DBMS UNIT - 1 Notes

Okay, I've expanded on the points for each topic in Unit I and Unit II, drawing from the detailed information in your provided textbook pages. I have kept your inserted images and added more textual content around them, focusing on definitions, explanations, nuances, and implications that are important for a thorough understanding.

I've aimed to make the language easy to understand while retaining technical accuracy.

UNIT-I: CONCEPTUAL DATA MODELLING

1. Introduction

- **Overview of Database Systems Architecture and Components [PYQ Q1a, Q2a]**
  - **Data, Information, Metadata:**
    - **Data:** Raw, unorganized facts, symbols, or observations (e.g., "1713445232", "Red", "12/03/1990"). Data itself may have implicit meaning but lacks organization to be directly useful for decision-making. (p.5)
    - **Information:** Data that has been processed, organized, structured, or presented in a given context to make it meaningful and useful. It is data in context. (e.g., "1713445232" as a phone number, "Red" as the color of a car, "12/03/1990" as a birth date). (p.5)
    - **Metadata:** "Data about data." It provides a complete definition or description of the database structure and other constraints on the stored data. It includes information like file structure, data type, storage format of each data item, and constraints. Metadata transforms raw data into information by providing context. In a database environment, metadata is typically stored in a data dictionary. (p.6)
      (Table 1.1, p.6)

Table 1.1 Some metadata for a manufacturing plant

Record Type	Data Element	Data Type	Size	Source	Role	Domain
PLANT	Pl_name	Alphabetic	30	Stored	Non-key	
PLANT	Pl_number	Numeric	2	Stored	Key	Integer values from 10 to 20
PLANT	Budget	Numeric	7	Stored	Non-key	
PLANT	Building	Alphabetic	20	Stored	Non-key	
PLANT	No_of_employees	Numeric	4	Derived	Non-key	

      - *The table above shows metadata for a manufacturing plant, detailing elements like 'Record Type' (e.g., PLANT), 'Data Element' (e.g., Pl_name, Pl_number), its 'Data Type' (e.g., Alphabetic, Numeric), 'Size', 'Source' (Stored/Derived), 'Role' (Key/Non-key), and 'Domain' (e.g., integer values from 10 to 20 for Pl_number).*

- **Data Management:** The discipline that focuses on the proper acquisition, storage, maintenance, and retrieval of data. It involves four primary actions: (a) data creation, (b) data retrieval, (c) data modification/updating, and (d) data deletion. To support these actions, data must be accessed and organized. (p.6-7)
  - **Access Methods:**
    - **Sequential Access:** To reach the *n*th record, the previous *n-1* records must be traversed. Essential for batch processing like payroll.
    - **Direct Access:** The *n*th record can be accessed without traversing previous records. Useful for ad-hoc querying.
  - **Organization Methods for Access:**
    - **Sequential Organization:** Records are stored in a specific order based on a unique identifier. Supports sequential access.
    - **Serial Organization (Heap File):** Records are unordered. Cannot provide efficient sequential access.
    - **Random Organization (Hashing):** A unique identifier is processed by a hashing algorithm to compute the record's storage location. Supports direct access.
    - **Indexed Organization:** An external index (like a book's index) is used to identify the physical location of records. Supports direct access.
- **Limitations of File-Processing Systems [PYQ - Implied in understanding DB advantages]** (p.7-9)
  Traditional file-processing systems store data in separate, isolated files, leading to several drawbacks:
  - **Lack of data integrity:** Data integrity ensures values are correct, consistent, complete, and current. Duplication across isolated files leads to inconsistencies (e.g., an address updated in one file but not another), making it hard to identify the correct version. This causes maintenance inefficiencies and data integrity problems.
  - **Lack of standards:** Difficult to enforce consistent naming conventions, data formats, access protocols, and security measures across different applications and files. This can lead to unauthorized access or accidental/intentional data damage.
  - **Lack of flexibility/maintainability:** New information requirements or changes to existing ones (e.g., new reports, queries) often require significant programmer intervention to write or modify code. Programs become inefficient, poorly documented, and hard to maintain.
  - **Data Redundancy:** The same data is stored in multiple files, leading to wasted storage space and the integrity issues mentioned above.
  - **Limited Data Sharing:** Due to data isolation, sharing data across applications is complex and often requires custom programs to integrate data from different files.
  - **Root Causes of these Limitations:**

        - **Lack of data integration:** Data is physically separated in different files, often owned by different departments or applications, leading to limited data sharing and difficulty in providing a unified view of organizational data.
        - **Lack of program-data independence:** The definition of data (metadata, file structure) is embedded within each application program that uses the data. If the data structure changes (e.g., expanding a zip code field), every program accessing that data must be identified, modified, recompiled, and retested. This makes systems rigid and expensive to maintain.
    - **ANSI/SPARC Three-Schema Architecture [PYQ - Important for data independence]** (p.9-11) Proposed by the American National Standards Institute (ANSI) Standards Planning and Requirements Committee (SPARC) to address the limitations of file-processing systems, particularly program-data independence. It provides a framework for organizing data in a

database from different perspectives.

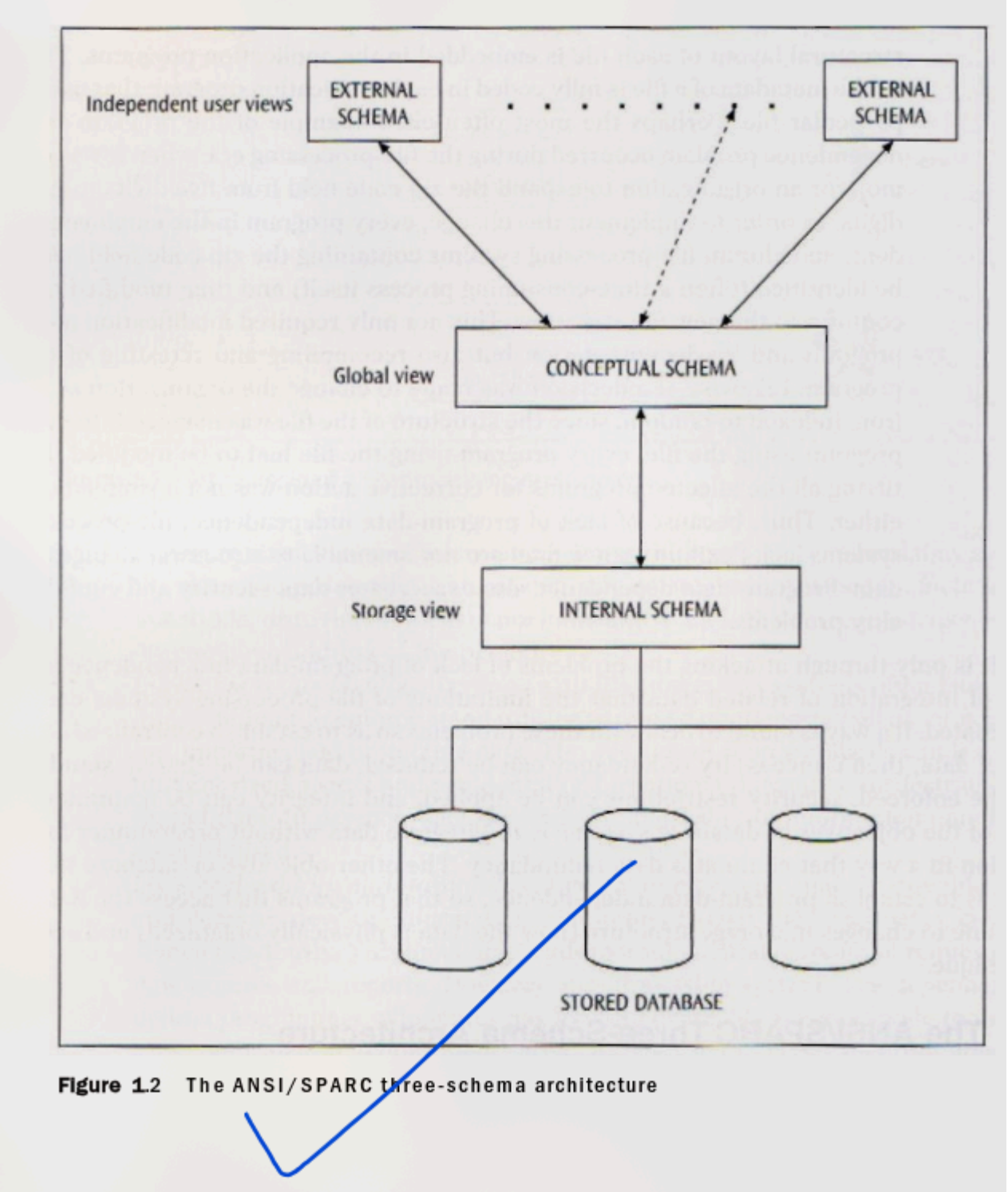


Figure 1.2 The ANSI/SPARC three-schema architecture

- **Purpose:** The primary goal is to separate user views of data from the physical storage of data, thereby achieving program-data independence and insulating applications from changes in physical storage or access strategies.
- **Levels (Schemas):**
  - **External Schema (User Views/Subschemas):** Represents the database as seen by individual end-users or application programs. It describes the part of the database that is relevant to a particular user, hiding the rest. There can be multiple external schemas for a single database. This level is technology-independent.
  - **Conceptual Schema (Global View/Community View):** Provides a comprehensive, logical description of the entire database for a community of users. It defines all entities,

attributes, relationships, and constraints, independent of any specific storage considerations or user views. It acts as the intermediary between external and internal schemas and is the nucleus of the architecture. This level is technology-independent.

- **Internal Schema (Physical View/Storage View):** Describes the physical storage structure of the database on storage devices. It deals with how the data is actually stored (e.g., file organization, indexes, data compression, data encryption) and the access mechanisms used. This level is technology-dependent and concerned with storage efficiency and performance.

- **Mappings:** The DBMS is responsible for mapping requests and data between these schema levels.
  - External/Conceptual mapping: Transforms requests from an external schema to the conceptual schema.
  - Conceptual/Internal mapping: Transforms requests from the conceptual schema to the internal schema for actual data access.
- **Data Independence:** This architecture facilitates two types of data independence:
  - **Logical Data Independence:** The ability to change the conceptual schema without having to change the external schemas or application programs. For example, adding a new entity or attribute to the conceptual schema should not affect existing users whose views do not include these new elements.
  - **Physical Data Independence:** The ability to change the internal schema without having to change the conceptual schema (and consequently, the external schemas). For example, changing file organization or adding an index to improve performance should not require modifications to the conceptual or external views. This is a key advantage over file systems.
- *In contrast, file-processing systems can be viewed as a two-schema architecture where the programmer's view (external) is directly tied to the physical structure (internal), lacking a conceptual intermediary and thus lacking data independence.*

- **Database System vs. DBMS [PYQ Q1a]**
  - **Database System:** (p.12-14) A broader term encompassing the database itself, the DBMS software, application programs, hardware, and personnel (like DBAs and users). It is a self-describing collection of interrelated data, meaning the metadata (description of the data) is stored within the database itself, not in application programs.
    - **Properties:** Data has implicit meaning, which becomes explicit through metadata; logically related records; embedded pointers/indexes for access.
    - **Types:**
      - **Single-user (Desktop):** Supports one user at a time (e.g., MS Access on a PC).
      - **Multi-user:** Supports multiple users concurrently.
        - **Workgroup:** Supports a small number of users (e.g., <50) within a department.

        - **Enterprise:** Supports many users (hundreds/thousands) across an entire organization, often geographically dispersed. Large organizations might have multiple enterprise databases.
    - **Distributed Database (DDB):** A collection of multiple logically interrelated databases physically distributed over a computer network. A DDBMS manages it transparently.
    - **Data Warehouse:** Optimized for decision support and analytical queries rather than transaction processing. Subject-oriented, integrated, nonvolatile, and time-variant.
- **Database Management System (DBMS):** (p.15-16) A software package that enables users to define, create, maintain, and control access to the database. It acts as an interface between the users/applications and the physical database files.

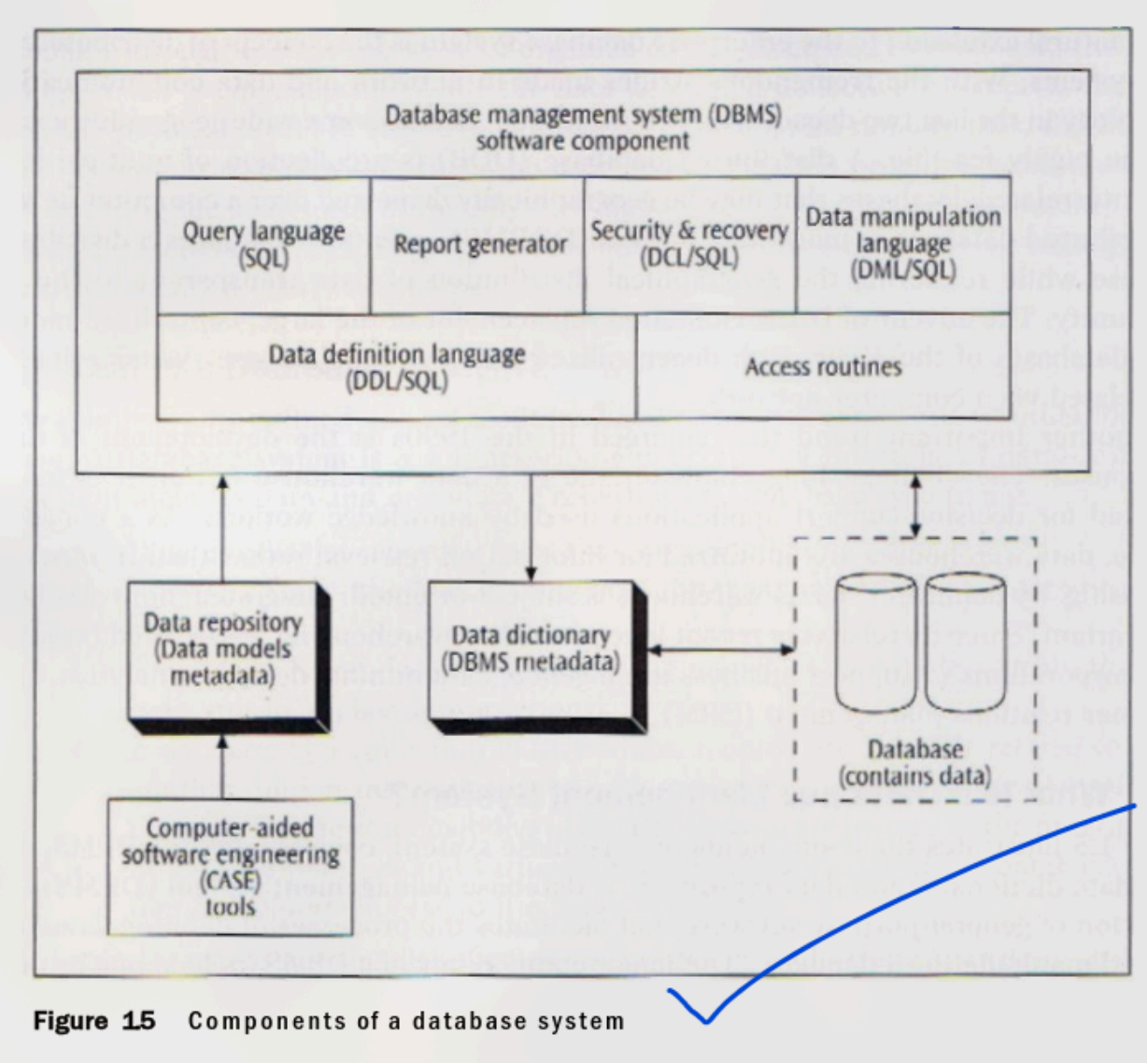


Figure 1.5 Components of a database system

    - *The image shows users and application programs interacting with the DBMS, which in turn manages the database containing data items with minimal/controlled redundancy. The DBMS also provides user-friendly interrogation tools.*
    - **Components [PYQ Q1a]:** *(Fig 1.5, p.16)*
        - **Query Processor/Languages (e.g., SQL):** Allows users to retrieve and manipulate data. SQL (Structured Query Language) is the standard.
        - **Report Generators:** Tools to create formatted reports from database data.

        - **Utilities for Security, Integrity, Backup & Recovery:**
            - Security: Prevents unauthorized access.
            - Integrity: Enforces data accuracy and consistency rules.
            - Backup & Recovery: Protects data from loss due to failures.
        - **Data Definition Language (DDL):** Used to define the database schema (e.g., `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`). Specifies structures and constraints.
        - **Data Manipulation Language (DML):** Used for accessing and manipulating data (retrieval, insertion, deletion, modification). SQL includes DML commands like `SELECT`, `INSERT`, `UPDATE`, `DELETE`.
            - Procedural DML: Specifies *how* to get data.
            - Non-procedural DML (like SQL): Specifies *what* data to get.
        - **Data Control Language (DCL):** Used to control access to data (e.g., `GRANT` permissions, `REVOKE` permissions).
        - **Data Dictionary/Repository (System Catalog):** A system database that stores metadata – definitions of data items, structures, relationships, constraints, authorizations, usage statistics. The DBMS consults the data dictionary to understand data structures, relieving developers from embedding this in applications. Changes to physical structure are automatically recorded here.
        - **Application Program Interfaces (APIs):** Allow application programs (written in Java, C++, etc.) to interact with the DBMS.
        - **Access Routines:** Handle database access at runtime, interacting with the OS file manager.
- **Advantages of Database Systems** (p.17-18)
  Compared to file-processing systems, database systems offer significant advantages:
    - **Controlled Redundancy:** While not eliminating all redundancy (some is needed for relationships), DBMSs allow control over it, minimizing inconsistencies.
    - **Improved Data Integrity and Consistency:** Business rules and constraints can be enforced by the DBMS, ensuring data is accurate, consistent, and reliable.
    - **Improved Data Sharing and Accessibility:** Centralized (or controlled distributed) data allows easier sharing among authorized users and applications.
    - **Enforcement of Standards:** The DBA can enforce data standards for naming, formats, documentation, and access.
    - **Improved Data Security:** The DBMS provides mechanisms for access control, authorization, and encryption.
    - **Program-Data Independence:** Applications are insulated from changes in data storage and structure (as per ANSI/SPARC architecture).
    - **Increased Productivity of Application Development:** Developers can focus on application logic rather than low-level data management.

- **Reduced Maintenance:** Changes are easier to manage due to data independence and centralized control.
- **Provision of Ad-hoc Querying:** Users can often directly query the database using tools like SQL, without needing custom programs.
- **Backup and Recovery Services:** DBMS provides mechanisms for recovering from hardware/software failures.
- **Concurrency Control:** Manages simultaneous access by multiple users to prevent interference and maintain data consistency.

- **Database Design Life Cycle [PYQ Q2b]** (p.20-22)
  A structured process for designing and implementing a database.
  (Fig 1.7, p.19 & p.25)

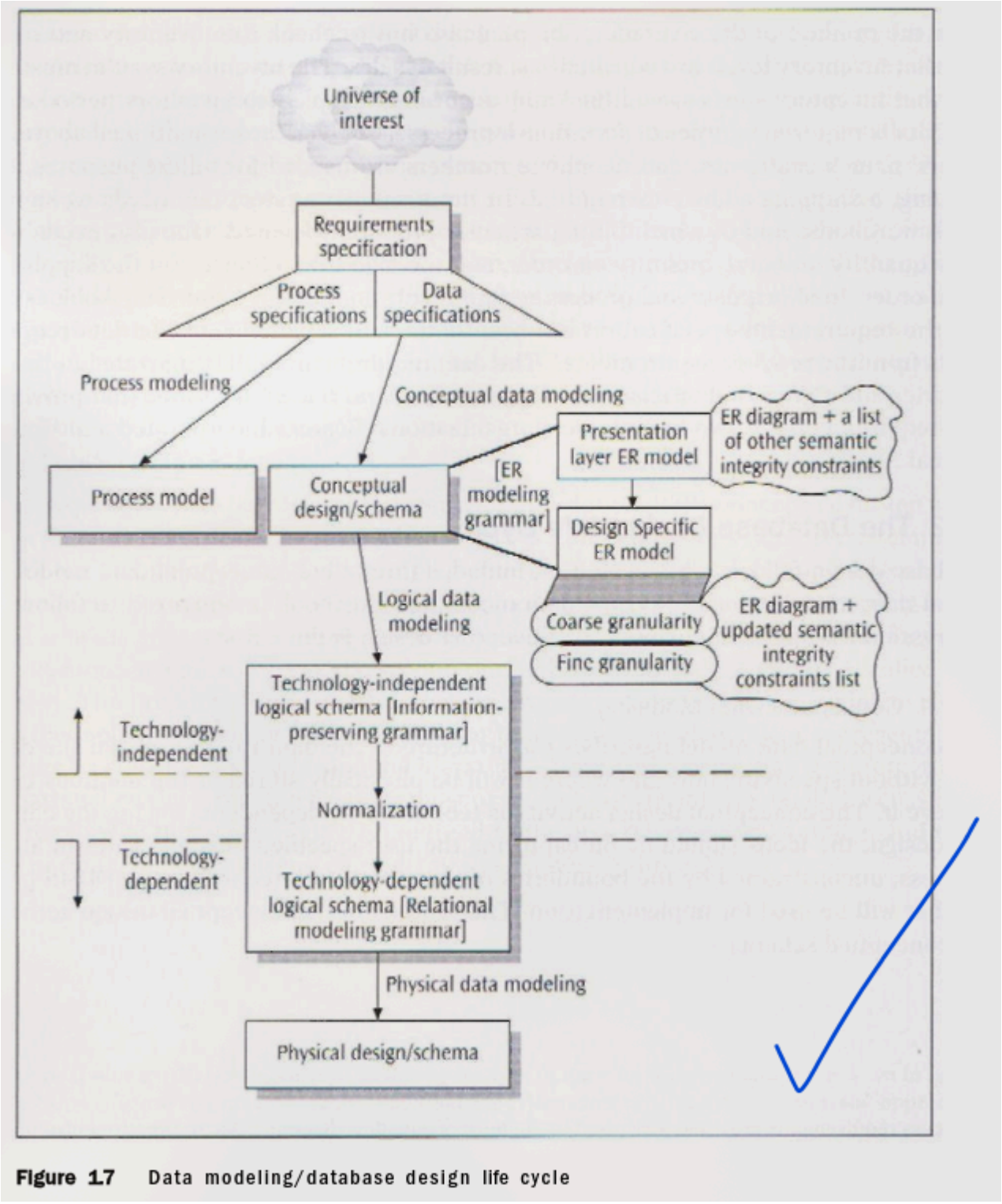


Figure 1.7 Data modeling/database design life cycle

- *The diagram illustrates the progression from requirements specification (universe of interest) through conceptual data modeling (ER modeling grammar leading to Presentation Layer and Design-Specific ER models), logical data modeling (technology-independent to technology-dependent schema, including normalization), and finally physical data modeling/schema.*
- **1. Requirements Specification (and Analysis):** The initial and most critical phase.
  - Involves understanding the "universe of interest" for the database.
  - Systems analysts interview prospective users, review existing documents, forms, and systems.
  - Goal: To identify the objectives of the database system, the data to be stored, the operations (processes) to be performed, and the business rules that govern the data and processes.
  - Output: A detailed set of data specifications (user-specified restrictions, business rules) and process specifications. Business rules are crucial as they often translate into integrity constraints.
- **2. Conceptual Data Modeling (Technology-Independent):** Focuses on *what* data is needed and *how* it is related, without considering *how* or *where* it will be physically stored or the specific DBMS to be used.
  - Describes the structure of the data at a high level of abstraction.
  - The primary aim is to capture all user-specified business rules and data requirements accurately and completely.
  - **Tools:** Entity-Relationship (ER) modeling, Enhanced ER (EER) modeling are common techniques.
  - **Layers within Conceptual Modeling (as per textbook):**
    - **Presentation Layer ER Model:** Intended for user-analyst communication. Develops a script (ER diagram) and a set of semantic integrity constraints not captured in the diagram. (Ch 3)
    - **Design-Specific ER Model:** Translates the Presentation Layer model into a more technically detailed format for database designers. Progresses through coarse and fine granularity. (Ch 3)
  - **Validation:** A crucial step to verify that the conceptual design indeed answers all questions and captures all rules from the requirements specification. (End of Ch 5)
  - Product: A conceptual schema (e.g., a validated ER/EER diagram and associated textual constraints).
- **3. Logical Data Modeling (Target Data Model Dependent):** Transforms the technology-independent conceptual schema into a schema that is compatible with the chosen class of DBMS (e.g., relational, object-oriented, network, hierarchical).
  - For relational DBMS, this involves mapping ER/EER constructs to relations (tables).
  - **Normalization:** A key activity in this phase for relational models, aiming to reduce redundancy and improve data integrity by decomposing relations into well-structured forms

(1NF, 2NF, 3NF, BCNF, etc.). (Ch 6-9)
    - The logical schema evolves from a technology-independent representation to a technology-dependent one.
    - Product: A logical schema (e.g., a set of normalized relations with primary and foreign keys defined).
- **4. Physical Data Modeling (DBMS-Specific):** This is the final stage before implementation.
    - Specifies how the logical schema will be physically realized on storage using the features of a specific DBMS product.
    - Involves defining internal storage structures (e.g., file organizations, record layouts), access paths (e.g., indexes, hashing schemes), data types specific to the DBMS, and security measures.
    - Performance considerations (storage efficiency, query speed) are paramount.
    - Product: A physical design/schema, often expressed in the DDL of the chosen DBMS.

