SOFTWARE ENGINEERING & PROCESS MODELS

Software & Software Engineering
Software is instructions (computer programs) that when executed provide desired function and performance. Some of the characteristics of Software are:

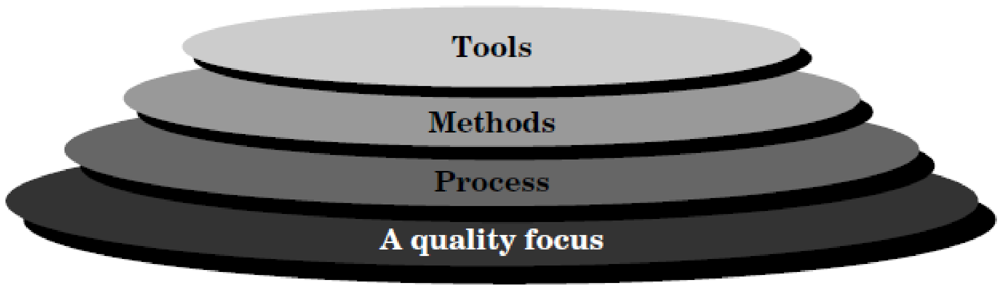
a) Software is developed or engineered; it is not manufactured in the classical sense.
b) Software doesn't "wear out."
c) Although the industry is moving toward component-based assembly, most software continues to be custom built.

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. 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 (KPAs). 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 analysis, design, program construction, testing, and support. 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 semi-automated 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, 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 analogous to CAD/CAE (computer-aided design/engineering) for hardware.
The phases and related steps in the generic view of software engineering are complemented by a number of umbrella activities. Typical activities in this category include:

• Software project tracking and control
• Formal technical reviews
• Software quality assurance
• Software configuration management
• Document preparation and production
• Reusability management
• Measurement
• Risk management

Umbrella activities are applied throughout the software process.

FIGURE
Software engineering layers



Software Process

A software process can be characterized as shown below. A common process framework is established by defining a small number of framework activities that are applicable to all software projects, regardless of their size or complexity. A number of task sets—each a collection of software engineering work tasks, project milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities—such as software quality assurance, software configuration management, and measurement—overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

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

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

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

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

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

Level 5: Optimizing: Continuous process improvement is enabled by quantitative feedback from the process and from testing innovative ideas and technologies. This level includes all characteristics defined for level 4. The five levels defined by the SEI were derived as a consequence of evaluating responses to the SEI assessment questionnaire that is based on the CMM. The results of the questionnaire are distilled to a single numerical grade that provides an indication of an organization's process maturity.

The SEI has associated key process areas (KPAs) with each of the maturity levels. The KPAs describe those software engineering functions (e.g., software project planning, requirements management) that must be present to satisfy good practice at a particular level. 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.

Feasibility Analysis

A major but optional activity within systems analysis is feasibility analysis. A wise person once said, "All things are possible, but not all things are profitable." Simply stated, this quote addresses feasibility. Systems analysts are often called upon to assist with feasibility analysis for proposed systems development projects. Therefore, let's take a brief look at this topic.

Consider your answer to the following questions. Can you ride a bicycle? Can you drive a car? Can you repair a car's transmission? Can you make lasagna? Can you snow ski? Can you earn an "A" in this course? Can you walk on the moon? As you considered your response to each of these questions, you quickly did some kind of feasibility analysis in your mind. Maybe your feasibility analysis and responses went something like this: Can you ride a bicycle? "Of course I can! I just went mountain bike riding last weekend with my best friend." Can you drive a car? "Naturally. I drove to school today and gasoline is sure expensive." Can you repair a car's transmission? "Are you kidding? I don't even know what a transmission is!" Can you make lasagna? "I never have, but with a recipe and directions I'm sure that I could. My mom makes the best lasagna, yum!" Can you snow ski? "I tried it once and hated it. It was so cold and it cost a lot of money." Can you earn an "A" in this course? "I think it would be easier to walk on the moon." Can you walk on the moon? "People have done it. With training, I think I could, and I would like to also."

Each of us does hundreds or thousands of feasibility analyses every day. Every time we think words like **"can I...?"** we are assessing our feasibility to do something beneficial the development or enhancement of an information system would be to the business. Feasibility analysis is the process by which feasibility is measured. It is an ongoing process done frequently during systems development projects in order to achieve a creeping commitment from the user and to continually assess the current status of the project. A creeping commitment is one that continues over time to reinforce the user's commitment and ownership of the information system being developed. Knowing a project's current status at strategic points in time gives us and the user the opportunity to (1) continue the project as planned, (2) make changes to the project, or (3) cancel the project.

