Unit-1
Introduction to Software Engineering

1.1 Definition of software engineering

1.2 The evolving role of software

1.3 Changing nature of software

1.4 Characteristics of Software

1.5 A generic view of software engineering

1.6 Software engineering-layered technology

Software:

- Software is a set of instructions, data or programs used to operate computers and execute specific tasks.
- It is the opposite of hardware, which describes the physical aspects of a computer.
- It acts as the intangible counterpart to the hardware and is essential for making hardware functional.

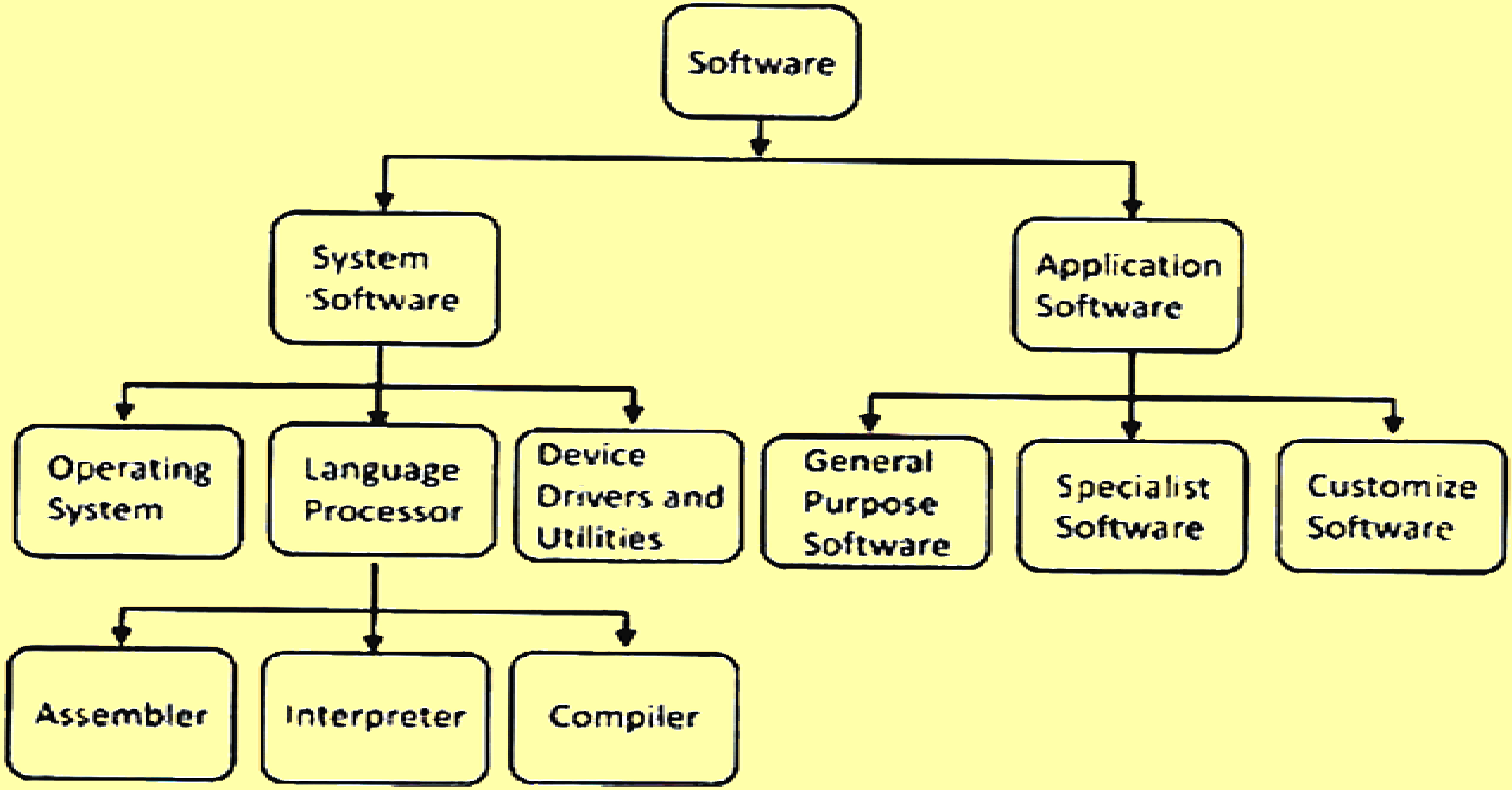
Software is categorized into two main types:

1. **System Software:**

   System software is the foundation upon which all other software runs. It manages the hardware resources of a computer and provides essential services to other software.

