Software

Software is (1) instructions (computer programs) that when executed provide desired function and performance, (2) data structures that enable the programs to adequately manipulate information, and (3) documents that describe the operation and use of the programs.

Software Characteristics

Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware:

1. Software is developed or engineered; it is not manufactured in the classical sense.
   Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.
2. Software doesn't "wear out."
   Figure 1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (ideally, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

Figure1: Failure curve for hardware

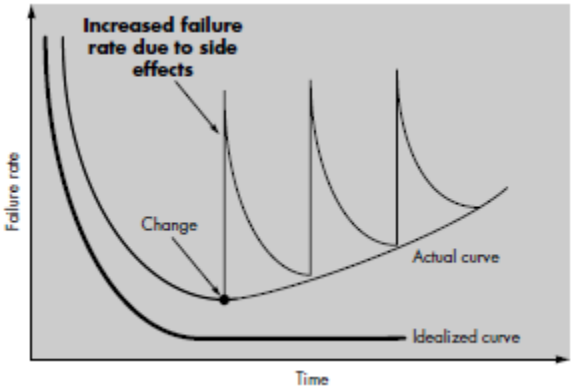


Figure2: Idealized and actual failure curves for software

   Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the "idealized curve" shown in Figure2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected (ideally, without introducing other errors) and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear—software doesn't wear out. But it does deteriorate.
3. Although the industry is moving toward component-based assembly, most software continues to be custom built.
   A software component should be designed and implemented so that it can be reused in many different programs. In the 1960s, scientific subroutine libraries that were built were reusable in a broad array of engineering and scientific applications. These subroutine libraries reused well-defined algorithms in an effective manner but had a limited domain of application. Modern reusable components encapsulate both data and the processing applied to the data, enabling the software engineer to create new applications from reusable parts. For example, today's graphical user interfaces are built using reusable components that enable the creation of graphics windows,

pull-down menus, and a wide variety of interaction mechanisms. The data structure and processing detail required to build the interface are contained with a library of reusable components for interface construction.

Types of Software

- **System software.** System software is a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) process complex, but determinate, information structures. Other systems applications (e.g., operating system components, drivers, telecommunicationsprocessors) process largely indeterminate data. In either case, the system software area is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.
- **Real-time software.** Software that monitors/analyzes/controls real-world events as they occur is called *real time.* Elements of real-time software include a data gathering component that collects and formats information from an external environment, an analysis component that transforms information as required by the application, a control/output component that responds to the external environment, and a monitoring component that coordinates all other components so that real-time response (typically ranging from 1 millisecond to 1 second) can be maintained.
- **Business software.** Business information processing is the largest single software application area. Discrete "systems" (e.g., payroll, accounts receivable/payable, inventory) have evolved into management information system (MIS) software that accesses one or more large databases containing business information. Applications in this area restructure existing data in a way that facilitates business operations or management decision making. In addition to conventional data processing application, business software applications also encompass interactive computing (e.g., point-of-sale transaction processing).
- **Engineering and scientific software.** Engineering and scientific software have been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from conventional
- numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.
- **Embedded software.** Intelligent products have become commonplace in nearly every consumer and industrial market. Embedded software resides in read-only memory and is used to control products and systems for the consumer and industrial markets. Embedded software can perform very limited and esoteric functions (e.g., keypad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).
- **Personal computer software.** The personal computer software market has burgeoned over the past two decades. Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications, external network, and database access are only a few of hundreds of applications.
- **Web-based software.** The Web pages retrieved by a browser are software that incorporates executable instructions (e.g., CGI, HTML, Perl, or Java), and data (e.g., hypertext and a variety of visual and audio formats). In essence, the network becomes a massive computer providing an almost unlimited software resource that can be accessed by anyone with a modem.
- **Artificial intelligence software.** Artificial intelligence (AI) software makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Expert systems, also called knowledge-based systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing are representative of applications within this category.

Software Myths

Management myths: Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality.

Myth: We already have a book that's full of standards and procedures for building software, won't that provide my people with everything they need to know?

Reality: The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it streamlined to improve time to delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is "no."

Myth: My people have state-of-the-art software development tools; after all, we buy them the newest computers.