Feasibility Types

Information systems development projects are subjected to at least three interrelated feasibility types:

- operational feasibility
- technical feasibility
- economic feasibility

Operational feasibility is the measure of how well particular information systems will work in a given environment. Just because XYZ Corporation's payroll clerks all have PCs that can display and allow editing of payroll data doesn't necessarily mean that ABC Corporation's payroll clerks can do the same thing. Part of the feasibility analysis study would be to assess the current capability of ABC Corporation's payroll clerks in order to determine the next best transition for them. Depending on the current situation, it might take one or more interim upgrades prior to them actually getting the PCs for display and editing of payroll data. Historically, of the three types of feasibility, operational feasibility is the one that is most often overlooked, minimized, or assumed to be okay. For example, several years ago many supermarkets installed "talking" point-of-sale terminals only to discover that customers did not like having people all around them hearing the names of the products they were purchasing. Nor did the cashiers like to hear all of those talking point-of-sale terminals because they were very distracting. Now the point-of-sale terminals are once again mute.

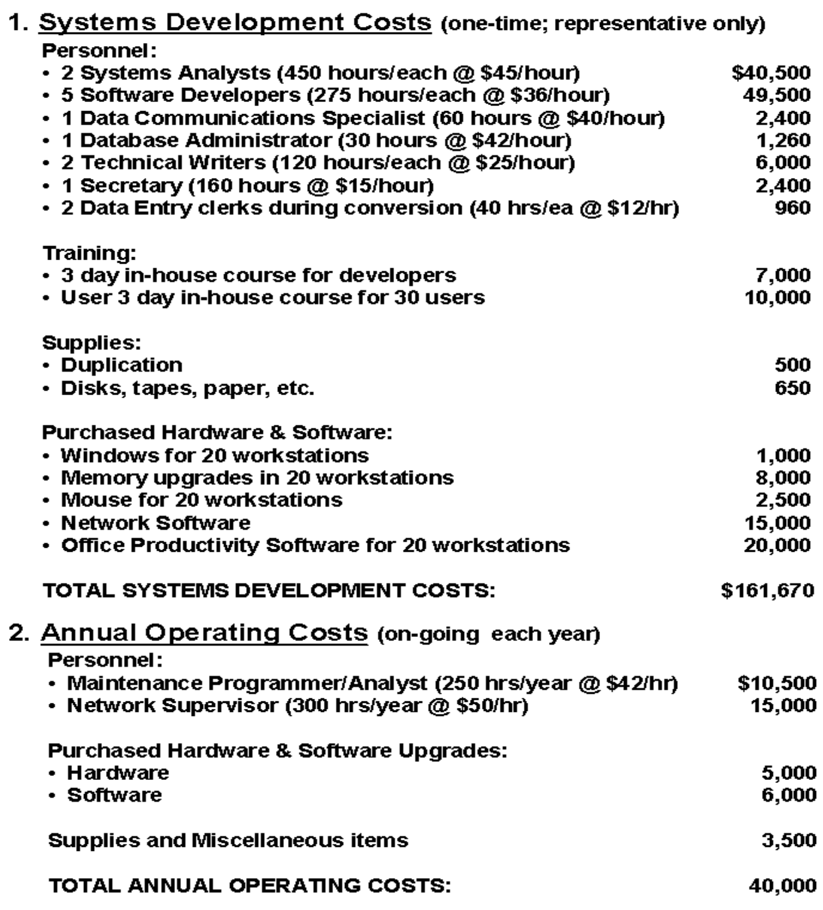
Technical feasibility is the measure of the practicality of a specific technical information system solution and the availability of technical resources. Often new technologies are solutions looking for a problem to solve. As voice recognition systems become more sophisticated, many businesses will consider this technology as a possible solution for certain information systems applications. When CASE technology was first introduced in the mid-1980s, many businesses decided it was impractical for them to adopt it for a variety of reasons, among them being the limited availability of the technical expertise in the marketplace to use it. Adoption of Smalltalk, C++, and other object-oriented programming for business applications is slow for similar reasons.

Economic feasibility is the measure of the cost-effectiveness of an information system solution. Without a doubt, this measure is most often the most important one of the three. Information systems are often viewed as capital investments for the business, and, as such, should be subjected to the same type of investment analyses as other capital investments. Financial analyses such as return on investment (ROI), internal rate of return (IRR), cost/benefit, payback period, and the time value of money are utilized when considering information system development projects.

Cost/benefit analysis identifies the costs of developing the information system and operating it over a specified period of time. It also identifies the benefits in financial terms in order to compare them with the costs. Economically speaking, when the benefits exceed the costs, the system has economic value to the business; just how much value is a function of management's perspective on investments.