2. **Application Software:**

   Application software is designed to perform specific tasks for users. It interacts with system software to carry out its functions.


Software
System Software
Application Software
Operating System
Language Processor
Device Drivers and Utilities
General Purpose Software
Specialist Software
Customize Software
Assembler
Interpreter
Compiler

System Software:

- These software programs are designed to run a computer's application programs and hardware. System software coordinates the activities and functions of the hardware and software.
- In addition, it controls the operations of the computer hardware and provides an environment or platform for all the other types of software to work in.
- The OS is the best example of system software; it manages all the other computer programs.

Operating System:

- An **Operating System (OS)** is a type of system software that manages computer hardware, software, and resources, while providing services to users and applications.
- It acts as an intermediary between the hardware and the user, ensuring that computer resources are used efficiently and effectively.

Functions of an Operating System:

- ✓ Process Management
- ✓ Memory Management
- ✓ File System Management
- ✓ Device Management
- ✓ User Interface
- ✓ Security and Access Control
- ✓ Resource Allocation
- ✓ Error Detection and Handling

Language Processor:

- A **Language Processor** is a type of system software that translates high-level programming languages (understandable by humans) into machine language (understandable by computers).
- It bridges the gap between programming languages and the binary instructions that computer hardware executes.

Types of Language Processors:

- Assembler : Translates Assembly Language code into machine code.
- Compiler: Translates the entire high-level source code into machine code in one go.
- Interpreter: Translates and executes high-level code line by line.

Device Drivers:

- Device drivers are specialized software programs that enable the operating system to communicate with hardware devices.
- They act as a bridge between the hardware and the OS, translating OS instructions into commands that the hardware can execute and vice versa.

Utilities

- Utilities are system software designed to help analyze, configure, optimize, or maintain the computer system.
- Unlike device drivers, utilities provide additional functionality to enhance the system's performance and usability.

Types of Utilities:

- System Maintenance Utilities (Disk Cleanup, Defragmentation tools)
- Security Utilities (Antivirus Software, Firewall, Encryption tools)
- File Management Utilities (File compression tools, Backup software)
- Performance Monitoring Utilities (Task Manager, Resource Monitor)
- System Recovery Utilities (System Restore, Data recovery tools)
- Network Utilities (Ping, Trace-route, Bandwidth monitors)

Application Software:

- **Application software** refers to programs or a group of programs designed for end-users to perform specific tasks or functions.
- Unlike system software, which manages and operates computer hardware, application software is focused on user tasks such as creating documents, managing databases, editing photos, or browsing the internet.

General-Purpose Software

- General-purpose software refers to computer programs designed to perform a wide variety of tasks that are not specific to any one industry or user group.
- These applications are flexible, versatile, and commonly used in daily activities across various domains.
- Examples: Word Processing software, Spreadsheet Software, Presentation Software etc.

