**SWDBS401 SOFTWARE DEVELOPMENT**

**Backend System Design**

**MODULE CODE AND TITLE: SWDBS401 BACKEND SYSTEM DESIGN**

**Key Competencies for Learning Outcome 1: Analyze System Backend**

**Duration: 5hrs**

**Theoretical Activity 1.1.1: Description of FURPS key terms in Backend System**

**Key Terms in Backend System**

**Backend**

* The backend refers to the server-side of an application where data is processed, and business logic is executed.
* It includes server technologies, databases, and other components that are not directly seen by end-users.

**System**

* A system is a set of interconnected components working together to achieve a common goal
* In software context, comprises both frontend (user interface) and backend components
* An organized set of doctrines, ideas, or principles intended to explain the arrangement or working of a systematic whole
* Systems management is the administration of IT systems in an enterprise network or data center
* Example: Shop Management systems and E-learning systems

**Database**

* An organized collection of data stored and accessed electronically
* Designed to efficiently manage, store, retrieve, and manipulate data
* Used in various applications from simple record-keeping to complex information management systems

**Operating System**

* Software that manages hardware and other software on a computer
* Provides services for computer programs and acts as an intermediary between users and computer hardware
* Manages computer hardware and software resources, providing common services for computer programs

**Server**

* At its core, a server is a computer or software system that provides a specific service or resource to other computers, known as clients, over a network.

The term "server" can refer to two things:

* Hardware: The physical computer dedicated to running server software.
* Software: The program or suite of programs that performs the serving function.

**Systems Development Life Cycle (SDLC)**

* The overall process for developing information systems from planning and analysis through implementation and maintenance
* Foundation for all systems development methodologies
* Begins with a business need, ends when benefits no longer outweigh maintenance costs
* Comprised of seven distinct phases: planning, analysis, design, development, testing, implementation, and maintenance

**Framework**

* A pre-built, reusable set of tools, libraries, and conventions that provide a structured way to develop, organize, and deploy backend applications
* Helps streamline development process by offering standard components and features
* Reduces need to write repetitive code and allows focus on unique aspects of applications

**JSON (JavaScript Object Notation)**

* A lightweight data interchange format widely used in backend system design
* Easy for humans to read and write, and easy for machines to parse and generate
* Often used to transmit data between server and client, as well as for configuration files, data storage, and APIs
* Standard text-based format for representing structured data based on JavaScript object syntax

**UML (Unified Modeling Language)**

* Standardized modeling language used in software engineering to visualize, specify, construct, and document components of a system
* Helps developers and architects communicate complex ideas clearly
* Ensures all stakeholders have a shared understanding of the system

**FURPS**

* **FURPS** stands for Functionality, Usability, Reliability, Performance, and Supportability
* Used to categorize software requirements into five key areas to ensure comprehensive assessment of system's needs
  + **Functionality**: Covers features and capabilities
  + **Usability**: Refers to user interface and experience
  + **Reliability**: Relates to system stability and fault tolerance
  + **Performance**: Focuses on speed and efficiency
  + **Supportability**: Involves maintainability and documentation

**API (Application Programming Interface) and Server**

* An API is a set of rules and protocols that allows one software application to interact with another

**Theoretical Activity 1.1.2: Description of System development life cycle (SDLC) models**

**Key readings 1.1.2 Description of System development life cycle (SDLC) models**

**System Development Life Cycle (SDLC) Phases**

* Software Development life cycle (SDLC) is a spiritual model used in project management that defines the stages in an information system development project, from initial feasibility study to maintenance of completed application

**Planning**

* Identifying project scope, goals, risks, and resources required
* Initial project planning and feasibility analysis take place in this phase

**Analysis**

* Understanding and defining system requirements
* Involves studying existing systems, identifying limitations, and proposing solutions

**Design**

* Creating a blueprint for the system based on requirements gathered
* Includes architecture, user interfaces, database design, and overall system structure

**Implementation (Coding)**

* The actual coding of the system based on design specifications
* Developers write code and build system according to approved design

**Testing**

* Systematic testing of developed system to identify and fix defects
* Various testing methods: unit testing, integration testing, user acceptance testing

**Deployment**

* Releasing system for public use
* Involves installing system on servers, configuring for production, and making available to end-users

**Maintenance**

* Post-deployment activities including ongoing support, bug fixes, updates, and improvements
* Based on user feedback and changing requirements

**SDLC Models**

**Waterfall Model**

* Universally accepted SDLC model
* Whole process of software development divided into various phases
* Continuous software development model where development flows steadily downwards through steps
* Linear ordering of activities with certification techniques at end of each step
* Verification and validation ensure output of each stage is consistent with input and overall requirements

**RAD Model**

* Rapid Application Development process
* Adoption of waterfall model targeting developing software in short period
* Based on concept that better system can be developed in lesser time using focus groups
* Components:
  + Business Modeling
  + Data Modeling
  + Process Modeling
  + Application Generation
  + Testing and Turnover

**Spiral Model**

* Risk-driven process model
* Helps group adopt elements of one or more process models
* Combination of rapid prototyping and concurrency in design and development activities
* Each cycle begins with identification of objectives, alternatives, and constraints
* Evaluation of alternatives based on objectives and constraints with focus on risk perception
* Development of strategies to solve uncertainties and risks

**V-Model**

* Testing and development steps planned in parallel
* Verification phases on one side and validation phase on other side
* Joined by Coding phase

**Incremental Model**

* Series of waterfall cycles
* Requirements divided into groups at start of project
* SDLC model followed for each group to develop software
* Process repeated with each release adding more functionality until all requirements met
* Each cycle acts as maintenance phase for previous software release
* Allows development cycles to overlap

**Agile Model**

* Practice promoting continuous interaction of development and testing during SDLC process
* Entire project divided into small incremental builds provided in iterations
* Each iteration lasts from one to three weeks
* Addresses key assumptions:
  + Difficult to predict which requirements will persist and which will change
  + Difficult to predict how user priorities will change
  + Design and development are interleaved
  + Analysis, design, development, and testing are not as predictable as desired

**Iterative Model**

* Implementation of SDLC focusing on initial, simplified implementation
* Progressively gains more complexity and broader feature set until final system complete
* Way of breaking down software development of large application into smaller pieces

**Big Bang Model**

* Focuses on all resources in software development and coding with no or very little planning
* Requirements understood and implemented when they come
* Works best for small projects with smaller development teams
* Useful for academic software development projects
* Ideal when requirements unknown or final release date not given

[Software Development Process in 7 Stages and Types](https://www.excellentwebworld.com/software-development-process/)

**Theoretical Activity 1.1.3: Description of backend development technologies**

**Top 5 Frameworks in Python**

**Django**

* Popular open-source full-stack Python framework
* Includes all necessary Python features by default

**Web2Py**

* Popular open-source full-stack Python framework
* Platform independent, runs on all popular operating systems

**Flask**

* Micro-framework for Python
* Lightweight and easily adaptable to developer's needs
* Comes under BSD(Berkeley Software Distribution) license
* Requires Werzeug WSGI(web server Gateway Interface) toolkit and Jinja2 templates

**Bottle**

* Micro-framework for prototyping and building simple personal applications
* Originally meant for building APIs
* Considered one of finest Python web frameworks
* Allows developers to work closely with hardware to build small and simplistic apps

**CherryPy**

* Open-source Python framework following minimalist approach
* One of oldest Python frameworks still popular (released in 2002)
* No need to install apache server to run CherryPy
* Allows use of any type of technology for creating templates and data access

**PHP and its Frameworks**

**Laravel**

* Among most widely used PHP frameworks
* Offers benefits including robust security and authentication features
* Known for easy-to-read syntax, powerful features, and developer-friendly environment
* Features: routing, ORM(Object-Relational Mapping), authentication, caching, and more
* Active community ensures regular updates and excellent documentation
* Robust testing capabilities ensure bug-free web solutions

**CodeIgniter**

* Open-source rapid development framework for dynamic web solutions
* Known for simplicity, speed, and small footprints
* Intuitive MVC pattern simplifies web development
* Features: database abstraction, caching, security, session management
* Extensive library collection enables developers to perform common tasks effortlessly

**Symfony**

* Highly flexible PHP framework using modular component system
* Recommended for large-scale applications with complex requirements
* Used by major companies like BlaBlaCar, Spotify, Magento
* Perfect choice for enterprise-level web projects
* Reusable components save time
* Modular architecture allows developers to pick components based on requirements
* Strong emphasis on testing and robust community support

**Cake PHP**

* First PHP MVC(Model, View, Controller) framework created, still widely used
* Helps develop web solutions with visually impressive features
* Emphasizes convention over configuration approach
* Built-in features: ORM, validation, caching
* Good choice for commercial web applications due to security features
* Offers scaffolding feature that automates code generation

**Phalcon**

* Full-stack PHP framework with source code written in C and C++
* Implemented as web server extension in C language
* Known for speed and low overhead cuts
* Well-documented and easy to use
* Architecture focuses on performance optimization and minimal resource consumption
* Compiles components into PHP extensions for lightning-fast execution
* Efficient memory usage optimizes web application performance
* Supports MySQL, PostgreSQL, and SQLite natively

***JAVA and its Frameworks***

* *Java is predominant programming language for groundbreaking software development*
* *Frameworks include: Spring, Hibernate, Struts, Wicket, GWT, Dropwizard, Play, Vaadin, Blade, Grails*

**JavaScript and its frameworks**

**Angular**

* Framework written in TypeScript and developed by Google
* Open-source web application framework for developing single-page applications (SPA)

**React**

* Introduced in 2013 by Facebook
* Reusable framework for building interactive user interfaces

**Vue.js**

* Free, open-source JavaScript framework
* Progressive front-end framework
* Approachable, easy to learn, suitable for small and large applications

**Node.js**

* Open-source runtime environment built to execute JavaScript outside web browser

**Polymer**

* JavaScript library developed by Google
* Open-source framework suited for Single Page Applications
* Supports both one-way and two-way data binding
* Helps create custom elements

**Backbone.js**

* JavaScript framework used for Single Page Applications
* Popular due to features allowing complex functions with lesser codes
* Debugging could be an issue at times

**Ruby and its Framework**

**Ruby on Rails**

* Most popular Ruby framework for web development
* One of most well-known web development frameworks regardless of language

**Sinatra**

* More of a DSL than full-fledged framework
* Allows quick creation of small web applications with minimal effort

**Hanami**

* Ruby on Rails alternative for building feature-rich, full-stack web apps
* MVC framework supporting features like routing, controllers/actions, models, views, migrations, validations, mailers, and assets

**Grape**

* Used to add REST API capabilities to existing web apps
* Example: adding REST API to full-stack Rails web app for Android app development

**Cuba**

* Micro framework for web development
* Lightweight and offers minimal features

**System Analysis Tools**

* System Analysis: "process of studying a procedure or business to identify its goals and purposes and create systems and procedures that will achieve them efficiently"

**Tools and Techniques of System Analysis**

**Grid Charts**

* Used to represent relationship between two sets of factors in tabular method
* Helpful in eradicating unnecessary reports or data items
* Can identify responsibilities of various managers for particular sub-system
* Effectively used to trace flow of transactions and reports in organization

**System Flow Chart**

* Pictorial or diagrammatic representation of logical flow of operations and information
* Depicts clear relationship between input processing and output considering entire system
* Uses standard symbols for construction

**Decision Tree**

* Model for decisions involving series of steps where outcome of first decision guides second
* Deals with decision making uncertainty
* Important in probabilistic situations where various alternatives can be drawn as branches of tree

**Simulation**

* Describes operation of system in terms of individual events and components
* Involves development of mathematical model rather than directly describing behavior of overall system
* System divided into elements whose behavior is predicted in terms of probability distributions

**Theoretical Activity 1.1.4: Description of data gathering and identification FURPS Requirements**

**Data Types**

**Qualitative data**

* Can't be measured or expressed as number
* Less structured than quantitative data
* Information acquired to understand underlying motivations - answering "how" and "why" questions
* Descriptive in nature, consists of words, pictures, or symbols
* Example questions: "How do you feel about using products from XYZ brand?" or "Why is product A your favorite?"

**Quantitative data**

* Structured and can be analyzed statistically
* Expressed in numbers, can be used to measure variables
* Results are objective and conclusive
* Questions usually "how many," "how much," or "how often?"
* Can be measured by numerical variables, analyzed through statistical methods, represented in charts and graphs
* Example questions: "How often do you purchase laundry detergent?" with options like once weekly, every two weeks, etc.