Systems development and annual operating costs are the two primary components used to determine the cost estimates for a proposed information system. These two components are similar to the costs associated with constructing and operating a new building on the university campus. The building has a one-time construction cost—usually quite high. For example, a new library addition on campus recently costs $20 million to build. Once ready for occupancy and use, the library addition will incur operating costs, such as electricity, custodial care, maintenance, and library staff. The operating costs per year are probably a fraction of the construction costs. However, the operating costs continue for the life of the library addition and will more than likely exceed the construction costs at some time in the future.

Systems development costs are a one-time cost similar to the construction cost of the library addition. The annual operating costs are an ongoing cost once the information system is implemented. Figure below illustrates an example of these two types of costs. In this example, the annual operating costs are a very small fraction of the development costs. If the system is projected to have a useful life of ten years, the operational costs will still be significantly **more** than the development costs.



1. Systems Development Costs (one-time; representative only)	
Personnel:	
• 2 Systems Analysts (450 hours/each @ $45/hour)	$40,500
• 5 Software Developers (275 hours/each @ $36/hour)	49,500
• 1 Data Communications Specialist (60 hours @ $40/hour)	2,400
• 1 Database Administrator (30 hours @ $42/hour)	1,260
• 2 Technical Writers (120 hours/each @ $25/hour)	6,000
• 1 Secretary (160 hours @ $15/hour)	2,400
• 2 Data Entry clerks during conversion (40 hrs/ea @ $12/hr)	960
Training:	
• 3 day in-house course for developers	7,000
• User 3 day in-house course for 30 users	10,000
Supplies:	
• Duplication	500
• Disks, tapes, paper, etc.	650
Purchased Hardware & Software:	
• Windows for 20 workstations	1,000
• Memory upgrades in 20 workstations	8,000
• Mouse for 20 workstations	2,500
• Network Software	15,000
• Office Productivity Software for 20 workstations	20,000
TOTAL SYSTEMS DEVELOPMENT COSTS:	$161,670
2. Annual Operating Costs (on-going each year)	
Personnel:	
• Maintenance Programmer/Analyst (250 hrs/year @ $42/hr)	$10,500
• Network Supervisor (300 hrs/year @ $50/hr)	15,000
Purchased Hardware & Software Upgrades:	
• Hardware	5,000
• Software	6,000
Supplies and Miscellaneous items	3,500
TOTAL ANNUAL OPERATING COSTS:	40,000

Figure 2.1 Systems Development and Annual Operating Costs

Two types of benefits are usually identified and quantified—tangible and intangible. Tangible benefits are those that can objectively be quantified in terms of dollars. Figure below lists several tangible benefits. Intangible benefits are those that cannot be objectively quantified in terms of dollars. These benefits must be subjectively quantified in terms of dollars. A list of several intangible benefits is shown in Figure below.

Tangible Benefits	**Intangible Benefits**
Fewer Processing errors	Improved customer goodwill
Increased throughput	Improved employee morale
Decreased response time	Improved employee job satisfaction
Elimination of job steps	Better service to the community
Reduced expenses	Better decision making
Increased sales	
Faster turnaround	
Better credit	
Reduced credit losses	
Reduction of receivables	

Comparing the benefit dollars to the cost dollars, one can tell if the proposed information system is going to break even, cost the business, or save the business money. Once a project is started, financial analyses should continue to be done at periodic intervals to determine if the information system still makes economic sense. Sometimes systems development projects are canceled before they become operational, many because they no longer make economic sense to the business. Operational and technical feasibility should also be continually assessed during the life of a systems development project in order to make adjustments when necessary.

Software Development Life Cycle (SDLC)

