**MODULE 2**

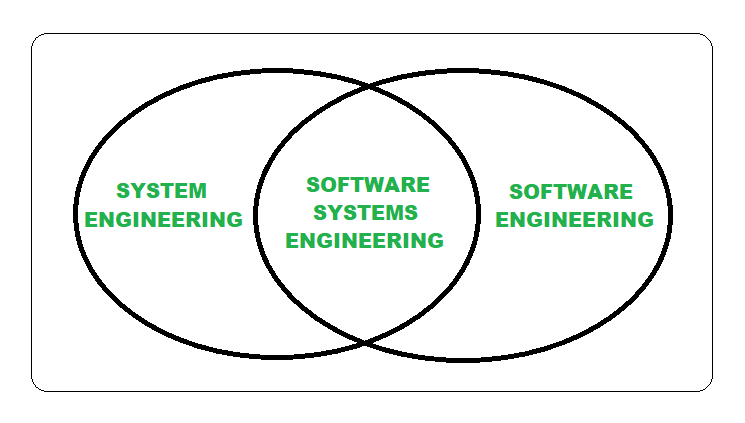
**System Engineering**

Contents:

System Requirement Definition, System Architectural design, System configuration management, System Requirements .

**1)SYSTEM ENGINEERING**

* Systems Engineering is an engineering field that takes an interdisciplinary approach to product development. Systems engineers analyze the collection of pieces to make sure when working together, they achieve the intended objectives or purpose of the product.



## **What is system engineering about?**

Systems Engineering focuses on:

* Starting early in the development cycle, developing a concept of operations and defining required functionality, establishing stakeholder goals and objectives, and balancing stakeholders’ success criteria
* Consider the following questions while developing a lifecycle model, process approach, and governance structures that match the degree of complexity, uncertainty, change, and variety
* Alternative solution concepts and architecture evaluations
* For each stage of the project, we developed a bespoke risk assessment framework and solution architecture to meet the base lining and modeling needs
* performing design synthesis and system verification and validation
* When evaluating both the problem and solution domains, taking into account necessary enabling systems and services, identifying the role that parts and relationships among components play in the overall system behavior and performance, and determining how to balance all of these factors to achieve an acceptable result.

## **Techniques for System Engineering:**

Some of the most common tools and techniques include:

* **System Modeling:** This is a way of representing a system using diagrams or equations. System models can be used to understand how a system works, identify potential problems, and test possible solutions.
* **System Analysis:** This is the process of examining a system to identify its strengths and weaknesses. System analysts use a variety of methods to analyze systems, including simulations, mathematical models, and data analysis.
* **System Design:** This is the process of creating a new system or modifying an existing one. System designers must have a deep understanding of how systems work in order to create effective and efficient designs.
* **System Testing:** This is the process of verifying that a system meets its requirements. System tests can be conducted using simulations, prototypes, or actual systems.

**Fundamentals of system engineering**

* Systems engineering is based around systems thinking principles, and the goal of a systems engineer is to help a product team produce an engineered system that performs a useful function as defined by the requirements written at the beginning of the project.
* The final product should be one where the individual systems work together in a cohesive whole that meet the requirements of the product.

**2)System Requirements**

System requirements typically refer to the minimum hardware and software specifications needed to run particular application or software program. These requirements can vary depending on the complexity and resource demands of the software.

General breakdown of what system requirements usually include:

* Operating System: Specifies which operating systems are supported, such as Windows, macOS, Linux, etc., and sometimes specific versions within those operating systems.
* Processor (CPU): Specifies the minimum required CPU type and speed. This could include information about the number of cores or whether a specific instruction set (like SSE, AVX) is required.
* Memory (RAM): Specifies the minimum amount of RAM required for the software to run smoothly.
* Graphics Card (GPU): Specifies whether a dedicated graphics card is required and, if so, the minimum specifications for it. This might include details like GPU model, VRAM (video memory) size, and support for certain graphics APIs like DirectX or OpenGL.
* Storage: Specifies the minimum amount of available disk space needed for installation, as well as any additional space required for temporary files or user data.
* Other Hardware: Sometimes, specific peripherals or hardware components might be required or recommended, such as a DVD drive, a certain type of monitor, or input devices like a keyboard, mouse, or graphics tablet.
* Software Dependencies: Lists any additional software packages or libraries that must be installed for the program to function correctly. This might include runtime environments like Java or .NET Framework, or specific versions of system libraries.
* Internet Connection: Specifies whether an internet connection is required for certain features or functionalities, such as online multiplayer gaming or cloud-based services.

**Difference between System requirements and technical requirements**

**System Requirements**:

* System requirements describe the external behavior and features expected from the software system as perceived by its users or stakeholders.
* These requirements are typically expressed in terms of functionalities, constraints, and quality attributes that the software should exhibit.
* System requirements address questions such as "What does the software need to accomplish?" and "What features should it provide to users?"
* Examples of system requirements include functional requirements (what the system should do), non-functional requirements (constraints or quality attributes like perform

**Technical Requirements**:

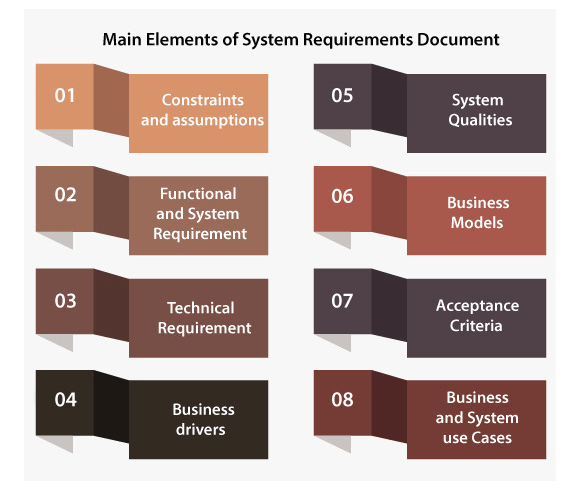
* Technical requirements, on the other hand, describe the internal architecture, design, and implementation considerations of the software system.
* These requirements are more concerned with the technology stack, frameworks, platforms, and development methodologies that will be used to build the system.
* Technical requirements address questions such as "How will the software be implemented?" and "What technologies and tools will be used?"
* Examples of technical requirements include the choice of programming languages, databases, operating systems, development frameworks, coding standards, and testing methodologies.

**System Requirement**

* System requirements document is a set of documentation that describes the behavior and features of a software or system.
* The system requirements document (SRD) describes the system-level performance and functional requirements for a system.
* It characterizes the functional and non-functional requirements of the system.
* The document doesn't layout design or technology solutions.

According to the IEEE standards, SRDs must cover the following important topics.

* + Quality
  + Security/Privacy
  + Functional capabilities
  + Safety
  + Performance levels
  + Constraints and Limitations
  + Data Structure/Elements
  + Reliability
  + Interface*s*



**1)Constraints and Assumption**

* Underline the lack of any design imposed by the client on the system design.
* This section also comprises assumptions which are made by the team of requirement engineering at the time of requirements gathering and analysis.
* If there is an incorrect assumption, it is needed to re-evaluate the system requirements specification in order to ensure that the documented requirements are still valid.

**2)Functional and System Requirements**

* This normally comprises a hierarchical arrangement of requirements, with the functional/business requirements at the uppermost level and the detailed system requirements are listed as their child items.
* The requirements are written like "System requires the capability to do x," with supporting information comprised as required.

**3)Technical Requirements**

* This part is utilized to list any of the non-functional requirements, which basically personifies the technical environment that the product requires to work in and incorporate the technical limitations that it requires to work under.
* These technical requirements are important in deciding how high-level functional requirements can decompose into more definite system requirements.

