UNIT - I

INTRODUCTION TO SOFTWARE ENGINEERING

The invention of one technology can have very great and unexpected effects on other technology is called the **law of unintended consequences**.

The term **software engineering** is composed of two words, software and engineering.

Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be a collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product.**

Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods.

Computer Software is the single most important technology. As software's importance has grown, the software community has attempted to develop technologies that will make it easier, faster & less expensive to build & maintain high quality computer programs.
The framework encompasses a process, set of methods and an array of tools called as software engineering.

Software engineering is an engineering discipline which is concerned with all aspects of software production, from early stages of system specification to maintaining the system after its use.

Engineering discipline- Engineers make things work by applying theories, methods & tools selectively & try to discover solutions for problems with organizational & financial constraints.

All aspects of the software production – software engineering is concerned with technical process of development & also activities like software project management, development of tools, methods & theories to support software production.
Software engineers use a systematic & organized approach in most effective way to produce high quality software.

Computer science is concerned with theory and fundamentals. Software engineering is concerned with practicalities of developing & delivering useful software.
System engineering is concerned with all aspects of development, including hardware, policy & process design and system development as well as software engineering. Software engineering is part of this process.

IEEE defines software engineering as:

The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software. that is, the application of engineering to software.

We can alternatively view it as a systematic collection of past experience. The experience is arranged in the form of methodologies and guidelines. A small program can be written without using software engineering principles. But if one wants to develop a large software product, then software engineering principles are absolutely necessary to achieve a good quality software cost effectively.

Without using software engineering principles it would be difficult to develop large programs. In industry it is usually needed to develop large programs to accommodate multiple functions. A problem with developing such large commercial programs is that the complexity and difficulty levels of the programs increase exponentially with their sizes. Software engineering helps to reduce this programming complexity.

Software engineering principles use two important techniques to reduce problem complexity: ***abstraction*** and ***decomposition***. The principle of abstraction suggests that a problem can be simplified by neglect irrelevant details. In other words, the main purpose of abstraction is to consider only those aspects of the problem that are relevant for certain purpose and suppress other aspects that are not relevant for the given purpose. Once the simpler problem is solved, then the neglected details can be taken into consideration to solve the next lower level abstraction, and so on. Abstraction is a powerful way of reducing the complexity of the problem. The other approach to try problem complexity is decomposition. In this technique, a complex problem is divided into several smaller problems and then the smaller problems are solved one by one. However, in this technique any random decomposition of a problem into smaller parts will not help. The problem has to be decomposed such that each component of the decomposed problem can be solved independently and then the solution of the different components can be combined to get the full solution. A good decomposition of a problem should minimize interactions among various components. If the different subcomponents are interrelated, then the different components cannot be solved separately and the desired reduction in complexity will not be achieved.

EVOLVING ROLE OF SOFTWARE:

Software is both a product and a vehicle for delivering a product.

As a product, it delivers the computing potential embodied by computer hardware. Software may be in cellular phone or operates inside a mainframe computer, it is an information transformer-producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation.

As the vehicle for delivering the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments).

Software delivers the most important product of our time-information. It transforms personal data (e.g.an individual's financial transactions) so that the data can be more useful in a local context, it manages business information to enhance competitiveness, it provides a gateway to worldwide information networks (e.g., the Internet) and provides the means for acquiring information in all of its forms.

The role of computer software has undergone significant change over a span of little more than 50 years.

Dramatic improvements in hardware performance, very great changes in computing architectures, vast increases in memory and storage capacity, wide variety of input and output options, made more sophisticated and complex computer-based systems.

A huge software industry has become a dominant factor in the economies of the industrialized world. Teams of software specialists, each focusing on one part of the technology required to deliver a complex application. the questions that were asked when modem computer based systems are built.

Why does it take so long to get software finished?
Why are development costs so high?
Why can't we find all errors before we give the software to our customers?
Why do we spend so much time and effort maintaining existing programs?
Why do we continue to have difficulty in measuring progress as software is
being developed and maintained?

SOFTWARE:

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

CHARACTERISTICS OF SOFTWARE:

It is important to examine the characteristics of software that make it different from other things that human beings build. Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware.

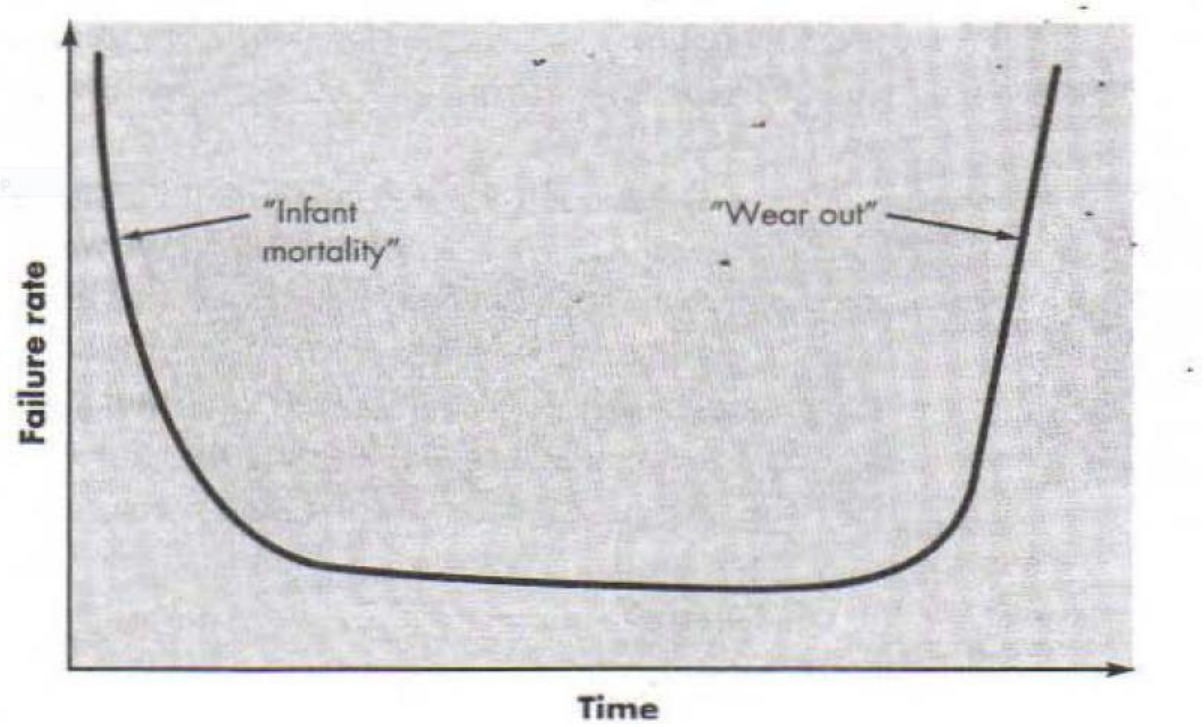
1. Software is developed or engineered; it is not manufactured in the classical sense.

Although some similarities exist between software development and hardware manufacturing, 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. Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different. Both activities require the construction of a product, but the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

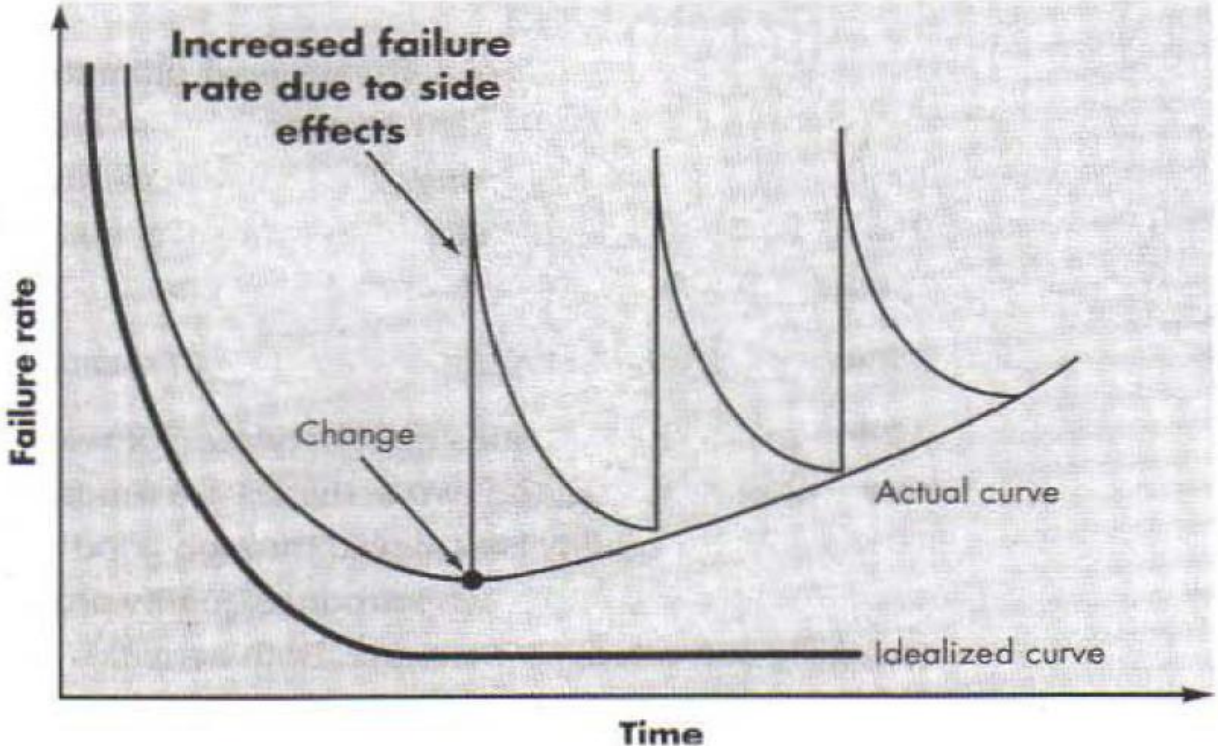
2. Software doesn't "wear out"

Below figure 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 then corrected, and failure rate drops to a steady state-level (hopefully, quite low) for some period of time.