Reality: It takes much more than the latest model mainframe, workstation, or PC to do high-quality software development. Computer-aided software engineering (CASE) tools are more important than hardware for achieving good quality and productivity, yet the majority of software developers still do not use them effectively.

Myth: If we get behind schedule, we can add more programmers and catch up

Reality: Software development is not a mechanistic process like manufacturing. In the words of Brooks: "adding people to a late software project makes it later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well-coordinated manner.

Myth: If I decide to outsource the software project to a third party, I can just relax and let that firm build it.

Reality*:* If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

Customer myths: A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and ultimately, dissatisfaction with the developer.

Myth: A general statement of objectives is sufficient to begin writing programs— we can fill in the details later.

Reality: A poor up-front definition is the major cause of failed software efforts. A formal and detailed description of the information domain, function, behavior, performance, interfaces, design constraints, and validation criteria is essential. These characteristics can be determined only after thorough communication between customer and developer.

Myth: Project requirements continually change, but change can be easily accommodated because software is flexible.

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. If serious attention is given to up-front definition, early requests for change can be accommodated easily. The customer can review requirements and recommend modifications with relatively little impact on cost. When changes are requested during software design, the cost impact grows rapidly. Resources have been committed and a design framework has been established. Change can cause upheaval that requires additional resources and major design modification, that is, additional cost. Changes in function, performance, interface, or other characteristics during implementation (code and test) have a severe impact on cost.

Practitioner's myths: Myths that are still believed by software practitioners have been fostered by 50 years of programming culture.

Myth: Once we write the program and get it to work, our job is done.

Reality: Someone once said that "the sooner you begin 'writing code', the longer it'll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.
Myth: Until I get the program "running" I have no way of assessing its quality.
Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the *formal technical review.* Software reviews are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.
Myth: The only deliverable work product for a successful project is the working program.
Reality: A working program is only one part of a *software configuration* that includes many elements. Documentation provides a foundation for successful engineering and, more important, guidance for software support.
Myth: Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.
Reality: Software engineering is not about creating documents. It is about creating quality. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

Software Engineering
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.

OR

Software engineering is defined as the systematic approach to the development, operation, maintenance, and retirement of software

OR

The IEEE has developed a more comprehensive definition when it states:
Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

The Software Engineering Approach
The basic approach of software engineering is to separate the process for developing software from the developed product (i.e., the software). The premise is that to a large degree the software process determines the quality of the product and productivity achieved. Hence to tackle the problem domain and successfully face the challenges that software engineering faces, one must focus on the software process.

Phased Development Process
A development process consists of various phases, each phase ending with a defined output. The phases are performed in an order specified by the process model being followed. The main reason for having a phased process is that it breaks the problem of developing software into successfully performing a set of phases, each handling a different concern of software development. This ensures that the cost of development is lower than what it would have been if the whole problem was tackled together. Furthermore, a phased process allows proper checking for quality and progress at some defined points during the development (end of phases). Without this, one would have to wait until the end to see what software has been produced. Clearly, this will not work for large systems. Hence, for managing the complexity, project tracking, and quality, all the development processes consist of a set of phases. A phased development process is central to the software engineering approach for solving the software crisis.
Various process models have been proposed for developing software. In general, however, we can say that any problem solving in software must consist of requirement specification for understanding and clearly stating the problem, design for deciding a plan for a solution, coding for implementing the planned solution, and testing for verifying the programs.

Requirements Analysis
Requirements analysis is done in order to understand the problem the software system is to solve. The emphasis in requirements analysis is on identifying what is needed from the system, not how the system

will achieve its goals. For complex systems, even determining what is needed is a difficult task. The goal of the requirements activity is to document the requirements in a *software requirements specification* document.

There are two major activities in this phase: problem understanding or analysis and requirement specification. In problem analysis, the aim is to understand the problem and its context, and the requirements of the new system that is to be developed.

Once the problem is analyzed and the essentials understood, the requirements must be specified in the requirement specification document. The requirements document must specify all functional and performance requirements; the formats of inputs and outputs; and all design constraints that exist due to political, economic, environmental, and security reasons.