Specialized Software

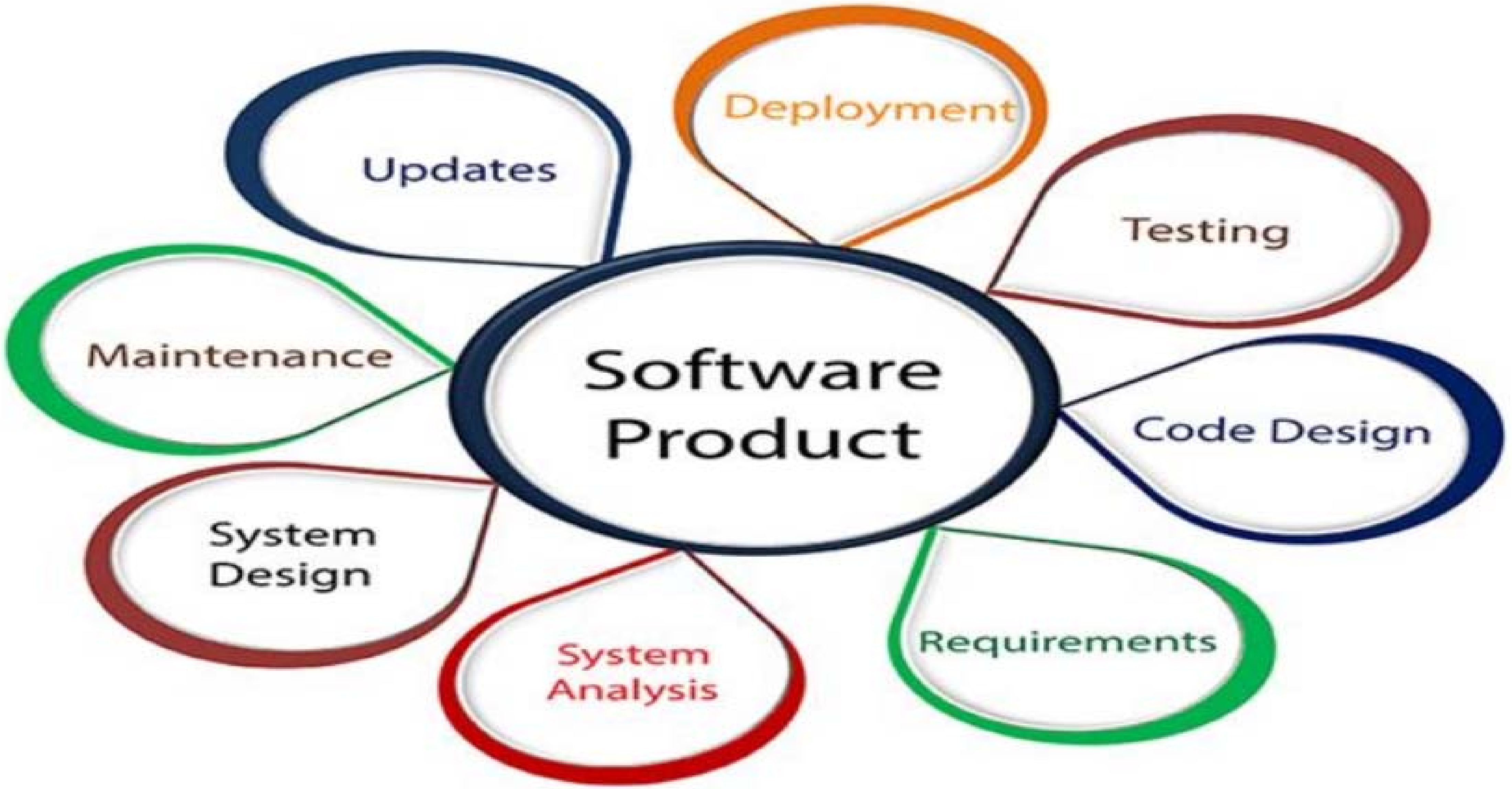
- Specialized software refers to applications designed to perform specific tasks or cater to the needs of a particular industry or field.
- Unlike general-purpose software, which serves broad functions, specialized software focuses on unique, detailed functionalities tailored for specific users or organizations.
- Example: AutoCAD, Tally, Blackboard etc.

Customized Software

- Customized software is tailor-made software developed to meet the unique needs and requirements of a specific organization, individual, or business.
- It is designed from scratch or adapted from existing software to align perfectly with the intended purpose.
- Examples: Hospital management system, Custom E-commerce platforms, Learning Management system etc.

Software Engineering:

- **Software Engineering** is a systematic approach to designing, developing, testing, deploying, and maintaining software applications.
- It applies principles of engineering, computer science, and mathematics to ensure the production of reliable, efficient, and maintainable software systems.
- Software Engineering provides a standard procedure to design and develop a software.
- The term software engineering is the product of two words, software, and engineering.
- The software is a collection of integrated programs.
- Engineering is the application of scientific and practical knowledge to invent, design, build, maintain, and improve frameworks, processes, etc


Software Product
Updates
Deployment
Testing
Code Design
Requirements
System Analysis
System Design
Maintenance

Reasons behind the popularity of software engineering

1. Rising Demand for Software Solutions

- The increasing dependence on software in every aspect of life—business operations, healthcare, education, entertainment, and communication—necessitates systematic development.
- Software engineering ensures the development of applications tailored to diverse needs

2. Systematic and Structured Approach

- Unlike ad-hoc(when necessary or needed) programming, software engineering uses well-defined methodologies and processes to manage complexity, reduce errors, and ensure reliability.
- Frameworks like SDLC, Agile, and DevOps ensure timely and efficient delivery of software

3. Cost Efficiency

- By following systematic processes, software engineering minimizes rework, reduces project costs, and prevents failure.
- Practices like **modular design** and **reusability** lower the cost of developing and maintaining software.