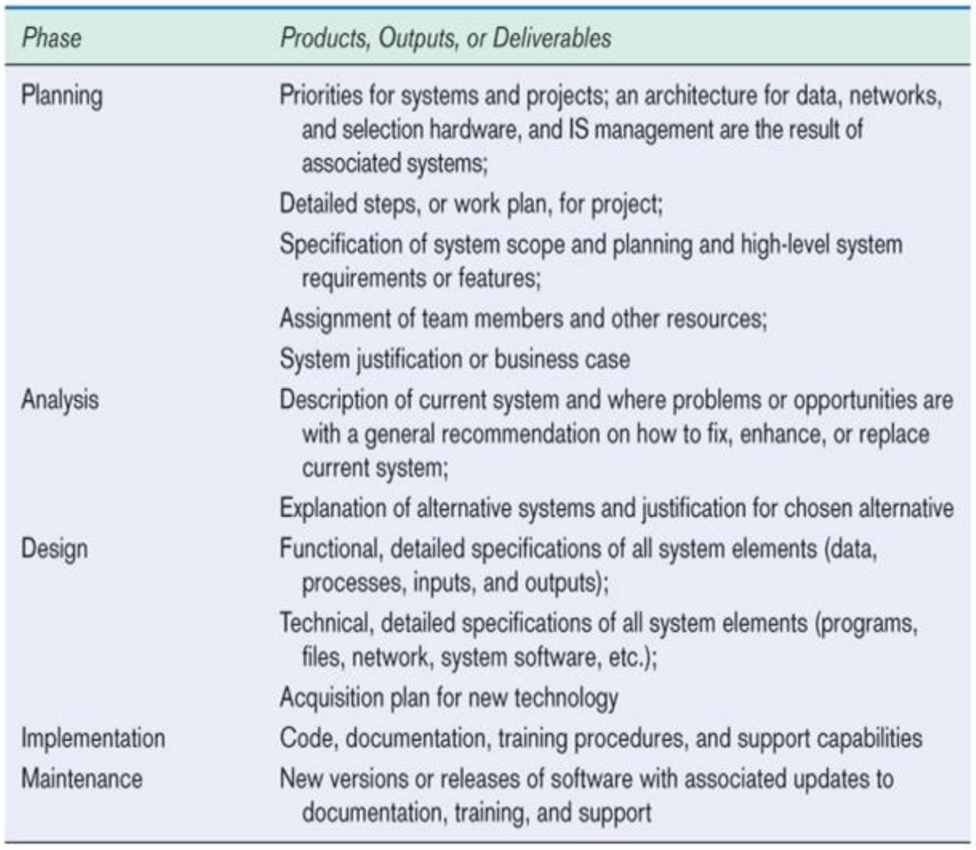
SDLC refers to a methodology for developing systems. It provides a consistent framework of tasks and deliverables needed to develop systems. The SDLC methodology may be condensed to include only those activities appropriate for a particular project, whether the system is automated or manual, whether it is a new system, or an enhancement to existing systems. The SDLC methodology tracks a project from an idea developed by the user, through a feasibility study, systems analysis and design, programming, pilot testing, implementation, and post-implementation analysis. Documentation developed during the project development is used in the future when the system is reassessed for its continuation, modification, or deletion.

Phases in SDLC are Planning, Analysis, Design, Implementation, and Maintenance/Sustainment/Staging

1) Project planning, feasibility study: Establishes a high-level view of the intended project and determines its goals.
2) Systems analysis, requirements definition: Refines project goals into defined functions and operation of the intended application. Analyzes end-user information needs.

3) Systems design: Describes desired features and operations in detail, including screen layouts, business rules, process diagrams, pseudo code and other documentation.
4) Implementation (Development): The real code is written here.
5) Integration and testing: Brings all the pieces together into a special testing environment, then checks for errors, bugs and interoperability.
6) Acceptance, installation, deployment: The final stage of initial development, where the software is put into production and runs actual business.
7) Maintenance: This phase involves activities which happens during the rest of the software's life: changes, correction, additions, moves to a different computing platform and more.

Table Products of SDLC Phases



Phase	Products, Outputs, or Deliverables
Planning	Priorities for systems and projects; an architecture for data, networks, and selection hardware, and IS management are the result of associated systems;
	Detailed steps, or work plan, for project;
	Specification of system scope and planning and high-level system requirements or features;
	Assignment of team members and other resources;
	System justification or business case
Analysis	Description of current system and where problems or opportunities are with a general recommendation on how to fix, enhance, or replace current system;
	Explanation of alternative systems and justification for chosen alternative
Design	Functional, detailed specifications of all system elements (data, processes, inputs, and outputs);
	Technical, detailed specifications of all system elements (programs, files, network, system software, etc.);
	Acquisition plan for new technology
Implementation	Code, documentation, training procedures, and support capabilities
Maintenance	New versions or releases of software with associated updates to documentation, training, and support

Software Process Model

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 earlier in the generic phases. 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.

A variety of different process models for software engineering will be discussed below. 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.