**4)Business Drivers**

* This part depicts the reasons why the client is hoping to build the system.
* The drivers may comprise issues as well as opportunities. Generally, a mix of issues and opportunities are required to give inspiration to a new system.

**5)System Qualities**

* The system qualities are the non-functional requirements which are used to define the system's quality. They are comprised of items like availability, maintainability, security, reliability, and scalability.
* Unlike functional requirements, the quality of the system generally contains tables of particular metrics that the system should meet to be acknowledged.

**6)Business Models**

This part portrays the fundamental business model of the client that the system will require to support. This contains current-state, future-state and organizational context state diagrams, key business functions, business context and process flow diagrams. This part is generally made during the functional analysis phase.

**7)Acceptance Criteria**

* This part will portray the measures by which the client will "sign-off" the final system. Based on the procedure, this may occur toward the end of the testing and quality assurance stage, or in agile methodology, toward the finish of every iteration.
* This measure refers to the requirement in order to complete all user acceptance tests and fix all bugs/defects which meet a pre-defined priority or severity limits.

**8)Business and System Use Cases**

* This segment ordinarily comprises a UML use case diagram, which shows the main external entities that will collaborate with the system along with diverse use cases that should be met.
* There will be a formal definition of the steps for every use case which is required to fulfill the purpose of the business, along with the essential pre-conditions and post-conditions.
* The system use cases are derived from the system requirements and the business use case is derived from the functional requirements.

**3)System Architectural design**

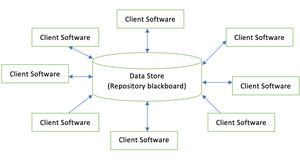
* IEEE defines architectural design as “the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.”

We explore the following architectural design:

* Data centered architecture
* Monolith architecture
* Micro service architecture
* Service-Oriented Architecture
* Event-Driven Architecture

**1)Data centered architecture**

* A data store will reside at the center of this architecture and is accessed frequently by the other components that update, add, delete, or modify the data present within the store.
* The figure illustrates a typical data-centered style. The client software accesses a central repository. Variations of this approach are used to transform the repository into a blackboard when data related to the client or data of interest for the client change the notifications to client software.
* This data-centered architecture will promote integrability. This means that the existing components can be changed and new client components can be added to the architecture without the permission or concern of other clients.
* Data can be passed among clients using the blackboard mechanism.



2) **Monolithic architecture**



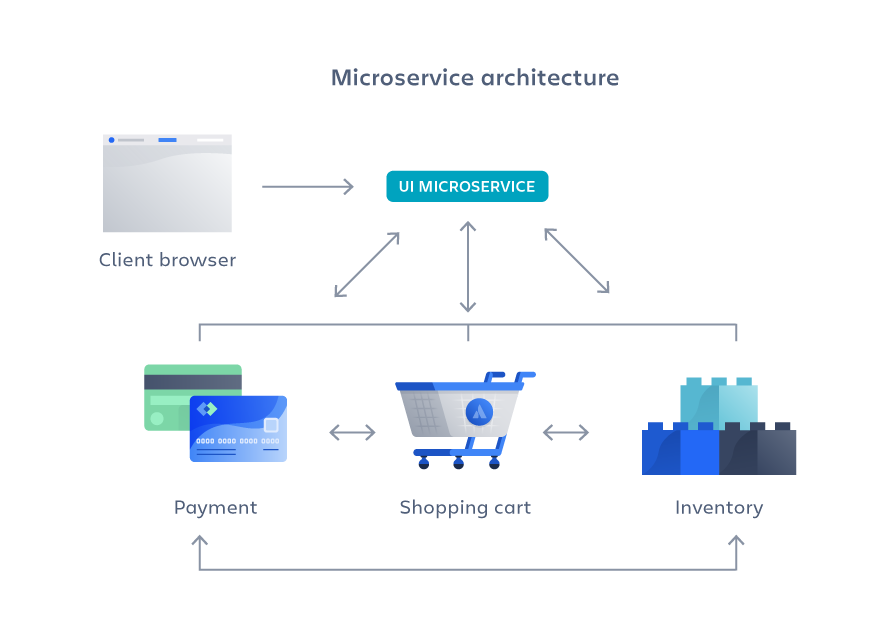
* A monolithic architecture is a traditional model of a software program, which is built as a unified unit that is self-contained and independent from other applications.
* A monolithic architecture is a singular, large computing network with one code base that couples all of the business concerns together.
* To make a change to this sort of application requires updating the entire stack by accessing the code base and building and deploying an updated version of the service-side interface. This makes updates restrictive and time-consuming

The advantages of a monolithic architecture include:

* **Easy deployment** – One executable file or directory makes deployment easier.
* **Development** – When an application is built with one code base, it is easier to develop.
* **Performance** – In a centralized code base and repository, one API can often perform the same function that numerous APIs perform with microservices.
* **Simplified testing** – Since a monolithic application is a single, centralized unit, end-to-end testing can be performed faster than with a distributed application.   
    
  **Easy debugging** – With all code located in one place, it’s easier to follow a request and find an issue.

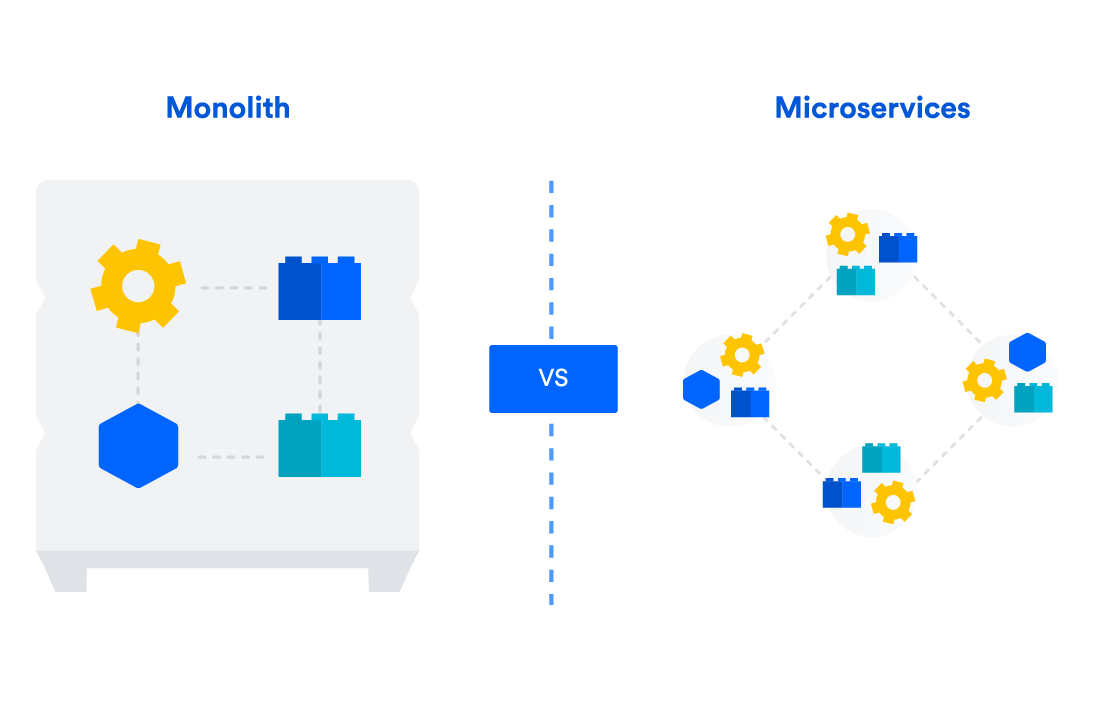
The disadvantages of a monolith include:

* **Slower development speed** – A large, monolithic application makes development more complex and slower.
* **Scalability** – You can’t scale individual components.
* **Reliability** – If there’s an error in any module, it could affect the entire application’s availability.
* **Barrier to technology adoption** – Any changes in the framework or language affects the entire application, making changes often expensive and time-consuming.
* **Lack of flexibility** – A monolith is constrained by the technologies already used in the monolith.
* **Deployment** – A small change to a monolithic application requires the redeployment of the entire monolith.

