This image cannot currently be displayed.

Chapter 4

Software Requirement Specification

Declaration

- These slides are made for UIT, BU students only. I am not holding any copy write of it as I had collected these study materials from different books and websites etc. I have not mentioned those to avoid complexity.

Topics as per Syllabus

Software Requirement Specification:

- Problem analysis, Requirement Specification, Requirement Types, Requirement Gathering Techniques, feasibility Study Validation, metrics, Use Case diagram, ER Diagram.

Requirement Engineering

Requirements describe

What **not** **How**

Produces one large document written in natural language contains a description of what the system will do without describing how it will do it.

- Without well written document
  - Developers do not know what to build
  - Customers do not know what to expect
  - What to validate

Requirement Engineering

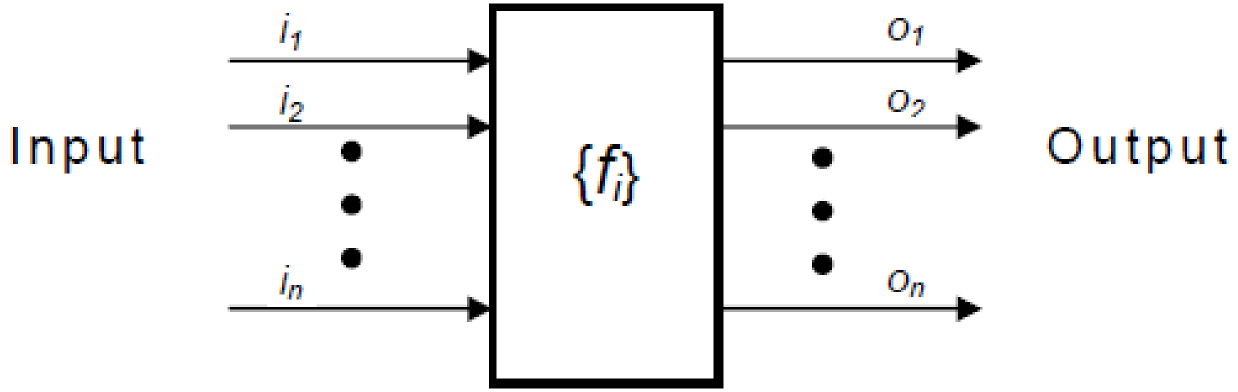
- Requirement Engineering is the disciplined application of proven principles, methods, tools, and notations to describe a proposed system's intended behavior and its associated constraints.
- SRS may act as a contract between developer and customer.
  - Requirements are difficult to uncover
  - Requirements change
  - Over reliance on CASE Tools
  - Tight project Schedule
  - Communication barriers
  - Market driven software development
  - Lack of resources

Parts of a SRS document

- The important parts of SRS document are:
  - Functional requirements of the system
  - Non-functional requirements of the system, and
  - Goals of implementation

Functional requirements

The functional requirements part discusses the functionalities required from the system. The system is considered to perform a set of high-level functions {f_i}. The functional view of the system is shown in figure Each function f_i of the system can be considered as a transformation of a set of input data (ii) to the corresponding set of output data (o_i). The user can get some meaningful piece of work done using a high-level function.



Non-functional requirements

Non-functional requirements deal with the characteristics of the system which can not be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc.

Goals of implementation

The goals of implementation documents some general suggestions regarding development. These suggestions guide trade-off among design goals. The goals of implementation section might document issues such as revisions to the system functionalities that may be required in the future, new devices to be supported in the future, reusability issues, etc. These are the items which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately.

Requirement Gathering Techniques

- Brainstorming

Brainstorming is used in requirement gathering to get as many ideas as possible from group of people. Generally used to identify possible solutions to problems, and clarify details of opportunities.

- Document Analysis

Reviewing the documentation of an existing system can help when creating AS - IS process document, as well as driving gap analysis for scoping of migration projects. In an ideal world, we would even be reviewing the requirements that drove creation of the existing system – a starting point for documenting current requirements. Nuggets of information are often buried in existing documents that help us ask questions as part of validating requirement completeness.

Requirement Gathering Techniques

- Focus Group

A focus group is a gathering of people who are representative of the users or customers of a product to get feedback. The feedback can be gathered about needs / opportunities / problems to identify requirements, or can be gathered to validate and refine already elicited requirements. This form of market research is distinct from brainstorming in that it is a managed process with specific participants.

- Interface analysis

Interfaces for a software product can be human or machine. Integration with external systems and devices is just another interface. User centric design approaches are very effective at making sure that we create usable software. Interface analysis – reviewing the touch points with other external systems is important to make sure we don't overlook requirements that aren't immediately visible to users.

Requirement Gathering Techniques

- Interview

Interviews of stakeholders and users are critical to creating the great software. Without understanding the goals and expectations of the users and stakeholders, we are very unlikely to satisfy them. We also have to recognize the perspective of each interviewee, so that, we can properly weigh and address their inputs. Listening is the skill that helps a great analyst to get more value from an interview than an average analyst.

- Observation

By observing users, an analyst can identify a process flow, steps, pain points and opportunities for improvement. Observations can be passive or active (asking questions while observing). Passive observation is better for getting feedback on a prototype (to refine requirements), where active observation is more effective at getting an understanding of an existing business process. Either approach can be used.

Requirement Gathering Techniques

- Prototyping

Prototyping is a relatively modern technique for gathering requirements. In this approach, you gather preliminary requirements that you use to build an initial version of the solution - a prototype. You show this to the client, who then gives you additional requirements. You change the application and cycle around with the client again. This repetitive process continues until the product meets the critical mass of business needs or for an agreed number of iterations.

- Requirement Workshops

Workshops can be very effective for gathering requirements. More structured than a brainstorming session, involved parties collaborate to document requirements. One way to capture the collaboration is with creation of domain-model artifacts (like static diagrams, activity diagrams).

Requirement Gathering Techniques

- Reverse Engineering

When a migration project does not have access to sufficient documentation of the existing system, reverse engineering will identify what the system does. It will not identify what the system should do, and will not identify when the system does the wrong thing.

- Survey/Questionnaire

When collecting information from many people – too many to interview with budget and time constraints – a survey or questionnaire can be used. The survey can force users to select from choices, rate something ("Agree Strongly, agree…"), or have open ended questions allowing free-form responses. Survey design is hard – questions can bias the respondents.

Feasibility Study

- **Is cancellation of a project a bad news?**

As per IBM report,

- 31% projects get cancelled before they are completed,
- 53% over-run their cost estimates by an average of 189%
- For every 100 projects, there are 94 restarts.

How do we cancel a project with the least work?

CONDUCT A FEASIBILTY STUDY

Feasibility Study

- Technical feasibility
  - Is it technically feasible to provide direct communication connectivity through space from one location of globe to another location?
- Feasibility depends upon non technical Issues like:
  - Are the project’s cost and schedule assumption realistic?
  - Does the business model realistic?
  - Is there any market for the product?