**First-party data**

* Collected directly from research participants
* Valuable because gathered straight from sources, eliminating misinterpretation and errors
* Most useful and reliable data for research
* Sources: survey responses, web analytics, social media analytics, reviews, email analytics, interviews, focus groups, experiments, observations
* Information includes demographics, purchasing behaviors, interests, habits, likes, dislikes

**Second-party data**

* Data already collected by someone else in the past
* Less reliable because methodology of data collection uncertain
* Performed with different hypothesis, analyses may not align with research needs
* Sources: previous research, books, professional journal publications, websites, libraries, newspapers, public records
* May be collected before primary data to find knowledge gaps or augment primary research

**Third-party data**

* Data sets put together from various sources
* Usually gathered by companies without direct relationships with consumers
* Often sold on data marketplaces

**Data gathering process**

**Define Objectives**

* Clearly outline purpose of gathering data, specifying what needs to be measured or analyzed
* Helps focus data collection on relevant information

**Identify Data Sources**

* Determine where data will come from:
  + Primary Data: Collected directly from original sources through surveys, interviews, observations, experiments
  + Secondary Data: Gathered from existing records, databases, published materials (reports, articles, public databases)

**Choose Data Collection Methods**

* Select appropriate techniques:
  + Surveys/Questionnaires: Structured or semi-structured forms with questions
  + Interviews: Direct questioning of individuals or groups
  + Observations: Monitoring behaviors, events, conditions in natural setting
  + Experiments: Conducting controlled tests
  + Document/Record Review: Analyzing existing data from books, reports, online sources

**Design Data Collection Instruments**

* Develop tools (questionnaires, checklists) that are valid and reliable

**Pilot Testing**

* Test data collection instruments on small scale to identify issues before full-scale collection

**Collect Data**

* Implement chosen methods and gather actual data
* Requires organization, consistency, adherence to ethical guidelines

**Data Validation**

* Ensure data collected is accurate, complete, consistent
* May involve cross-checking responses or statistical methods to detect errors

**Store and Organize Data**

* Properly store data in structured format (databases, spreadsheets)
* Security and confidentiality important considerations

**Analyze Data**

* Analyze data to derive insights, patterns, trends
* May involve statistical analysis, qualitative coding, or other methods

**Report Findings**

* Present results in clear, accessible way through written reports, visualizations, presentations

**Identification of FURPS Requirements**

**Functionality Requirements**

* Specify what system should do
* Define features and capabilities software must provide to meet user needs
* Key Aspects:
  + Features: Specific functionalities (user authentication, data processing)
  + Business Rules: Constraints and conditions governing system operation
  + Use Cases: Scenarios detailing user interaction with system
  + Interoperability: Requirements for system integration with other systems/services

**Usability Requirements**

* Focus on how easy and intuitive software is for users
* Encompass user experience and interface design
* Key Aspects:
  + Learnability: How quickly users can learn to use system
  + Efficiency: How effectively users can perform tasks once familiar
  + Memorability: How easily users can re-engage after period of not using
  + Satisfaction: Overall user satisfaction with interface and experience

**Reliability Requirements**

* Pertain to system's ability to perform intended functions consistently over time
* Include stability and fault tolerance
* Key Aspects:
  + Availability: Proportion of time system is operational and accessible
  + Failure Rate: Frequency of system failures or errors during operation
  + Recoverability: Ability to recover from failures and restore data
  + Consistency: Stable and predictable performance under varying conditions

**Performance Requirements**

* Specify speed, responsiveness, and overall efficiency of system
* Crucial for ensuring system meets user expectations
* Key Aspects:
  + Response Time: Time for system to respond to user inputs or requests
  + Throughput: Amount of work system can handle in given time period (transactions per second)
  + Resource Usage: Efficiency in utilizing resources like CPU, memory, bandwidth
  + Scalability: Ability to handle increased loads by scaling up or out

**Practical Activity 1.1.5: Conducting data gathering (See in the notes some other)**

**Pilot Testing**

* Test instruments on small sample to identify issues

**Collect Data**

* Implement Methods: Apply chosen data collection methods systematically
* Maintain Consistency: Ensure data collected uniformly to avoid biases

**Data Verification**

* Check Accuracy: Review collected data for completeness and correctness
* Address Issues: Resolve identified inconsistencies or errors

**Analyze Data**

* Data Processing: Organize and prepare data for analysis
* Statistical Analysis: Apply appropriate statistical methods to interpret data

**Report Findings**

* Summarize Results: Compile findings into clear, concise report
* Visual Representation: Use charts or graphs to illustrate key points

**Review and Reflect**

* Evaluate Process: Assess effectiveness of data gathering process
* Identify Improvements: Consider how process can be enhanced for future projects

**Points to Remember**

* Data gathering is first step in research, involving collecting and analyzing information
* Data can be qualitative or quantitative
* First-party data collected directly from sources
* Second-party data collected by others
* Third-party data collected by companies and sold on marketplaces
* Backend refers to server-side of web applications, built using various technologies
* Python frameworks like Django, Flask, and Web2Py popular for backend development
* Other popular languages include PHP and Java with frameworks like Laravel, Symfony, and Spring
* Tools and techniques of system analysis include: Grid charts, System flowcharts, Decision trees, Simulation, Decision tables
* Grid charts represent relationships between factors
* System flowcharts depict logical flow of operations
* Decision trees model decision making with uncertainty
* Simulation involves developing mathematical model to describe system behavior
* Decision tables represent logical decisions in tabular form
* FURPS is acronym for functionality, usability, reliability, performance, and security
* Usability requirements relate to user interface and experience
* Reliability requirements focus on system dependability
* Performance requirements concern system efficiency
* Security requirements address system protection

**Theoretical Activity 1.2.1: Description the Scope of Backend System**

**Duration: 4 hrs**

**Key readings 1.2.1.: Description the Scope of Backend System**

**Define the Scope of Backend System**

* Determining specific functionalities and features system will encompass
* Identifying data processing, storage, and retrieval capabilities
* Integration with other systems and services
* Generally includes:
  + Data management and storage
  + User authentication and authorization
  + Business logic implementation
  + Integration with external APIs and services
  + Performance monitoring and optimization
  + Security and compliance measures

**Database**

* Organized collection of structured information stored electronically in computer system
* Usually controlled by database management system (DBMS)

**Types of databases**

**Relational databases**

* Became dominant in 1980s
* Items organized as set of tables with columns and rows
* Provide efficient and flexible way to access structured information

**Object-oriented databases**

* Information represented in form of objects, as in object-oriented programming

**Distributed databases**

* Consist of two or more files located in different sites
* May be stored on multiple computers in same physical location or scattered over different networks

**Data warehouses**

* Central repository for data
* Type of database specifically designed for fast query and analysis

**NoSQL databases**

* Nonrelational database allowing unstructured and semi-structured data to be stored and manipulated
* Grew popular as web applications became more common and complex

**Graph databases**

* Store data in terms of entities and relationships between entities

**OLTP databases (Online Transaction Processing)**

* Speedy, analytic database designed for large numbers of transactions by multiple users

**Other database types**

* Open-source databases: Source code open source, could be SQL or NoSQL
* Cloud databases: Collection of data on private, public, or hybrid cloud platform
  + Two types: traditional and database as a service (DBaaS)
* Multimodel databases: Combine different types of database models into single integrated back end
* Document/JSON databases: Store data in JSON format rather than rows and columns
* Self-driving databases: Cloud-based using machine learning to automate database tuning, security, backups, updates, and other management tasks

**APIs**

* Application Programming Interface
* "Application" refers to any software with distinct function
* "Interface" is contract of service between two applications

**Servers**

* Computer program or device providing service to another computer program and user (client)
* Physical computer running server program also referred to as server

**Frameworks**

* Backend frameworks are server-side frameworks designed to make tasks easier for developers
* Provide tools, libraries, and components to create framework for website or application
* Automate aspects of web development, making it faster and simpler

**Points to Remember**

* Defining scope of backend system involves determining specific functionalities and features
* Includes identifying data processing, storage, retrieval capabilities, and integration with other systems
* Database is organized collection of structured information controlled by DBMS
* API is software intermediary allowing two applications to communicate
* Server provides service to another computer program and user
* Backend frameworks provide tools, libraries, and components to create website or application framework

**Theoretical Activity 1.3.1: Description of System Interaction**

**Duration: 3 hrs**

**Key readings 1.3.1. Description of System Interaction**

**Purpose of System Interaction**

* Interactions between systems are necessity, source of opportunity, and source of difficulty
* Essential in building, implementing, and maintaining IT-reliant systems in organizations

**Main components of System Interaction**

**Web server**

* Computer system capable of delivering web content to end users over internet via web browser

**How web servers work**

* End user processes request via web browser installed on web server
* Communication between web server/browser and end user uses Hypertext Transfer Protocol (HTTP)
* Primary role: store, process, and deliver requested information or webpages to end users
* Uses Physical Storage: All website data stored on physical web server to ensure safety
* When user enters URL or searches using keyword, request generated and sent to web server

**Application of webserver**

* Process and manage HTTP/HTTPS requests and responses from client system
* Other functions:
  + Store and protect website data from unauthorized users
  + Control bandwidth to regulate network traffic and eliminate downtime from high traffic
  + Server-side web scripting: create dynamic web pages using Ruby, Python, PHP
  + Virtual hosting: run multiple applications, websites, data, and services

**Web browser**

* Role of browsers like Firefox, Chrome, Internet Explorer is to find web server where website data located
* Once browser finds server, reads request and processes information

**Application Server**

* Modern form of platform middleware
* System software residing between operating system, external resources, and users' applications
* Acts as host for user's business logic while facilitating access to and performance of business application

**Database Server**

* Type of hardware that runs database software
* Database software helps store, manage, retrieve, update or change files, information logs and digital data

**External Services and API**

* Services supplied to Company from external suppliers
* Services and interfaces provided by third-party applications and platforms
* Allow developers to integrate applications with these services
* Examples:
  + Google Maps API - allows integration of Google Maps into applications
  + Facebook API - provides access to Facebook's social graph and other features for developers
  + Twitter API - allows developers to access Twitter's data and functionality
  + Amazon Web Services - provides a wide range of cloud-based services and APIs for developers

**Message queues or event Streams**

They focus on the delivery of messages. Once a message is delivered to its

supposed recipients, the mission is accomplished, and the message is usually

deleted.

**Event streaming services**

* Messages are stored persistently (stored for a long time).
* The focus is on the sequence of the messages when they are saved. Messagesare only appended to the end of the stream.

**Points to Remember**

* Interactions between systems are necessity, source of opportunity, and difficulty in building IT-reliant systems
* Main components of System Interaction:
  + Web server
  + Application Server
  + Database Server
  + External Services and API
  + Message queues or event Streams

**Theoretical Activity 1.4.1: Reporting the System Backend Requirements.**

**Duration: 5 hrs**

**Key readings 1.4.1.: Report of the System Backend Requirements.**

**Report Structure**

**Executive Summary**

* Purpose: High-level overview of entire report, summarizing main findings, conclusions, recommendations
* Content: Brief background, objectives, current state summary, key gaps/issues, proposed recommendations summary, anticipated impact

**Analysis of the Current State**

* Purpose: Detailed examination of existing backend architecture, technologies, processes, performance metrics
* Content:
  + Architecture Overview: Description of current system including hardware, software, databases, APIs
  + Technology Stack: List and description of technologies and tools in use
  + Performance Metrics: Current performance data (load times, response times, uptime, KPIs)
  + Scalability and Reliability: Analysis of system's ability to handle growth and reliability
  + Security Posture: Overview of current security measures and protocols

**Findings on Gaps and Issues**

* Purpose: Identify shortcomings, inefficiencies, or risks within current system
* Content:
  + Technical Gaps: Areas where technology/architecture doesn't meet needs or best practices
  + Performance Issues: Specific problems (slow response times, bottlenecks, frequent downtimes)
  + Security Vulnerabilities: Identified security risks or compliance issues
  + Scalability Concerns: Issues hindering system's ability to scale effectively
  + Operational Challenges: Problems in day-to-day management or maintenance

**Recommendations**

* Purpose: Propose actionable solutions to address gaps and issues
* Content:
  + Technical Upgrades: Suggestions for updating/replacing technologies
  + Process Improvements: Recommendations for optimizing operational processes
  + Performance Enhancements: Strategies to address performance bottlenecks
  + Security Enhancements: Steps to strengthen security measures
  + Scalability Solutions: Approaches to ensure efficient scaling
  + Training and Skill Development: Recommendations for team training

**Points to Remember**

While Reporting the System Backend Requirements:

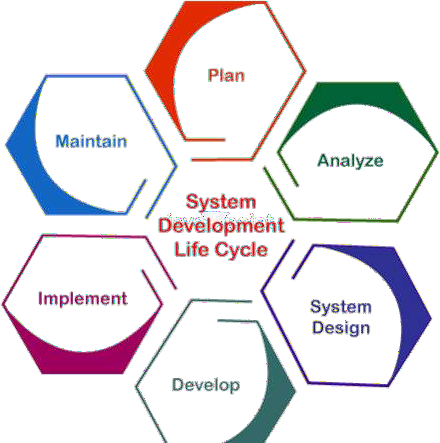
* Develop executive summary
* Detailed analysis of the Current State
* Findings on Gaps and issues
* Write recommendations

**Learning Outcome 2: Develop System Structure**

**Key readings 2.1.1. Identification of system design tools**

**1. Introduction to System Design**

**System Design** is the comprehensive process of conceiving, defining, and creating the architecture, structure, and components of a system to meet specified criteria, functions, and purposes. It involves translating user needs and constraints into a detailed blueprint or model that specifies how different components or modules will work together cohesively to achieve the intended results.



During this design phase, system architects must consider critical elements such as:

* Performance
* Scalability
* Dependability
* Security
* Usability

**2. Key Components of System Design**

The system design process encompasses several important components:

1. **Understanding and Documenting Requirements:** Eliciting and documenting the system's needs, goals, and constraints through stakeholder interactions.
2. **Architectural Design:** Creating the system's high-level structure, defining its main components, and describing the links and interactions between them.
3. **Detailed Design:** Elaborating on the architectural design by specifying precise requirements for individual components, including algorithms, databases, and interfaces.
4. **Prototyping and Testing:** Building models or prototypes to validate the design, uncover flaws, and ensure the system's functionality matches requirements.
5. **Implementation:** Creating the system based on the finalized design, which includes coding, integration, and testing to guarantee proper operation.