3) **Micro service architecture**

* A micro services architecture, also simply known as micro services, is an architectural method that relies on a series of independently deployable services.
* These services have their own business logic and database with a specific goal. Updating, testing, deployment, and scaling occur within each service.
* Micro services decouple major business, domain-specific concerns into separate, independent code bases.

Microservices don’t reduce complexity, but they make any complexity visible and more manageable by separating tasks into smaller processes that function independently of each other and contribute to the overall whole.



The advantages of microservices are:

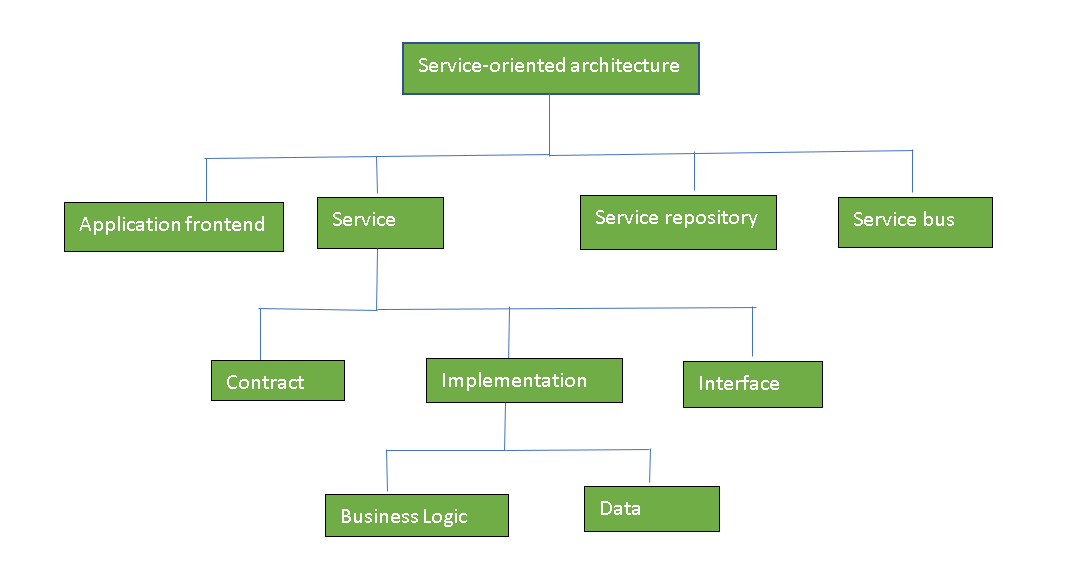
* **Agility** – Promote agile ways of working with small teams that deploy frequently.
* **Flexible scaling** – If a microservice reaches its load capacity, new instances of that service can rapidly be deployed to the accompanying cluster to help relieve pressure. We are now multi-tenant and stateless with customers spread across multiple instances. Now we can support much larger instance sizes.
* **Continuous deployment** – We now have frequent and faster release cycles. Before we would push out updates once a week and now we can do so about two to three times a day.
* **Highly maintainable and testable** – Teams can experiment with new features and roll back if something doesn’t work. This makes it easier to update code and accelerates time-to-market for new features. Plus, it is easy to isolate and fix faults and bugs in individual services.
* **Independently deployable** – Since microservices are individual units they allow for fast and easy independent deployment of individual features.
* **Technology flexibility** – Microservice architectures allow teams the freedom to select the tools they desire.
* **High reliability** – You can deploy changes for a specific service, without the threat of bringing down the entire application.

**The disadvantages of microservices can include:**

* **Development sprawl** – Micro services add more complexity compared to a monolith architecture, since there are more services in more places created by multiple teams. If development sprawl isn’t properly managed, it results in slower development speed and poor operational performance.
* **Exponential infrastructure costs** – Each new micro service can have its own cost for test suite, deployment playbooks, hosting infrastructure, monitoring tools, and more.
* **Added organizational overhead** – Teams need to add another level of communication and collaboration to coordinate updates and interfaces.
* **Debugging challenges** – Each microservice has its own set of logs, which makes debugging more complicated. Plus, a single business process can run across multiple machines, further complicating debugging.
* **Lack of standardization** – Without a common platform, there can be a proliferation of languages, logging standards, and monitoring.
* **Lack of clear ownership** – As more services are introduced, so are the number of teams running those services. Over time it becomes difficult to know the available services a team can leverage and who to contact for support.

4) **Service-Oriented Architecture**

* SOA is an architectural approach in which applications make use of services available in the network. In this architecture, services are provided to form applications, through a network call over the internet.
* It uses common communication standards to speed up and streamline the service integrations in applications.
* Each service in SOA is a complete business function in itself. The services are published in such a way that it makes it easy for the developers to assemble their apps using those services. Note that SOA is different from microservice architecture.
* SOA allows users to combine a large number of facilities from existing services to form applications.
* SOA encompasses a set of design principles that structure system development and provide means for integrating components into a coherent and decentralized system.
* SOA-based computing packages functionalities into a set of interoperable services, which can be integrated into different software systems belonging to separate business domains.



**Guiding Principles of SOA:**

1. **Standardized service contract:** Specified through one or more service description documents.
2. **Loose coupling:** Services are designed as self-contained components, maintain relationships that minimize dependencies on other services.
3. **Abstraction:** A service is completely defined by service contracts and description documents. They hide their logic, which is encapsulated within their implementation.
4. **Reusability:** Designed as components, services can be reused more effectively, thus reducing development time and the associated costs.
5. **Autonomy:** Services have control over the logic they encapsulate and, from a service consumer point of view, there is no need to know about their implementation.
6. **Discoverability:** Services are defined by description documents that constitute supplemental metadata through which they can be effectively discovered. Service discovery provides an effective means for utilizing third-party resources.
7. **Composability:** Using services as building blocks, sophisticated and complex operations can be implemented. Service orchestration and choreography provide a solid support for composing services and achieving business goals.

The defining concepts of SOA are the following:

* Business value is more important than technical strategy.
* Strategic goals are more important than benefits related to specific projects.
* Basic interoperability is more important than custom integration.
* Shared services are more important than implementations with a specific purpose.
* Continuous improvement is more important than immediate perfection.

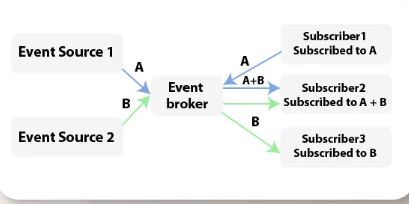
Components of SOA:

1. Service – Service Contract
2. Service Provider
3. Service Consumer

Service Registry – Service Repository

5)Event driven Architecture

Event-driven architecture (EDA) is a design pattern where system components communicate by generating, detecting, and responding to External events. Events represent significant occurrences, such as user actions or changes in the system state. In EDA, components are decoupled, allowing them to operate independently.



