**ERD**

Proj\_1

A diagram of a flowchart

AI-generated content may be incorrect.

Proj\_2

A diagram of a flowchart

AI-generated content may be incorrect.

Proj\_3

A diagram of a company

AI-generated content may be incorrect.

Proj\_4

A diagram of a company

AI-generated content may be incorrect.

Proj\_5

A diagram of a network

AI-generated content may be incorrect.

**Self**

* **Relationship between ERD & UML:**

**1. ERD and UML are both modeling tools**

* Both are used during the **system design phase**.
* They help visualize and plan the system **before implementation**.
* But each focus on a different aspect:
  + **ERD** focuses on **data structure**
  + **UML** focuses on **system structure and behavior**

**2. ERD can be represented inside UML**

* **Class Diagrams** in UML can **represent the same information as ERDs**.
  + Entities → Classes
  + Attributes → Class attributes
  + Relationships → Associations between classes

So, **an ERD can be considered a simplified form of a UML Class Diagram**, focused only on data.

* **Where is the equation written?**

The derived entity equation is usually documented in the **design documents** such as the **ERD (Entity-Relationship Diagram)**, **Data Dictionary**, or **Business Logic Specifications**. It may also be written directly in the **backend code** (e.g., SQL queries or business logic functions).

**How does the developer receive it?**  
The developer receives the derived equation either from:

* The **System Analyst** or **Database Designer** through proper documentation.
* A shared **business logic document** that outlines how each derived attribute is calculated.
* Or by direct communication with the analyst or client if documentation is missing.

**How is it implemented?**  
The equation is usually implemented as:

* A computed column in a SQL query or view.
* A function or formula in backend code (Python, Java, etc.).
* **Relationship Between Strong Entity and Weak Entity (as Classes):**

A **Strong Entity** is like an **independent class (Parent/Base Class)**. It has its own primary key and can exist on its own.

A **Weak Entity** is like a **dependent class (Child or Composed Class)**. It **cannot exist without the strong entity** and depends on it for identification.

The relationship between them is similar to **Composition** in Object-Oriented Programming:

* The weak entity is **owned** by the strong entity.
* If the strong entity is deleted, the weak one usually cannot exist.
* **Enhanced ERD, Inheritance, and SOLID:**

**Enhanced ERD (EERD)** extends the basic ERD by introducing advanced concepts like:

* **Generalization**
* **Specialization**
* **Inheritance**
* **Aggregation**

These features directly support **object-oriented programming (OOP)** by allowing entities to be modeled using **class inheritance**, where:

* Superclasses (general entities) define shared attributes.
* Subclasses (specialized entities) inherit and extend those attributes.

It helps in building structured class hierarchies and supports **SOLID principles** by:

* Promoting responsibility separation (SRP)
* Enabling extension without modification (OCP)
* Supporting substitutability (LSP)
* Encouraging cleaner, focused models (ISP)
* Allowing abstraction and dependency inversion (DIP

* **The selection of a primary key depends:**

**1. Nature of the Data**

* Is there a naturally unique attribute already in the table?  
  → If **yes**, it may be used as a **natural key** (e.g., National ID, ISBN).  
  → If **no**, then use a **surrogate key** (e.g., auto-incremented ID).

**2. Stability of the Attribute**

* Will the value **remain constant** over time?  
  → Choose attributes that **don’t change** (e.g., employee ID, not name/email).

**3. Data Integrity**

* Is the attribute **always available and non-null**?  
  → Must choose an attribute that always exists for every row.

**4. Performance**

* Is the attribute suitable for **fast indexing and searching**?  
  → Numeric keys (like integers) are better for performance than long strings.

**5. Simplicity and Clarity**

* Is the key **simple and easy to use** in queries and joins?  
  → A single-column key is better than a multi-column (composite) key unless necessary.

**6. Uniqueness Across the System**

* Will the key remain **unique even in the future**?  
  → Make sure it won’t cause duplication later when data grows or integrates with other systems.
* **Why Do We Define a Primary Key When Modeling the System as Classes**

| **Reason** | **Explanation** |
| --- | --- |
| Unique Object Identity | Helps distinguish each object (like a row in a table) |
| ORM and Database Mapping | Required for mapping classes to database tables |
| Object Updates and Tracking | Makes it easy to update/delete specific instances |
| Relationships Between Classes | Enables linking classes just like foreign keys |
| System Consistency | Keeps object models aligned with database structure |