Feasibility Study

- Purpose of feasibility study

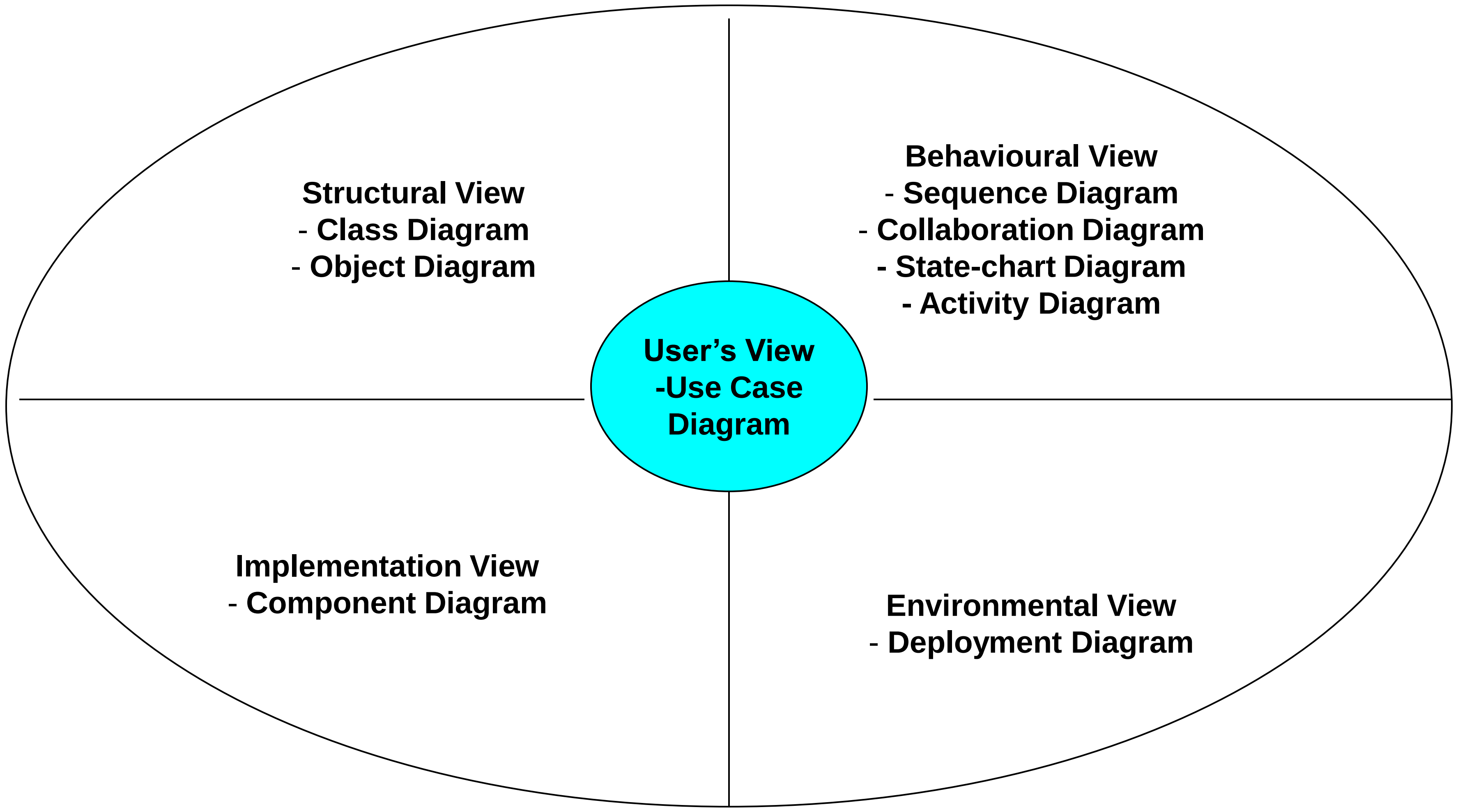
"evaluation or analysis of the potential impact of a proposed project or program."

- Focus of feasibility studies
  - Is the product concept viable?
  - Will it be possible to develop a product that matches the project's vision statement?
  - What are the current estimated cost and schedule for the project?
  - How big is the gap between the original cost & schedule targets & current estimates?
  - Is the business model for software justified when the current cost & schedule estimate are considered?
  - Have the major risks to the project been identified & can they be surmounted?

Feasibility Study

- Is the specifications complete & stable enough to support remaining development work?
- Have users & developers been able to agree on a detailed user interface prototype? If not, are the requirements really stable?
- Is the software development plan complete & adequate to support further development work?