**Components of Event-Driven Architecture(EDA)**

1. **Event Source:** An event source is any component or system that generates events. This can include user interfaces, sensors, databases, or other external systems.
2. **Event:** The fundamental unit of communication in EDA. Events represent significant occurrences or state changes and are emitted by event sources.
3. **Event Broker/Event Bus:** The event broker or event bus acts as an intermediary that facilitates the communication of events between different components. It can handle the distribution, filtering, and routing of events.
4. **Publisher:** A component that generates and sends events to the event bus. It’s responsible for publishing events when certain conditions or actions occur.
5. **Subscriber:** A component that expresses interest in specific types of events and subscribes to them. Subscribers listen for events on the event bus and respond accordingly.
6. **Event Handler:** A piece of code or logic associated with a subscriber that specifies how to respond when a particular type of event is received. Event handlers are responsible for processing events.
7. **Dispatcher:** In some systems, a dispatcher may be used to route events to the appropriate event handlers. It helps manage the flow of events within the system.
8. **Aggregator:** An aggregator may be used to combine or aggregate multiple related events into a single, more meaningful event. This can help reduce the complexity of handling numerous individual events.

**Events in Event-Driven Architecture(EDA)**

* **Representation:** They are represented as messages or signals that convey information about a particular occurrence.
* **Triggering:** Events can be triggered by various sources, such as user actions, changes in data, external stimuli, or system processes.
* **Asynchronicity:** EDA often involves asynchronous communication, where components operate independently and asynchronously in response to events, allowing for parallel processing.
* **Publish-Subscribe Model:** Events are typically handled using a publish-subscribe model. Components interested in certain types of events subscribe to them, while components that generate events publish them.
* **Event Types:** Events are categorized into different types based on their nature and purpose. Examples include “UserLoggedIn,” “OrderPlaced,” or “TemperatureChanged.”
* **Payload:** Events often carry additional information known as the payload. This payload provides context and details about the event. For example, a “PaymentReceived” event may include information about the payment amount and the payer.
* **Event Handling:** Components have event handlers that specify how they respond to specific types of events. When an event occurs, the associated event handler is invoked.
* **Real-Time Processing:** Events enable real-time processing by allowing components to react immediately to changes, making EDA suitable for scenarios where responsiveness and agility are crucial.

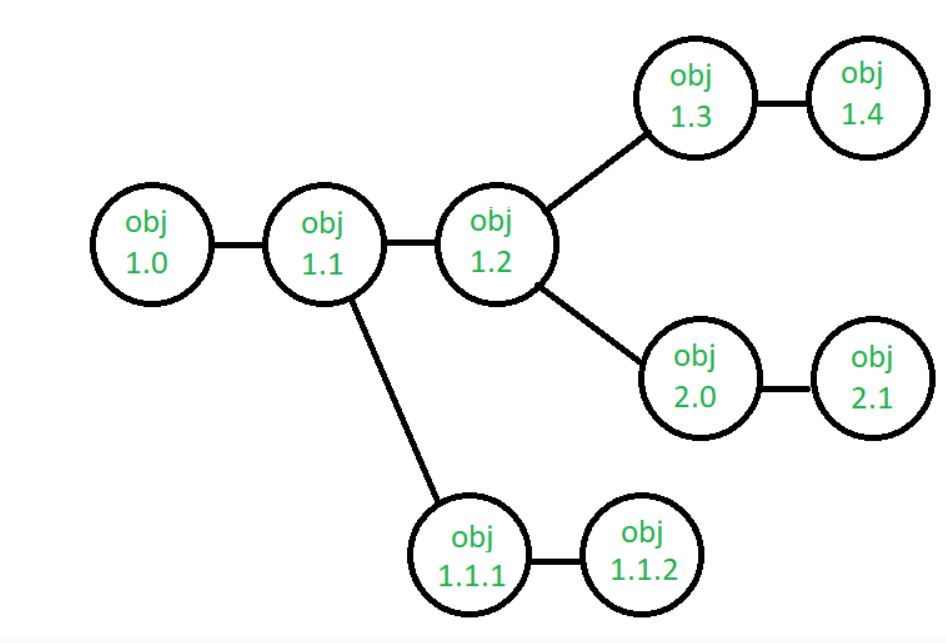
**4)System configuration management**

* **System Configuration Management (SCM)** is an arrangement of exercises that controls change by recognizing the items for change, setting up connections between those things, making/characterizing instruments for overseeing diverse variants, controlling the changes being executed in the current framework, inspecting and revealing/reporting on the changes made.

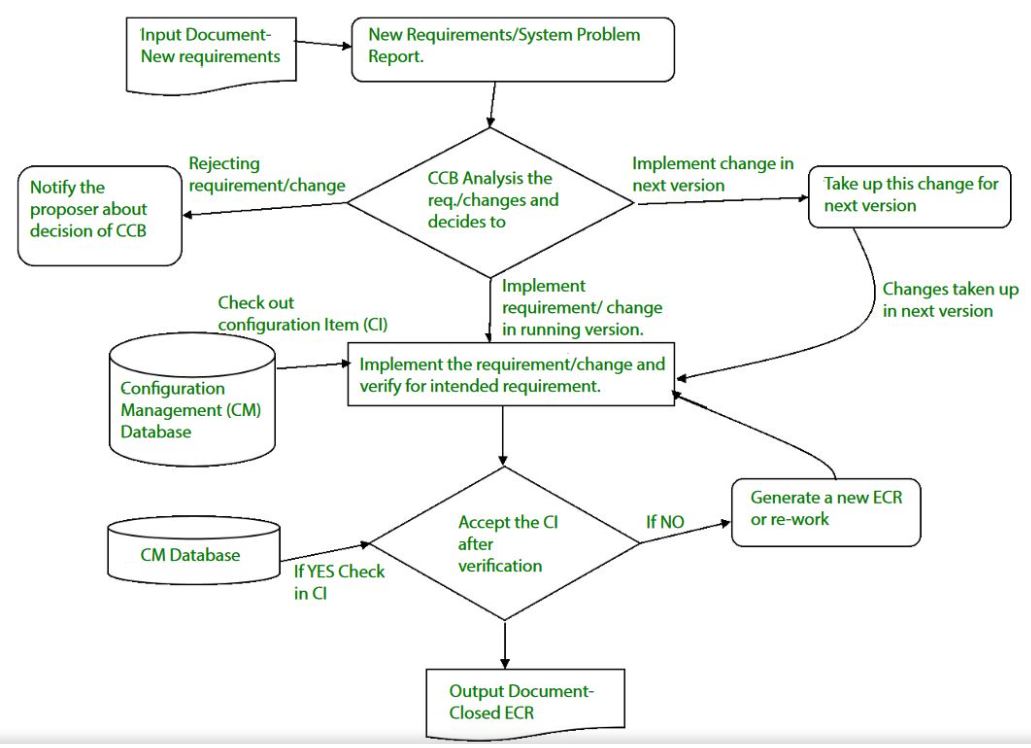
**Processes involved in SCM –** Configuration management provides a disciplined environment for smooth control of work products. It involves the following activities:

**1)Identification and Establishment –** Identifying the configuration items from products that compose baselines at given points in time (a baseline is a set of mutually consistent Configuration Items, which has been formally reviewed and agreed upon, and serves as the basis of further development

**2)Version control –** Creating versions/specifications of the existing product to build new products with the help of the SCM system. A description of the version is given below:



**3)Change control –** Controlling changes to Configuration items (CI). The change control process is explained in Figure below



**4)Configuration auditing –** A software configuration audit complements the formal technical review of the process and product. It focuses on the technical correctness of the configuration object that has been modified. The audit confirms the completeness, correctness, and consistency of items in the SCM system and tracks action items from the audit to closure.

**5)Reporting –** Providing accurate status and current configuration data to developers, testers, end users, customers, and stakeholders through admin guides, user guides, FAQs, Release notes, Memos, Installation Guide, Configuration guides, etc.

**Key objectives of SCM**

1)Control the evolution of software systems: SCM helps to ensure that changes to a software system are properly planned, tested, and integrated into the final product.