**3. The System Design Process: Major Steps**

The process typically follows these major steps:

1. **Requirements Analysis:** Understanding and obtaining needs from stakeholders, including system goals, features, restrictions, and user demands.
2. **System Architecture:** Creating the system's high-level blueprint, specifying components, their relationships, and their interactions to deliver the intended functionality.
3. **Precise (Detailed) Design:** Extending the system architecture with detailed specifications for each component, such as algorithms, data structures, and interfaces.
4. **Prototyping and Testing:** Building prototypes to validate the design and functionality, helping to identify and correct potential errors before full-scale implementation.

**4. Common System Design Principles**

1. **Meeting Stakeholder Needs:** The fundamental goal is to understand and fulfill user demands, functions, and objectives.
2. **Scalability:** The system's capacity to grow and handle increased workloads without major changes or performance loss.
3. **Reliability:** Building robust and resilient systems that operate consistently and accurately, with minimal failure and graceful error recovery.
4. **Performance Optimization:** Ensuring the system operates efficiently by optimizing speed, response times, resource usage, and overall throughput.
5. **User Experience (UX) and Usability:** Developing user-friendly, intuitive interfaces and interactions that meet user needs.
6. **Security:** Implementing measures to protect against unauthorized access, data breaches, and other threats to ensure data integrity, confidentiality, and availability.

**5. Advantages and Disadvantages of System Design**

**Advantages:**

1. **Efficiency:** Well-designed systems maximize resource use and enhance performance.
2. **Scalability:** Properly built systems can grow to handle rising demands without significant reengineering.
3. **Reliability:** The process aims to produce durable systems, lowering the risk of failures.
4. **Customization:** Systems can be tailored to specific requirements, improving usability.

**Disadvantages:**

1. **Complexity:** Complex designs can be difficult to understand, increasing development time and error potential.
2. **Cost:** Elaborate designs may require more time, money, and expertise to build and maintain.
3. **Over-engineering:** Excessively complex designs can lead to inefficiencies and unused features.
4. **Rigidity:** Highly specified systems may struggle to adapt to unforeseen changes.
5. **Dependency:** Tightly integrated components can create problems if one fails or needs modification.

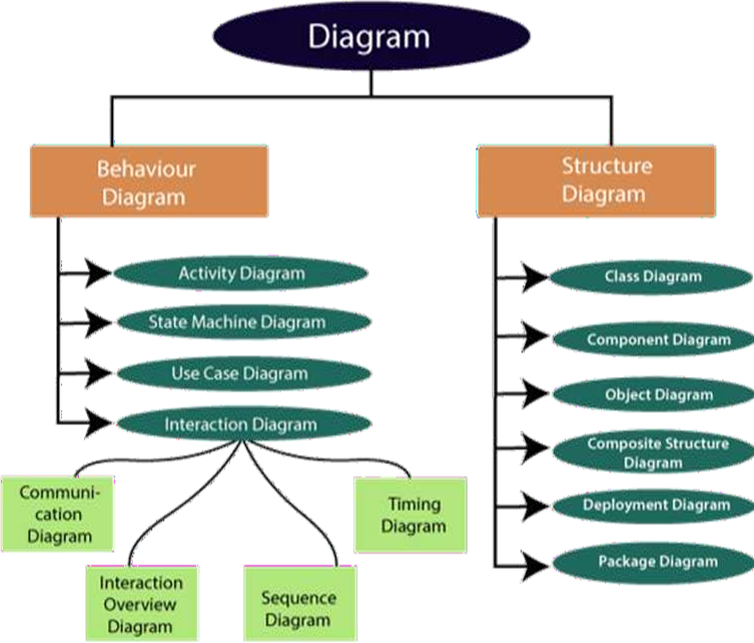
**6. Essential System Design Tools**

System design tools are crucial for planning, visualizing, and structuring software systems effectively.

**6.1. UML (Unified Modeling Language)**

* **Overview:** A standardized modeling language used to visualize system design through various diagrams (e.g., class diagrams, sequence diagrams, use case diagrams).
* **Use Cases:** Software architecture design, object-oriented programming, and system blueprinting.
* **Tools:** Lucidchart, [Draw.io](https://draw.io/), StarUML, Visual Paradigm, PlantUML, Microsoft Visio.

UML diagrams are categorized as follows:



**A. Structural Diagrams** (Static view of a system's structure)

* **Class Diagram:** Shows the system's classes, attributes, methods, and relationships. The backbone of object-oriented systems.
* **Composite Structure Diagram:** Shows the internal structure of a class, including parts and their configuration.
* **Object Diagram:** Depicts a snapshot of the system's instances at a specific point in time.
* **Component Diagram:** Illustrates the organization of physical software components.
* **Deployment Diagram:** Shows the system's hardware and the software components running on each node.
* **Package Diagram:** Organizes elements into packages and shows dependencies between them.

**B. Behavioral Diagrams** (Dynamic view of a system's behavior)

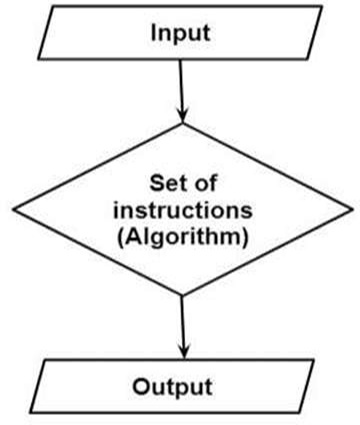
* **Use Case Diagram:** Represents system functionality from a user's perspective, showing actors and use cases.
* **Activity Diagram:** Models the flow of control from one activity to another, depicting workflows.
* **State Machine Diagram:** Portrays the behavior of an entity through finite state transitions in response to events.

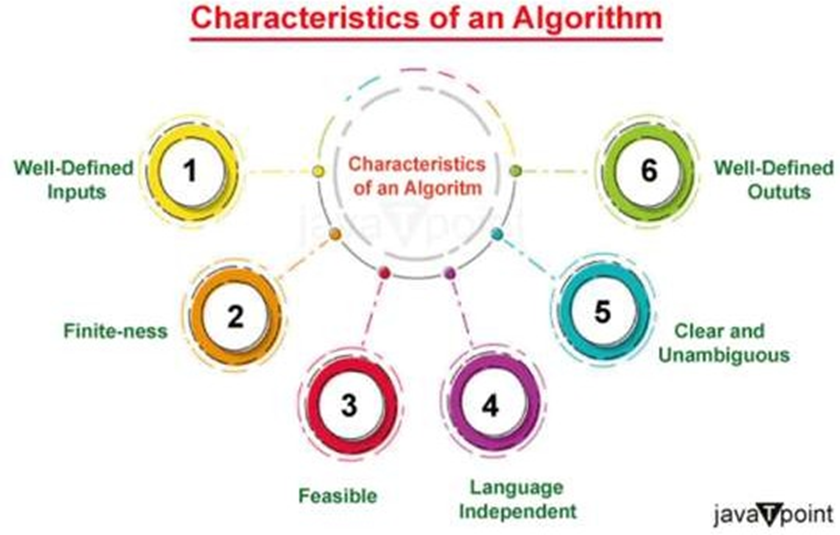
**C. Interaction Diagrams** (A subset of behavioral diagrams focusing on object interactions)

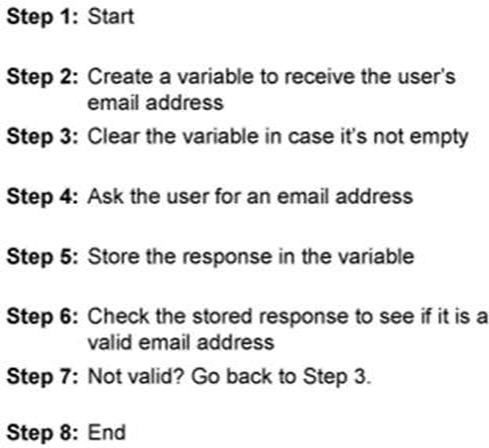
* **Sequence Diagram:** Shows how objects interact in terms of messages exchanged over time.
* **Communication Diagram:** Focuses on the relationships between objects and their message interchange.
* **Timing Diagram:** Depicts object behavior over a specific period, emphasizing timing constraints.
* **Interaction Overview Diagram:** A high-level diagram that combines elements of activity and sequence diagrams.

**6.2. Algorithms and Flowcharts**

* **Algorithm:** A set of defined steps designed to perform a specific objective.

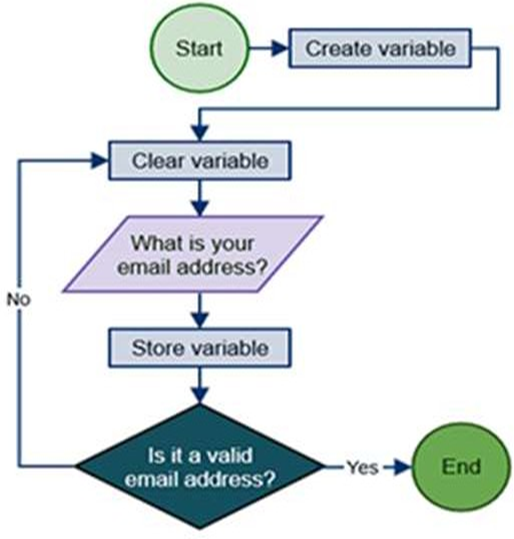






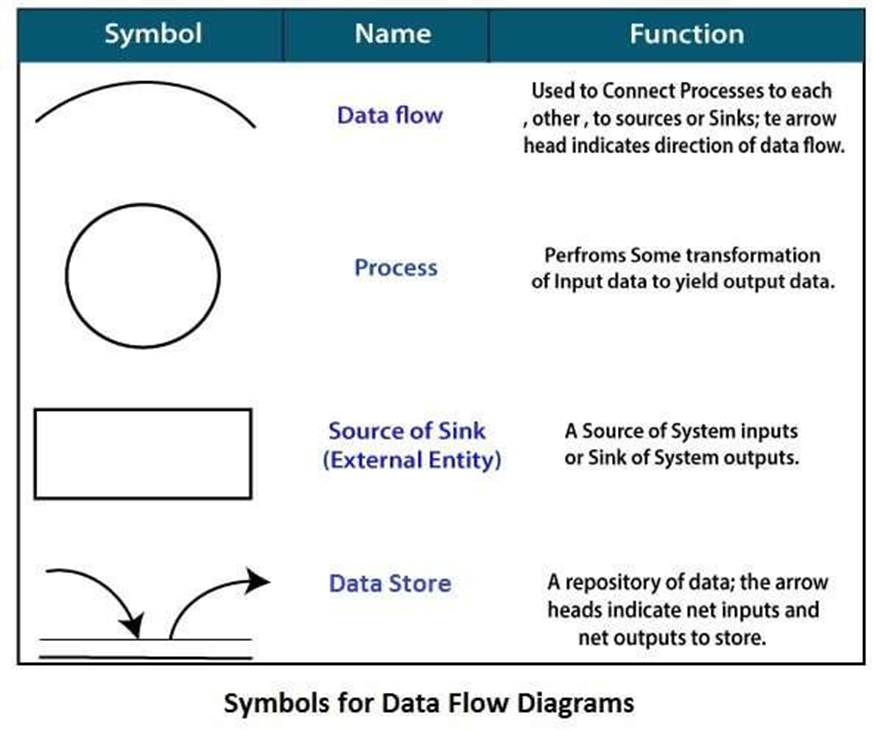
How we write an Algorithm

* **Flowchart:** A diagram that represents a process, showing steps as boxes of various kinds and their order with connecting arrows.
  + *Example:* A flowchart for a user login process.



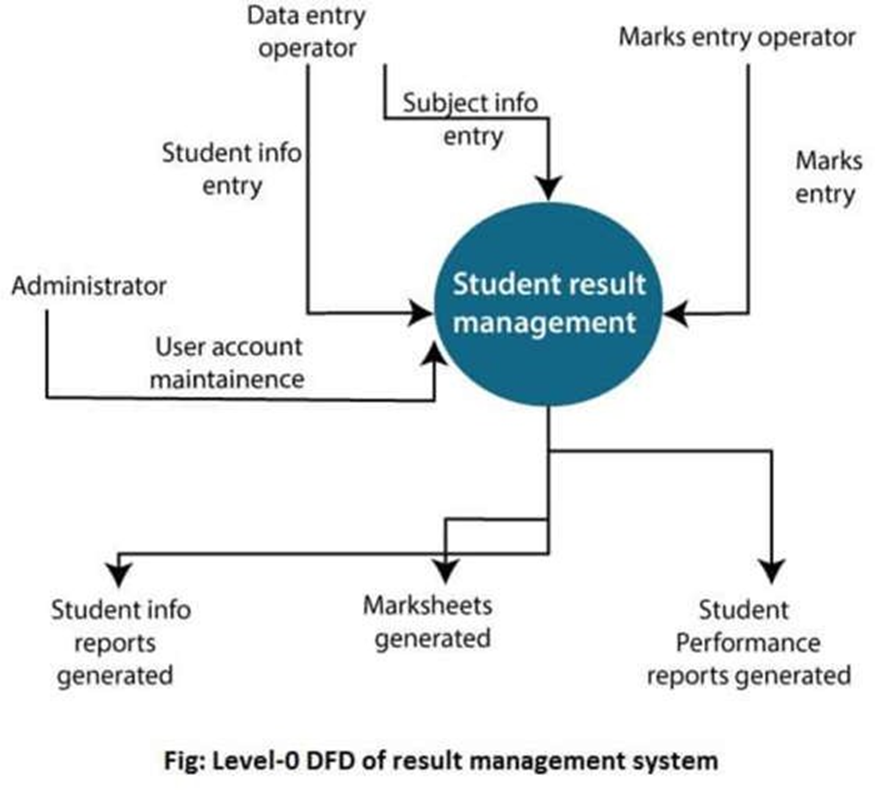
**6.3. Data Flow Diagram (DFD)**

* **Definition:** A visual representation of the information flows within a system, showing how data enters and leaves the system, where it is processed, and where it is stored.
* **Key Rules:**
  1. All names should be unique.
  2. DFDs show data flow, not the order of events (unlike flowcharts). Arrows is a flow chart that represents the order of events; arrows in DFD represent flowing data. A DFD does not involve any order of events.
  3. Suppress logical decisions (no diamond-shaped decision boxes).
  4. Defer error handling details until later stages.
* **Standard Symbols:**
  1. **Process (Circle):** Transforms data inputs into outputs.
  2. **Data Flow (Arrow):** Shows the direction of data movement.
  3. **Data Store (Parallel Lines):** A repository for data.
  4. **External Entity (Square):** A source or destination of data outside the system.



