JAMAL MOHAMED COLLEGE (Autonomous)
Department of Computer Applications
III B.C.A.
SOFTWARE ENGINEERING
Unit-I

INTRODUCTION

- A software is a computer program along with the associated documents and the configuration data that make these programs operate correctly.
- A program is a set of instructions (written in form of human-readable code) that performs a specific task.

Types of Software Systems:

- There are many different types of software systems from simple to complex systems.
- These systems may be developed for a particular customer, like systems to support a particular business process, or developed for a general purpose, like any software for our computers such as word processors.

Successful Software System:

- A good software should deliver the main required functionality.
- Other set of attributes — called quality or non-functional — should be also delivered. Examples of these attributes are, the software is written in a way that can be adapted to changes, response time, performance (less use of resources such as memory and processor time), usable; acceptable for the type of the user it's built for, reliable, secure, safe, …etc.
- **What is Software Engineering?**
- The **software** is a collection of integrated programs.
- Software consists of carefully-organized instructions and code written by developers in any of various particular computer languages.
- Computer programs and related documentation such as requirements, design models and user manuals.
- **Engineering** is the application of **scientific** and **practical** knowledge to **invent, design, build, maintain**, and **improve frameworks, processes, etc**.
- Software Engineering provides a standard procedure to design and develop a software.
- Software Engineering is the processes of designing and building something that serves a particular purpose and find a cost effective solution to problems.
- ***Software Engineering** is a systematic approach to the design, development, operation, and maintenance of a software system.*
- The term **software engineering** is the product of two words, **software**, and **engineering**.
- Software engineering is an engineering discipline that's applied to the development of software in a *systematic* approach (called a software process).
- It's the application of theories, methods, and tools to design build a software that meets the specifications efficiently, cost-effectively, and ensuring quality.

- It’s not only concerned with the technical process of building a software, it also includes activities to manage the project, develop tools, methods and theories that support the software production.
- *Not applying software engineering methods results in more expensive, less reliable software, and it can be vital on the long term, as the changes come in, the costs will dramatically increase.*
- Software Engineering is required due to the following reasons:
    - To manage Large software
    - For more Scalability
    - Cost Management
    - To manage the dynamic nature of software
    - For better quality Management

The Evolving Role of Software

- Today, software takes on a dual role.
- It is a product and, at the same time, the vehicle for delivering a product.
- As a product, it delivers the computing potential embodied by computer hardware.
- Whether it resides within a cellular phone or operates inside a mainframe computer, software 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 used to deliver 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.
- Software 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., Internet) and provides the means for acquiring information in all of its forms.
- The role of computer software has undergone significant change over a time span of little more than 50 years.
- Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options have all precipitated more sophisticated and complex computer-based systems.
- Sophistication and complexity can produce dazzling results when a system succeeds, but they can also pose huge problems for those who must build complex systems.
- The lone programmer of an earlier era has been replaced by a team of software specialists, each focusing on one part of the technology required to deliver a complex application.
- And yet, the same questions asked of the lone programmer are being asked when modern 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 the errors before we give the software to customers?

• Why do we continue to have difficulty in measuring progress as software is being developed?

The Changing Nature of Software

Four broad categories of software are evolving to dominate the industry.

1. WebApps

- In the early days of the World Wide Web (circa 1990 to 1995), websites consisted of little more than a set of linked hypertext files that presented information using text and limited graphics.
- As time passed, the augmentation of HTML by development tools (e.g., XML, Java) enabled Web engineers to provide computing capability along with informational content.
- Web-based systems and applications 5 (we refer to these collectively as WebApps) were born.
- Today, WebApps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business applications.
- Semantic Web technologies (often referred to as Web 3.0) have evolved into sophisticated corporate and consumer applications that encompass "semantic databases [that] provide new functionality that requires Web linking, flexible [data] representation, and external access APIs."

