# Entity-Relationship Diagram

An Entity-Relationship Diagram (ERD) is a visual representation used in database design to depict the structure and relationships between different entities (or tables) within a database system. ERDs are a critical tool for database developers, designers, and stakeholders to understand and communicate how data is organized and connected in a database.

Entities: These represent the main objects or concepts in the database, such as customers, products, employees, or orders. Each entity is typically depicted as a rectangle in the diagram.

Attributes: Attributes describe the properties or characteristics of each entity. For example, a "Customer" entity may have attributes like "CustomerID," "FirstName," "LastName," and "Email.";

Relationships: Relationships illustrate how entities are connected to one another. They show how data is related and can be used to establish links between different entities. Common types of relationships include one-to-one, one-to-many, and many-to-many.

# Entities

Entities can be categorized into different types based on their roles and characteristics within the database model.

**Strong Entities**: These are entities that can exist independently and have their attributes. They represent primary objects in the database. For example, in a university database, the "Student" entity is a strong entity with attributes like student ID, name, and date of birth.

**Weak Entities**: Weak entities depend on a related strong entity for their existence. They do not have a primary key attribute of their own but instead have a partial key that connects them to a strong entity. A common example is the "Address" entity, which may rely on a "Person" entity to exist.

**Associative Entities**: Also known as junction entities or linking entities, associative entities are used to represent many-to-many relationships between two other entities. They often contain attributes that are specific to the relationship itself. For instance, in a library database, an "Authorship" entity could link "Author" and "Book" entities, indicating which author(s) wrote which book(s).

**Derived Entities**: These entities contain attributes that can be calculated or derived from other attributes within the database. They are often used to store information that can be computed when needed, rather than being explicitly stored. An example might be a "Total Price" entity in an e-commerce system, which can be derived from the quantity and unit price of items in a shopping cart.

Subtypes and Supertypes: Subtype entities represent specific categories or subsets of a supertype entity. This is commonly used in situations where entities share common attributes but also have attributes unique to their subtype. For instance, "Employee" could be a supertype with subtypes "Manager" and "Clerk," each having their own set of attributes.

**Singleton Entities:** These entities represent a **single**, unique instance within the database. They are often used for settings or configuration data that should have only one record. For example, a "Settings" entity might contain global settings for an application.

**Historical Entities**: These entities are used to store historical data, such as records of past transactions or changes in status. Historical entities often have attributes like "Effective Date" and "End Date" to track when data was valid.

**External Entities:** In some database models, external entities represent entities from external systems or sources that interact with the database. They are often used in data modeling to illustrate how the database interfaces with external systems or data.

# **Types of Attributes**

**Attributes:** Attributes are **characteristics or properties** that describe entities. Attributes provide detailed information about the entities represented in a database.

**Simple Attributes:** These are atomic attributes that cannot be divided any further. For instance, "EmployeeID" or "FirstName" are simple attributes because they represent singular data elements.

**Composite Attributes**: Composite attributes can be divided into sub-parts that have independent meaning. For example, an "Address" attribute may consist of sub-attributes like "Street," "City," "State," and "ZIP Code."

**Derived Attributes:** Derived attributes are calculated or derived from other attributes within the database. They do not store data directly but instead are generated based on certain rules or calculations. For instance, the "Age" attribute can be derived from the "DateOfBirth" attribute by subtracting the birthdate from the current date.

**Multi-valued Attributes**: Multi-valued attributes can have multiple values associated with them for a single entity. For example, an "Interests" attribute for a "Person" entity could contain multiple hobbies or interests.

**Key Attributes**: Key attributes are used to uniquely identify each instance of an entity. In many cases, one of the attributes is designated as the primary key, which ensures its uniqueness within the entity.

**Composite Key**: A composite key consists of multiple attributes that, when combined, uniquely identify an entity. This is often used when a single attribute is insufficient for ensuring uniqueness.

**Null Attributes**: Null attributes are those that may not have a value for every instance of an entity. They are used to represent missing or unknown data.

# **Relationship**

**Relationship:** A relationship refers to the association or connection between two or more entities within a database. Relationships are used to model how data in different entities is related or connected to each other; They provide a way to represent the interactions and dependencies between entities in a structured and organized manner.

