

Project:

A **project** is a planned series of tasks or activities that are undertaken to achieve a specific goal or objective within a defined timeframe.

They often include the following key elements:

1. **Objective:** The purpose or goal of the project, which can be to create something new, solve a problem, improve a process, or implement a change.
2. **Scope:** The specific boundaries of the project, including what will and will not be included in its work.
3. **Resources:** The people, materials, technology, and budget required to complete the project.
4. **Timeline:** The schedule, including deadlines for tasks and milestones.
5. **Tasks:** The individual activities or steps that need to be completed to achieve the project's objective.
6. **Team:** The group of individuals responsible for carrying out the project tasks

Software project

A **software project** is a specialized project focused on the development, maintenance, or improvement of a software application or system. It involves a series of tasks, processes, and methodologies to design, build, test, deploy, and maintain software. Software projects can range from small applications developed by individuals to large-scale systems developed by teams or organizations.

Key Components of a Software Project:

1. **Objective:**
 - Develop or improve a software application to meet specific requirements or solve a problem.
2. **Scope:**
 - Defines what features and functionality the software will include, and sometimes what will be excluded.
3. **Requirements Gathering:**
 - Identifying the needs of users or stakeholders to ensure the software meets business goals and user expectations. Requirements can be functional (what the software does) and non-functional (performance, security, etc.).
4. **Planning:**
 - Detailed planning of tasks, timelines, and resources to ensure the project progresses smoothly. This can include task breakdowns, assigning team members, and scheduling milestones.
5. **Design:**
 - The process of defining the architecture, components, interfaces, and data of the software. This includes both high-level system design and detailed design for each feature.
6. **Development:**
 - The actual writing of code, implementation of features, and integration of different components of the software.
7. **Testing:**
 - Ensuring the software works as intended and is free from defects. Testing can include unit testing, integration testing, system testing, and user acceptance testing.
8. **Deployment:**
 - The process of making the software available for use. This could be deploying to production servers, distributing apps, or releasing updates.
9. **Maintenance:**

- Ongoing work to fix bugs, improve performance, or add new features after the software is deployed.

10. **Documentation:**

- Creating user manuals, technical documentation, or internal documentation to help with future development or troubleshooting.

Software project manager

A **software project manager** is responsible for planning, executing, and overseeing software development projects. Their role is to ensure that the project is completed on time, within budget, and meets the required specifications and quality standards. They work closely with software developers, designers, testers, and stakeholders to coordinate the project's different aspects, making sure all tasks align with the project's goals.

Key Responsibilities of a Software Project Manager:

1. **Project Planning:**

- Define the scope, objectives, and deliverables of the project.
- Create detailed project plans, including timelines, milestones, and resource allocation.
- Determine the project budget and ensure cost-efficiency.

2. **Team Management:**

- Assemble and manage the software development team, including developers, testers, designers, and other key members.
- Delegate tasks, assign roles, and ensure the team has the tools and resources they need.
- Foster communication and collaboration within the team.

3. **Risk Management:**

- Identify potential risks that could affect the project's success (e.g., technical challenges, budget overruns, and time constraints).
- Develop mitigation strategies to handle these risks effectively.
- Monitor and manage risks throughout the project lifecycle.

4. **Scheduling and Monitoring:**

- Track progress against the project plan, ensuring that tasks and milestones are completed on schedule.
- Monitor the development process and make adjustments to keep the project on track, such as reallocating resources or adjusting deadlines when necessary.
- Use project management software tools to monitor tasks, timelines, and team productivity.

5. **Stakeholder Communication:**

- Communicate project status, challenges, and progress to stakeholders, including clients, upper management, or external partners.
- Manage stakeholder expectations and ensure that their requirements are met.

6. **Quality Assurance:**

- Ensure that the software meets quality standards, testing, and user requirements before deployment.
- Coordinate with quality assurance (QA) teams to set up testing processes and feedback loops.

7. **Budget Management:**

- Track project costs and manage the budget to prevent overspending.
- Optimize resources to achieve the project's goals while staying within financial constraints.

8. **Documentation:**

- Maintain detailed records of the project's progress, including changes to the scope, timelines, and requirements.
- Ensure proper documentation for future reference or project audits.

The **Software Development Life Cycle (SDLC)** is a structured process followed during the development of software applications. It outlines the various stages involved in creating a software product, from initial planning to deployment and maintenance. SDLC ensures that the software meets customer requirements, is completed on time, and within budget.

Phases of the SDLC:

1. System Study/Requirement Gathering and Analysis:

- In this phase, the project's stakeholders, users, and business analysts gather and document detailed requirements. These requirements define what the system should do, the functionality it must provide, and the business problems it aims to solve.
- The outcome is a requirements specification document.

Common Requirement Gathering Techniques:

1. Interviews:

- **Description:** Interviews involve one-on-one or group discussions between business analysts and stakeholders (users, clients, subject matter experts) to understand their needs.
- **Types:** Structured (pre-defined set of questions) and unstructured (open-ended discussions).

2. Questionnaires/Surveys:

- **Description:** Predefined questions are distributed to a large audience to gather information about requirements from a broad range of stakeholders.

3. Workshops:

- **Description:** Interactive group sessions where stakeholders, users, and the project team collaborate to identify requirements, discuss issues, and propose solutions.

4. Focus Groups:

- **Description:** A moderated group discussion involving selected users or stakeholders to gather opinions and feedback on specific aspects of the system.

5. Observation (Job Shadowing):

- **Description:** Analysts observe users in their natural working environment to understand how they perform their tasks and identify their needs and challenges.

6. Document Analysis:

- **Description:** Review of existing documentation such as business process flows, system manuals, reports, or previous project records to understand current processes and systems.

7. Prototyping:

- **Description:** Creating an early version or mockup of the system to allow users to visualize and interact with the proposed solution. Users provide feedback, which is then used to refine the requirements.

8. Brainstorming:

- **Description:** A collaborative session where stakeholders and team members suggest ideas and solutions without criticism. All ideas are documented for further analysis.

2. Feasibility Study/system Analysis:

- The feasibility study analyzes whether the project is technically, financially, and operationally viable. This includes assessing available resources, budget, time, and technology.
- It leads to deciding whether to proceed with the project.

Types of Feasibility Study in SDLC:

1. Technical Feasibility:

- **Focus:** Determines whether the organization has the technical expertise, infrastructure, and technology required to complete the project.
 - **Questions addressed:**
 - Is the required technology available, and does the organization have the technical skills to implement the solution?
 - Are there software, hardware, or technical tools that support the proposed system?
 - Can the system be integrated with existing technologies?
2. **Economic Feasibility (Cost-Benefit Analysis):**
- **Focus:** Analyzes the financial aspects of the project, including the costs of development, implementation, and ongoing maintenance, and compares them with the expected financial benefits.
 - **Questions addressed:**
 - What are the initial and ongoing costs (e.g., software, hardware, labor)?
 - What are the projected benefits, such as increased revenue or cost savings?
 - Is the return on investment (ROI) positive, and when will the project break even?
3. **Operational Feasibility:**
- **Focus:** Evaluates whether the project meets the operational requirements and whether it will be accepted by users and fit within the current workflows.
 - **Questions addressed:**
 - Will the system meet the needs of the users?
 - Are the organization's current operations and processes compatible with the new system?
 - How will the users and stakeholders react to the new system? Will it cause resistance?
4. **Schedule Feasibility:**
- **Focus:** Assesses whether the project can be completed within the required timeframe.
 - **Questions addressed:**
 - Is the proposed timeline realistic given the scope of the project?
 - Are there enough resources (time, people, and tools) to meet the deadlines?
 - What are the consequences of missing the deadline?
5. **Legal and Regulatory Feasibility:**
- **Focus:** Evaluates whether the project complies with legal, regulatory, and contractual requirements.
 - **Questions addressed:**
 - Does the project comply with data protection laws?
 - Are there any intellectual property, licensing, or copyright issues?
 - Will the project meet industry-specific regulatory standards (e.g., healthcare, finance)?
6. **Environmental Feasibility:**
- **Focus:** Analyzes the potential environmental impact of the project, particularly for industries where environmental regulations are important.
 - **Questions addressed:**
 - Does the project comply with environmental regulations and standards?
 - Will the system's development or use cause any significant environmental impact?
3. **System Design:**
- **System design** is the phase where the structure and architecture of the system are defined, laying out how the system will meet the specified requirements.
 - The system design phase translates requirements into a blueprint for the system's architecture. This includes both **high-level design** (defining system components and their interactions) and **low-level design** (detailed design of each component).