2. Conceptual Data Modelling

Conceptual modeling is the activity of formally describing some aspects of the physical and social world around us for the purposes of understanding and communication. In database design, it refers to creating a high-level, abstract representation of the data requirements of an application domain, independent of implementation details.

- **ER Modeling [PYQ Q3b, Q4a]**
  The Entity-Relationship (ER) model, originally proposed by Peter Chen (1976), is a widely used conceptual data modeling tool. It aims to capture the overall data semantics of an application concisely, using graphical representations.
    - **Framework:** (p.26) The ER modeling process involves:
        - **Context:** The application domain or universe of interest being modeled.
        - **Grammar:** A set of constructs (entity types, attributes, relationships) and rules for how they can be combined.
        - **Method:** A procedure describing how to use the grammar to create a model (script).
        - **Script:** The output, typically an ER diagram and associated textual constraints.
    - **ER Modeling Primitives:** (p.26-27) These are the fundamental terms correlating the real world to the conceptual world.
      (Table 2.1, p.26-27)

Table 2.1 Equivalence between real world primitives and conceptual primitives

Real World Primitive	Conceptual Primitive
Object {type}	Entity(ies) {type}
Object (occurrence)	Entity(ies) {instance}
Property	Attribute
Fact	Value

Table 2.1 Equivalence between real world primitives and conceptual primitives (continued)

Real World Primitive	Conceptual Primitive
Property value set	Domain
Association	Relationship
Object class	Entity class

- *The tables illustrate the mapping: Real World Primitive -> Conceptual Primitive. For example, 'Object [type]' (e.g., a specific student) maps to 'Entity(ies) [type]', 'Property' (e.g., student number) maps to 'Attribute', 'Association' (e.g., students enroll in courses) maps to 'Relationship'. 'Object class' (generalization of object types) maps to 'Entity class'.*
- **Entity Type:** A class of real-world objects that have common properties and an independent existence (e.g., `STUDENT`, `COURSE`, `DEPARTMENT`). It's a "thing" of significance about which information needs to be stored. Represented by a rectangle in an ERD.
- **Entity Instance (Entity):** A specific, uniquely identifiable occurrence of an entity type (e.g., the student 'John Smith', the course 'CS101').
- **Entity Set:** The collection of all entity instances of a particular entity type at a point in time.
- **Attribute:** A property or characteristic that describes an entity type (e.g., `StudentName`, `CourseID`, `DepartmentName`). Attributes refine the understanding of an entity type. Represented by an oval connected to the entity type rectangle.
  - **Types of Attributes:** (p.28-30) Attributes can be classified based on several characteristics:

(Table 2.2, p.28; Fig 2.1, p.29; Fig 2.2, p.30)

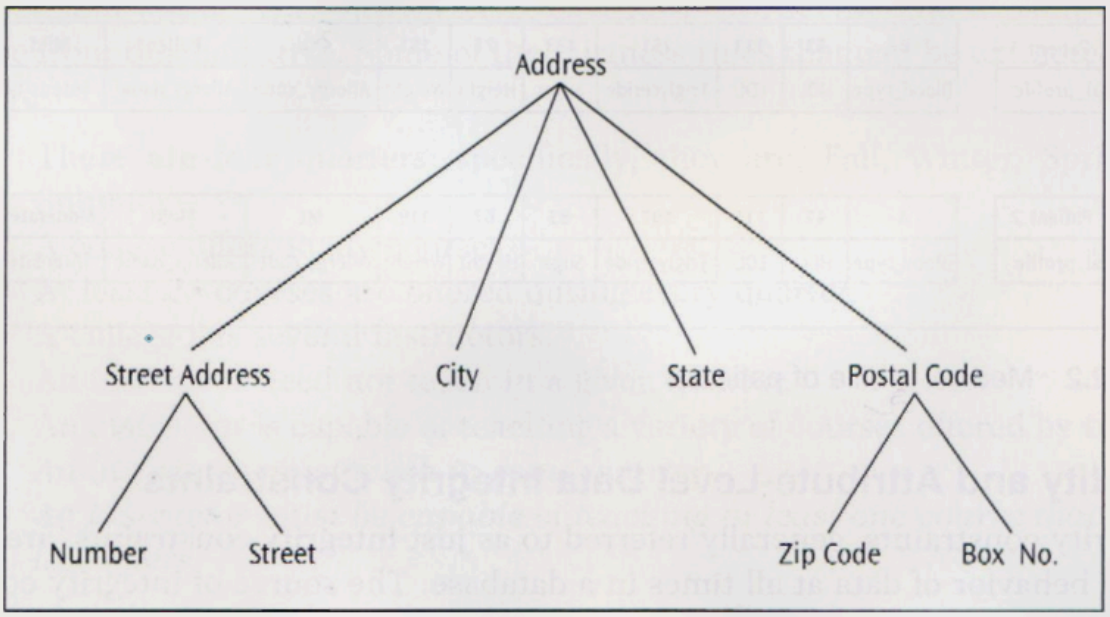


Figure 2.1 An example of a composite attribute hierarchy

- *Table 2.2 lists characteristics: Name, Type (numeric, alphabetic, etc.), Classification (atomic/composite), Category (single/multi-valued), Source (stored/derived), Domain, Value (optional/mandatory), Role (key/non-key).*

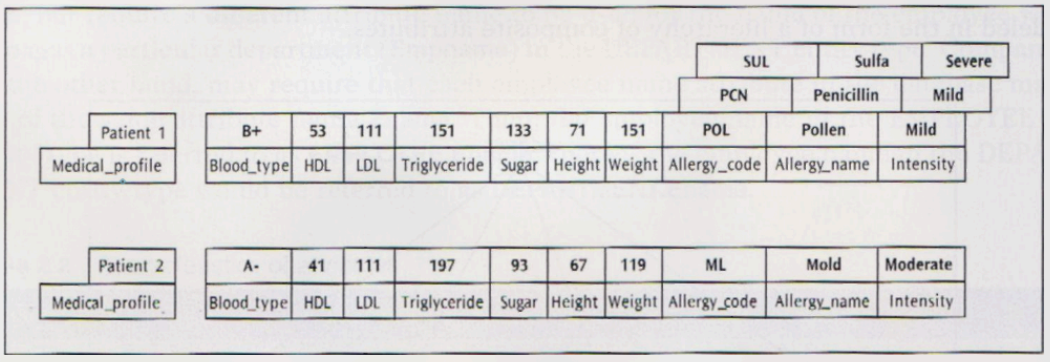


Figure 2.2 Medical profile of patients

- *Fig 2.1 shows a composite attribute 'Address' broken down into 'Street Address', 'City', 'State', 'Postal Code'. 'Street Address' itself can be composite.*
- *Fig 2.2 shows 'Medical_profile' as a complex attribute with nested composite ('Blood') and multi-valued ('Allergy') attributes.*
- **Simple (Atomic):** An attribute that cannot be meaningfully subdivided into smaller components (e.g., `Age`, `Salary`, `Gender`).
- **Composite:** An attribute that can be meaningfully subdivided into smaller, independent atomic attributes (e.g., `Address` can be composed of `Street`, `City`, `State`, `ZipCode`; `Name` can be `FirstName`, `MiddleInitial`, `LastName`). Represented by an oval connected to ovals for its components.
- **Single-valued:** An attribute that holds only one value for each entity instance (e.g., a person's `DateOfBirth`). This is the most common type.

        - **Multi-valued:** An attribute that can hold multiple values for a single entity instance (e.g., `Skills` of an employee, `PhoneNumbers` of a person, `Hobbies` of a student). Represented by a double oval. *These are generally problematic for direct implementation in relational models and often lead to new entity types during logical design.*
        - **Stored:** An attribute whose value is explicitly stored in the database (e.g., `BirthDate`).
        - **Derived:** An attribute whose value can be calculated or derived from the values of other stored attributes (e.g., `Age` can be derived from `BirthDate` and the current date; `NumberOfEmployees` in a department can be derived by counting employees). Represented by a dotted oval. Derived attributes are typically not stored to avoid redundancy and potential update anomalies, but their derivation rule must be documented.
        - **Mandatory:** An attribute that must have a value for every entity instance; it cannot be null (e.g., `EmployeeID`). Indicated by a dark (filled) circle in some notations.
        - **Optional:** An attribute that may not have a value for some entity instances; it can be null (e.g., `MiddleInitial`, `ApartmentNumber`). Indicated by an empty (unfilled) circle.
        - **Complex:** Attributes that are nested combinations of composite and/or multi-valued attributes (e.g., a `MedicalProfile` that includes a multi-valued `Allergy` attribute, where each `Allergy` has `Code`, `Name`, and `Intensity`).
    - **Domain:** The set of all possible legal values that an attribute can take. It defines the data type (e.g., string, integer, date), length, and any specific value constraints (e.g., `Gender` can be 'Male' or 'Female'; `Salary` must be between $10,000 and $2,000,000).
        - **Explicit Domain:** Values are explicitly listed (e.g., `Semester` = {Fall, Winter, Spring, Summer}).
        - **Implicit Domain:** Values are defined by a rule or range (e.g., `Age` > 0).
- **Unique Identifiers (Keys):** [Related to PYQ Q1b] (p.32-33) An attribute or a set of attributes whose values are unique for each entity instance within an entity set, thus serving to distinguish one entity instance from another. In ER diagrams, attributes forming a unique identifier are typically underlined.
    - **Candidate Key:** A minimal set of attributes that uniquely identifies an entity instance. "Minimal" means no proper subset of the candidate key attributes can uniquely identify an entity instance. An entity type can have more than one candidate key. (e.g., For an `EMPLOYEE` entity, `EmployeeID` could be one candidate key, and perhaps a combination of `FullName` and `DateOfBirth` could be another, if unique).
    - **Primary Key:** One of the candidate keys selected by the database designer to be the principal means of uniquely identifying entity instances within an entity type. Its values cannot be null (entity integrity).
    - **Alternate Key:** Any candidate key that is not chosen as the primary key.

- **Superkey:** Any attribute or set of attributes that uniquely identifies an entity instance. A superkey may contain redundant attributes (i.e., it might not be minimal). Every candidate key is a superkey, but not every superkey is a candidate key.
- **Key Attribute:** An attribute that is part of at least one candidate key.
- **Non-Key Attribute:** An attribute that is not part of any candidate key.

- **Relationship Type:** Represents a meaningful association or interaction among instances of one or more entity types (e.g., `STUDENT` *Enrolls_In* `COURSE`; `EMPLOYEE` *Works_For* `DEPARTMENT`). Represented by a diamond shape connecting the participating entity types.

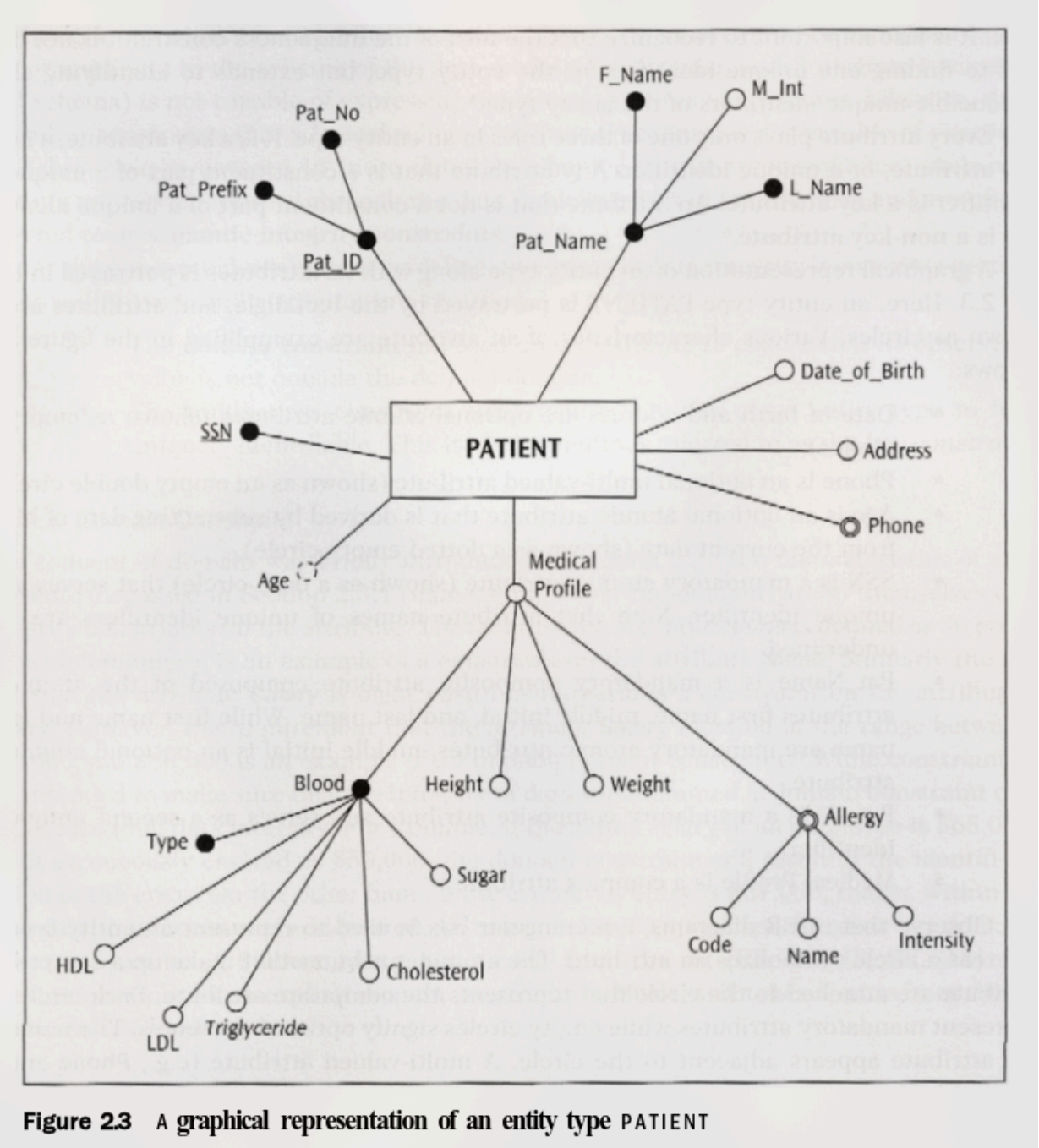


Figure 2.3 A graphical representation of an entity type PATIENT

- *The image shows the graphical representation of an entity type `PATIENT` with various attributes: `Pat_ID` (composite, unique identifier), `SSN` (unique identifier), `Pat_Name` (composite), `Age` (derived), `Address`, `Phone` (multi-valued), `MedicalProfile` (complex), `Date_of_Birth`, `Blood` (composite), `Height`, `Weight`, `Allergy` (composite, multi-valued).*

- **Relationship Instance:** A specific association between specific entity instances from the participating entity types (e.g., student 'John Smith' is enrolled in course 'CS101').
- **Relationship Set:** The collection of all relationship instances of a particular relationship type at a point in time.
- **Degree of a Relationship:** The number of entity types participating in the relationship. (p.33)
  - **Binary (degree 2):** Involves two (possibly distinct) entity types. Most common type. *(Fig 2.4, p.35)*

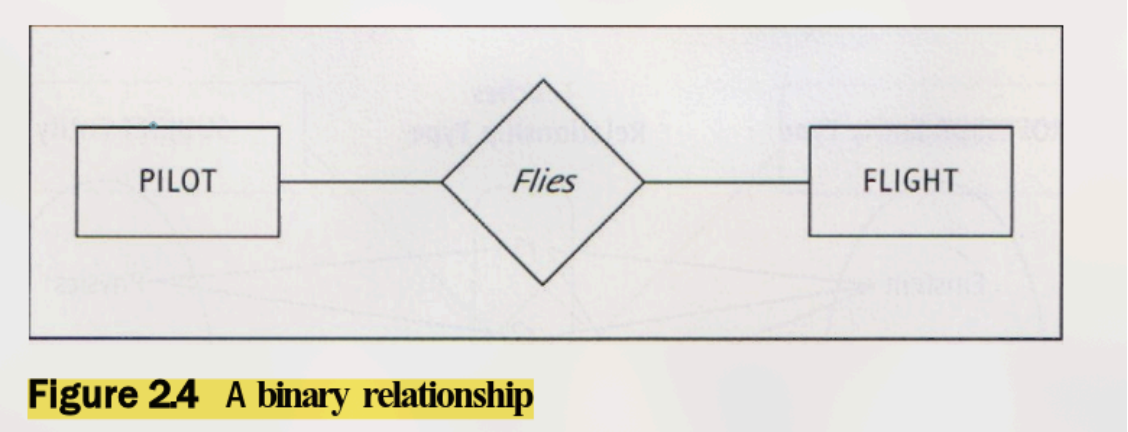


Figure 2.4 A binary relationship

    - *Fig 2.4 shows a binary relationship 'Flies' between `PILOT` and `FLIGHT`.*
  - **Ternary (degree 3):** Involves three entity types. Used when an association naturally links three entities and cannot be accurately represented by multiple binary relationships.
    (Fig 2.5, p.35)

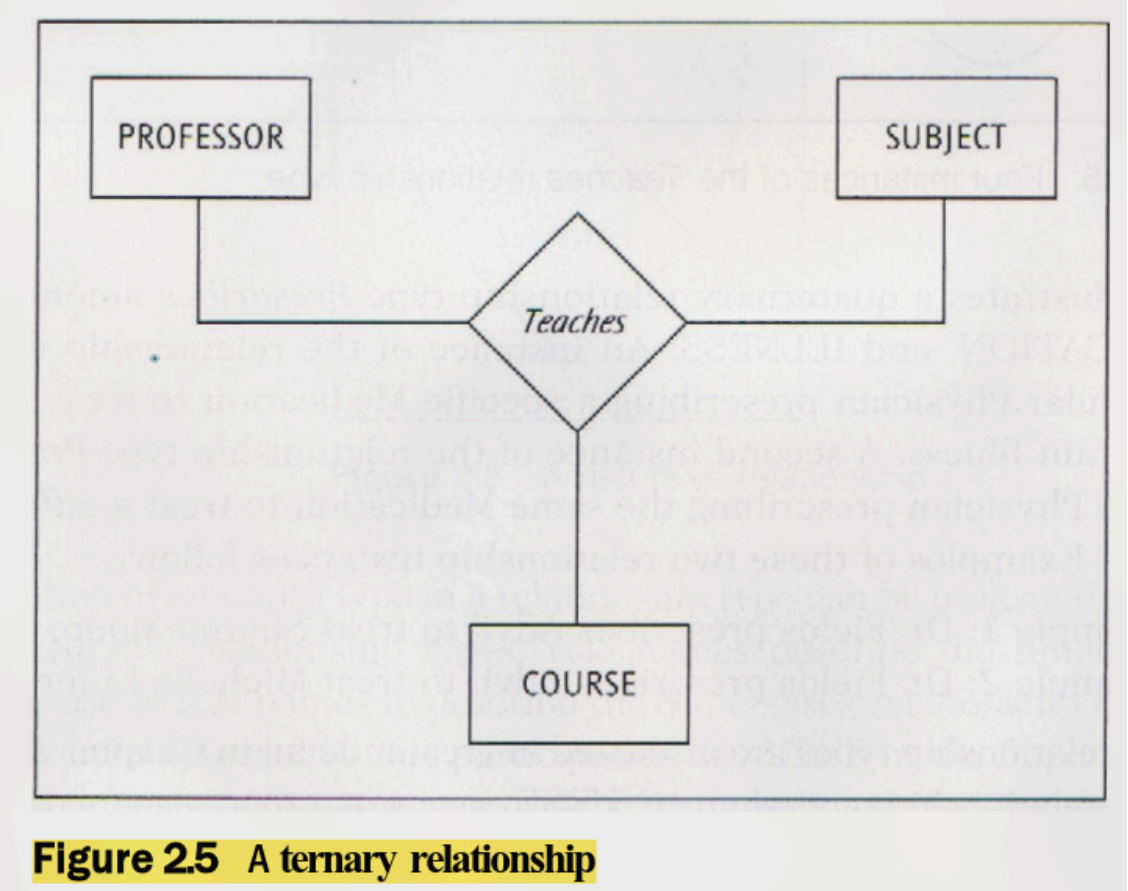


Figure 2.5 A ternary relationship

    - *Fig 2.5 shows a ternary relationship 'Teaches' among `PROFESSOR`, `SUBJECT`, and `COURSE`.*
  - **N-ary (degree n):** Involves 'n' entity types. Relationships of degree higher than three are rare and often can be decomposed. *(Fig 2.7 for Quaternary, p.37)*
  - **Recursive (Unary):** A relationship where instances of the same entity type participate in different roles (e.g., an `EMPLOYEE` *Supervises* another `EMPLOYEE`). *(Fig 2.8, p.37)*

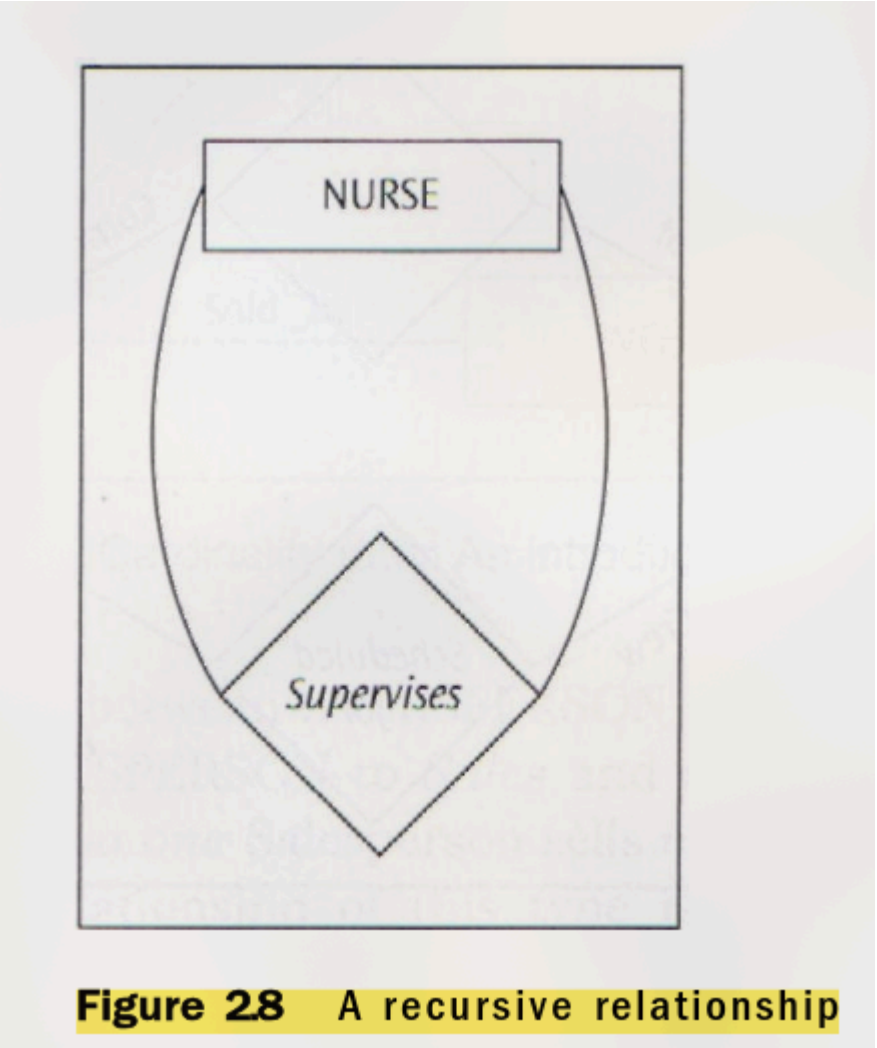


Figure 2.8 A recursive relationship

    - *Fig 2.8 shows a recursive relationship 'Supervises' on the `NURSE` entity type.*
- **Role Names:** Used to clarify the meaning of an entity's participation in a relationship, especially when:
  - An entity type participates multiple times in the same relationship (recursive relationships are a key example, e.g., `Supervisor` and `Supervisee` roles for `EMPLOYEE` in a *Supervises* relationship).
  - Multiple distinct relationship types exist between the same pair of entity types. Role names are written on the line connecting the entity type to the relationship diamond, near the entity type.

