- Software subcontract management
- Software project tracking and oversight
- Software project planning
- Requirements management

Process maturity level 3

- Peer reviews
- Intergroup coordination
- Software product engineering
- Integrated software management
- Training program
- Organization process definition
- Organization process focus

Process maturity level 4

- Software quality management
- Quantitative process management

Process maturity level 5

- Process change management
- Technology change management
- Defect prevention

WebRef

A tabular version of the complete SEI-CMM, including all goals, commitments, abilities, and activities, is available at **sepo.nosc.mil/CMMmatrices.html**

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

2.3 SOFTWARE PROCESS MODELS

Quote:

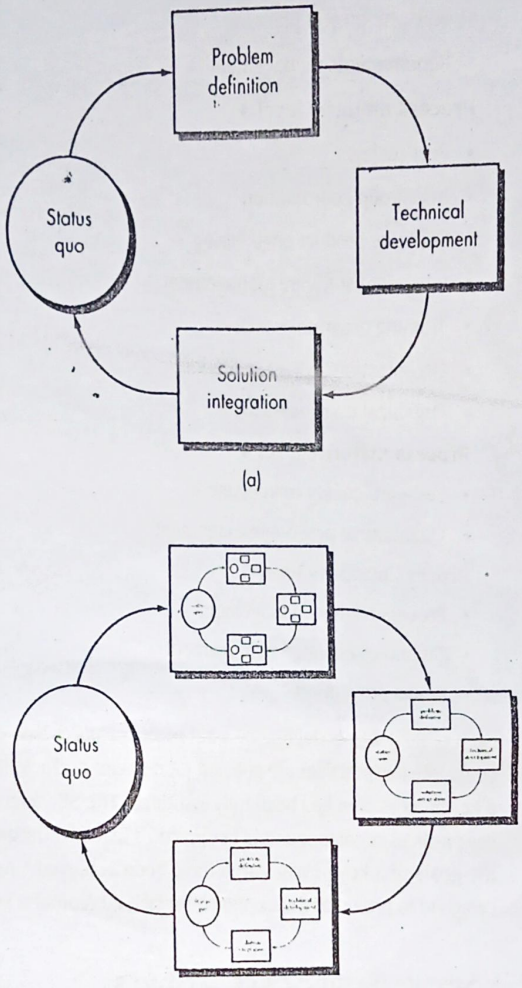
"Too often, software work follows the first law of bicycling: No matter where you're going, it's uphill and against the wind."
author unknown

To solve actual problems in an industry setting, a software engineer or a team of engineers must incorporate a development strategy that encompasses the process, methods, and tools layers described in Section 2.1.1 and the generic phases discussed in Section 2.1.2. This strategy is often referred to as a *process model* or a *software engineering paradigm*. A process model for software engineering is chosen based on the nature of the project and application, the methods and tools to be used, and the controls and deliverables that are required. In an intriguing paper on the nature of the software process, L. B. S. Raccoon [RAC95] uses fractals as the basis for a discussion of the true nature of the software process.

FIGURE 2.3

(a) The phases of a problem solving loop [RAC95]

(b) The phases within phases of the problem solving loop [RAC95]



All software development can be characterized as a problem solving loop (Figure 2.3a) in which four distinct stages are encountered: status quo, problem definition, technical development, and solution integration. Status quo "represents the current state of affairs" [RAC95]; problem definition identifies the specific problem to be solved; technical development solves the problem through the application of some technology, and solution integration delivers the results (e.g., documents, programs, data, new business function, new product) to those who requested the solution in the first place. The generic software engineering phases and steps defined in Section 2.1.2 easily map into these stages.

This problem solving loop applies to software engineering work at many different levels of resolution. It can be used at the macro level when the entire application is considered, at a mid-level when program components are being engineered, and

even at the line of code level. Therefore, a fractal[4] representation can be used to provide an idealized view of process. In Figure 2.3b, each stage in the problem solving loop contains an identical problem solving loop, which contains still another problem solving loop (this continues to some rational boundary; for software, a line of code).

Realistically, it is difficult to compartmentalize activities as neatly as Figure 2.3b implies because cross talk occurs within and across stages. Yet, this simplified view leads to a very important idea: regardless of the process model that is chosen for a software project, all of the stages—status quo, problem definition, technical development, and solution integration—coexist simultaneously at some level of detail. Given the recursive nature of Figure 2.3b, the four stages discussed apply equally to the analysis of a complete application and to the generation of a small segment of code.

KEY POINT

All stages of a software process—status quo, problem definition, technical development, and solution integration—coexist simultaneously at some level of detail.