There are two primary types of system design in SDLC:

1. High-Level Design (HLD):

- **Focus:** Provides an overall system architecture and design framework. It defines the structure of the system at a macro level, focusing on how different components and subsystems interact with each other.
- **Key Components:**
 - **System Architecture:** Outlines the overall architecture of the system, including hardware, software, networks, databases, and interfaces.
 - **Modules/Subsystems:** Identifies the major components (subsystems or modules) of the system and how they interact.
 - **Data Flow:** Describes the flow of data across the system and between components, including any data exchanges with external systems.
 - **Technological Choices:** Specifies the technologies, platforms, frameworks, and databases that will be used in the system.
 - **High-Level Diagrams:** Includes diagrams like **Entity-Relationship Diagrams (ERD)**, **Data Flow Diagrams (DFD)**, and **Component Diagrams** to visualize the system's structure.
- **Example:**
 - Designing the architecture of an online shopping platform where components like the user interface, payment system, inventory system, and database interact with each other.

2. Low-Level Design (LLD):

- **Focus:** Provides detailed information about each system component and module. It zooms in on the design at a micro level, specifying how individual components will be implemented.
- **Key Components:**
 - **Detailed Module Design:** Breaks down each component or module identified in the HLD into smaller, more detailed elements, defining their functionality, logic, and interaction.
 - **Data Structures and Algorithms:** Specifies the data structures (arrays, lists, trees, etc.) and algorithms to be used in the implementation of each module.
 - **Database Design:** Includes table structures, data types, constraints, relationships, and indexes in detail, often supported by **Data Dictionary**.
 - **Interfaces:** Defines detailed interactions between different components, systems, and external systems, including input/output formats, protocols, and communication methods.
 - **Logic Flow:** Provides a detailed flow of logic, often described using **pseudocode**, **flowcharts**, or **sequence diagrams**.
 - **Detailed Diagrams:** Includes more diagrams such as **Class Diagrams**, **State Diagrams**, and **Sequence Diagrams** that define specific interactions and behaviors of the system.
- **Example:**
 - Designing the internal logic and data structure of the payment module for the online shopping platform, including transaction handling, payment verification, and error handling.

In the **Software Development Life Cycle (SDLC)**, **top-down** and **bottom-up** approaches refer to two distinct methods of designing, developing, and implementing systems or software applications. Both approaches focus on how the system is structured and the order in which components or modules are developed and integrated.

1. Top-Down Approach:

The top-down approach is a method where the system design starts at the highest level of abstraction and progressively breaks down into smaller, more detailed components. This approach focuses on defining the overall structure and gradually refining the system's subsystems and modules.

Characteristics of Top-Down Approach:

- **Focus:** Starts with the overall system design and then breaks it into smaller, manageable components or modules.
- **Process:**
 1. Begin with a high-level description of the entire system (often based on requirements and use cases).
 2. Break down the system into subsystems, then further break down subsystems into smaller modules.
 3. Continue refining each component until the design is detailed enough to implement.
 4. Code and develop each module from the top level to the bottom.

Example:

In the top-down approach, an online banking system would first be divided into major subsystems such as **account management**, **transactions**, and **customer service**. Each subsystem would then be further broken down into smaller components like **login**, **balance inquiry**, **fund transfer**, etc.

Bottom-Up Approach:

The bottom-up approach is a method where the development starts with the individual components or modules at the lowest level of abstraction, which are then integrated to form higher-level systems. The system is built from the ground up by focusing first on designing and developing the smaller, more detailed parts.

Characteristics of Bottom-Up Approach:

- **Focus:** Starts with the development of low-level modules and components and integrates them to build up the overall system.
- **Process:**
 1. Begin by identifying and developing low-level, independent modules (which can be as small as individual functions or classes).
 2. These modules are gradually combined or integrated to form larger subsystems.
 3. Continue integrating the subsystems until the entire system is built.
 4. The final system is formed through the assembly of previously created low-level modules.
- **Control Flow:** Control flows from lower-level modules to the top-level system.

Example:

In the bottom-up approach, a payment processing system might start by developing small, reusable modules like **payment validation**, **currency conversion**, and **transaction logging**. These modules would then be combined into a larger **transaction processing system** before integrating the complete payment system with other parts of the software.

4. System Development :

- In the development phase, the actual coding takes place. The design is translated into a functional software product by writing code based on the specifications.

- Developers build individual components, integrate them, and implement databases, algorithms, and user interfaces.

In system development, a variety of tools are used to facilitate and streamline different aspects of the software development process. These tools help in planning, designing, coding, testing, and maintaining software systems. Here's an overview of different types of system development tools:

1. Project Management Tools:

- **Purpose:** Manage project schedules, tasks, resources, and communication.
- **Examples:**
 - **Microsoft Project:** Provides comprehensive project management features including scheduling, resource allocation, and progress tracking.
 - **Jira:** An issue tracking and project management tool used for agile development.
 - **Trello:** A visual tool for organizing tasks and projects using boards and cards.
 - **Asana:** Helps teams plan, track, and manage work with task assignments and project timelines

2. Requirement Management Tools:

- **Purpose:** Capture, document, and manage software requirements.
- **Examples:**
 - **IBM Engineering Requirements Management DOORS:** Used for capturing and managing requirements, ensuring traceability and compliance.
 - **Atlassian Confluence:** A collaboration tool that can be used to document and manage requirements.
 - **Helix RM:** Offers features for requirement capture, traceability, and versioning.

3. Design Tools:

- **Purpose:** Create visual representations of system architecture, data flow, and user interfaces.
- **Examples:**
 - **Microsoft Visio:** A diagramming tool used for creating flowcharts, network diagrams, and other visual representations.
 - **Lucidchart:** An online tool for creating diagrams, flowcharts, and other design documents.
 - **Balsamiq Mockups:** For creating wireframes and mockups of user interfaces.
 - **Figma:** A collaborative interface design tool for creating high-fidelity UI designs and prototypes.

4. Development Environments:

- **Purpose:** Provide an integrated environment for coding, debugging, and testing software.
- **Examples:**
 - **Integrated Development Environments (IDEs):** Such as **Visual Studio**, **Eclipse**, and **IntelliJ IDEA**.
 - **Code Editors:** Such as **Visual Studio Code**, **Sublime Text**, and **Atom**.

5. Version Control Tools:

- **Purpose:** Manage and track changes to source code over time.
- **Examples:**
 - **Git:** A distributed version control system used for source code management.
 - **GitHub:** A platform for hosting Git repositories and facilitating collaboration.
 - **GitLab:** Provides Git repository management, CI/CD, and other DevOps features.
 - **Bitbucket:** A Git repository hosting service with built-in CI/CD.

6. Testing Tools:

- **Purpose:** Perform automated and manual testing to ensure software quality and performance.
- **Examples:**
 - **Selenium:** An open-source tool for automating web browsers.
 - **JUnit:** A framework for writing and running unit tests in Java.
 - **TestNG:** A testing framework inspired by JUnit, used for test configuration and management.
 - **Jenkins:** An automation server that supports continuous integration and continuous delivery (CI/CD).

7. Database Management Tools:

- **Purpose:** Design, manage, and interact with databases.
- **Examples:**
 - **MySQL Workbench:** A graphical tool for database design and management.
 - **SQL Server Management Studio (SSMS):** Provides tools for managing Microsoft SQL Server databases.
 - **pgAdmin:** A management tool for PostgreSQL databases.