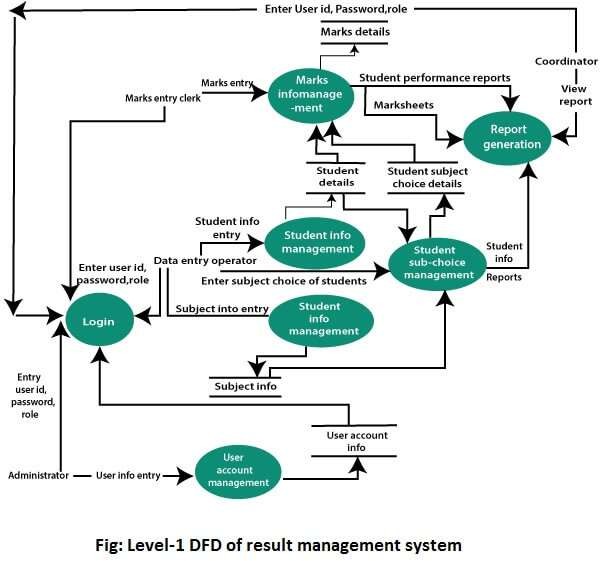
* **DFD Levels:**
  1. **Level 0 (Context Diagram):** A single bubble representing the entire system. It is essential to preserve the number of inputs and outputs between levels; this concept is called leveling by DeMacro.

The Level 0 DFD: also called Context diagram of the result management system is shown in fig. As the bubbles are decomposed into less and less abstract bubbles, the corresponding data flow may also be needed to be decomposed.



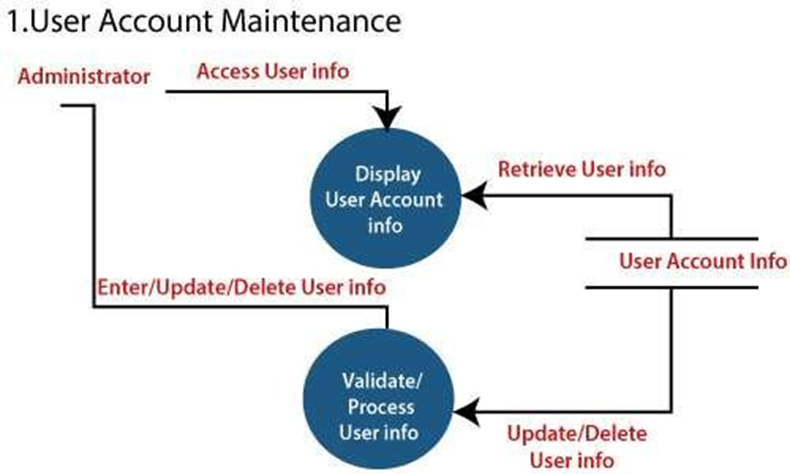
* 1. **Level 1 DFD:** Decomposes the major processes of the Level 0 DFD.

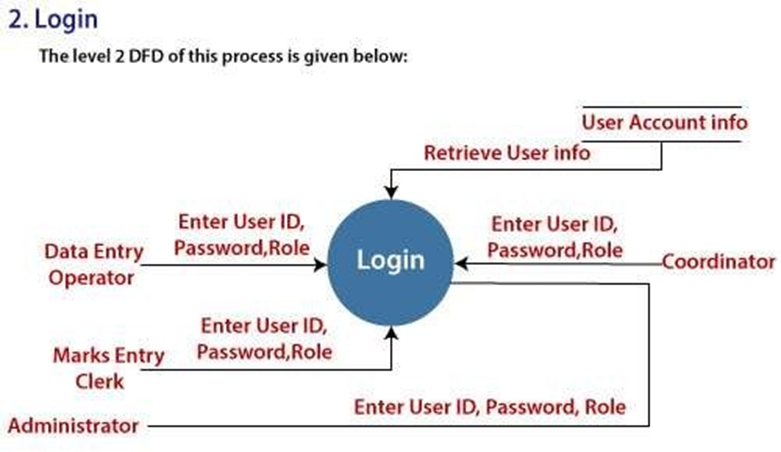
In 1-level DFD, a context diagram is decomposed into multiple bubbles/processes. In this level, we highlight the main objectives of the system and breakdown the high-level process of 0-level DFD into sub processes.

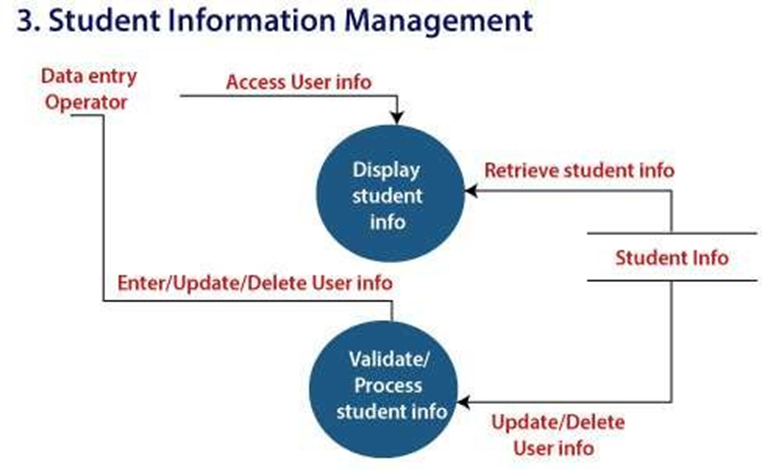


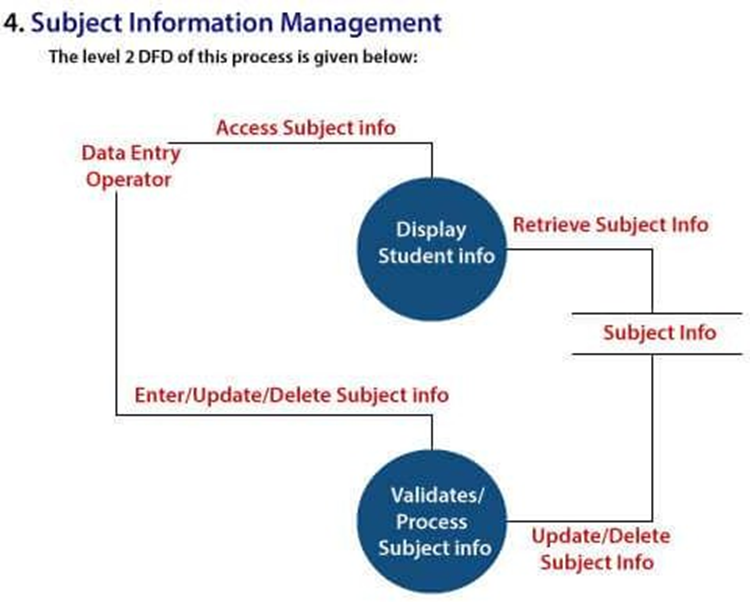
* 1. **Level 2 DFD:** Provides further detail by breaking down Level 1 processes.
* **2-Level DFD**

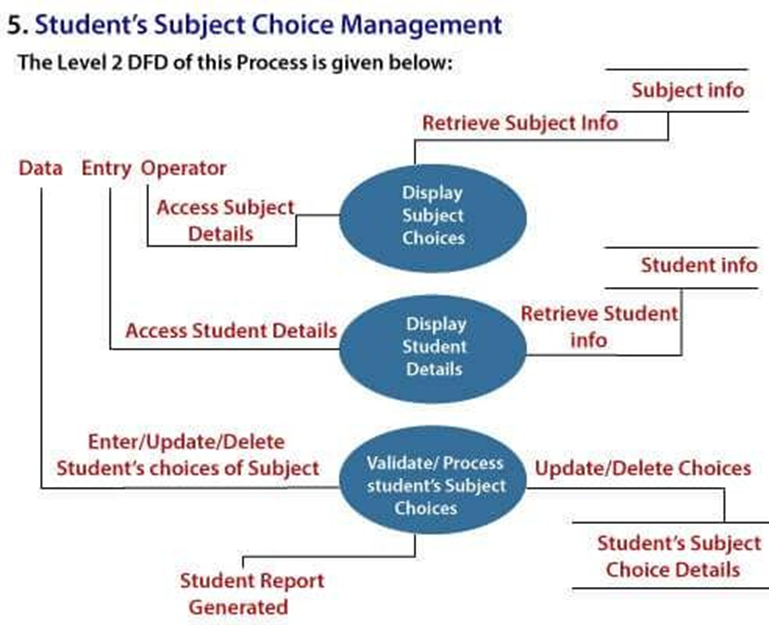
2-level DFD goes one process deeper into parts of 1-level DFD. It can be used to project or record the specific/necessary detail about the system's functioning.

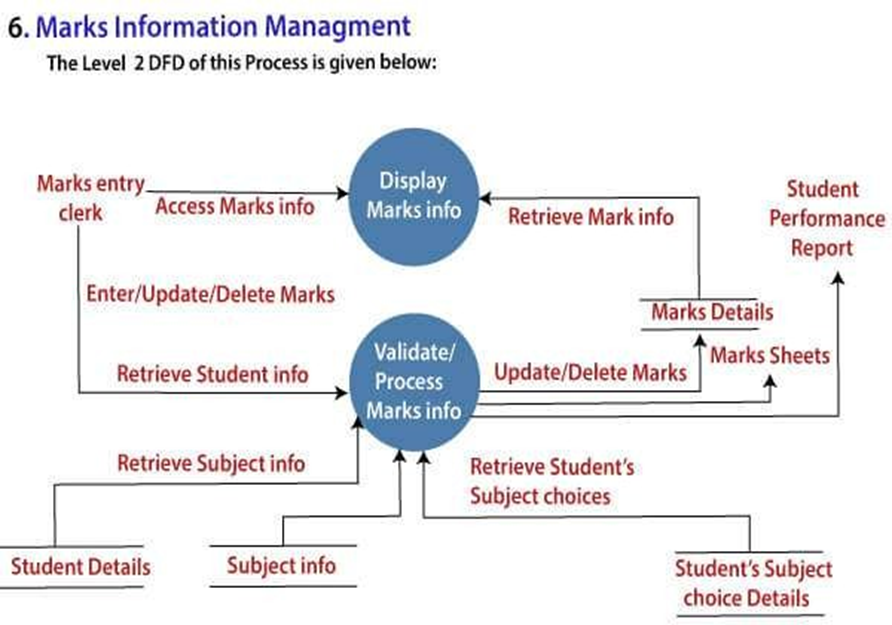
****

****

****

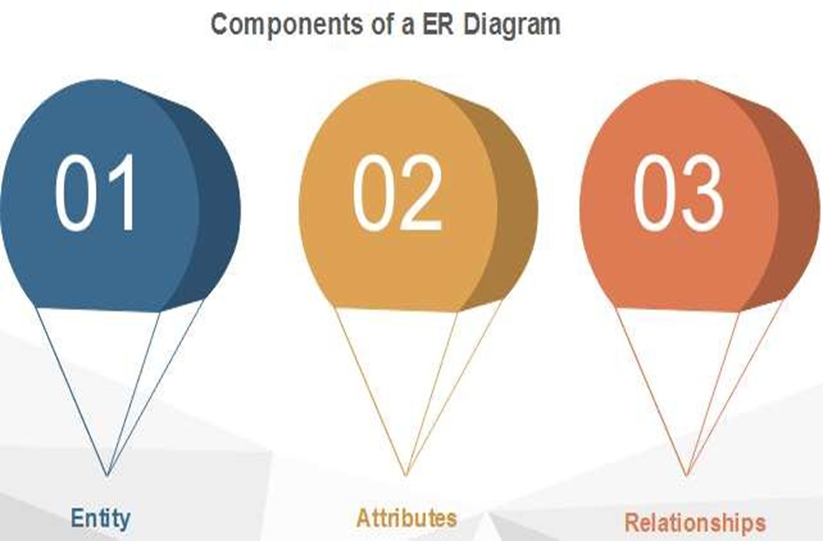
****

****

****

**6.4. Entity-Relationship Diagrams (ERD)**

* **Purpose:** Used in data modeling to produce a conceptual data model of an information system. It helps analysts understand data requirements and communicates the database logic to users.