**Cardinality**: Cardinality in a relationship defines the number of instances (records or rows) in one entity that can be associated with the number of instances in another entity. It specifies the nature of the relationship;

One-to-One (1:1): Each instance in one entity is associated with exactly one instance in another entity;

One-to-Many (1:N): Each instance in one entity is associated with one or more instances in another entity, but each instance in the second entity is associated with only one instance in the first entity;

**Many-to-Many (N:N):** Many instances in one entity can be associated with many instances in another entity. To represent many-to-many relationships in a database;

# **Multiplicity**

The Multiplicity attribute of a relationship specifies the cardinality or number of instances of an EntityType that can be associated with the instances of another EntityType;

multiplicity refers to the numeric constraints or restrictions placed on the number of instances that can be associated between two entities in a relationship.

Common multiplicity constraints include;

**One (1):** This constraint indicates that one instance from one entity can be associated with one instance from another entity, it represents a **one-to-one (1:1)** relationship.

**Zero or One (0..1)**: This constraint means that an instance from one entity can be associated with either zero instances or one instance from another entity, and vice versa. It represents a one-to-one (1:1) relationship where the **association is optional for one or both entities**;

**One or Many (1..N)**: This constraint signifies that one instance from one entity can be associated with one or more instances from another entity, but each instance from the second entity can be associated with only one instance from the first entity. It represents a one-to-many (1:N) relationship.

**Zero or Many (0..N)**: This constraint indicates that an instance from one entity can be associated with zero or more instances from another entity, and vice versa. It represents a one-to-many (1:N) relationship where the association is optional for one or both entities.

**Exactly N (N)**: This constraint specifies that an instance from one entity must be associated with exactly N instances from another entity, and vice versa. It represents a **one-to-N (1:N) relationship with a fixed number of associations**.

**Many (N):** This constraint means that an instance from one entity can be associated with many instances from another entity, and vice versa. It represents a many-to-many (N:N) relationship;

# **ER diagram notations and symbols**

Similar to other diagram shapes, in [ER diagrams](https://www.gleek.io/blog/erd-database-design) entities are represented with rectangles, attributes are represented by ovals or inside entities, and relationships are shown in diamonds or with lines. There are six notation types;

## **Chen notation symbols**

An entity is shown in a **rectangle**, just like in many other notations. But, that is where the similarities stop.

There are 2 more ways to describe entities:

* Weak entity – A rectangle within a rectangle
* Associative entity – A diamond within a rectangle; An entity used in **a many-to-many** relationship

Attributes are in ovals. Here are some other symbols used to define attributes:

* **Key attribute** – The **title of the attribute is underlined**
* **Partial key attribute** – The attribute’s name is underlined with a dashed line
* [**Composite attributes**](https://www.gleek.io/blog/er-model-attributes) – a composite attribute is an attribute that **can be divided into smaller, more basic sub-attributes**, each with its own meaning and significance. Composite attributes are used when you want to represent a complex attribute that is composed of multiple parts, and are a different color;
* **Multivalued attribute** – An oval within an oval
* **Computed/derived attribute** – An oval with a dashed line

Relationships are defined with optionality, cardinality, degree, participation, and type; using lines and diamonds.

**Type**

* Strong relationship – **A solid-lined diamond**
* Weak relationship – A **diamond within a diamond**, like a weak entity

**Optionality**

* Mandatory – A solid line
* Optional – A dashed line

[**Cardinality (degree)**](https://www.gleek.io/blog/er-model-cardinality)

* One to one – A **1 is at each end of the relationship**
* One to many – A **1 is at one end**, and **N is at the other.** N represents ‘many’.
* Many to many – **M is on one** end, and **N is on the other**

**Participation**

* Total participation – Two parallel lines
* Partial participation – One line

|  |
| --- |
| Chen ERD Symbols |

|  |
| --- |
| Chen ERD Symbols |

## **Crow Notaion**

Entities are in rectangles with their attributes inside. Relationships are defined much like they are in other notations, but major difference is the presence of multiplicities. Multiplicities are symbols that tell the reader the number of times instances can associate with others.

**Multiplicity symbols**

|  |
| --- |
| Crow’s Foot Notation |

|  |
| --- |
| Cardinality in Crow's Foot Notation |

**Example of Crew Notaion**

|  |
| --- |
| ER Diagram - Crow's Foot Notation |

# **Unified Modeling Language**

UML is a modern approach **to modeling and documenting** software. In fact, it’s one of the most popular [business process modeling techniques](https://tallyfy.com/business-process-modeling-techniques);

UML diagrams, are used to communicate different aspects and characteristics of a system; However, this is only a top-level view of the system and will most probably not include all the necessary details to execute the project until the very end.