5. System Testing:

- After development, the system undergoes testing to identify and fix any defects or bugs.
- The goal is to ensure that the software is free of defects, meets the requirements, and works under expected conditions.

Types of system testing

Black Box Testing

Definition:

Black box testing is a testing technique where the internal structure or workings of the application are not known to the tester. The tester focuses on inputs and outputs rather than the internal code structure. This is often used to validate functional and non-functional requirements.

Characteristics:

- Tests are based on specifications and requirements.
- The tester does not need to know the code or logic behind the application.
- Focuses on **what** the software does, not **how** it does it.

Types of Black Box Testing:

- **Functional Testing:** Testing functionalities based on requirements.
- **Non-Functional Testing:** Testing performance, usability, etc.
- **Regression Testing:** Ensuring that recent changes haven't broken existing functionality.

Examples:

1. Login Form Testing:

- Input: Valid and invalid username/password combinations.
- Output: Check whether the user is allowed or denied access.
- The tester is not concerned with how the login code works internally

White Box Testing

Definition:

White box testing (also known as clear box or glass box testing) is a testing method where the tester has knowledge of the internal workings of the system, such as the code, data flow, and control structures. The goal is to test the internal logic, code paths, and structures of the software.

Characteristics:

- Requires understanding of the code.
- Focuses on internal mechanisms (e.g., code paths, loops, conditions).
- Aims to ensure that **how** the software works is correct.

Types of White Box Testing:

- **Unit Testing:** Testing individual functions or methods.
- **Integration Testing:** Testing interactions between different units of code.
- **Code Coverage Testing:** Ensuring that all possible paths and conditions in the code are tested.

Examples:

Loop Testing in a Function:

- If a function contains a loop (e.g., `for`, `while`), the tester would analyze the loop's logic to ensure it handles all conditions properly (e.g., boundary conditions, infinite loops).
- Example: In a function that calculates the factorial of a number, the tester checks that the loop runs the correct number of times and returns the right result for input values like 0, 1, 5, etc.

User Acceptance Testing (UAT)

Definition:

User Acceptance Testing (UAT) is the final phase of the software testing process, where the end users or clients test the system to verify whether it meets their business requirements and is ready for deployment. The goal of UAT is to validate that the system functions as expected in real-world scenarios and satisfies the needs of the end users.

Types of UAT:

1. **Alpha Testing:**
 - Performed by internal users at the developer's site before the software is released to external users.
2. **Beta Testing:**
 - Performed by a limited number of external users in a real environment to gather feedback before full deployment.

6. System Deployment/Implementation:

- Once the software passes the testing phase, it is deployed to the production environment for end-users. This may involve setting up the infrastructure, migrating data, and installing the software.

Types of system Implementation

1. Direct Implementation (Big Bang)

In **direct implementation**, the old system is completely replaced by the new system in one go. It's a quick and decisive change, but also risky because if the new system fails, there is no fallback.

- **Advantages:** Simple, low cost in terms of running two systems, and fast implementation.
- **Disadvantages:** High risk if the new system fails.

Example: A retail company upgrades its point-of-sale (POS) system. The old POS system is turned off, and the new one is immediately used the next day across all stores.

2. Parallel Implementation

In **parallel implementation**, both the old and new systems run simultaneously for a period. This allows users to compare the two and ensures a smooth transition since the old system can serve as a backup if the new one fails.

- **Advantages:** Lower risk because the old system is still operational.
- **Disadvantages:** High operational costs since two systems are running simultaneously.

Example: A bank introduces a new online banking platform while keeping the old one active for three months, giving customers the choice to use either system until the new one is proven reliable.

3. Pilot Implementation

In **pilot implementation**, the new system is implemented in a small, controlled section of the organization (e.g., a single department or location) before it is rolled out to the rest of the organization. This allows for testing on a smaller scale before full implementation.

- **Advantages:** Reduces risk by identifying and solving problems before a full rollout.
- **Disadvantages:** Limited scope, so issues may arise when scaling up.

Example: A hospital introduces a new patient management system in just one department (e.g., Pediatrics) to test its functionality before rolling it out to all departments.

4. Phased Implementation

In **phased implementation**, the new system is introduced in stages or modules over a period. This method reduces risk by allowing parts of the system to be tested and adapted before the full system is operational.

- **Advantages:** Reduces risk and allows for a gradual transition.
- **Disadvantages:** Prolonged implementation time and potential integration issues between old and new systems.

Example: A university introduces a new student information system, first implementing the registration module, followed by student records, and then financial services over the course of a year.

7. **Maintenance:**

- After deployment, the software enters the maintenance phase. This includes addressing bugs discovered after release, implementing updates, and adding new features based on user feedback.
- Maintenance can also involve performance optimization, security patches, and adapting the system to changing environments.

The types of system maintenance in software and IT systems include:

1. **Corrective Maintenance:** Focuses on identifying and fixing errors, bugs, or defects in a system to ensure it works as intended.
2. **Adaptive Maintenance:** Involves updating the system to adapt to changes in the environment, such as new operating systems, hardware, or regulatory requirements.
3. **Perfective Maintenance:** Aims to enhance the system by adding new features, improving performance, or optimizing existing functions based on user feedback.
4. **Preventive Maintenance:** Proactively addresses potential issues by making adjustments or updates to prevent future problems.

A **System Analyst** is an IT professional responsible for analyzing, designing, and implementing information systems to support organizational needs. They bridge the gap between business needs and IT, working closely with stakeholders to ensure that technical solutions align with business goals.

Roles and Responsibilities of a System Analyst:

1. **Requirement Gathering and Analysis:**
 - Work with stakeholders, end-users, and clients to understand and document the system's requirements.
2. **System Design and Specification:**
 - Create data flow diagrams, entity-relationship diagrams, and process models to represent system functionalities.
 - Design user interfaces and workflows to ensure usability.
3. **Feasibility Analysis:**
 - Conduct feasibility studies to assess the technical, operational, and financial feasibility of a proposed system.
4. **Communication Between Stakeholders and Technical Teams:**
 - Act as a bridge between end-users, management, and the development team.
5. **Implementation and Support:**
 - Provide support during and after implementation to troubleshoot and resolve issues.

6. **Continuous Improvement and Optimization:**

- Monitor system performance and collect user feedback to identify areas for improvement.

7. **Documentation and Training:**

- Create detailed documentation for system operations, including user manuals and technical guides.

Key Skills of a System Analyst:

- Strong analytical and problem-solving skills
- Proficiency in system modeling, design, and documentation tools
- Good understanding of software development and project management methodologies
- Excellent communication and interpersonal skills
- Knowledge of database management and programming (often beneficial)

A **Software Engineer** is a professional responsible for designing, developing, testing, and maintaining software applications and systems. They apply engineering principles and knowledge of programming to solve complex software problems, develop new applications, and improve existing software solutions.

Roles and Responsibilities of a Software Engineer:

1. **Software Development and Coding:**

- Write clean, efficient, and maintainable code for software applications based on project requirements.

2. **Software Design and Architecture:**

- Work with system architects and other engineers to design software structures that align with system requirements and constraints.

3. **Requirements Analysis:**

- Translate business requirements into technical specifications for development.

4. **Testing and Debugging:**

- Identify, troubleshoot, and fix bugs or issues in the software.

5. **Continuous Integration and Deployment (CI/CD):**

- Ensure new code integrates seamlessly with existing code and doesn't introduce issues in the production environment.

6. **Software Maintenance and Updates:**

- Monitor software performance and make updates or patches as necessary to maintain smooth functionality.

Key Skills of a Software Engineer:

- Proficiency in programming languages and software development frameworks
- Strong problem-solving and analytical skills
- Knowledge of software development methodologies like Agile, Scrum, or DevOps
- Understanding of database management and system architecture
- Good teamwork and communication skills

System Design Tools

System design tools are the tools that are used for essential for planning, visualizing, and documenting software architectures, workflows, and system components.