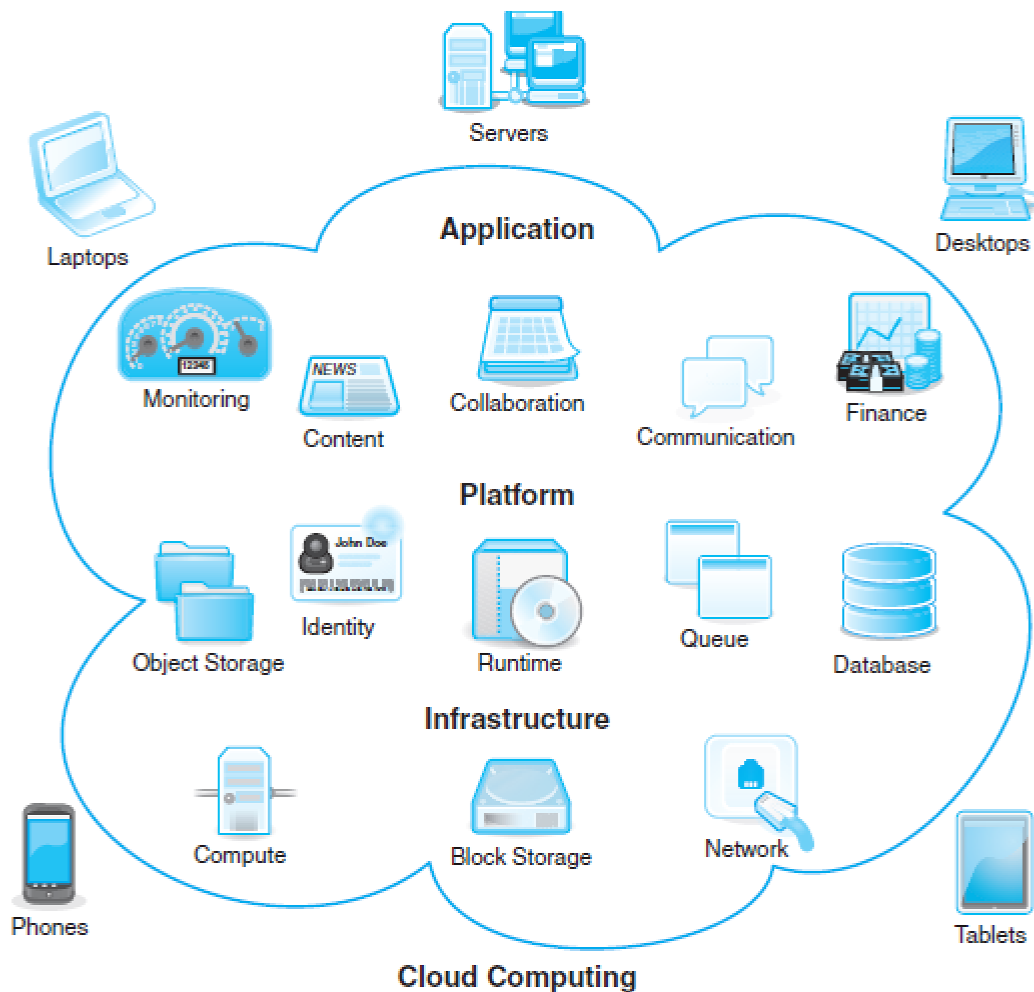
2. Mobile Applications

- The term *app* has evolved to connote software that has been specifically designed to reside on a mobile platform (e.g., iOS, Android, or Windows Mobile).
- In most instances, mobile applications encompass a user interface that takes advantage of the unique interaction mechanisms provided by the mobile platform, interoperability with Web-based resources that provide access to a wide array of information that is relevant to the app, and local processing capabilities that collect, analyze, and format information in a manner that is best suited to the mobile platform.
- In addition, a mobile app provides persistent storage capabilities within the platform.
- It is important to recognize that there is a subtle distinction between mobile web applications and mobile apps.
- A *mobile web application* (WebApp) allows a mobile device to gain access to web-based content via a browser that has been specifically designed to accommodate the strengths and weaknesses of the mobile platform.

- A *mobile app* can gain direct access to the hardware characteristics of the device (e.g., accelerometer or GPS location) and then provide the local processing and storage capabilities noted earlier.
- As time passes, the distinction between mobile WebApps and mobile apps will blur as mobile browsers become more sophisticated and gain access to device level hardware and information.

3. Cloud Computing

- *Cloud computing* encompasses an infrastructure or “ecosystem” that enables any user, anywhere, to use a computing device to share computing resources on a broad scale. The overall logical architecture of cloud computing is:



- Referring to the figure, computing devices reside outside the cloud and have access to a variety of resources within the cloud.
- These resources encompass applications, platforms, and infrastructure.
- In its simplest form, an external computing device accesses the cloud via a Web browser or analogous software.
- The cloud provides access to data that resides with databases and other data structures.

- In addition, devices can access executable applications that can be used in lieu of apps that reside on the computing device.
- The implementation of cloud computing requires the development of an architecture that encompasses front-end and back-end services.
- The front-end includes the client (user) device and the application software (e.g., a browser) that allows the back-end to be accessed.
- The back-end includes servers and related computing resources, data storage systems (e.g., databases), server-resident applications, and administrative servers that use middleware to coordinate and monitor traffic by establishing a set of protocols for access to the cloud and its resident resources.
- The cloud architecture can be segmented to provide access at a variety of different levels from full public access to private cloud architectures accessible only to those with authorization.

4. Product Line Software

- The Software Engineering Institute defines a *software product line* as "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."
- The concept of a line of software products that are related in some way is not new.
- But the idea that a line of software products, all developed using the same underlying application and data architectures, and all implemented using a set of reusable software components that can be reused across the product line provides significant engineering leverage.
- A software product line shares a set of assets that include requirements, architecture , design patterns, reusable components, test cases, and other software engineering work products.
- In essence, a software product line results in the development of many products that are engineered by capitalizing on the commonality among all the products within the product line.

Legacy Software

Hundreds of thousands of computer programs fall into one of the seven broad application domains given below:

- System software – They are a a collection of programs written to service other programs (e.g., compilers, editors, and file management utilities).
- Application software – They are stand-alone programs that solve a specific business need (e.g., point-of-saletransaction processing, real-time manufacturing process control).
- Engineering/scientific software - Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing.
- Embedded software – It 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.

- Product-line software - 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, and personal and business financial applications).
- Esoteric = (requiring or exhibiting knowledge that is restricted to a small group)
- Web applications - WebApps are evolving into sophisticated computing environments that not only provide stand-alone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.
- Artificial intelligence software - Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.
- Some of these are state-of-the-art software—just released to individuals, industry, and government. But other programs are older, in some cases *much* older.
- These older programs—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 of 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.
- Unfortunately, there is sometimes one additional characteristic that is present in legacy software—*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 to do?
- The only reasonable answer may be: 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 other 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".

Software Myths

Software myths—erroneous beliefs about software and the process that is used to build it—can be traced to the earliest days of computing.

Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike.

However, old attitudes and habits are difficult to modify, and remnants of software myths remain.

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: Software 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 requirements changes are requested early (before design or code has been started), the cost impact is relatively small. However, as time passes, the 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, 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 I get the program "running" I have no way of assessing its quality.

Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the technical review. Software reviews are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.

Myth: The only deliverable work product for a successful project is the working program.

Reality: A working program is only one part of a software configuration that includes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineering and, more important, guidance for software support.

Myth: Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.