Software Design

The purpose of the design phase is to plan a solution of the problem specified by the requirements document. This phase is the first step in moving from the problem domain to the solution domain.

The design activity often results in three separate *outputs: architecture design, high level design,* and *detailed design. Architecture* focuses on looking at a system as a combination of many different components, and how they interact with each other to produce the desired results. The *high level design* identifies the modules that should be built for developing the system and the specifications of these modules. At the end of system design all the major data structures, file formats, output formats, etc., are also fixed. In *detailed design,* the internal logic of each of the modules is specified.

In architecture the focus is on identifying components or subsystems and how they connect; in high level design the focus is on identifying the modules; and during detailed design the focus is on designing the logic for each of the modules.

Coding

The goal of the coding phase is to translate the design of the system into code in a given programming language. For a given design, the aim in this phase is to implement the design in the best possible manner.

The coding phase affects both testing and maintenance profoundly. Well written code can reduce the testing and maintenance effort. Because the testing and maintenance costs of software are much higher than the coding cost, the goal of coding should be to reduce the testing and maintenance effort.

Testing

Testing is the major quality control measure used during software development. Its basic function is to detect defects in the software. After coding, computer programs are available that can be executed for testing purposes. This implies that testing not only has to uncover errors introduced during coding, but also errors introduced during the previous phases. Thus, the goal of testing is to uncover requirement, design, and coding errors in the programs.

The starting point of testing is *unit testing,* where the different modules or components are tested individually. As modules are integrated into the system, *integration testing* is performed, which focuses on testing the interconnection between modules. After the system is put together, *system testing* is performed. Here the system is tested against the system requirements to see if all the requirements are met and if the system performs as specified by the requirements. Finally, *acceptance testing* is performed to demonstrate to the client, on the real-life data of the client, the operation of the system.

Software Development Life Cycle Models

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.

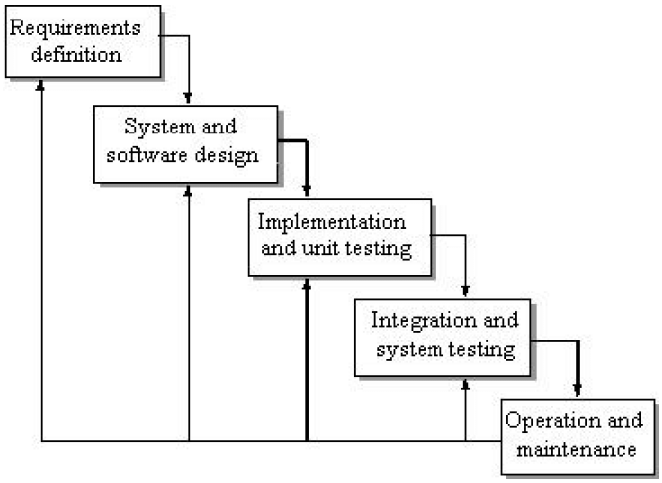
The goal of Software Engineering is to provide models and processes that lead to the production of well-documented maintainable software in a manner that is predictable.

“The period of time that starts when a software product is conceived and ends when the product is no longer available for use. The software life cycle typically includes a requirement phase, design phase, implementation phase, test phase, installation and check out phase, operation and maintenance phase, and sometimes retirement phase”.

Build & Fix Model

- Product is constructed without specifications or any attempt at design
- Adhoc approach and not well defined
- Simple two phase model
- Suitable for small programming exercises of 100 or 200 lines
- Unsatisfactory for software for any reasonable size
- Code soon becomes unfixable & unenhanceable
- No room for structured design
- Maintenance is practically not possible

