


<b>Gopalan College of Engineering and Management</b>			
<b>Department(s): Computer Science &amp; Engineering</b>			
<b>Semester: 06</b>	<b>Section(s): A</b>	<b>Lectures/week: 04</b>	
<b>Subject: Software Engineering and Project Management</b>		<b>Code: 21CS61</b>	
<b>Course Instructor(s): Dr. Manoj Challa</b>			
<b>Course duration: Apr 2024 – Aug 2024</b>			

### **Module 1:**

Software and Software Engineering: The nature of Software, The unique nature of WebApps, Software Engineering, The software Process, The software Engineering practice, The software myths, How it all starts

Process Models: A generic process model, Process assessment and improvement, Prescriptive process models, Waterfall model, Incremental process models, Evolutionary process models, Concurrent models, Specialized process models.

---

### **Defining Software:**

Software is: (1) instructions (computer programs) that when executed provide desired features, function, and performance;

(2) data structures that enable the programs to adequately manipulate information, and

(3) descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

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

**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 non-existent (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.

## Software doesn't "wear out."

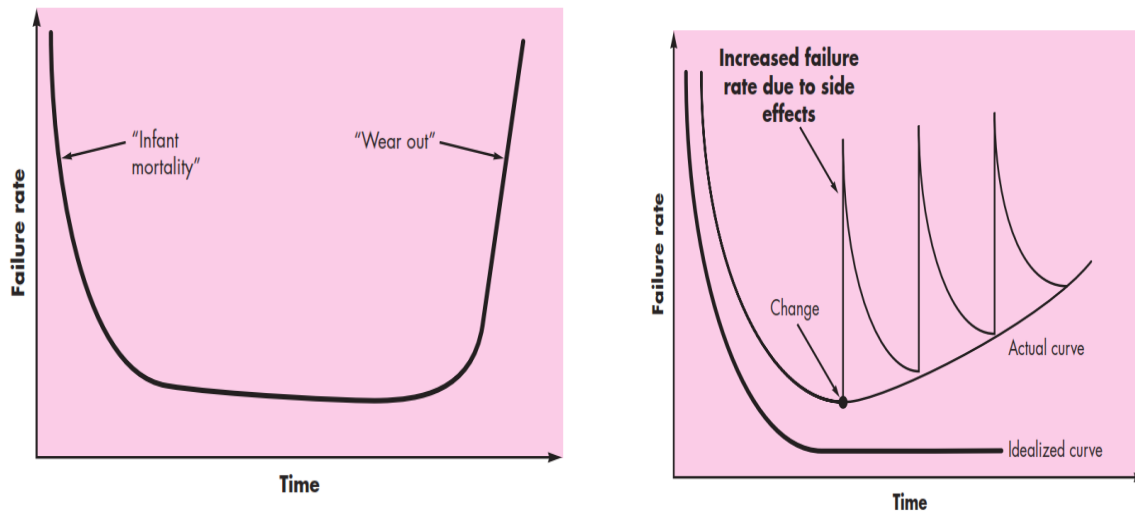


Figure 1 above depicts the failure rate as a function of time for hardware. The relationship, often called the “bathtub curve,” indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (hopefully, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear—the software doesn't wear out. But it does deteriorate! This seeming contradiction can best be explained by considering the actual curve in Figure 2. During its life,<sup>2</sup> 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 the “actual curve” (Figure 1.2). 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.

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

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, that is, the parts of the design that represent something new. In the hardware world, component reuse is a natural part of the engineering process. In the software world, it is something that 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. Modern 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.

### **Characteristics of software:**

- Reliability: The ability of the software to consistently perform its intended tasks without unexpected failures or errors.
- Usability: How easily and effectively users can interact with and navigate through the software.
- Efficiency: The optimal utilization of system resources to perform tasks on time.
- Maintainability: How easily and cost-effectively software can be modified, updated, or extended.
- Portability: The ability of software to run on different platforms or environments without requiring significant modifications.

### **Software Application Domains**

Seven broad categories of computer software present continuing challenges for software engineers:

1. System software—a collection 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.
2. Application software—stand-alone 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).
3. Engineering/scientific software—has been characterized by “number crunching” algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.
4. 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., key pad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).
5. 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, and personal and business financial applications).

6. Web applications—called “WebApps,” this network-centric software category spans 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 Web 2.0 emerges, 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.
7. Artificial intelligence software—makes use of nonnumerical 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.
8. Open-world computing—the rapid growth of wireless networking may soon lead to true pervasive, distributed computing. The challenge for software engineers will be to develop systems and application software that will allow mobile devices, personal computers, and enterprise systems to communicate across vast networks.
9. 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 a benefit to targeted end-user markets worldwide.
10. 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 many people can contribute to its development. 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.

### **The Unique nature of the web Apps**

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.

WebApps are one of a number of distinct software categories. And yet, it can be argued that WebApps are different. Powell [Pow98] suggests that Web-based systems and applications “involve a mixture between print publishing and software development, between marketing and computing, between internal communications and external relations, and between art and technology.”