Reality: Software engineering is not about creating documents. It is about creating a quality product. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

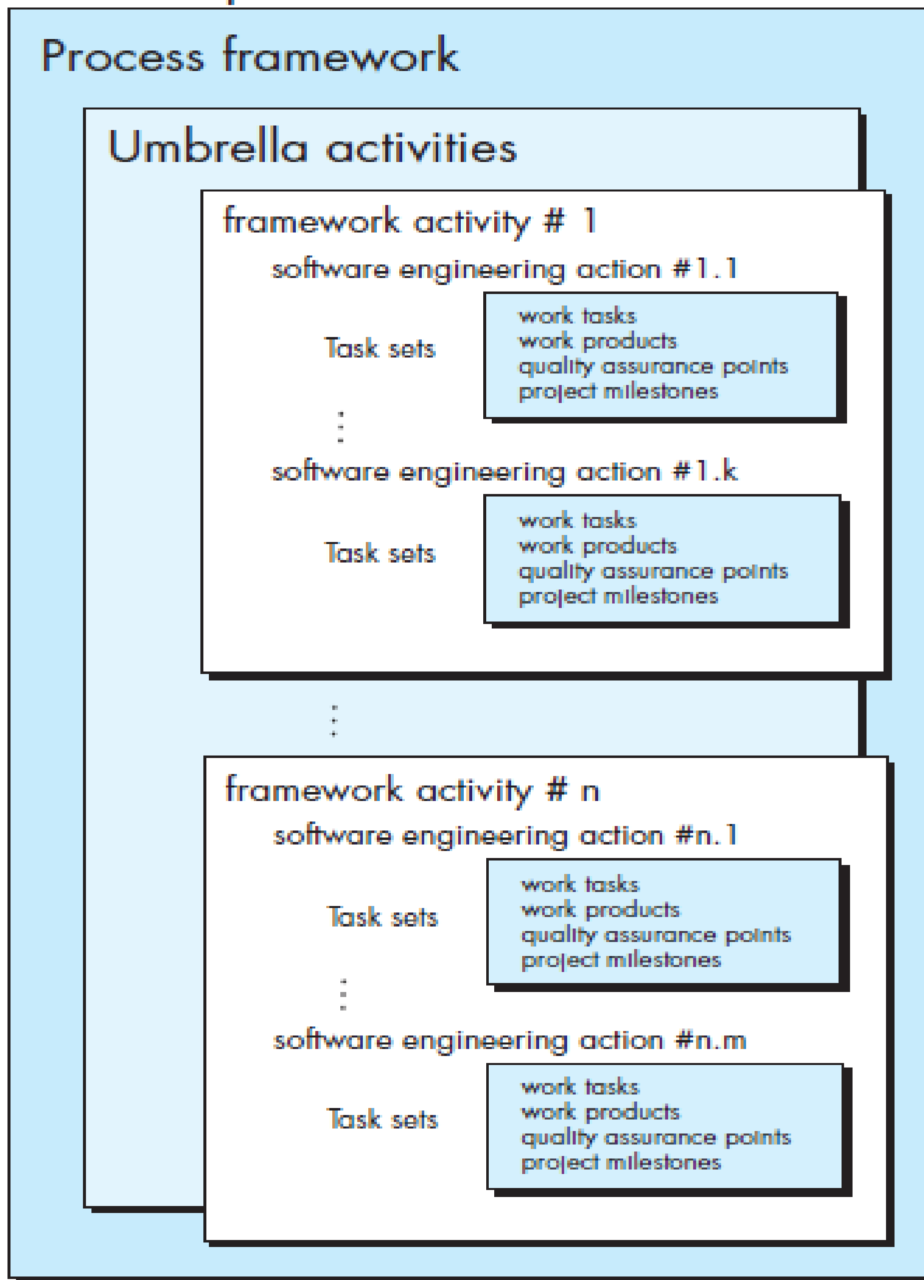
A Generic View of Process

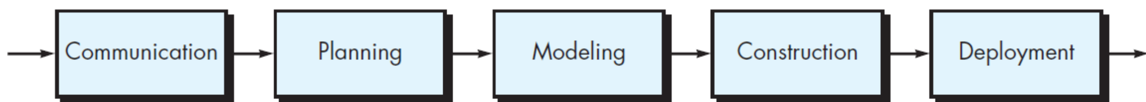
- A process was defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created.
- Each of these activities, actions, and tasks resides within a framework or model that defines their relationship with the process and with one another.
- The software process is represented schematically in the next slide.

- Each framework activity is populated by a set of software engineering actions.
- Each software engineering action is defined by a task set that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.
- A generic process framework for software engineering defines five framework activities— communication, planning, modelling, construction, and deployment .
- In addition, a set of umbrella activities—project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others—are applied throughout the process.
- A process flow —describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time.
- An iterative process flow repeats one or more of the activities before proceeding to the next.
- An evolutionary process flow executes the activities in a “circular” manner.
- A parallel process flow executes one or more activities in parallel with other activities.