Waterfall Model



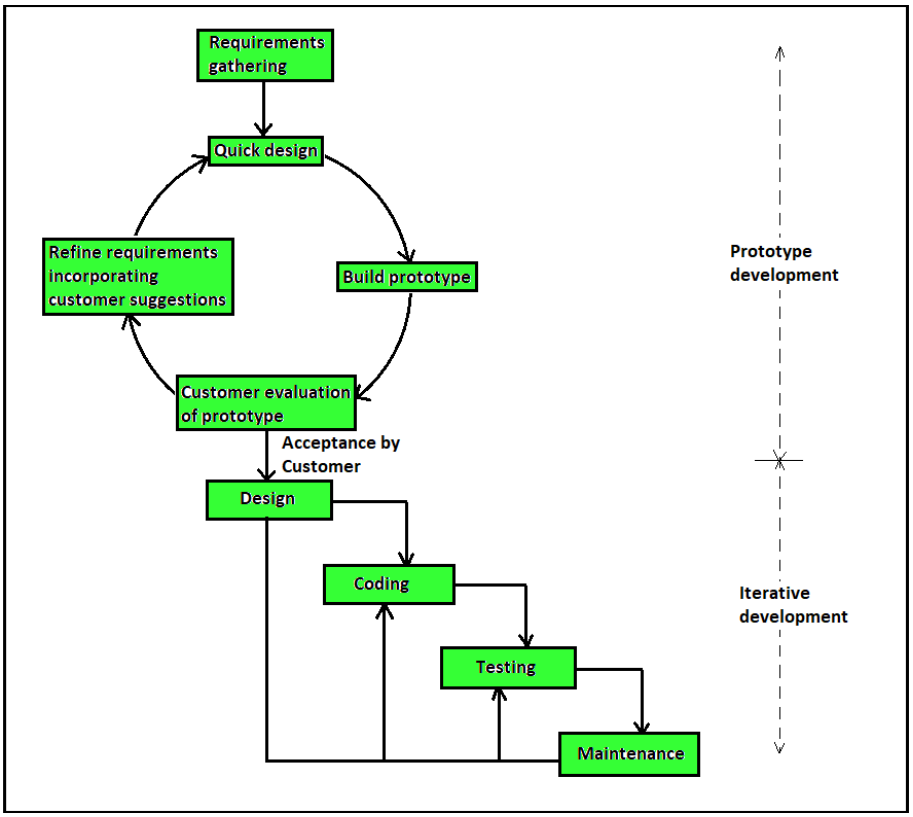
The simplest process model is the *waterfall model*, which states that the phases are organized in a linear order. The model was originally proposed by Royce, though variations of the model have evolved depending on the nature of activities and the flow of control between them. In this model, a project begins with feasibility analysis. Upon successfully demonstrating the feasibility of a project, the requirements analysis and project planning begins. The design starts after the requirements analysis is complete, and coding begins after the design is complete. Once the programming is completed, the code is integrated and testing is done. Upon successful completion of testing, the system is installed. After this, the regular operation and maintenance of the system takes place.

One of the main advantages of this model is its simplicity. It is conceptually straightforward and divides the large task of building a software system into a series of cleanly divided phases, each phase dealing with a separate logical concern. It is also easy to administer in a contractual setup—as each phase is completed and its work product produced, some amount of money is given by the customer to the developing organization. The waterfall model, although widely used, has some strong limitations. Some of the key limitations are:

1. It assumes that the requirements of a system can be frozen (i.e., baselined) before the design begins. This is possible for systems designed to automate an existing manual system. But for new systems, determining the requirements is difficult as the user does not even know the requirements. Hence, having unchanging requirements is unrealistic for such projects.
2. Freezing the requirements usually requires choosing the hardware (because it forms a part of the requirements specification). A large project might take a few years to complete. If the hardware is selected early, then due to the speed at which hardware technology is changing, it is likely that the final software will use a hardware technology on the verge of becoming obsolete. This is clearly not desirable for such expensive software systems.

3. It follows the "big bang" approach—the entire software is delivered in one shot at the end. This entails heavy risks, as the user does not know until the very end what they are getting. Furthermore, if the project runs out of money in the middle, then there will be no software. That is, it has the "all or nothing" value proposition.
4. It is a document-driven process that requires formal documents at the end of each phase.
Despite these limitations, the waterfall model has been the most widely used process model. It is well suited for routine types of projects where the requirements are well understood. That is, if the developing organization is quite familiar with the problem domain and the requirements for the software are quite clear, the waterfall model works well.