UML Diagrams



Diagrams and views in UML

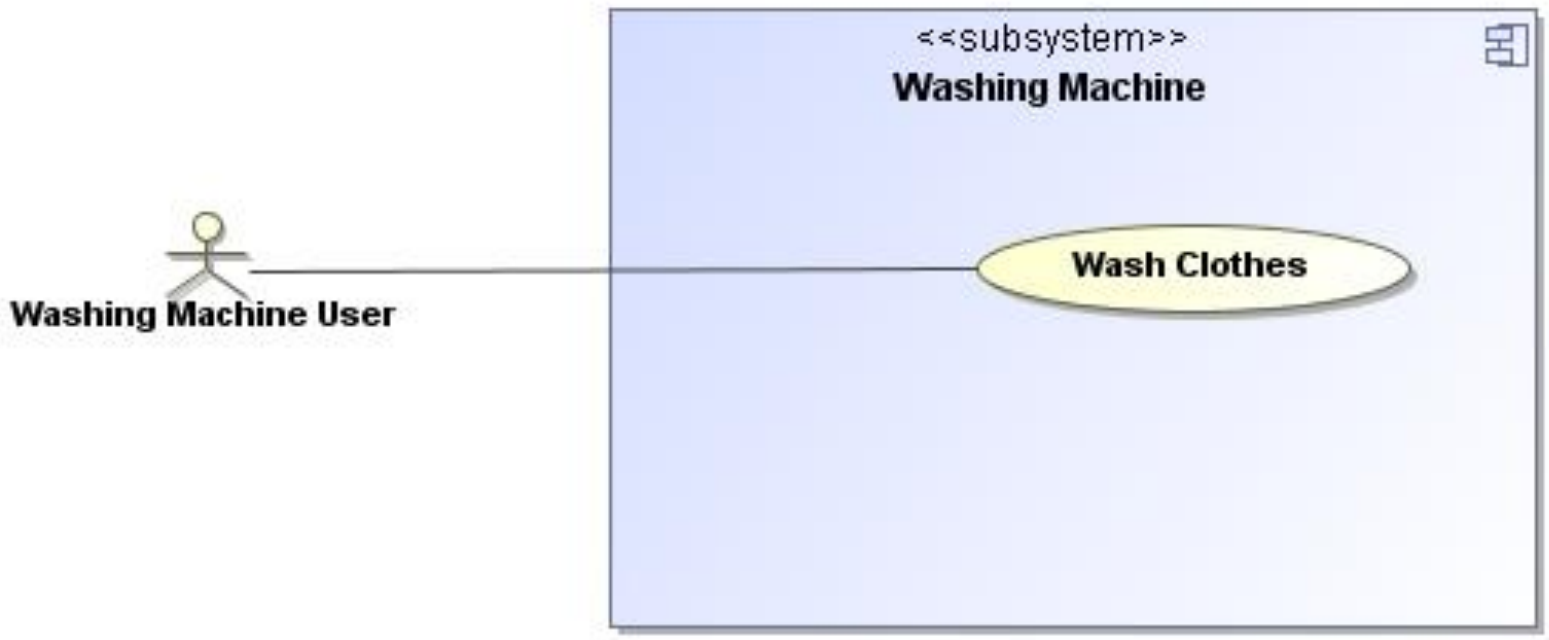
Use Case Diagram

- A Use Case Model describes the proposed functionality of a new system.
- A Use Case represents a discrete unit of interaction between a user (human or machine) and the system.
- This interaction is a single unit of meaningful work.
- Create Account
- View Account Details.

Representation of Use Cases

- **A use case** is represented by an **ellipse**
- **System boundary** is represented by a **rectangle**
- **Users** are represented by **stick person** icons (**actor**)
- **Communication relationship** between actor and use case by a **line**
- **External system** by a **stereotype**

An Use Case Diagram



Use Cases

- Different ways in which a system can be used by the users
- Corresponds to the high-level requirements
- Represents transaction between the user and the system
- Defines external behavior without revealing internal structure of system
- Set of related scenarios tied together by a common goal

Use Cases cont…

- Normally, use cases are independent of each other
- Implicit dependencies may exist

Example: In Library Automation System,

renew-book & reserve-book are independent use cases.

But in actual implementation of renew-book, a check is made to see if any book has been reserved using reserve-book.

Example of Use Cases

- For library information system
  - issue-book
  - query-book
  - return-book
  - create-member
  - add-book, etc.

Components

- **Actors**: An actor portrays any entity (or entities) that performs certain roles in a given system.
- **Use case:** A use case in a use case diagram is a visual representation of a distinct business functionality in a system.
- **System boundary:** A system boundary defines the scope of what a system will be.

Components cont…

- **Include:** When a use case is depicted as using the functionality of another use case in a diagram, this relationship between the use cases is named as an *include* relationship.
- **Extend:** In an *extend* relationship between two use cases, the child use case adds to the existing functionality and characteristics of the parent use case.
- **Generalizations:** A *generalization* relationship is also a parent-child relationship between use cases.

Components cont…

- General comments and notes describing the use case.
- Requirements - The formal functional requirements of things that a Use Case must provide to the end user.

- ability to update order

- These correspond to the functional specifications found in structured methodologies.
- Form a contract that the Use Case performs some action.

Components cont…

- Constraints :The formal rules and limitations a Use Case operates under, defining what can and cannot be done.

1. Pre-conditions that must have already occurred or be in place before the use case is run.

- <create order> must precede <modify order>

2. Post-conditions that must be true once the Use Case is complete;

-<order is modified and consistent>

3. Invariants that must always be true throughout the time the Use Case operates;

- an order must always have a customer number.

Components cont…

- Scenarios:Formal, sequential descriptions of the steps taken to carry out the use case
- Scenario diagrams:Sequence diagrams to depict the workflow
- Additional attributes:

- implementation phase
- version number
- complexity rating
- Stereotype
- status.

Coverage of Use Case

- A use case specification document should cover the following areas:
- **Actors:** List the actors that interact and participate in this use case.
- **Pre-conditions:** Pre-conditions that need to be satisfied for the use case to perform.
- **Post-conditions:** Define the different states in which you expect the system to be in, after the use case executes.

Coverage of Use Case cont...

- **Basic Flow:** List the basic events that will occur when this use case is executed. Include all the primary activities that the use case will perform. Be fairly descriptive when defining the actions performed by the actor and the response of the use case to those actions. This description of actions and responses are your functional requirements. These will form the basis for writing the test case scenarios for the system.
- **Alternative flows:** Any subsidiary events that can occur in the use case should be listed separately. Each such event should be completed in itself to be listed as an alternative flow. A use case can have as many alternative flows as required. But remember, if there are too many alternative flows, you need to revisit your use case design to make it simpler and, if required, break the use case into smaller discrete units.

Coverage of Use Case cont…

- **Special Requirements:** Business rules for the basic and alternative flows should be listed as special requirements in the use case narration. These business rules will also be used for writing test cases. Both success and failure scenarios should be described here.
- **Use case relationships:** For complex systems, it is recommended that you document the relationships between use cases. If this use case extends from other use cases or includes the functionality of other use cases, these relationships should be listed here. Listing the relationships between use cases also provides a mechanism for traceability.

Courseware Management System

- The organization offers a variety of courses in a variety of areas such as learning management techniques and understanding different software languages and technologies. Each course is made up of a set of topics. Tutors in the organization are assigned courses to teach according to the area that they specialize in and their availability. The organization publishes and maintains a calendar of the different courses and the assigned tutors every year. There is a group of course administrators in the organization who manage the courses including course content, assign courses to tutors, and define the course schedule. The training organization aims to use the Courseware Management System to get a better control and visibility to the management of courses as also to streamline the process of generating and managing the schedule of the different courses.

Terms and entities

- Courses and Topics that make up a course
- Tutors who teach courses
- Course administrators who mange the assignment of the courses to tutors
- Calendar or Course Schedule is generated as a result
- Students who refer to the Course schedule or Calendar to decide which courses they wish to take up for study

Actors

- Tutors, Course administrators , Students

students are not the potential active participants for this system, we will drop them from the list of actors. Similarly, tutors are not active participants from our system's perspective, and hence, we will exclude tutors from our list if roles. Yet, we will still record them in our use case model since we do not wish to lose this business information. Our final list of primary actors has now come down to only one:

- Course administrators

Use Cases

- Manage courses
- Manage course assignments

Within the "Manage courses" use case, we can identify the following sub processes:

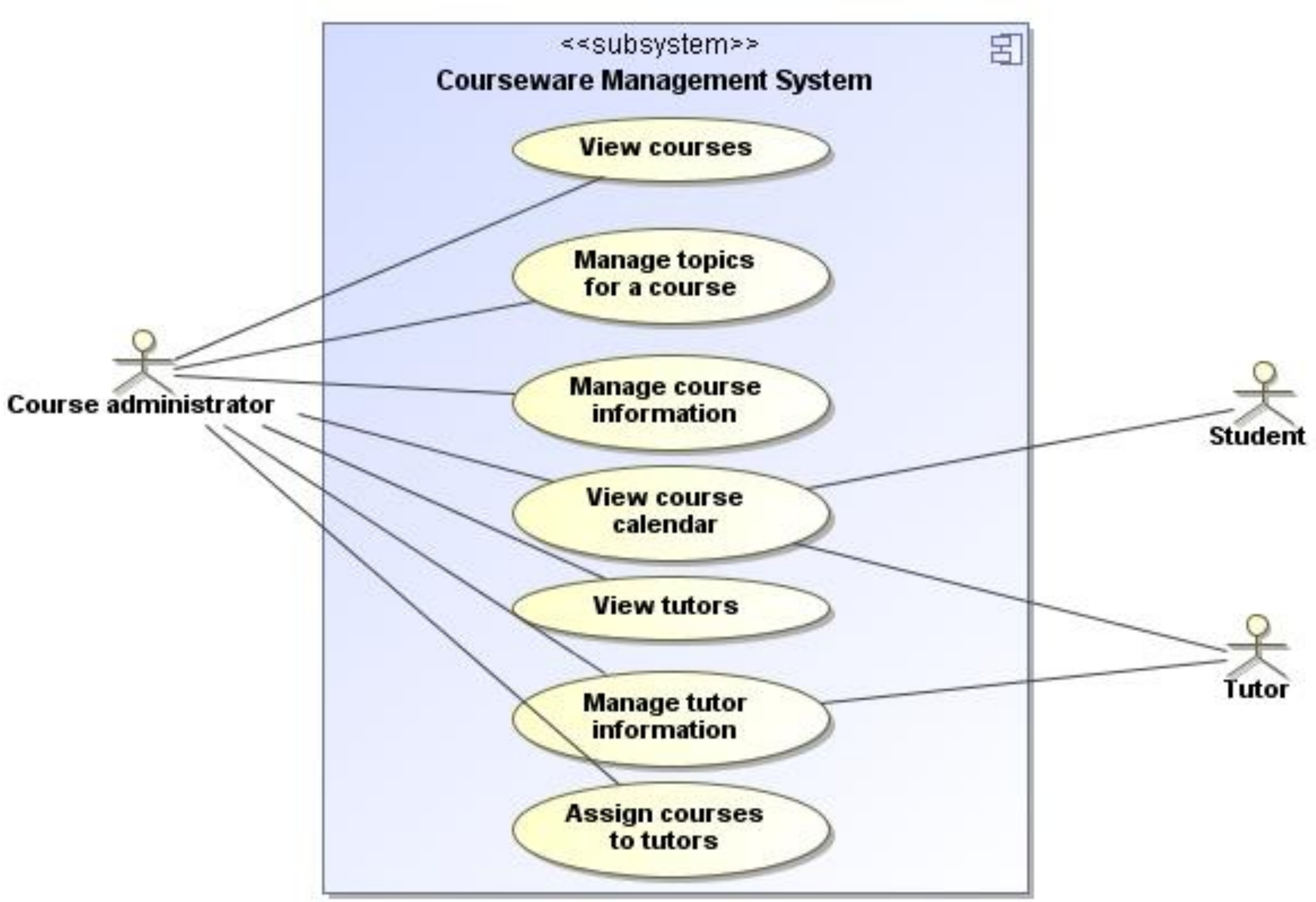
- View courses
- Manage topics for a course
- Manage course information

Use Cases cont…

Within the "Manage course assignment" use case are:

- View course calendar
- View tutors
- Manage tutor information
- Assign courses to tutors

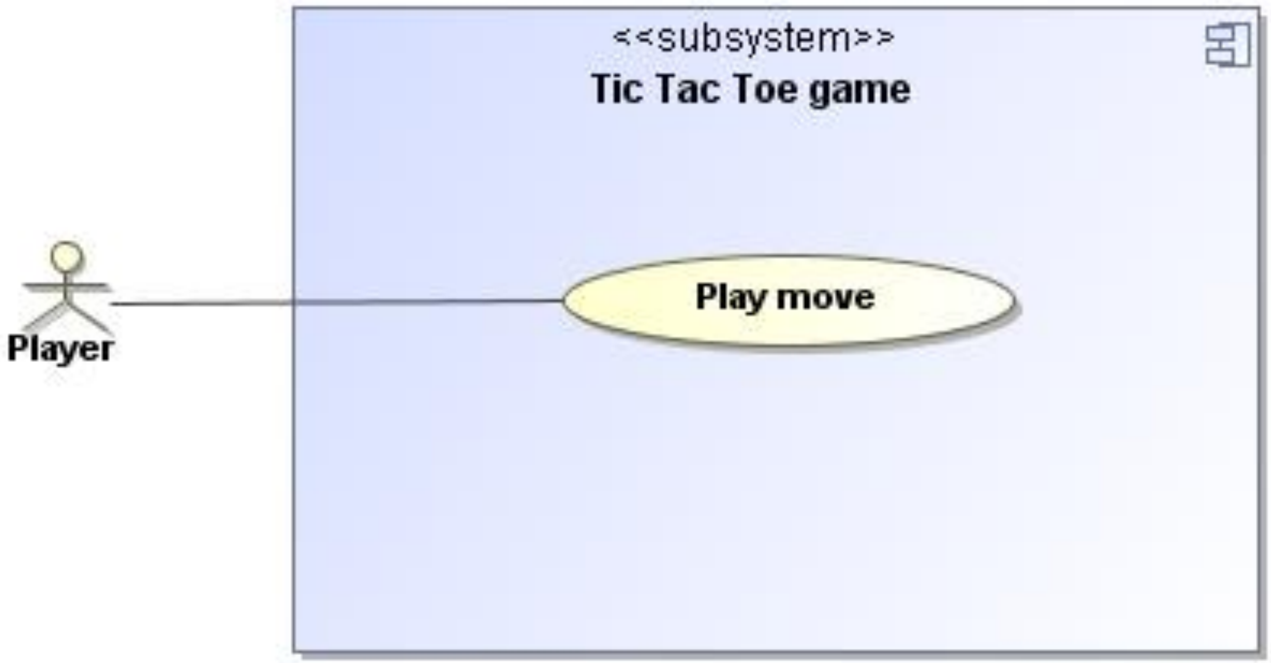
Use Case Diagram



Tic-Tac-Toe Computer Game:

- Tic-tac-toe is a computer game in which a human player and the computer make alternate moves on a 3 × 3 square. A move consists of marking a previously unmarked square. The player who is first to place three consecutive marks along a straight line (i.e., along a row, column, or diagonal) on the square wins. As soon as either of the human player or the computer wins, a message congratulating the winner should be displayed. If neither player manages to get three consecutive marks along a straight line, and all the squares on the board are filled up, then the game is drawn. The computer always tries to win a game.