Failure curve for hardware:



Failure curve for software:



As time passes, however, the failure rate rises again as hardware components suffer from the cumulative affects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

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 Figure Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected (hopefully, 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.

This seeming contradiction can best be explained by considering the **"actual curve"** in Figure during its life, software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in Figure. Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise-the software is deteriorating due to change.

Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine-executable code. Therefore, software maintenance involves considerably more complexity than hardware maintenance.

3. Although the industry is moving toward component-based construction, most software continues to be custom built.

Consider the manner in which the control hardware for a computer based product is designed and built. The design engineer draws a simple schematic of the digital circuitry, does some fundamental analysis to ensure that proper function will be achieved, and then goes to the shelf where catalogs of digital components exist. Each integrated circuit has a part number, a defined and validated function, a well-defined interface, and a standard set of integration guidelines. After each component is selected, it can be ordered off the shelf.

As an engineering discipline evolves, a collection of standard design components is created. Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems. The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, i.e., the parts that represent something new. In the hardware world, component reuse is a natural part of the engineering process. In the software world, it has only begun to be achieved on a broad scale.

A software component should be designed and implemented so that it can be reused in many different programs. Modem reusable components encapsulate both data and the processing that is applied to the data, enabling the software engineer to create new applications from reusable parts. For example, today's user interfaces are built with reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structures and processing detail required to build the interface are contained within a library of reusable components for interface construction.

THE CHANGING NATURE OF SOFTWARE:

System software: System software is a collection's of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) processes complex, but determinate, information structures. Other systems applications (e.g, operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data. In either case, the systems 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.

Application software: Application software consists of standalone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making. In addition to conventional data processing applications, application software is used to control business functions in real-time (e.g., point-of-sale transaction processing, real-time manufacturing process control).

Engineering/scientific software: Formerly characterized by "number crunching" algorithms. engineering and scientific software 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: Embedded software resides within a product or system and is used to implement and control features and functions for the end-user and for the system itself. Embedded software can perform 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, braking systems, etc.).

Product-Line software: Designed to provide a specific capability for use by many different customers, product-line software can focus on a limited and esoteric marketplace (e.g., inventory control products) or address mass consumer markets (e.g., word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications).

Web-applications: "WebApps," span a wide array of applications. In their simplest form, WebApps can be little more than a set of linked hypertext files that present information using text and limited graphics. However, as e-commerce and B2B applications grow in importance, WebApps are evolving into sophisticated computing environments that not only provide standalone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.

Artificial intelligence software: AI software makes use of non numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

Ubiquitous computing: The rapid growth of wireless networking may soon lead to true distributed' computing. The challenge for software engineers will be to develop systems and application software that will allow small devices, personal computers, and enterprise-system to communicate across vast networks.

Netsourcing: The World Wide Web is rapidly becoming a computing engine as well as a content provider. The challenge for software engineers is to architect simple (e.g., personal financial planning) and sophisticated applications that provide benefit to targeted end-user markets worldwide.

Open source: A growing trend that results in distribution of source code for systems applications (e.g., operating systems, database, and development environments) so that customers can make local modifications. The challenge for software engineers is to build source code that is self-descriptive, but, more importantly, to develop techniques that will enable both customers and developers to know what changes have been made and how those changes manifest themselves within the software.

LEGACY SOFTWARE:

These older programs are often referred to as legacy software have been the focus of continuous attention and concern since the 1960s. Dayani-Fard and his colleagues describe legacy software in the following way:

Legacy software systems were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms. The proliferation or such systems is causing headaches for large organizations who find them costly to maintain and risky to evolve.

Liu and his colleagues extend this description by noting that "many legacy systems remain supportive to core business functions and are indispensable to the business." Hence, legacy software is characterized by longevity and business criticality.

1) The Quality of Legacy Software:

There is one additional characteristic that can be present in legacy software is poor quality. Legacy systems sometimes have inextensible designs, convoluted code, poor or nonexistent documentation, test cases and results that were never archived, a poorly managed change history the list can be quite long. And yet, these systems support "core business functions and are indispensable to the business". What can one do?

The only reasonable answer may be to do nothing, at least until the legacy system must undergo some significant change. If the legacy software meets the needs of its users and runs reliably, it isn't broken and does not need to be fixed. However, as time passes legacy systems often evolve for one or more of the following reasons.

The software must be adapted to meet the needs of new computing environments or technology.
The software must be enhanced to implement new business requirements.
The software must be extended to make it interoperable with more modern systems or databases.
The software must be re-architected to make it viable within a network environment.

When these modes of evolution occur, a legacy system must be reengineered so that it remains viable into the future. The goal of modern software engineering is to "devise methodologies that are founded on the notion of evolution;" that is, the notion that "software systems continually change, new software systems are built from the old ones, and, all must interoperate and cooperate with each other".

2) Software Evolution:

Regardless of its application domain, size or complexity. Computer software will evolve over time. Change (often referred to as software maintenance) drives this process and occurs when errors are corrected, when the software is adapted to a new environment, when the customer requests new features or functions, and when the application is reengineered to provide benefit in a modern context. Sam Williams describes this when he writes:

A large-scale programs such as Windows and Solaris expand well into the range of 30 to 50 million lines of code, successful project managers have learned to devote as much time to combing the together out of legacy code as to adding new code. Simply put, in a decade that saw the average PC microchip performance increase a hundredfold, software's inability to scale at even linear rates has gone from dirty little secret to an industry wide embarrassment.

E-Type (Embedded-Type):

This software works closely as the requirement of real-world environment. This software has a high degree of evolution as there are various changes in laws, taxes etc. in the real world situations. For example, online trading software.

The Law of Continuing Change (1974): E-type systems must be continually adapted, or else they become progressively less satisfactory.

The Law of Increasing Complexity (1974): As an E-type system evolves its complexity increases unless work is done to maintain or reduce it.

The Law of self-Regulation (1979): The E-type system evolution process is self-regulating with distribution of product and process measures close to normal.

The Law of Conservation of organizational Stability (1980): The average effective global activity rate in an evolving E-type system is invariant over product lifetime.

The Law of Conservation of Familiarity (1980): As an E-type system evolves all associated with it, developers, sales personnel, and users, for example, must maintain mastery of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.

The Law of Continuing Growth (1980): The functional content of E-type systems must be continually increased to maintain user satisfaction over the system's lifetime.

The Law of Declining Quality (1996): The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

The Feedback System Law (1996): E-type evolution processes constitute multilevel, multiloop, multiagent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

SOFTWARE MYTHS:

Software myths are beliefs about software and the process used to build it can be traced to the earliest days of computing. Myths have a number of attributes that have made them insidious. For instance, myths appear to be reasonable statements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often promulgated by experienced practitioners who "know the score,"

Today, most knowledgeable software engineering professionals recognize myths for what they are misleading attitudes that have caused serious problems for managers and technical people alike. However, old attitudes and habits are difficult to modify and remnants of software myths are still believed.

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. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

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 adaptable? 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: If we get behind schedule, we can add more programmers and catch up (sometimes called the Mongolian horde concept).
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: Although a comprehensive and stable statement of requirements is not always possible, an ambiguous statement of objectives is a recipe for disaster. Unambiguous requirements (usually derived iteratively) are developed only through effective and continuous 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. When requirement changes are requested early (before design or code has been started), cost impact is relatively small. However, as time passes, cost impact grows rapidly resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modification.

Practitioner's myths:

Myths that are still believed by software practitioners have been fostered by over 50 years of programming culture. During the early days of software, programming was viewed as an art form. Old ways and attitudes die hard.

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 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 errors.

Myth: The only deliverable work product/or 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 importantly, 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 – A LAYERED TECHNOLOGY:

The IEEE [IEE93] has developed a more comprehensive definition when it states:

Software Engineering: The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software, that is, the application of engineering to software. what is "systematic", "disciplined" and "quantifiable" to one software team may be burdensome to another. We need discipline, but we also need adaptability and agility.

Software Engineering Layers:

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

The foundation for software engineering is the process layer. Software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework that must be established for effective delivery of sotlware engineering technology. The software process forms the basis for management control of software projects and establishes 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.