Raccoon [RAC95] suggests a "Chaos model" that describes "software development [as] a continuum from the user to the developer to the technology." As work progresses toward a complete system, the stages are applied recursively to user needs and the developer's technical specification of the software.

In the sections that follow, a variety of different process models for software engineering are discussed. Each represents an attempt to bring order to an inherently chaotic activity. It is important to remember that each of the models has been characterized in a way that (ideally) assists in the control and coordination of a real software project. And yet, at their core, all of the models exhibit characteristics of the Chaos model.

2.4 THE LINEAR SEQUENTIAL MODEL

Sometimes called the *classic life cycle* or the *waterfall model*, the *linear sequential model* suggests a systematic, sequential approach[5] to software development that begins at the system level and progresses through analysis, design, coding, testing, and support. Figure 2.4 illustrates the linear sequential model for software engineering. Modeled after a conventional engineering cycle, the linear sequential model encompasses the following activities:

System/information engineering and modeling. Because software is always part of a larger system (or business), work begins by establishing requirements for all system elements and then allocating some subset of these requirements to software. This system view is essential when software must interact with other elements such as hardware, people, and databases. System engineering and analysis encompass requirements gathering at the system level with a small amount of top level

4 Fractals were originally proposed for geometric representations. A pattern is defined and then applied recursively at successively smaller scales; patterns fall inside patterns.

5 Although the original waterfall model proposed by Winston Royce [ROY70] made provision for "feedback loops," the vast majority of organizations that apply this process model treat it as if it were strictly linear.

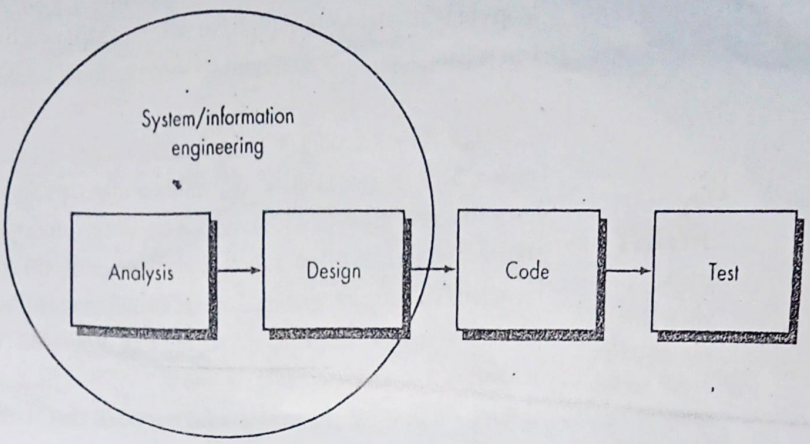


FIGURE 2.4 The linear sequential model

design and analysis. Information engineering encompasses requirements gathering at the strategic business level and at the business area level.

Software requirements analysis. The requirements gathering process is intensified and focused specifically on software. To understand the nature of the program(s) to be built, the software engineer ("analyst") must understand the information domain (described in Chapter 11) for the software, as well as required function, behavior, performance, and interface. Requirements for both the system and the software are documented and reviewed with the customer.

Design. Software design is actually a multistep process that focuses on four distinct attributes of a program: data structure, software architecture, interface representations, and procedural (algorithmic) detail. The design process translates requirements into a representation of the software that can be assessed for quality before coding begins. Like requirements, the design is documented and becomes part of the software configuration.

ADVICE

Although the linear model is often derided as "old fashioned," it remains a reasonable approach when requirements are well understood.

Code generation. The design must be translated into a machine-readable form. The code generation step performs this task. If design is performed in a detailed manner, code generation can be accomplished mechanistically.

Testing. Once code has been generated, program testing begins. The testing process focuses on the logical internals of the software, ensuring that all statements have been tested, and on the functional externals; that is, conducting tests to uncover errors and ensure that defined input will produce actual results that agree with required results.

Support. Software will undoubtedly undergo change after it is delivered to the customer (a possible exception is embedded software). Change will occur because errors have been encountered, because the software must be adapted to accommodate changes in its external environment (e.g., a change required because of a new operating system or peripheral device), or because the customer requires functional or performance enhancements. Software support/maintenance reapplies each of the preceding phases to an existing program rather than a new one.

The linear sequential model is the oldest and the most widely used paradigm for software engineering. However, criticism of the paradigm has caused even active supporters to question its efficacy [HAN95]. Among the problems that are sometimes encountered when the linear sequential model is applied are:

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

Why does the linear model sometimes fail?