4. Scalability and Flexibility

- Software engineering makes it possible to design scalable systems that grow with user needs.
- Flexible designs accommodate future upgrades, changes, and integration with emerging technologies.

5. High Demand for Quality and Reliability

- Modern systems, especially in critical areas like banking, healthcare, and aviation, require software that is robust and error-free.
- Software engineering incorporates rigorous **testing** and **quality assurance** to ensure reliability.

6. Innovations in Technology

- Rapid advancements in AI, IoT, big data, and cloud computing demand systematic development approaches.
- Software engineering bridges the gap between cutting-edge technology and user-friendly applications.

7. Support for Large-Scale Projects

- Complex projects involving millions of users require efficient collaboration, design, and management—core principles of software engineering.
- Tools like **project management systems** and **version control** streamline such projects.

8. Globalization and Remote Accessibility

- Businesses today rely on software for remote operations, global communication, and digital services.
- Software engineering ensures these systems operate seamlessly and securely across borders.

9. Integration of Security

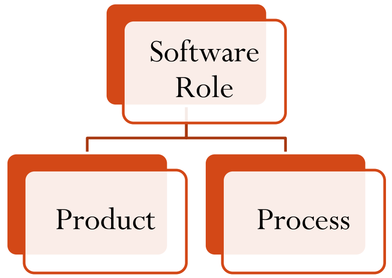
- With cyber threats on the rise, secure software design is essential.
- Software engineering integrates security practices like encryption, authentication, and vulnerability testing.

10. Continuous Evolution of User Needs

- As user expectations grow, software must evolve with features like user-friendly interfaces, personalization, and cross-platform support.
- Software engineering methodologies allow for ongoing maintenance and feature enhancements.

The evolving role of software:

- The role of software in **software engineering** has evolved significantly over time, reflecting advancements in technology, changes in user expectations, and the increasing complexity of systems.
- Software has dual role



1. Software as a product

→ Software as a product refers to the tangible deliverable created to meet user needs, solve specific problems, or enable functionality. Examples include applications, operating systems, and games.

Evolving Role

- **Customization**: Modern software products are designed to be user-centric, offering high levels of customization.
- **Cloud-based Solutions**: Shift from traditional on-premise software to cloud-based services (e.g., SaaS(Software as a Service)) for flexibility and scalability.
- **Automation**: Incorporation of AI and machine learning to automate tasks and make products smarter.
- **Global Access**: With mobile-first and cross-platform designs, software products now cater to users globally across various devices.
- **Security**: With increasing cybersecurity threats, robust security features have become essential in software products.

2. Software as a Process

→Software as a process encompasses the methods, practices, and tools used to design, develop, test, and maintain software.

Evolving Role

- **Agile Development**: Adoption of Agile methodologies for iterative development, focusing on flexibility and collaboration.
- **DevOps Integration**: Combining development and operations to streamline deployment and improve delivery cycles.
- **Automation in Development**: Use of CI/CD pipelines, automated testing, and deployment to enhance efficiency.
- **Quality Assurance**: Strong emphasis on ensuring software reliability through rigorous(diligent) testing and feedback loops.
- **Data-driven Processes**: Leveraging analytics and feedback to refine and optimize software development processes.

Where to use software as process?

- ✓ To supports or directly provides system functions.
- ✓ Controls other programs
- ✓ Effective communication
- ✓ Helps in building other software.

Changing Nature of Software:

- The **changing nature of software** reflects its evolution from basic computational tools to highly complex, interconnected systems driving innovation and shaping every aspect of modern life.
- This transformation has been driven by advancements in technology, growing user expectations, and new application areas.

1. System Software:

- System software is a collection of programs which are written to service other programs. Some system software processes complex but determinate, information structures. Other system application process largely indeterminate data. Sometimes when, the system software area is characterized by the heavy interaction with computer hardware that requires scheduling, resource sharing, and sophisticated process management.

2. Application Software:

- Application software is defined as programs that solve a specific business need. Application in this area process business or technical data in a way that facilitates business operation or management technical decision making. In addition to convention data processing application, application software is used to control business function in real time.

3. Engineering and Scientific Software:

- This software is used to facilitate the engineering function and task. however modern application within the engineering and scientific area are moving away from the conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take a real-time and even system software characteristic.