(Fig 2.9, p.38; Fig 2.10, p.38)

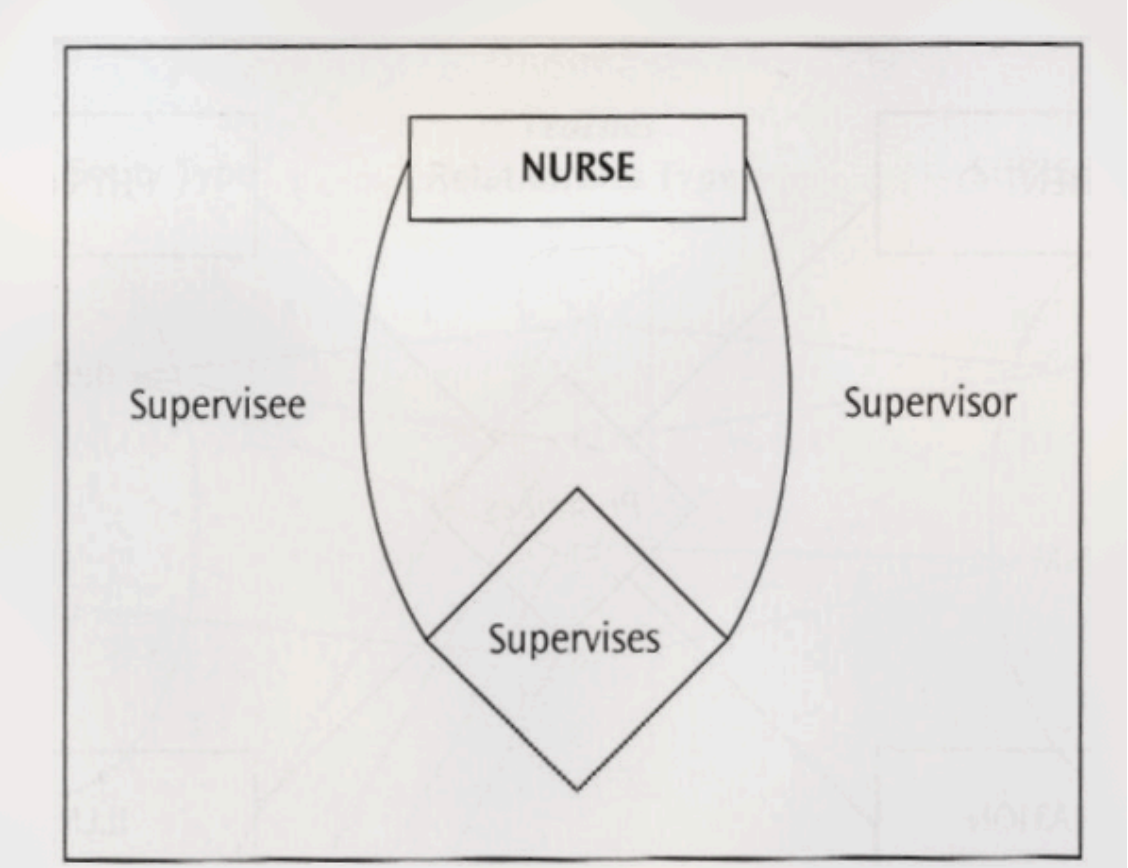


Figure 2.9 Role names in a recursive relationship

- *Fig 2.9 shows role names 'Supervisor' and 'Supervisee' for the recursive 'Supervises' relationship on* `NURSE`.

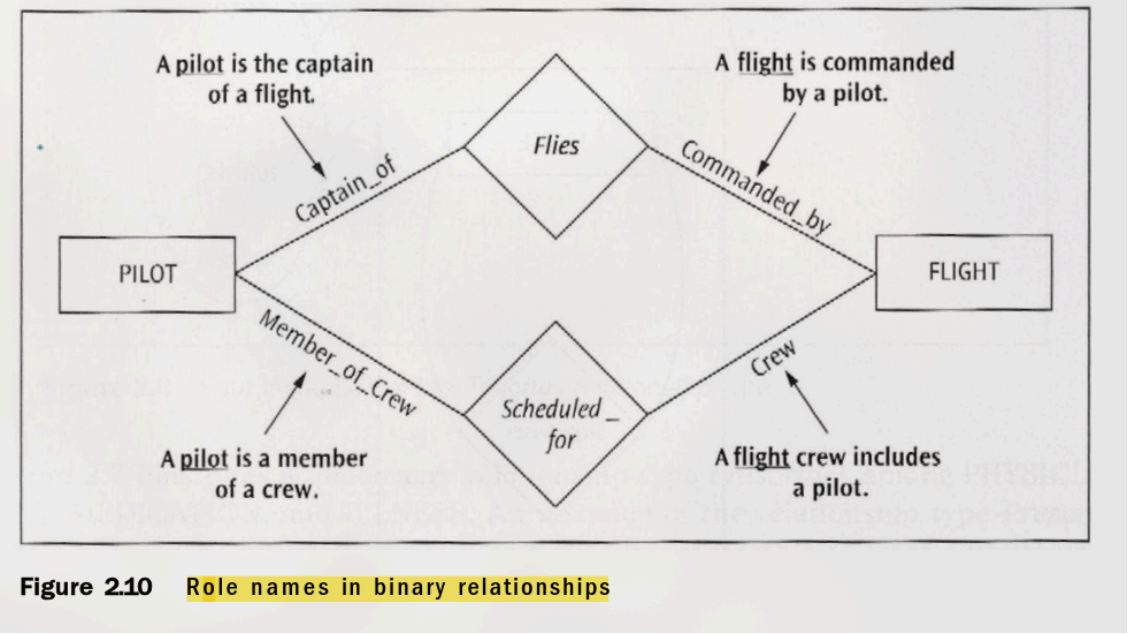


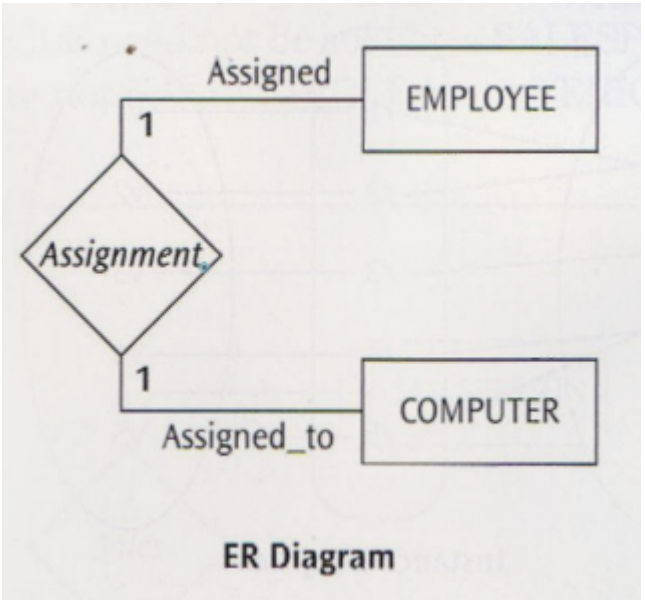
Figure 2.10 Role names in binary relationships

- *Fig 2.10 shows two relationships, 'Flies' (Captain_of_flight, Commanded_by_pilot) and 'Scheduled_for' (Member_of_Crew, Crew), between* `PILOT` *and* `FLIGHT`*, clarified by role names.*

- **Structural Constraints (on Relationship Types):** These define the rules for how entities can participate in relationships. They are crucial for maintaining data integrity.

(p.38-46)

- **Cardinality Ratio (Connectivity or Maximum Cardinality):** Specifies the maximum number of relationship instances in which an entity instance can participate. Expressed as a ratio (e.g., 1:1, 1:N, M:N) for binary relationships.
  - **1:1 (One-to-One):** An entity instance from one entity type can be related to at most one entity instance from the other entity type, and vice-versa (e.g., an `EMPLOYEE` *Manages* at most one `DEPARTMENT`, and a `DEPARTMENT` is *Managed_By* at most one `EMPLOYEE`).
    (Fig 2.14, p.41)



    - *Fig 2.14 shows a 1:1 'Assigned' relationship between `EMPLOYEE` and `COMPUTER`.*
  - **1:N (One-to-Many):** An entity instance from one entity type (the '1' side) can be related to zero, one, or many entity instances from the other entity type (the 'N' side), but an entity instance from the 'N' side can be related to at most one entity instance from the '1' side (e.g., one `DEPARTMENT` *Has* many `EMPLOYEEs`, but each `EMPLOYEE` *Works_In* at most one `DEPARTMENT`).

(Fig 2.11 & 2.13, p.39-40)

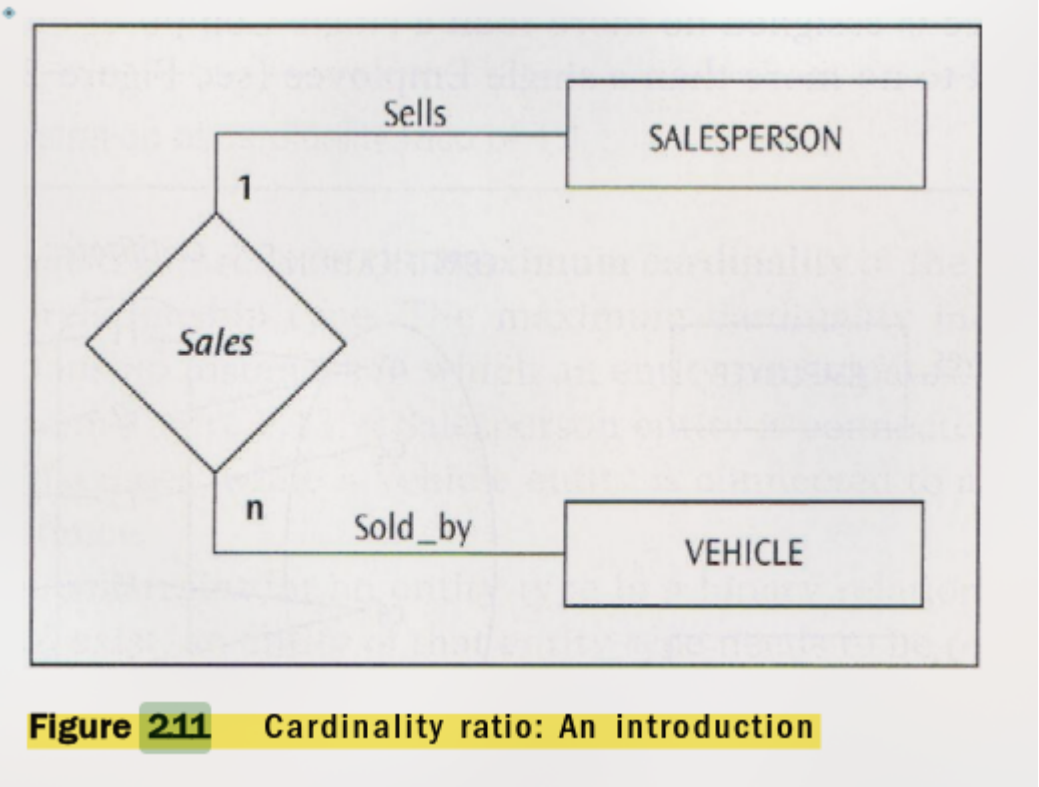
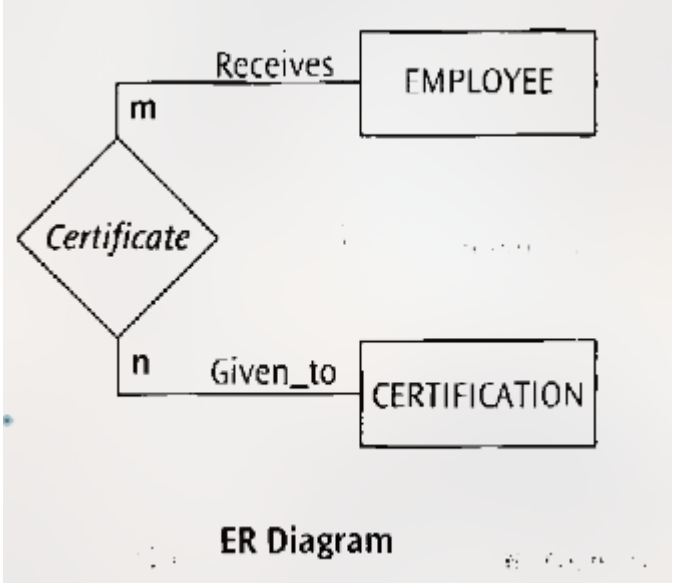


Figure 211 Cardinality ratio: An introduction

- *Fig 2.11 & 2.13 illustrate a 1:N 'Sells' relationship between `SALESPERSON` and `VEHICLE`.*

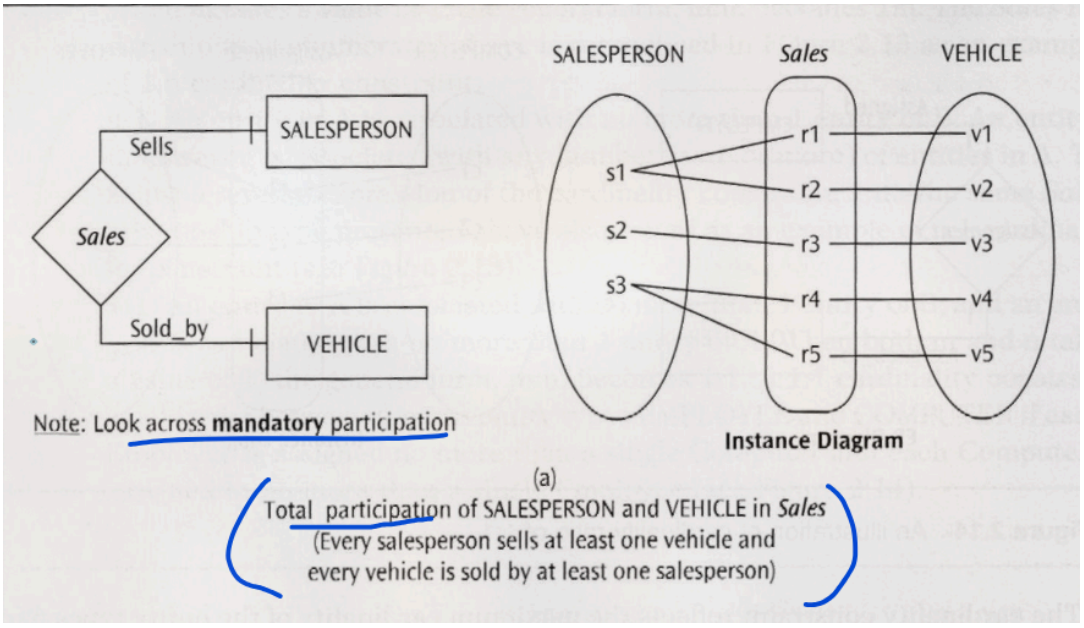
- **M:N (Many-to-Many):** An entity instance from one entity type can be related to zero, one, or many entity instances from the other entity type, and vice-versa (e.g., a `STUDENT` can *Enroll_In* many `COURSEs`, and a `COURSE` can *Have_Enrolled* many `STUDENTs`).
  (Fig 2.12, p.40)



  - *Fig 2.12 illustrates an M:N 'Receives'/'Given_to' relationship between `EMPLOYEE` and `CERTIFICATION`.*

- **Participation Constraint (Minimum Cardinality / Existence Dependency):** Specifies whether the existence of an entity instance depends on its being related to another entity instance through a particular relationship type.

- **Total (Mandatory) Participation:** Every entity instance of a given entity type *must* participate in at least one relationship instance of that relationship type (e.g., if every `EMPLOYEE` must work for a `DEPARTMENT`). Indicated by a bar (|) or a minimum cardinality of 1 (e.g., (1,N)) on the participation line near the entity type. This implies an existence dependency.
- **Partial (Optional) Participation:** An entity instance of a given entity type is *not required* to participate in the relationship type (e.g., an `EMPLOYEE` may or may not *Manage* a `DEPARTMENT`). Indicated by an oval (O) or a minimum cardinality of 0 (e.g., (0,N)).
  (Fig 2.15a, p.42; Fig 2.16a,c,d p.43-44 for total examples)
  (Fig 2.15b, p.42; Fig 2.16b,c,d p.43-44 for partial examples)
  Notations for (min, max) cardinality are often shown on the edge connecting an entity type to a relationship. For example, (0,N) on the `EMPLOYEE` side of a Manages relationship with `DEPARTMENT` means an employee manages zero to many departments.



- *Fig 2.15a shows total participation for both `SALESPERSON` and `VEHICLE` in 'Sells' (every salesperson sells at least one vehicle, every vehicle is sold by at*

least one salesperson).

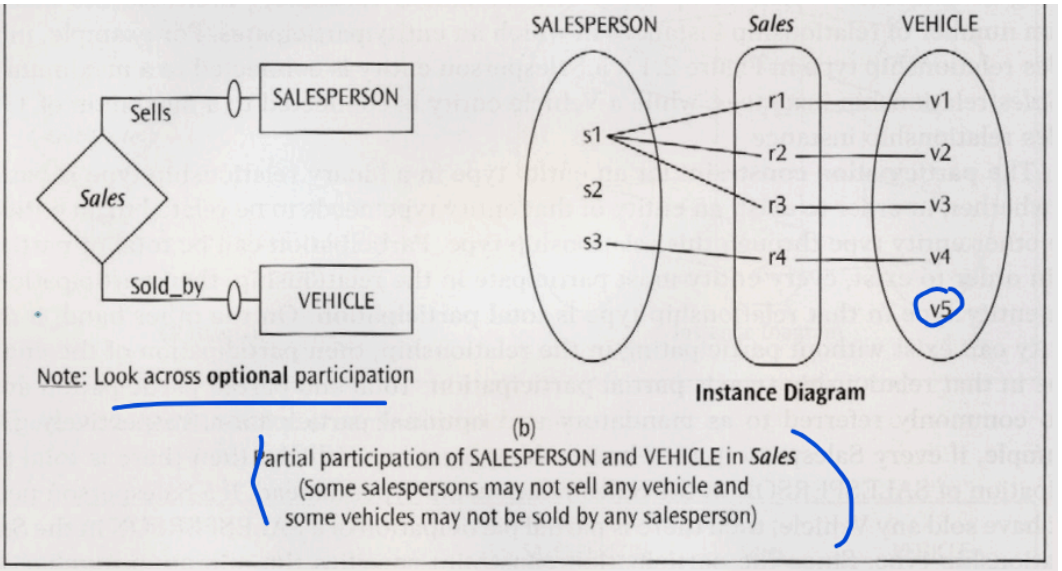


Figure 2.15 **Examples of the participation constraint**

- *Fig 2.16a shows total participation of `VEHICLE` and `SALESPERSON` in the 1:N 'Sells' relationship.*

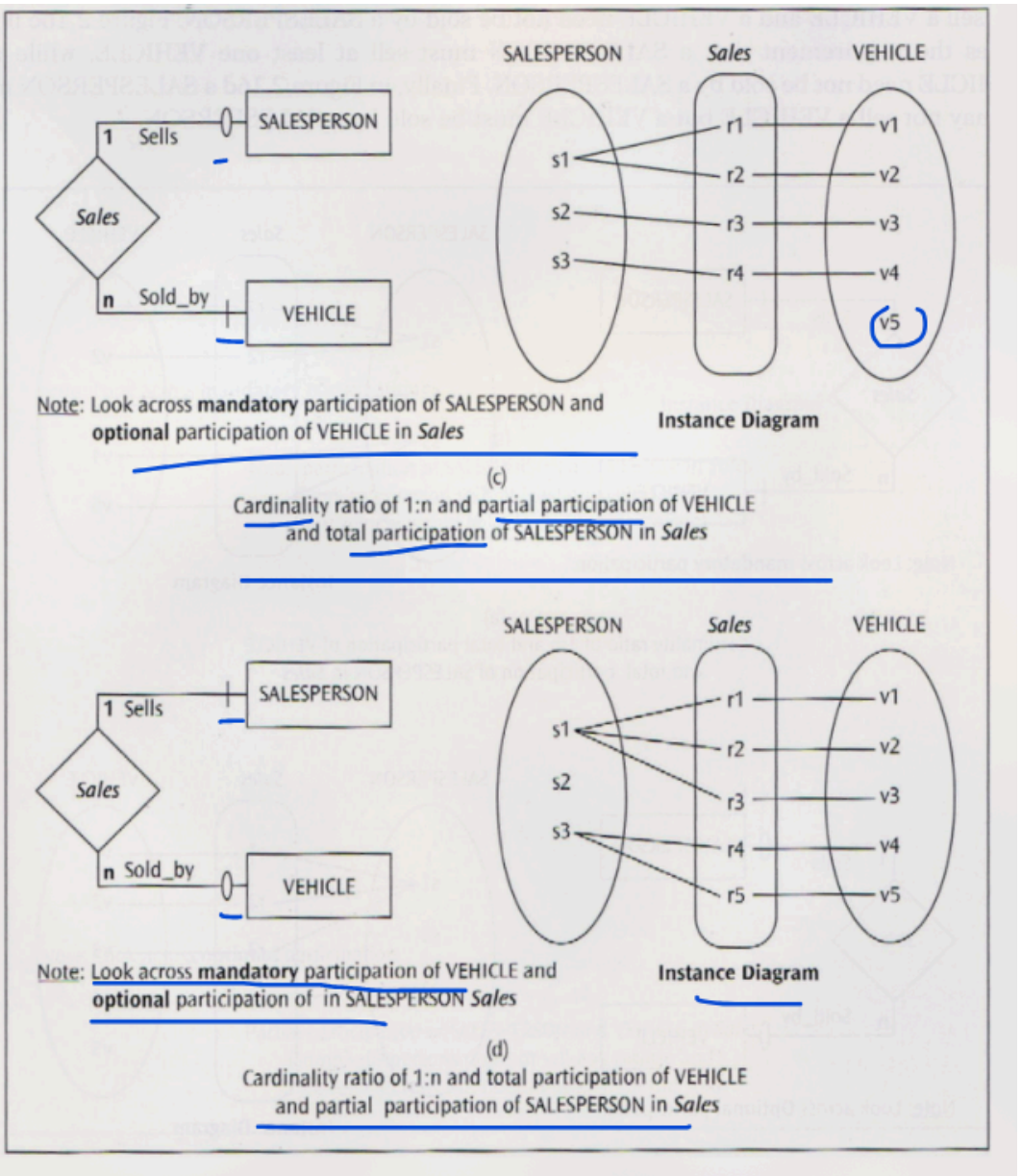


Figure 2.16 **Cardinality ratio and participation constraints for a relationship (continued)**

- *Fig 2.17 illustrates structural constraints for recursive relationships: (a) 1:N with partial participation for both supervisor and supervisee roles; (b) 1:N with total participation for supervisee and partial for supervisor.*

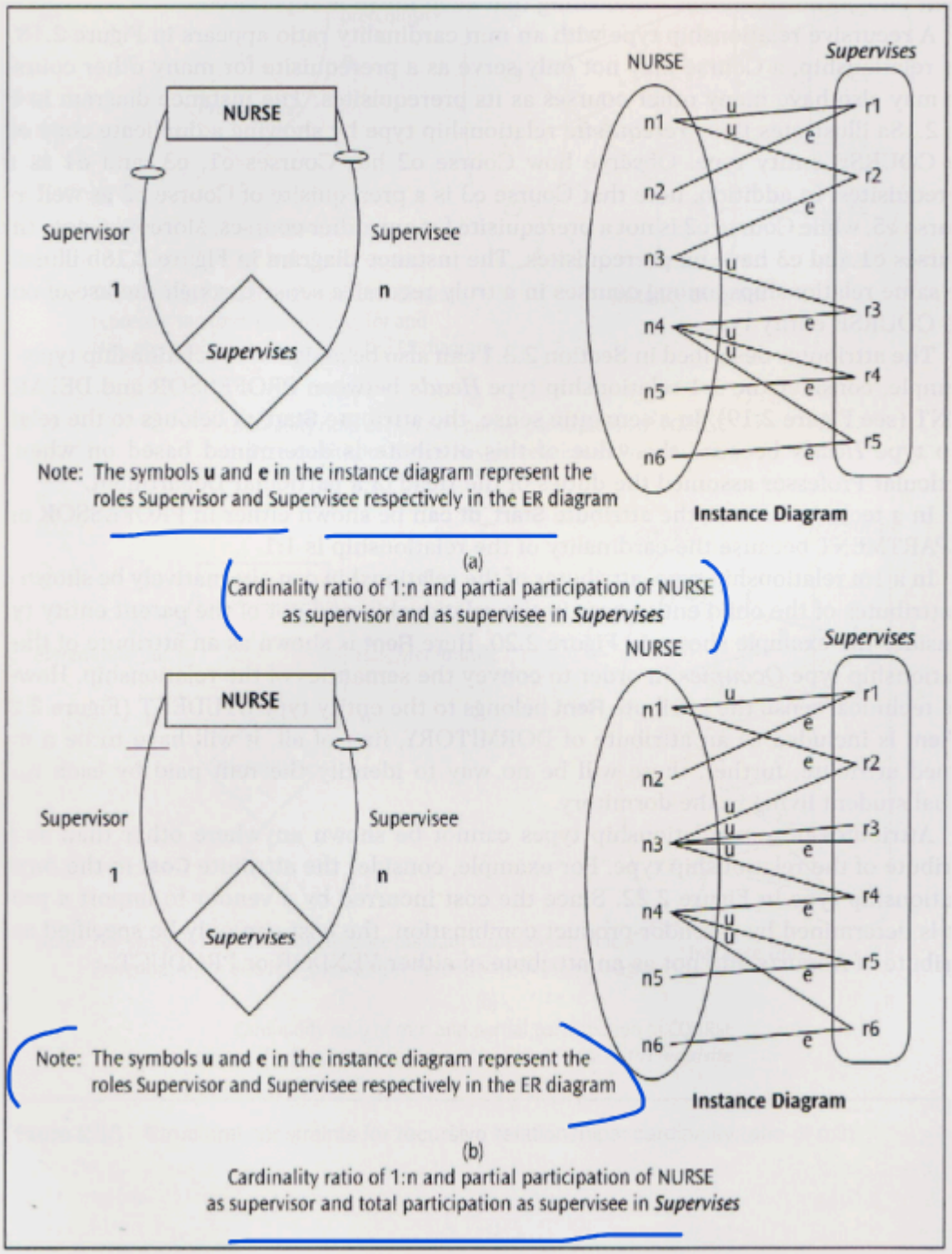


Figure 2.17 Structural constraints for recursive relationships: cardinality ratio of 1:n