Prototyping Model



The goal of a prototyping-based development process is to counter the first two limitations of the waterfall model. The basic idea here is that instead of freezing the requirements before any design or coding can proceed, a throwaway prototype is built to help understand the requirements. This prototype is developed based on the currently known requirements. Development of the prototype obviously undergoes design, coding, and testing, but each of these phases is not done very formally or thoroughly. By using this prototype, the client can get an actual feel of the system; because the interactions with the prototype can enable the client to, better understand the requirements of the desired system. This results in more stable requirements that change less frequently.
Prototyping is an attractive idea for complicated and large systems for which there is no manual process or existing system to help determine the requirements. In such situations, letting the client "play" with the prototype provides invaluable and intangible inputs that help determine the requirements for the system. It is also an effective method of demonstrating the feasibility of a certain approach. This might be needed for novel systems, where it is not clear those constraints can be met or that algorithms can be developed to

implement the requirements. In both situations, the risks associated with the projects are being reduced through the use of prototyping.

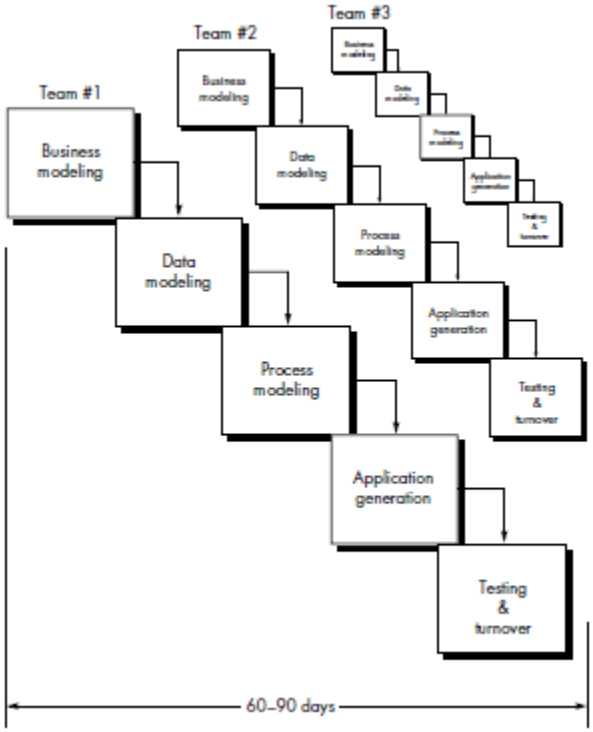
Prototyping is well suited for projects where requirements are hard to determine and the confidence in the stated requirements is low. In such projects, a waterfall model will have to freeze the requirements in order for the development to continue, even when the requirements are not stable. This leads to requirement changes and associated rework while the development is going on. Requirements frozen after experience with the prototype are likely to be more stable. Overall, in projects where requirements are not properly understood in the beginning, using the prototyping process model can be the most effective method for developing the software. It is an excellent technique for reducing some types of risks associated with a project.

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.

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). Used primarily for information systems applications, the RAD approach encompasses the following phases:

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?

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 characteristics (called *attributes*) of each object are identified and the relationships between these objects defined.

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. 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.

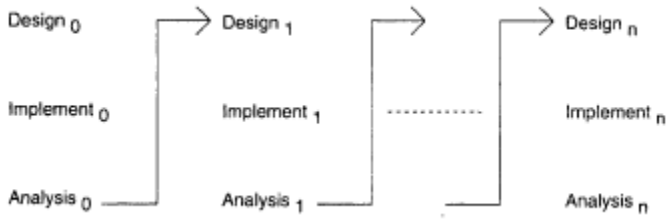
Testing and turnover. Since the RAD process emphasizes reuse, many of the program components have already been tested. This reduces overall testing time. However, new components must be tested and all interfaces must be fully exercised.

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.

Like all process models, the RAD approach has drawbacks:

- 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.