2)Enable collaboration and coordination: SCM helps teams to collaborate and coordinate their work, ensuring that changes are properly integrated and that everyone is working from the same version of the software system.

3)Provide version control: SCM provides version control for software systems, enabling teams to manage and track different versions of the system and to revert to earlier versions if necessary.

4)Facilitate replication and distribution: SCM helps to ensure that software systems can be easily replicated and distributed to other environments, such as test, production, and customer sites.

5)SCM is a critical component of [software development](https://www.geeksforgeeks.org/what-is-software-development/?ref=lbp), and effective SCM practices can help to improve the quality and reliability of software systems, as well as increase efficiency and reduce the risk of errors.

**The main advantages of SCM**

* Improved productivity and efficiency by reducing the time and effort required to manage software changes.
* Reduced risk of errors and defects by ensuring that all changes were properly tested and validated.
* Increased collaboration and communication among team members by providing a central repository for software artifacts.
* Improved quality and stability of software systems by ensuring that all changes are properly controlled and managed.

**Components of Software Requirement Analysis**

1. **Stakeholder Identification and Analysis**: Identifying all stakeholders involved in the project and understanding their needs, expectations, and constraints.
2. **Requirements Elicitation**: Gathering requirements from stakeholders through techniques like interviews, surveys, and workshops.
3. **Requirement Documentation**: Documenting requirements in a clear and structured manner using user stories, use cases, and formal specifications.
4. **Requirement Analysis and Prioritization**: Analyzing and prioritizing requirements to ensure they are complete, consistent, and feasible.
5. **Requirement Validation**: Validating requirements with stakeholders to ensure they accurately represent their needs and expectations.
6. **Requirement Management**: Managing changes to requirements throughout the project lifecycle and ensuring traceability between requirements and other artifacts.
7. **Requirement Communication**: Communicating requirements effectively to all stakeholders involved in the project to prevent misunderstandings.
8. **Requirement Review and Sign-off**: Reviewing documented requirements with stakeholders to obtain their approval or sign-off.

**5)Software Requirement Analysis**

Software Requirement Analysis is a critical phase in software development where requirements for a software system are gathered, documented, analyzed, and managed. Here are the key aspects:

**Needs**

* **Understanding Stakeholder Requirements**: Identify and understand the needs and expectations of all stakeholders involved, including end-users, clients, managers, and developers.
* **Problem Definition**: Clearly define the problem that the software system is supposed to solve or the opportunity it aims to exploit.
* **Scope Identification**: Determine the boundaries and extent of the software system's functionality, including both functional and non-functional requirements.

**Characteristics:**

* **Comprehensive**: The analysis should cover all aspects of the software system, including functionality, performance, usability, reliability, and security.
* **Clear and Unambiguous**: Requirements should be expressed in a way that leaves no room for interpretation or ambiguity, ensuring a common understanding among all stakeholders.
* **Verifiable**: Requirements should be verifiable to ensure that they can be tested to determine whether they have been met.
* **Prioritized**: Requirements should be prioritized based on their importance and impact on the software system's success.
* **Feasible**: Requirements should be technically and economically feasible within the constraints of the project.

**Components needed dratf a SRS Document**

**Table of Content**

* [**Introduction**](https://www.geeksforgeeks.org/software-requirement-specification-srs-format/)
* [**General description**](https://www.geeksforgeeks.org/software-requirement-specification-srs-format/)
* [**Functional Requirements**](https://www.geeksforgeeks.org/software-requirement-specification-srs-format/)
* [**Interface Requirements**](https://www.geeksforgeeks.org/software-requirement-specification-srs-format/)
* [**Performance Requirements**](https://www.geeksforgeeks.org/software-requirement-specification-srs-format/)
* [**Design Constraints**](https://www.geeksforgeeks.org/software-requirement-specification-srs-format/)
* [**Non-Functional Attributes**](https://www.geeksforgeeks.org/software-requirement-specification-srs-format/)
* [**Preliminary Schedule and Budget**](https://www.geeksforgeeks.org/software-requirement-specification-srs-format/)
* [**Appendices**](https://www.geeksforgeeks.org/software-requirement-specification-srs-format/)

1)Introduction

* Purpose of this Document – At first, main aim of why this document is necessary and what’s purpose of document is explained and described.
* Scope of this document – In this, overall working and main objective of document and what value it will provide to customer is described and explained. It also includes a description of development cost and time required.
* Overview – In this, description of product is explained. It’s simply summary or overall review of product.

2)General description

* In this, general functions of product which includes objective of user, a user characteristic, features, benefits, about why its importance is mentioned. It also describes features of user community.

3)Functional Requirements

* In this, possible outcome of software system which includes effects due to operation of program is fully explained. All functional requirements which may include calculations, data processing, etc. are placed in a ranked order. Functional requirements specify the expected behavior of the system-which outputs should be produced from the given inputs. They describe the relationship between the input and output of the system. For each functional requirement, detailed description all the data inputs and their source, the units of measure, and the range of valid inputs must be specified.

**4)Interface Requirements**

* In this, software interfaces which mean how software program communicates with each other or users either in form of any language, code, or message are fully described and explained. Examples can be shared memory, data streams, etc.

**5)Performance Requirements**

* In this, how a software system performs desired functions under specific condition is explained. It also explains required time, required memory, maximum error rate, etc. The performance requirements part of an SRS specifies the performance constraints on the software system. All the requirements relating to the performance characteristics of the system must be clearly specified. There are two types of performance requirements: static and dynamic. Static requirements are those that do not impose constraint on the execution characteristics of the system. Dynamic requirements specify constraints on the execution behaviour of the system.

6) **Design Constraints**

* In this, constraints which simply means limitation or restriction are specified and explained for design team. Examples may include use of a particular algorithm, hardware and software limitations, etc. There are a number of factors in the client’s environment that may restrict the choices of a designer leading to design constraints such factors include standards that must be followed resource limits, operating environment, reliability and security requirements and policies that may have an impact on the design of the system. An SRS should identify and specify all such constraints.

**7)Non-Functional Attributes**

* In this, non-functional attributes are explained that are required by software system for better performance. An example may include Security, Portability, Reliability, Reusability, Application compatibility, Data integrity, Scalability capacity, etc.

**8)Preliminary Schedule and Budget**

* In this, initial version and budget of project plan are explained which include overall time duration required and overall cost required for development of project.

**9)Appendices**

* In this, additional information like references from where information is gathered, definitions of some specific terms, acronyms, abbreviations, etc. are given and explained.

**Feasibility Study**

****

**1)Technical feasibility**

* Technical feasibility includes checking for accessibility to technical resources and applications within the organization. If the resources already exist, you must then determine if the technical team can customize the technology into new working systems for the project. Not only do you need the correct technical resources, but the equipment also needs to be evaluated to ensure it has the proper hardware and software for the proposed plan

**2.)Economic feasibility**

* Economic feasibility allows the company to determine the cost and benefits analyses, which helps provide decision-makers with a list of potential economic benefits to the organization. They need to know the total cost, including accidental expenses, so that during the project, they may be able to anticipate any potential unforeseen monetary challenges.

**3.)Operational feasibility**

* Operational feasibility assesses how well a proposed plan fits within the existing business environment, and if developed, whether current purchasers will use it. Some variables that affect the outcome of this analysis are whether management support, how buyers feel about the current system in place and if the proposed system will benefit the organization.

**4.)Legal feasibility**

* Before beginning a project, legal feasibility checks to ensure that all parts of the proposed project adhere to legal rules and requirements in that specific geographic area. Zoning laws, social media laws and many others need to be examined because the law must permit all aspects of the project for an organization to get consent to begin.

**5.)Schedule feasibility**

* The final, but very important feasibility study is that of the schedule check. It estimates how much time a team needs to complete the project. All invested groups should recognize and agree that the project is to be finished within an agreed-upon timeframe for the proposed plan to be successful.