4. Embedded Software:

- Embedded software resides within the system or product and is used to implement and control feature and function for the end-user and for the system itself. Embedded software can perform the limited and esoteric(hard to understand) function or provided significant function and control capability.

Example: Anti-lock brakes, Motion Detector etc.

5. Web Application:

- It is a client-server computer program which the client runs on the web browser. In their simplest form, Web apps can be little more than a set of linked hypertext files that present information using text and limited graphics. However, as e-commerce and B2B application grow in importance. Web apps are evolving into a sophisticate computing environment that not only provides a standalone feature, computing function, and content to the end user.

6. Artificial Intelligence Software:

- Artificial intelligence software makes use of a non numerical algorithm to solve a complex problem that is not amenable to computation or straightforward analysis. Application within this area includes robotics, expert system, pattern recognition, artificial neural network, theorem proving and game playing.

Characteristics of Software:

1. Functionality

- The functionality of software refers to its ability to perform and function according to design specifications.
- In simple terms, software systems should function correctly, i.e. perform all the functions for which they are designed.

2. Usability (User-friendly)

- The user-friendliness of the software is characterized by its ease of use.
- In other words, learning how to use the software should require less effort or time.
- Navigating the software is extremely important since it helps determine the journey the user takes within the software.

3. Complexity

- Software often handles complex tasks and systems with numerous interdependent modules.
- As systems grow in size, managing and maintaining software complexity becomes a significant challenge.

4. Scalability

- Software can scale to accommodate increasing loads or new functionality without significant changes to the core architecture.
- Example: Cloud-based applications can handle thousands to millions of users with proper design.

5. Reusability

- Software components or code can be reused across multiple applications or projects, reducing development time and costs.
- Example: Libraries and frameworks like React or TensorFlow.

6. Reliability

- Software must perform consistently under specified conditions to be reliable.
- This includes handling errors gracefully, maintaining data integrity, and providing accurate results.

7. Maintainability

- Good software design enables easy updates, debugging, and enhancements over its lifecycle.
- Proper documentation and modular design contribute to high maintainability.

8. Portability

- Software can be designed to operate on multiple platforms with minimal changes.
- Example: Cross-platform applications using frameworks like Flutter or Electron.

9. Efficiency

- Software should use resources like memory, processing power, and storage optimally.

Generic view of software Engineering:

- A **generic view of software engineering** provides a high-level understanding of the principles, processes, and goals involved in developing and maintaining software systems.
- It encompasses systematic practices aimed at producing high-quality software efficiently and effectively while addressing the dynamic needs of users and organizations.

There are three phases of any software process:

1. Definition phase

→It includes activities such as

-- System / Information Engineering

-- Software project planning

-- Requirement analysis

2. Development phase

→ It include activities such as:

-- Design

-- Coding

-- Testing

3. Maintenance

→ It include activities such as:

-- Fixing the bugs

-- Adaptation

-- Enhancement

Generic Principles of Software Engineering

- **Understand the Problem**: Thoroughly analyze user needs and system constraints.
- **Plan Before Designing**: Develop a roadmap for efficient and organized execution.
- **Reuse and Modularization**: Leverage reusable components and divide the system into manageable modules.
- **Quality Assurance**: Incorporate testing and validation throughout the development process.
- **Continuous Improvement**: Adapt to feedback and evolving requirements.

Layered Technology in Software Engineering

- Layered technology is an architectural pattern that separates a software system into separate logical layers.
- It is sometimes referred to as layered architecture or layered design.
- Every layer is in charge of a certain component of the functionality of the program and it mostly communicates with the layers that are just above and below it.
- This division of responsibilities encourages modularity which improves extensibility and maintainability.

The Layers

- ❑ **Presentation Layer:**
  The highest layer directly interacting with the user interface is this one. Its duties include presenting information to the user handling user input and rendering the user interface elements. This layer comprises client side JavaScript, HTML and CSS in web applications.
- ❑ **Application Layer:**
  The application's main functionality is contained in this layer which is sometimes referred to as the business logic layer. It carries out calculations and enforces business rules and also processes and manages data and coordinates the interactions of many parts.
- ❑ **Domain Layer:**
  This layer encapsulates the business logic and rules specific to the domain of the application. It defines the objects, entities and their relationships often represented using models or classes. The domain layer is independent of any specific implementation or technology.
- ❑ **Infrastructure Layer:**
  Low-level issues including database access, external service interaction and system-to-system communication are handled by the infrastructure layer. It offers the support required for the higher levels to operate efficiently.