In an interesting analysis of actual projects Bradac [BRA94], found that the linear nature of the classic life cycle leads to "blocking states" in which some project team members must wait for other members of the team to complete dependent tasks. In fact, the time spent waiting can exceed the time spent on productive work! The blocking state tends to be more prevalent at the beginning and end of a linear sequential process.

Each of these problems is real. However, the classic life cycle paradigm has a definite and important place in software engineering work. It provides a template into which methods for analysis, design, coding, testing, and support can be placed. The classic life cycle remains a widely used procedural model for software engineering. While it does have weaknesses, it is significantly better than a haphazard approach to software development.

2.5 THE PROTOTYPING MODEL

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

The prototyping paradigm (Figure 2.5) begins with requirements gathering. Developer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A "quick design" then occurs. The quick design focuses on a representation of those aspects of the software that will be visible to the customer/user (e.g., input approaches and output formats). The quick design leads to the construction of

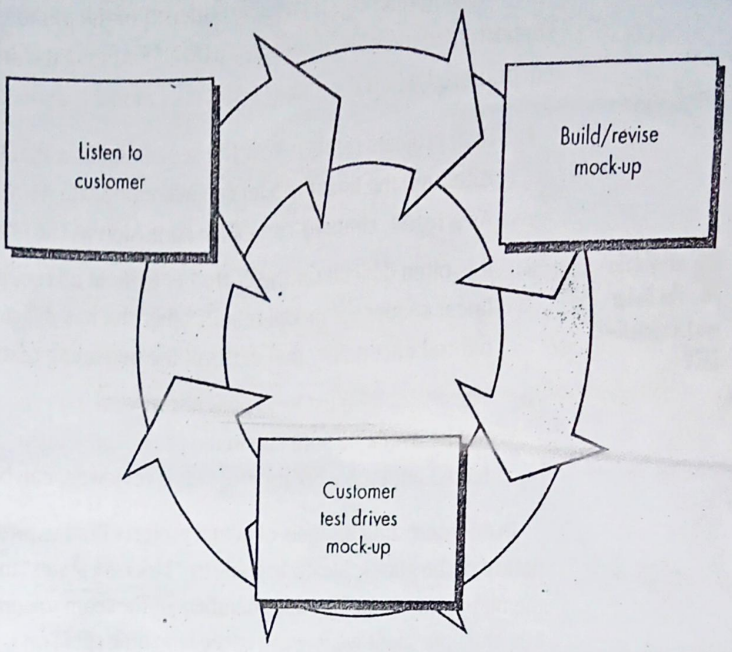


FIGURE 2.5
The prototyping paradigm

ADVICE

When your customer has a legitimate need but is clueless about the details, develop a prototype as a first step.

a prototype. The prototype is evaluated by the customer/user and used to refine requirements for the software to be developed. Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.

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

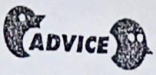
But what do we do with the prototype when it has served the purpose just described? Brooks [BRO75] provides an answer:

> In most projects, the first system built is barely usable. It may be too slow, too big, awkward in use or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved . . . When a new system concept or new technology is used, one has to build a system to throw away, for even the best planning is not so omniscient as to get it right the first time. The management question, therefore, is not whether to build a pilot system and throw it away. You will do that. The only question is whether to plan in advance to build a throwaway, or to promise to deliver the throwaway to customers . . .

The prototype can serve as "the first system." The one that Brooks recommends we throw away. But this may be an idealized view. It is true that 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," unaware that in the rush to get it working no one has considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high levels of quality can be maintained, the customer cries foul and demands that "a few fixes" be applied to make the prototype a working product. Too often, software development management relents.
2. The developer often makes implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, the developer may become familiar with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.



Resist pressure to extend a rough prototype into a production product. Quality almost always suffers as a result.

Although problems can occur, prototyping can be 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.

2.6 THE RAD MODEL

Rapid application development (RAD) is an incremental software development process model that emphasizes an extremely short development cycle. The RAD model is a "high-speed" adaptation of the linear sequential model in which rapid development is achieved by using component-based construction. If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a "fully functional system" within very short time periods (e.g., 60 to 90 days) [MAR91]. Used primarily for information systems applications, the RAD approach encompasses the following phases [KER94]:

Business modeling. The information flow among business functions is modeled in a way that answers the following questions: What information drives the business process? What information is generated? Who generates it? Where does the information go? Who processes it? Business modeling is described in more detail in Chapter 10.