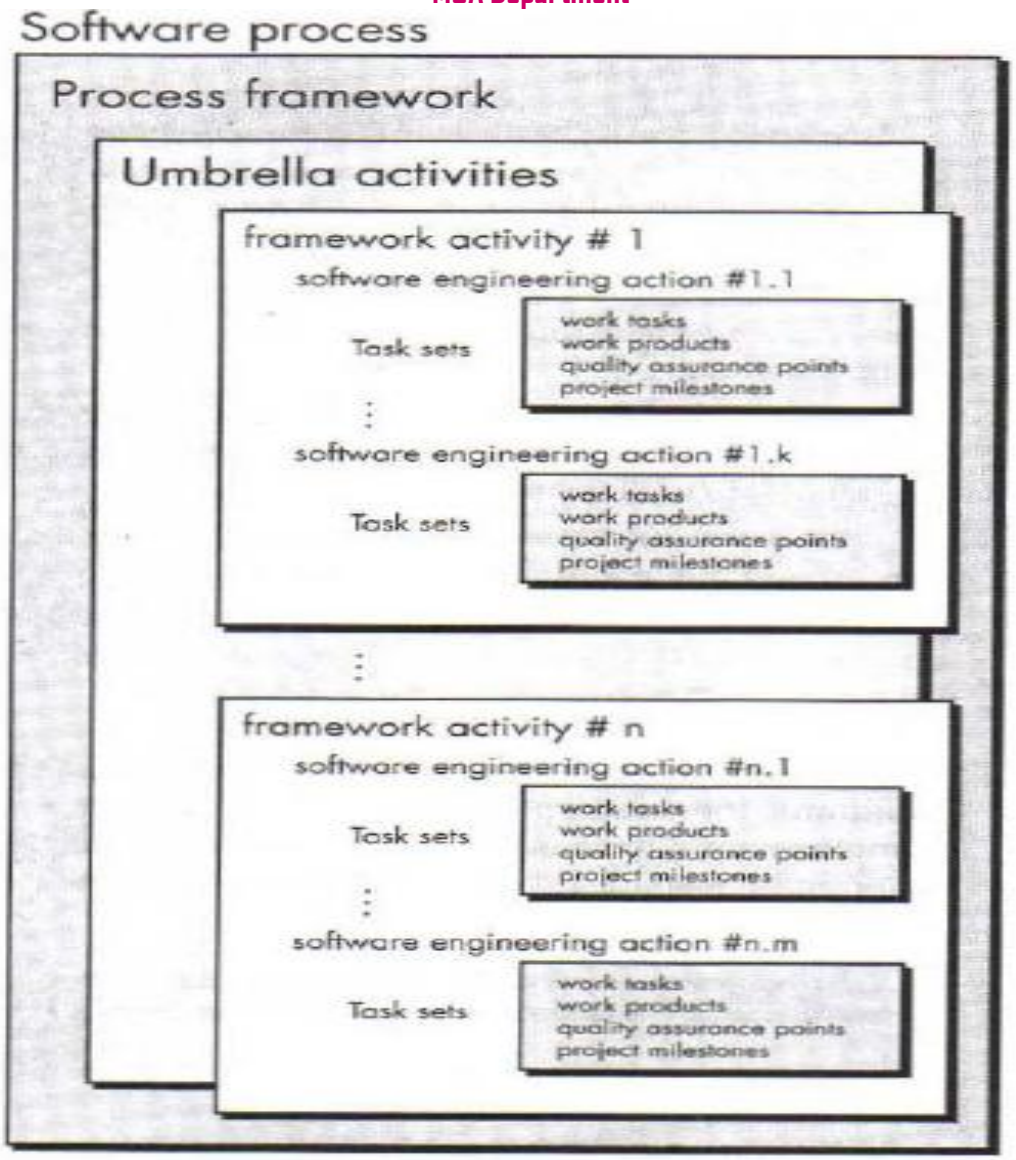
Software engineering methods provide the technical "how to 's" for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support. Software engineering methods rely on a set of basic principles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

Software engineering tools provide automated or semiautomated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer aided software engineering, is established.

A PROCESS FRAMEWORK:

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

Referring to figure, each framework activity is populated by a set of software engineering actions is a collection of related tasks that produces a major software engineering



work product (e.g. design is a software engineering action). Each action is populated with individual work tasks that accomplish some part of the work implied by the action.
The following generic process framework (used as a basis for the description of process models in subsequent chapters) is applicable to the vast majority of software projects.

Communication: This framework activity involves heavy communication and collaboration with the customer (and other stakeholders) and encompasses requirements gathering and other related activities.

Planning: This activity establishes a plan for the software engineering work that follows. It describes the technical tasks to be conducted, the risks that are likely, the resources that will be required. the work products to be produced, and a work schedule.

Modeling: This activity encompasses the creation of models that allow the developer and the customer to better understand software requirements and the design that will achieve those requirements.

Construction: This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.

Deployment: The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation. using an example derived from the generic process framework, the modeling activity is composed of two software engineering actions---analysis and design. Analysis encompasses a set of work tasks (e.g., requirements gathering, elaboration, negotiation, specification, and validation) that lead to the creation of the analysis model (and/or requirements specification). Design encompasses work tasks (data design, architectural design, interface design, and component-level design) that create a design model (and/or a design specification).

Referring again to Figure, each software engineering action is represented by a number of different task sets-each a collection of software engineering work tasks, related work products, quality assurance points, and project milestones. The task set that best accommodates the needs of the project and the characteristics of the team is chosen. This implies that a software engineering action (e.g., design) can be adapted to the specific needs of the software project and the characteristics of the project team.
The framework described in the generic view of software engineering is complemented by a number of umbrella activities. Typical activities in this category include:

Software project tracking and control: allows the software team to assess progress against the project plan and take necessary action to maintain schedule.
Risk management: assesses risks that may effect the outcome of the project or the quality of the product.
Software quality assurance: defines and conducts the activities required to ensure software quality.
Formal technical reviews: assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next action or activity.
Measurement: defines and collects process, project, and product measures that assist the learn in delivering software that meets customers' needs, can be used in conjunction with all other framework and umbrella activities.
Software configuration management: manages the effects of change throughout the software process.
Reusability management: defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.
Work product preparation and production: encompasses the activities required to create work products such as models, documents, logs, forms, and lists.

All process models can be characterized within the process framework shown in Figure. Intelligent application of any software process model must recognize that adaptation (to the problem, project, team, and organizational culture) is essential for success. But process models do differ fundamentally in:
- The overall flow of activities and tasks and the interdependencies among activities and tasks.
- The degree to which work tasks are defined within each framework activity.
- The degree to which work products are identified and required.
- The manner which quality assurance activities are applied.

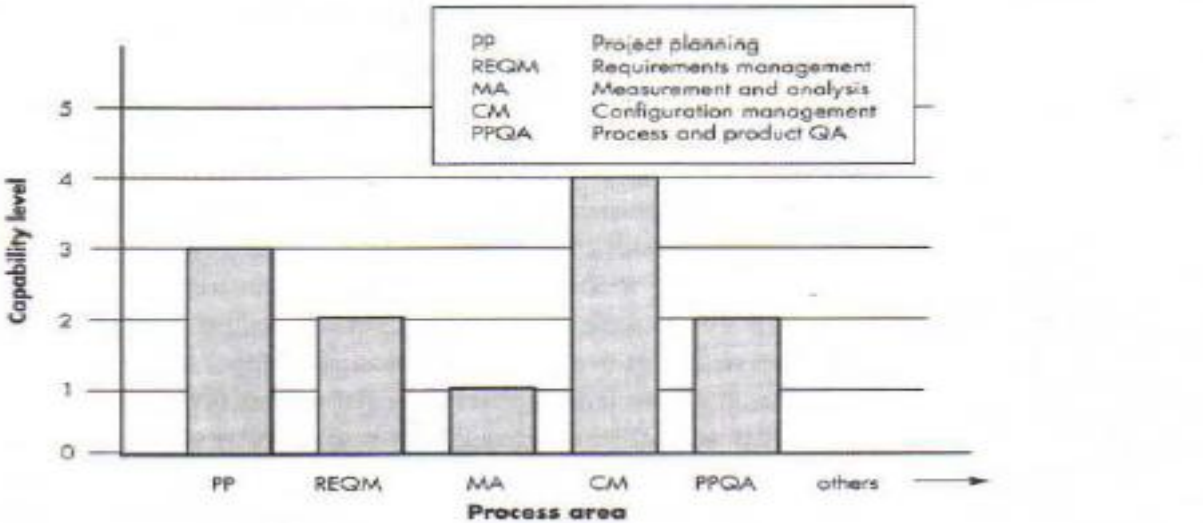
- The manner in which project tracking and control activities are applied.
- The overall degree of detail and rigor with which the process is described.
- The degree to which customer and other stakeholders are involved with the project.
- The level of autonomy given to the software project team.
- The degree to which team organization and roles are prescribed.

THE CAPABILITY MATURITY MODEL INTEGRATION (CMMI) :

The Software Engineering Institute (SEI) has developed a comprehensive process meta-model that is predicated on a set of system and software engineering capabilities that should be present as organizations reach different levels of process capability and maturity. To achieve these capabilities, the SEI contends that an organization should develop a process model that conforms to the capability Maturity Model Integration (CMMI) guidelines [CMM02].