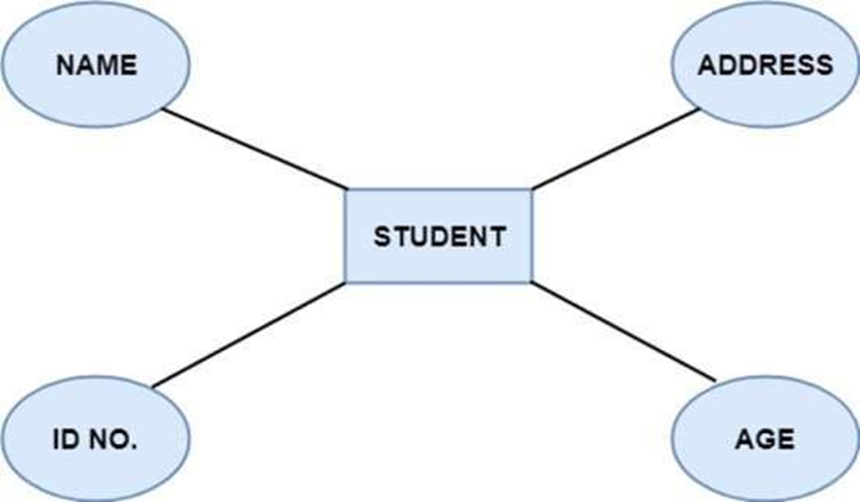


* **Components:**
  + **Entity:** A real-world object (e.g., Student, Teacher). Represented by a rectangle.
  + **Entity Set:** A collection of related entities.

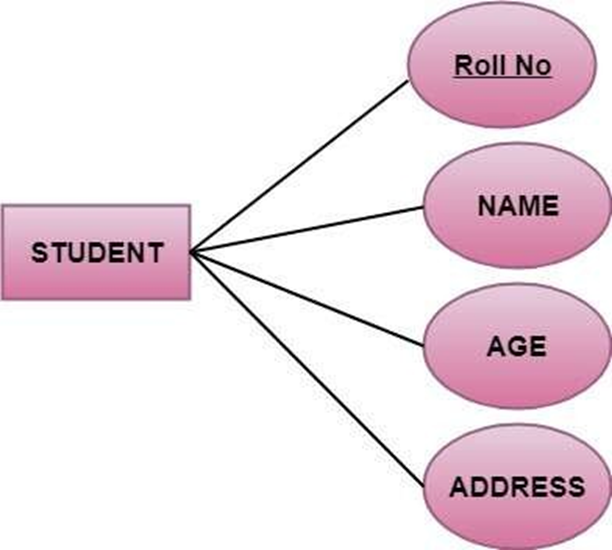
For example, a Student set may contain all the students of a school; likewise, a Teacher set may include all the teachers of a school from all faculties. Entity set need not be disjoint.



* + **Attributes:** Properties of an entity (e.g., name, age).

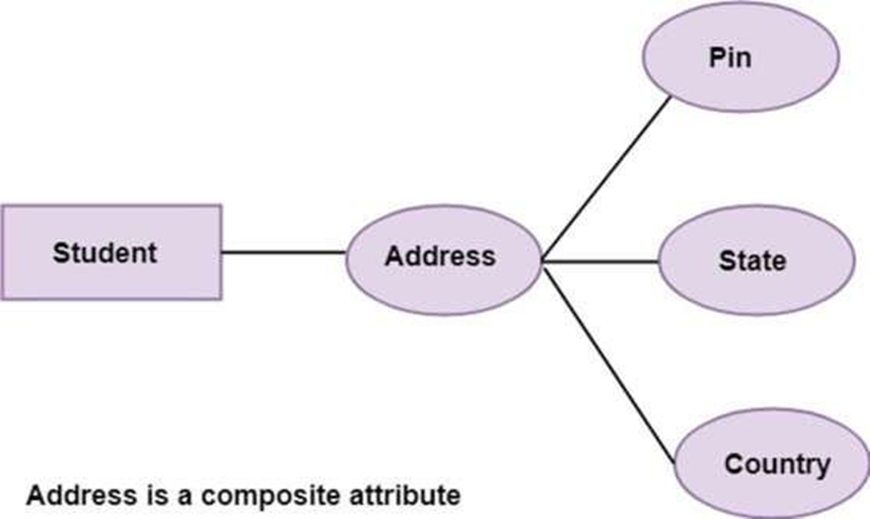


* + - **Key Attribute:** Uniquely identifies an entity (e.g., roll number).



**There are mainly three types of keys:**

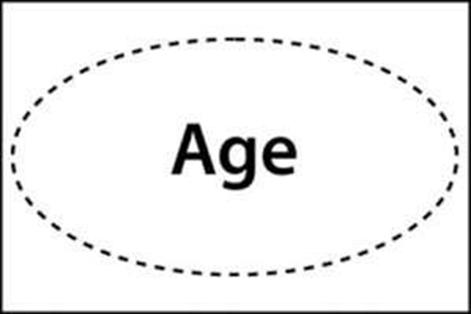
1. **Super key:** A set of attributes that collectively identifies an entity in the entity set.
2. **Candidate key:** A minimal super key is known as a candidate key. An entity set may have more than one candidate key.
3. **Primary key:** A primary key is one of the candidate keys chosen by the database designer to uniquely identify the entity set.
   * + **Composite Attribute:** A combination of other attributes (e.g., address). For example, In student entity, the student address is a composite attribute as an address is composed of other characteristics such as pin code, state, country.

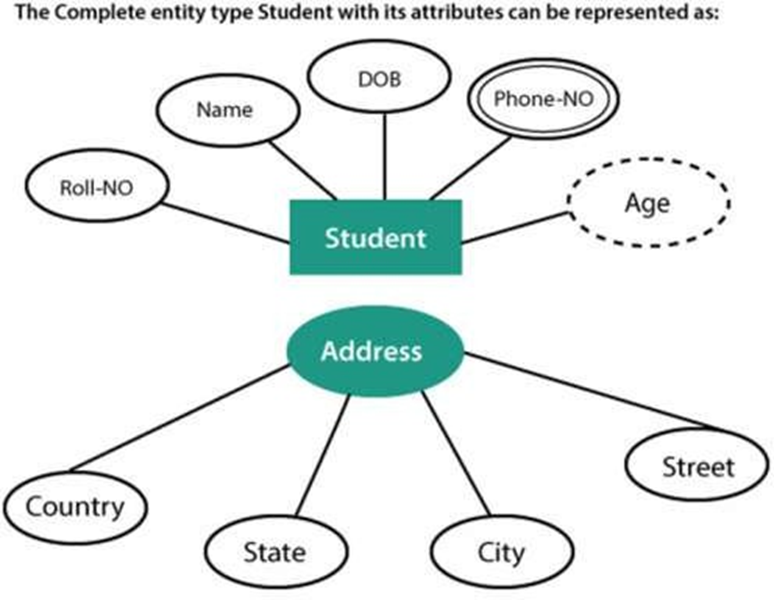


* + - **Single-valued attribute:** Single-valued attribute contain a single value. For example, Social\_Security\_Number.
    - **Multi-valued Attribute:** Can have more than one value (e.g., phone numbers). For example, a person can have more than one phone number, email-address.



* + - **Derived Attribute:** Value is derived from another attribute (e.g., age from date of birth).



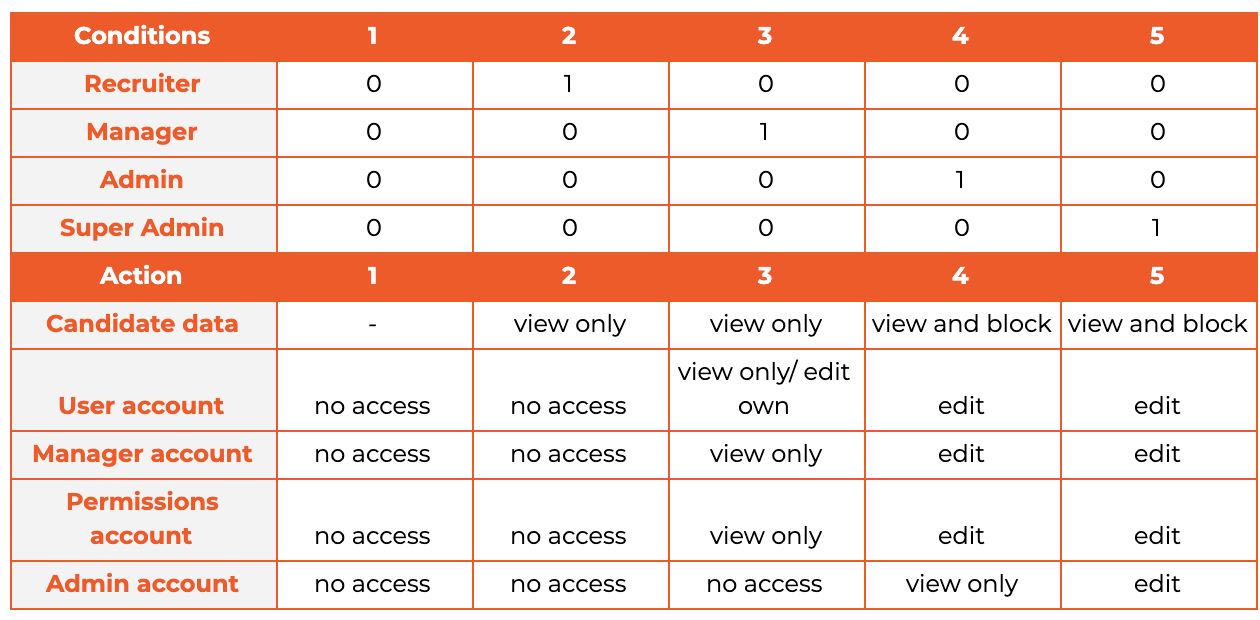


* + **Relationships:** The association among entities. Represented by a diamond-shaped box. For example, an employee works\_at a department, a student enrolls in a course. Here, Works\_at and Enrolls are called relationships.



**6.5. Decision Table**

* **Definition:** A tabular representation that captures complex decision logic by mapping conditions to corresponding actions.

****

[Decision table: How to use it in software testing?](https://www.merixstudio.com/blog/decision-table-software-testing)

* **Structure:**
  + **Conditions:** List all possible scenarios.
  + **Actions:** List the possible outcomes.
  + **Rules:** Columns defining specific condition combinations and their resulting actions.
* **Benefit:** Simplifies understanding and ensures all possible decision outcomes are considered.

**6.6. Use Case Diagram**

**Definition and Purpose**

A **Use Case Diagram** is used to represent the dynamic behavior of a system. It encapsulates the system's functionality by incorporating **use cases**, **actors**, and their **relationships**. This diagram models the tasks, services, and functions required by a system or subsystem, depicting its high-level functionality and illustrating how users interact with the system.

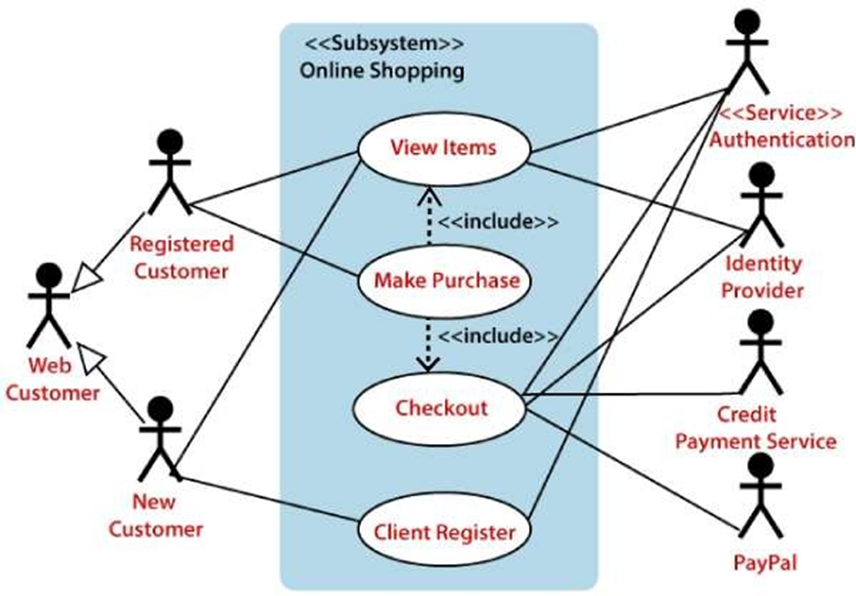
The primary purpose of a use case diagram is to portray the dynamic aspect of a system by:

1. **Gathering the system's needs.**
2. **Depicting the external view of the system.**
3. **Identifying both internal and external factors that influence the system.**
4. **Representing the interactions between the actors.**

**How to Draw a Use Case Diagram**

Creating a use case diagram involves a systematic process:

1. **Analyze the System:** Before drawing, conduct a thorough analysis of the entire system to identify all its functionalities.
2. **Identify Use Cases:** Transform each identified system functionality into a distinct use case.
3. **Identify Actors:** List all entities (persons or other systems) that interact with the system. An actor is anything that invokes system functionality.
4. **Define Relationships:** Inspect and define the relationships between the actors and the use cases. Determine how often and in what way an actor interacts with each use case.