Data modeling. The information flow defined as part of the business modeling phase is refined into a set of data objects that are needed to support the business. The char-

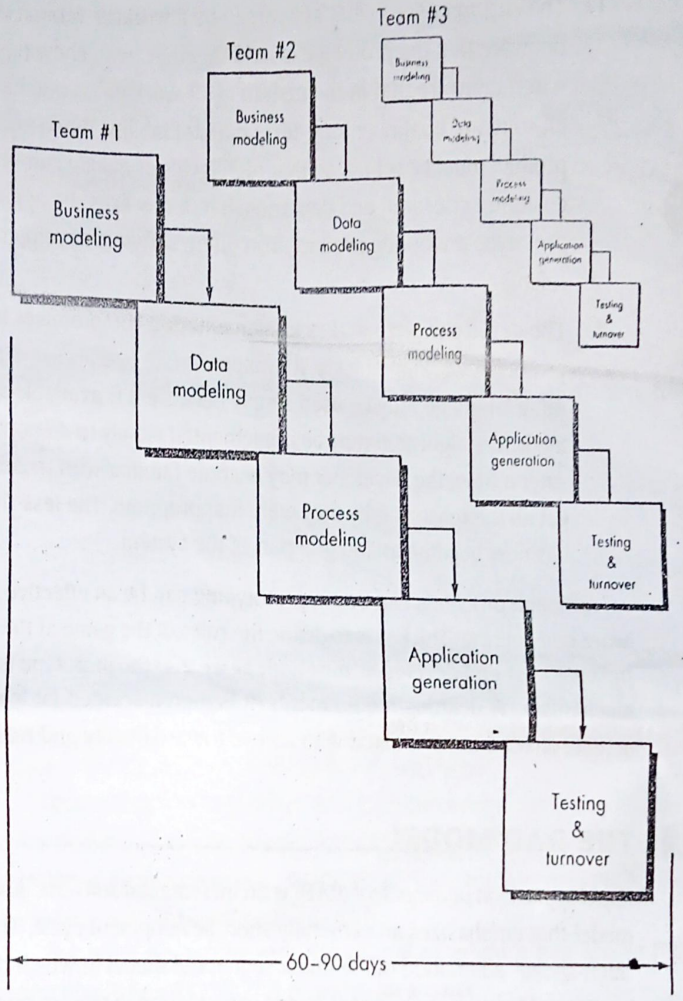


FIGURE 2.6 The RAD model

acteristics (called *attributes*) of each object are identified and the relationships between these objects defined. Data modeling is considered in Chapter 12.

Process modeling. The data objects defined in the data modeling phase are transformed to achieve the information flow necessary to implement a business function. Processing descriptions are created for adding, modifying, deleting, or retrieving a data object.

Application generation. RAD assumes the use of fourth generation techniques (Section 2.10). Rather than creating software using conventional third generation programming languages the RAD process works to reuse existing program components (when possible) or create reusable components (when necessary). In all cases, automated tools are used to facilitate construction of the software.

Testing and turnover. Since the RAD process emphasizes reuse, many of the program components have already been tested. This reduces overall testing time. However, new components must be tested and all interfaces must be fully exercised.

The RAD process model is illustrated in Figure 2.6. Obviously, the time constraints imposed on a RAD project demand "scalable scope" [KER94]. If a business application can be modularized in a way that enables each major function to be completed in less than three months (using the approach described previously), it is a candidate for RAD. Each major function can be addressed by a separate RAD team and then integrated to form a whole.

XRef

RAD makes heavy use of reusable components. For further information on component-based development, see Chapter 27.

Like all process models, the RAD approach has drawbacks [BUT94]:

- For large but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.
- RAD requires developers and customers who are committed to the rapid-fire activities necessary to get a system complete in a much abbreviated time frame. If commitment is lacking from either constituency, RAD projects will fail.
- Not all types of applications are appropriate for RAD. If a system cannot be properly modularized, building the components necessary for RAD will be problematic. If high performance is an issue and performance is to be achieved through tuning the interfaces to system components, the RAD approach may not work.
- RAD is not appropriate when technical risks are high. This occurs when a new application makes heavy use of new technology or when the new software requires a high degree of interoperability with existing computer programs.

2.7 EVOLUTIONARY SOFTWARE PROCESS MODELS

There is growing recognition that software, like all complex systems, evolves over a period of time [GIL88]. Business and product requirements often change as development proceeds, making a straight path to an end product unrealistic; tight market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to meet competitive or business pressure; a set of core product or system requirements is well understood, but the details of product or system extensions have yet to be 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.

The linear sequential model (Section 2.4) is designed for straight-line development. In essence, this waterfall approach assumes that a complete system will be delivered after the linear sequence is completed. The prototyping model (Section 2.5) is designed to assist the customer (or developer) in understanding requirements. In general, it is not designed to deliver a production system. The evolutionary nature of software is not considered in either of these classic software engineering paradigms.