1. Algorithm

An **algorithm** is a set of well-defined, step-by-step instructions designed to perform a specific task or solve a particular problem. Algorithms are foundation in computer science and programming, providing clear guidance for automating processes and solving computational problems.

Key Characteristics of an Algorithm:

- **Input:** Information or data the algorithm needs to start.
- **Output:** The result or solution after executing the algorithm.
- **Definiteness:** Each step is clear and unambiguous.
- **Effectiveness:** Each step must be simple enough to be performed, ideally by a computer.
- **Finiteness:** The algorithm must eventually terminate, i.e., it has a clear endpoint.

2. Flowchart

A **flowchart** is a diagram that represents the flow or sequence of steps in a process. It's commonly used to outline algorithms, procedures, and workflows, using various symbols to illustrate steps, decisions, inputs, outputs, and loops.

Basic Flowchart Symbols:

1. **Oval:** Indicates the **Start** and **End** of the flowchart.
2. **Rectangle:** Represents a **Process** or **Step** (e.g., calculations, instructions).
3. **Diamond:** Denotes a **Decision** or branching point (e.g., yes/no, true/false).
4. **Parallelogram:** Shows **Input** and **Output** (e.g., getting user input or displaying results).
5. **Arrow:** Shows the **Flow Direction** from one step to another.

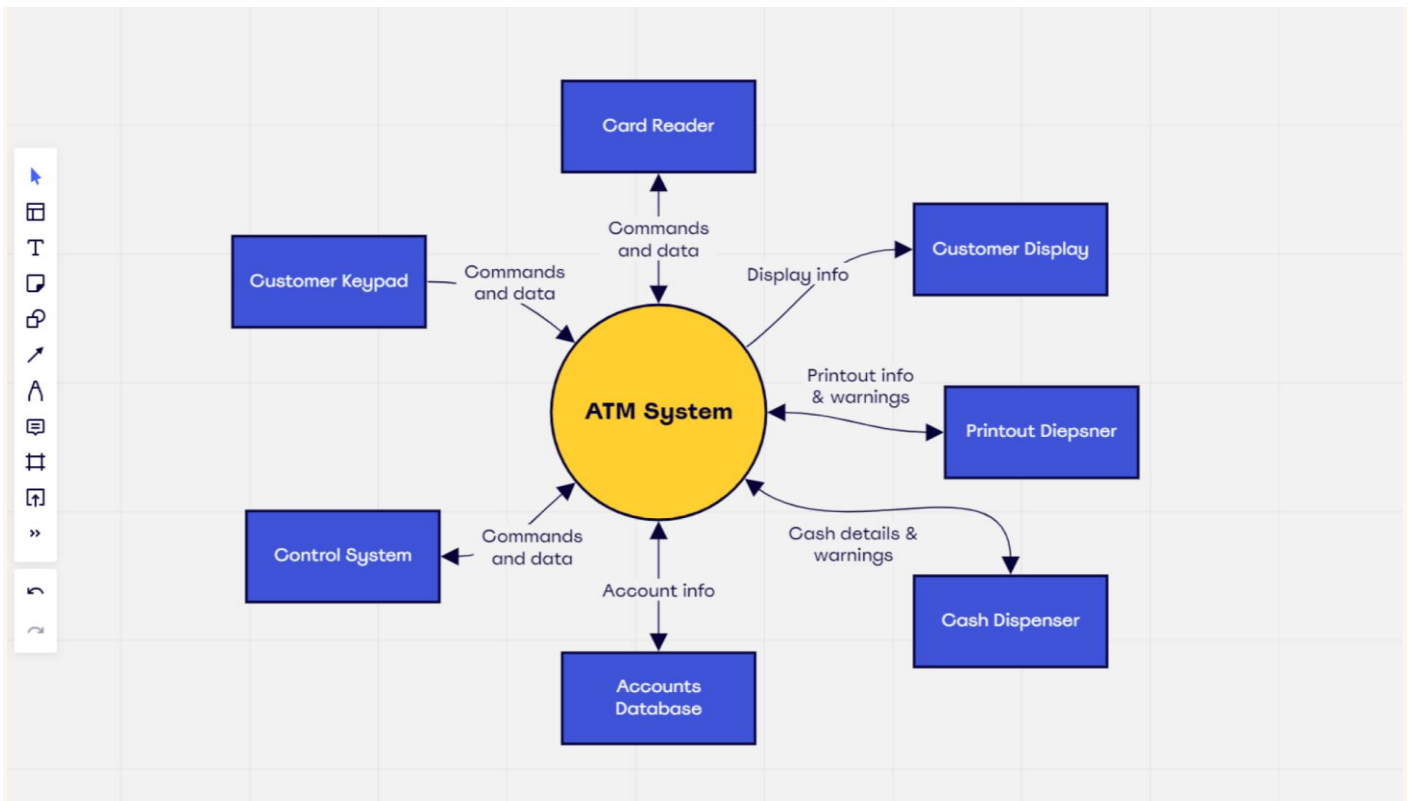
3. Pseudocode

Pseudocode is a plain-language way of writing algorithms or instructions that resembles programming logic but avoids specific syntax of any language. It's commonly used to plan or outline code before actual programming.

4. Context diagram

A context diagram outlines how external entities interact with an internal software system.

Context diagrams are high-level diagrams, meaning they don't go into the detailed ins and outs of the system. Instead, they map out an entire system in a way that's simple, clear, and easy to understand. Because of its simplicity, it's sometimes called a level 0 data flow diagram.

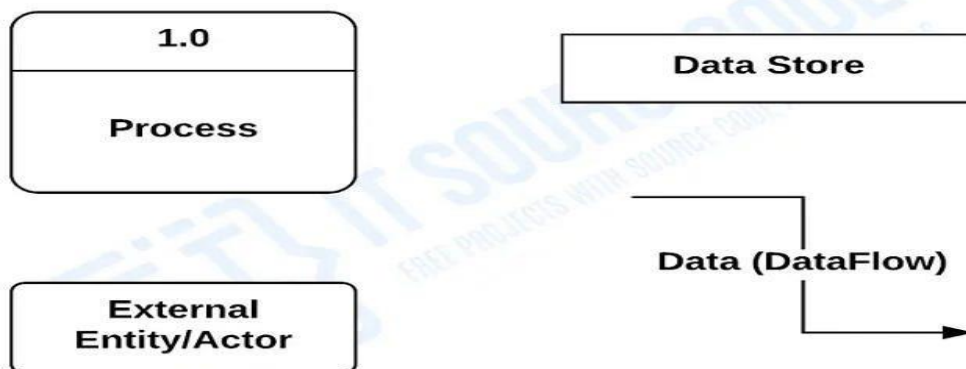


5. Data flow diagrams

- The **DFD or data flow diagram** is an overall flow of how the data moves through a system, describing its inputs and outputs process within the entire system.
- Furthermore, the **data flow diagram** (DFD) is a graphical representation of the system data process management structure.


These **Data Flow Diagram** symbols are the following:

DATA FLOW DIAGRAM SYMBOLS

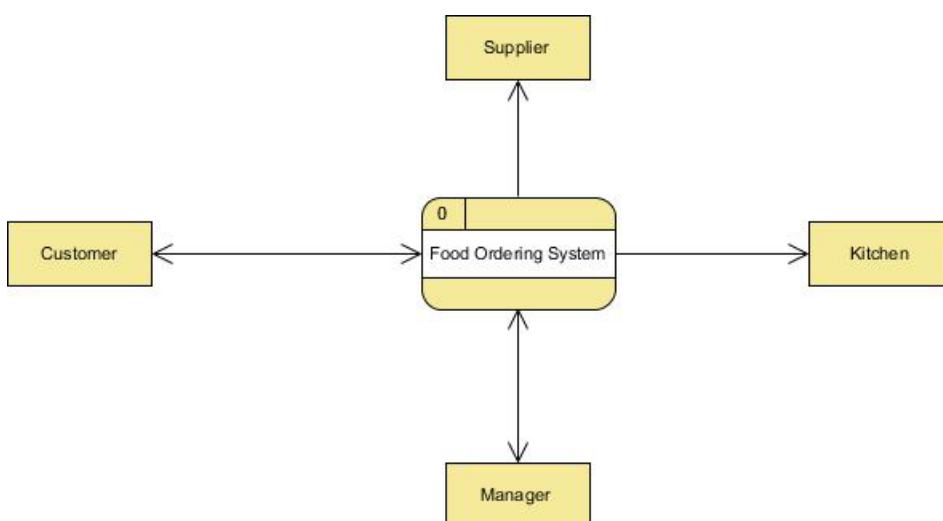


- **External Entities** are the entry and exit points for data entering and leaving the system. Entities are referred to as terminators, sources, sinks, and actors.