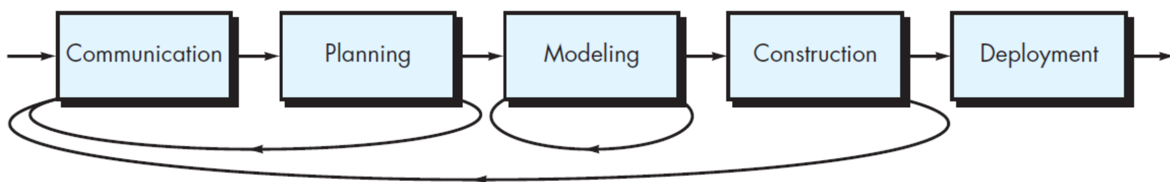
FIGURE

A software process framework

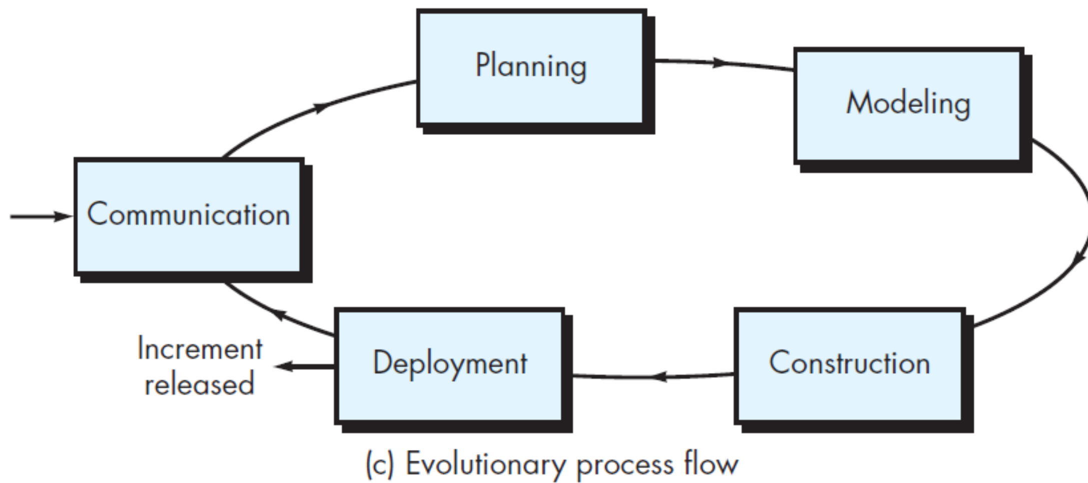




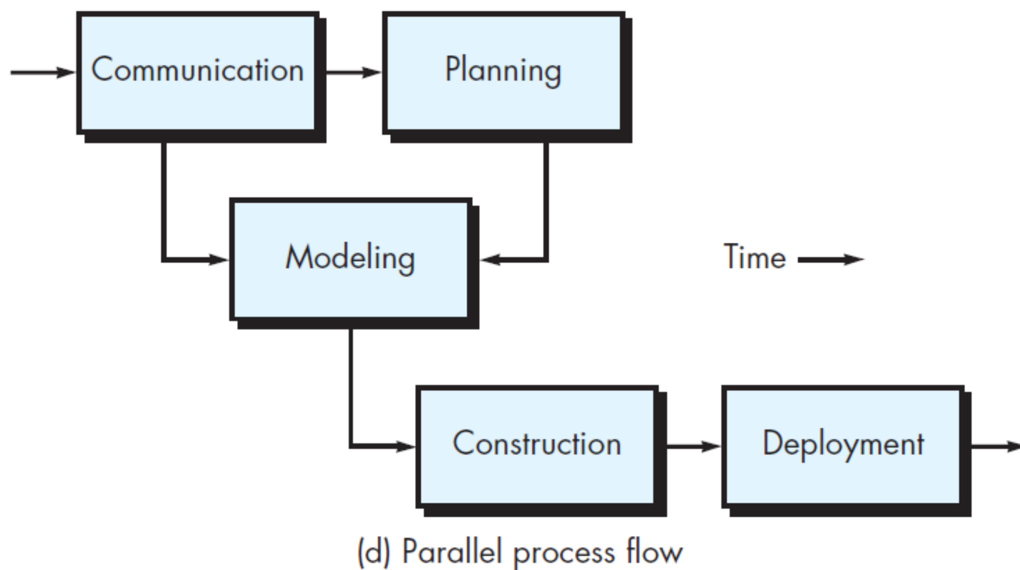
(a) Linear process flow



(b) Iterative process flow



(c) Evolutionary process flow



(d) Parallel process flow

- When you build a product or system, it's important to go through a series of predictable steps—a road map that helps you create a timely, high-quality result. The road map that you follow is called a 'software process.'
- Engineering is the analysis, design, construction, verification, and management of technical (or social) entities. Regardless of the entity to be engineered, the following questions must be asked and answered:
- What is the problem to be solved?
- What characteristics of the entity are used to solve the problem?
- How will the entity (and the solution) be realized?
- How will the entity be constructed?
- What approach will be used to uncover errors that were made in the design and construction of the entity?
- How will the entity be supported over the long term, when corrections, adaptations, and enhancements are requested by users of the entity.

- The work associated with software engineering can be categorized into three generic phases, regardless of application area, project size, or complexity.
- The ***definition phase*** focuses on ***what***.
- That is, during definition, the software engineer attempts to identify what information is to be processed, what function and performance are desired, what system behavior can be expected, what interfaces are to be established, what design constraints exist, and what validation criteria are required to define a successful system.
- The key requirements of the system and the software are identified. Although the methods applied during the definition phase will vary depending on the software engineering paradigm (or combination of paradigms) that is applied, three major tasks will occur in some form: **system or information engineering, software project planning, and requirements analysis .**
- The ***development phase*** focuses on ***how***.
- That is, during development a software engineer attempts to define how data are to be structured, how function is to be implemented within a software architecture, how procedural details are to be implemented, how interfaces are to be characterized, how the design will be translated into a programming language (or nonprocedural language), and how testing will be performed.
- The methods applied during the development phase will vary, but three specific technical tasks should always occur: **software design, code generation, and software testing**.
- The ***support phase*** focuses on change associated with error correction, adaptations required as the software's environment evolves, and changes due to enhancements brought about by changing customer requirements. The support phase reapplies the steps of the definition and development phases but does so in the context of existing software.
- **Four types of change** are encountered during the **support phase**:
  - ***Correction***. Even with the best quality assurance activities, it is likely that the customer will uncover defects in the software. Corrective maintenance changes the software to correct defects.
  - ***Adaptation.*** Over time, the original environment (e.g., CPU, operating system, business rules, external product characteristics) for which the software was developed is likely to change. Adaptive maintenance results in modification to the software to accommodate changes to its external environment.
  - ***Enhancement.*** As software is used, the customer/user will recognize additional functions that will provide benefit. Perfective maintenance extends the software beyond its original functional requirements.
  - ***Prevention.*** Computer software deteriorates due to change, and because of this, preventive maintenance, often called software reengineering, must be conducted to enable the software to serve the needs of its end users. In essence, preventive maintenance makes changes to computer programs so that they can be more easily corrected, adapted, and enhanced.