The CMMI represents a process meta-model in two different ways: (1) as a continuous model and (2) as a staged model. The continuous CMMI meta-model describes a process in two dimensions as illustrated in Figure. Each process area (e.g., project planning or requirements management) is formally assessed against specific goals and practices and is rated according to the following capability levels.

Level 0: Incomplete. The process area (e.g., requirements management) is either not performed or does not achieve all goals and objectives defined by the CMMI for level 1 capability.
Level 1: Performed. All of the specific goals of the process area (as defined by the CMMI) have been satisfied. Work tasks required to produce defined work products are being conducted.
Level 2: Managed. All level 1 criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy, all people doing the work have access to adequate resources to get the job done. Stakeholders are actively involved in the process area as required, all work tasks and work products are "monitored, controlled, and reviewed; and are evaluated for adherence to the process description [CMM02].
Level 3: Defined. All level 2 criteria have been achieved. In addition, the process is -tailored from the organization's set of standard processes according to the organization's tailoring guidelines, and contributes work products, measures and other process-improvement information to the organizational process assets" [CMM02].
Level 4: Quantitatively managed. All level .3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment. "Quantitative objectives for quality and process performance are established and used as criteria in managing the process" [CMM02].



Level 5: Optimized. All capability level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative (statistical) means to meet changing customer needs and to continually improve the efficacy of the process area under consideration" [CMM02].

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

For example, **project planning** is one of eight process areas defined by the CMMI for the "project management" category. The specific goals (SG) and the associated specific practices (SP) defined for project planning are [CMM02]:

SG 1 Establish estimates
SP 1.1-1 Estimate the scope of the project
SP 1.2-1 Establish estimates of work product and task attributes
SP 1.3-1 Define project life cycle
SP 1.4-1 Determine estimates of effort and cost
SG 2 Develop a Project Plan
SP 2.1-1 Establish the budget and schedule
SP 2.2-1 Identify project risks
SP 2.3-1 Plan for data management
SP 2.4-1 Plan for project resources
SP 2.5-1 Plan for needed knowledge and skills
SP 2.6-1 Plan stakeholder involvement
SP 2.7-1 Establish the project plan
SG 3 Obtain commitment to the plan
SP 3.1-1 Review plans that affect the project
SP 3.2-1 Reconcile work and resource levels
SP 3.3-1 Obtain plan commitment

In addition to specific goals and practices, the CMMI also defines a set of five generic goals and related practices for each process area. Each of the five generic goals corresponds to one of the five capability levels. Hence, to achieve a

particular capability level, the generic goal for that level and the generic practices that correspond to that goal must be achieved. To illustrate, the generic goals (GG) and practices (GP) for the project planning process area are [CMM02].

GG 1 Achieve specific goals

GP 1.1 Perform base practices

GG 2 Institutionalize a managed process

GP 2.1 Establish an organizational policy
GP 2.2 Plan the process
GP 2.3 Provide resources
GP 2.4 Assign responsibility
GP 2.5 Train people
GP 2.6 Manage configurations
GP 2.7 Identify and involve relevant stakeholders
GP 2.8 Monitor and control the process
GP 2.9 Objectively evaluate adherence
GP 2.10 Review status with higher level management

GG 3 Institutionalize a defined process

GP 3.1 Establish a defined process
GP 3.2 Collect improvement information

GG 4 Institutionalize a quantitatively managed process

GP 4.1 Establish quantitative objectives for the process
GP 4.2 Stabilize sub process performance

GG 5 Institutionalize an optimizing process

GP 5.1 Ensure continuous process improvement
GP 5.2 Correct root causes of problems

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

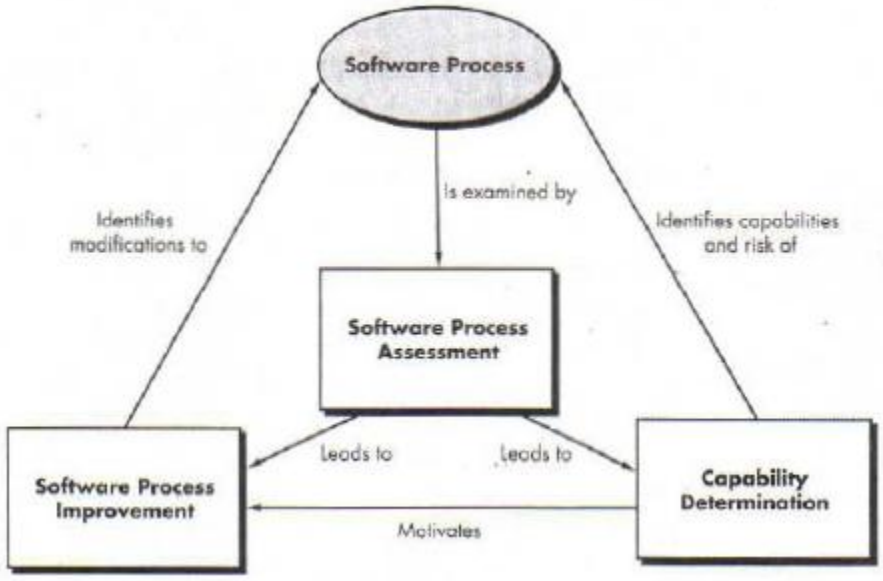
Process areas required to achieve a maturity level:

Optimizing	**Continuous process improvement**	**Organizational Innovation and Deployment Causal Analysis and Resolution**
Quantitatively Managed	**Quantitative management**	**Organizational Process Performance Quantitative Project Management**

Defined	Process standardization	Requirements Development Technical Solution Product Integration Verification Validation Organizational Process Focus Organizational Process Definition Organizational Training Integrated Project Management Integrated Supplier Management Risk Management Decision Analysis and Resolution Organizational Environment for Integration Integrated Teaming
Managed	Basic Project management	Requirements Management Project Planning Project Monitoring and Control Supplier Agreement Management Measurement and Analysis Process and Product Quality Assurance Configuration Management
Performed		

PROCESS ASSESSMENT

The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs, or that it will exhibit the technical characteristics that will lead to long-term quality characteristics. Process patterns must be coupled with solid software engineering practice. In addition, the process itself should be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering. The relationship between the software process and the methods applied for assessment and improvement is shown in Figure. A number of different approaches to software process assessment have been proposed over the past few decades.



Standard CMMI Assessment Method for Process Improvement (SCAMPI)

Provides a five-step process assessment model that incorporates initiating, diagnosing, establishing, acting, and learning. The SCAMPI method uses the SEI CMMI as the basis for assessment [SEI00].

CMM-Based Appraisal for Internal Process Improvement (CBA IPI)

Provides a diagnostic technique for assessing the relative maturity of a software organization, using the SEI CMM (a precursor to the CMMI discussed in Section 2.3) as the basis for the assessment [DUN01].

SPICE (ISO/ IEC15504) standard defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process [SPI99].

ISO 9001:2000 for Software is a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies.

The International Organization for Standardization (ISO) has developed the ISO 9001:2000 standard [ISO00] to define the requirements for a quality management system (Chapter 26) that will serve to produce higher quality products and thereby improve customer satisfaction.

ISO 9001:2000 has adopted a "plan-do-check-act" cycle that is applied to the quality management elements of a software project. Within a software context, "plan" establishes the process objectives, activities, and tasks necessary to achieve high quality software and resultant customer satisfaction. "Do" implements the software process (including both framework and umbrella activities). "Check" monitors and measures the process to ensure that all requirements established for quality management have been achieved. "Act" initiates software process improvement activities that continually work to improve the process.

PERSONAL AND TEAM PROCESS MODELS:

The best software process is one that is close to the people who will be doing the work. If a software process model has been developed at a corporate or organizational level, it can be effective only if it is acceptable to significant adaptation to meet the needs of the project team that is actually doing software engineering work. In an ideal setting, each software engineer would create a process that best fits his or her needs, and at the same time meets the broader needs of the team and the organization. Alternatively, the team itself would create its own process, and at the same time meet the narrower needs of individuals and the broader needs of the organization.

Watts Humphrey argues that it is possible to create a "personal software process" and a team software process." Both require hard work, training and coordination, but both are achievable.

Personal Software Process (PSP):

Every developer uses some process to build computer software. The process may be understand, may change on a daily basis, may not be efficient, effective or even successful, but a process does exist. Watts Humphrey suggests that in order to change an ineffective personal process, an individual must move through four phases. each requiring training and careful instrumentation. The personal software process (PSP) emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product. In addition, PSP makes the practitioner responsible for project planning (e.g., estimating and scheduling) and empowers the practitioner to control the quality of all software work products that are developed.

The PSP process model defines five framework activities: planning, high-level design, high-level design review, development, and postmortem.

Planning. This activity isolates requirements and, based on these, develops both size and resource estimates. In addition, a defect estimate (the number of defects projected for the work) is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.

High-level design. External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists, all issues are recorded and tracked.

High-level design review. Formal verification methods are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.

Development. The component level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for all important tasks and work results.

Postmortem. Using the measures and metrics collected (a substantial amount of data that should be analyzed statistically), the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.