The following attributes are encountered in the vast majority of WebApps.

- **Network intensiveness.** A WebApp resides on a network and must serve the needs of a diverse community of clients. The network may enable worldwide access and communication (i.e., the Internet) or more limited access and communication (e.g., a corporate Intranet).
- **Concurrency.** A large number of users may access the WebApp at one time. In many cases, the patterns of usage among end users will vary greatly. Unpredictable load. The number of users of the WebApp may vary by orders of magnitude from day to day. One hundred users may show up on Monday; 10,000 may use the system on Thursday.

- **Performance.** If a WebApp user must wait too long (for access, for serverside processing, for client-side formatting and display), he or she may decide to go elsewhere.
- **Availability.** Although expectation of 100 percent availability is unreasonable, users of popular WebApps often demand access on a 24/7/365 basis. Users in Australia or Asia might demand access during times when traditional domestic software applications in North America might be taken off-line for maintenance.
- **Data driven.** The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end user. In addition, WebApps are commonly used to access information that exists on databases that are not an integral part of the Web-based environment (e.g., e-commerce or financial applications).
- **Content sensitive.** The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp.
- **Continuous evolution.** Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously. It is not unusual for some WebApps (specifically, their content) to be updated on a minute-by-minute schedule or for content to be independently computed for each request.

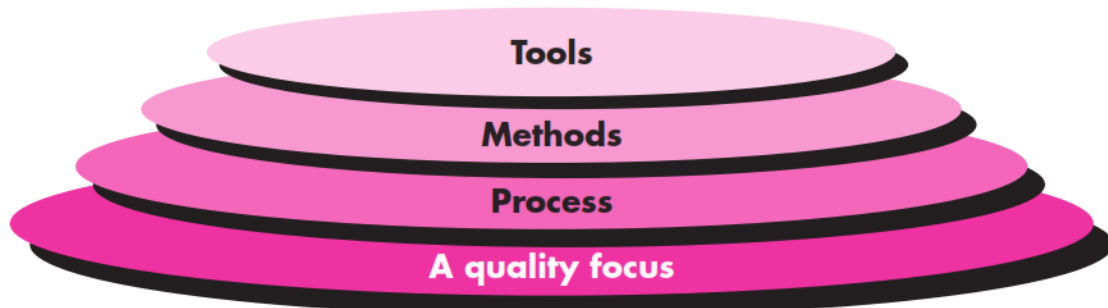
### **Software Engineering**

- Software has become deeply embedded in virtually every aspect of our lives, and as a consequence, the number of people who have an interest in the features and functions provided by a specific application has grown dramatically. When a new application or embedded system is to be built, many voices must be heard. And it sometimes seems that each of them has a slightly different idea of what software features and functions should be delivered. *It follows that a concerted effort should be made to understand the problem before a software solution is developed.*
- The information technology requirements demanded by individuals, businesses, and governments grow increasingly complex with each passing year. Large teams of people now create computer programs that were once built by a single individual. Sophisticated software that was once implemented in a predictable, self-contained, computing environment is now embedded inside everything from consumer electronics to medical devices to weapons systems. The complexity of these new computer-based systems and products demands careful attention to the interactions of all system elements. *It follows that design becomes a pivotal activity.*
- Individuals, businesses, and governments increasingly rely on software for strategic and tactical decision making as well as day-to-day operations and control. If the software fails, people and major enterprises can experience anything from minor inconvenience to catastrophic failures. *It follows that software should exhibit high quality.*

Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

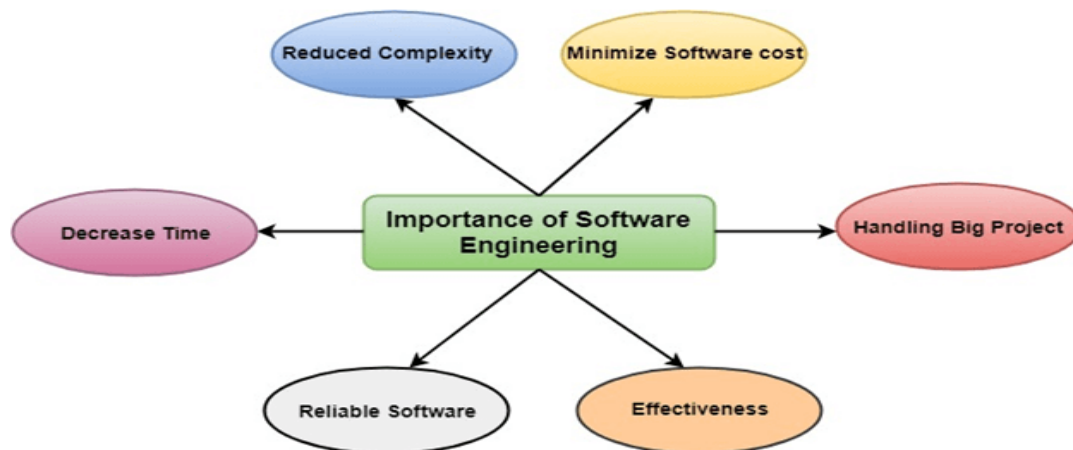
Software engineering is a layered technology. Referring to Figure 1.3, any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total quality management, Six Sigma, and similar philosophies foster a continuous

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.