Linear Sequential Model

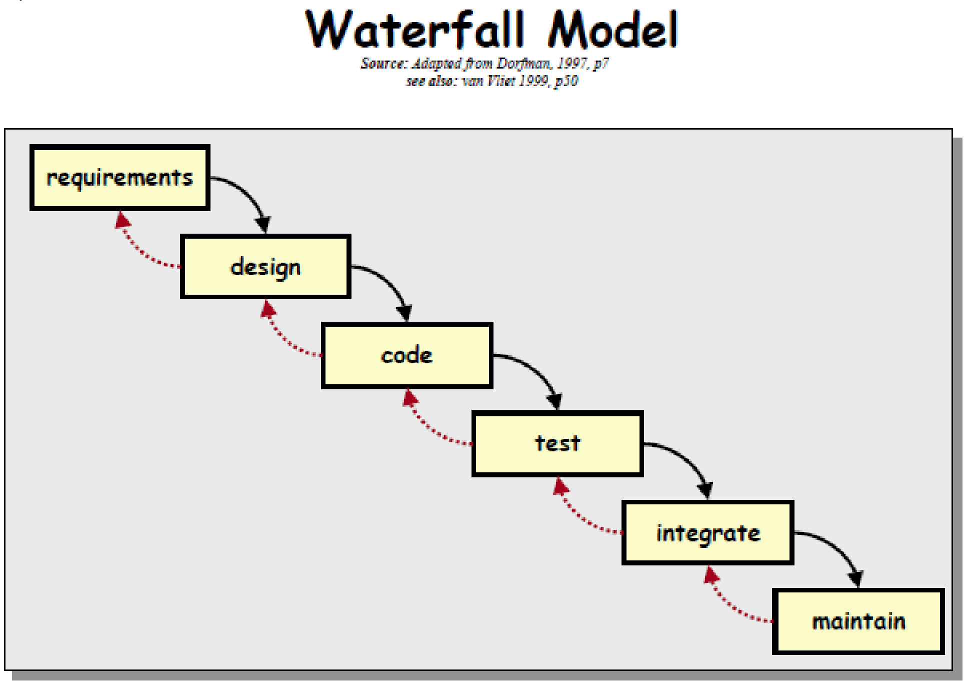
Sometimes called the **classic life cycle** or the **waterfall model**, the **linear sequential model** suggests a systematic, sequential approach to software development that begins at the system level and progresses through analysis, design, coding, testing, and support. Figure below illustrates the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

1) <u>System/information engineering and modeling</u>: 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 design and analysis. Information engineering encompasses requirements gathering at the strategic business level and at the business area level.
2) <u>Software requirements analysis</u>: 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 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.
3) <u>Design</u>: 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.
4) <u>Code generation</u>: 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.
5) <u>Testing</u>: 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.
6) <u>Support/Maintenance</u>: 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. Some of the problems that are sometimes encountered when the linear sequential model is applied are:

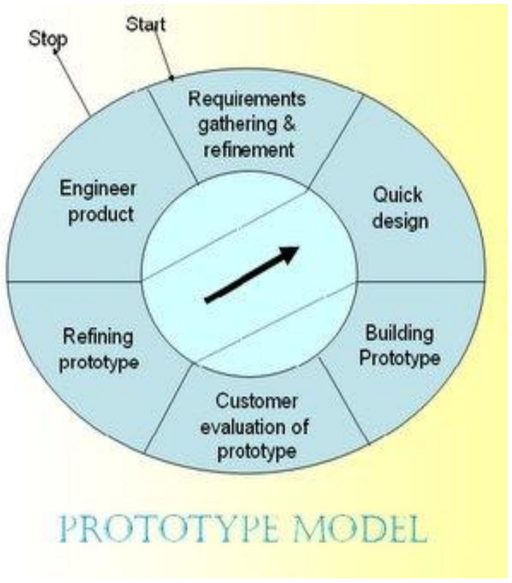
a) Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.

b) 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.
c) 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.


Waterfall Model
Source: Adapted from Dorfman, 1997, p7
see also: van Vliet 1999, p50
requirements
design
code
test
integrate
maintain

On analyzing actual projects it has been 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.
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.

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 below) 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 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.