An Example Use Case Diagram



Why Develop A Use Case Diagram?

- Serves as requirements specification
- How are actor identification useful in software development:

− User identification helps in implementing

- appropriate interfaces for different categories of users

− Another use in preparing appropriate documents (e.g. **user's manual**).

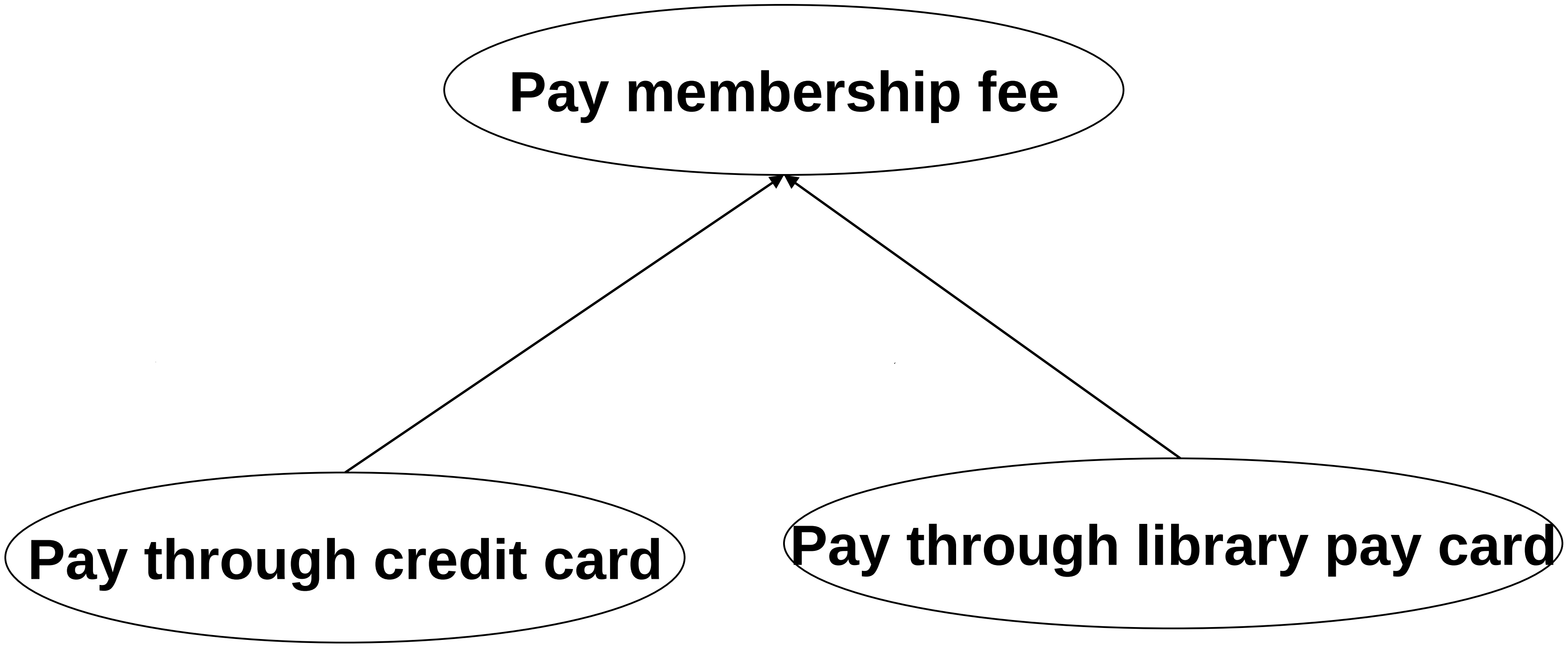
Factoring Use Cases

- Complex use cases need to be factored into simpler use cases
- Represent common behavior across different use cases
- Three ways of factoring:
  - **Generalization**
  - **Includes**
  - **Extends**

Generalization

- When one use case is slightly different from other.
- Works the same way with use cases as it does with classes.
- Child use case inherits the behavior and meaning of the present use case.

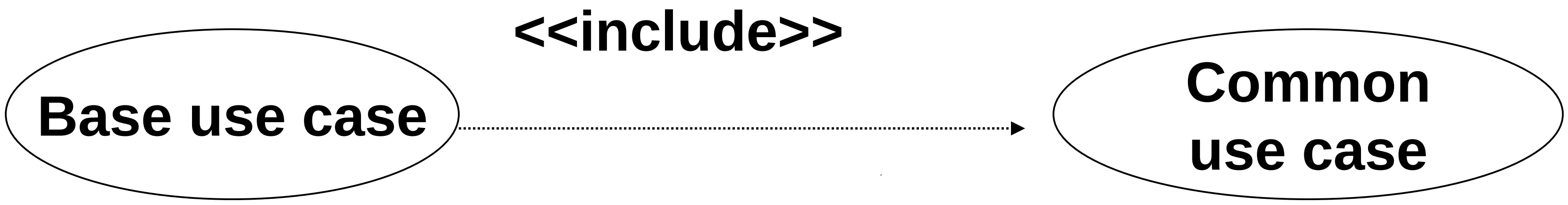
Factoring Use Cases Using Generalization



Includes

- Uses in the older versions of UML (prior to UML 1.1)
- One use case including the behavior of another use case in its sequence of events and actions.

Factoring Use Cases Using Includes



Personal Library System:

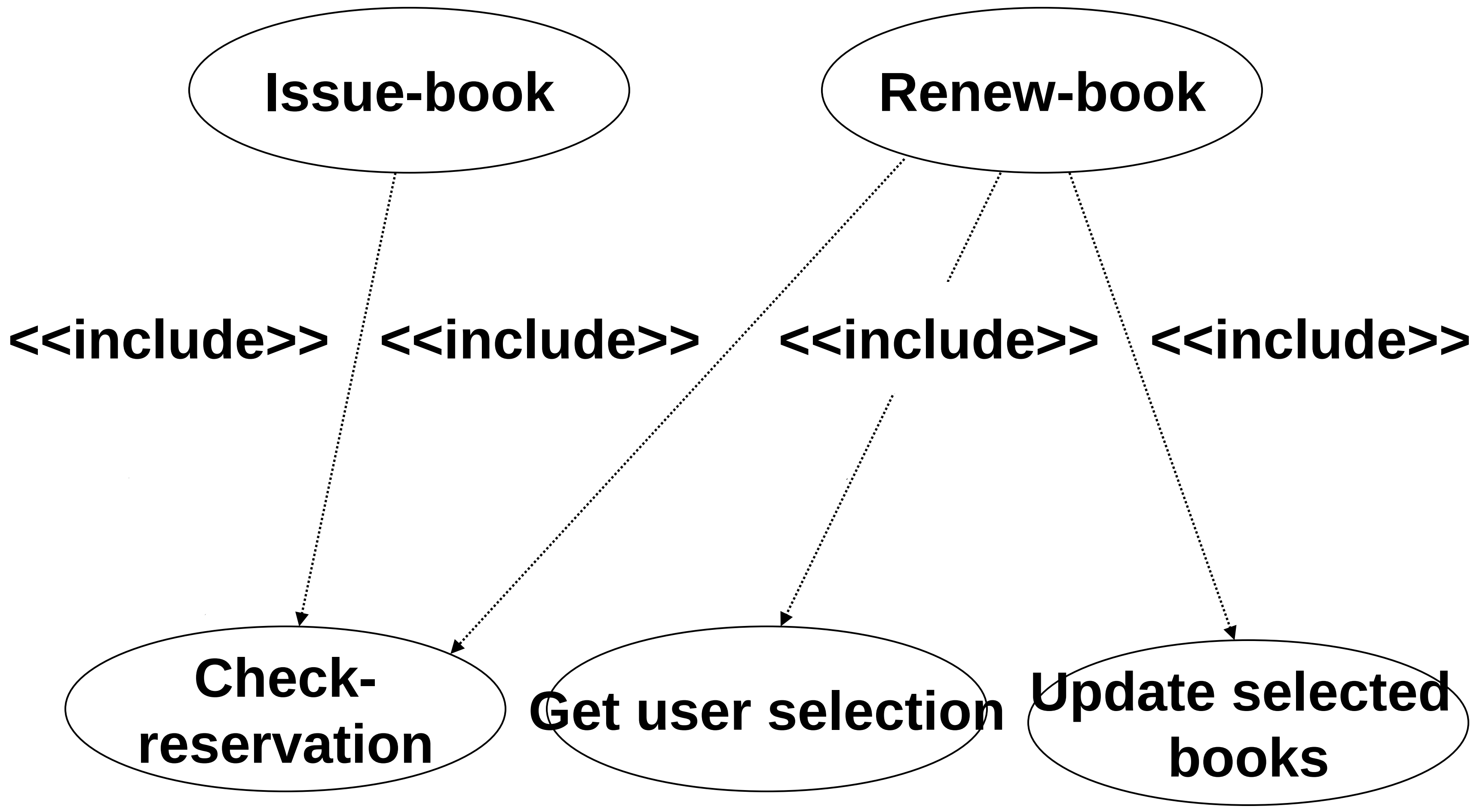
It is required to develop a software to manage the collection of books by individuals. A person can have a few hundreds of books. The details of all the books such as name of the book, year of publication, date of purchase, price, and publisher must be entered. A book is to be given a unique serial number by the computer which written by pen on the book. Before a friend can be lended a book, he must be registered. The registration data would include name of the friend, address, land line number, and mobile number. Before a friend is issued a book, the various books outstanding against him also with the date borrowed are displayed. The date of issue and the title of the book are stored. When a friend returns a book, the date of return is stored and the book is removed from his borrowing list. Up on query, the software should display the name, address, and telephone numbers of each friend against whom books are outstanding along with the titles of the outstanding books and the date issued.

Personal Library System: cont…

The owner of the library software, when he borrows books from his friends, enters the details regarding the title of books borrowed, and the date borrowed. It should be able display all the books borrowed from various friends.

It should be query about the availability a particular book, the total number of books in the personal library, and the total capital invested in the library.

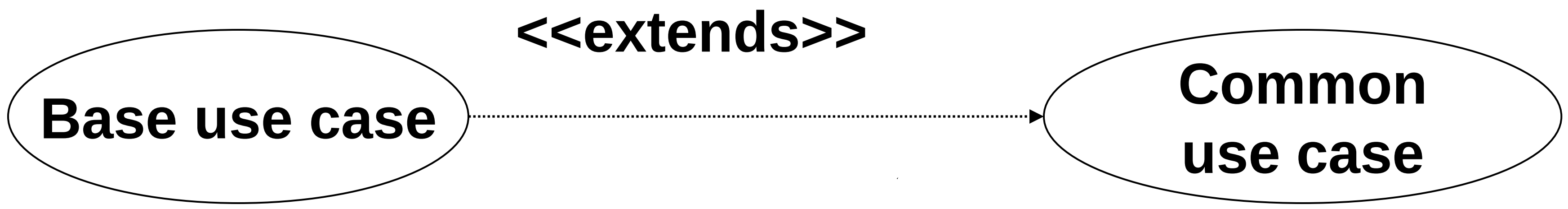
Factoring Use Cases Using Includes cont…