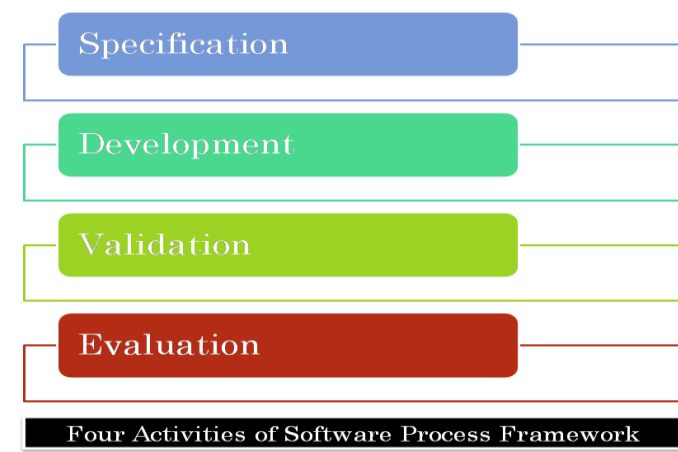


### Why is Software Engineering required?

- To manage Large software
- For more Scalability
- Cost Management
- To manage the dynamic nature of software
- For better quality Management



### Software Process





A *process* is a collection of activities, actions, and tasks that are performed when some work product is to be created. An *activity* strives to achieve a broad objective (e.g., communication with stakeholders) and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied. An *action* (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural design model). A *task* focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

- **Software specification**, where customers and engineers define the software that is to be produced and the constraints on its operation.
- **Software development**, where the software is designed and programmed.
- **Software validation**, where the software is checked to ensure that it is what the customer requires.
- **Software evaluation**, where the software is modified to reflect changing customer and market requirements.

### Software process model -

- **A workflow model:** This shows the series of activities in the process along with their inputs, outputs and dependencies. The activities in this model perform human actions.
- **A dataflow or activity model:** This represents the process as a set of activities, each of which carries out some data transformations. It shows how the input to the process, such as a specification is converted to an output such as a design.
- **A role/action model:** This means the roles of the people involved in the software process and the activities for which they are responsible.

### Software Crisis -

- **Size:** Software is becoming more expensive and more complex with the growing complexity and expectation out of software. For example, the code in the consumer product is doubling every couple of years.
- **Quality:** Many software products have poor quality, i.e., the software products defects after putting into use due to ineffective testing technique. For example, Software testing typically finds 25 errors per 1000 lines of code.
- **Cost:** Software development is costly i.e. in terms of time taken to develop and the money involved. For example, Development of the Advanced Automation System cost over \$700 per lines of code.
- **Delayed Delivery:** Serious schedule overruns are common. Very often the software takes longer than the estimated time to develop, which in turn leads to cost shooting up. For example, one in four large-scale development projects is never completed.

### The software Engineering practice

Generic framework activities—**communication, planning, modeling, construction, and deployment**—and umbrella activities establish a skeleton architecture for software engineering work. But how does the practice of software engineering fit in? In the sections that follow, you'll gain a basic understanding of the generic concepts and principles that apply to framework activities. Essence of problem solving, and consequently, the essence of software engineering practice:

- *Understand the problem* (communication and analysis).
- *Plan a solution* (modeling and software design).
- *Carry out the plan* (code generation).
- *Examine the result for accuracy* (testing and quality assurance).

### General Principles

- **The First Principle: *The Reason It All Exists*** - A software system exists for one reason: *to provide value to its users*. All decisions should be made with this in mind. Before specifying a system requirement, before noting a piece of system functionality, before determining the hardware platforms or development processes, ask yourself questions such as: “Does this add real value to the system?” If the answer is “no,” don’t do it. All other principles support this one.
- **The Second Principle: *KISS (Keep It Simple, Stupid!)*** - Software design is not a haphazard process. There are many factors to consider in any design effort. *All design should be as simple as possible, but no simpler*. This facilitates having a more easily understood and easily maintained system. This is not to say that features, even internal features, should be discarded in the name of simplicity. Indeed, the more elegant designs are usually the more simple ones. Simple also does not mean “quick and dirty.” In fact, it often takes a lot of thought and work over multiple iterations to simplify. The payoff is software that is more maintainable and less error-prone.
- **The Third Principle: *Maintain the Vision*** - *A clear vision is essential to the success of a software project*. Without one, a project almost unfailingly ends up being “of two [or more] minds” about itself. Without conceptual integrity, a system threatens to become a patchwork of in-compatible designs, held together by the wrong kind of screws. . . . Compromising the architectural vision of a software system weakens and will eventually break even the well-designed systems. Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.
- **The Fourth Principle: *What You Produce, Others Will Consume*** - Seldom is an industrial-strength software system constructed and used in a vacuum. In some way or other, someone else will use, maintain, document, or otherwise depend on being able to understand your system. So, *always specify, design, and implement knowing someone else will have to understand what you are doing*. The audience for any product of software development is potentially large. Specify with an eye to the users. Design, keeping the implementers in mind. Code with concern for those that must maintain and extend the system. Someone may have to debug the code you write, and that makes them a user of your code. Making their job easier adds value to the system.
- **The Fifth Principle: *Be Open to the Future*** - A system with a long lifetime has more value. In today’s computing environments, where specifications change on a moment’s notice and hardware platforms are obsolete just a few months old, software lifetimes are typically measured in months instead of years. However, true “industrial-strength” software systems must endure far longer. To do this successfully, these systems must be ready to adapt to these and other changes. Systems that do this successfully are those that have been designed this way from the start. *Never design yourself into a corner*.