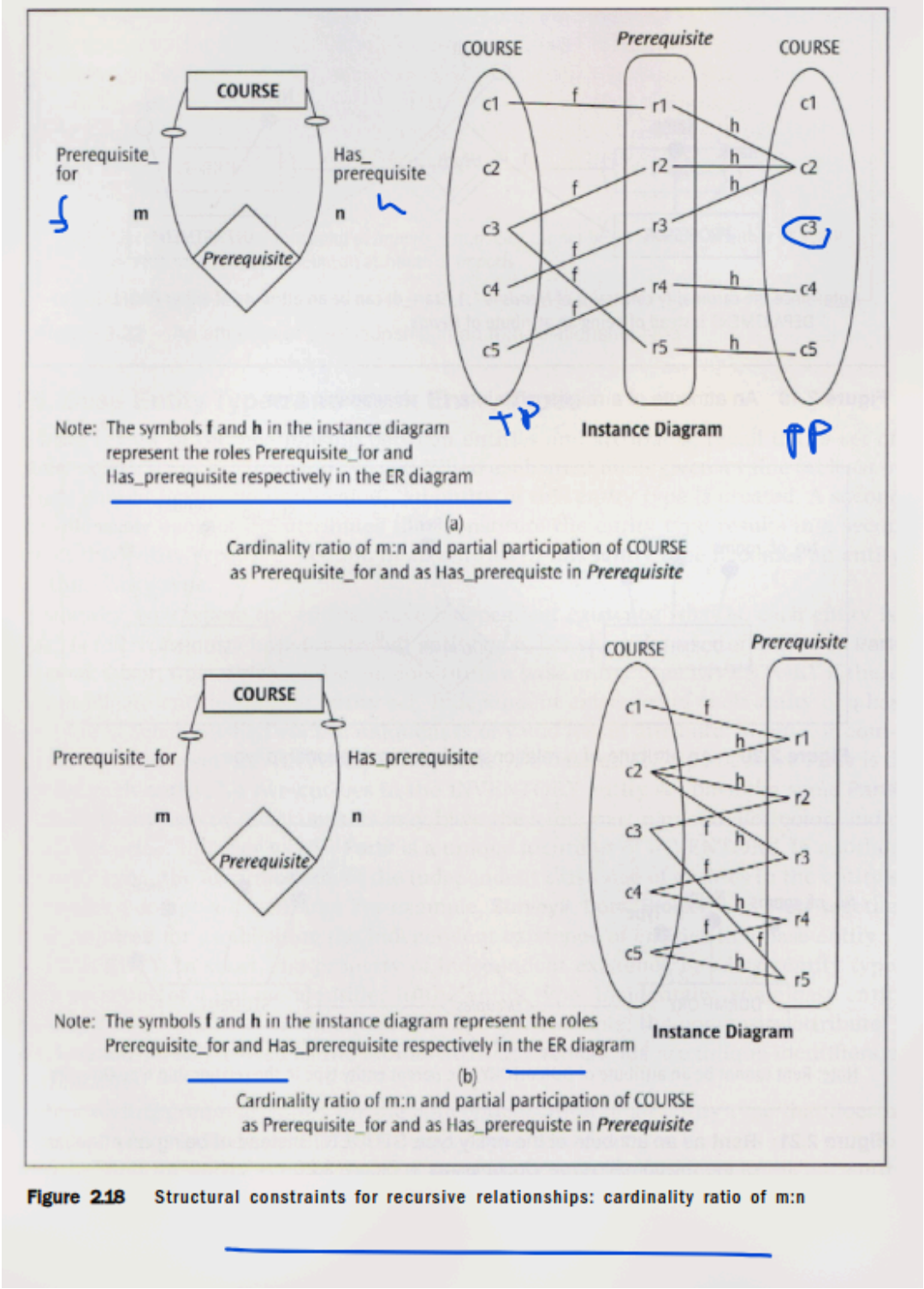
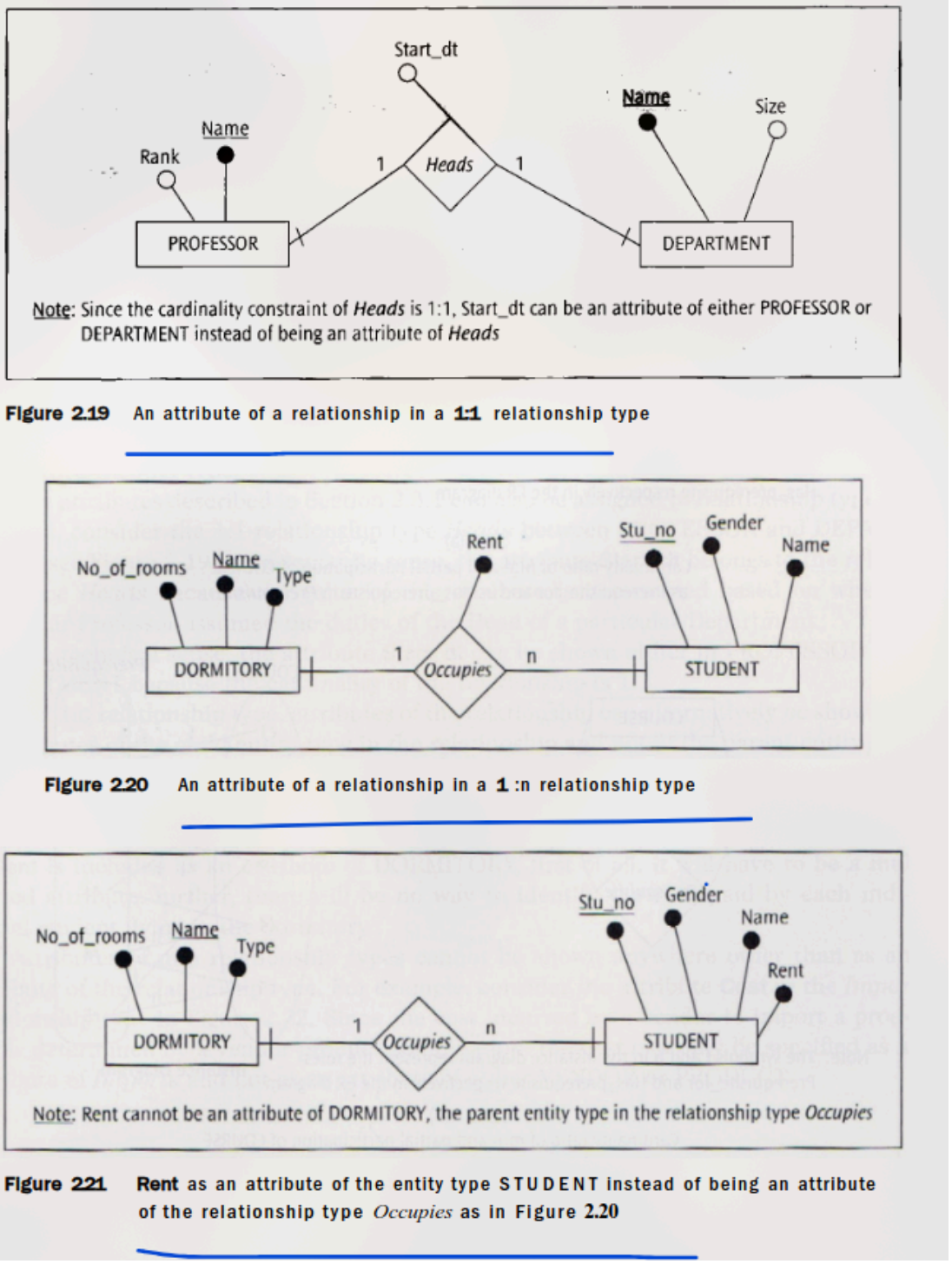


Figure 2.18 Structural constraints for recursive relationships: cardinality ratio of m:n

- **Attributes on Relationships:** Sometimes, an attribute describes the relationship itself rather than one of the participating entities.
  - For **M:N** relationships, attributes describing the relationship *must* be placed on the relationship type (e.g., `DateEnrolled` in an *Enrolls* M:N relationship between `STUDENT` and `COURSE`).
  - For **1:N** relationships, attributes of the relationship can often be migrated to the entity type on the 'N' side. However, placing it on the relationship can sometimes offer better semantic clarity. (e.g., `Rent` on `Occupies` (1:N) between `STUDENT` (N) and `DORMITORY` (1) - *Fig 2.20, p.48*. Technically, `Rent` could be an attribute of `STUDENT` *if a student occupies only one dormitory at a time and rent is specific to that student-dormitory occupation - Fig 2.21, p.48*).

- For **1:1** relationships, attributes of the relationship can be placed on either participating entity type, or on the relationship type. If participation is total on one side, it's often placed there. (e.g., `StartDate` for `Heads` (1:1) between `PROFESSOR` and `DEPARTMENT` - *Fig 2.19, p.48*).
  (Fig 2.19, 2.20, 2.22, p.48-49)



- *Fig 2.22 shows 'Cost' as an attribute of the M:N 'Imports' relationship between `VENDOR` and `PRODUCT`.*

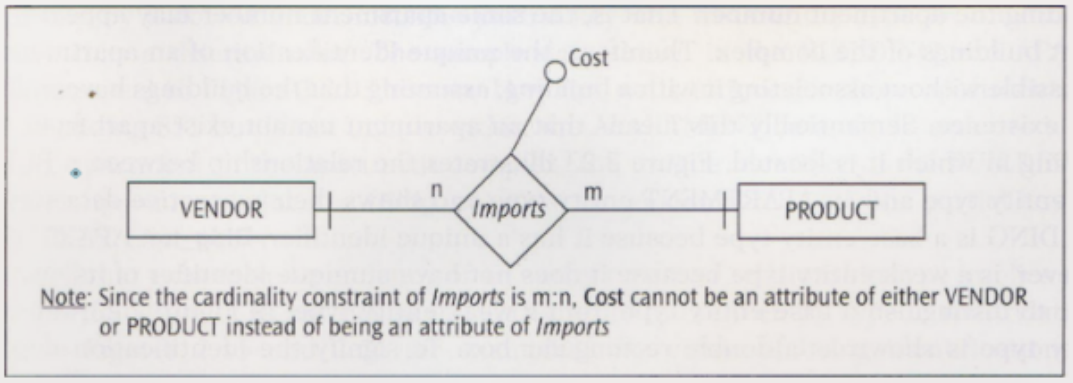
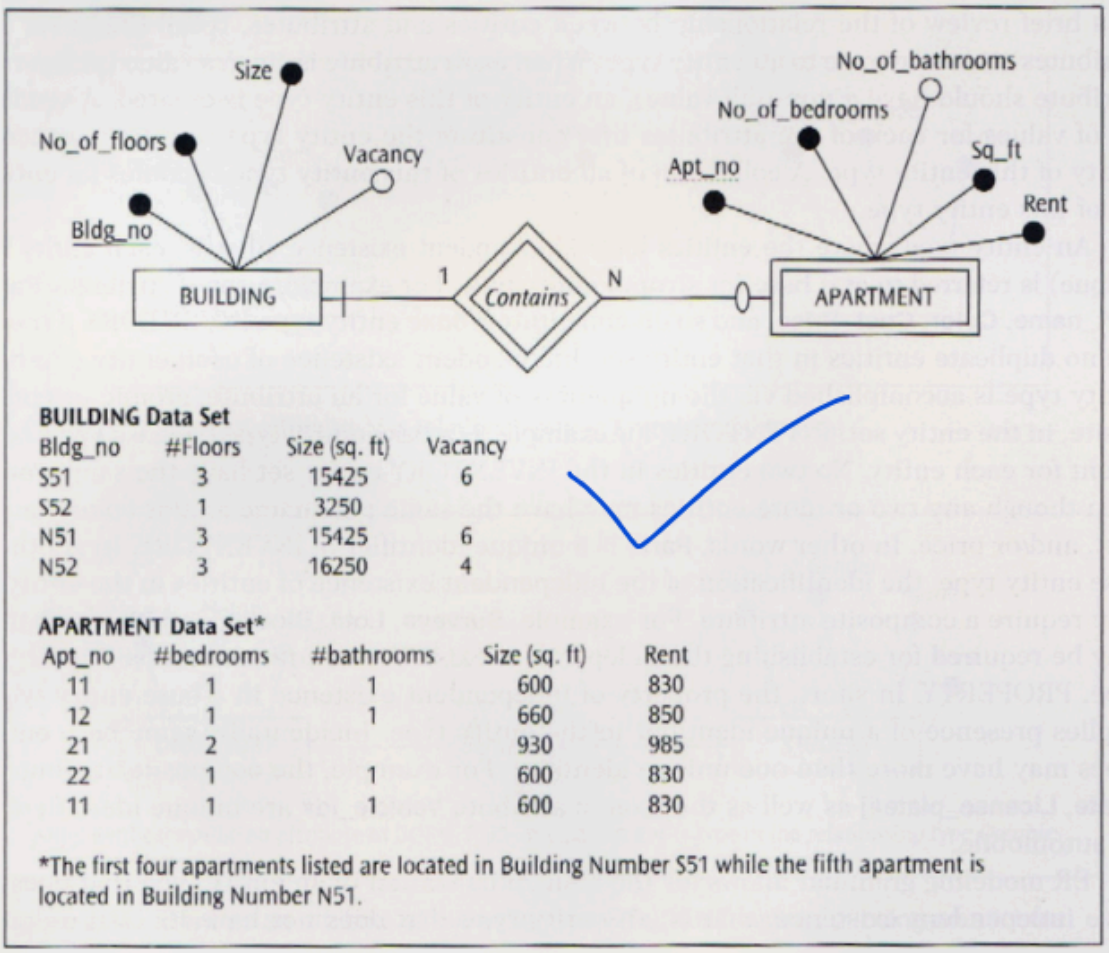


Figure 2.22 An attribute of a relationship in an m:n relationship type

- **Base (Strong) Entity Type:** An entity type that has a primary key consisting of attributes inherent to it and does not depend on any other entity type for its identification. It can exist independently. (p.49)
- **Weak Entity Type:** An entity type that does not have enough attributes to form its own primary key. Its existence and identification depend on another entity type, called the owner (or identifying) entity type. A weak entity is related to its owner via an identifying relationship. (p.49-53)
  - Represented by a double rectangle.
  - **Identifying Relationship:** The relationship connecting a weak entity type to its owner entity type. It contributes to the identification of weak entity instances. Represented by a double diamond. Participation of the weak entity in the identifying relationship is always total.
  - **Partial Key (Discriminator):** An attribute or set of attributes of a weak entity type that, in combination with the primary key of the owner entity type, uniquely identifies instances of the weak entity type. The partial key distinguishes among weak entity instances that are related to the *same* owner entity instance. Represented with a dotted underline.
  - The primary key of a weak entity type is formed by the primary key of its owner entity type(s) (propagated as a foreign key) plus its own partial key.
  - Example: `APARTMENT` might be a weak entity type dependent on `BUILDING`. `Apt_no` would be the partial key. The PK of `APARTMENT` would be (`Bldg_no`, `Apt_no`).
    (Fig 2.23, p.50; Fig 2.24, p.51 - `INTERNSHIP` *weak with two identifying parents; Fig 2.26,*

p.53 - `TRAINING_PROGRAM` weak, identified by another weak entity `INTERNSHIP`)



BUILDING Data Set

Bldg_no	#Floors	Size (sq. ft)	Vacancy
S51	3	15425	6
S52	1	3250	
N51	3	15425	6
N52	3	16250	4

APARTMENT Data Set*

Apt_no	#bedrooms	#bathrooms	Size (sq. ft)	Rent
11	1	1	600	830
12	1	1	660	850
21	2		930	985
22	1	1	600	830
11	1	1	600	830

*The first four apartments listed are located in Building Number S51 while the fifth apartment is located in Building Number N51.

Figure 2.23 Weak entity type: an example

- *Fig 2.23 shows `APARTMENT` as a weak entity type identified by `BUILDING` via the 'Contains' identifying relationship. `Apt_no` is the partial key.*

- **Data Modeling Errors (Vignettes):** (p.54)
  - **Semantic Errors:** Arise from misinterpreting the business rules or requirements. More difficult to detect as conceptual modeling is not an exact science. Requires careful attention to detail and user validation.
  - **Syntactic Errors:** Violations of the ER modeling grammar rules (e.g., connecting an attribute directly to another attribute). Easier to avoid by knowing the rules.
  - *Vignette 1 (p.54): Illustrates errors like modeling "Course" and "Instructor" as single-valued attributes of `COLLEGE` when a college offers many courses and has many instructors (should be multi-valued or, better, separate entity types with relationships).*
- **ER Modeling Process (using Bearcat Inc. example in Ch 3):** (p.55-93)
  Chapter 3 walks through a comprehensive case study (Bearcat Incorporated) to demonstrate

the ER modeling process, progressing from user requirements to a refined ER model.

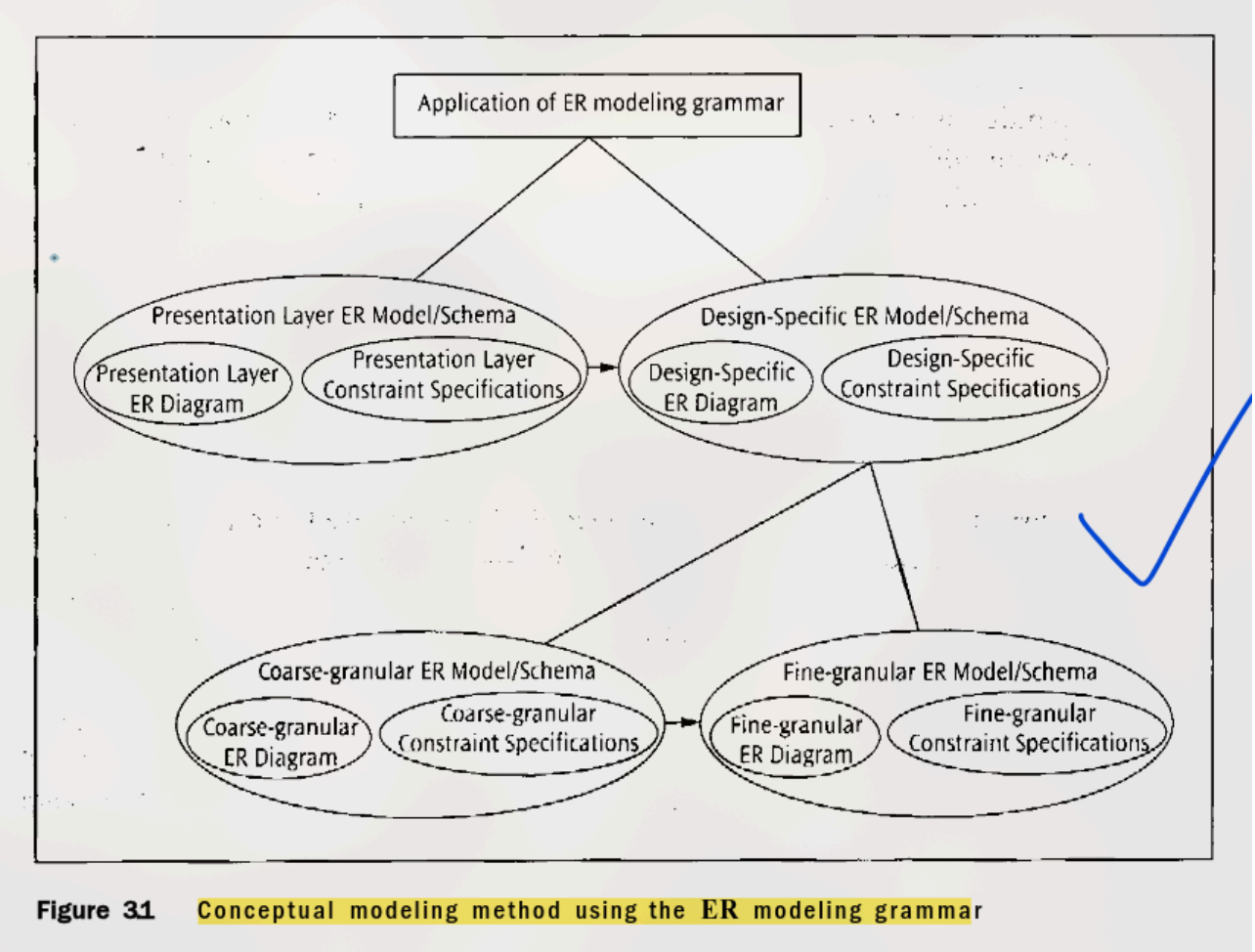


Figure 3.1 Conceptual modeling method using the ER modeling grammar

- *Fig 3.1 illustrates the conceptual modeling method: Presentation Layer ER Model/Schema (ER Diagram + Constraint Specifications) -> Design-Specific ER Model/Schema (Coarse-granular, then Fine-granular).*
- **Presentation Layer ER Model:** (p.58-75)
  - Primary purpose: Communication with end-users.
  - Focus: High-level, surface representation of the application domain.
  - Notation: Uses basic ER constructs (rectangles for entities, diamonds for relationships, ovals for attributes). Cardinality and participation constraints are typically shown using look-across notation (e.g., Chen's notation or Crow's Foot).
  - Includes:
    - **Presentation Layer ER Diagram:** Graphical depiction.
    - **Semantic Integrity Constraints List:** Textual business rules not captured directly in the ERD (e.g., domain constraints, specific deletion rules, complex inter-attribute rules). *(Table 3.1, p.75 provides examples for Bearcat Inc.)*

(Fig 3.2 notation summary, p.60; Fig 3.3 Bearcat ERD, p.72)

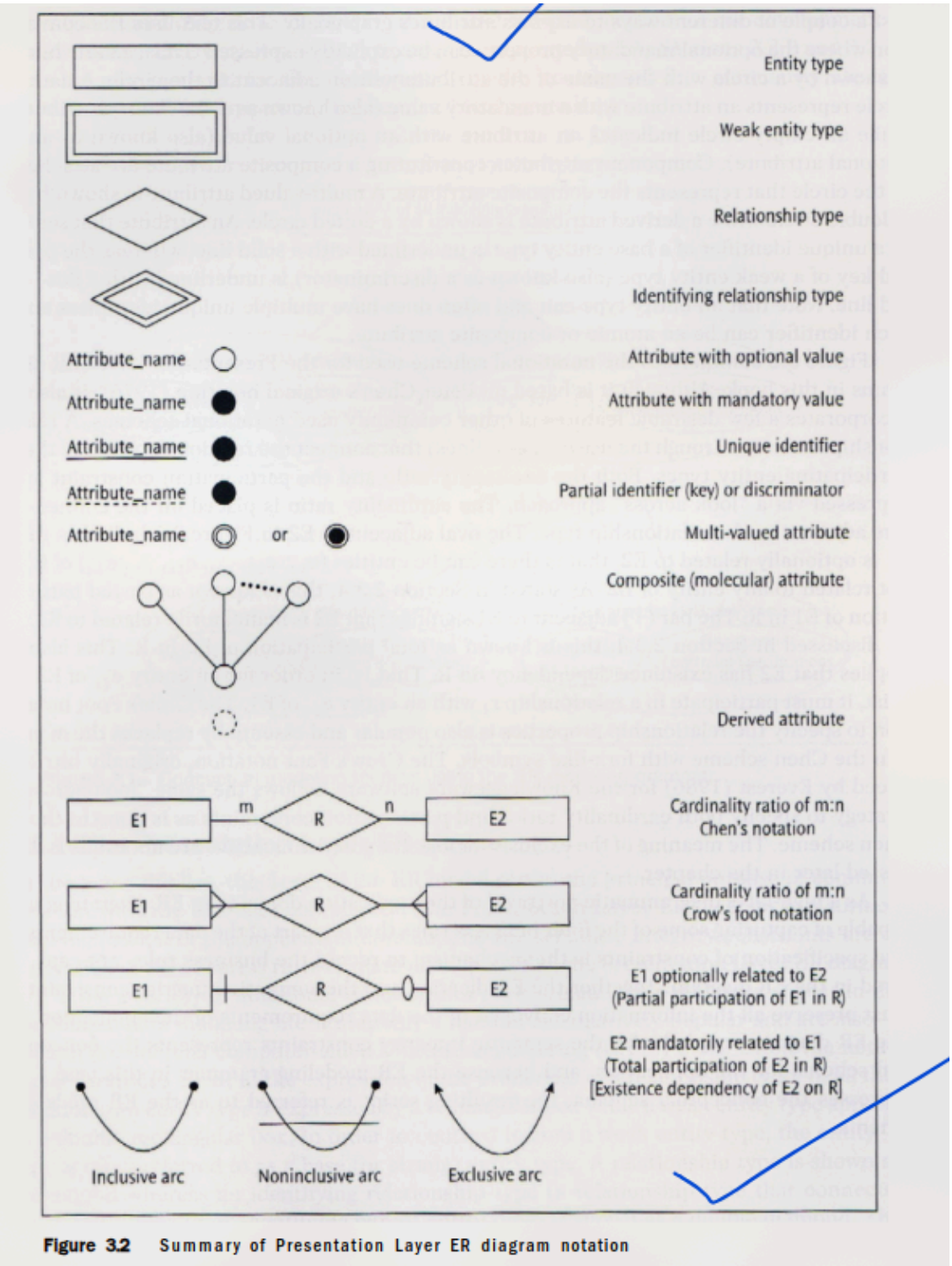


Figure 3.2 Summary of Presentation Layer ER diagram notation

- *Fig 3.2 summarizes the Chen-like notation used: single/double rectangles, single/double diamonds, solid/dotted/double circles for attributes, underlines for keys, (min,max) look-*

across cardinality, arcs (inclusive, noninclusive, exclusive).

