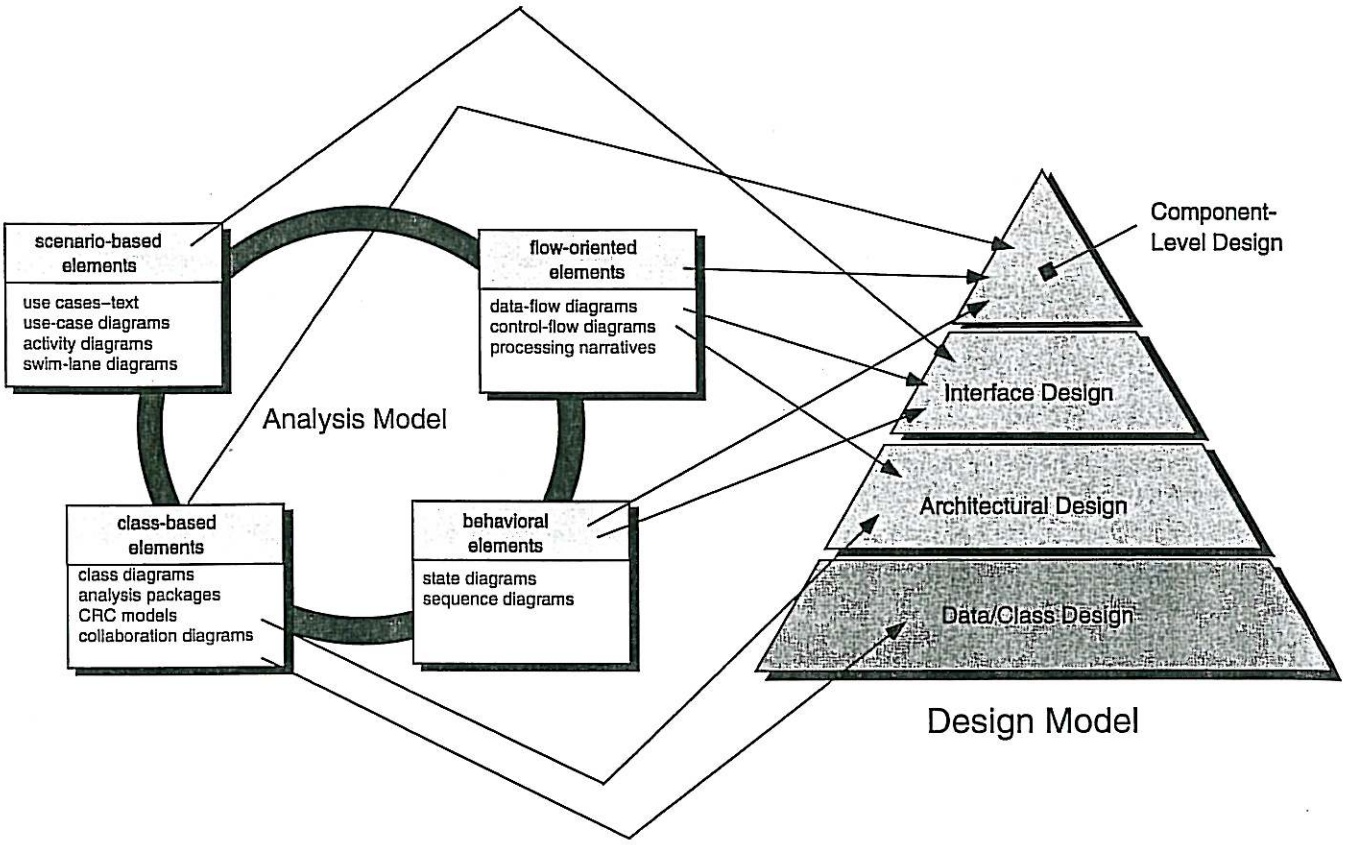

Figure 9 The design pyramid.

8.4. 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 often 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.

8.5. Software Testing

Glen Myers [51] states a number of 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.

The above 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 amount 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, and that per-

formance 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.

8.5.1. Strategy

A strategy for software testing integrates software test case design techniques [1] 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.

8.5.2. 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 amount of time and effort.

Over the past three 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 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* [52].

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; and that the integrity of external information (e.g., 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.

9. SOFTWARE ENGINEERING PATTERNS

The software process can be described as a collection of patterns that define a set of activities, actions, work tasks, work products, and/or related behaviors required to develop computer software. Stated in more general terms, a *software engineering pattern* provides us with a template—a consistent method for describing an element or characteristic of the software process or the product that is produced by the process. By combining patterns, a software team can construct a process (and software) that best meets the needs of the customer.