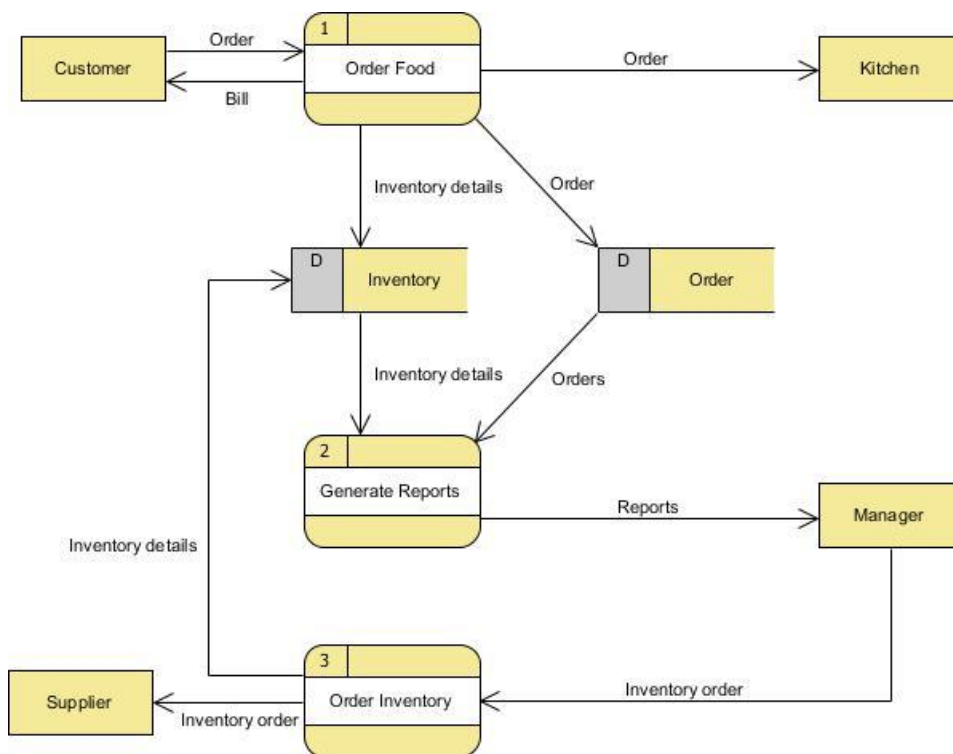
- The **process** is the portion of DFD that modifies and generates output from data.
- A **Data Store (database)** is a table that stores the files or repositories for future use.
- **Data Flow** is the flow of data between external entities, processes, and data stores.

Notation	Yourdon & De Marco	Gene & Sarson	SSADM	Unified
External Entity				
Process				
Data Store				
Data Flow				

Level 0 DFD



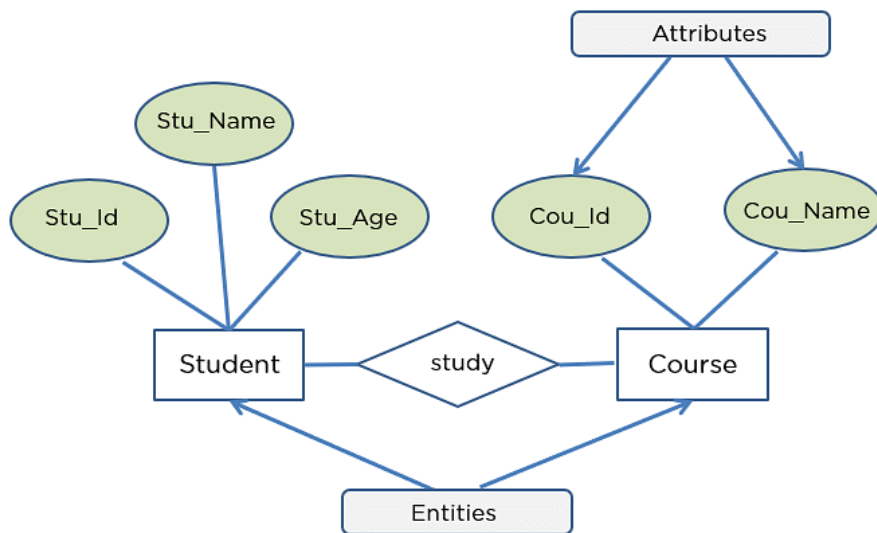
Level 1 DFD



Entity- Relationship Diagram

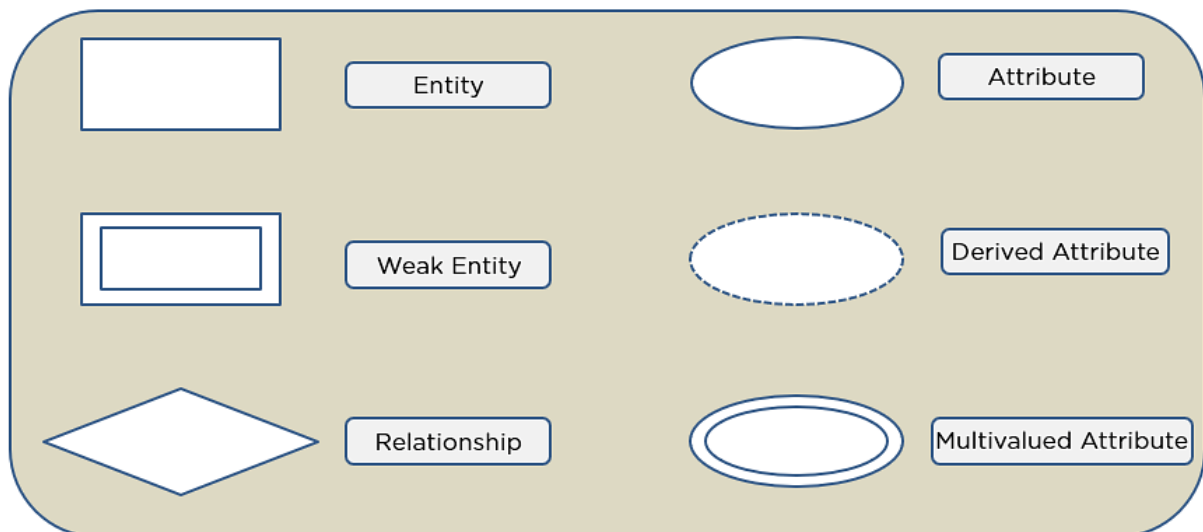
An Entity Relationship Diagram is a diagram that represents relationships among entities in a database. It is commonly known as an ER Diagram. An ER Diagram in **DBMS** plays a crucial role in designing the database. Today's business world previews all the requirements demanded by the users in the form of an ER Diagram. Later, it's forwarded to the database administrators to design the database.