2) Team Software Process (TSP):

Because many industry-grade software projects are addressed by a team of practitioners, Watts Humphrey extended the lessons learned from the introduction of PSP and proposed a team software process (TSP). The goal of TSP is to build a "self-directed" project team that organizes itself to produce high-quality software. Humphrey defines the following objectives for TSP:

- Build self-directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams (IPT) of 3 to about 20 engineers.
- Show managers how to coach and motivate their teams and how to help them sustain peak performance.
- Accelerate software process improvement by making CMM level 5 behavior normal and expected.
- Provide improvement guidance to high-maturity organizations.
- Facilitate university teaching of industrial-grade learn skills.

A self-directed team has a consistent understanding of its overall goals and objectives. It defines roles and responsibilities for each team member, tracks quantitative project data (about productivity and quality), identifies a team process that is appropriate for the project and a strategy for implementing the process; defines local standards that are applicable to the team's software engineering work; continually assesses risk and reacts to it: and tracks, manages, and reports project status.

TSP defines the following framework activities: launch, high-level design, implementation, integration and test, and postmortem Like their counterparts in PSP (note that terminology is somewhat different), these activities enable the team to plan, design, and construct software in a disciplined manner while at the same time quantitatively measuring the process and the product. The postmortem sets the stage for process improvements.

TSP makes use of a wide variety of scripts, forms, and standards that serve to guide team members in their work. Scripts define specific process activities (i.e., project launch, design, implementation, integration and testing, and postmortem) and other more detailed work functions (e.g., development planning, requirements development, software configuration management, and unit test) that are part of the team process. To illustrate, consider the initial process activity-project launch.

Each project is "launched" using a sequence of tasks (defined as a script) that enables the team to establish a solid basis for starting the project: The following launch script (outline only) is recommended.

- Review project objectives with management and agree on and document team goals.
- Establish team roles.
- Define the team's development process.
- Make a quality plan and set quality targets.
- Plan for the needed support facilities.
- Produce an overall development strategy.
- Make a development plan for the entire project.
- Make detailed plans for each engineer for the next phase.
- Merge the individual plans into a team plan.
- Rebalance team workload to achieve a minimum overall schedule.
- Assess project risks and assign tracking responsibility for each key risk.

This accommodates the iterative nature of many projects and allows the team to adapt to changing customer needs and lessons learned from previous activities.

TSP recognizes that the best software teams are self-directed. Team members set project objectives, adapt the process to meet their needs, have control over schedule, and through measurement and analysis of the metrics collected, work continually to improve the team's approach to software engineering.

UNIT – II

UNIT II

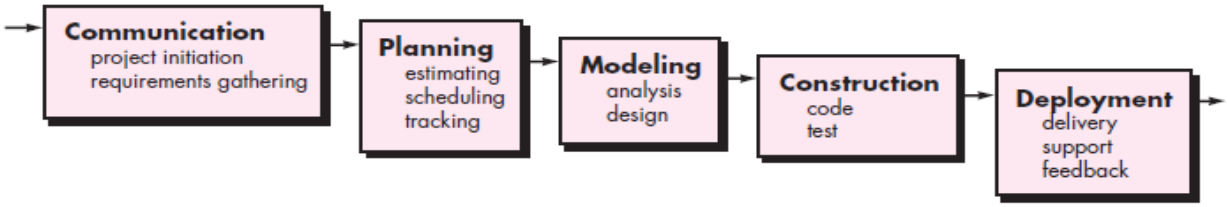
Process Model

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

1. The Waterfall Model

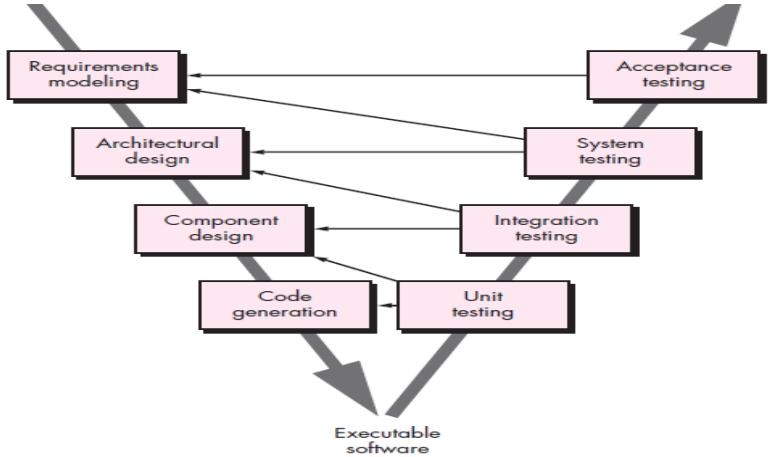
There are times when the requirements for a problem are well understood—when work flows from **communication** through **deployment** in a reasonably linear fashion. This situation is sometimes encountered when well-defined adaptations or enhancements to an existing system must be made (e.g., an adaptation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

The *waterfall model,* sometimes called the *classic life cycle*, suggests a systematic, sequential approach6 to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software



A variation in the representation of the waterfall model is called the *V-model.* Represented in below Figure , the V-model [Buc99] depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities. As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution. Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moved down the left side.7 In reality, there is no fundamental difference between the classic life cycle and the V-model. The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work

The V-model



verification and validation actions are applied to earlier engineering work. The waterfall model is the oldest paradigm for software engineering. However, over the past three decades, criticism of this process model has caused even ardent supporters to question its efficacy. Among the problems that are sometimes encountered when the waterfall model is applied are:

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.

It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.

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.

In an interesting analysis of actual projects, Bradac 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 states tend to be more prevalent at the beginning and end of a linear sequential process.

Today, software work is fast-paced and subject to a never-ending stream of changes (to features, functions, and information content). The waterfall model is often inappropriate for such work. However, it can serve as a useful process model in situations where requirements are fixed and work is to proceed to completion in a linear manner.

2. Incremental Process Models

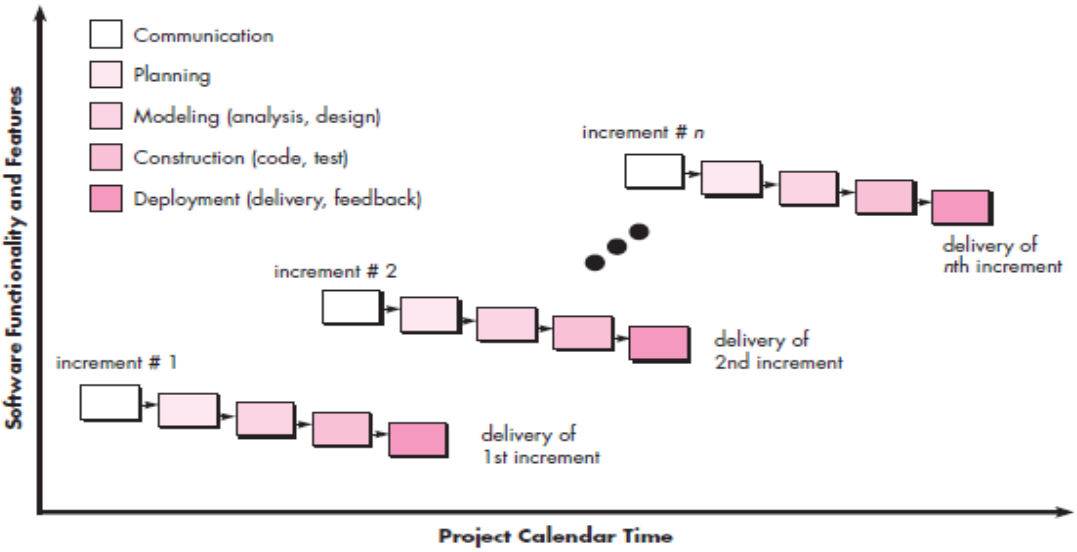
There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort precludes a purely linear process. In addition, there may be a compelling need to provide a limited set of software functionality to users quickly and then refine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

The *incremental* model combines elements of linear and parallel process flows, the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software in a manner that is similar to the increments produced by an evolutionary process flow

.

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 evaluation). As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.



The incremental process 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.8 Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment. In addition, increments can be planned to manage technical risks. For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain. It might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end users without inordinate delay.

3. Evolutionary Process Models

Software, like all complex systems, evolves over a period of time. Business and product requirements often change as development proceeds, making a straight line path to an end product unrealistic; tight market deadlines 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, you need a process model that has been explicitly designed to accommodate a product that evolves over time. Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software. In the paragraphs that follow, I present two common evolutionary process models.

Prototyping

Often, a customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human-machine interaction should take. In these, and many other situations, a *prototyping paradigm* may offer the best approach. Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models noted in this chapter. Regardless of the manner in which it is applied, the prototyping paradigm assists you and other stakeholders to better understand what is to be built when requirements are fuzzy. The prototyping paradigm begins with communication. You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are

known, and outline areas where further definition is mandatory. A prototyping iteration is planned quickly, and modeling (in the form of a "quick design") occurs. A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display formats). The quick design leads to the construction of a prototype.