Patterns can be defined at any level of abstraction.[15] In some cases, a pattern might be used to describe a complete process (e.g., prototyping). In other situation, patterns can be used to describe an important framework activity (e.g., planning) or as task within a framework activity (e.g., project estimating), or some element of the software itself.

9.1. Process Patterns

Process patterns provide an effective mechanism for describing any software process [53]. The patterns enable a software engineering organization to develop a hierarchical process description that begins at a high level of abstraction. The description is then refined into a set of patterns that describe framework activities, and further refined in hierarchical fashion into more detailed task patterns for each activity. Once process patterns have been developed, they can be reused for the definition of process variants, that is, a customized process model can be defined by a software team using the patterns as building blocks for the process model.

9.2. Analysis Patterns

Anyone who has analyzed requirements for more than a few software projects begins to notice that certain things reoccur across all projects within a specific application domain.[16] These can be called "analysis patterns" [54] and represent something (e.g., a class, a function, a behavior) within the application domain that can be reused when modeling many applications.

Requirements engineering (and software engineering in general) sometimes reinvents the wheel. If application-specific analysis patterns are available, the software team can create the analysis model more quickly by using proven patterns.

Analysis patterns are integrated into the analysis model by reference to the pattern name. They are also stored in a repository so that requirements engineers can use search facilities to find and reuse them.

9.3. Design Patterns

Brad Appleton defines a design pattern as "a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns" [55]. Stated in another way, a design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine (1) whether the pattern is applicable to the current work, (2) whether the pattern can be reused (hence, saving design time), and (3) whether the pattern can serve as a guide for developing a similar but functionally or structurally different pattern.

The design patterns are:

Pattern name—describes the essence of the pattern in a short but expressive name

Intent—describes the pattern and what it does

Also-known-as—lists any synonyms for the pattern

Motivation—provides an example of the problem

Applicability—notes specific design situations in which the pattern is applicable

Structure—describes the classes that are required to implement the pattern

Participants—describes the responsibilities of the classes that are required to implement the pattern

Collaborations—describes how the participants collaborate to carry out their responsibilities

Consequences—describes the "design forces" that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented

Related patterns—cross-references related design patterns

9.4 Testing Patterns

Testing patterns are described in much the same way as analysis and design patterns. Dozens of testing patterns have been proposed in the literature (see, e.g., [56, 57]).

---

[15]Patterns are applicable to many software engineering activities including (but no limited to) analysis, design, coding, testing, quality assurance, and configuration management. In addition, patterns can be used to describe specific characteristics of software.

[16]In some cases, things reoccur regardless of the application domain. For example, the features and functions of user interfaces are similar regardless of the application domain under consideration.

10. THE ROAD AHEAD AND THE THREE Rs

Software is a child of the latter half of the 20th 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 from anything that baby boomers have yet experienced. Staff downsizing, the growing reality of international outsourcing, and the increasingly complex demands of customers who will not 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 [58].

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

10.1. 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 [59]. 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 component 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 21st century.

10.2. Reengineering

Almost every business relies on the day-to-day operation of an aging software plant. Major companies spend as much of 70% or more of their software budget on the care and feeding of legacy systems. Many of these systems were poorly designed more than two decades 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 resources with little hope of abatement. The software plant must be rebuilt and that demands a reengineering strategy.

Reengineering takes time, costs significant amounts of money, and absorbs resources that might be otherwise applied to 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.

10.3. Retooling

To achieve the first two Rs, 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.

11. SUMMARY

As each of us in the software business looks to the future, a small set of questions is asked and reasked. Will we continue to struggle to produce software that meets the needs of a new breed of customers? Will generation X and Y 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] Pressman, R. S., *Software Engineering: A Practitioner's Approach,* 6th ed., McGraw-Hill, 2005.

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

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

[4] Royce, W. W., "Managing the Development of Large Software Systems: Concepts and Techniques," in *Proceedings of WESCON,* August 1970.

[5] Hanna, M., "Farewell to Waterfalls," *Software Magazine,* May 1995, 38–46.

[6] Boehm, B., "A Spiral Model for Software Development and Enhancement," *Computer, 21,* 5, May 1988, 61–72.

[7] Boehm, B., "Using the WINWIN Spiral Model: A Case Study," *Computer, 31,* 7, July 1998, 33–44.

[8] Monin, F. and M. Hinchey, *Understanding Formal Methods,* Springer-Verlag, 2003.

[9] Casey, C., *A Programming Approach to Formal Methods,* McGraw-Hill, 2000.

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

[11] Dyer, M., *The Cleanroom Approach to Quality Software Development,* Wiley, 1992.