**6)Components of specification languages for SRS**

1)Functional Requirements:

* + Use Cases: Descriptions of interactions between the system and external actors to achieve specific goals.
  + Functional Decomposition: Breaking down system functions into smaller, manageable components.
  + State Diagrams: Representations of how the system transitions between different states in response to events.

2)Non-functional Requirements:

* + Performance Requirements: Specifications regarding response times, throughput, and resource utilization.
  + Reliability Requirements: Criteria for the system's reliability, availability, and fault tolerance.
  + Security Requirements: Measures to ensure data confidentiality, integrity, and availability.
  + Usability Requirements: Criteria related to the user interface, accessibility, and user experience.
  + Scalability Requirements: Specifications for how the system should handle increasing loads or growing user bases.
  + Maintainability Requirements: Guidelines for how the system should be designed to facilitate maintenance and updates.

3)Data Requirements:

* + Data Dictionary: Definitions of data elements used in the system and their characteristics.
  + Data Flow Diagrams: Representations of how data flows through the system, including inputs, outputs, and data stores.
  + Entity-Relationship Diagrams (ERD): Models showing the relationships between different data entities in the system.

4)Interface Requirements:

* + User Interface (UI) Design: Descriptions or mockups of how the system's user interface should look and behave.
  + Hardware Interfaces: Specifications for interactions between the software system and hardware components.
  + Software Interfaces: Specifications for interactions between the software system and other software components or systems.

5)Contraints

* + Regulatory Constraints: Requirements imposed by laws, regulations, or industry standards.
  + Technical Constraints: Limitations imposed by technology, infrastructure, or compatibility with existing systems.
  + Resource Constraints: Limitations related to budget, time, or available resources.

6)Assumptions and Dependencies:

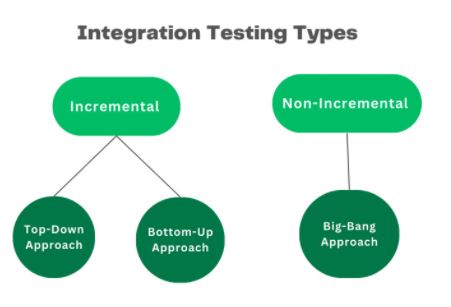
* + Assumptions: Statements about conditions or factors that are assumed to be true for the system to function as intended.
  + Dependencies: External factors or components that the system relies on to fulfill its requirements.

7)Traceability Matrix:

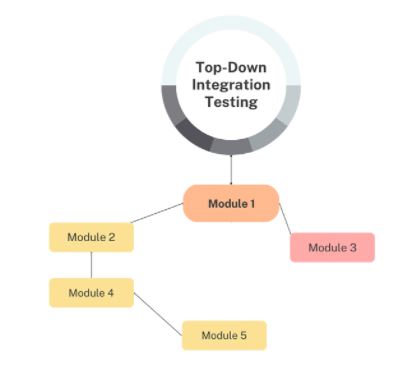
* + Requirements Traceability Matrix (RTM): A matrix linking requirements to their sources, justifications, and verification methods, ensuring each requirement is adequately addressed

**7)Integration Testing**

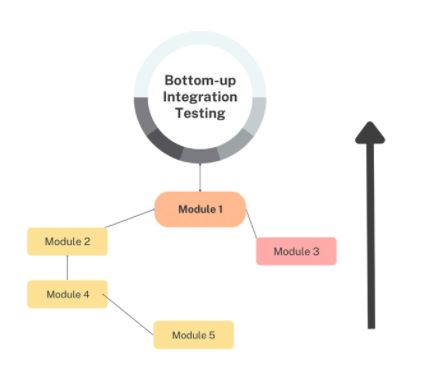
The program is divided into more components, known as modules or units. Each module is responsible for a specific task. The real challenge comes when we combine these components to develop the entire software system.



1)Top-Down Integration

* Top-down testing employs a systematic approach to testing the software modules from the top level down through the system hierarchy. The testing begins with the main module of the software and then proceeds to test the submodules of the application.
* Its primary goal is to ensure functionality between the higher-level modules and their submodules. As the testing procedure progresses through the hierarchy, module relationships are checked to ensure the software components operate as designed.

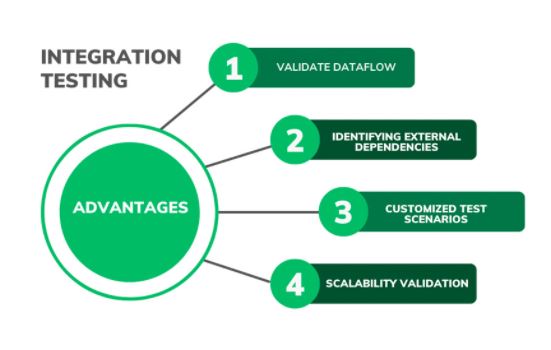
2) Bottom-Up Integration

* When performing Bottom-Up testing, lower-level modules are tested first. It gradually moves to the higher-level modules, and so on, until all facets of the software have been thoroughly tested. This strategy is referred to as inductive reasoning. It is beneficial when incorporating pre-made or already-existing components into the final product.

**3)Big bang approach**

* It entails compiling all software modules into one structure and evaluating it as one unit. Individual modules are not examined separately. Instead, they are combined and tested in a single process.
* It necessitates firm communication between the development and testing teams to appropriately identify and solve any issues detected during the testing procedure. The big-bang strategy can be quicker and less expensive than alternative approaches, as developers don’t need incremental testing.
* This strategy could work for software systems with fewer and less intricate component relationships. However, it might be challenging to identify the precise module when flaws are discovered during testing. Let’s check out the integration testing example to see how it works.

Advantages of Integration Testing



**8)Performance testing**

* Performance testing involves evaluating how a system performs in terms of responsiveness and stability under a particular workload. Different approaches are utilized to cover various aspects of performance. Here are the main approaches involved in performance testing:

**1. Load Testing**

**Objective:** To measure system performance under expected user loads. **Approach:**

* Simulate the expected number of users or transactions to see how the system handles it.
* Identify the system’s behavior under normal and peak load conditions.
* Metrics include response time, throughput, and error rates.

**2. Stress Testing**

* **Objective:** To determine the system’s breaking point and how it behaves under extreme conditions. **Approach:**
* Gradually increase the load beyond the expected peak until the system fails.
* Identify the maximum capacity and point of failure.
* Assess system recovery after failure.

**3. Spike Testing**

* **Objective:** To evaluate the system's response to sudden, significant increases in load. **Approach:**
* Introduce a sudden surge of users or transactions to the system.
* Observe how the system handles the sudden spike and how quickly it recovers.

**4)Endurance (Soak) Testing**

* **Objective:** To check system performance over an extended period. **Approach:**
* Run the system under a significant load for an extended period to identify potential memory leaks or performance degradation.
* Ensure the system can handle sustained use over time without failure.

**5)Volume Testing**

* **Objective:** To test the system's ability to handle large volumes of data. **Approach:**
* Input a large volume of data into the system to test its database and overall performance.

Identify any issues related to data processing, storage, and retrieval.