In relationship to databases and ERD, cardinality specifies how many instances of an entity relate to one instance of another entity



Symbols Used in ER Diagrams

- Rectangles: This Entity Relationship Diagram symbol represents entity types
- Ellipses: This symbol represents attributes
- Diamonds: This symbol represents relationship types
- Lines: It links attributes to entity types and entity types with other relationship types
- Primary key: Here, it underlines the attributes
- Double Ellipses: Represents multi-valued attributes



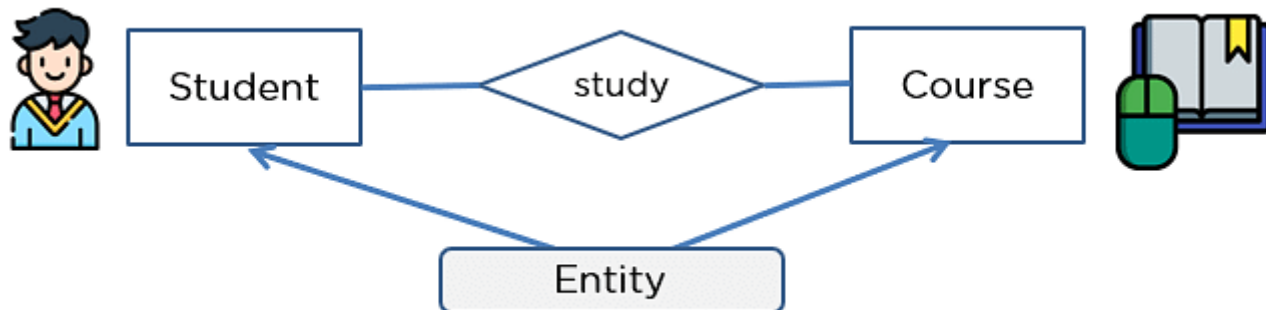
Entities

COMPILED BY: Er.Gaurab Mishra
HOD, COMPUTER SCIENCE
K.M.C COLLEGE, +2

An entity can be either a living or non-living component.

It showcases an entity as a rectangle in an ER diagram.

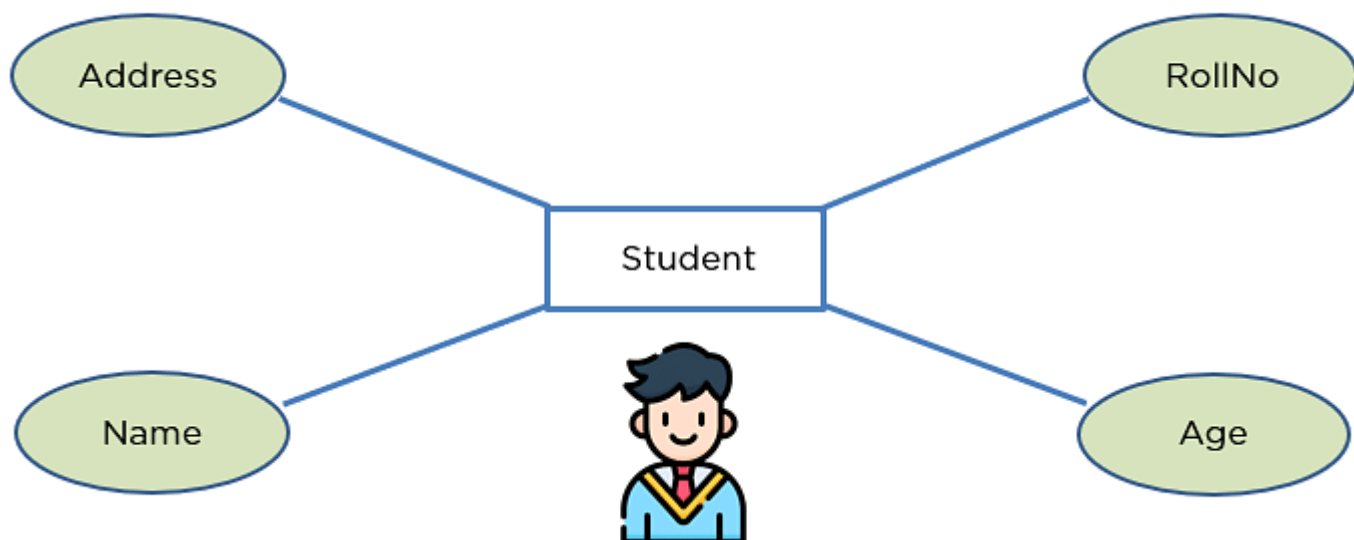
For example, in a student study course, both the student and the course are entities.



Attribute

An attribute exhibits the properties of an entity.

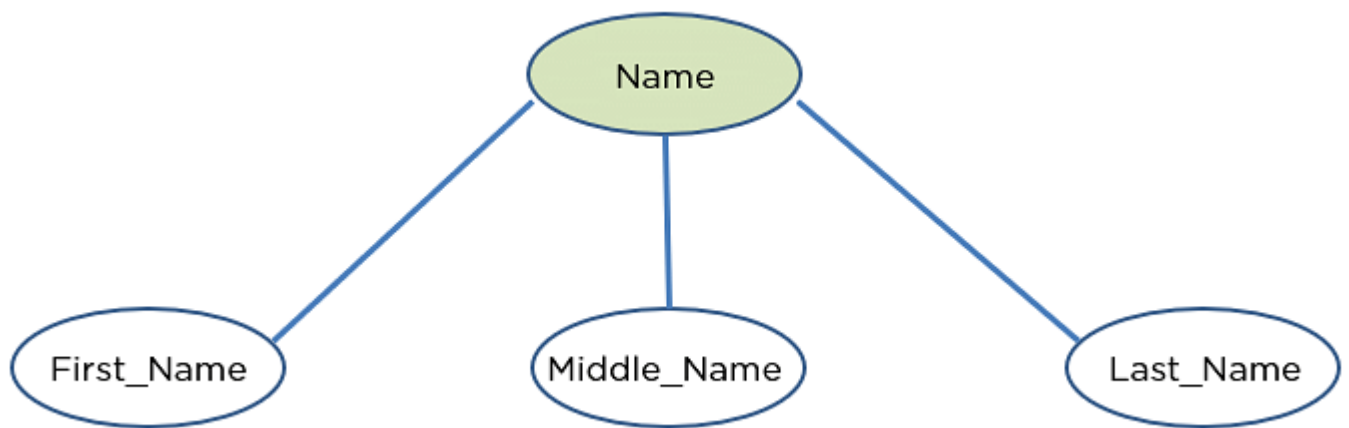
You can illustrate an attribute with an oval shape in an ER diagram.



Composite Attribute

An attribute that is composed of several other attributes is known as a composite attribute.

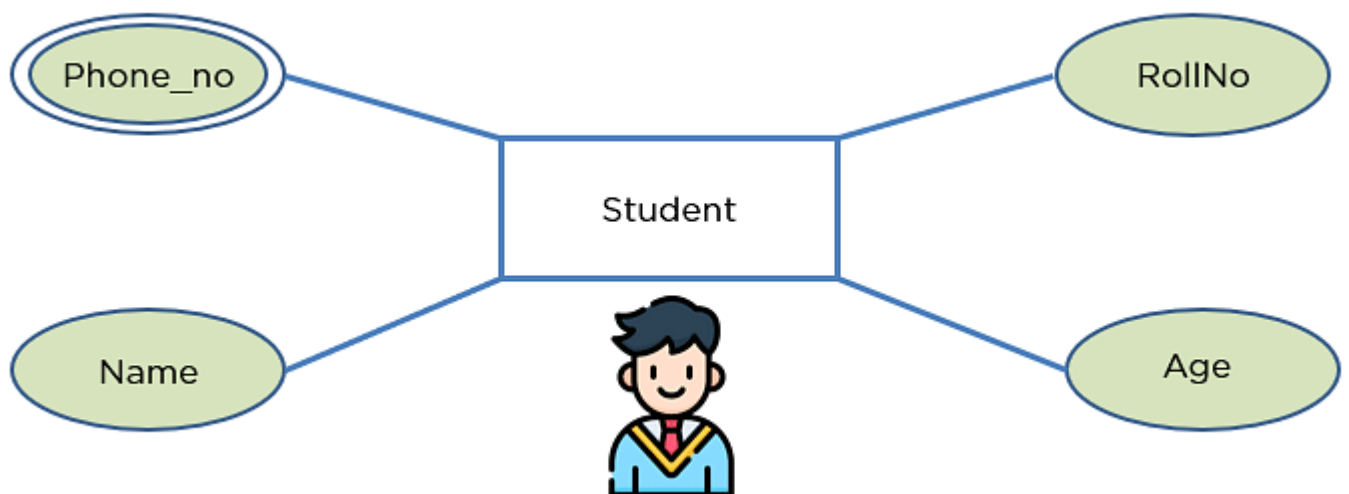
An oval showcases the composite attribute, and the composite attribute oval is further connected with other ovals.