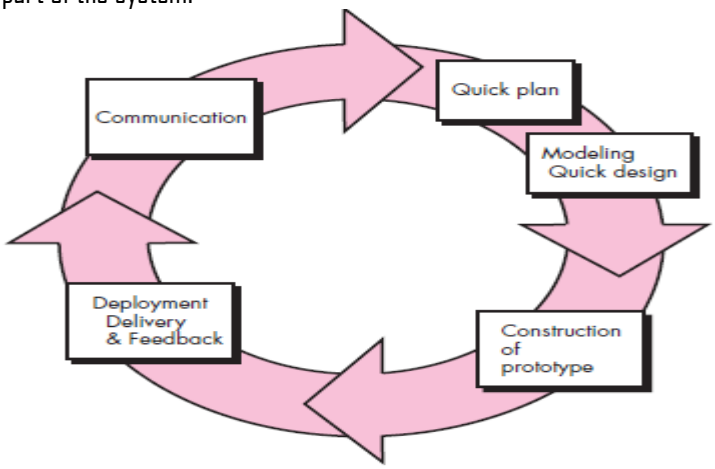
The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements. Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done.

Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is to be built, you can make use of existing program fragments or apply tools (e.g., report generators and window managers) that enable working programs to be generated quickly. But what do you do with the prototype when it has served the purpose described earlier? Brooks provides one 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.

The prototype can serve as "the first system." The one that Brooks recommends you throw away. But this may be an idealized view. Although some prototypes are built as "throwaways," others are evolutionary in the sense that the prototype slowly evolves into the actual system. Both stakeholders and software engineers like the prototyping paradigm. Users get a feel for the actual system, and developers get to build something immediately. Yet, prototyping can be problematic for the following reasons:

Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven't 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, stakeholders cry foul and demand that "a few fixes" be applied to make the prototype a working product. Too often, software development management relents.

As a software engineer, you often make 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, you may become comfortable 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.



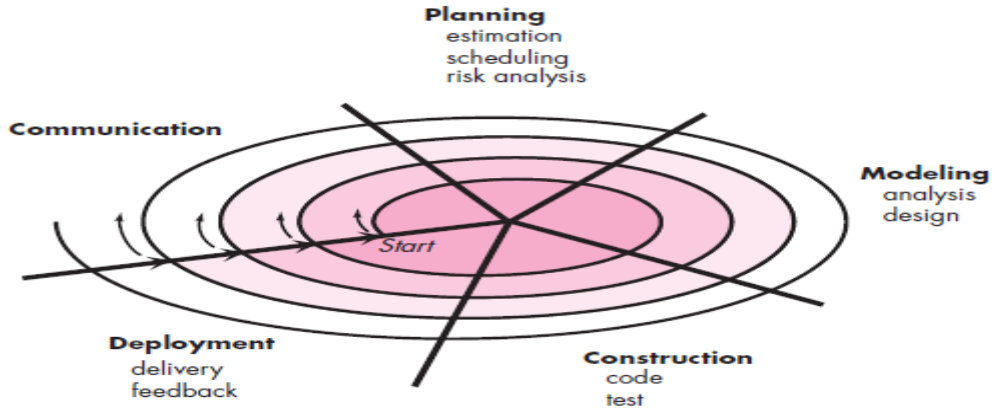
Although problems can occur, prototyping can be an effective paradigm for software engineering. The key is to define the rules of the game at the beginning; that is, all stakeholders should agree that the prototype is built to serve as a mechanism for defining requirements.

The Spiral Model

Originally proposed by Barry Boehm, the *spiral model* is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more complete versions of the software. Boehm describes the model in the following manner:

The spiral development model is a *risk*-driven *process model* generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems. It has two main distinguishing features. One is a *cyclic* approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of *anchor point milestones* for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

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



A spiral model is divided into a set of framework activities defined by the software engineering team. For illustrative purposes, I use the generic framework activities discussed earlier Each of the framework activities represent one segment of the spiral path illustrated in Figure. As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center. Risk is considered as each revolution is made. *Anchor point milestones*—a combination of work products and conditions that are attained along the path of the spiral—are noted for each evolutionary pass. The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manager adjusts the planned number of iterations required to complete the software.

Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software. Therefore, the first circuit around the spiral might represent a "concept development project that starts at the core of the spiral and continues for multiple iterations10 until concept development is complete. If the concept is to be developed into an actual product, the process proceeds outward on the spiral and a "new product

development project" commences. The new product will evolve through a number of iterations around the spiral. Later, a circuit around the spiral might be used to represent a "product enhancement project." 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. 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 you 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.

Specialized Process Model

Specialized process models take on many of the characteristics of one or more of the traditional models presented in the preceding sections. However, these models tend to be applied when a specialized or narrowly defined software engineering approach is chosen

1. Component-Based Development

Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built. The *component-based development model* incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software. However, the component-based development model constructs applications from prepackaged software components.

Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented classes or packages of classes. Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps (implemented using an evolutionary approach):

1. Available component-based products are researched and evaluated for the application domain in question.
2. Component integration issues are considered.
3. A software architecture is designed to accommodate the components.
4. Components are integrated into the architecture.
5. Comprehensive testing is conducted to ensure proper functionality.

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits. Your software engineering team can achieve a reduction in development cycle time as well as a reduction in project cost if component reuse becomes part of your culture

2 The Formal Methods Model

The *formal methods model* encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable you to specify,develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation on this approach, called *cleanroom software engineering* , is currently applied by some software development organizations.

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

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

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

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

3 Aspect-Oriented Software Development

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

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

Grundy provides further discussion of aspects in the context of what he calls *aspect-oriented component engineering* (AOCE):

AOCE uses a concept of horizontal slices through vertically-decomposed software components, called "aspects," to characterize cross-cutting functional and non-functional properties of components. Common, systemic aspects include user interfaces, collaborative work, distribution, persistency, memory management, transaction processing, security, integrity and so on. Components may provide or require one or more "aspect details"relating to a particular aspect, such as a viewing mechanism, extensible affordance and interface kind (user interface aspects); event generation, transport and receiving (distribution aspects); data store/retrieve and indexing (persistency aspects); authentication, encoding and access rights (security aspects); transaction atomicity, concurrency control and logging strategy (transaction aspects); and so on. Each aspect detail has a number of properties, relating to functional and/or non-functional characteristics of the aspect detail.

A distinct aspect-oriented process has not yet matured. However, it is likely that such a process will adopt characteristics of both evolutionary and concurrent process models. The evolutionary model is appropriate as aspects are identified and then constructed. The parallel nature of concurrent development is essential because aspects are engineered independently of localized software components and yet, aspects have a direct impact on these components. Hence, it is essential to instantiate asynchronous communication between the software process activities applied to the engineering and construction of aspects and components.

Agile Process Model

What is Agility?

Just what is agility in the context of software engineering work? Ivar Jacobson provides a useful discussion:

Agility has become today's buzzword when describing a modern software process. Everyone is agile. An agile team is a nimble team able to appropriately respond to changes. Change is what software development is very much about. Changes in the software being built, changes to the team members, changes because of new technology, changes of all kinds that may have an impact on the product they build or the project that creates the product. Support for changes should be built-in everything we do in software, something we embrace because it is the heart and soul of software. An agile team recognizes that software is developed by individuals working in teams and that the skills of these people, their ability to collaborate is at the core for the success of the project.

In Jacobson's view, the pervasiveness of change is the primary driver for agility. Software engineers must be quick on their feet if they are to accommodate the rapid changes that Jacobson describes

But agility is more than an effective response to change It encourages team structures and attitudes that make communication (among team members, between technologists and business people, between software engineers and their managers) more facile. It emphasizes rapid delivery of operational software and de-emphasizes the importance of intermediate work products (not always a good thing); it adopts the customer as a part of the development team and works to eliminate the "us and them" attitude that continues to pervade many software projects; it recognizes that planning in an uncertain world has its limits and that a project plan must be flexible.

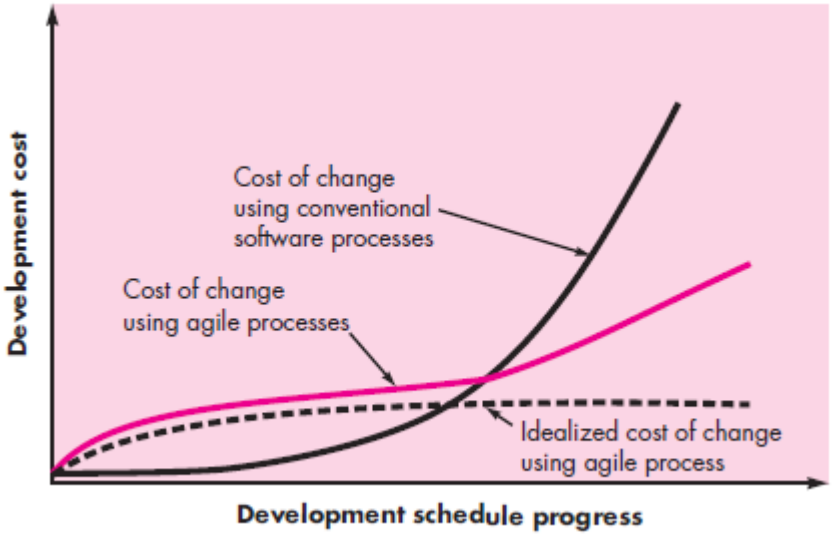
Agility can be applied to any software process. However, to accomplish this, it is essential that the process be designed in a way that allows the project team to adapt tasks and to streamline them, conduct planning in a way that understands the fluidity of an agile development approach, eliminate all but the most essential work products and keep them lean, and

emphasize an incremental delivery strategy that gets working software to the customer as rapidly as feasible for the product type and operational environment.