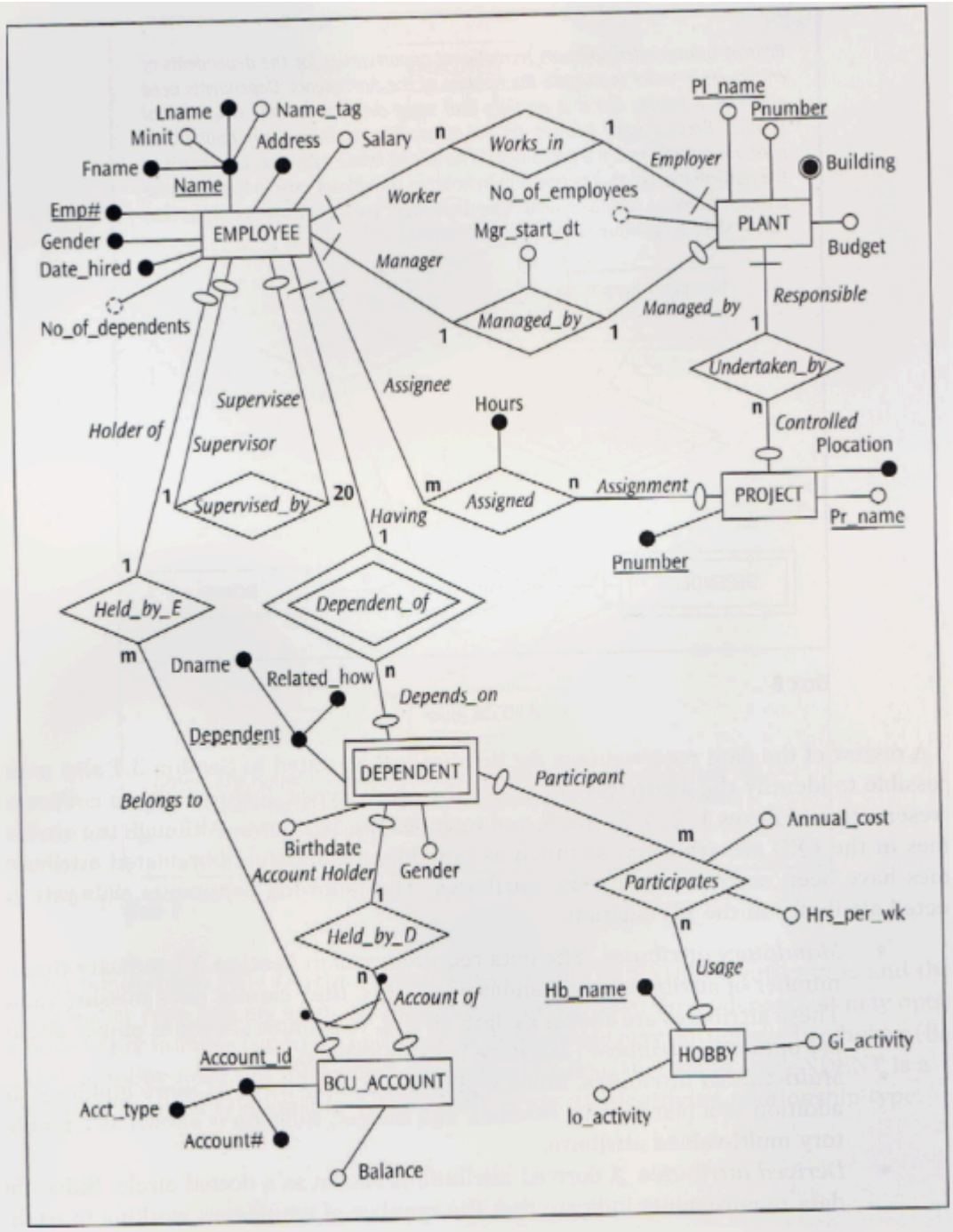


Figure 3.3 Presentation Layer ER diagram for Bearcat Incorporated

- *Fig 3.3 shows the initial Presentation Layer ERD for Bearcat Inc., including entities like* `PLANT`, `EMPLOYEE`, `PROJECT`, `DEPENDENT`, `BCU_ACCOUNT`, `HOBBY` *and their relationships with attributes.*

- **Design-Specific ER Model:** (p.75-93) Transforms the user-oriented Presentation Layer model into a more technically precise model for database designers, preparing it for logical schema mapping.
  - **Coarse Granularity Stage:** (p.75-86)
    - Incorporates more details about attribute characteristics (data type, size - see *Table 3.2, p.76*).

- Introduces the **(min, max) notation** for structural constraints on relationships, applied using a "look here" approach (i.e., (min,max) pair on an edge specifies participation of the entity at *that end* of the edge). This is more precise, especially for higher-degree relationships.
- Explicitly incorporates **deletion rules** (R-Restrict, C-Cascade, N-Set Null, D-Set Default) on the relationship lines in the ERD.
  (Fig 3.4 (min,max) intro, p.78; Fig 3.5 Stage 1 Coarse-granular ERD, p.79; Fig 3.6 deletion rules example, p.82; Fig 3.8 Final Coarse-granular ERD for Bearcat Inc., p.85)

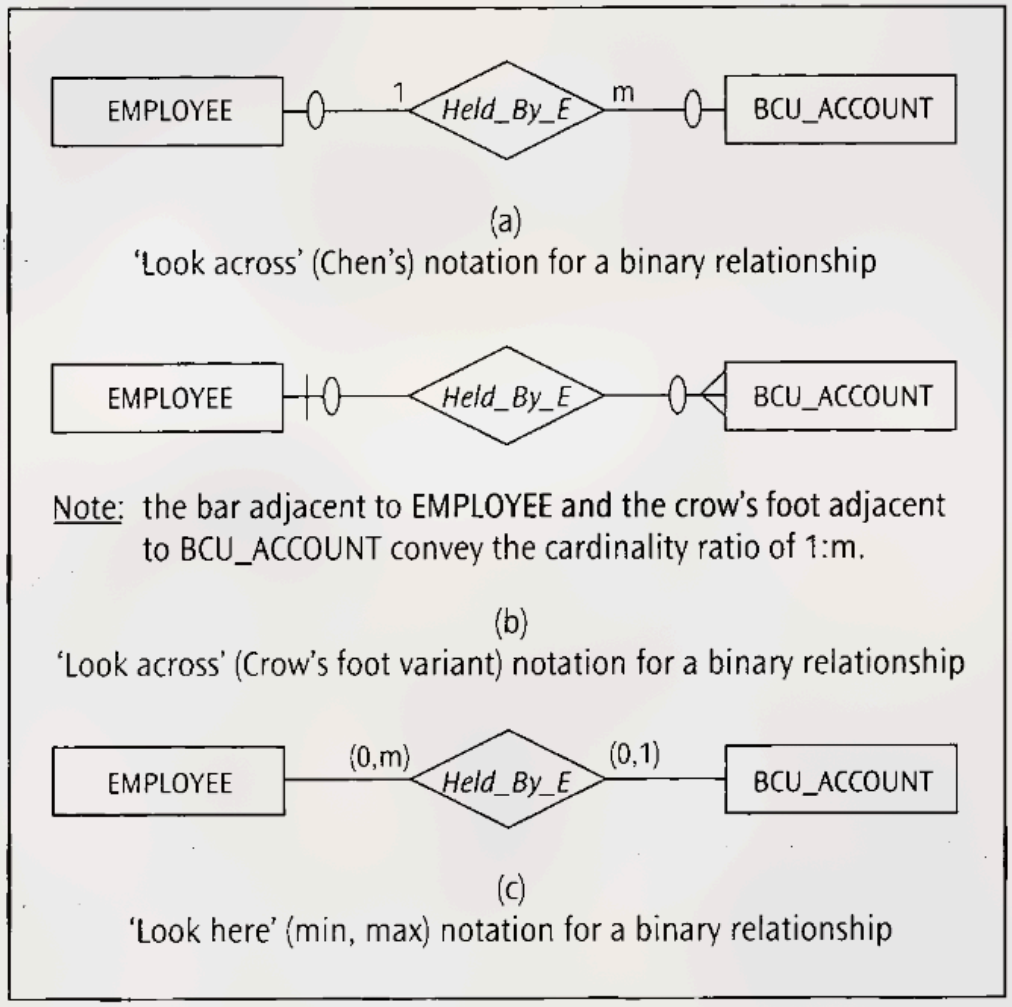


Figure 3.4 Introduction of (min, max) notation for a binary relationship

- *Fig 3.4 introduces the (min,max) notation using the* `Held_By_E` *relationship as an example, contrasting it with Chen's look-across and Crow's Foot.*

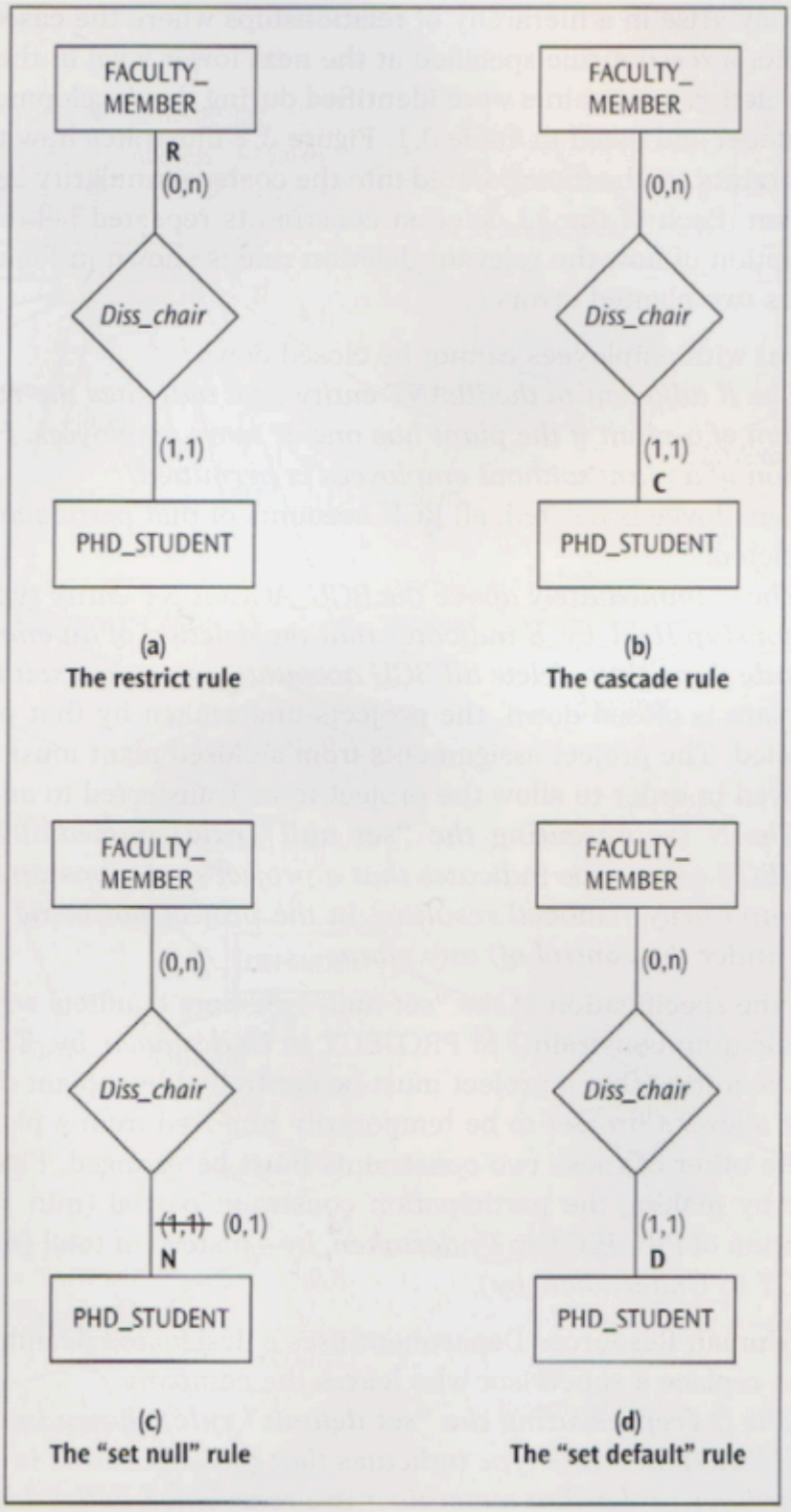


Figure 3.6 Examples of deletion rules

- *Fig 3.8 shows the Bearcat ERD at the coarse-granular stage, with (min,max) cardinalities and deletion rules (R, C, N) added.*

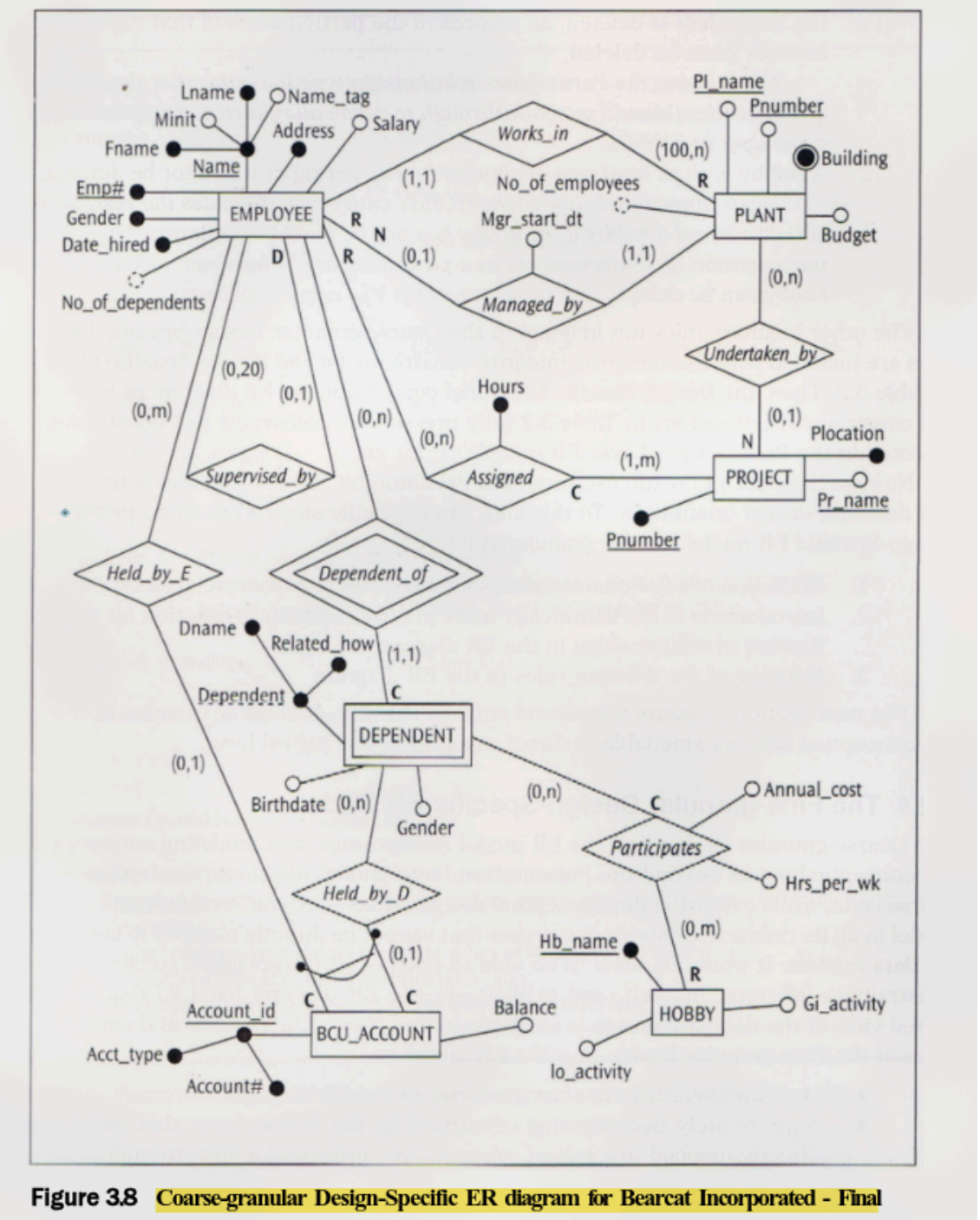


Figure 3.8 Coarse-granular Design-Specific ER diagram for Bearcat Incorporated - Final

- **Fine Granularity Stage:** (p.86-93)
  - Further refines the Coarse-granular model to make it directly mappable to a logical schema (especially relational).
  - **Resolves Multi-valued Attributes:** Transforms them into separate weak entity types (or by incorporating into the PK if appropriate, though less common in robust designs). *(Fig 3.9, p.88 shows two methods: 1. appending to PK to make new unique ID, 2. transforming to a weak entity).*
  - **Resolves M:N Relationships:** Decomposes them into two 1:N relationships with an intermediate (gerund/associative) entity type. Attributes of the M:N relationship become attributes of this new gerund entity. *(Fig 3.10, p.91 shows* `Assigned` *M:N relationship becoming* `ASSIGNMENT` *gerund entity).*

- Incorporates attribute data types and sizes directly into the ER diagram (e.g., `[A,20]` for an alphabetic attribute of size 20).
- The semantic integrity constraints list is updated/translated (*Table 3.3, p.87*).
- The resulting Fine-granular Design-Specific ER model, along with its constraint specifications, is fully information-preserving and ready for direct mapping to a logical schema.
  (Fig 3.9 multi-valued resolution, p.88; Fig 3.10 M:N resolution, p.91; Fig 3.11 Bearcat Fine ERD, p.92)

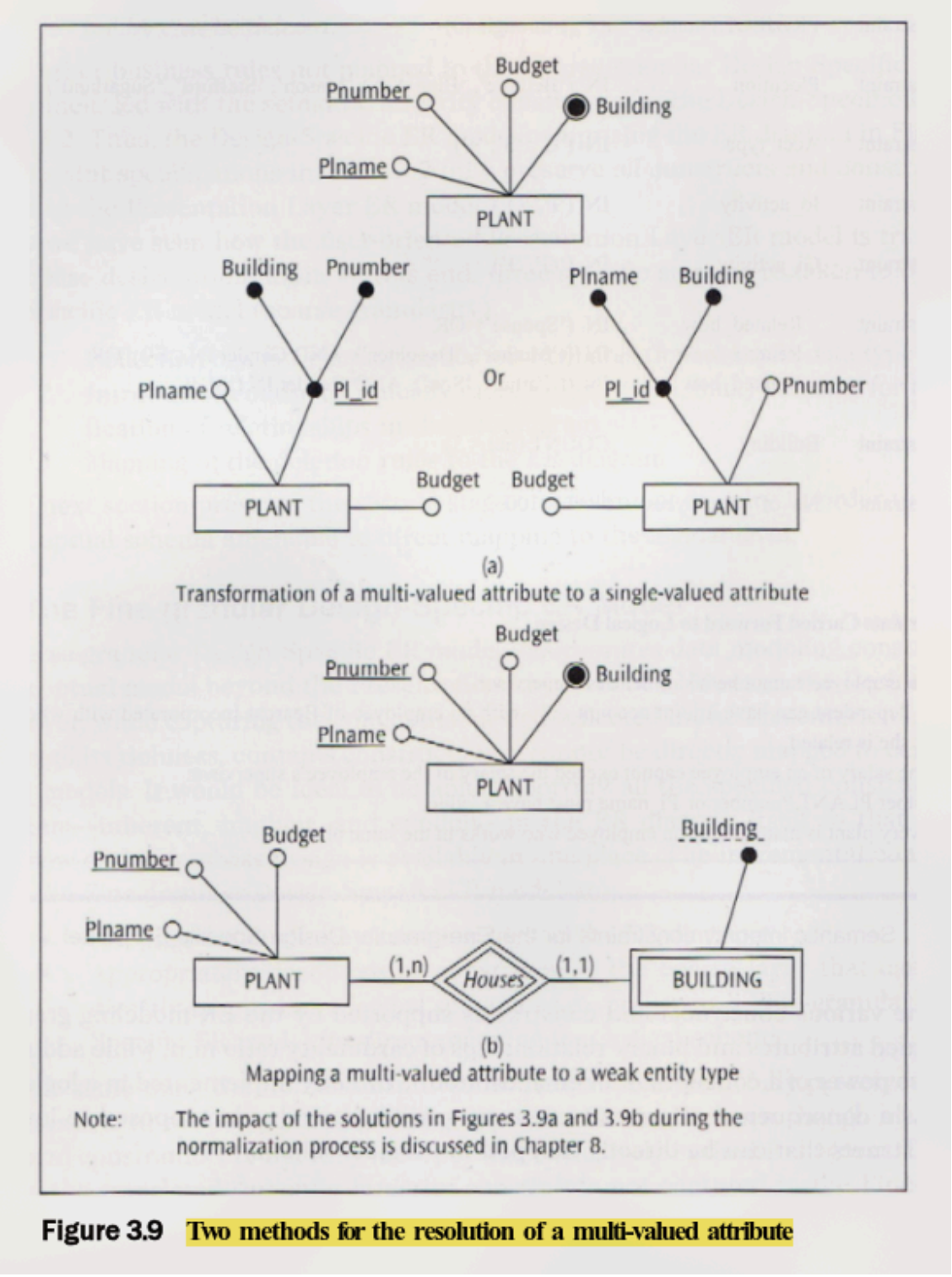


Figure 3.9 **Two methods for the resolution of a multi-valued attribute**

- *Fig 3.9 illustrates resolving the multi-valued attribute* `Building` *of* `PLANT` *either by making* `(Pnumber, Building)` *a new PK or by creating a weak entity* `BUILDING`

related to `PLANT`.

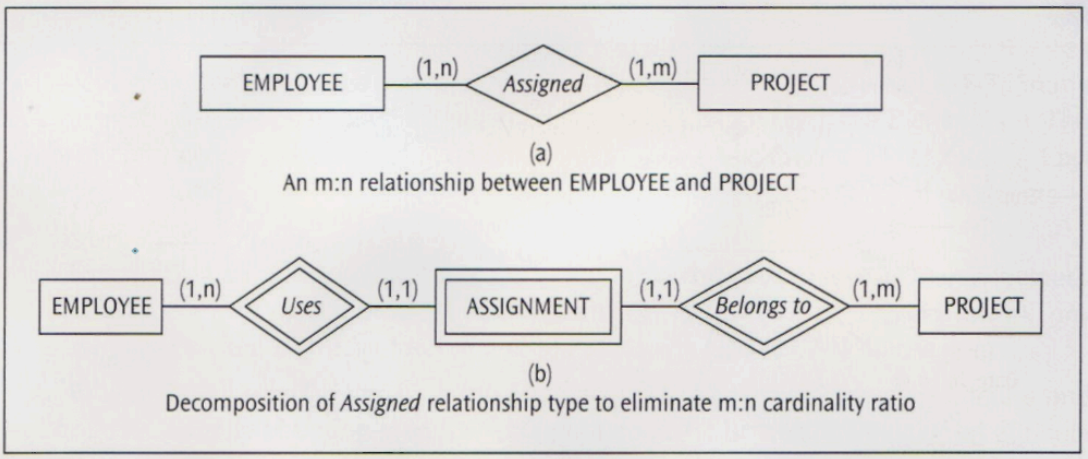


Figure 3.10 Resolution of the m:n cardinality ratio

- *Fig 3.10 shows decomposing an M:N* `Assigned` *relationship between* `EMPLOYEE` *and* `PROJECT` *into a gerund entity* `ASSIGNMENT` *with two identifying 1:N*

relationships.