Benefits of Layered Technology

- **Modularity and Separation of Concerns**
  A system can be divided into layers with each layer concentrating on a certain functional area. Because of this division developers are able to work on different aspects of the programme individually which facilitates understanding, maintenance and extension.
- **Scalability**
  Layered architectures facilitate scalability. Individual layers can be scaled independently based on the application needs. For example in a web application if there is a sudden surge in user interactions the presentation layer can be scaled independently of the backend logic.
- **Reusability**
  It is common practise to reuse layers between projects or between modules within a project. For example several client apps or user interfaces can use the same business logic in a well-defined application layer.
- **Interoperability**
  The clear separation of layers enables easier integration with external systems or services. For instance the infrastructure layer can be designed to interact with various types of databases or APIs.
- **Testability**
  Layered architectures promote effective testing. Each layer can be tested independently allowed for unit testing and also for integration testing and system testing to be performed with precision.
- **Maintainability**
  Impacts requiring updates or modifications are frequently restricted to a single layer. In addition to lowering the possibility of unforeseen side effects in other system components this simplifies maintenance.

Real-World Examples

- **Model-View-Controller (MVC):**

  MVC is among the most popular applications of layered architecture. The domain layer is represented by the model the presentation layer is represented by the view and the application layer is represented by the controller.

- **Microservices:**

  In a microservices architecture each microservice can be seen as a self contained layer responsible for a specific aspect of the applications functionality. These services interact through well defined APIs.

Implementing Layered Technology in Software Development

- **Choosing the Right Layers**
  It is critical to choose the right layers for a given project. Although the presentation, application, domain and infrastructure are the fundamental layers the precise arrangement may differ based on the type of application. A user interface-centric application might prioritise the presentation layer whereas a data-intensive application might need a more complex data access layer.
- **Communication between Layers**
  A coherent application requires effective interlayer communication. Usually the well defined interfaces or APIs are used to do this. Every layer makes available a collection of services or features that the levels above it can utilise. Because of this the layers are no longer connected and enabling them to change or evolve separately from the system as a whole.
- **Managing Dependencies**
  It is essential to give careful thought to interlayer dependencies in order to prevent tight coupling. Only the layers directly underneath a layer should constitute its sources of dependence. This lowers the possibility of unexpected consequences while applying updates or modifications. Effective dependency management often involves the use of Dependency Injection (DI) frameworks and Inversion of Control (IoC) containers.

❖ Handling Exceptions and Errors

Layered architectures need a clear plan for managing mistakes and exceptions. Every layer should be in charge of managing exceptions related to its particular domain. This can stop a minor problem in one layer from affecting the programme as a whole.

❖ Security Considerations

Every layer should address security concerns. Two essential components of layered technology are access control and data validation. For instance only authorised users should be able to carry out specific actions and business rules should be enforced by the application layer.

❖ Scaling Layers

In some cases certain layers may need to be scaled more than others. For example in a heavily trafficked web application the presentation layer may require multiple servers, while the other layers can remain relatively unchanged. Load balancers and distributed computing technologies can be used to distribute the load effectively.

- **Monitoring and Debugging**
  Robust monitoring and debugging techniques are advantageous for layered architectures. To monitor its performance and behaviour and every layer should have its own set of metrics and logs. In addition to enabling focused enhancements or optimisations and this makes troubleshooting easier.
- **Evolution and Future-Proofing**
  Triggered architectures offer a strong basis for adjusting as needs and technology develop. Not every layer of the system needs to be completely redesigned in order to incorporate new technologies or components. Remaining competitive and adaptable to user needs depends on this flexibility.

Software Engineering/ Challenges :

- Unclear software requirement
- Lack of communication and Collaboration
- Poor code quality and bugs
- Unrealistic timelines
- Unrealistic budget estimate
- Slow product launch
- Hiring and retaining top tech talent
- Security
- Difficulty with cross-platform functionality
- Inadequate scalability planning

Software Engineering costs

- Cost estimation simply means a technique that is used to find out the cost estimates.
- The cost estimate is the financial spend that is done on the efforts to develop and test software in Software Engineering.
- Roughly 60% of costs are development costs, 40% are testing costs. Costs vary depending on the type of system being developed and the requirements of system attributes such as performance and system reliability.
- Distribution of costs depends on the development model that is used A software cost estimating methodology is an indirect metric used by software professionals to estimate project costs.