[12] Elrad, T., R. Filman, and A. Bader (Eds.), "Aspect Oriented Programming," *Communications of ACM, 44,* 10, October 2001, special issue.

[13] Gradecki, J. and N. Lesiecki, *Mastering AspectJ: Aspect-Oriented Programming in Java,* Wiley, 2003.

[14] Kiselev, I., *Aspect-Oriented Programming with AspectJ,* Sams Publishers, 2002.

[15] Jacobson, I., Booch, G., and J. Rumbaugh, *The Unified Software Development Process,* Addison-Wesley, 1999.

[16] Arlow, J. and I. Neustadt, *UML and the Unified Process,* Addison-Wesley, 2002.

[17] Beck, K., et al., "Manifesto for Agile Software development," http://www.agilemanifesto.org/.

[18] Fowler. M., "The New Methodology," June 2002, http://www.martinfowler.com/articles/newMethodology.html#N8B.

[19] Highsmith, J., *Adaptive Software Development: An Evolutionary Approach to Managing Complex Systems,* Dorset House Publishing, 1998.

[20] Stapleton, J., *DSDM—Dynamic System Development Method: The Method in Practice,* Addison-Wesley, 1997.

[21] Schwaber, K., and M. Beedle, *Agile Software Development with SCRUM,* Prentice-Hall, 2001.

[22] Cockburn, A., *Agile Software Development,* Addison-Wesley, 2002.

[23] Coad, P., Lefebvre, E., and J. DeLuca, *Java Modeling in Color with UML,* Prentice-Hall, 1999.

[24] Wells, D., "XP—Unit Tests," 1999, http://www.extremeprogramming.org/rules/unittests.html.

[25] Cougar, J and R. Zawacki, *Managing and Motivating Computer Personnel,* Wiley, 1980.

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

[27] Weinberg, G., *Understanding the Professional Programmer,* Dorset Hosue, 1988.

[28] Curtis, B., "People Management Maturity Model," in *Proceedings of International Conference on Software Engineering,* Pittsburgh, 1989.

[29] August, J. H., *Joint Application Design,* Prentice-Hall, 1991.

[30] Wood, J., and D. Silver, *Joint Application Design,* Wiley, 1989.

[31] *Capability Maturity Model Integration (CMMI),* Version 1.1, Software Engineering Institute, March 2002, available at: http://www.sei.cmu.edu/cmmi/.

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

[33] Jones, C., *Applied Software Measurement,* McGraw-Hill, 1991.

[34] Dreger, J. B., *Function Point Analysis,* Prentice-Hall, 1989.

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

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

[37] Lorenz, M. and J. Kidd, *Object-Oriented Software Metrics,* Prentice-Hall, 1994.

[38] Boehm, B., *Software Cost Estimation with COCOMO II,* Prentice-Hall, 2000.

[39] Charette, R. N., *Software Engineering Risk Analysis and Management,* McGraw-Hill/Intertext, 1989.

[40] Hall, E. M., *Managing Risk: Methods for Software Systems Development,* Addison-Wesley, 1998.

[41] Chapman, C. B. and S. Ward, *Project Risk Management: Processes, Techniques and Insights,* Wiley, 1997.

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

[43] Cavano, J. P. and J. A. McCall, "A Framework for the Measurement of Software Quality," in *Proceedings of ACM Software Quality Assurance Workshop,* November, 1978, pp. 133–139.

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

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

[46] Babich, W., *Software Configuration Management,* Addison-Wesley, 1986.

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

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

[49] Parnas, D. L., "On Criteria to be used in Decomposing Systems into Modules," *CACM, 14,* 1, April, 1972, 221–227.

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

[51] Myers, G., *The Art of Software Testing,* Wiley, 1979.

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

[53] Ambler, S., *Process Patterns: Building Large-Scale Systems using Object Technology,* Cambridge University Press/SIGS Books, 1998.

[54] Fowler, M., *Analysis Patterns: Reusable Object Models,* Addison-Wesley, 1997.

[55] Appleton, B., "Patterns and Software: Essential Concepts and Terminology," downloadable at: http://www.enteract.com/~bradapp/docs/patterns-intro.html.

[56] Binder, R., *Testing Object-Oriented Systems: Models, Patterns, and Tools,* Addison-Wesley, 1999.

[57] Marick, B., "Software Testing Patterns," 2002, http://www.testing.com/test-patterns/index.html.

[58] Pressman, R., "Software According to Nicollo Machiavelli," *IEEE Software,* January, 1995.

[59] Tracz, W., *Software Reuse: Emerging Technology,* IEEE Computer Society Press, 1988.