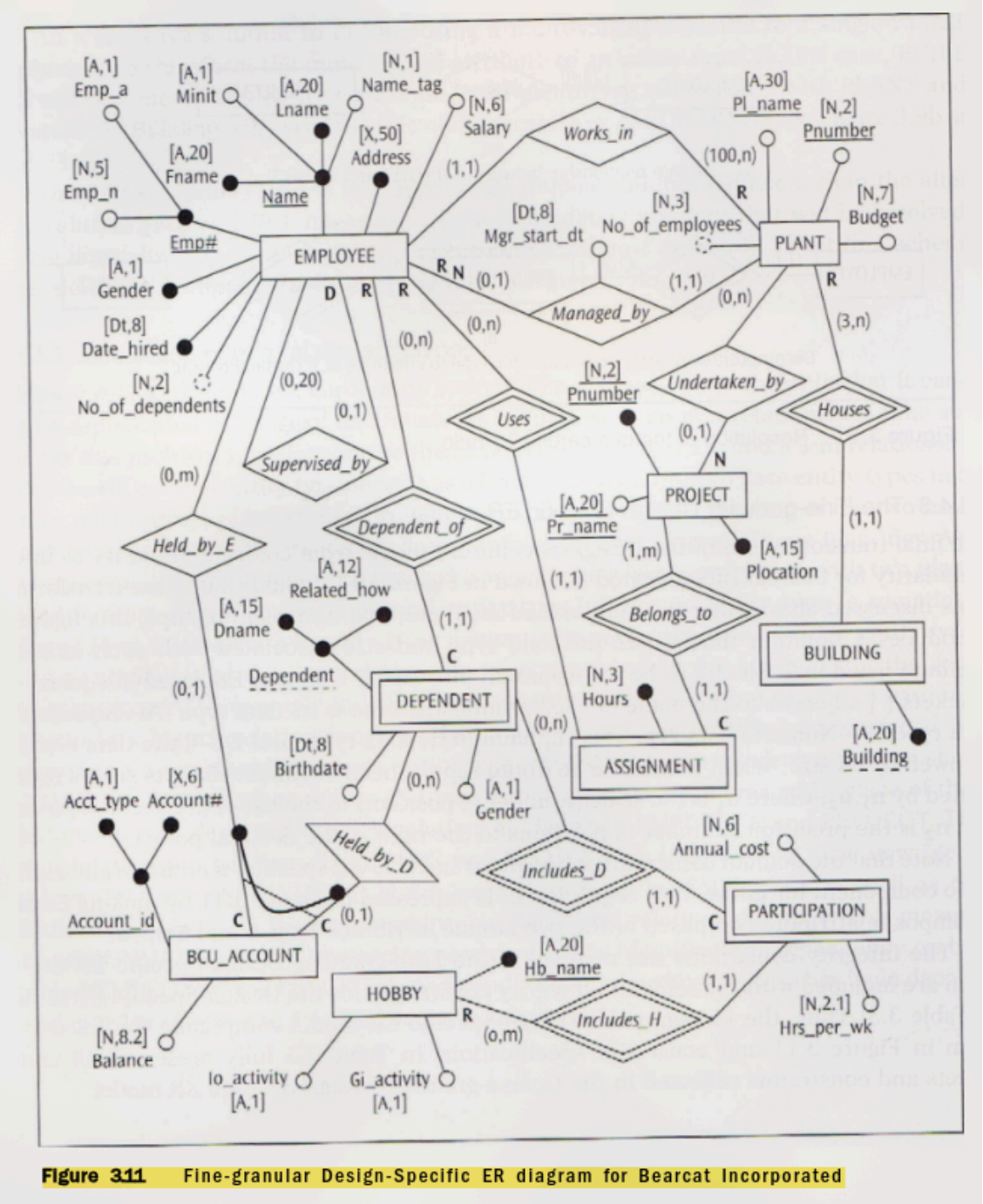


Figure 3.11 Fine-granular Design-Specific ER diagram for Bearcat Incorporated

- *Fig 3.11 presents the final Fine-granular Design-Specific ERD for Bearcat Inc., ready for logical mapping, showing resolved M:N and multi-valued attributes, and attribute details like type/size.*

- **EER Modeling (Enhanced Entity-Relationship) [PYQ Q3b, Q4a]**
  EER modeling extends the basic ER model by introducing constructs that allow for more sophisticated representation of data semantics, particularly for complex applications like those found in CAD/CAM, software engineering (object modeling), and artificial intelligence (knowledge representation). These constructs were developed in response to the demands of these applications in the 1980s when the basic ER model proved inadequate.
  - **Purpose:** To enhance the capability of ER modeling grammar by adding constructs to model:
    - Subgroupings within an entity type (subclasses).

- Relationships between these subgroupings and their more generic parent entity type (superclass).
- Situations where an entity type is a collection of entities from different entity types.

- **Superclass/Subclass (SC/sc) Relationship (Is-A Relationship):** (p.119-125)
  This is the fundamental concept in EER modeling. It represents an "is-a" (or "is-a-kind-of") relationship between a more general entity type (superclass) and one or more specialized entity types (subclasses).
  (Fig 4.4, p.124; Fig 4.5, p.125)

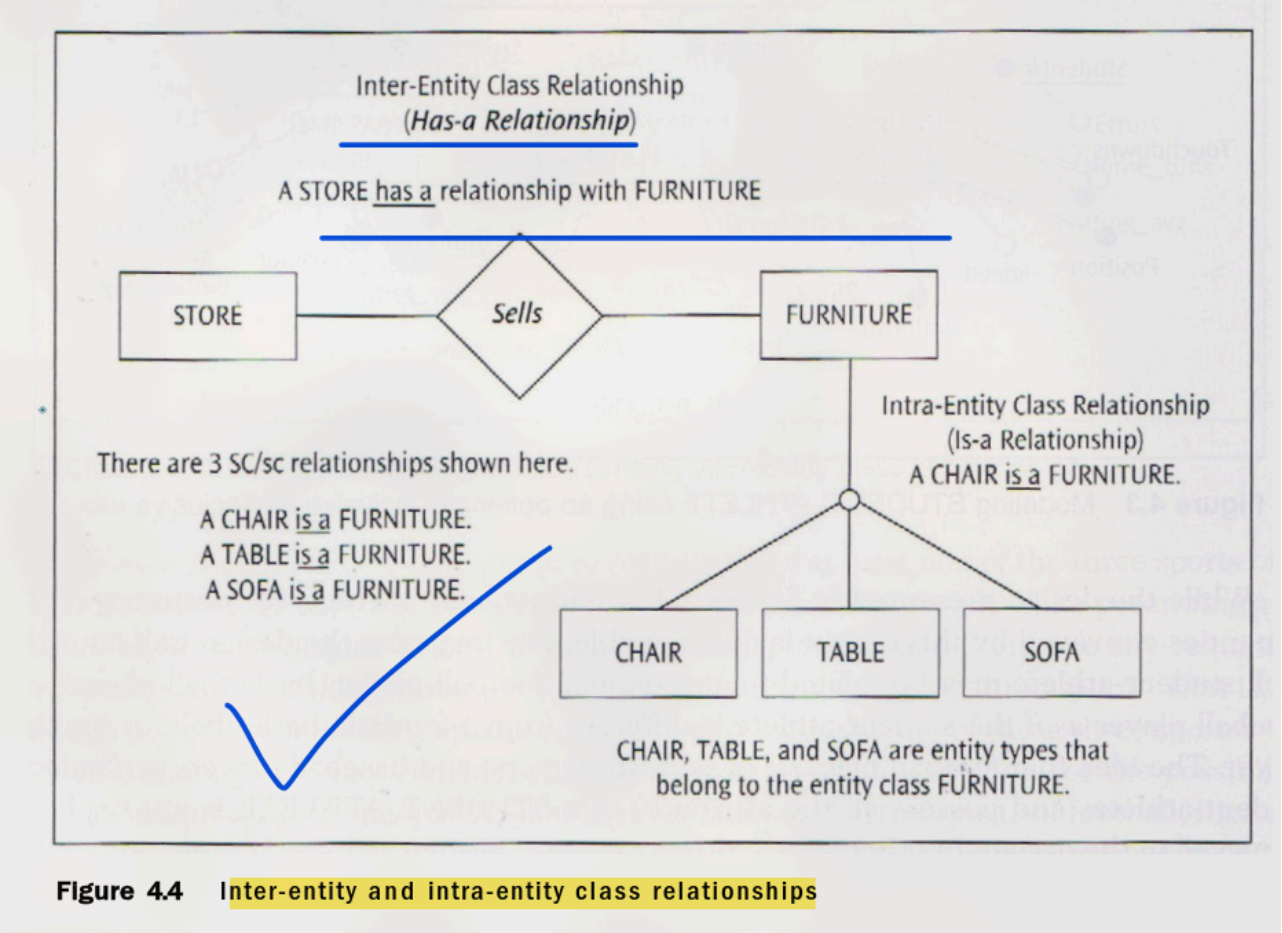


Figure 4.4 Inter-entity and intra-entity class relationships

- *Fig 4.4 illustrates: Inter-Entity Class Relationship (`STORE` Has-a `FURNITURE`) vs. Intra-Entity Class Relationship (`CHAIR` Is-a `FURNITURE`, `TABLE` Is-a `FURNITURE`, `SOFA` Is-a*

FURNITURE).

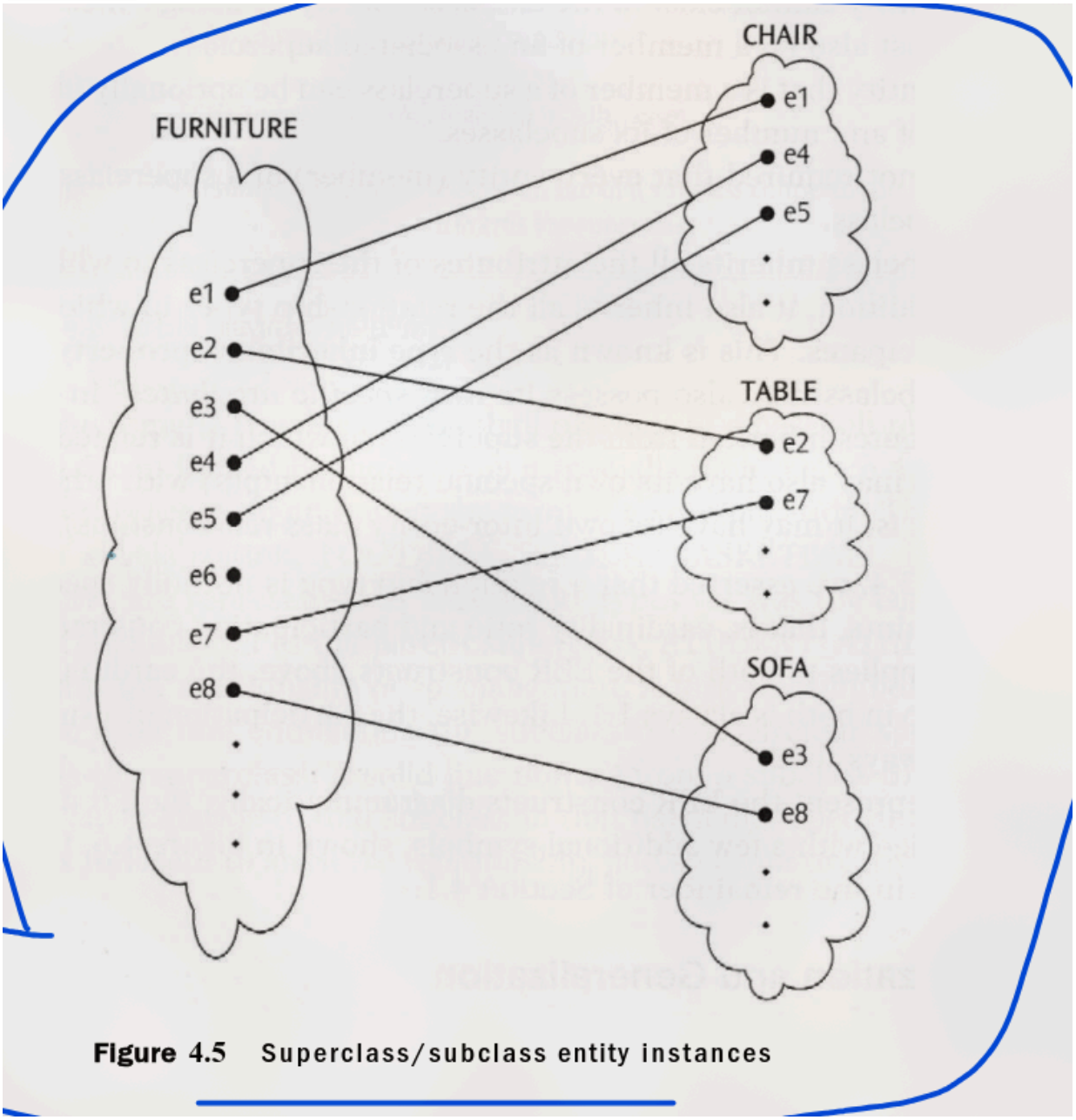


Figure 4.5 Superclass/subclass entity instances

- *Fig 4.5 shows instances: entities e1, e2, e3, etc. belonging to* `FURNITURE` *(superclass). Some of these instances also belong to subclasses* `CHAIR`, `TABLE`, `SOFA`. *For example, e1 is a* `FURNITURE` *instance and also a* `CHAIR` *instance.*
- **Superclass (SC):** A generic entity type that has one or more distinct subgroupings (subclasses) with attributes or relationships unique to those subgroups (e.g., `EMPLOYEE`, `VEHICLE`, `FURNITURE`).
- **Subclass (sc):** A specialized entity type that represents a distinct subgrouping of a superclass. It inherits all attributes and relationships of its superclass (this is called **type inheritance** or **inheritance property**). Additionally, a subclass can have its own specific attributes and can participate in its own specific relationships (inter-entity class relationships) not applicable to other subclasses or the superclass in general (e.g., `SALARIED_EMPLOYEE` and `HOURLY_EMPLOYEE` are subclasses of `EMPLOYEE`; `CAR` and `TRUCK` are subclasses of `VEHICLE`).

- An entity instance of a subclass is also an instance of its superclass. For example, every `CAR` is a `VEHICLE`.
- The relationship between a superclass and any of its subclasses is always **1:1** (one-to-one) in terms of entity instances. One instance of a superclass can correspond to at most one instance in a specific subclass (if disjoint) or one instance in each of the overlapping subclasses it belongs to.
- The **participation of a subclass instance in the SC/sc relationship is always total**: an entity cannot exist in the database merely by being a member of a subclass; it must also be a member of an associated superclass.

- **Specialization and Generalization [PYQ Q3b]:** (p.125-133) These are two complementary processes for defining SC/sc relationships.
  (Fig 4.6 notation, p.127; Fig 4.7 example, p.128)

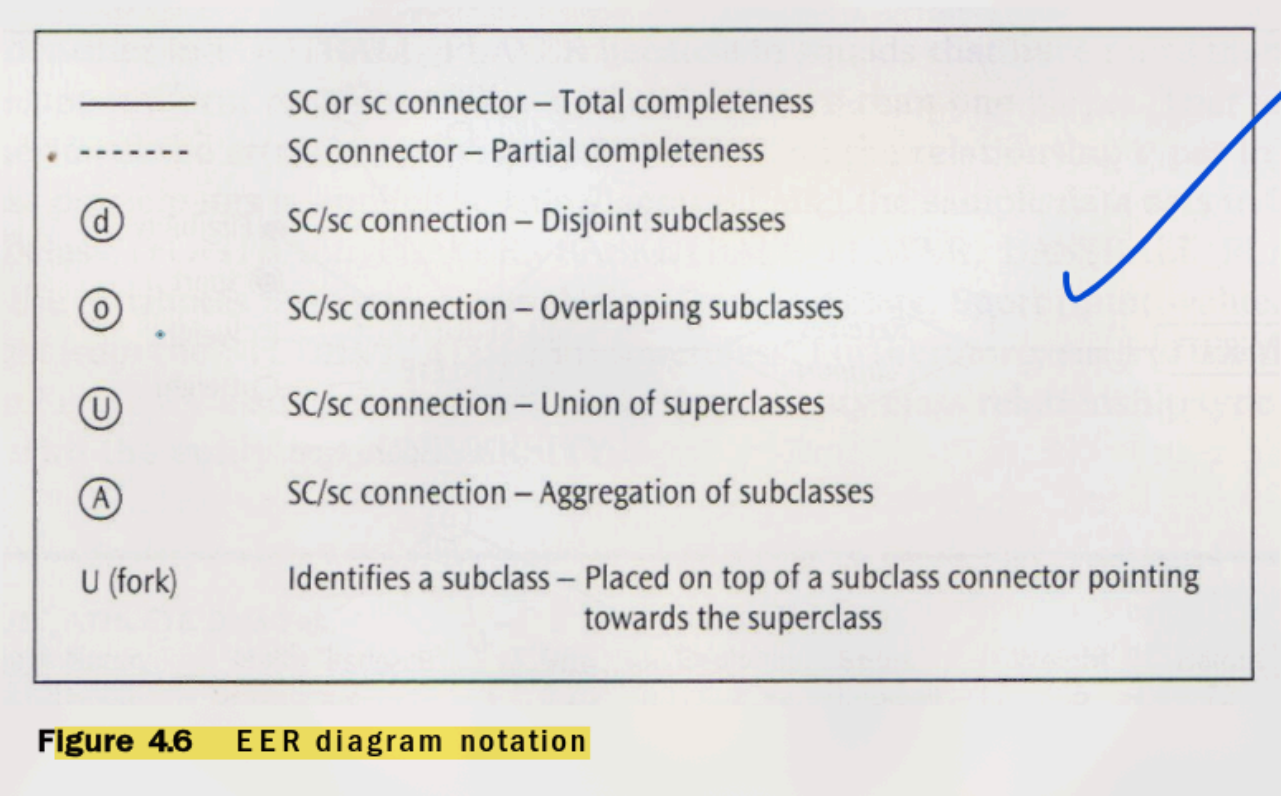


Figure 4.6 EER diagram notation

- *Fig 4.6 shows EER diagram notation: SC/sc connector (total/partial completeness for SC), SC/sc connection (disjoint/overlapping subclasses via 'd' or 'o' in circle), Union of superclasses ('U' in circle for categorization), Aggregation ('A' in circle), U-fork opening*

towards SC for subclasses.

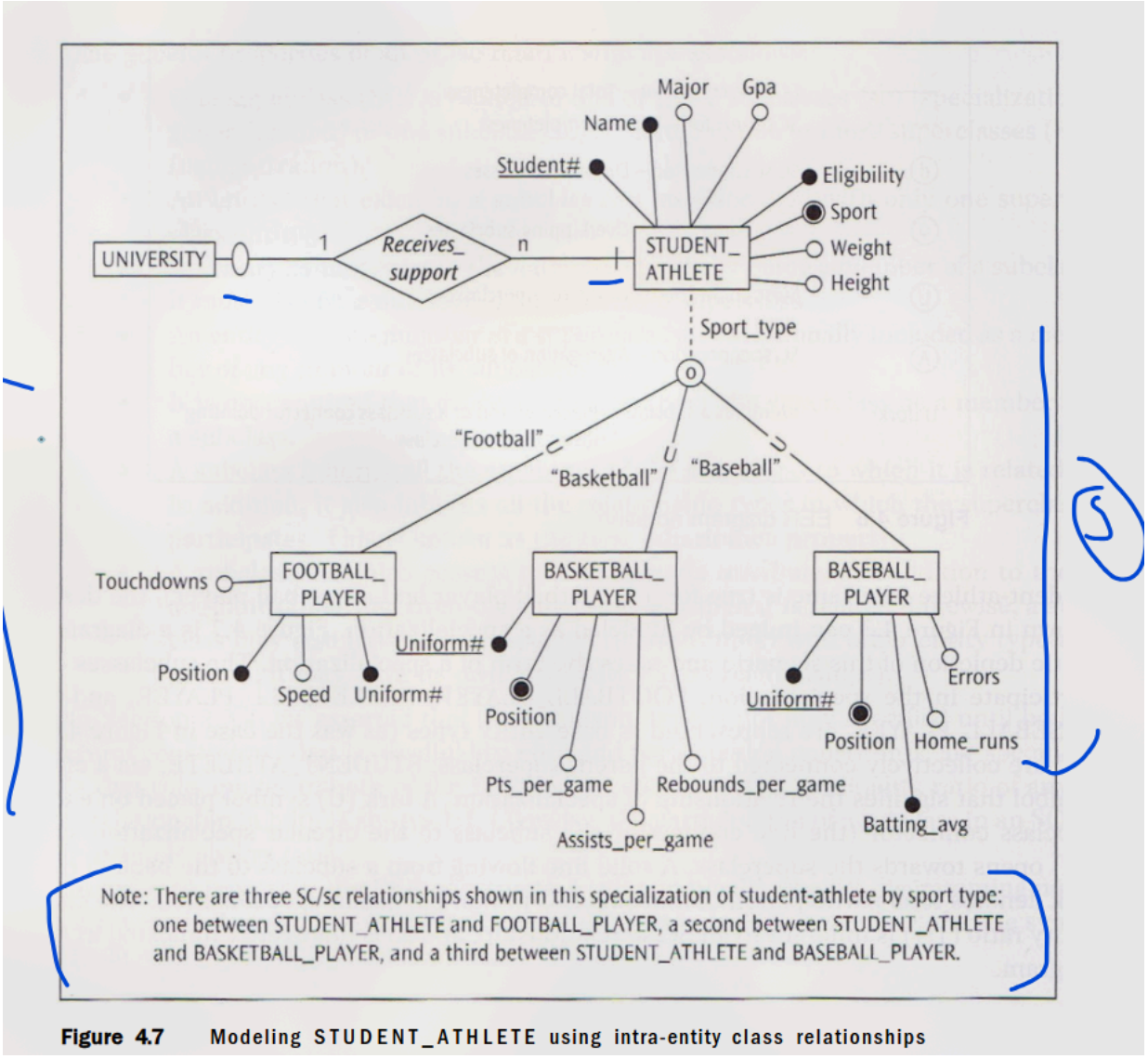


Figure 4.7 Modeling STUDENT_ATHLETE using intra-entity class relationships

- *Fig 4.7 models `STUDENT_ATHLETE` (SC) specialized into `FOOTBALL_PLAYER`, `BASKETBALL_PLAYER`, `BASEBALL_PLAYER` (sc's). It shows partial participation of SC (dotted line from SC to circle - meaning some student-athletes might not be in these three specific sports) and overlapping subclasses ('o' in circle - meaning a student-athlete could be, e.g., both a football and basketball player).*
- **Specialization:** A top-down process of defining a set of subclasses of a superclass. The subclasses are distinguished by some specific properties (attributes or relationships) not common to all instances of the superclass. (e.g., Starting with `EMPLOYEE` and defining `SECRETARY`, `ENGINEER`, `TECHNICIAN` based on job type).
- **Generalization:** A bottom-up process of identifying common features (attributes and relationships) among a set of entity types and abstracting them into a common superclass. The original entity types become subclasses. (e.g., Starting with `CAR` and `TRUCK` and generalizing them to `VEHICLE` by factoring out common attributes like `VehicleID`, `LicensePlateNo`).
- Specialization and generalization are essentially two sides of the same coin, representing the same SC/sc relationship structure.
- **Constraints on Specialization/Generalization:**

        - **Disjointness Constraint:** Specifies whether an instance of a superclass can be a member of one or more subclasses within a *single specialization*.
            - **Disjoint (d):** An instance of the superclass can be a member of **at most one** of the subclasses in the specialization. Indicated by 'd' inside the circle symbol of the specialization. (e.g., An `EMPLOYEE` can be either `SALARIED` or `HOURLY`, but not both, within a "pay_type" specialization).
            - **Overlapping (o):** An instance of the superclass can be a member of **more than one** of the subclasses in the specialization. Indicated by 'o' inside the circle. (e.g., A `PERSON` can be both an `EMPLOYEE` and a `STUDENT` in a "role" specialization).
        - **Completeness (Totalness) Constraint:** Specifies whether an instance of a superclass *must* be a member of at least one subclass in the specialization.
            - **Total Specialization/Generalization:** Every entity instance in the superclass *must* be a member of at least one subclass in the specialization. Indicated by a double line from the superclass to the circle (or a solid line in the textbook's notation, as in *Fig 4.7* if the line from SC to circle were solid). (e.g., If every `EMPLOYEE` *must* be either `SALARIED` or `HOURLY`).
            - **Partial Specialization/Generalization:** An entity instance in the superclass is *not required* to belong to any of the subclasses in the specialization. Indicated by a single line from the superclass to the circle (or a dotted line in the textbook's notation, as in *Fig 4.7*). (e.g., An `EMPLOYEE` might not be a `MANAGER`; some employees are just regular staff).
    - **Defining Subclasses:**
        - **Predicate-defined (or condition-defined) Subclass:** Membership in a subclass is determined by an explicit condition (predicate) based on the value of some attribute of the superclass (called the defining attribute or subclass discriminator). (e.g., For `EMPLOYEE` SC, if `JobType = 'Secretary'`, then the employee is an instance of the `SECRETARY` sc). The condition is shown next to the line from SC to the circle, and values for each sc are shown on lines to sc's.
        - **Attribute-defined Specialization:** If all subclasses in a specialization are defined by predicates on the *same* attribute of the superclass (e.g., `JobType` defining `SECRETARY`, `ENGINEER`).
        - **User-defined Subclass:** Membership in a subclass is not based on any explicit condition but is specified individually by the user or application when an entity instance is created. (e.g., A `MANAGER` subclass of `EMPLOYEE` might be user-defined).
- **Hierarchy and Lattice of Specializations/Generalizations [PYQ Q3b]:** (p.133-136) Subclasses can themselves become superclasses for further specializations, leading to multi-level structures.
    - **Specialization Hierarchy:** A structure where each subclass belongs to only one superclass/subclass relationship as a subclass (i.e., a subclass has only one direct parent superclass). This forms a tree-like structure. Attributes and relationships are inherited down

the hierarchy from the root superclass to all its descendant subclasses.
(Fig 4.9, p.135)