**Forward Design** – The design of the sketch is done before coding the application. This is done to get a better view of the system or workflow that you are trying to create. Many design issues or flaws can be revealed, thus improving the overall project health and well-being.

**Backward Design** – After writing the code, the UML diagrams are drawn as a form of documentation for the different activities, roles, actors, and workflows.

# **OOP Concepts**

#### **Inheritance:** Inheritance in OOPs makes you inherit a sub-class from the superclass. When you inherit a class from the superclass, you can access the superclass properties, constructors, methods, and objects besides the properties or methods private to the superclass.

#### **Abstraction**: Abstraction makes you hide the details of functionalities of a class from a user. In abstraction, you initialize methods in an abstract class and define those methods in the inherited sub-class.

#### **Encapsulation:** Encapsulation makes you **provide security to your properties and methods from the outer world**. You define encapsulation by making getter and setter methods.

#### **Polymorphism:** "Poly" means "many," and "mor" means "form or face." Binding together makes "many forms.” Polymorphism makes you make **many attributes and methods from a single method or attribute in a class** by method overloading and method overriding concept.

# **Difference between ERD and UML**

UML is used to design the whole software architecture and to track it and ERD is used to design and implement the databases;

## **Types of UML Diagrams**

There are several types of UML diagrams and each one of them serves a different purpose;

It’s a rich language to model software solutions, application structures, system behavior and [business processes](https://creately.com/blog/diagrams/importance-of-business-process-modeling/).

The two broadest categories that encompass all other types are **Behavioral** UML diagram and **Structural** UML diagram. As the name suggests, some UML diagrams try to analyze and depict the structure of a system or process, whereas other describe the behavior of the system, its actors, and its building components. The different types are broken down as follows:

The most frequently used ones in software development are: Activity diagrams, Use Case diagrams, Class diagrams, and Sequence diagrams.

### **Behavioral UML Diagram**

[**Activity Diagram**](https://tallyfy.com/uml-diagram/#activity-diagram)

Activity diagrams are probably the most important UML diagrams for doing [business process modeling](https://tallyfy.com/business-process-modeling/). In software development, it is generally used to describe the flow of different activities and actions. These can be both sequential and in parallel. They describe the objects used, consumed or produced by an activity and the relationship between the different activities;

|  |
| --- |
| Activity UML Diagram |

[**Use Case Diagram**](https://tallyfy.com/uml-diagram/#use-case-diagram)

A cornerstone part of the system is the [**functional requirements**](https://reqtest.com/requirements-blog/functional-vs-non-functional-requirements/)**that the system fulfills**. Use Case diagrams are used to analyze the system’s [high-level requirements](http://www.testablerequirements.com/testablerequirements/ident_hlrs.htm). These requirements are expressed through different use cases. We notice three main components of this UML diagram:

A [Use Case Diagram](https://www.edrawsoft.com/uml-use-case.html) can be regarded as a good starting point for discussing project key actors and processes without going into too many implementation details;

**Functional requirements** – represented as use cases; a verb describing an action

**Actors** – they interact with the system; an actor can be a human being, an organization or an internal or external application

**Relationships** between actors and use cases – represented using straight arrows

|  |
| --- |
| Use Case UML Diagram |

[**Sequence Diagram**](https://tallyfy.com/uml-diagram/#sequence-diagram)

Sequence diagrams describe the sequence of messages and interactions that happen between actors and objects in a sequential order.  Actors or objects can be active only when needed or when another object wants to communicate with them. All communication is represented in a chronological manner. To get a better idea, check the example of a UML sequence diagram below.

|  |
| --- |
| Sequence UML Diagram |

### **Structural UML Diagram**

[**Class Diagram**](https://tallyfy.com/uml-diagram/#class-diagram)

Since most software being created nowadays is still based on the [**Object-Oriented Programming paradigm**](https://en.wikipedia.org/wiki/Object-oriented_programming), using class diagrams to document the software turns out to be a common-sense solution. This happens because OOP is based on classes and the relations between them.

In a nutshell, class diagrams contain classes, alongside with their attributes and their behaviors (member functions). More specifically, each class has 3 fields: **the class name at the top, the class attributes right below the name, the class operations/behaviors at the bottom.** The relation between different classes (represented by a connecting line), makes up a class diagram

|  |
| --- |
| Class UML Diagram |

# **UML Diagram Notations**

|  |
| --- |
| uml diagram symbols |