Process Framework

- A ***process framework*** establishes the foundation for a complete software engineering process by identifying a small number of *framework activities* that are applicable to all software projects, regardless of their size or complexity.
- A generic process framework for software engineering encompasses five activities:
- **Communication.**
  - Before any technical work can commence, it is critically important to communicate and collaborate with the customer (and other stakeholders).
  - The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.
- **Planning.**
  - Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey.
  - The map—called a software project plan—defines the software engineering work by describing 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.
- **Modelling.**
  - Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day. You create a "sketch" of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics.
  - If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.
- **Construction.**
  - What you design must be built. 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.

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

Capability Maturity Model Integration

Maturity is a measurement of the ability of an organization for continuous improvement in a particular discipline.

A maturity model is a tool that helps people assess the current effectiveness of a person or group and supports figuring out what capabilities they need to acquire next in order to improve their performance.

Maturity models are structured as a series of levels of effectiveness. It's assumed that anyone in the field will pass through the levels in sequence as they become more capable.

The Capability Maturity Model was originally developed as a tool for objectively assessing the ability of government contractors' processes to implement a contracted software project.

Working with a maturity model begins with assessment, determining which level the subject is currently performing in. Once you've carried out an assessment to determine your level, then you use the level above your own to prioritize what capabilities you need to learn next. This prioritization of learning is really the big benefit of using a maturity model. It's founded on the notion that if you are at level 2 in something, it's much more important to learn the things at level 3 than level 4. The model thus acts as guide to what to learn, putting some structure on what otherwise would be a more complex process.

The vital point here is that the true outcome of a maturity model assessment isn't what level you are but the list of things you need to work on to improve.

The Capability Maturity Model (CMM) is a methodology used to develop and refine an organization's software development process.

Capability Maturity Model is used as a benchmark to measure the maturity of an organization's software process.

The model describes a five-level evolutionary path of increasingly organized and systematically more mature processes.

CMM was developed and is promoted by the Software Engineering Institute (SEI), a research and development center sponsored by the U.S. Department of Defense (DoD).

SEI was founded in 1984 to address software engineering issues and, in a broad sense, to advance software engineering methodologies.

More specifically, SEI was established to optimize the process of developing, acquiring, and maintaining heavily software-reliant systems for the DoD.

Because the processes involved are equally applicable to the software industry as a whole, SEI advocates industry-wide adoption of the CMM.

Capability Maturity Model Integration (CMMI) is a process level improvement training and appraisal program.

The Capability Maturity Model Integration (CMMI) helps organizations streamline process improvement, encouraging a productive, efficient culture that decreases risks in software, product and service development.

The CMMI starts with an appraisal process that evaluates three specific areas: process and service development, service establishment and management, and product and service acquisition. It's designed to help improve performance by providing businesses with everything they need to consistently develop better products and services.

But the CMMI is more than a process model; it's also a behavioural model. Businesses can use the CMMI to tackle the logistics of improving performance by developing measurable benchmarks, but it can also create a structure for encouraging productive, efficient behavior throughout the organization.

CMMI Maturity Levels

The CMMI model breaks down organizational maturity into five levels. For businesses that embrace CMMI, the goal is to raise the organization up to Level 5, the "optimizing" maturity level. Once businesses reach this level, they aren't done with the CMMI. Instead, they focus on maintenance and regular improvements.

CMMI's five Maturity Levels are:

Initial: Processes are viewed as unpredictable and reactive. At this stage, "work gets completed but it's often delayed and over budget." This is the worst stage a business can find itself in — an unpredictable environment that increases risk and inefficiency.

Ad hoc activities characterize a software development organization at this level. Very few or no processes are described and followed. Since software production processes are not limited, different engineers follow their process and as a result, development efforts become chaotic. Therefore, it is also called a chaotic level.

Managed: There's a level of project management achieved. Projects are "planned, performed, measured and controlled" at this level, but there are still a lot of issues to address.

At this level, the fundamental project management practices like tracking cost and schedule are established. Size and cost estimation methods, like function point analysis, COCOMO, etc. are used.

Defined: At this stage, organizations are more proactive than reactive. There's a set of "organization-wide standards" to "provide guidance across projects, programs and portfolios." Businesses understand their shortcomings, how to address them and what the goal is for improvement.

Quantitatively managed: This stage is more measured and controlled. The organization is working off quantitative data to determine predictable processes that align with stakeholder needs. The business is ahead of risks, with more data-driven insight into process deficiencies.

Optimizing: Here, an organization's processes are stable and flexible. At this final stage, an organization will be in constant state of improving and responding to changes or other opportunities. The organization is stable, which allows for more "agility and innovation," in a predictable environment.

Once organizations hit Levels 4 and 5, they are considered high maturity, where they are "continuously evolving, adapting and growing to meet the needs of stakeholders and customers." That is the goal of the CMMI: To create reliable environments, where products, services and departments are proactive, efficient and productive.

Except for SEI CMM level 1, each maturity level is featured by several Key Process Areas (KPAs) that contains the areas an organization should focus on improving its software process to the next level. The focus of each level and the corresponding key process areas are shown in the fig.

CMM Level	Focus	Key Process Areas
1. Initial	Competent People	NO KPA'S
2. Repeatable	Project Management	Software Project Planning software Configuration Management
3. Defined	Definition of Processes	Process definition Training Program Peer reviews
4. Managed	Product and Process quality	Quantitiative Process Metrics Software Quality Management
5. Optimizing	Continuous Process improvement	Defect Prevention Process change management Technology change management

The focus of each SEI CMM level and the Corresponding Key process areas.