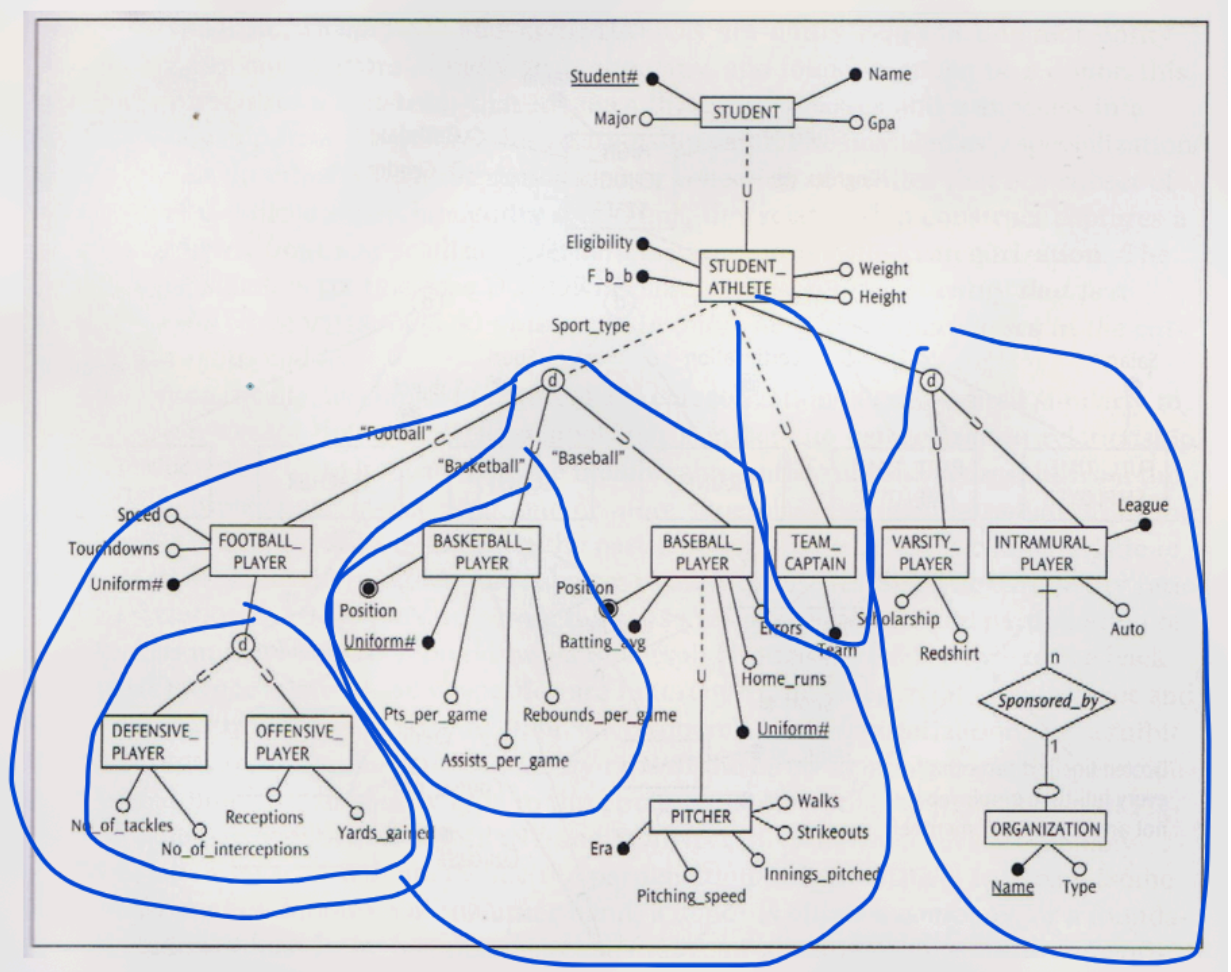


Figure 4.9 A specialization/generalization hierarchy

- *Fig 4.9 shows a three-level hierarchy:* `STUDENT` -> `STUDENT_ATHLETE` -> `FOOTBALL_PLAYER` -> {`DEFENSIVE_PLAYER`, `OFFENSIVE_PLAYER`}. *An* `OFFENSIVE_PLAYER` *inherits attributes from* `FOOTBALL_PLAYER`, `STUDENT_ATHLETE`, *and* `STUDENT`.

- **Specialization Lattice (Multiple Inheritance):** A structure where a subclass can be a subclass in more than one distinct superclass/subclass relationship (i.e., a subclass can have multiple direct parent superclasses from *different* specializations). This allows the subclass (called a **shared subclass**) to inherit attributes and relationships from all its parent superclasses. Any attribute or relationship type inherited more than once via different paths is not duplicated in the shared subclass.

(Fig 4.10, p.136)

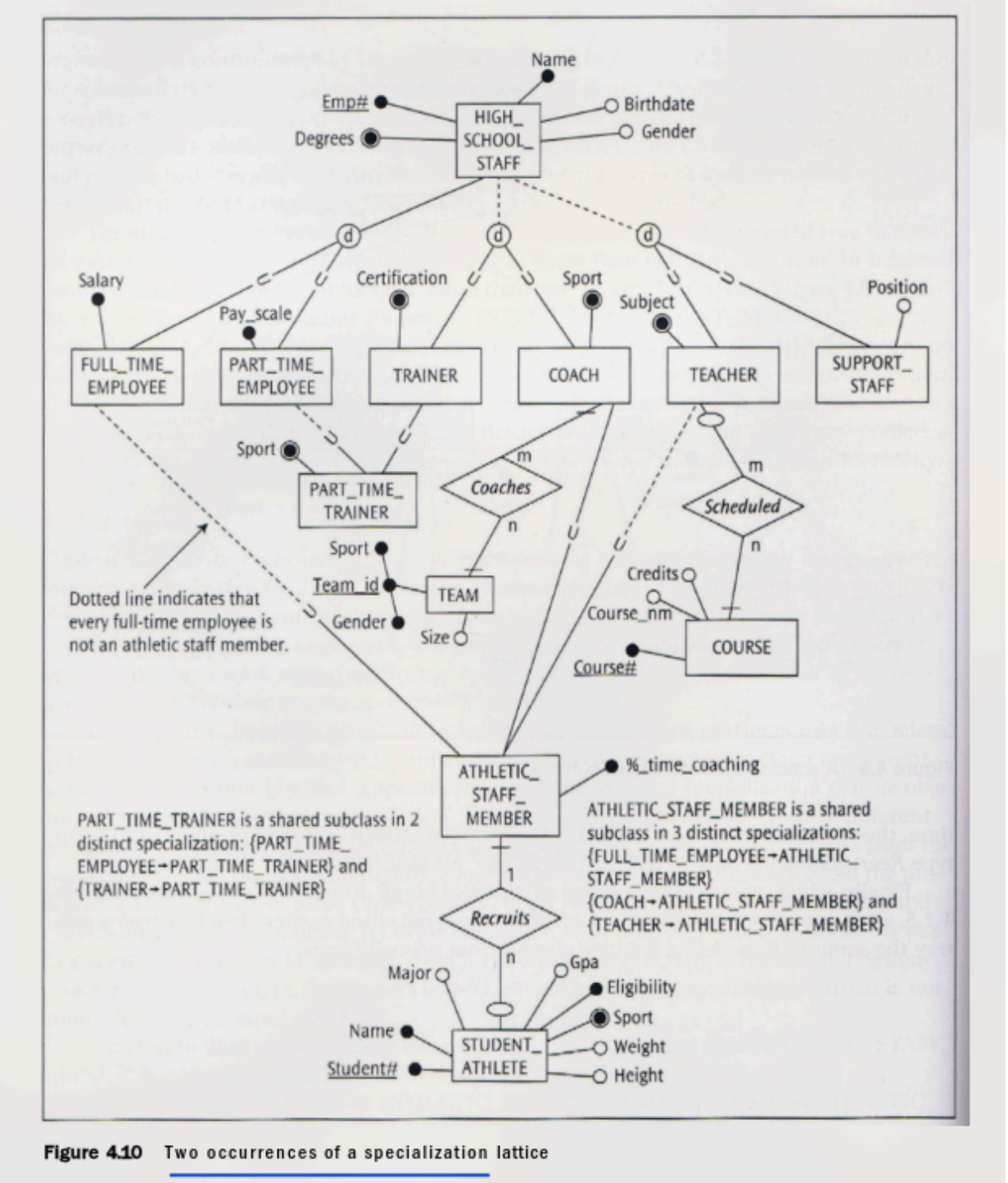


Figure 4.10 Two occurrences of a specialization lattice

- *Fig 4.10 shows* `ATHLETIC_STAFF_MEMBER` *as a shared subclass of* `FULL_TIME_EMPLOYEE`, `COACH`, *and* `TEACHER` *(each from a different specialization originating from* `HIGH_SCHOOL_STAFF`*). It inherits properties from all three, and indirectly from* `HIGH_SCHOOL_STAFF` *(only once).*
- `PART_TIME_TRAINER` is also shown as a shared subclass of `PART_TIME_EMPLOYEE` and `TRAINER`.

- **Categorization [PYQ Q3b]:** (p.136-139)
  Used when a subclass (called a **category**) represents a collection of entities that is a *subset of the union* of entities from two or more distinct superclasses (which usually have different entity types). This is different from specialization/generalization where subclasses are subsets of *one* superclass.

(Fig 4.11, p.138)

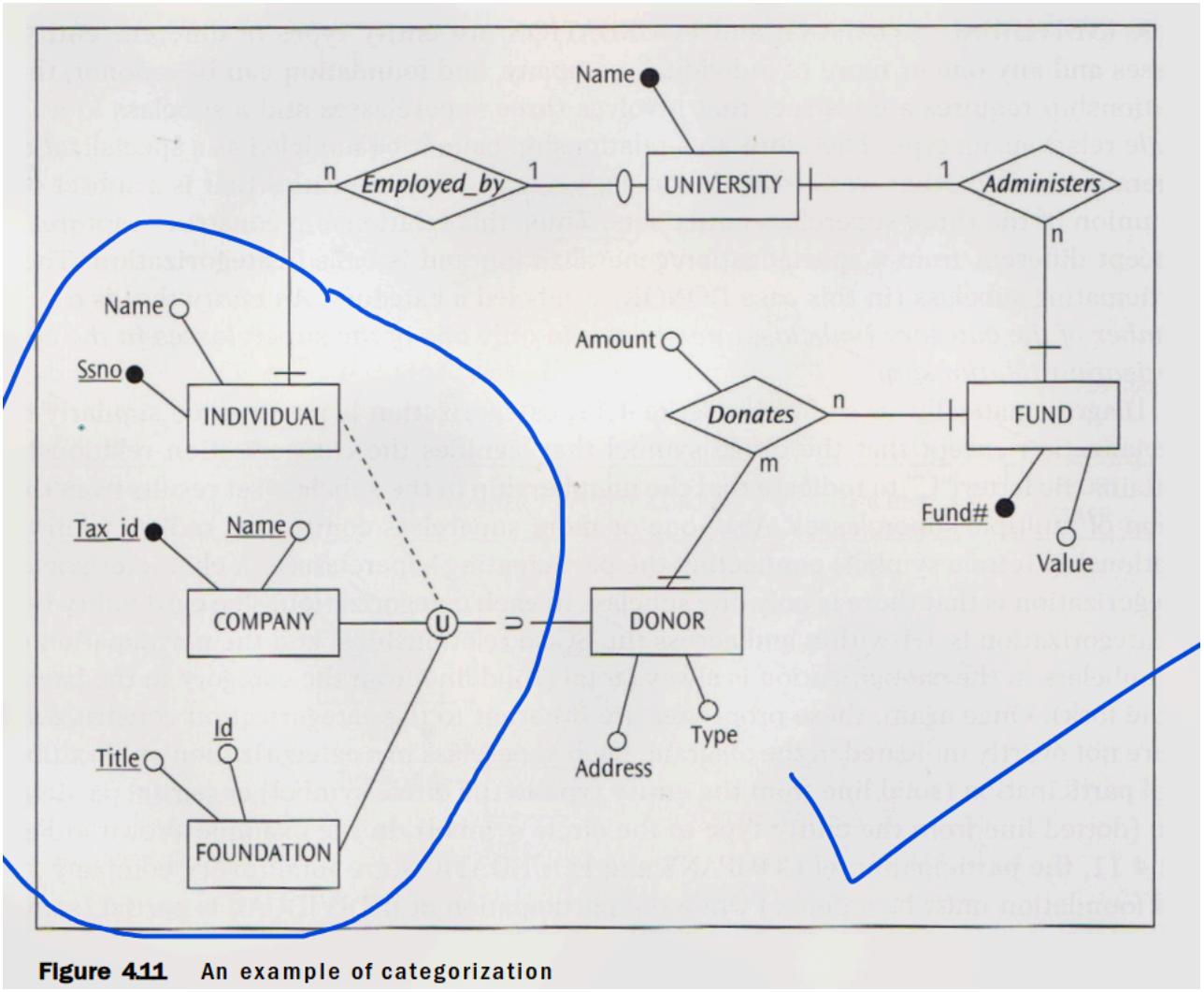


Figure 4.11 An example of categorization

- *Fig 4.11 models* `DONOR` *as a category. A* `DONOR` *can be an* `INDIVIDUAL`*, a* `COMPANY`*, or a* `FOUNDATION`*. The 'U' in the circle signifies union. The* `DONOR` *entity is related to* `FUND` *via* `Donates` *relationship.*
- A category has a single SC/sc relationship type involving multiple superclasses and one subclass (the category itself).
- An instance of the category *must exist in only one* of its superclasses.
- **Type Inheritance is Selective:** An instance of the category inherits attributes and relationships from *only that superclass to which it belongs*. This is in contrast to multiple inheritance in a specialization lattice where a shared subclass inherits from all its parents.
- **Notation:** Represented by a circle containing 'U' (for union) connected to multiple superclasses and one category subclass. The U-fork symbol on the subclass line opens towards the category.
- **Participation of Superclasses:** Can be total or partial for each superclass (e.g., every `COMPANY` might be a `DONOR`, but only some `INDIVIDUALS` are `DONORS`).
- **Participation of Category:** Always total (an entity cannot be just a `DONOR`; it must be an `INDIVIDUAL` donor, `COMPANY` donor, or `FOUNDATION` donor).
- **Total Category vs. Partial Category:** If all superclasses have total participation, the category is total (union of all superclass instances). If any superclass has partial

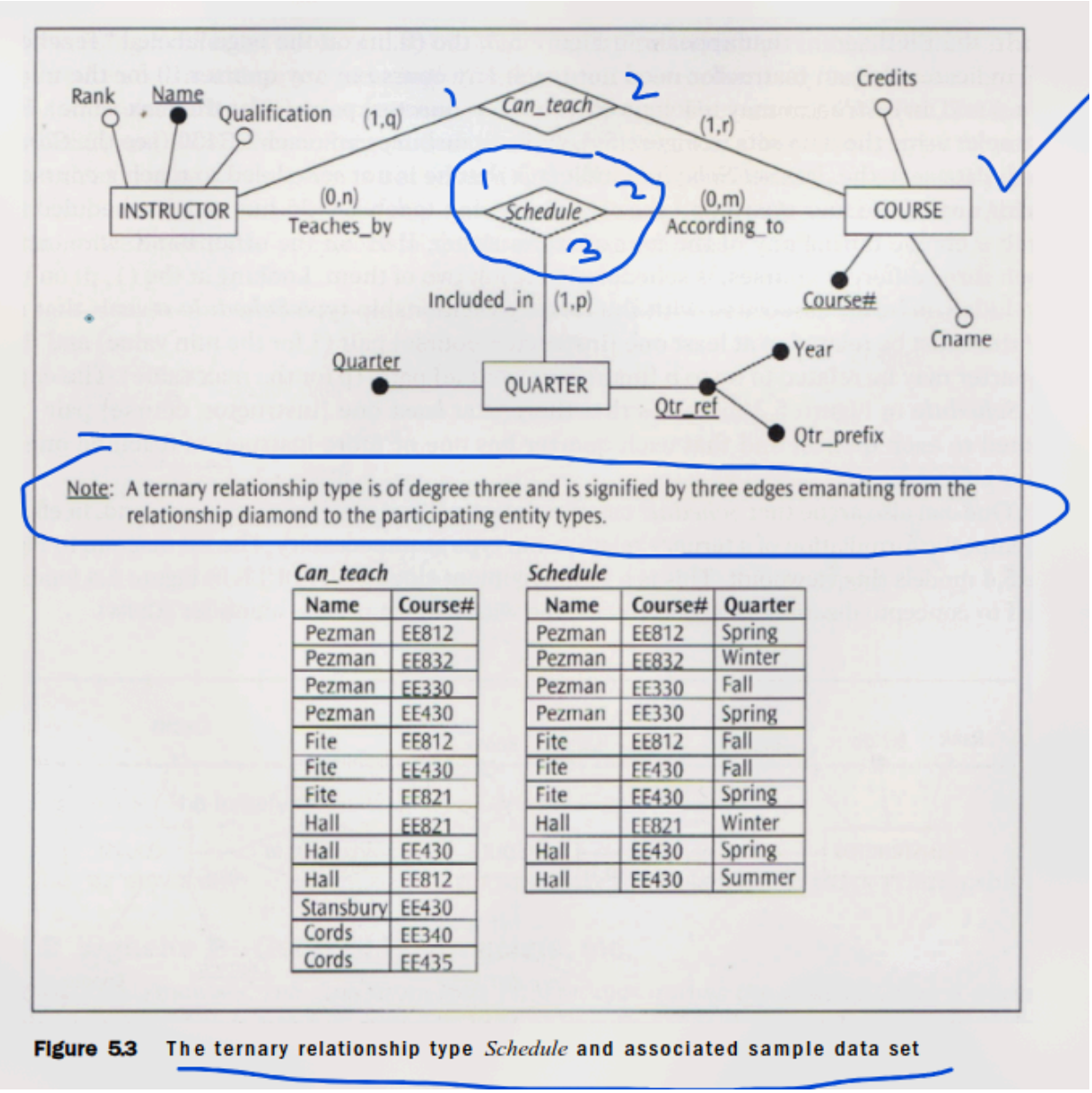
participation, the category is partial (proper subset of the union).

- Often used when the superclasses are of different types but share a common role represented by the category (e.g., `CAR_OWNER` as a category of `PERSON`, `COMPANY`, `BANK`).

- **Choosing the Appropriate EER Construct (Specialization vs. Categorization):** (p.139-144, *Fig 4.13-4.16*)
  - If subgroups are subsets of a *single* generic entity type and share common attributes of that generic type, use **Specialization/Generalization**. (e.g., `CAR`, `TRUCK`, `VAN` are all kinds of `VEHICLE`).
  - If a collection of entities comes from *multiple, different* entity types but they share a common role or are treated uniformly in some context, use **Categorization**. (e.g., `DONOR` can be an `INDIVIDUAL`, `COMPANY`, or `FOUNDATION`).
  - Specialization implies that every instance of a subclass *is an* instance of the superclass.
  - Categorization implies that an instance of the category *is an* instance of *one of* its superclasses.

- **Modeling Complex Relationships (Ch 5) [PYQ Q3b]**
  Chapter 5 delves into relationships that go beyond simple binary associations, including n-ary relationships and dependencies between relationships themselves.
  - **Ternary Relationship Type (and beyond):** (p.164-169, p.171-179)
    - As defined earlier, involves three or more entity types.
    - The **(min, max) notation** ("look here" approach) is essential for accurately specifying structural constraints for ternary and higher-degree relationships because the simple "look across" ratio notation is ambiguous.
    - A (min,max) pair on an edge connecting an entity type E to an n-ary relationship R means that each instance of E participates in 'min' to 'max' combinations of instances from the *other n-1* participating entity types.
    - Sometimes, an n-ary relationship can be re-conceptualized as a central entity type (often a gerund or a weak entity) with binary relationships to the original participating entities. This is particularly true if the n-ary relationship itself has attributes or participates in other relationships. *(Fig 5.4, p.168 - `SCHEDULE` ternary as a base entity; Fig 5.7, p.171 - `Prescribes` ternary as `PRESCRIPTION` base entity).*

(Fig 5.1-5.7, pp.164-169 show examples of ternary relationships and their semantics).



Can_teach

Name	Course#
Pezman	EE812
Pezman	EE832
Pezman	EE330
Pezman	EE430
Fite	EE812
Fite	EE430
Fite	EE821
Hall	EE821
Hall	EE430
Hall	EE812
Stansbury	EE430
Cords	EE340
Cords	EE435

Schedule

Name	Course#	Quarter
Pezman	EE812	Spring
Pezman	EE832	Winter
Pezman	EE330	Fall
Pezman	EE330	Spring
Fite	EE812	Fall
Fite	EE430	Fall
Fite	EE430	Spring
Hall	EE821	Winter
Hall	EE430	Spring
Hall	EE430	Summer

Figure 5.3 The ternary relationship type *Schedule* and associated sample data set

- *Fig 5.3 shows the ternary relationship* `Schedule` *among* `INSTRUCTOR`, `COURSE`, *and* `QUARTER` *using (min,max) notation. (0,n) on* `INSTRUCTOR` *means an instructor teaches 0 to n (Course, Quarter) pairs.*

- **Cluster Entity Type (Entity Clustering):** (p.171-179)
  - An abstraction where a group of related entity types and their inter-relationships are "rolled up" or consolidated to form a higher-level conceptual entity called a cluster entity type.
  - Useful for managing complexity in large ERDs by presenting them in layers. The cluster entity hides internal details at a higher level of abstraction.
  - The cluster entity itself can then participate in relationships with other entities (either basic or other cluster entities).
  - Example: (`ADVISOR` --Advising-- `STUDENT` --for-- `MAJOR`) where the ternary `Advising` could be part of a cluster. Or, `ADVISOR` and `STUDENT` could form a `COUNSELING` cluster entity which is then related to `MAJOR`.
    (Fig 5.10, p.176 - `COUNSELING` *as a cluster of (*`ADVISOR`, `STUDENT`*) related to* `MAJOR`*; Fig 5.12, p.178 - shows both* `COUNSELING` *and* `ENROLLMENT` *as cluster entities for Madeira*

College example).

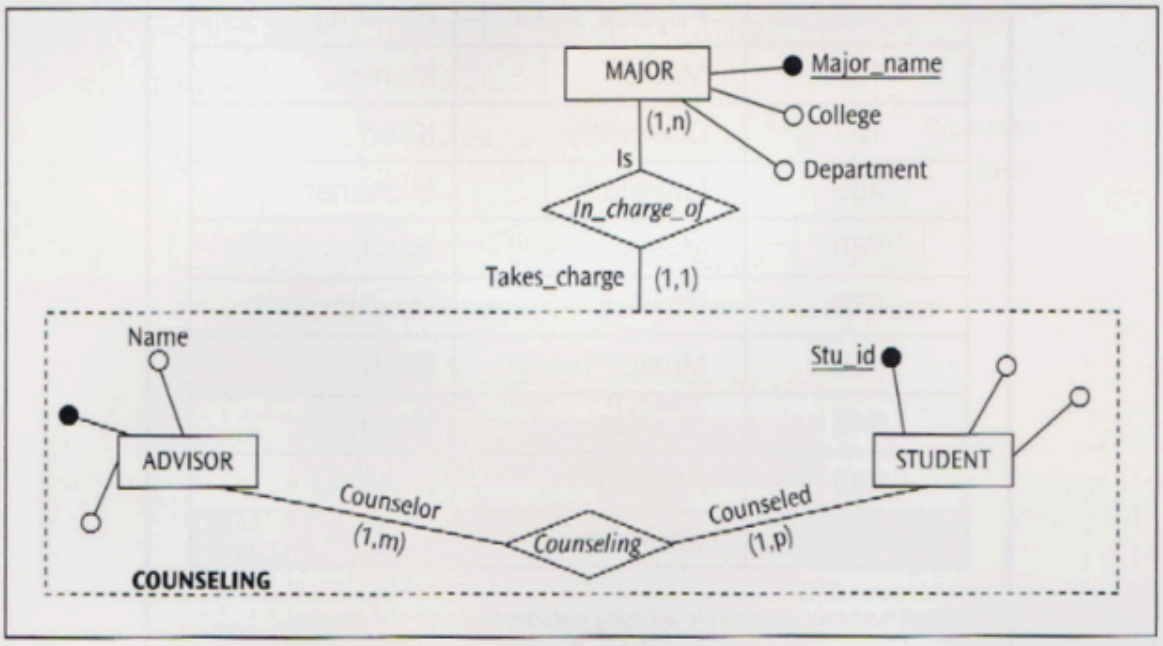


Figure 5.10 The relationship type *In_charge_of* in the cluster entity type COUNSELING

- *Fig 5.17 shows* `SECTION` *as a cluster entity (implicitly formed from* `Offering` *relationship involving* `INSTRUCTOR`, `COURSE`, `QUARTER`, `TIME_SLOT`, `CLASS_ROOM`*) reducing an n-ary relationship concept for clarity.*

- **Weak Relationship Type (Inter-relationship Integrity Constraint):** (p.190-197)
  This construct, introduced by Dey, Storey, and Barron (1999), defines a dependency where the existence of instances in one relationship set (the weak relationship) depends on the existence of corresponding instances in another relationship set (the precedent relationship). Denoted by a double diamond (like an identifying relationship).
  (Fig 5.28-5.33, pp.191-195)

  - **Inclusion Dependency (Subset Relationship):** One relationship set R2 must be a subset of another relationship set R1 (R2 $\subseteq$ R1). Indicated by a solid arrow from the dependent (weak) relationship R2 to the precedent relationship R1.