**Rules for Drawing**

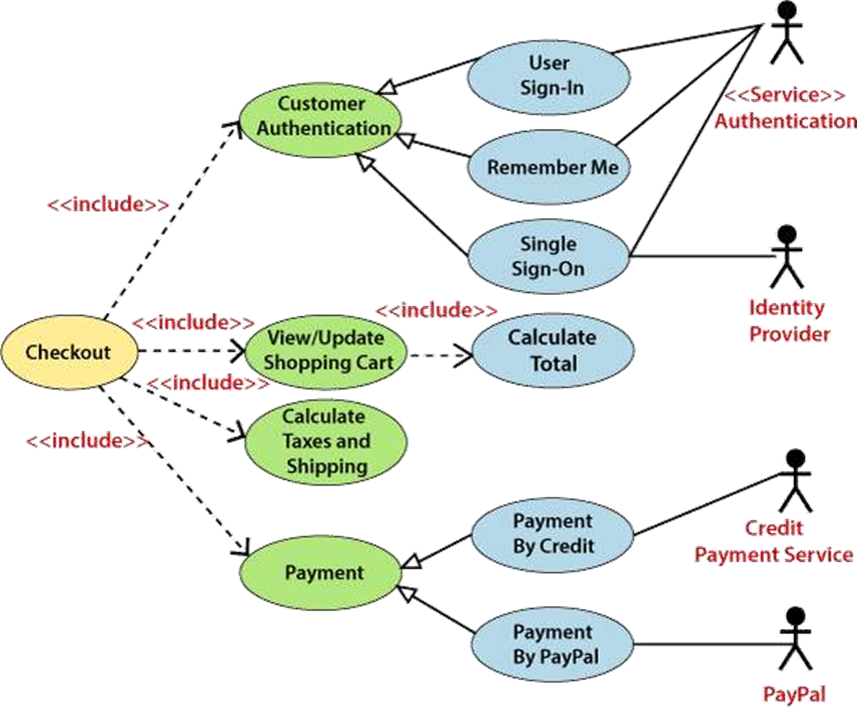
1. Assign a **pertinent and meaningful name** to each actor and use case.
2. Define the communication between an actor and a use case in a clear and understandable way.
3. Use standard UML notations correctly and consistently.
4. Focus on representing the **most significant interactions**; avoid clutter by omitting minor or infrequent interactions.



**Example: Online Shopping Website**

The provided diagram illustrates a use case diagram for an online shopping website:

* **Actor:** Web Customer
* **Top-Level Use Cases:** View Items, Make Purchase, Checkout, Client Register.
* **Extended Use Cases:** The View Items use case is extended by more specific functions like Search Items, Browse Items, View Recommended Items, Add to Shopping Cart, and Add to Wish List.
* **Included Use Cases:**
  + Checkout is an **included** use case within Make Purchase (i.e., you cannot check out without making a purchase).
  + Customer Authentication is **included** in View Recommended Item and Add to Wish List, as these actions require a logged-in user.
  + The Checkout process **includes** a Payment use case, which can be fulfilled by different payment methods.



**Important Tips**

1. Strive for a **simple and complete** diagram that is easy to understand.
2. Represent the **most significant interactions**.
3. Ensure the diagram represents **at least one coherent module** of the system.
4. If the system is large, create **multiple, more generalized use case diagrams** for different subsystems rather than one overly complex diagram.

**6.7. Class Diagram**

**Definition and Purpose**

A **Class Diagram** depicts a **static view** of an application. It represents the types of objects (classes) residing in the system and the relationships between them. A class describes a group of objects with common relationships, attributes, operations, and semantics. It is a structural diagram used to visualize, describe, document, and construct executable software code.

**Purposes of Class Diagrams.**

The main purpose of class diagrams is to build a static view of an application. It is the only diagram that is widely used for construction, and it can be mapped with object-oriented languages. It is one of the most popular UML diagrams. Following are the purpose of class diagrams given below:

1. To analyze and design the static view of an application.
2. To describe the major responsibilities of a system.
3. To form a base for component and deployment diagrams.
4. To support forward and reverse engineering.

**Benefits**

1. Can represent the object model for **complex systems**.
2. **Reduces maintenance time** by providing a clear overview of the application's structure.
3. Provides a **general schematic** for better understanding.
4. Offers a **detailed chart** that highlights the code to be programmed.
5. Helpful for both **stakeholders and developers**.

**Components of a Class Diagram(The class diagram is made up of three sections)**

A class is represented by a rectangle divided into three compartments:

**1. Upper Section (Name Compartment)**

* Contains the **name of the class**.
* **Rules:**
  + Capitalize the initial letter of the class name.
  + Center the class name within the compartment.
  + Write the class name in **bold**.
  + Write the name of an **abstract class** in *italics*.

**2. Middle Section (Attributes Compartment)**

* Lists the **attributes** (properties) of the class.
* Attributes are written with their **visibility**:
  + + Public
  + - Private
  + # Protected
  + ~ Package
* A meaningful name should be assigned to each attribute to explain its purpose.

**3. Lower Section (Operations/Methods Compartment)**

* Lists the **methods** or **operations** of the class.
* Each method is written on a single line.
* This section demonstrates how a class interacts with data.

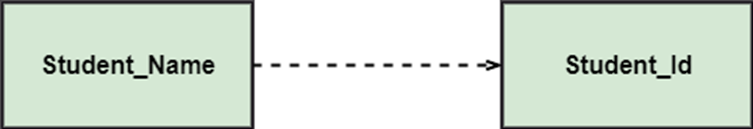
**Relationships in Class Diagrams**

Relationships define how classes are connected to one another.

**In UML, relationships are of three types**:

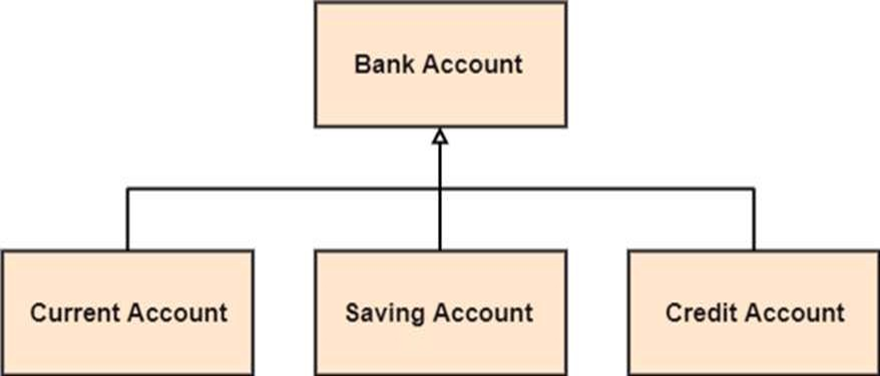
**1. Dependency**

* A **semantic relationship** where a change in one class may cause changes in another.
* Represents a **weaker**, using relationship.
* *Example:* Student\_Name is dependent on Student\_Id.



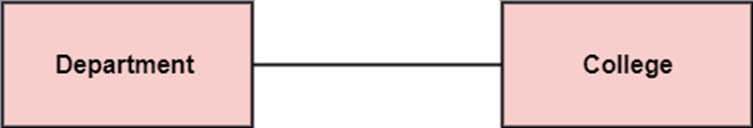
**2. Generalization (Inheritance)**

* A relationship between a **parent class (superclass)** and a **child class (subclass)**.
* The child class **inherits** from the parent class.
* *Example:* Current Account, Saving Account, and Credit Account are all generalized forms of a Bank Account superclass.

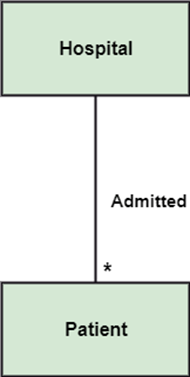


**3. Association**

* Describes a **static or physical connection** between two or more objects.



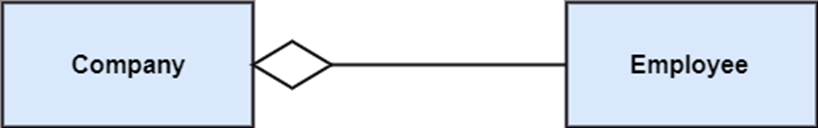
* **Multiplicity** defines the number of instances involved in the relationship (e.g., 1, \*, 0..1, 1..\*).
* *Example:* A Department is associated with a College. Multiple Patient instances are admitted to one Hospital.



**Specialized Forms of Association:**

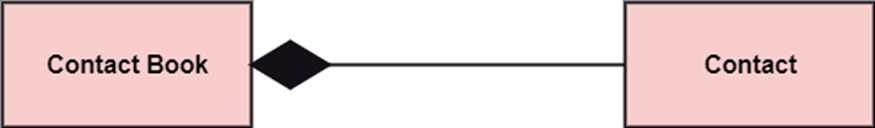
**A. Aggregation**

* A "**has-a**" relationship that represents a **part-whole** or **part-of** relationship.
* The child class can exist **independently** of the parent class.
* *Example:* A Company encompasses a number of Employees. If an employee resigns, the company still exists.



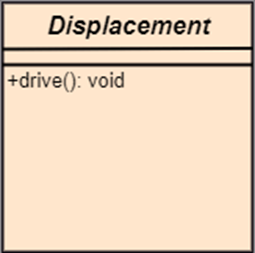
**B. Composition**

* A **stronger form of aggregation**.
* The child class **cannot exist independently** of the parent class; if the parent is deleted, the child is also destroyed.
* Represents a **whole-part** relationship.
* *Example:* A ContactBook consists of multiple Contacts. If the contact book is deleted, all the contacts within it are also lost.



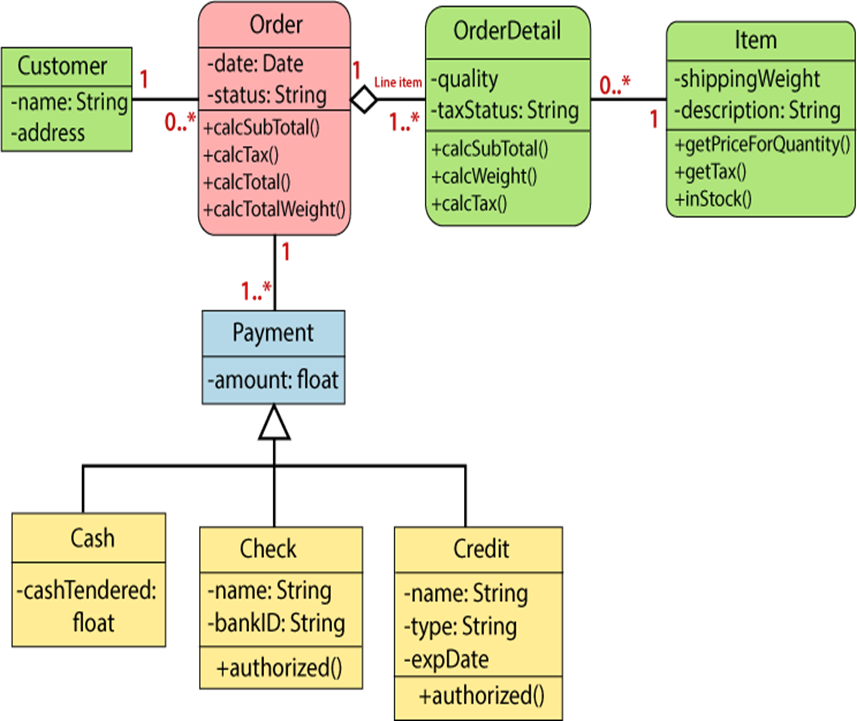
**Abstract Classes**

* An **abstract class** cannot be directly instantiated (you cannot create an object from it).
* It is used to define common functionality for a group of related subclasses.
* Its name is written in *italics*.
* *Example:* An abstract class Displacement with a method drive(). This method can be implemented by subclasses like Car, Bike, and Scooter.



**How to Draw a Class Diagram**

1. Give the diagram a **meaningful name**.
2. Identify the **objects, their attributes, methods, and relationships** in advance.
3. Specify only a **minimum number of essential properties** to avoid complexity.
4. Use **notes** to clarify aspects of the diagram if necessary.
5. **Iterate and refine** the diagram multiple times to ensure correctness before finalizing.



**Usage of Class Diagrams**

Class diagrams are primarily used for:

1. Describing the **static view** of a system.
2. Showing the **collaboration** among objects.
3. Describing the **functionalities** performed by the system.
4. Constructing software applications using **object-oriented languages**.

**6.9. Decision Tree**

**Definition and Purpose**

A **Decision Tree** is a decision-making tool represented as a tree-like structure. It visually maps out various **decision points (nodes)**, possible **actions (branches)**, and **outcomes (leaves)** based on specific conditions or rules. Each path from the root of the tree to a leaf represents a unique decision path.

* **Primary Use:** It is used to model complex decision processes in software systems, helping to visualize choices, conditions, and their potential consequences in a clear, logical manner.

**Decision Trees in UML Context**

It is important to note that a "UML Decision Tree" is **not a standard, standalone UML diagram type**. Instead, decision-making logic can be effectively represented using several official UML diagrams:

**1. Activity Diagram**

* **Decision Nodes:** These are diamond-shaped symbols used to represent points in the process where a decision is made, splitting the flow into multiple branches.
* **Merge Nodes:** These nodes are used to combine multiple branches back into a single path after the decision-based actions are completed.
* **Example:** Modeling an approval process. An activity diagram would show an initial action (e.g., "Submit Request"), a decision node with branches labeled [approved] and [rejected], leading to subsequent actions like "Process Request" or "Notify Rejection."