Capability Maturity Model Integration (CMMI) is a successor of CMM and is a more evolved model that incorporates best components of individual disciplines of CMM like Software CMM, Systems Engineering CMM, People CMM, etc. Since CMM is a reference model of matured practices in a specific discipline, so it becomes difficult to integrate these disciplines as per the requirements. This is why CMMI is used as it allows the integration of multiple disciplines as and when needed.

Objectives of CMMI :

- Fulfilling customer needs and expectations.
- Value creation for investors/stockholders.
- Market growth is increased.
- Improved quality of products and services.
- Enhanced reputation in Industry.

CMMI Representation – Staged and Continuous :

A representation allows an organization to pursue a different set of improvement objectives. There are two representations for CMMI :

Staged Representation :

- uses a pre-defined set of process areas to define improvement path.
- provides a sequence of improvements, where each part in the sequence serves as a foundation for the next.

- an improved path is defined by maturity level.
- maturity level describes the maturity of processes in organization.
- Staged CMMI representation allows comparison between different organizations for multiple maturity levels.

Continuous Representation :

- allows selection of specific process areas.
- uses capability levels that measures improvement of an individual process area.
- Continuous CMMI representation allows comparison between different organizations on a process-area-by-process-area basis.
- allows organizations to select processes which require more improvement.
- In this representation, order of improvement of various processes can be selected which allows the organizations to meet their objectives and eliminate risks.

Process Models

What is it?

- A process model provides a specific roadmap for software engineering work.
- It defines the flow of all activities, actions and tasks, the degree of iteration, the work products, and the organization of the work that must be done.

Who does it?

- Software engineers and their managers adapt a process model to their needs and then follow it.
- In addition, the people who have requested the software have a role to play in the process of defining, building, and testing it.

Why is it important?

- Because process provides stability, control, and organization to an activity that can, if left uncontrolled, become quite chaotic.
- However, a modern software engineering approach must be "agile."
- It must demand only those activities, controls, and work products that are appropriate for the project team and the product that is to be produced.

- **What are the steps?**
  - The process model provides you with the "steps" you'll need to perform disciplined software engineering work.

- **What is the work product?**
  - From the point of view of a software engineer, the work product is a customized description of the activities and tasks defined by the process.

- **How do I ensure that I've done it right?**

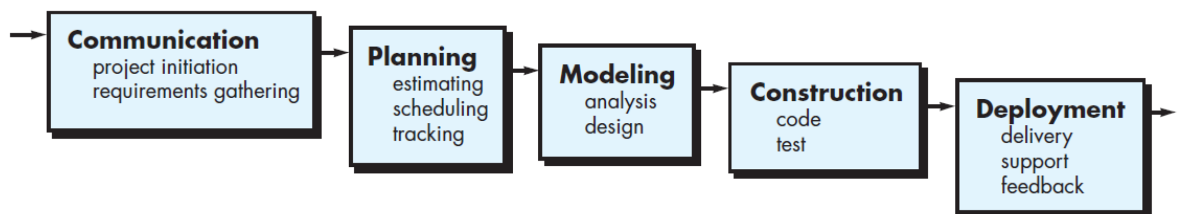
- There are a number of software process assessment mechanisms that enable organizations to determine the "maturity" of their software process.
- However, the quality, timeliness, and long-term viability of the product you build are the best indicators of the efficacy of the process that you use

- 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.
- Different models are:
  - Waterfall model
  - The RAD model
  - The Evolutionary software process models
    - Prototyping model
    - The Spiral model
  - Specialized process model.

Process Models-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.

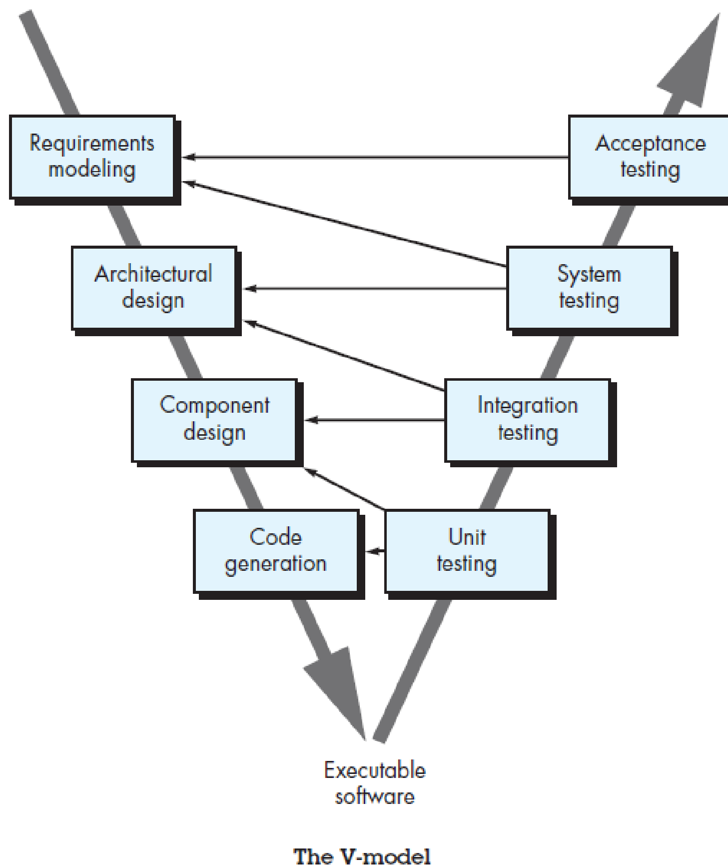
The waterfall model



-
- The waterfall model, sometimes called the classic life cycle , suggests a systematic, sequential approach to software development that begins with customer specification of requirements and progresses through planning, modelling, 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.
- The V-model 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 moves down the left side.
- 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.

Process Models-Waterfall Model



The V-model

The waterfall model is the oldest paradigm for software engineering.

But it has the following disadvantages, that is the reasons why the waterfall model sometimes fails:

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

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

3. The customer must have patience. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

Today, software work is fast paced and subject to a never-ending stream of changes (to features, functions, and information content).