The Iterative Enhancement Model



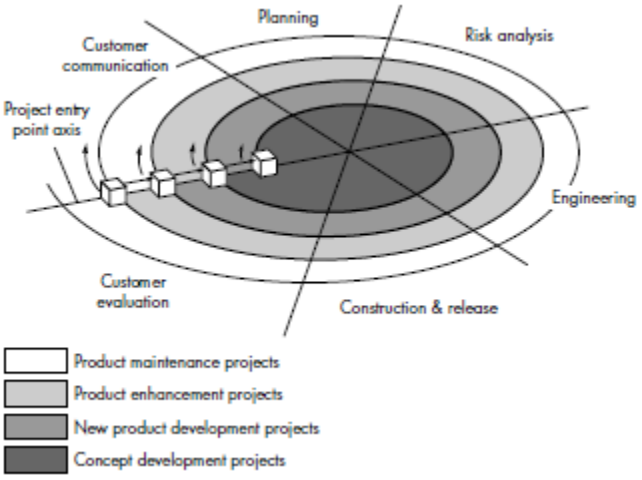
The iterative development process model counters the third limitation of the waterfall model and tries to combine the benefits of both prototyping and the waterfall model. The basic idea is that the software should be developed in increments, each increment adding some functional capability to the system until the full system is implemented. At each step, extensions and design modifications can be made. An advantage of this approach is that it can result in better testing because testing each increment is likely to be easier than testing the entire system as in the waterfall model. Furthermore, as in prototyping, the

increments provide feedback to the client that is useful for determining the final requirements of the system.

The iterative enhancement model is an example of this approach. In the first step of this model, a simple initial implementation is done for a subset of the overall problem. This subset is one that contains some of the key aspects of the problem that are easy to understand and implement and which form a useful and usable system. A *project control list* is created that contains, in order, all the tasks that must be performed to obtain the final implementation. This project control list gives an idea of how far along the project is at any given step from the final system. Each step consists of removing the next task from the list, designing the implementation for the selected task, coding and testing the implementation, performing an analysis of the partial system obtained after this step, and updating the list as a result of the analysis. These three phases are called *the design phase, implementation phase,* and *analysis phase.* The process is iterated until the project control list is empty, at which time the final implementation of the system will be available.

The Spiral Model

The *spiral model,* originally proposed by Boehm, is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the linear sequential model. It provides the potential for rapid development of incremental versions of the software. Using the spiral model, software is developed in a series of incremental releases. During early iterations, the incremental 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 number of framework activities, also called *task regions.* Typically, there are between three and six task regions. Figure depicts a spiral model that contains six task regions:



- **Customer communication**—tasks required to establish effective communication between developer and customer.
- **Planning**—tasks required to define resources, timelines, and other project related information.
- **Risk analysis**—tasks required to assess both technical and management risks.
- **Engineering**—tasks required to build one or more representations of the application.
- **Construction and release**—tasks required to construct, test, install, and provide user support (e.g., documentation and training).
- **Customer evaluation**—tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering stage and implemented during the installation stage.

Each of the regions is populated by a set of work tasks, called a *task set,* that are adapted to the characteristics of the project to be undertaken. For small projects, the number of work tasks and their

formality is low. For larger, more critical projects, each task region contains more work tasks that are defined to achieve a higher level of formality.

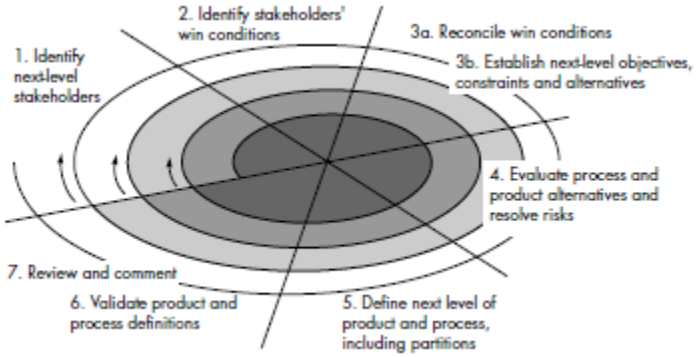
As this evolutionary process begins, the software engineering team moves around the spiral in a clockwise direction, beginning at the center. The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a 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 customer evaluation. In addition, the project manager adjusts he planned number of iterations required to complete the software.