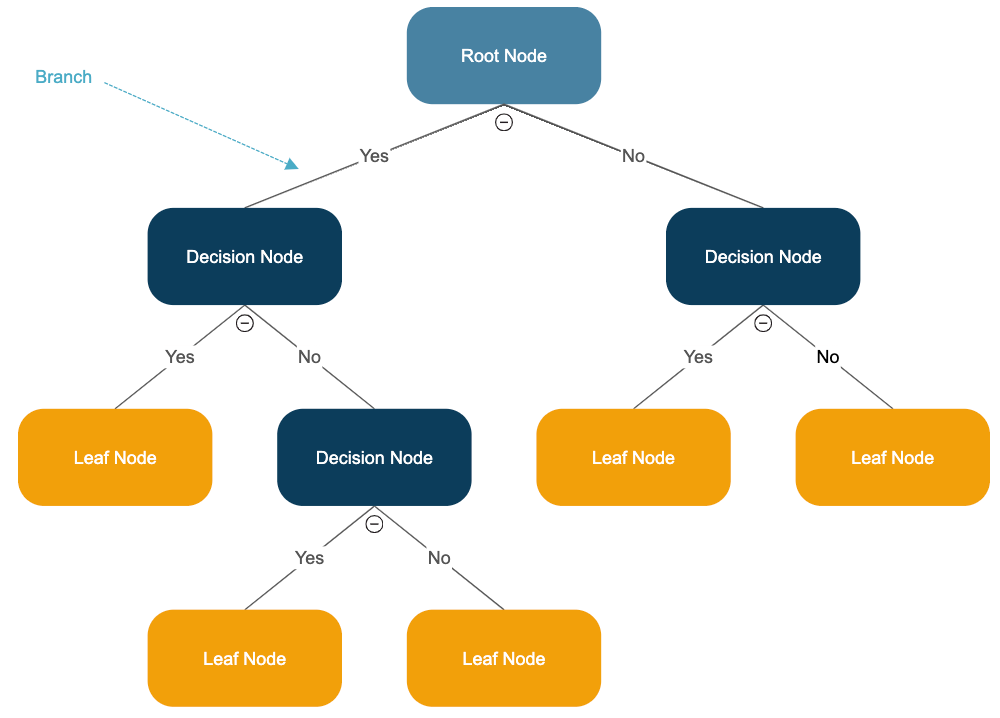
**2. State Machine Diagram**

* **States and Transitions:** This diagram represents the different states of an object and the transitions between those states, which are often triggered by decisions or events.
* **Example:** A document's lifecycle. A state machine diagram can show states like Draft, Under Review, and Approved. The transitions between these states (e.g., from Draft to Under Review when "submitted") are based on decisions or events.

**3. Flowchart (Non-UML but Common)**

* **Decision Points:** Flowcharts, while not part of the UML standard, are widely used and very intuitive for representing decision trees. They also use diamond shapes to represent decision points.
* **Example:** A simple login process. A flowchart can show a decision point after "Enter Credentials" with branches for "Valid?" leading to "Grant Access" or "Deny Access."

**Decision Tree Figure**

**

*https://www.smartdraw.com/decision-tree/*

**7. Practical Steps for Designing a System Application**

Designing a system application requires a structured approach to ensure the final product is efficient and meets all requirements. Here are the key starting steps:

**Phase 1: Planning and Analysis**

1. **Requirement Gathering and Analysis:** Actively collect and analyze the needs of all stakeholders to define what the system must do.
2. **Feasibility Study:** Assess the technical, economic, and operational feasibility of the project.
3. **System Design Planning:** Create a high-level plan outlining the design approach, methodologies, and tools to be used.

**Phase 2: Modeling and Design**

1. **System Modelling:** Create abstract models of the system to understand its structure and behavior.
2. **Tool Setup:**
   * Download and install the necessary software design tools (e.g., Visual Paradigm, [Draw.io](https://draw.io/)).
   * Check that all software, whether online or offline, is correctly installed and configured.
3. **Create UML Diagrams:** Use Unified Modeling Language (UML) to create diagrams that model the system’s structure and behaviour. Essential diagrams include:
   * **Class Diagrams** (for static structure)
   * **Sequence Diagrams** (for interaction over time)
   * **State Diagrams** (for object lifecycle)
   * **Use Case Diagrams** (for user interactions)
4. **Design Database Schema:** Based on the ERD (Entity-Relationship Diagram), create a detailed database schema that defines tables, relationships, primary and foreign keys, and data types.

**Phase 3: Implementation Planning and Review**

1. **Develop a Project Plan:** Outline the timeline, resources, milestones, and tasks for the development phase.
2. **Review and Finalize the Design:** Conduct a thorough review of all design artifacts (diagrams, models, schemas) with stakeholders to ensure accuracy and completeness before moving to the implementation (coding) phase.
3. **Export Designs:** Once finalized, export the diagrams and documentation for use by the development team.

**Indicative content 2.2: Identification of Hardware and Software Technology**

**2.2.1. Identification of Hardware and software technology**

To develop a system structure, it's important to understand how different types of hardware and software technologies contribute to the overall architecture.

**Computer Hardware**

1. **Central Processing Unit (CPU):** The brain of the computer, executing

instructions from programs.

**2. Memory (RAM):** Temporary storage for data and instructions that the CPU

needs while performing tasks.

**3. Storage Devices:** Hard drives or SSDs for long-term data storage.

**4. Networking Equipment**: Routers, switches, and network interface cards (NICs)

to connect and communicate over networks.

**5. Input/Output Devices:** Keyboards, mice, monitors, and printers used for

interaction with the system.

**System Software**

**1. Operating System (OS):** Manages hardware resources and provides services for application software. Examples include Windows, mac OS, and Linux.

**2. Device Drivers:** Specialized software that allows the OS to communicate with

hardware components (e.g., graphics drivers, printer drivers).

**3. Utilities:** Tools for system maintenance and optimization (e.g., antivirus

software, disk clean up tools).

**Application Software**

**1. Microsoft Office:** A suite of productivity applications including Word, Excel,

PowerPoint, and Outlook. Useful for document creation, data analysis,

presentations, and email.

**2. Visual Paradigm:** A UML tool for modeling and designing system architectures,

including diagrams such as use case diagrams, class diagrams, and sequence

diagrams.

**3. E-Draw Max:** A diagramming tool that allows you to create flowcharts,

network diagrams, and other visual representations of systems and processes.

**Browser**

**•Web Browser:** Software such as Chrome, Firefox, or Edge that allows users to access and interact with web-based applications and resources. Essential for cloud-based tools and online collaboration.

**Indicative content 2.3: Application of SSADM (Structured System Analysis and Design Methods)**

**2.3.1. Description of SSADM**

SSADM is a systematic approach to software development, particularly useful in the analysis and design of information systems.

**1. Objective of SSADM**

**Structured Approach:** SSADM aims to provide a clear and structured methodology for developing systems to ensure all requirements are captured accurately.

**Requirements Analysis:** It emphasizes detailed analysis of user requirements to create systems that meet business needs.

**Documentation:** SSADM promotes detailed documentation throughout the development process, facilitating better communication among stakeholders. **Quality Assurance:** By following a structured methodology, SSADM helps ensure the quality and reliability of the developed systems.

**2. SSADM Techniques**

**Logical Data Modelling**

To define and represent the data requirements of the system independently of any physical implementation.

**Components:**

**Entities:** Represent objects or concepts within the system. **Attributes:** Describe the properties of entities.

**Relationships:** Define how entities interact with one another.

**Outcome:** A logical data model that serves as a blueprint for the data structure of the system.

**Data Flow Modelling**

**Purpose:** To visualize how data moves through the system, identifying inputs, processes, and outputs.

**Components:**

**Processes:** Activities that transform inputs into outputs. **Data Stores:** Repositories where data is held.

**External Entities:** Sources or destinations of data outside the system.

**Outcome:** Data flow diagrams (DFDs) that depict the flow of information and help identify inefficiencies or redundancies in processes.

**Entity Behaviour Modelling**

**Purpose:** To capture the dynamic behaviour of entities within the system. Components:

**State Transitions:** Define how an entity changes states in response to events. **Events:** Triggers that cause changes in the state of an entity.

**Outcome**: State transition diagrams that illustrate how entities behave over time, helping to clarify requirements related to system interactions.

**Conclusion**

SSADM provides a comprehensive framework for analysing and designing information systems. By utilizing techniques like logical data modelling, data flow modelling, and entity behaviour modelling, developers can ensure that the resulting system is well-structured, meets user needs, and is of high quality.

**3. Advantages**

1. Separation of logical and physical aspects of the system

2. Well-defined techniques and documentation

3. User involvement

**4. Disadvantages**

1. The size of SSADM (in some circumstances)

2. Cost and time in training people

3. The learning curve can be considerable if the full method is used.

**2.3.2: Implementing stages for drawing SSADM**

Implementing SSADM (Structured Systems Analysis and Design Method) involves several stages that guide the development process from initial analysis through to design.

Here’s a structured approach to the stages of SSADM:

**Stages for Drawing SSADM**

**1. Feasibility Study**

Assess whether the proposed system is viable and aligns with business objectives.

**Activities:**

* Conduct initial discussions with stakeholders.
* Identify constraints, costs, and potential benefits.
* Produce a feasibility report outlining findings.

**2. Requirements Analysis**

**Objective:** Gather and document detailed user requirements. Activities:

**Interviews and Surveys**: Engage with users to understand their needs. **Workshops:** Facilitate group discussions to gather comprehensive requirements. Documentation: Create a requirements specification document.

Outcome: Clear and detailed understanding of what the system must achieve.

**3. Logical System Specification**

**Objective:** Develop a model of the system that reflects the identified

requirements. Activities:

**Logical Data Modelling:** Create entity-relationship diagrams.

**Data Flow Modelling:** Develop data flow diagrams to represent how data moves through the system.

**Entity behaviour Modelling:** Construct state transition diagrams to illustrate the behaviour of entities.

Outcome: A set of models that define the system's functionality and data structure.

**4. System Design**

**Objective:** Translate the logical specifications into a physical design for implementation.

Activities:

**Architecture Design:** Define the overall system architecture.

**Database Design:** Develop a physical data model based on the logical data model. **Interface Design:** Create design mockups for user interfaces.

**Outcome:** A detailed design specification that guides development.

**5. Implementation**

**Objective**: Build the system according to the design specifications.

**Activities:**

* **Coding:** Develop the software based on the design.
* **Testing:** Conduct unit tests and system tests to ensure functionality.
* **Documentation:** Produce user manuals and technical documentation.
* **Outcome:** A fully functional system ready for deployment.

**6. Testing and Validation**

**Objective:** Ensure the system meets all specifications and is ready for live use. Activities:

**User Acceptance Testing (UAT):** Engage end-users to validate the system against their requirements.

**Bug Fixing:** Address any issues identified during testing. **Outcome:** A validated system that meets user expectations.

**7. Deployment**

**Objective:** Roll out the system to users.

**Activities:**

* **Installation:** Set up the system in the production environment.
* **Training:** Provide training sessions for users.
* **Support:** Establish a support process for ongoing user assistance.
* **Outcome:** The system is live and operational.

**8. Maintenance**

**Objective:** Ensure the system continues to function effectively over time.

**Activities:**

* **Monitoring:** Regularly check system performance.
* **Updates and Upgrades:** Implement changes based on user feedback and

technological advancements.

* **Documentation:** Keep system documentation up to date. **Outcome:** A sustainable system that evolves with user needs.

**Indicative content 2.4: Application of Object-Oriented Analysis and Design**

**2.4.1.: Application of Object-Oriented Analysis and Design (OOAD)**

**Object-Oriented Analysis and Design (OOAD)**

**Object-Oriented Analysis and Design (OOAD)** is a systematic approach to software development that organizes systems around **"objects"** rather than actions and logic. Think of it as building with LEGO blocks - each block (object) contains both data and functionality, and you combine them to create complex structures.