### Software Myths in Software Engineering

**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 [Bro95]: “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 wellcoordinated 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 (described in Chapter 15) 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.

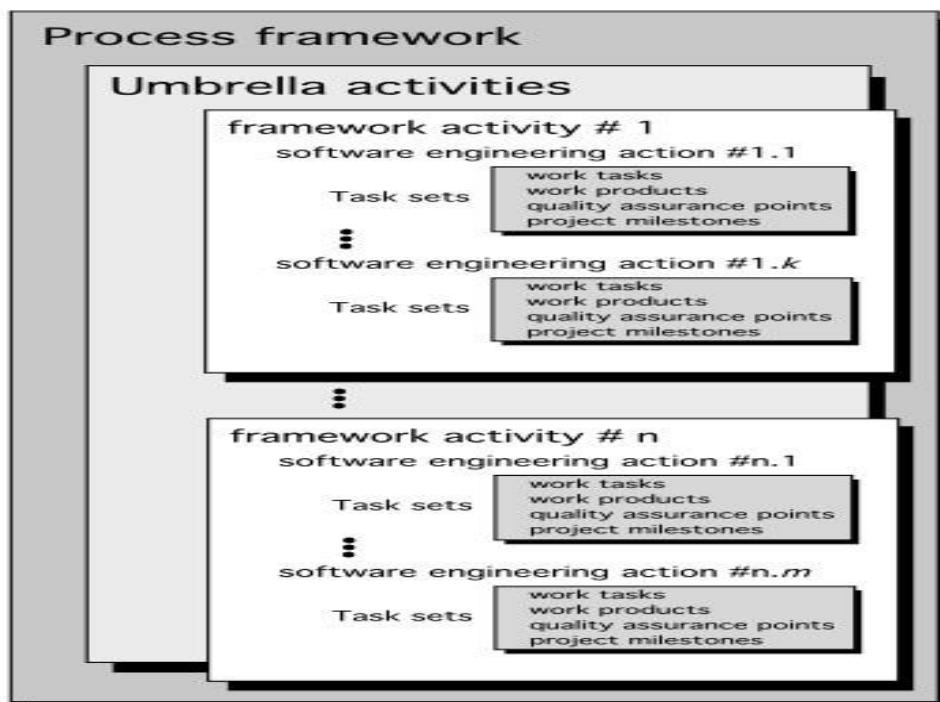
### How to Avoid Software Myths

Here are some steps you can take to avoid falling victim to common software myths:

- **Stay Informed:** Keep up to date with the latest trends, practices, and developments in the field of software engineering.
- **Continuous Learning:** Invest in ongoing education and professional development. Software engineering is an evolving field, and staying current is essential.
- **Data-Driven Decision-Making:** Make decisions based on data, evidence, and real-world experiences rather than relying on common misconceptions.
- **Testing and Validation:** Don't rely solely on assumptions. Test your software, gather data, and validate your ideas to confirm their accuracy.
- **Educate Team Members:** Ensure that everyone on your development team is aware of common software myths and is committed to avoiding them.
- **Risk Assessment:** When dealing with software development decisions, conduct risk assessments to identify potential pitfalls and myths that might affect the project.

### A Generic Process Model

Software process



## **Process Flow**

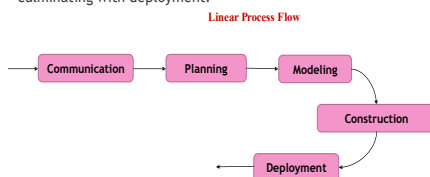
Process Flow shows how the framework activities and the actions and the tasks that occur within each activity are organized with respect to sequence and time.

### **Types of process flows in SDLC**

1. Linear Process Flow
2. Iterative Process Flow
3. Evolutionary Process Flow
4. Parallel Process Flow

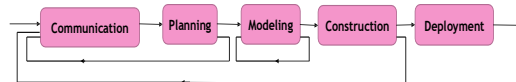
#### **Linear Process Flow**

A linear process flow executes each of the five framework activities in sequence, beginning with communication and culminating with deployment.