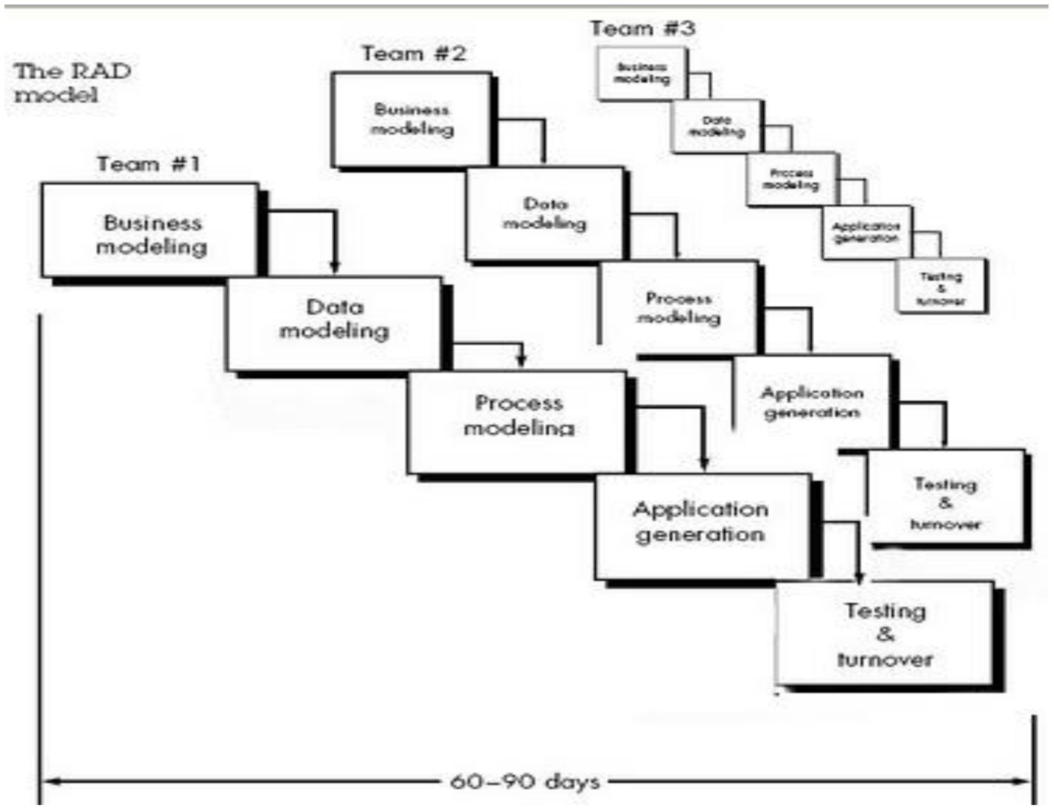
Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to use existing program fragments or applies tools (e.g., report generators, window managers) that enable working programs to be generated quickly.
But what do we do with the prototype when it has served the purpose just described? The prototype can serve as "the first system." There are several advantages to the Prototype model:
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:

a) 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.
b) 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.

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.

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:

1) <u>Business modeling:</u> 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?
2) <u>Data modeling:</u> 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.
3) <u>Process modeling:</u> 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.
4) <u>Application generation:</u> 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.
5) <u>Testing and turnover:</u> 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 time constraints imposed on a RAD project demand "scalable scope". 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:

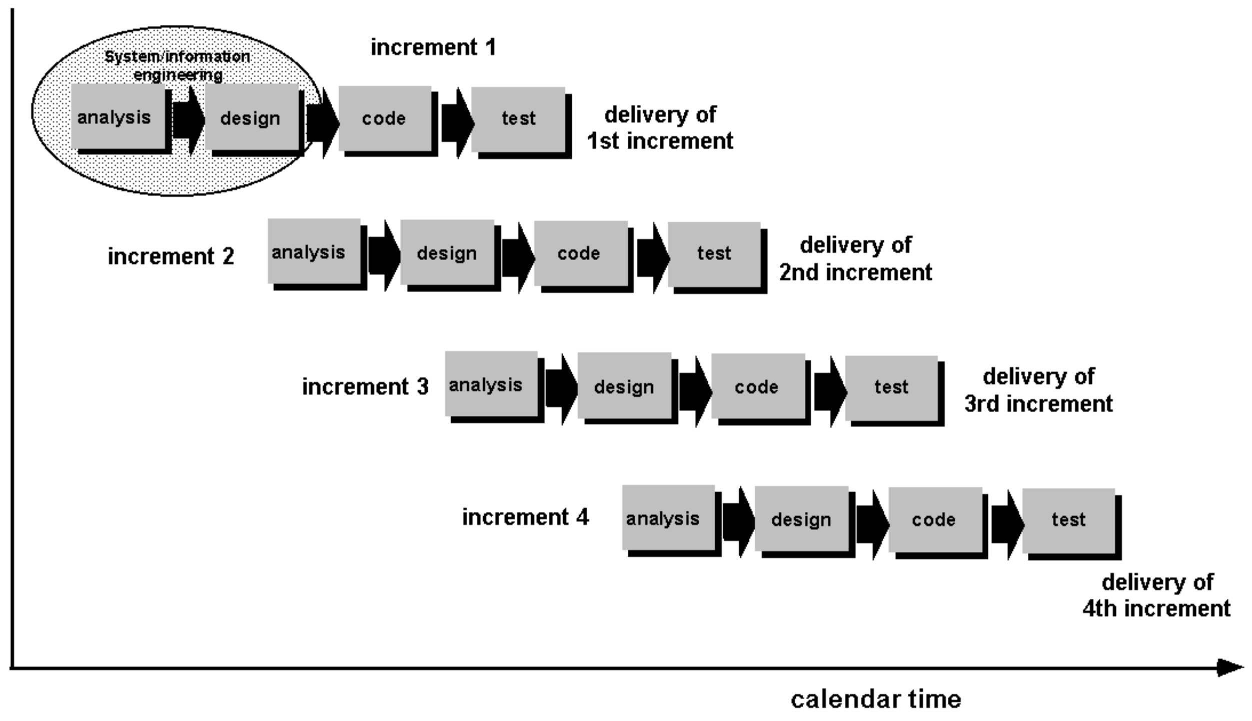
a. For large but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.
b. RAD requires developers and customers who are committed to the rapid-fire activities necessary to get a system complete in a much abbreviated time frame. If commitment is lacking from either constituency, RAD projects will fail.
c. 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.
d. 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.