**Core Concept:**

An **object** is a software bundle of related data (attributes) and behaviours (methods). For example:

* A **"Car"** object might have:
  + **Data:** colour, speed, fuel level
  + **Behaviours:** start(), accelerate(), brake()

**Key Principles of OOAD**

**1. Encapsulation**

* **What it is:** Bundling data and methods that work on that data within one unit (object)
* **Simple analogy:** A capsule medicine - the outer shell protects the ingredients inside
* **Benefit:** Internal details are hidden, only essential features are exposed

**2. Inheritance**

* **What it is:** Creating new classes based on existing ones
* **Simple analogy:** A family tree - children inherit characteristics from parents
* **Example:**
  + Base class: Vehicle (with wheels, engine)
  + Child classes: Car, Truck, Motorcycle (inherit from Vehicle)

**3. Polymorphism**

* **What it is:** Same action, different implementations
* **Simple analogy:** "Draw" command - works differently for Circle vs. Square
* **Example:** calculateArea() works for both Circle and Rectangle objects

**Advantages of OOAD**

|  |  |  |
| --- | --- | --- |
| Advantage | What It Means | Real Benefit |
| **Modularity** | Break system into independent objects | Fix one part without breaking others |
| **Reusability** | Use existing objects in new projects | Save time and reduce errors |
| **Maintainability** | Clear structure makes updates easier | Lower long-term costs |
| **Real-World Mapping** | Objects represent real entities | Easier to understand and discuss |

**Disadvantages of OOAD**

|  |  |  |
| --- | --- | --- |
| Disadvantage | What It Means | Impact |
| **Complexity** | More planning and structure needed | Steeper learning curve |
| **Overhead** | More code for simple tasks | Can be slower for simple applications |
| **Design Time** | Requires thorough upfront design | Longer initial development phase |

**The OOAD Process: Step by Step**

**Phase 1: Analysis - "What needs to be built?"**

* **Goal:** Understand requirements
* **Activities:**
  + Identify key objects from real world
  + Define how objects interact
  + Create use cases (user scenarios)
* **Output:** Analysis model showing objects and relationships

**Phase 2: Design - "How will we build it?"**

* **Goal:** Create detailed blueprint
* **Activities:**
  + Design classes with attributes and methods
  + Define relationships between classes
  + Plan overall architecture
* **Output:** Detailed design specifications

**Phase 3: Implementation - "Building the system"**

* **Goal:** Write working software
* **Activities:**
  + Code classes and methods
  + Test individual components
  + Integrate pieces together
* **Output:** Functional software system

**Real-World Example: Library System**

**Analysis Phase:**

* Identify objects: Book, Member, Librarian, Loan
* Define relationships: Members borrow Books, Librarians manage Loans

**Design Phase:**

* Create class diagrams:

*Book Class:*

*- Attributes: title, author, ISBN, available*

*- Methods: borrow(), return()*

*Member Class:*

*- Attributes: name, memberID, borrowedBooks*

*- Methods: login(), searchBook(), borrowBook()*

**Implementation Phase:**

* Write actual code based on the design
* Test that members can successfully borrow and return books

**When to Use OOAD**

* **Good for:** Complex systems, large teams, long-term projects
* **Less suitable for:** Simple scripts, performance-critical applications

**Key Takeaway**

OOAD helps you build software like an architect builds a house - with careful planning, reusable components, and a clear structure that's easy to understand, modify, and extend over time. It's particularly valuable when you need to create software that can evolve and grow without becoming unmanageable.

**Learning Outcome 3: Build System Design**

**Indicative content 3.1: Development of Data Flow Diagram**

**Data Flow Diagrams (DFD)**

A **Data Flow Diagram (DFD)** is a visual map that shows how information moves through a system. Think of it as a **roadmap for data** - it shows where data comes from, where it goes, where it's processed, and where it's stored.

**Key Purpose:** To give a clear picture of how data moves without getting bogged down in technical details.

**The 4 Building Blocks of DFD**

|  |  |  |  |
| --- | --- | --- | --- |
| Symbol | Name | What It Represents | Real-World Example |
| ○ or ▢ | **Process** | Transforms input data into output data | "Process Order", "Calculate Total", "Verify Login" |
| → | **Data Flow** | Movement of data between components | "Customer Info", "Order Details", "Payment Data" |
| 〚 〛 | **Data Store** | Where data is stored for later use | "Customer Database", "Product Inventory", "Order History" |
| ▭ | **External Entity** | Source or destination of data OUTSIDE the system | "Customer", "Bank", "Shipping Company" |

**Rules for Creating Good DFDs**

1. **Use Standard Symbols** - Stick to the 4 basic symbols for clarity
2. **Every Process Must Transform** - Each process should have at least **one input and one output**
3. **Label Everything Clearly** - Every element needs a meaningful name
4. **Show Data, Not Control** - DFDs show **what** data moves, not **when** or **how** decisions are made
5. **Think in Layers** - Start simple and add detail gradually

**The 3 Levels of DFD:**

**Level 0: Context Diagram (The Bird's-Eye View)**

* Shows the ENTIRE system as **one single process**
* Identifies all external entities and their data exchanges
* Answers: "Who interacts with our system and what do they send/receive?"

**Example - Online Shopping:**

[Customer] → "Order Request" → [Online Store System] → "Confirmation" → [Customer]

↓

"Payment Info" → [Bank]

**Level 1: Major Processes (The Main Departments)**

* Breaks the single process from Level 0 into **3-7 key sub-processes**
* Shows how data moves between major functions
* Answers: "What are the main things our system does?"

**Example - Online Shopping Level 1:**

[Customer] → "Browse Products" → "Shopping Cart" → "Checkout Process" → [Bank]

↓ ↓

"Product Catalog" "Order Database"

**Level 2: Detailed Processes (The Inner Workings)**

* Zooms into ONE Level 1 process to show its internal steps
* Provides detailed view of complex operations
* Answers: "Exactly how does this specific process work?"

**Example - "Checkout Process" Level 2:**

"Cart Items" → "Calculate Total" → "Apply Discount" → "Process Payment" → "Confirm Order"

↓ ↓ ↓

"Tax Rules" "Promo Database" "Payment Gateway"

**Tools for Creating DFDs**

|  |  |  |
| --- | --- | --- |
| Tool | Best For | Key Feature |
| **Microsoft Visio** | Corporate environments | Integration with Office suite |
| **Visual Paradigm** | Professional developers | Full UML + DFD support |
| **Lucidchart** | Teams & collaboration | Easy online sharing |
| [**Draw.io**](https://draw.io/) | Free option | Completely free, good basic features |

**How to Create a DFD - Simple Steps**

**Step 1-3: Setup**

1. Choose your tool (Lucidchart, Visio, etc.)
2. Select a DFD template
3. Name your diagram

**Step 4-6: Add Core Elements**  
4. Place External Entities (customers, banks, etc.)  
5. Add major Processes (what your system does)  
6. Include Data Stores (where information is kept)

**Step 7-8: Connect Everything**  
7. Draw Data Flows between elements  
8. Label every data flow clearly

**Step 9-10: Polish**  
9. Arrange for readability, add colours  
10. Review and share with your team

**Real Examples: Online Shopping System**

**Level 0 Example:**

* **One process:** "Online Store"
* **External entities:** Customer, Bank, Shipping Company
* **Data flows:**
  + Customer → Orders, Payments
  + Store → Order Confirmations, Tracking Numbers
  + Store ↔ Bank: Payment Authorization

**Level 1 Example Processes:**

1. **Product Browsing** - Search, view products
2. **Shopping Cart** - Add/remove items, calculate totals
3. **Checkout** - Payment processing, order confirmation
4. **Order Management** - Track shipments, handle returns

**Level 2 Example (Checkout Process):**

Shopping Cart → Validate Cart → Calculate Total → Process Payment → Create Order → Send Confirmation

↓ ↓ ↓ ↓

Inventory DB Pricing Rules Payment Gateway Order Database

**Common Mistakes to Avoid**

1. **❌ Creating "Spaghetti Diagrams"** - Too many crossings lines
2. **❌ Mixing Data and Control Flow** - Don't show "if yes, then..." decisions
3. **❌ Skipping Levels** - Don't jump from Level 0 to super-detailed
4. **❌ Unlabelled Flows** - Every arrow needs a clear name
5. **❌ Too Much Detail** - DFDs are about data movement, not every tiny step

**When to Use DFDs**

**✅ Great for:**

* Understanding existing systems
* Communicating with non-technical stakeholders
* Planning new system architecture
* Identifying data requirements

**❌ Not ideal for:**

* Showing timing or sequence of operations
* Detailed programming logic
* User interface design

**Key Benefits**

1. **Clear Communication** - Everyone can understand the data flow
2. **Early Problem Detection** - Spot missing data or processes early
3. **Documentation** - Serves as clear system documentation
4. **Foundation for Design** - Provides blueprint for developers

**Remember:** A good DFD is like a good map - it should help people understand the territory without overwhelming them with unnecessary detail!

**Indicative content 3.2: Application of Physical Data Model**

**3.2.1: Description of database Objects**

In a database management system (DBMS), various objects are used to store, manage, and manipulate data. Here’s a breakdown of the primary database objects:

1. Tables

**Definition**: The fundamental objects that store data in rows and columns.

**Components**:

**Columns**: Define the attributes of the data (e.g., name, age).

**Rows**: Each row represents a single record or entry.

2. Views

**Definition**: Virtual tables that represent the result of a query. They do not store data themselves but provide a way to present data from one or more tables.

**Use**: Simplify complex queries and enhance security by restricting access to specific data.

3. Indexes

**Definition**: Objects that improve the speed of data retrieval operations on a table by providing quick access paths to the data.

**Types**:

**Unique Indexes**: Ensure that the indexed columns do not contain duplicate

values.

**Composite Indexes**: Involve multiple columns to increase retrieval efficiency.

4. Stored Procedures

**Definition**: Precompiled collections of one or more SQL statements that can be executed as a single unit.

**Use**: Encapsulate business logic, perform complex calculations, and improve performance by reducing network traffic.

5. Functions

**Definition**: Similar to stored procedures, but they return a single value and can be used in SQL expressions.

**Use**: Perform calculations, manipulate data, and return results for use in queries.

6. Triggers

**Definition**: Special types of stored procedures that automatically execute in response to certain events on a particular table (e.g., insert, update, delete).

**Use**: Enforce business rules, maintain audit trails, and ensure data integrity.

7. Schemas

**Definition**: Logical containers that hold database objects such as tables, views, and procedures.

**Use**: Organize and group related objects, enhancing security and manageability.

8. Sequences

**Definition**: Objects that generate unique numeric values, often used for primary key columns.

**Use**: Automatically generate sequential numbers for new records.

9. Synonyms

**Definition**: Aliases for database objects that simplify access to them.

**Use**: Allow users to refer to objects using different names, often used for simplifying references to complex object names.

10. Constraints

**Definition**: Rules applied to table columns to enforce data integrity.

**Types**:

* **Primary Key**: Uniquely identifies each record in a table.
* **Foreign Key**: Ensures referential integrity between tables.
* **Unique Constraint**: Ensures that all values in a column are distinct.
* **Check Constraint**: Enforces domain integrity by limiting the values that can be
* entered in a column.

**3.2.1: Designing Database in E drawMax**

To design a database in E-Draw Max, including tables and relationships, follow these steps. Here’s a detailed guide tailored for creating an EntityRelationship Diagram (ERD) for a library management system.

**Step-by-Step Guide to Design a Database in E-Draw Max**

**Step 1: Open E-Draw Max**

Launch E-Draw Max on your computer.

**Step 2: Create a New Document** Click on **"File"** > **"New"** to create a new drawing.

**Step 3: Select ER Diagram Template** ✓ Look for the **"Templates"** section and search for **"ER Diagram"**. ✓ Choose an appropriate template to start.

**Step 4: Add Tables (Entities)**

1. **Authors Table**

✓ Drag a **rectangle shape** to represent the table.

✓ Label it **"Authors"**.

✓ Inside, add attributes:

* AuthorID (PK)
* Name
* Birthdate

2. **Books Table**

Repeat the process to create a **"Books"** table:

* BookID (PK)
* Title
* AuthorID (FK)
* Genre
* PublishedYear

3. **Members Table**

Create a **"Members"** table:

* MemberID (PK)
* Name
* Email
* PhoneNumber

4. **Loans Table**

Finally, create a **"Loans"** table:

* LoanID (PK)
* BookID (FK)
* MemberID (FK)
* LoanDate
* ReturnDate

**Step 5: Define Relationships**

1. **Authors to Books**

Draw a line from the **"Authors"** table to the **"Books"** table.

Add a label indicating the relationship: **"1"** on the **Authors** side and

**"M"** on the **Books** side.

2. **Members to Loans**

* Draw a line from the **"Members"** table to the **"Loans"** table.
* Label it **"1"** on the **Members** side and **"M"** on the **Loans** side.

3. **Books to Loans**

* Draw a line from the **"Books"** table to the **"Loans"** table.
* Label it **"1"** on the **Books** side and **"M"** on the **Loans** side.

**Step 6: Arrange and Format**

* Adjust the layout for clarity, ensuring that the tables and
* relationships are well spaced and easy to read.
* Use formatting options to style the tables and lines as needed.

**Step 7: Save Your Diagram**

Click on **"File"** > **"Save As"** to save your work in your desired format

(e.g., E-Draw Max format, PDF, or image).

**Indicative content 3.3: Documentation of System Design**

**3.3.1: Documentation of system design**

**Definition**

The System Design Document describes the system requirements, operating environment, system and subsystem architecture, files and database design, input formats, output layouts, human-machine interfaces, detailed design, processing logic, and external interfaces.

**Types of documentation of system design 1. System Design document (SDD)**

The System Design Document (SDD) describes how the functional and nonfunctional requirements recorded in the Requirements Document, the preliminary user oriented functional design recorded in the High-Level Technical Design Concept/Alternatives document, and the preliminary data design documented in the Logical Data

**2. Functional Specification Document (FSD)**

The Functional Specification Document (FSD) is written by the project's Business Analyst and provides detailed information on how the system solution will function based on what the requested behaviour is.

**3. Technical Specification Document(TSD)**

This specification outlines the details of a product’s design and the technical requirements for its development. It is a roadmap for engineers and developers to follow during the design and implementation phase.

The document specifies the technical attributes and requirements of the product, including the tools, technologies, and programming languages that will be used. It also describes the intended user experience, including the product’s features and functionality.

**4. Database design Document**

The Database Design Document maps the logical data model to the target database management system with consideration to the system's performance requirements.

**5. Use case Document**

A use case document is a standardized document that describes a use case in detail. It includes the steps, preconditions, assumptions and expected outcomes or results. On the other hand, a use case is a general term that describes a specific scenario or interaction between a user or system and a product or service.

System documentation.

**3.3.2: Developing system documentation for a backend system**