The waterfall model is often inappropriate for such work. 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.

Process Models-The RAD MODEL

Rapid application development (RAD) is an incremental software development process model that emphasizes an extremely short development cycle.

The RAD model is a "high-speed" adaptation of the linear sequential model in which rapid development is achieved by using component-based construction. If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a "fully functional system" within very short time periods (e.g., 60 to 90 days).

The RAD approach encompasses the following phases:

Business modeling.

- The information flow among business functions is modeled in a way that answers the following questions:
- What information drives the business process?
- What information is generated? Who generates it?
- Where does the information go?
- Who processes it?

Data modeling.

- The information flow defined as part of the business modeling phase is refined into a set of data objects that are needed to support the business.
- The characteristics (called *attributes*) of each object are identified and the relationships between these objects defined.

Process modeling.

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

Application generation.

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

Application generation.

- The term *fourth generation techniques* (4GT) encompasses a broad array of software tools that have one thing in common: each enables the software engineer to specify some characteristic of software at a high level.
- The tool then automatically generates source code based on the developer's specification.
- There is little debate that the higher the level at which software can be specified to a machine, the faster a program can be built.
- The 4GT paradigm for software engineering focuses on the ability to specify software using specialized language forms or a graphic notation that describes the problem to be solved in terms that the customer can understand.
- Currently, a software development environment that supports the 4GT paradigm includes some or all of the following tools: nonprocedural languages for database query, report generation, data manipulation, screen interaction and definition, code generation; high-level graphics capability; spreadsheet capability, and automated generation of HTML and similar languages used for Web-site creation using advanced software tools.

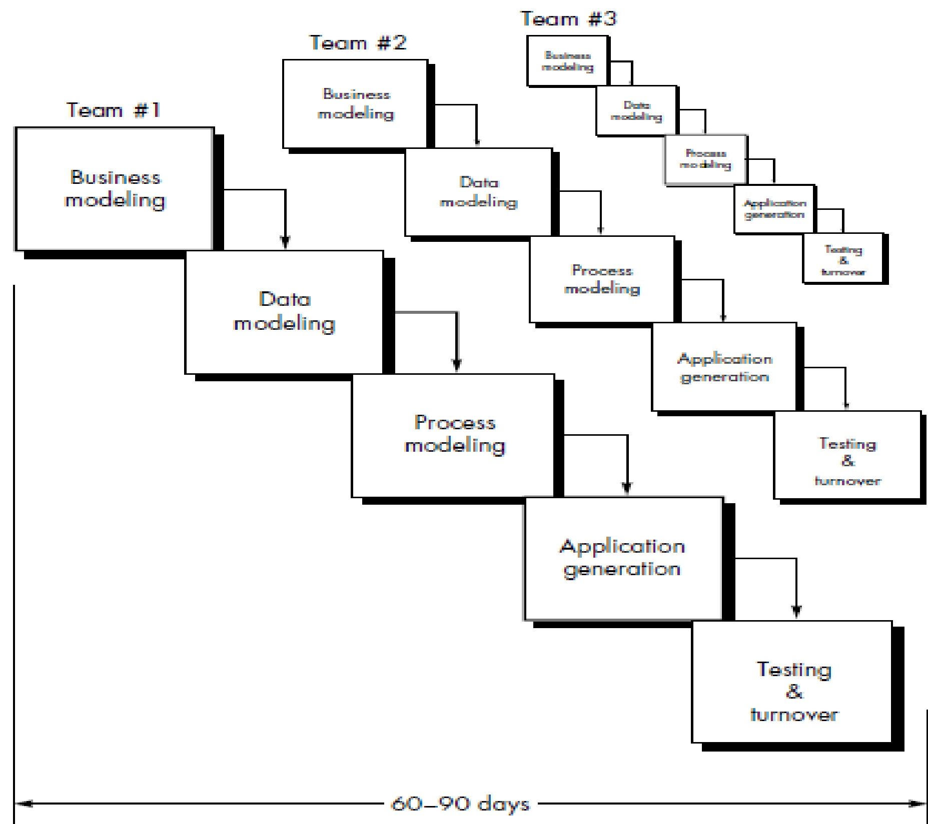
Testing and turnover.

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

Like all process models, the RAD approach has drawbacks:

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

The RAD model



Process Models-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 grows and changes.

Evolutionary models are iterative.