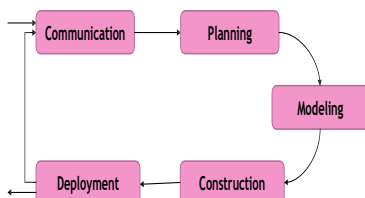
#### **Iterative Process Flow**

An iterative process flow repeats one or more of the activities before proceeding to the next.



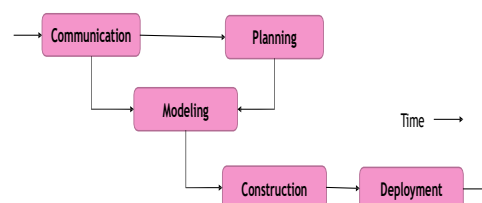
#### **Evolutionary Process Flow**

An evolutionary process flow executes the activities in a "circular" manner.



#### **Parallel Process Flow**

A parallel process flow executes one or more activities in parallel with other activities.



## **Process Assessment and Improvement**

- Software Process Assessment is a disciplined and organized examination of the software process which is being used by any organization based on the process model.
- The Software Process Assessment includes many fields and parts like identification and characterization of current practices, the ability of current practices to control or avoid significant causes of poor (software) quality, cost, schedule and identifying areas of strengths and weaknesses of the software.

**A software assessment (or audit) can be of three types –**

- A self-assessment (first-party assessment) is performed internally by an organization's own personnel.
- A second-party assessment is performed by an external assessment team or the organization is assessed by a customer.
- A third-party assessment is performed by an external party or (e.g., a supplier being assessed by a third party to verify its ability to enter contracts with a customer).
- A number of different approaches to software process assessment and improvement 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 five phases: 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; uses the SEI CMM as the basis for the assessment [Dun01].
- SPICE (ISO/IEC15504)—a standard that 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 [ISO08].
- ISO 9001:2000 for Software—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 [Ant06].

**SDLC**



Stage1: Planning and requirement analysis

- Requirement Analysis is the most important and necessary stage in SDLC.

Stage2: Defining Requirements

- Once the requirement analysis is done, the next stage is to certainly represent and document the software requirements and get them accepted from the project stakeholders.
- This is accomplished through "SRS"- Software Requirement Specification document which contains all the product requirements to be constructed and developed during the project life cycle.

Stage3: Designing the Software

- The next phase is about to bring down all the knowledge of requirements, analysis, and design of the software project.

- This phase is the product of the last two, like inputs from the customer and requirement gathering.

**Stage4: Developing the project**

- In this phase of SDLC, the actual development begins, and the programming is built.
- The implementation of design begins concerning writing code.
- Developers have to follow the coding guidelines described by their management and programming tools like compilers, interpreters, debuggers, etc. are used to develop and implement the code.

**Stage5: Testing**

- After the code is generated, it is tested against the requirements to make sure that the products are solving the needs addressed and gathered during the requirements stage.
- During this stage, unit testing, integration testing, system testing, acceptance testing are done.

**Stage6: Deployment**

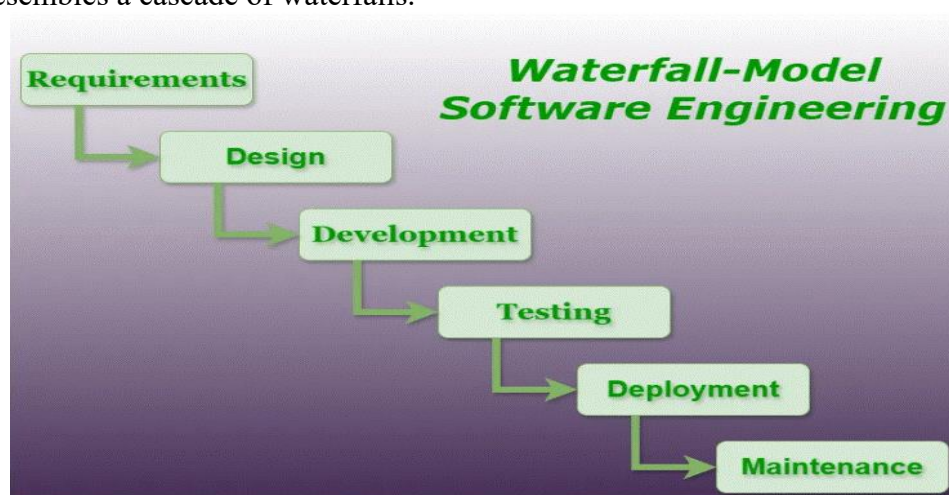
- Once the software is certified, and no bugs or errors are stated, then it is deployed.
- Then based on the assessment, the software may be released as it is or with suggested enhancement in the object segment.