    - Example: `Manages` $\subseteq$ `Works_in` (An employee must work in a plant to manage that plant). *(Fig 5.28, p.191)*
    - Example: `Teaches` $\subseteq$ `Can_Teach` (An instructor must be capable of teaching a course to actually teach it). *(Fig 5.29, p.192)*
    - This can be **condition-precedent** (based on meeting a condition) or **event-precedent** (based on occurrence of an event).

  - **Exclusion Dependency (Mutual Exclusion):** Two relationship sets R1 and R2 are mutually exclusive (an entity pair cannot participate in both). Indicated by a non-directional dotted line connecting the two weak relationship diamonds. This is conceptually equivalent to the "exclusive arc" construct.

    - Example: If a `BCU_ACCOUNT` can be `Held_by_E` (employee) OR `Held_by_D` (dependent), but not jointly, these two relationships are mutually exclusive. *(Fig 5.32 and*

5.33, p.194-195)

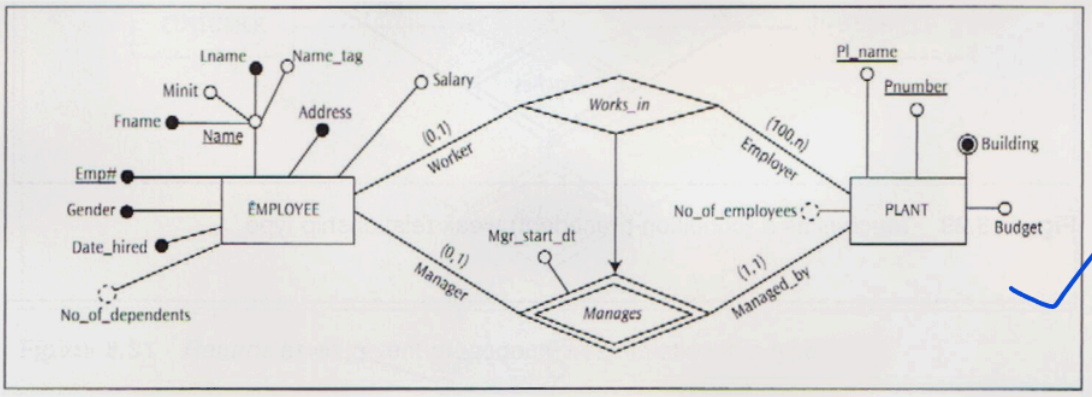


Figure 5.28 *Manages* as a (condition-precedent) weak relationship type

- *Fig 5.28 shows* `Manages` *as a weak relationship, dependent on (subset of)* `Works_in`.

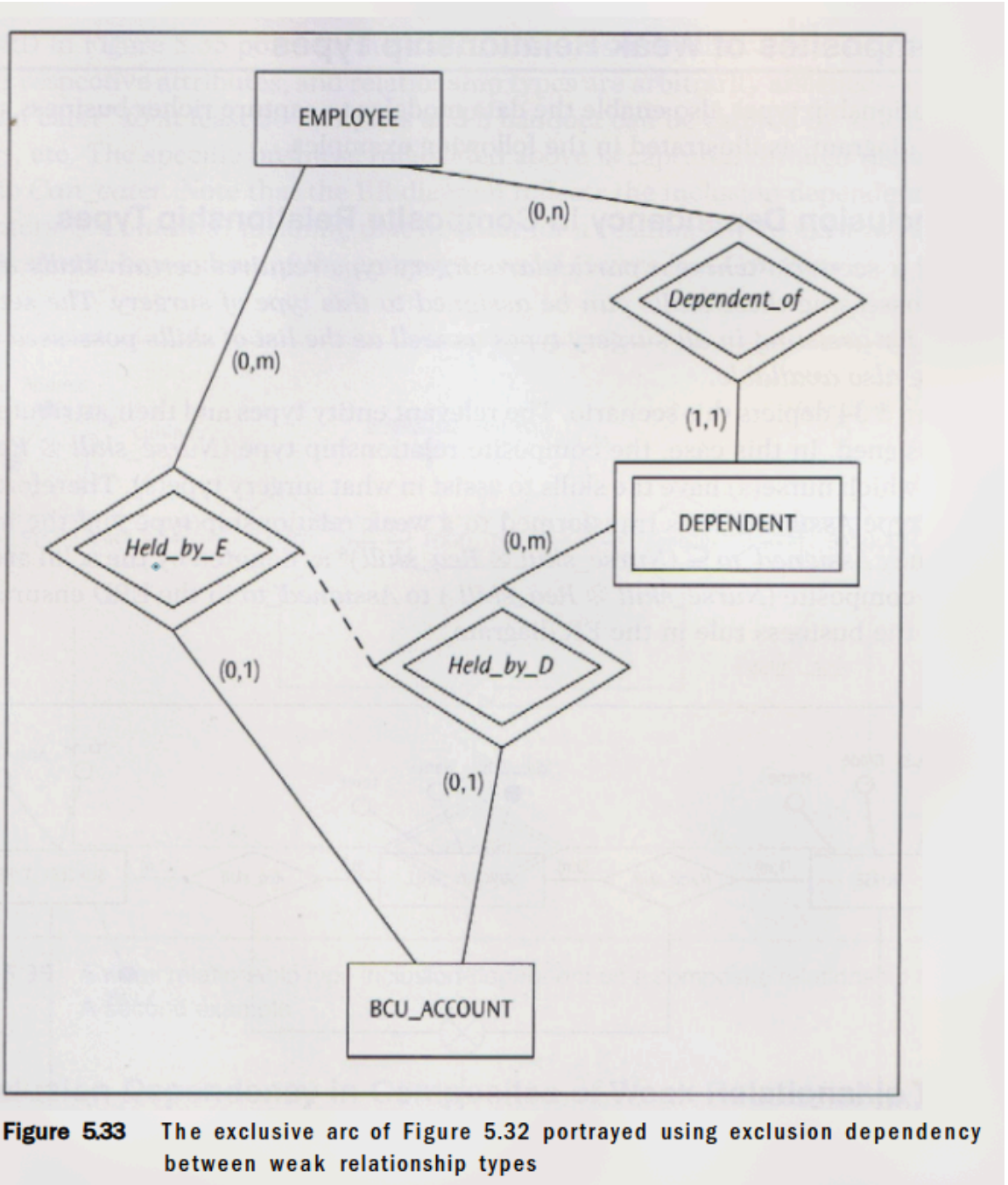


Figure 5.33 The exclusive arc of Figure 5.32 portrayed using exclusion dependency between weak relationship types

        - *Fig 5.33 shows `Held_by_E` and `Held_by_D` as mutually exclusive weak relationships using a dotted line between them.*

    - **Composites of Weak Relationship Types:** (p.197-198)
      Weak relationship types can also depend on or be excluded by a *composite* of other (regular or weak) relationship types.

        - **Inclusion Dependency on Composite:** e.g., `Assigned_to` ⊆ (`Nurse_skill` ⊗ `Req_skill`), meaning a nurse can be assigned to a surgery type only if the nurse possesses the skills required by that surgery type. (⊗ represents a conceptual join or composition). *(Fig 5.34, p.197)*
        - **Exclusion Dependency with Composite:** e.g., `Conflict_of_Interest` is mutually exclusive with (`Writes` ⊗ `Referees`), meaning a professor cannot be assigned to referee a paper if they are an author of it (or vice-versa, if `Conflict_of_Interest` captures this). *(Fig 5.37, p.198)*
          (Fig 5.34-5.37, pp.197-198)

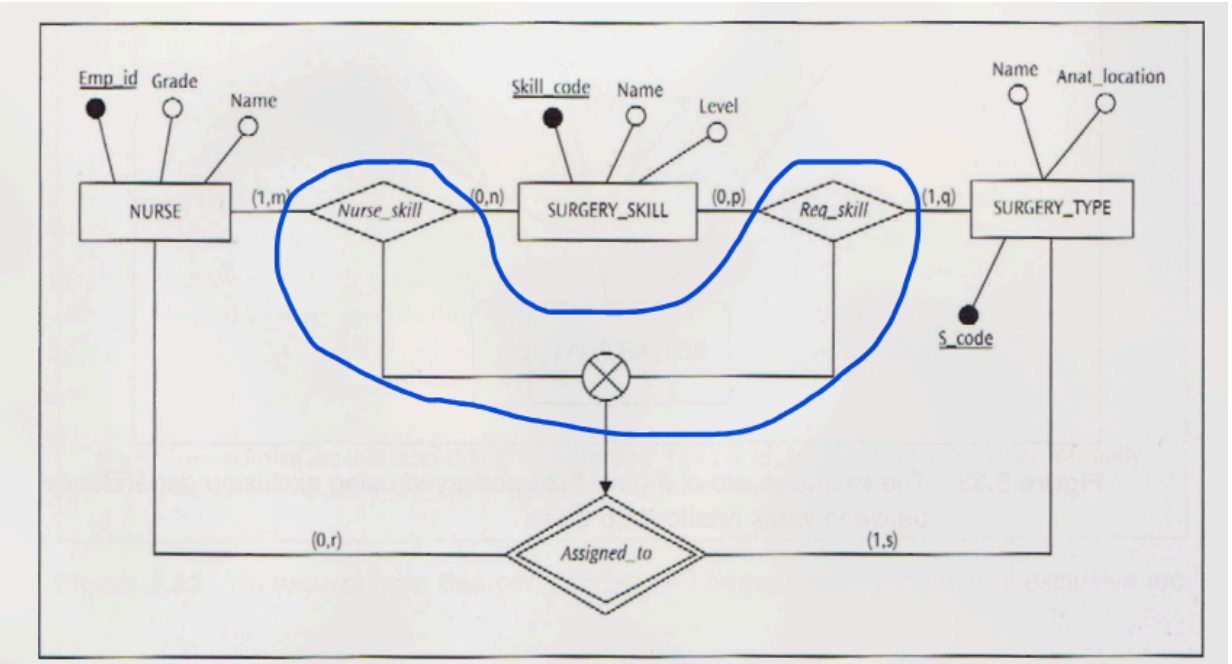


Figure 5.34 A weak relationship type inclusion-dependent on a composite relationship type

        - *Fig 5.34 shows weak relationship `Assigned_to` being inclusion-dependent on a composite of `Nurse_skill` and `Req_skill`.*

- **Design Issues in ER & EER Modeling [PYQ Q3b]**
  Throughout conceptual modeling, designers face several choices:

    - **Entity vs. Attribute:** When should a concept be modeled as an attribute versus a separate entity type? (e.g., `Address` as a composite attribute vs. `ADDRESS` as an entity type). If the concept has attributes of its own or participates in relationships, it's likely an entity.
    - **Binary vs. Higher-Degree Relationships:** Can a ternary/n-ary relationship be accurately represented by a set of binary ones? If not, an n-ary relationship is needed. (Refer to ternary examples in Ch 5).

- **Use of Weak Entities:** When is a weak entity appropriate versus a strong entity with a regular relationship? If an entity cannot be identified without its owner and has a natural partial key.
- **When to use EER Constructs:**
  - **Specialization/Generalization:** When there are clear subgroupings within a single entity type that have unique attributes/relationships, and an "is-a" relationship holds.
  - **Categorization:** When a subclass is formed by the union of different entity types sharing a common role.
- **Handling M:N Relationships and Multi-valued Attributes:** These are valid in conceptual models (especially Presentation Layer and Coarse Design-Specific) but must be decomposed in the Fine-Granular Design-Specific ER model before mapping to relational databases.
- **Validation of Conceptual Design (Connection Traps):** (p.209-216)
  Crucial for ensuring semantic completeness and avoiding misinterpretations. Connection traps arise from ambiguous pathways in the ERD.
  - **Fan Trap:** Occurs when an ERD structure allows a many-to-one relationship pathway that incorrectly suggests a direct many-to-many relationship between two entities that are not directly connected, or when ambiguity arises from a "fan-out" structure. Typically involves a central entity with two or more 1:N relationships fanning out from it. It obscures which instances on one "spoke" of the fan relate to instances on another "spoke." *(Fig 5.48, p.211;*

Resolution Fig 5.50, p.214)

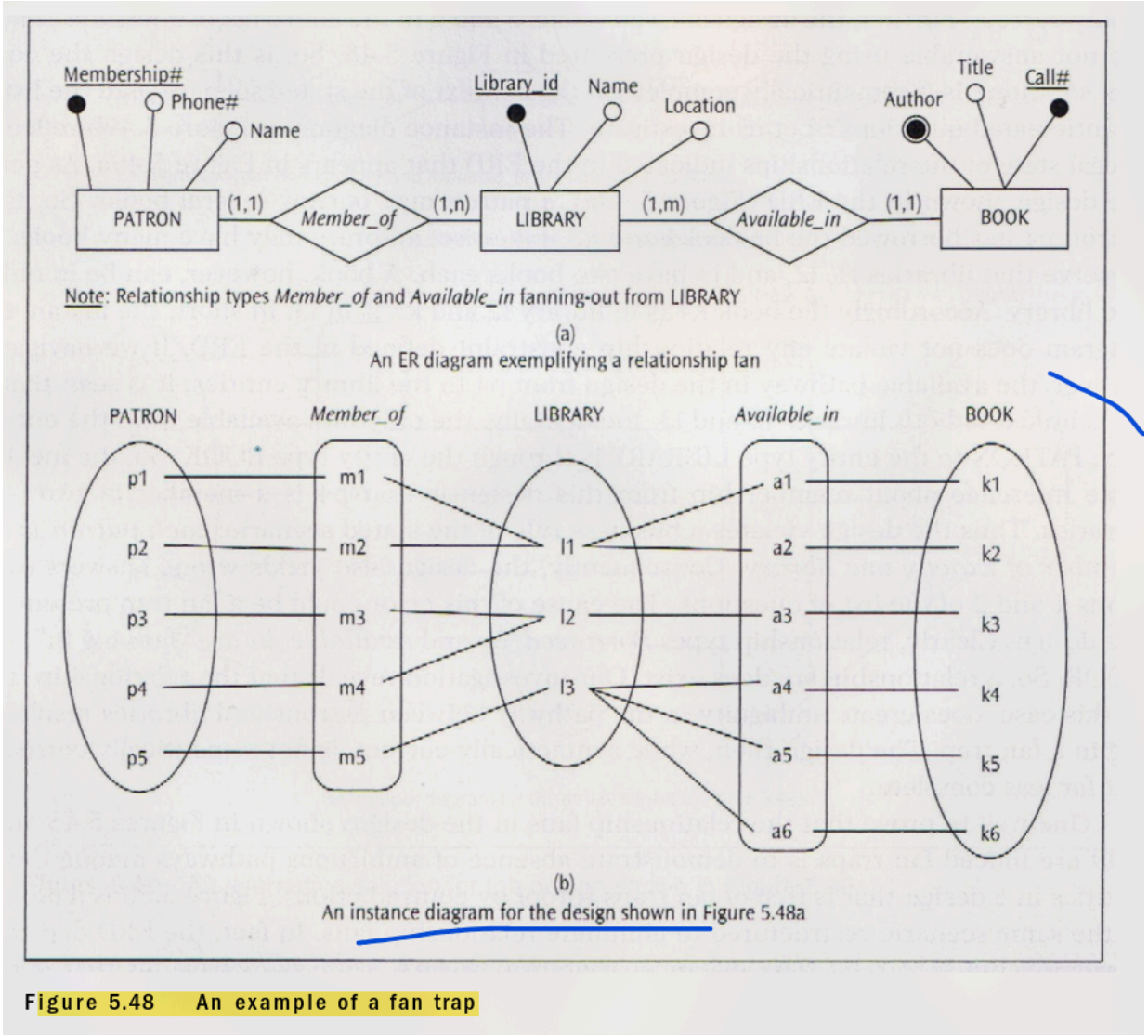


Figure 5.48 An example of a fan trap

- *Fig 5.48:* `LIBRARY` *has 1:N to* `PATRON` *(Member_of) and 1:N to* `BOOK` *(Available_in). It's ambiguous which patron borrowed which book if a patron can only borrow from their*

library.

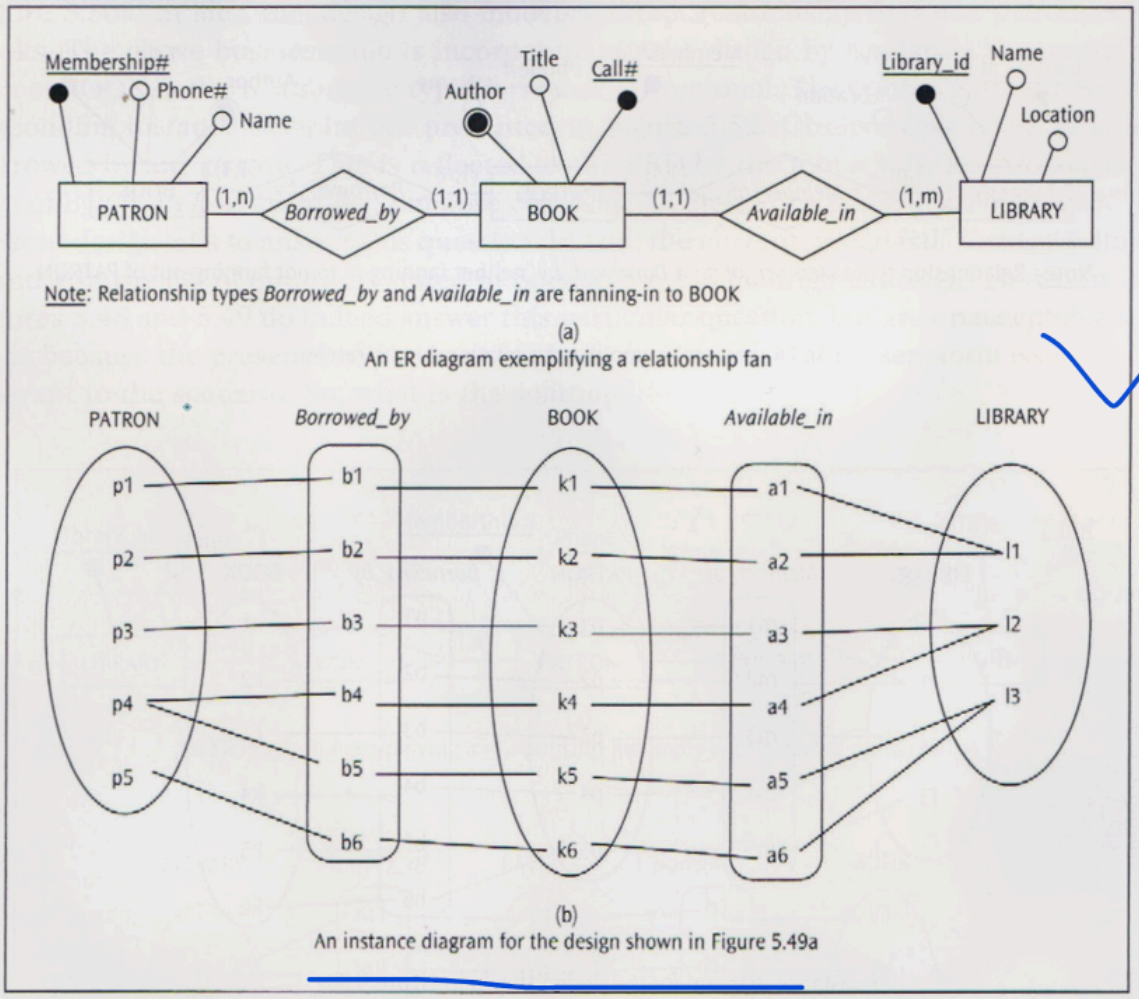


Figure 5.49 An alternative solution for the design shown in Figure 5.48

- *Fig 5.50 resolves the fan trap by directly relating* `PATRON` *and* `BOOK` *via* `Borrowed_by`*, and* `LIBRARY` *to* `PATRON` *and* `BOOK` *separately.*

- **Chasm Trap:** Occurs when a model suggests a pathway between entities, but the existence of optional participation (partial participation) along that pathway means that some instances cannot be related, creating a "chasm" or gap in information. It indicates that some queries

might not be answerable for all instances. *(Fig 5.52, p.215; Resolution Fig 5.53, p.217)*

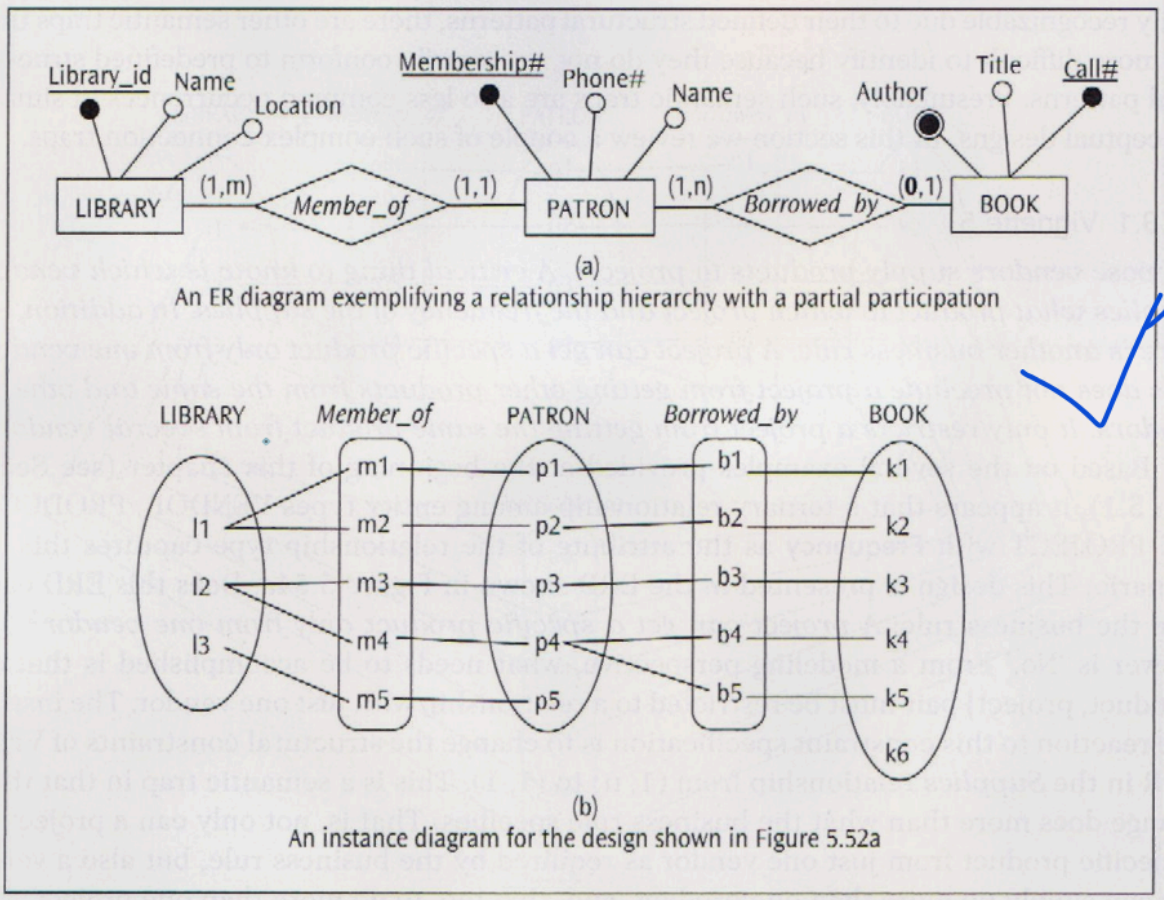


Figure 5.52 An example of a chasm trap

- *Fig 5.52: If `BOOK` has optional participation in `Borrowed_by` (min=0, meaning some books aren't borrowed), and `LIBRARY` is only related to `BOOK` via `PATRON` and `Borrowed_by`, we can't find which library holds an unborrowed book.*

- **Resolving Traps:** Often involves restructuring the ERD by adding direct relationships, changing cardinalities, or re-evaluating the business rules. Validation with users and sample data is key. Not all structural "fans" or optional participations lead to semantic traps; it depends on the queries intended to be supported.

UNIT - 1 Completed, Remember to actively draw the diagrams and work through the examples in your textbook to solidify your understanding. Good luck for your exams!!!!