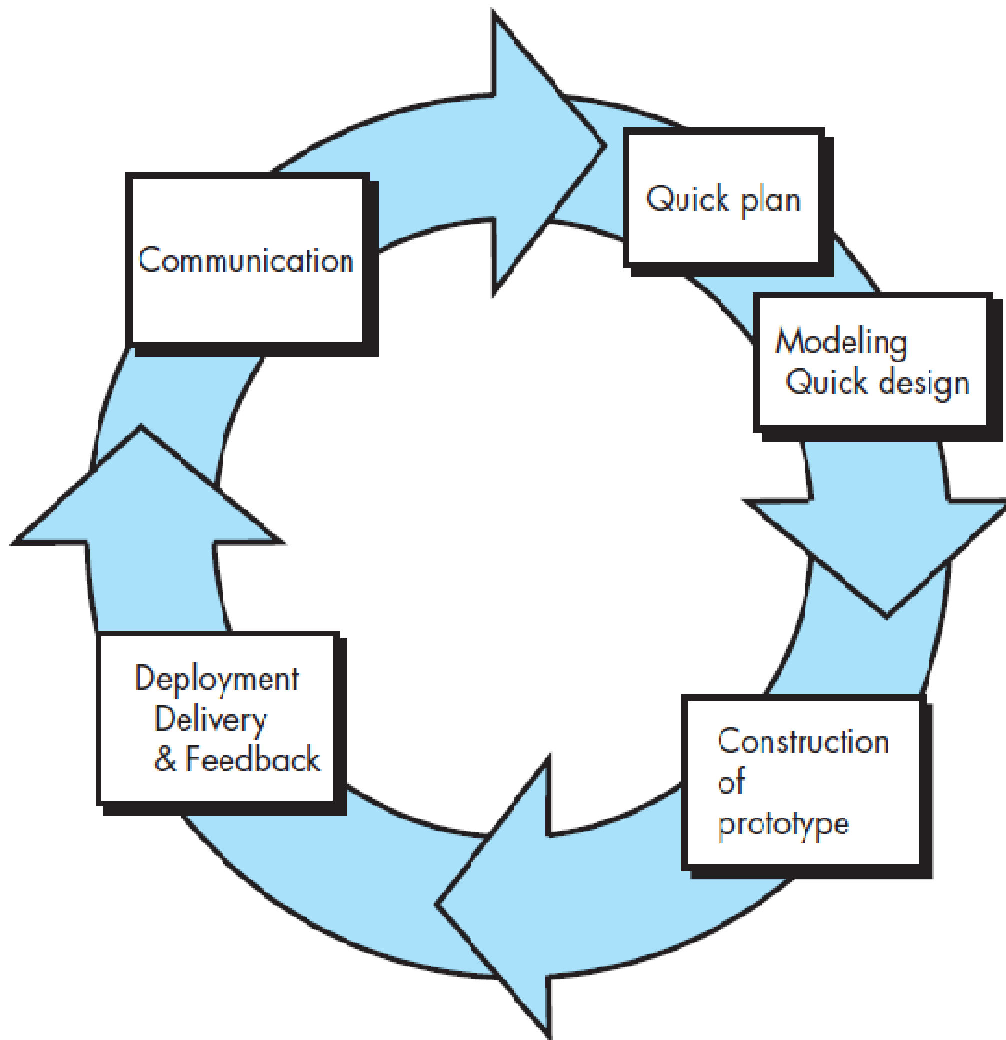
They are characterized in a manner that enables you to develop increasingly more complete versions of the software.

The two common evolutionary process models are:

- The Prototyping Model
- The Spiral Model

Prototyping .

- Many times 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 other process models.
- 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 that enable working programs to be generated quickly.



The prototyping paradigm

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:

1. 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.

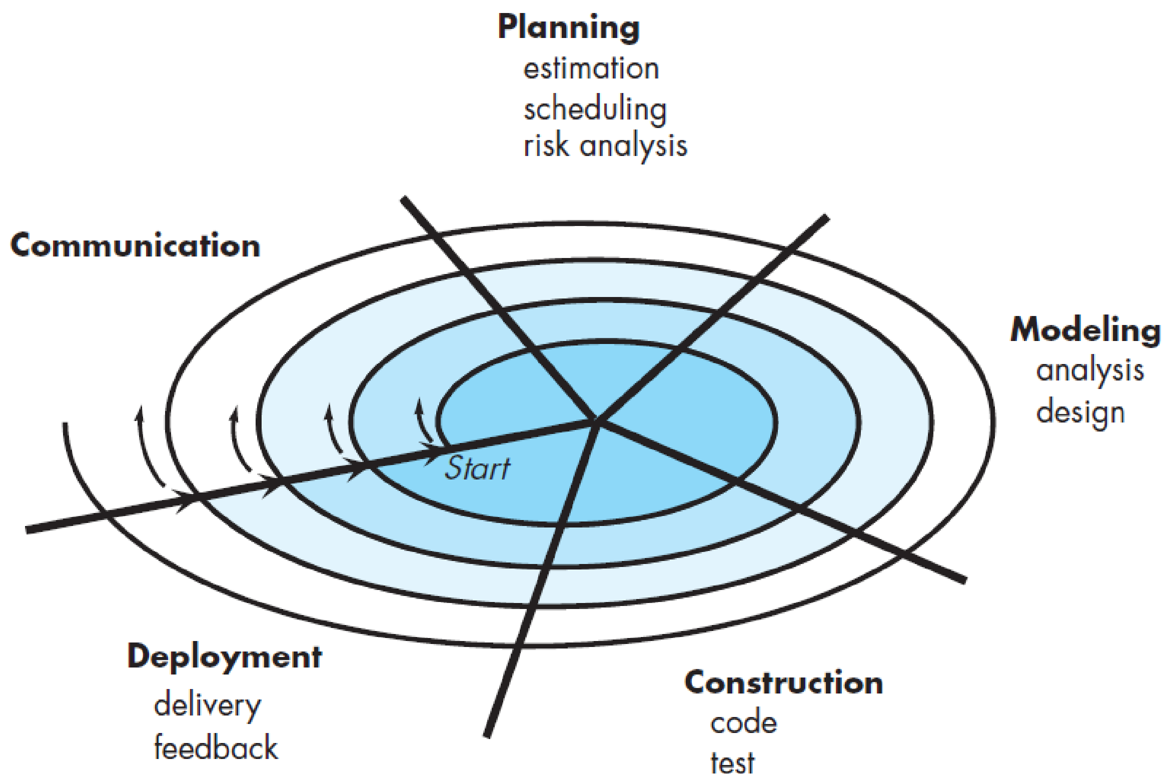
2. As a software engineer, you often make implementation compromises in order to get a prototype working quickly.

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.

The Spiral Model.

- 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.
- 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.
- Each of the framework activities represent one segment of the spiral path.
- 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.



A typical spiral model

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 iterations 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 in this model 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 occur.

Process Models-Specialized Process Models

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

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 comprises applications from pre-packaged 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 including a reduction in development cycle time and a reduction in project cost if component reuse becomes part of your organization’s culture.

- Examples of component models
    - EJB model (Enterprise Java Beans)
    - COM+ model (.NET model)
    - OMG Corba Component Model

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.
- Yet, there are concerns about its applicability in a business environment which include:
- • 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.

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"
- 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.