Multivalued Attribute

Some attributes can possess over one value, those attributes are called multivalued attributes.

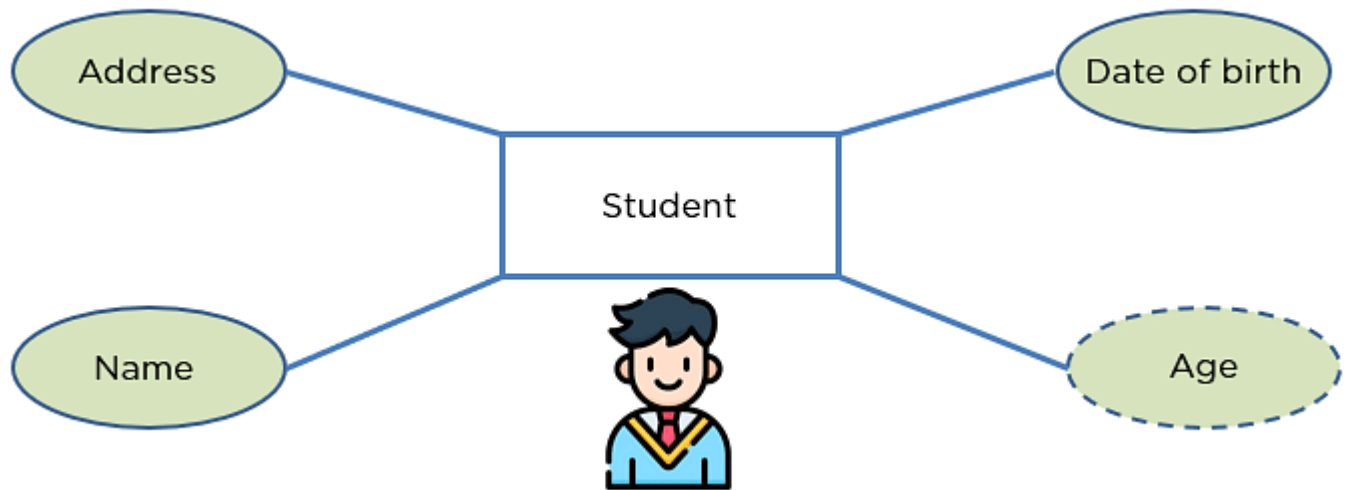
The double oval shape is used to represent a multivalued attribute.



Derived Attribute

An attribute that can be derived from other attributes of the entity is known as a derived attribute.

In the ER diagram, the dashed oval represents the derived attribute.

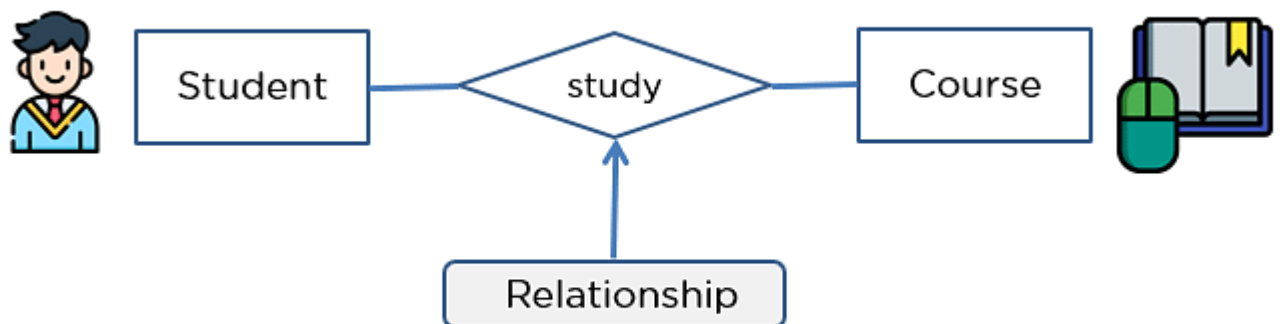


Relationship

The diamond shape showcases a relationship in the ER diagram.

It depicts the relationship between two entities.

In the example below, both the student and the course are entities, and study is the relationship between them.

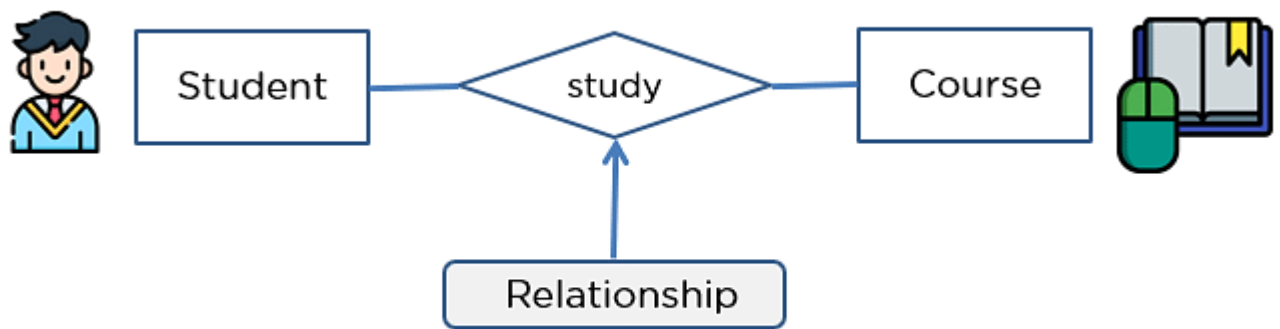


Relationship

The diamond shape showcases a relationship in the ER diagram.

It depicts the relationship between two entities.

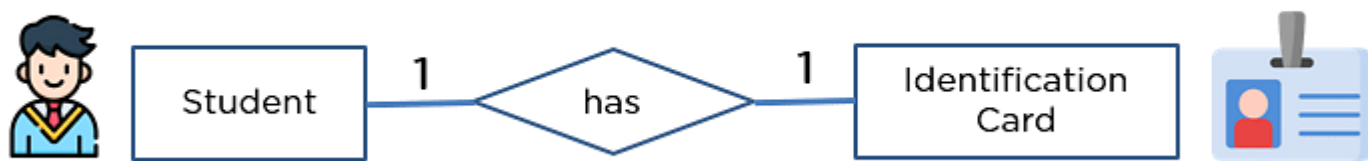
In the example below, both the student and the course are entities, and study is the relationship between them.



One-to-One Relationship

When a single element of an entity is associated with a single element of another entity, it is called a one-to-one relationship.

For example, a student has only one identification card and an identification card is given to one person.



One-to-Many Relationship

When a single element of an entity is associated with more than one element of another entity, it is called a one-to-many relationship

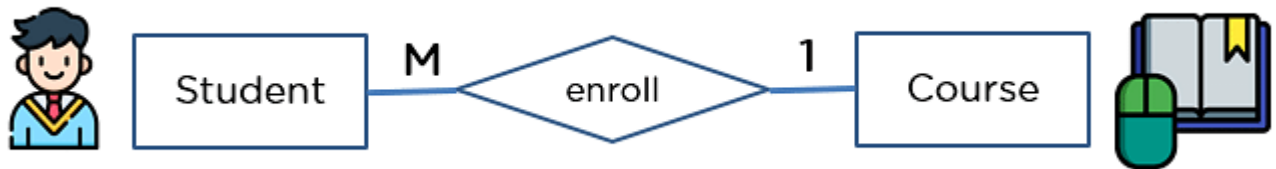
For example, a customer can place many orders, but an order cannot be placed by many customers.



Many-to-One Relationship

When more than one element of an entity is related to a single element of another entity, then it is called a many-to-one relationship.

For example, students have to opt for a single course, but a course can have many students.