The spiral model can be adapted to apply throughout the life of the computer software. An alternative view of the spiral model can be considered by examining the *project entry point axis*, also shown in figure. Each cube placed along the axis can be used to represent the starting point for different types of projects. A "concept development project" starts at the core of the spiral and will continue (multiple iterations occur along the spiral path that bounds the central shaded region) until concept development is complete. If the concept is to be developed into an actual product, the process proceeds through the next cube (new product development project entry point) and a "new development project" is initiated. The new product will evolve through a number of iterations around the spiral, following the path that bounds the region that has somewhat lighter shading than the core. In essence, the spiral, when characterized in this way, remains operative until the software is retired.

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.

The WINWIN Spiral Model



The spiral model suggests a framework activity that addresses customer communication. The objective of this activity is to elicit project requirements from the customer. In an ideal context, the developer simply asks the customer what is required and the customer provides sufficient detail to proceed. Unfortunately, this rarely happens. In reality, the customer and the developer enter into a process of negotiation, where the customer may be asked to balance functionality, performance, and other product or system characteristics against cost and time to market.

The best negotiations strive for a "win-win" result. That is, the customer wins by getting the system or product that satisfies the majority of the customer's needs and the developer wins by working to realistic and achievable budgets and deadlines.
Boehm's WINWIN spiral model defines a set of negotiation activities at the beginning of each pass around the spiral. Rather than a single customer communication activity, the following activities are defined:

1. Identification of the system or subsystem's key "stakeholders."
2. Determination of the stakeholders' "win conditions."
3. Negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned (including the software project team). Successful completion of these initial steps achieves a win-win result, which becomes the key criterion for proceeding to software and system definition.

In addition to the emphasis placed on early negotiation, the WINWIN spiral model introduces three process milestones, called *anchor points* that help establish the completion of one cycle around the spiral and provide decision milestones before the software project proceeds.
In essence, the anchor points represent three different views of progress as the project traverses the spiral. The first anchor point, *life cycle objectives* (LCO), defines a set of objectives for each major software engineering activity. For example, as part of LCO, a set of objectives establishes the definition of top-level system/product requirements. The second anchor point, *life cycle architecture* (LCA), establishes objectives that must be met as the system and software architecture is defined. *Initial operational capability* (IOC) is the third anchor point and represents a set of objectives associated with the preparation of the software for installation/distribution, site preparation prior to installation, and assistance required by all parties that will use or support the software.

Fourth Generation Techniques
The term *fourth generation techniques* (4GT) encompasses a broad array of software tools that have one thing in common: each enables the software engineer to specify some characteristic of software at a high level. The tool then automatically generates source code based on the developer's specification. The 4GT paradigm for software engineering focuses on the ability to specify software using specialized language forms or a graphic notation that describes the problem to be solved in terms that the customer can understand.
Currently, a software development environment that supports the 4GT paradigm includes some or all of the following tools: nonprocedural languages for database query, report generation, data manipulation, screen interaction and definition, code generation; high-level graphics capability; spreadsheet capability, and automated generation of HTML and similar languages used for Web-site creation using advanced software tools. Initially, many of the tools noted previously were available only for very specific application domains, but today 4GT environments have been extended to address most software application categories.
Like other paradigms, 4GT begins with a requirements gathering step. Ideally, the customer would describe requirements and these would be directly translated into an operational prototype. But this is unworkable. The customer may be unsure of what is required, may be ambiguous in specifying facts that are known, and may be unable or unwilling to specify information in a manner that a 4GT tool can consume.
For this reason, the customer/developer dialog described for other process models remains an essential part of the 4GT approach. For small applications, it may be possible to move directly from the requirements gathering step to implementation using a nonprocedural fourth generation language (4GL) or a model composed of a network of graphical icons. However, for larger efforts, it is necessary to develop a design strategy for the system, even if a 4GL is to be used. The use of 4GT without design (for large projects) will cause the same difficulties (poor quality, poor maintainability, poor customer acceptance) that have been encountered when developing software using conventional approaches.

Implementation using a 4GL enables the software developer to represent desired results in a manner that leads to automatic generation of code to create those results. Obviously, a data structure with relevant information must exist and be readily accessible by the 4GL.

To transform a 4GT implementation into a product, the developer must conduct thorough testing, develop meaningful documentation, and perform all other solution integration activities that are required in other software engineering paradigms. In addition, the 4GT developed software must be built in a manner that enables maintenance to be performed expeditiously.