**Stage7: Maintenance**

- Once when the client starts using the developed systems, then the real issues come up and requirements to be solved from time to time.
- This procedure where the care is taken for the developed product is known as maintenance.

**Waterfall model**

- Winston R introduced the Waterfall Model in 1970.
- This model has five phases: Requirements analysis and specification, design, implementation, and unit testing, integration and system testing, and operation and maintenance.
- The steps always follow in this order and do not overlap.
- The developer must complete every phase before the next phase begins.
- This model is named "Waterfall Model", because its diagrammatic representation resembles a cascade of waterfalls.



**Requirements analysis and specification phase:**

- The aim of this phase is to understand the exact requirements of the customer and to document them properly.

- Both the customer and the software developer work together so as to document all the functions, performance, and interfacing requirement of the software.
- It describes the "what" of the system to be produced and not "how."
- In this phase, a large document called Software Requirement Specification (SRS) document is created which contained a detailed description of what the system will do in the common language.

#### **Design Phase:**

- This phase aims to transform the requirements gathered in the SRS into a suitable form which permits further coding in a programming language.
- It defines the overall software architecture together with high level and detailed design.
- All this work is documented as a Software Design Document (SDD).

#### **Implementation and unit testing:**

- During this phase, design is implemented.
- If the SDD is complete, the implementation or coding phase proceeds smoothly, because all the information needed by software developers is contained in the SDD.
- During testing, the code is thoroughly examined and modified.
- Small modules are tested in isolation initially. After that these modules are tested by writing some overhead code to check the interaction between these modules and the flow of intermediate output.

#### **Integration and System Testing:**

- This phase is highly crucial as the quality of the end product is determined by the effectiveness of the testing carried out.
- The better output will lead to satisfied customers, lower maintenance costs, and accurate results.
- Unit testing determines the efficiency of individual modules.
- However, in this phase, the modules are tested for their interactions with each other and with the system.

#### **Operation and maintenance phase:**

- Maintenance is the task performed by every user once the software has been delivered to the customer, installed, and operational.

#### **When to use SDLC Waterfall Model?**

- When the requirements are constant and not changed regularly.
- A project is short
- The situation is calm
- Where the tools and technology used is consistent and is not changing
- When resources are well prepared and are available to use.

#### **Advantages of Waterfall model**

- This model is simple to implement also the number of resources that are required for it is minimal.
- The requirements are simple and explicitly declared; they remain unchanged during the entire project development.
- The start and end points for each phase is fixed, which makes it easy to cover progress.
- The release date for the complete product, as well as its final cost, can be determined before development.
- It gives easy to control and clarity for the customer due to a strict reporting system.



### Disadvantages of Waterfall model

In this model, the risk factor is higher, so this model is not suitable for more significant and complex projects.

- This model cannot accept the changes in requirements during development.
- It becomes tough to go back to the phase. For example, if the application has now shifted to the coding phase, and there is a change in requirement, It becomes tough to go back and change it.
- Since the testing done at a later stage, it does not allow identifying the challenges and risks in the earlier phase, so the risk reduction strategy is difficult to prepare.

### Evolutionary Process Models

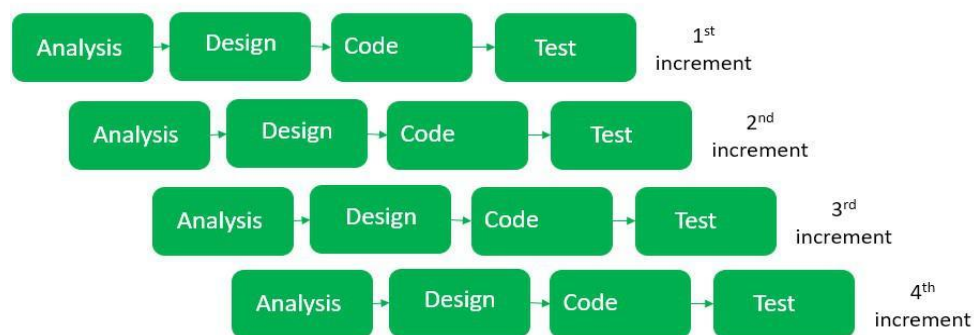
- The evolutionary model is based on the concept of making an initial product and then evolving the software product over time with iterative and incremental approaches with proper feedback.
- In this type of model, the product will go through several iterations and come up when the final product is built through multiple iterations.

### Types of Evolutionary Process Models

1. Incremental Model
2. Iterative Model
3. Spiral Model

### Incremental Process Model

- Incremental Model is a process of software development where requirements divided into multiple standalone modules of the software development cycle.
- In this model, each module goes through the requirements, design, implementation and testing phases.
- Every subsequent release of the module adds function to the previous release. The process continues until the complete system achieved.



**1. Requirement analysis:** System functional requirements are understood by the requirement analysis team. To develop the software under the incremental model, this phase performs a crucial role.

**2. Design & Development:** Design of the system functionality and the development method are finished with success. When software develops new practicality, the incremental model uses style and development phase.