Evolutionary Process Models

There is growing recognition that software, like all complex systems, evolves over a period of time. Business and product requirements often change as development proceeds, making a straight path to an end product is 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.
Waterfall model is designed for straight-line development. In essence, this approach assumes that a complete system will be delivered after the linear sequence is completed. The prototyping model 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. RAD makes heavy use of reusable components. Evolutionary models are iterative. They are characterized in a manner that enables software engineers to develop increasingly more complete versions of the software.

The Incremental Model

The incremental model combines elements of the linear sequential model (applied repetitively) with the iterative philosophy of prototyping. The incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces a deliverable “increment” 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 can incorporate the prototyping paradigm.
When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed, but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed review). As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.

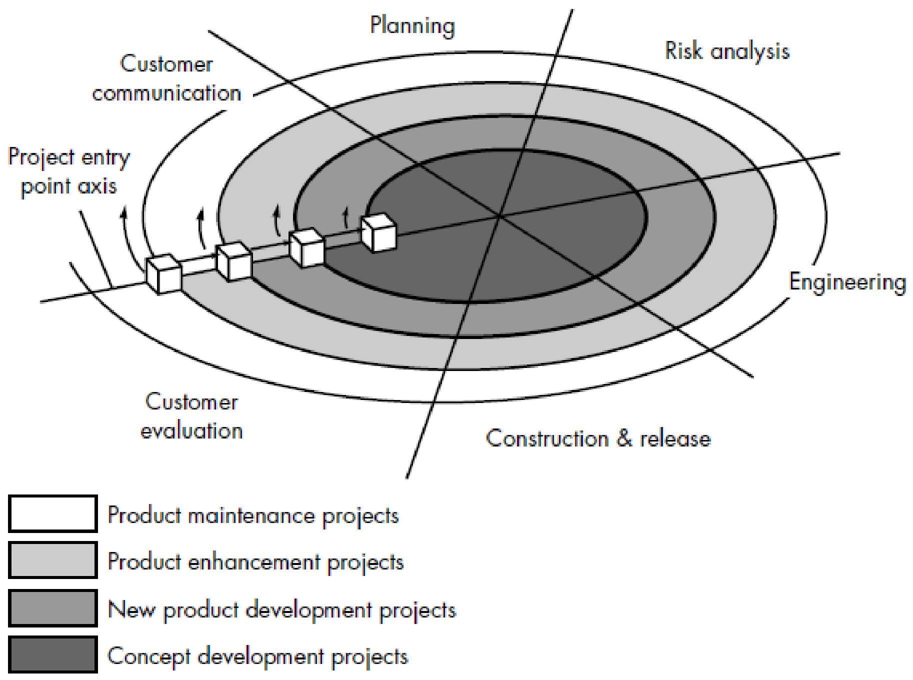


The incremental process model, like prototyping and other evolutionary approaches, is iterative in nature. But unlike prototyping, the incremental model focuses on the delivery of an operational product with each increment. Early increments are stripped down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.
Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment. In addition, increments can be planned to manage technical risks. For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain. It might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end-users without inordinate delay.

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 below 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.

FIGURE
A typical spiral model



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. In all cases, the umbrella activities (e.g., software configuration management and software quality assurance) are applied.
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 the planned number of iterations required to complete the software.
Unlike classical process models that end when software is delivered, 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 above 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. There are times when the process is dormant, but whenever a change is initiated, the process starts at the appropriate entry point (e.g., product enhancement).