Many-to-Many Relationship

When more than one element of an entity is associated with more than one element of another entity, this is called a many-to-many relationship.

For example, you can assign an employee to many projects and a project can have many employees.

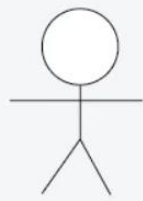


Use Case Diagrams

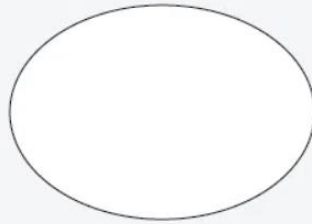
A Use Case Diagram in UML is a visual representation that illustrates the interactions between users (actors) and a system. It captures the functional requirements of a system, showing how different users engage with various use cases, or specific functionalities, within the system.

Use cases represent only the functional requirements of a system. Other requirements such as business rules, quality of service requirements, and implementation constraints must be represented separately, again, with other UML diagrams.

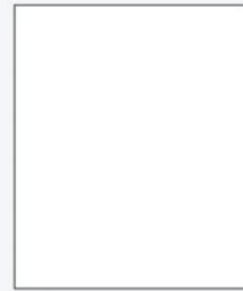
Use Case Diagram Notations



Actor



UseCase



System Boundary

Actors

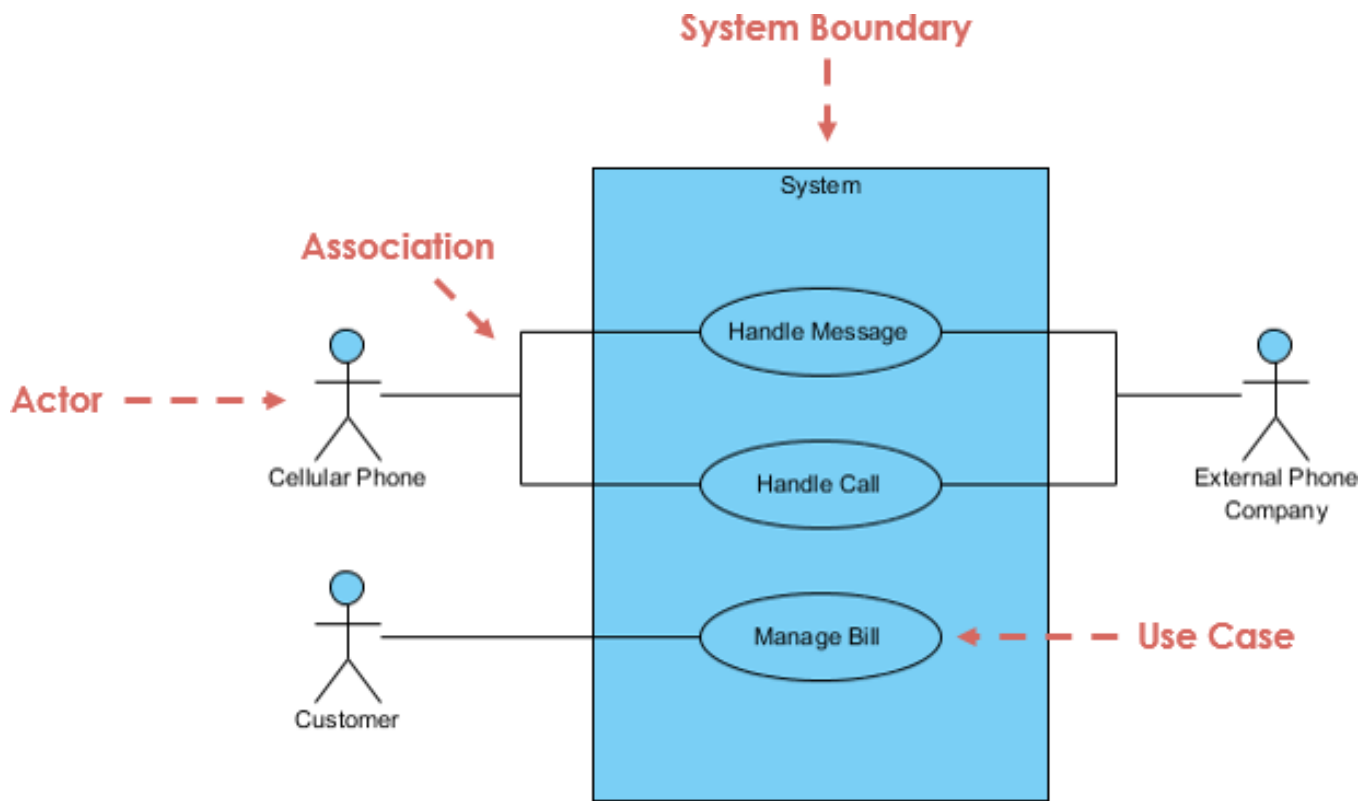
Actors are external entities that interact with the system. These can include users, other systems, or hardware devices.

Use Cases

Use cases are like scenes in the play. They represent specific things your system can do. In the online shopping system, examples of use cases could be “Place Order,” “Track Delivery,” or “Update Product Information”. Use cases are represented by ovals.

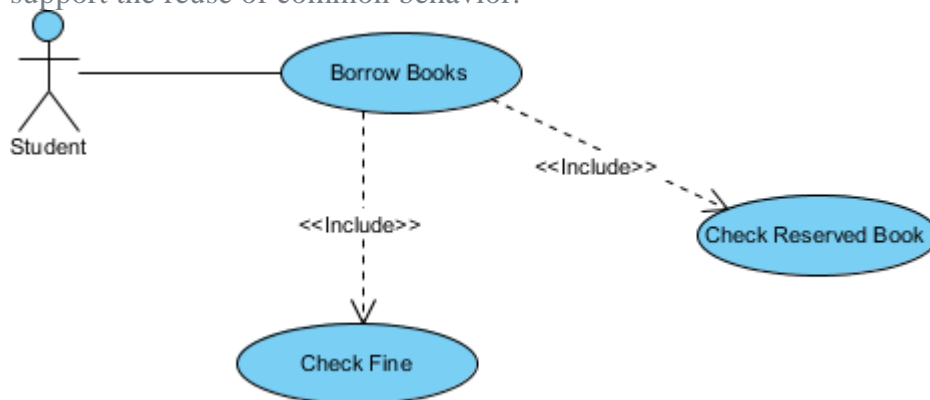
System Boundary

The system boundary is a visual representation of the scope or limits of the system you are modeling. It defines what is inside the system and what is outside.



Use Case Example – Include Relationship

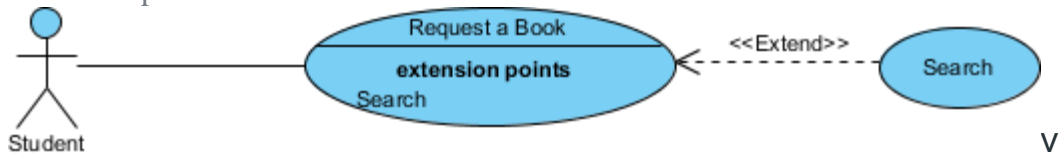
The include relationship adds additional functionality not specified in the base use case. The `<<include>>` relationship is used to include common behavior from an included use case into a base use case in order to support the reuse of common behavior.



Use Case Example – Extend Relationship

The extend relationships are important because they show optional functionality or system behavior. The `<<extend>>` relationship is used to include optional behavior from an extending use case in an extended

use case. Take a look at the use case diagram example below. It shows an extend connector and an extension point "Search".



How to Identify Actor

- Who uses the system?
- Who installs the system?
- Who starts up the system?
- Who maintains the system?
- Who shuts down the system?
- What other systems use this system?
- Who gets information from this system?
- Who provides information to the system?
- Does anything happen automatically at a present time?

How to Identify Use Cases?

- What functions will the actor want from the system?
- Does the system store information? What actors will create, read, update or delete this information?
- Does the system need to notify an actor about changes in the internal state?
- Are there any external events the system must know about? What actor informs the system of those events?