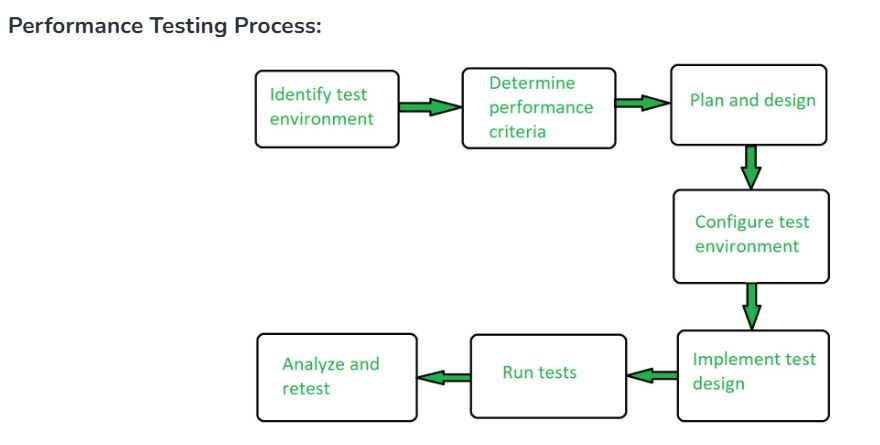
**6)Scalability Testing**

* **Objective:** To determine how well the system scales with increased load. **Approach:**
* Gradually increase the load and measure the system’s ability to scale up or down.
* Assess the system’s capacity to handle growth and its efficiency at different scales.

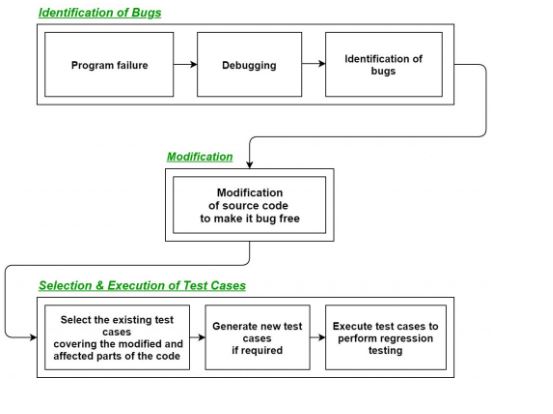
**7) Configuration Testing**

* **Objective:** To evaluate the system’s performance with different configurations. **Approach:**
* Test the system with various hardware and software configurations to find the optimal setup.

Identify how different configurations impact performance.



**9)Regression testing**

* Regression Testing is the process of testing the modified parts of the code and the parts that might get affected due to the modifications to ensure that no new errors have been introduced in the software after the modifications have been made.

**10)Principles of subsystem design**

Subsystem design is a critical aspect of systems engineering, involving the creation and integration of smaller units (subsystems) that together form a larger system. The principles of subsystem design ensure that these smaller units are effective, reliable, and work harmoniously within the larger system. Here are the key principles of subsystem design:

1)Modularity:

Definition: Design subsystems as discrete, interchangeable modules.

Benefits: Enhances maintainability, scalability, and ease of upgrades.

2)Encapsulation:

Definition: Encapsulate the internal workings of subsystems to hide their complexity.

Benefits: Reduces interdependencies, making the system more robust and easier to manage.

3)Separation of Concerns:

Definition: Divide the system into distinct sections, each addressing a specific concern or function.

Benefits: Simplifies development, testing, and maintenance by allowing focus on one aspect at a time.

4)High Cohesion:

Definition: Ensure each subsystem has a well-defined purpose and closely related functionalities.

Benefits: Increases the effectiveness and reliability of each subsystem.

5)Low Coupling:

Definition: Minimize dependencies between subsystems.

Benefits: Enhances flexibility and reduces the ripple effect of changes.

6)Interoperability:

Definition: Design subsystems to work seamlessly with other subsystems and the overall system.

Benefits: Ensures smooth integration and interaction within the system.

7)Reusability:

Definition: Create subsystems that can be reused in different systems or contexts.

Benefits: Reduces development time and costs by leveraging existing components.

8)Scalability:

Definition: Design subsystems to handle increasing loads or expanding functionalities without performance degradation.

Benefits: Ensures the system can grow and adapt to future needs.

9)Reliability:

Definition: Ensure subsystems are dependable and perform consistently under specified conditions.

Benefits: Enhances the overall robustness and trustworthiness of the system.

10)Performance:

Definition: Optimize subsystems for efficient operation and resource usage.

Benefits: Improves the overall performance and efficiency of the system.

**Top-down approach and bottom up approach of subsystem development.**

**1)Top-Down Approach**

Definition: The top-down approach starts with the highest level of the system and breaks it down into smaller, more manageable subsystems. This method emphasizes understanding the overall system before defining its components.

Process:

1)System Analysis:

Begin with a high-level understanding of the entire system's requirements and goals.

Define the system's architecture and major functions.

2)Decomposition:

Break down the system into smaller subsystems or modules.

Continue decomposing each subsystem into smaller components until reaching a manageable level of detail.

3)Specification:

Define the requirements and interfaces for each subsystem.

Ensure that each subsystem aligns with the overall system's objectives and architecture.

4)Development:

Implement each subsystem according to the defined specifications.

Integrate subsystems progressively to form the complete system.

5)Testing and Validation:

Test each subsystem individually and in combination with others.

Validate that the integrated system meets the original requirements.

**Advantages:**

Ensures a comprehensive understanding of the system’s goals and architecture from the outset.

Facilitates alignment of subsystems with overall system objectives.

Enhances control over the system design and integration process.

**Disadvantages:**

Can be less flexible in accommodating changes late in the development process.

May require significant upfront planning and analysis.

2)Bottom-Up Approach

Definition: The bottom-up approach begins with the development of individual components or subsystems, which are then integrated to form the complete system. This method focuses on building complex systems by combining simpler subsystems.

Process:

1)Component Development:

Start by designing and developing the smallest, most basic components or subsystems.

Ensure each component is functional and meets its specific requirements.

2)Subsystem Integration:

Combine individual components to form larger subsystems.

Test and validate each subsystem to ensure proper functionality.

3)System Integration:

Integrate the subsystems progressively to form the complete system.

Address any integration issues that arise.

4)System Testing and Validation:

Test the integrated system to ensure all components work together as intended.

Validate that the final system meets the overall requirements and objectives.

**Advantages:**

Provides flexibility to adapt and make changes during the development process.

Allows early testing and validation of individual components.

Can lead to innovative solutions by focusing on optimizing small parts first.

**Disadvantages:**

May result in integration challenges if subsystems are not designed with the overall system in mind.

Can be less efficient if there is insufficient initial planning and coordination.

Process of Component-Based Development

1)Requirements Analysis:

Identify and define the system’s requirements and functionality.

Determine the components needed to fulfill these requirements.

2)Component Design:

Design each component with a clear purpose and functionality.

Define the interfaces and interactions between components.

3)Component Development:

Develop each component as a standalone unit.

Ensure that each component adheres to its specified interface and functionality.

4)Component Testing:

Test each component independently to verify its functionality and reliability.

Use unit tests and integration tests to ensure components work as expected.

5)Component Integration:

Assemble the individual components into the larger system.

Ensure components interact correctly through their defined interfaces.

6)System Testing and Validation:

Perform comprehensive testing on the integrated system.

Validate that the system meets the overall requirements and performs as intended.

**Advantages of Component-Based Development**

1)Efficiency:

Reusable components save time and resources.

Reduces redundancy by leveraging existing components.

2)Scalability:

Systems can be easily scaled by adding or modifying components.

Facilitates incremental development and expansion.

3)Flexibility:

Components can be replaced or updated without impacting the entire system.

Supports adaptive and flexible system architectures.

4)Quality:

Independent testing of components enhances system reliability and quality.

Encourages the use of proven, tested components.

5)Maintainability:

Simplifies maintenance through isolated component updates.

Reduces the complexity of system-wide changes.

**Model-driven development of subsystem development**

Model-driven development (MDD) is a methodology that focuses on creating and exploiting domain models, which are conceptual models that describe the various aspects of a system. The primary goal of MDD is to increase productivity and quality by leveraging these high-level models to drive the development process. When applied to subsystem development, MDD can be particularly beneficial in ensuring that subsystems are well-defined, modular, and integrate seamlessly with other subsystems.