**3. Testing:** The testing phase checks the performance of each existing function as well as additional functionality. In the testing phase, the various methods are used to test the behavior of each task.

**4. Implementation:** Implementation phase enables the coding phase of the development system. It involves the final coding that design in the designing and development phase and tests the functionality in the testing phase. After completion of this phase, the number of the product working is enhanced and upgraded up to the final system product

#### When we use the Incremental Model?

- When the requirements are superior.
- A project has a lengthy development schedule.
- When Software team are not very well skilled or trained.
- When the customer demands a quick release of the product.

#### Advantage of Incremental Model

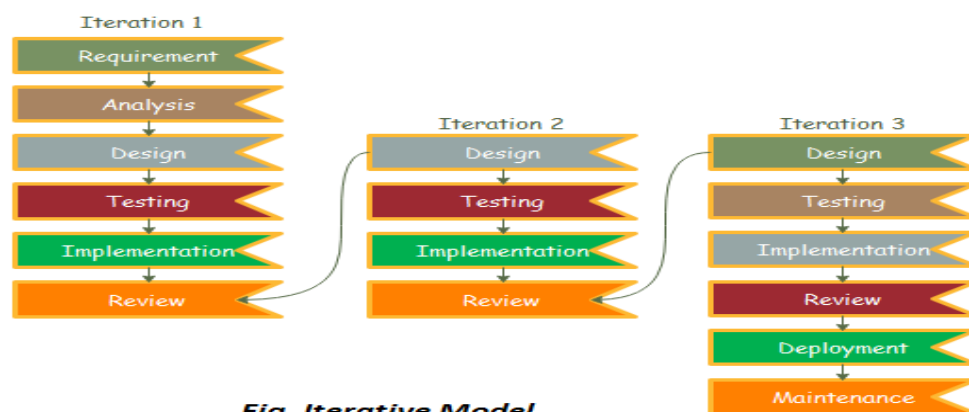
- Errors are easy to be recognized.
- Easier to test and debug
- More flexible.
- Simple to manage risk because it handled during its iteration.
- The Client gets important functionality early.

#### Disadvantage of Incremental Model

- Need for good planning
- Total Cost is high.
- Well defined module interfaces are needed.

#### Iterative Model

- In this Model, you can start with some of the software specifications and develop the first version of the software.
- After the first version if there is a need to change the software, then a new version of the software is created with a new iteration.
- Every release of the Iterative Model finishes in an exact and fixed period that is called iteration.
- The Iterative Model allows the accessing earlier phases, in which the variations made respectively. The final output of the project renewed at the end of the Software Development Life Cycle (SDLC) process.



**Fig. Iterative Model**

1. Requirement gathering & analysis: In this phase, requirements are gathered from customers and check by an analyst whether requirements will fulfil or not. Analyst checks that need will achieve within budget or not. After all of this, the software team skips to the next phase.
2. Design: In the design phase, team design the software by the different diagrams like Data Flow diagram, activity diagram, class diagram, state transition diagram, etc.
3. Implementation: In the implementation, requirements are written in the coding language and transformed into computer programmes which are called Software.
4. Testing: After completing the coding phase, software testing starts using different test methods. There are many test methods, but the most common are white box, black box, and grey box test methods.

### **When to use the Iterative Model?**

1. When requirements are defined clearly and easy to understand.
2. When the software application is large.
3. When there is a requirement of changes in future.

### **Advantage of Iterative Model**

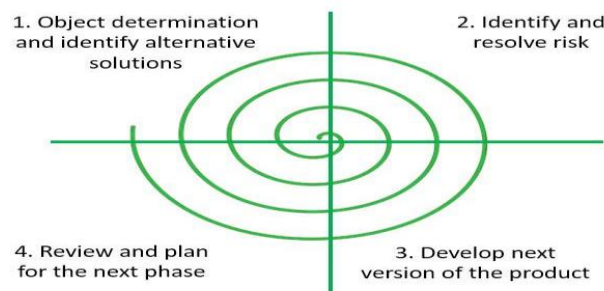
1. Testing and debugging during smaller iteration is easy.
2. It is easily acceptable to ever-changing needs of the project.
3. Risks are identified and resolved during iteration.
4. Limited time spent on documentation and extra time on designing.

### **Disadvantage of Iterative Model**

1. It is not suitable for smaller projects.
2. More Resources may be required.
3. Design can be changed again and again because of imperfect requirements.
4. Requirement changes can cause over budget.
5. Project completion date not confirmed because of changing requirements.

### **Spiral Model**

- The spiral model is a combination of increment and iterative models and in this, we focused on risk handling along with developing the project with the incremental and iterative approach, producing the output quickly as well as it is good for big projects.
- The software is created through multiple iterations using a spiral approach.
- Later on, after successive development the final product will develop, and the customer interaction is there so the chances of error get reduced.