Agility and the cost of change

The conventional wisdom in software development (supported by decades of experience) is that the cost of change increases nonlinearly as a project progresses It is relatively easy to accommodate a change when a software team is gathering requirements (early in a project). A usage scenario might have to be modified, a list of functions may be extended, or a written specification can be edited. The costs of doing this work are minimal, and the time required will not adversely affect the outcome of the project. But what if we fast-forward a number of months? The team is in the middle of validation testing (something that occurs relatively late in the project), and an important stakeholder is requesting a major functional change. The change requires a modification to the architectural design of the software, the design and construction of three new components, modifications to another five components, the design of new tests, and so on. Costs escalate quickly, and the time and cost required to ensure that the change is made without unintended side effects is nontrivial.

Proponents of agility argue that a well-designed agile process "flattens" the cost of change curve (Figure , shaded, solid curve), allowing a software team to accommodate changes late in a software project without dramatic cost and time impact. You've already learned that the agile process encompasses incremental delivery. When incremental delivery is coupled with other agile practices such as continuous unit testing and pair programming, the cost of making a change is attenuated. Although debate about the degree to which the cost curve flattens is ongoing, there is evidence [Coc01a] to suggest that a significant reduction in the cost of change can be achieved.



What is an agile process?

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

It is difficult to predict in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds.

- For many types of software, design and construction are interleaved.
- That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design.
- Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

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

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

Agility Principles

The Agile Alliance defines 12 agility principles for those who want to achieve agility:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self–organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Not every agile process model applies these 12 principles with equal weight, and some models choose to ignore (or at least downplay) the importance of one or more of the principles

The Politics of Agile Development

There is considerable debate (sometimes strident) about the benefits and applicability of agile software development as opposed to more conventional software engineering processes. Jim Highsmith (facetiously) states the extremes when he characterizes the feeling of the pro-agility camp ("agilists"). "Traditional methodologists are a bunch of stick-in-the-muds who'd rather produce flawless documentation than a working system that meets business needs." As a counterpoint, he states (again, facetiously) the position of the traditional software engineering camp: "Lightweight, er, 'agile' methodologists are a bunch of glorified hackers who are going to be in for a heck of a surprise when they try to scale up their toys into enterprise-wide software."

Like all software technology arguments, this methodology debate risks degenerating into a religious war. If warfare breaks out, rational thought disappears and beliefs rather than facts guide decision making. No one is against agility. The real question is: What is the best way to achieve it? As important, how do you build software that meets customers' needs today and exhibits the quality characteristics that will enable it to be extended and scaled to meet customers' needs over the long term?

There are no absolute answers to either of these questions. Even within the agile school itself, there are many proposed process models, each with a subtly different approach to the agility problem. Within each model there is a set of "ideas" (agilists are loath to call them "work tasks") that represent a significant departure from traditional software engineering. And yet, many agile concepts are simply adaptations of good software engineering concepts. Bottom line: there is much that can be gained by considering the best of both schools and virtually nothing to be gained by denigrating either approach.

Human Factors

Proponents of agile software development take great pains to emphasize the importance of "people factors." As Cockburn and Highsmith state, "Agile development focuses on the talents and skills of individuals, molding the process to specific people and teams." The key point in this statement is that *the process moldsto the needs of the people and team,* not the other way around

If members of the software team are to drive the characteristics of the process that is applied to build software, a number of key traits must exist among the people on an agile team and the team itself:

Competence. In an agile development (as well as software engineering) context, "competence" encompasses innate talent, specific software-related skills, and overall knowledge of the process that the team has chosen to apply. Skill and knowledge of process can and should be taught to all people who serve as agile team members.

Common focus. Although members of the agile team may perform different tasks and bring different skills to the project, all should be focused on one goal—to deliver a working software increment to the customer within the time promised. To achieve this goal, the team will also focus on continual adaptations (small and large) that will make the process fit the needs of the team.

Collaboration. Software engineering (regardless of process) is about assessing, analyzing, and using information that is communicated to the software team; creating information that will help all stakeholders understand the work of the team; and building information (computer software and relevant databases) that provides business value for the customer. To accomplish these tasks, team members must collaborate—with one another and all other stakeholders.

Decision-making ability. Any good software team (including agile teams) must be allowed the freedom to control its own destiny. This implies that the team is given autonomy—decision-making authority for both technical and project issues.

Fuzzy problem-solving ability. Software managers must recognize that the agile team will continually have to deal with ambiguity and will continually be buffeted by change. In some cases, the team must accept the fact that the problem they are solving today may not be the problem that needs to be solved tomorrow. However, lessons learned from any problem-solving activity (including those that solve the wrong problem) may be of benefit to the team later in the project.

Mutual trust and respect. The agile team must become what DeMarco and Lister call a "jelled" team. A jelled team exhibits the trust and respect that are necessary to make them "so strongly knit that the whole is greater than the sum of the parts."

Self-organization. In the context of agile development, self-organization implies three things: (1) the agile team organizes itself for the work to be done, (2) the team organizes the process to best accommodate its local environment, (3) the team organizes the work schedule to best achieve delivery of the software increment. Self-organization has a number of technical benefits, but more importantly, it serves to improve collaboration and boost team morale. In essence, the team serves as its own management. Ken Schwaber addresses these issues when he writes: "The team selects how much work it believes it can perform within the iteration, and the team commits to the work. Nothing demotivates a team as much as someone else making commitments for it. Nothing motivates a team as much as accepting the responsibility for fulfilling commitments that it made itself."

Extreme Programming

In order to illustrate an agile process in a bit more detail, I'll provide you with an overview of *Extreme Programming* (XP), the most widely used approach to agile software development. Although early work on the ideas and methods associated with XP occurred during the late 1980s, the seminal work on the subject has been written by Kent Beck. More recently, a variant of XP, called *Industrial XP* (IXP) has been proposed. IXP refines XP and targets the agile process specifically for use within large organizations.

XP Values

Beck [Bec04a] defines a set of five *values* that establish a foundation for all work performed as part of XP—communication, simplicity, feedback, courage, and respect. Each of these values is used as a driver for specific XP activities, actions, and tasks. In order to achieve effective *communication* between software engineers and other stakeholders (e.g., to establish required features and functions for the software), XP emphasizes close, yet informal (verbal) collaboration between customers and developers, the establishment of effective metaphors for communicating important concepts, continuous feedback, and the avoidance of voluminous documentation as a communication medium.

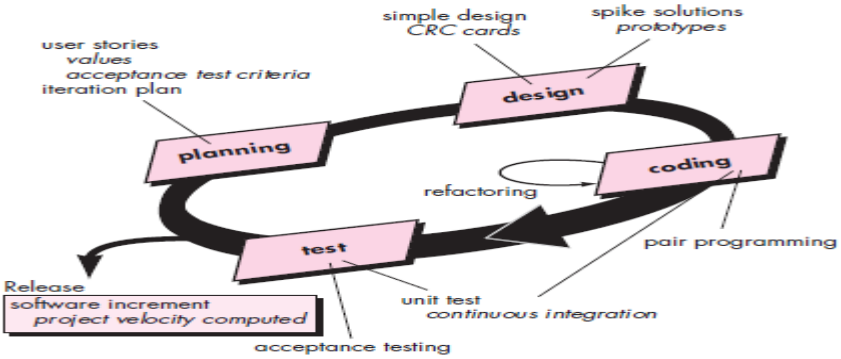
To achieve *simplicity,* XP restricts developers to design only for immediate needs, rather than consider future needs. The intent is to create a simple design that can be easily implemented in code). If the design must be improved, it can be *refactored*4 at a later time. *Feedback* is derived from three sources: the implemented software itself, the customer, and other software team members. By designing and implementing an effective testing strategy, the software (via test results) provides the agile team with feedback. XP makes use of the *unit test* as its primary testing tactic. As each class is developed, the team develops a unit test to exercise each operation according to its specified functionality. As an increment is delivered to a customer, the *user stories* or *use cases* (Chapter 5) that are implemented by the increment are used as a basis for acceptance tests. The degree to which the software implements the output, function, and behavior of the use case is a form of feedback. Finally, as new requirements are derived as part of iterative planning, the team provides the customer with rapid feedback regarding cost and schedule impact. Beck [Bec04a] argues that strict adherence to certain XP practices demands *courage.* A better word might be *discipline.* For example, there is often significant pressure to design for future requirements. Most software teams succumb, arguing that "designing for tomorrow" will save time and effort in the long run. An agile XP team must have the discipline (courage) to design for today, recognizing that future requirements may change dramatically, thereby demanding substantial rework of the design and implemented code.