Here are the key aspects of model-driven development in the context of subsystem development:

1. Modeling Subsystems

Domain-Specific Languages (DSLs): Create or use existing DSLs tailored to the specific domain of the subsystem. This allows developers to describe subsystem behaviors, structures, and interactions in a high-level, expressive manner.

Unified Modeling Language (UML): Use UML diagrams (such as class diagrams, sequence diagrams, state diagrams) to model the static structure and dynamic behavior of subsystems.

SysML: For systems engineering applications, the Systems Modeling Language (SysML) extends UML to cover requirements, behavior, structure, and parametrics, making it suitable for complex subsystem modeling.

2. Model Transformation

Code Generation: Automatically generate code from models. For example, a UML class diagram can be transformed into skeletal code in a target programming language.

Model-to-Model Transformation: Convert models from one form to another to refine them or to bridge between different stages of development (e.g., from a platform-independent model to a platform-specific model).

3. Executable Models

Simulation: Create executable models that can be simulated to validate subsystem behavior before actual implementation. This helps in identifying issues early in the development process.

Executable UML (xUML): Develop models that are not only descriptive but can also be executed directly to verify logic and behavior.

4. Model Verification and Validation

Formal Methods: Use formal verification techniques to ensure that the models are correct and conform to their specifications. This might involve model checking or theorem proving.

Testing Models: Create test cases directly from models to verify that the implementation aligns with the intended design. This can include unit tests, integration tests, and system tests derived from the model.

5. Iterative Refinement

Incremental Development: Start with high-level models and progressively refine them into more detailed models. Each iteration should bring the model closer to the final implementation.

Feedback Loop: Use feedback from testing and simulation to refine and improve models continuously.

6. Tool Support

Modeling Tools: Utilize tools like IBM Rational Rhapsody, MagicDraw, Enterprise Architect, or MATLAB/Simulink for creating and managing models.

Integrated Development Environments (IDEs): Use IDEs that support MDD principles, offering seamless integration between modeling, code generation, and testing.

7. Subsystem Integration

Interface Definition: Clearly define interfaces for subsystems to ensure they can interact correctly with other parts of the system.

Interoperability: Ensure that the models consider interoperability standards and protocols to facilitate subsystem integration.

8. Documentation and Communication

Model Documentation: Maintain comprehensive documentation generated from models to ensure that all stakeholders have a clear understanding of the subsystem design and functionality.

Collaborative Modeling: Engage stakeholders through collaborative modeling tools and practices to ensure that models meet the requirements and expectations of all parties involved.

Manual, scripted and continuous deployment

1. Manual Deployment

Manual deployment involves human intervention at each step of the deployment process. This typically includes steps such as preparing the environment, transferring files, configuring settings, and verifying the deployment.

Steps in Manual Deployment:

Preparation: Ensuring all prerequisites (like dependencies) are met.

Transfer: Manually copying files to the target environment (e.g., using FTP or a file share).

Configuration: Manually configuring application settings and environment variables.

Verification: Checking that the application is running correctly.

Advantages:

Simplicity: No need for complex setup or automation scripts.

Control: Direct oversight of each deployment step, allowing for immediate adjustments.

Flexibility: Easy to handle unique or unexpected scenarios.

Disadvantages:

Error-Prone: High potential for human error, leading to inconsistent deployments.

Time-Consuming: Each step requires manual effort, slowing down the process.

Scalability: Difficult to scale for frequent or large-scale deployments.

2. Scripted Deployment

Scripted deployment uses scripts (e.g., shell scripts, batch files) to automate parts of the deployment process. These scripts can perform tasks such as transferring files, setting up environments, and running configuration commands.

Steps in Scripted Deployment:

Create Scripts: Develop scripts to automate repetitive tasks.

Execute Scripts: Run scripts to perform deployment steps.

Monitor: Manually oversee the process and handle any exceptions.

Advantages:

Repeatability: Scripts ensure that the same steps are followed each time, reducing variability.

Efficiency: Automating repetitive tasks saves time and effort.

Reduction of Errors: Less manual intervention reduces the risk of human error.

Disadvantages:

Maintenance: Scripts need to be maintained and updated as the environment or application changes.

Complexity: More complex than manual deployment, requiring scripting skills.

Partial Automation: Some steps may still require manual intervention.

3. Continuous Deployment

Continuous deployment (CD) is an advanced approach that fully automates the deployment process, allowing new code changes to be automatically released to production as soon as they pass automated tests. Continuous deployment is a key practice in DevOps and continuous delivery pipelines.

Steps in Continuous Deployment:

Automated Testing: Implement extensive automated tests to ensure code quality.

Build Pipeline: Set up a CI/CD pipeline that includes building, testing, and deploying the application.

Monitoring and Alerts: Use monitoring tools to track the deployment and detect issues in real-time.

Rollback Mechanisms: Implement automated rollback procedures in case of deployment failures.

Advantages:

Speed: Rapid deployment of new features and fixes, enabling continuous delivery of value.

Reliability: Automated testing and deployment reduce human error and increase consistency.

Feedback: Quick feedback loops allow for fast identification and resolution of issues.

Disadvantages:

Complexity: Requires significant setup and integration of CI/CD tools and practices.

Initial Cost: High initial investment in terms of time and resources to set up the pipeline.

Quality Assurance: Reliance on automated tests means that test coverage and quality are critical.

Summary Comparison

**System requirements elicitation**

**1. Preparation**

Objective Definition: Clearly define the goals of the requirements elicitation process.

Stakeholder Identification: Identify all the stakeholders, including end-users, clients, and other relevant parties.

Resource Allocation: Allocate necessary resources, including tools, personnel, and time**.**

**2. Information Gathering**

Interviews: Conduct one-on-one or group interviews with stakeholders to gather detailed requirements.

Surveys and Questionnaires: Distribute surveys to collect quantitative data on stakeholder needs and preferences.

Observation: Observe end-users in their natural work environment to understand their interactions with current systems.

**3. Document Analysis**

Existing Documentation Review: Analyze current system documentation, business processes, and any other relevant documents to identify existing requirements and constraints.

**4. Workshops and Brainstorming Sessions**

Joint Application Development (JAD) Sessions: Conduct collaborative sessions with stakeholders and development teams to identify and refine requirements.

Brainstorming: Facilitate brainstorming sessions to generate ideas and capture diverse perspectives.

**5. Prototyping**

Low-Fidelity Prototypes: Create mockups or wireframes to visualize requirements and gather feedback.

High-Fidelity Prototypes: Develop more detailed prototypes to refine and validate requirements.

**6. Use Case and Scenario Development**

Use Cases: Define use cases to describe how the system will be used in different scenarios.

User Stories: Create user stories to capture requirements from the perspective of end-users**.**

**7. Modeling Techniques**

Process Modeling: Use flowcharts, data flow diagrams (DFDs), or business process modeling notation (BPMN) to represent processes.

Data Modeling: Develop entity-relationship diagrams (ERDs) or class diagrams to model data requirements.

**8. Requirement Analysis and Validation**

Requirement Prioritization: Prioritize requirements based on factors such as stakeholder value, feasibility, and impact.

Validation and Verification: Ensure that the gathered requirements are complete, consistent, feasible, and testable through reviews and validation sessions.

**9. Documentation**

Requirements Specification Document: Compile all elicited requirements into a structured document.

Stakeholder Review: Present the documented requirements to stakeholders for review and approval.

**10. Continuous Feedback and Refinement**

Iterative Feedback: Continuously seek feedback from stakeholders throughout the development process.

Requirement Refinement: Refine and update requirements based on feedback and changing project needs.