1. Objectives determination and identify alternative solutions: Requirements are gathered from the customers and the objectives are identified, elaborated, and analyzed at the start of every phase. Then alternative solutions possible for the phase are proposed in this quadrant.
2. Identify and resolve Risks: During the second quadrant, all the possible solutions are evaluated to select the best possible solution. Then the risks associated with that solution are identified and the risks are resolved using the best possible strategy. At the end of this quadrant, the Prototype is built for the best possible solution.
3. Develop the next version of the Product: Identified features are developed and verified through testing.
4. Review and plan for the next Phase: The Customers evaluate the so-far developed version of the software. In the end, planning for the next phase is started.

### When to use Spiral Model?

- When the project is large
- When requirements are unclear and complex
- When changes may require at any time
- Large and high budget projects

### Advantages

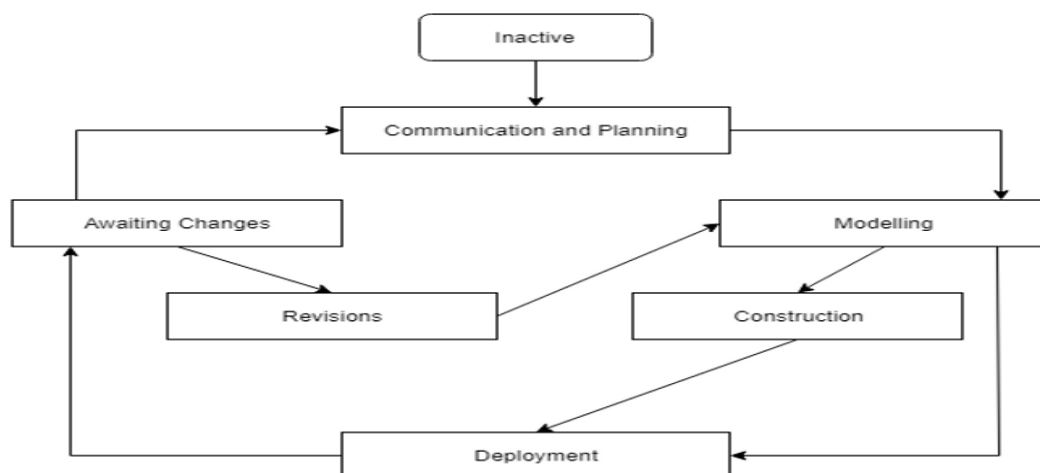
- High amount of risk analysis
- Useful for large and mission-critical projects.

### Disadvantages

- Can be a costly model to use.
- Risk analysis needed highly particular expertise
- Doesn't work well for smaller projects.

### Concurrent development model

- Concurrent development model is quite different from the incremental model.
- The incremental model is performed sequentially while the concurrent development model is performed simultaneously.
- Instead of having a linear process flow, development processes create a net-like workflow.



- In the concurrent development model, each activity has a state and occurs concurrently.

- Transitions take place when an activity is triggered to move to another state.
- Therefore, concurrent development model can be defined as a series of activities, tasks, and associated states.

#### **Advantages of Concurrent model:**

1. High flexibility
2. New features and designs may be further added
3. Each component occurs simultaneously

#### **Disadvantages of Concurrent model:**

1. Require a good teamwork
2. Least-planned design
3. Frequent errors and miscommunication during a process may trigger disruption in other states

#### **Specialised 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. There are 3 types of specialized process models:

1. Component Based Development
2. Formal Methods Model
3. Aspect Oriented Software development

**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 component. Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or objectoriented 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:

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 benefit

**2. Formal Methods Model :** The formal methods model encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable 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, 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. The formal methods model offers the promise of defect-free software. There are some of the disadvantages too:

- 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

**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 and then constructed within the context of a system architecture. As modern computer-based systems become more sophisticated certain concerns span the entire architecture. Some concerns are high-level properties of a system, Other concerns affect functions, while others are systemic. 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. 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. It is essential to instantiate asynchronous communication between the software process activities applied to the engineering and construction of aspects and components

### Software testing

- Software testing can be stated as the process of verifying and validating whether a software or application is bug-free, meets the technical requirements as guided by its design and development, and meets the user requirements effectively and efficiently by handling all the exceptional and boundary cases.

Software testing can be divided into two steps:

1. Verification: It refers to the set of tasks that ensure that the software correctly implements a specific function. It means “Are we building the product right?”.
2. Validation: It refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. It means “Are we building the right product?”.

### Importance of Software Testing

- Defects can be identified early: Software testing is important because if there are any bugs they can be identified early and can be fixed before the delivery of the software.
- Improves quality of software: Software Testing uncovers the defects in the software, and fixing them improves the quality of the software.
- Increased customer satisfaction: Software testing ensures reliability, security, and high performance which results in saving time, costs, and customer satisfaction.
- Helps with scalability: Software testing type non-functional testing helps to identify the scalability issues and the point where an application might stop working.



- Saves time and money: After the application is launched it will be very difficult to trace and resolve the issues, as performing this activity will incur more costs and time. Thus, it is better to conduct software testing at regular intervals during software development.