By following each of these values, the agile team inculcates *respect* among it members, between other stakeholders and team members, and indirectly, for the software itself. As they achieve successful delivery of software increments, the team develops growing respect for the XP process.

The XP Process

Extreme Programming uses an object-oriented approach as its preferred development paradigm and encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing. Figure illustrates the XP process and notes some of the key ideas and tasks that are associated with each framework activity. Key XP activities are summarized in the paragraphs that follow.

Planning. The planning activity (also called *the planning game*) begins with *listening*—a requirements gathering activity that enables the technical members of the XP team to understand the business context for the software and to get a broad feel for required output and major features and functionality. Listening leads to the creation of a set of "stories" (also called *user stories*) that describe required output, features, and functionality for software to be built. Each *story* is written by the customer and is placed on an index card. The customer assigns a *value* (i.e., a priority) to the story based on the overall business value of the feature or function. Members of the XP team then assess each story and assign a *cost*—measured in development weeks—to it. If the story is estimated to require more than three development weeks, the customer is asked to split the story into smaller stories and the assignment of value and cost occurs again. It is important to note that new stories can be written at any time.



Customers and developers work together to decide how to group stories into the next release (the next software increment) to be developed by the XP team. Once a basic *commitment* (agreement on stories to be included, delivery date, and other project matters) is made for a release, the XP team orders the stories that will be developed in one of three ways: (1) all stories will be implemented immediately (within a few weeks), (2) the stories with highest value will be moved up in the schedule and implemented first, or (3) the riskiest stories will be moved up in the schedule and implemented first.

After the first project release (also called a software increment) has been delivered, the XP team computes project velocity. Stated simply, *project velocity* is the number of customer stories implemented during the first release. Project velocity can then be used to (1) help estimate delivery dates and schedule for subsequent releases and (2) determine whether an over commitment has been made for all stories across the entire development project. If an over commitment occurs, the content of releases is modified or end delivery dates are changed.

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

Design. XP design rigorously follows the KIS (keep it simple) principle. A simple design is always preferred over a more complex representation. In addition, the design provides implementation guidance for a story as it is written—nothing less, nothing more. The design of extra functionality (because the developer assumes it will be required later) is discouraged.6

XP encourages the use of CRC cards as an effective mechanism for thinking about the software in an object-oriented context. CRC (class-responsibility collaborator) cards identify and organize the object-oriented classes7 that are relevant to the current software increment. The XP team conducts the design exercise using a process similar to the one described in Chapter 8. The CRC cards are the only design work product produced as part of the XP process.

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

In the preceding section, we noted that XP encourages *refactoring*—a construction technique that is also a method for design optimization. Fowler describes refactoring in the following manner:

Refactoring is the process of changing a software system in such a way that it does no alter the external behavior of the code yet improves the internal structure. It is a disciplined way to clean up code [and modify/simplify the internal design]

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

Because XP design uses virtually no notation and produces few, if any, work products other than CRC cards and spike solutions, design is viewed as a transient artifact that can and should be continually modified as construction proceeds. The intent of refactoring is to control these modifications by suggesting small design changes that "can radically improve the design" [Fow00]. It should be noted, however, that the effort required for refactoring can grow dramatically as the size of an application grows.

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

Coding. After stories are developed and preliminary design work is done, the team does *not* move to code, but rather develops a series of unit tests that will exercise each of the stories that is to be included in the current release (software increment).8 Once the unit test9 has been created, the developer is better able to focus on what must be implemented to pass the test. Nothing extraneous is added (KIS). Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers.

A key concept during the coding activity (and one of the most talked about aspects of XP) is *pair programming*. XP recommends that two people work together at one computer workstation to create code for a story. This provides a mechanism for realtime problem solving (two heads are often better than one) and real-time quality assurance (the code is reviewed as it is created). It also keeps the developers focused on the problem at hand. In practice, each person takes on a slightly different role. For example, one person might think about the coding details of a particular portion of the design while the other ensures that coding standards (a required part of XP) are being followed or that the code for the story will satisfy the unit test that has been developed to validate the code against the story.
As pair programmers complete their work, the code they develop is integrated with the work of others. In some cases this is performed on a daily basis by an integration team. In other cases, the pair programmers have integration responsibility.

This "continuous integration" strategy helps to avoid compatibility and interfacing problems and provides a "smoke testing" environment that helps to uncover errors early.

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

As the individual unit tests are organized into a "universal testing suite", integration and validation testing of the system can occur on a daily basis. This provides the XP team with a continual indication of progress and also can raise warning flags early if things go awry. Wells [Wel99] states: "Fixing small problems every few hours takes less time than fixing huge problems just before the deadline." XP *acceptance tests*, also called *customer tests,* are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer. Acceptance tests are derived from user stories that have been implementedas part of a software release.

IIndustrial XP

Joshua Kerievsky [Ker05] describes *Industrial Extreme Programming* (IXP) in the following manner: "IXP is an organic evolution of XP. It is imbued with XP's minimalist, customer-centric, test-driven spirit. IXP differs most from the original XP in its greater inclusion of management, its expanded role for customers, and its upgraded technical practices." IXP incorporates six new practices that are designed to help ensure that an XP project works successfully for significant projects within a large organization.

Readiness assessment. Prior to the initiation of an IXP project, the organization should conduct a *readiness assessment.* The assessment ascertains whether (1) an appropriate development environment exists to support IXP, (2) the team will be populated by the proper set of stakeholders, (3) the organization has a distinct quality program and supports continuous improvement, (4) the organizational culture will support the new values of an agile team, and (5) the broader project community will be populated appropriately.

Project community. Classic XP suggests that the right people be used to populate the agile team to ensure success. The implication is that people on the team must be well-trained, adaptable and skilled, and have the proper temperament to contribute to a self-organizing team. When XP is to be applied for a significant project in a large organization, the concept of the "team" should morph into that of a *community.* A community may have a technologist and customers who are central to the success of a project as well as many other stakeholders (e.g., legal staff, quality auditors, manufacturing or sales types) who "are often at the periphery of an IXP project yet they may play important roles on the project". In IXP, the community members and their roles should be explicitly defined and mechanisms for communication and coordination between community members should be established

Project chartering. The IXP team assesses the project itself to determine whether an appropriate business justification for the project exists and whether the project will further the overall goals and objectives of the organization. Chartering also examines the context of the project to determine how it complements, extends, or replaces existing systems or processes

Test-driven management. An IXP project requires measurable criteria for assessing the state of the project and the progress that has been made to date. Test-driven management establishes a series of measurable "destinations" and then defines mechanisms for determining whether or not these destinations have been reached.

Retrospectives. An IXP team conducts a specialized technical review after a software increment is delivered. Called a *retrospective,* the review examines "issues, events, and lessons-learned" across a software increment and/or the entire software release. The intent is to improve the IXP process.

Continuous learning. Because learning is a vital part of continuous process improvement, members of the XP team are encouraged (and possibly, incented) to learn new methods and techniques that can lead to a higher quality product. In addition to the six new practices discussed, IXP modifies a number of existing XP practices. *Story-driven development* (SDD) insists that stories for acceptance tests be written before a single line of code is generated. *Domain-driven design* (DDD) is an improvement on the "system metaphor" concept used in XP. DDD suggests the evolutionary creation of a domain model that "accurately represents how domain experts think about their subject". *Pairing* extends the XP pair programming concept to include managers and other stakeholders. The intent is to improve knowledge sharing among XP team members who may not be directly involved in technical development. *Iterative usability* discourages front-loaded

interface design in favor of usability design that evolves as software increments are delivered and users' interaction with the software is studied.

XP Debate

All new process models and methods spur worthwhile discussion and in some instances heated debate. Extreme Programming has done both. In an interesting book that examines the efficacy of XP, Stephens and Rosenberg argue that many XP practices are worthwhile, but others have been overhyped, and a few are problematic. The authors suggest that the codependent nature of XP practices are both its strength and its weakness. Because many organizations adopt only a subset of XP practices, they weaken the efficacy of the entire process. Proponents counter that XP is continuously evolving and that many of the issues raised by critics have been addressed as XP practice matures. Among the issues that continue to trouble some critics of XP are

Requirements volatility. Because the customer is an active member of the XP team, changes to requirements are requested informally. As a consequence, the scope of the project can change and earlier work may have to be modified to accommodate current needs. Proponents argue that this happens regardless of the process that is applied and that XP provides mechanisms for controlling scope creep.

Conflicting customer needs. Many projects have multiple customers, each with his own set of needs. In XP, the team itself is tasked with assimilating the needs of different customers, a job that may be beyond their scope of authority.

Requirements are expressed informally. User stories and acceptance tests are the only explicit manifestation of requirements in XP. Critics argue that a more formal model or specification is often needed to ensure that omissions, inconsistencies, and errors are uncovered before the system is built. Proponents counter that the changing nature of requirements makes such models and specification obsolete almost as soon as they are developed.

Lack of formal design. XP deemphasizes the need for architectural design and in many instances, suggests that design of all kinds should be relatively informal. Critics argue that when complex systems are built, design must be emphasized to ensure that the overall structure of the software will exhibit quality and maintainability. XP proponents suggest that the incremental nature of the XP process limits complexity (simplicity is a core value) and therefore reduces the need for extensive design.