The spiral model is a realistic approach to the development of large-scale systems and software. As 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.

a) It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable.
b) 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.
c) Finally, the model has not been used as widely as the linear sequential or prototyping paradigms. It will take a number of years before efficacy of this important paradigm can be determined with absolute certainty.

V-shaped SDLC Model

This model is a variant of the Waterfall model that emphasizes the verification and validation of the product. Here, testing of the product is planned in parallel with a corresponding phase of development. The stages (in parallel) in the V-shaped model are as follows:

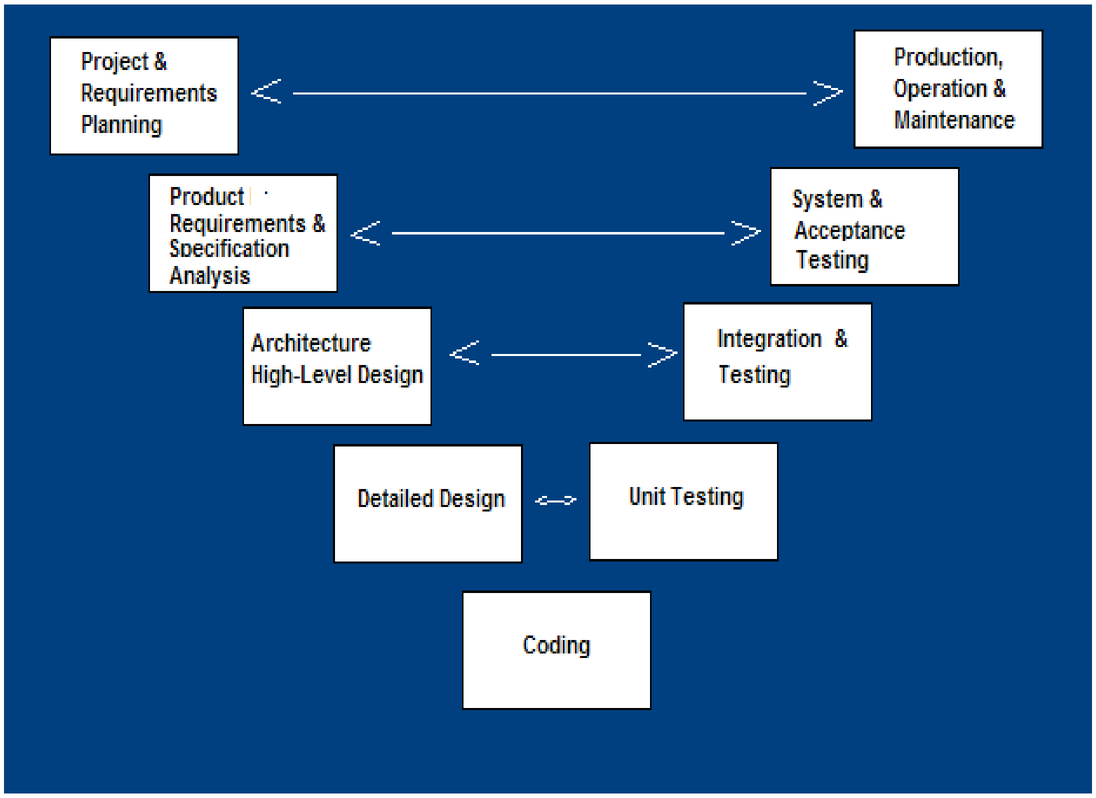
Project and Requirements Planning – allocate resources	Production, operation and maintenance – provide for enhancement and corrections
Product Requirements and Specification Analysis – complete specification of the software system	System and acceptance testing – check the entire software system in its environment
Architecture or High-Level Design – defines how software functions fulfill the design	Integration and Testing – check that modules interconnect correctly
Detailed Design – develop algorithms for each architectural component	Unit testing – check that each module acts as expected
Coding – transform algorithms into software	

Strengths of the V-shaped model are:

a) Emphasize planning for verification and validation of the product in early stages of product development
b) Each deliverable must be testable
c) Project management can track progress by milestones
d) Easy to use

The V-shaped model also has a few weaknesses, these are:

a) Does not easily handle concurrent events
b) Does not handle iterations or phases
c) Does not easily handle dynamic changes in requirements
d) Does not contain risk analysis activities


Project & Requirements Planning
Production, Operation & Maintenance
Product Requirements & Specification Analysis
System & Acceptance Testing
Architecture High-Level Design
Integration & Testing
Detailed Design
Unit Testing
Coding

The V-shaped model is an excellent choice during the following situations:

a) When systems require high reliability – e.g. hospital patient control applications
b) When all requirements are known up-front
c) When it can be modified to handle changing requirements beyond analysis phase
d) When Solution and technology are known