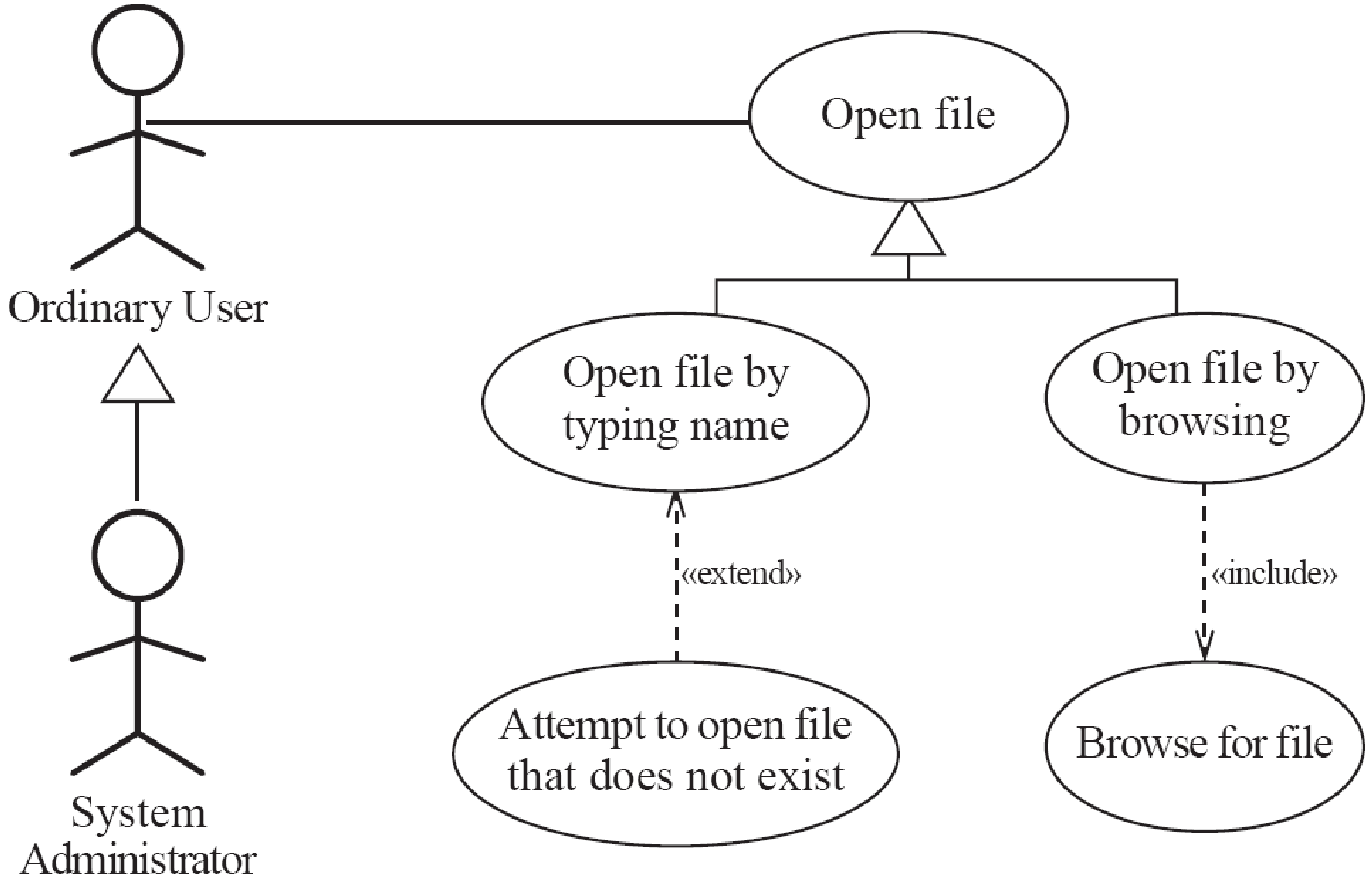
Extends

- It allows to show optional system behavior.
- An optional system behavior is executed only under certain conditions.
- Unlike generalization, the extend use case can add additional behavior only at an extension point when certain conditions specified.

Factoring Use Cases Using Extends



Example of generalization, extension and inclusion



Example description of a use case

Use case: **Open file**

Related use cases:
Generalization of:
- Open file by typing name
- Open file by browsing

Steps:

Actor actions	**System responses**
1. Choose 'Open…' command	2. File open dialog appears
3. Specify filename	
4. Confirm selection	5. Dialog disappears

Example cont…

Use case: **Open file by typing name**

Related use cases:
Specialization of: Open file

Steps:

Actor actions	**System responses**
1. Choose 'Open…' command	2. File open dialog appears
3a. Select text field	
3b. Type file name	
4. Click 'Open'	5. Dialog disappears

Example cont…

Use case: **Open file by browsing**

Related use cases:
Specialization of: Open file
Includes: Browse for file

Steps:

Actor actions	**System responses**
1. Choose 'Open…' command	2. File open dialog appears
3. Browse for file	
4. Confirm selection	5. Dialog disappears

Example cont…

Use case: **Attempt to open file that does not exist**

Related use cases:
Extension of: Open file by typing name

Actor actions	**System responses**
1. Choose 'Open…' command	2. File open dialog appears
3a. Select text field	
3b. Type file name	
4. Click 'Open'	5. System indicates that file does not exist
6. Correct the file name	
7. Click 'Open'	8 Dialog disappears

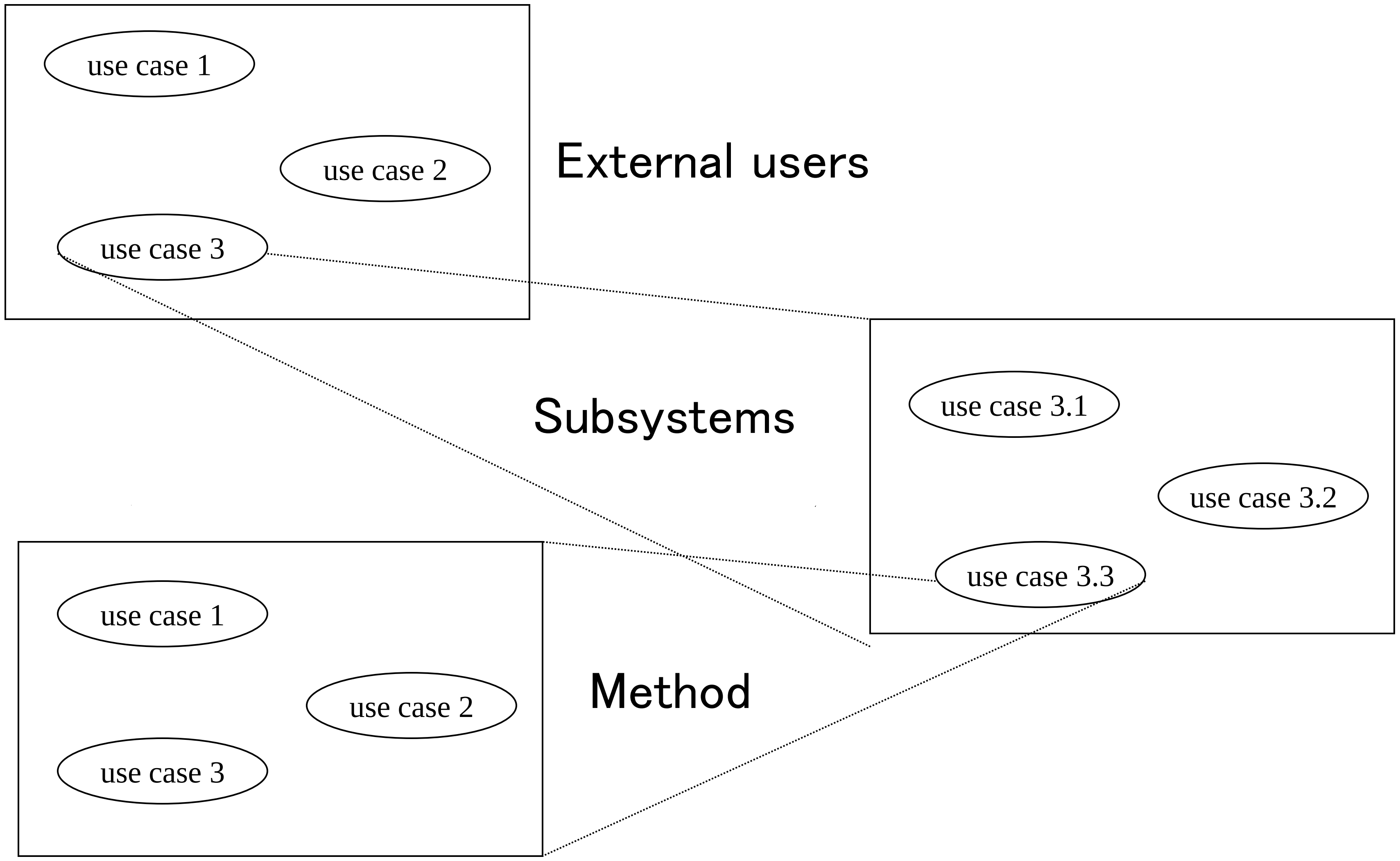
Example cont…

Use case: **Browse for file (inclusion)**

Steps:

Actor actions	**System responses**
1. If the desired file is not displayed, select a directory	2. Contents of directory is displayed
3. Repeat step 1 until the desired file is displayed	
4. Select a file	

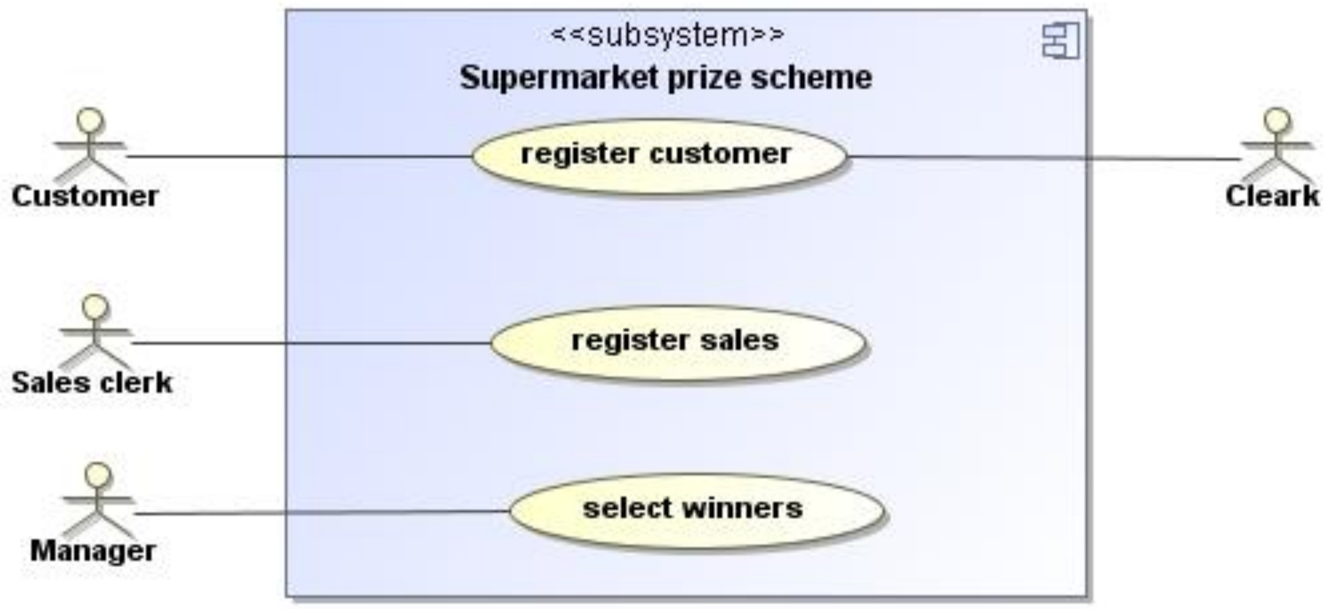
Hierarchical Organization of Use Cases



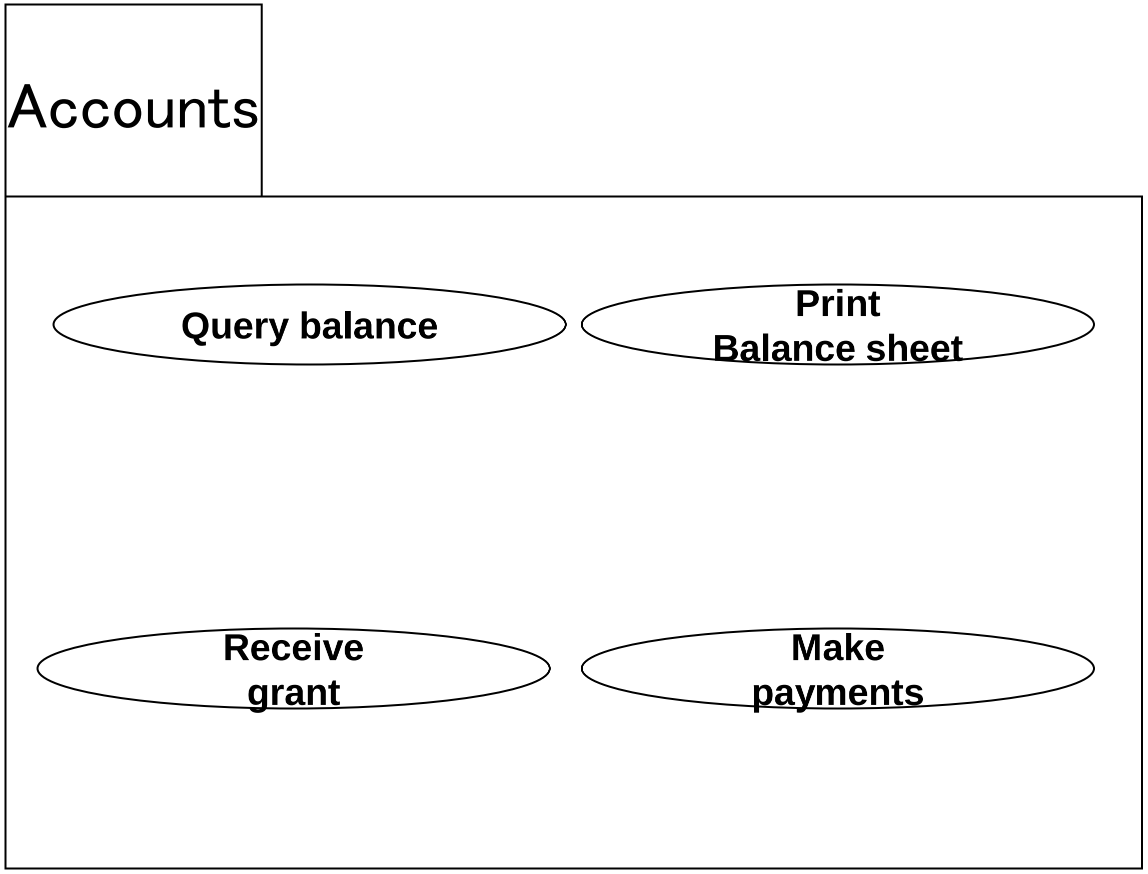
Supermarket Software

A supermarket needs to develop the following software to encourage regular customers. For this, the customer needs to supply his residence address, telephone number, and the driving license number. Each customers who registers for this scheme is assigned a unique customer number(CN) by the computer. A customer can present his CN to the check-out staff when he makes any purchase. In this case, the value of his purchase is credited against his CN. At the end of the year, the supermarket awards surprise gifts to 10 customers who make the highest total purchases over the year. Also, it awards a 22 carat gold coin to every customer whose purchases exceeds Rs. 10,000. The entries against the CN are reset on the last day of every year after the prize winners' lists are generated.

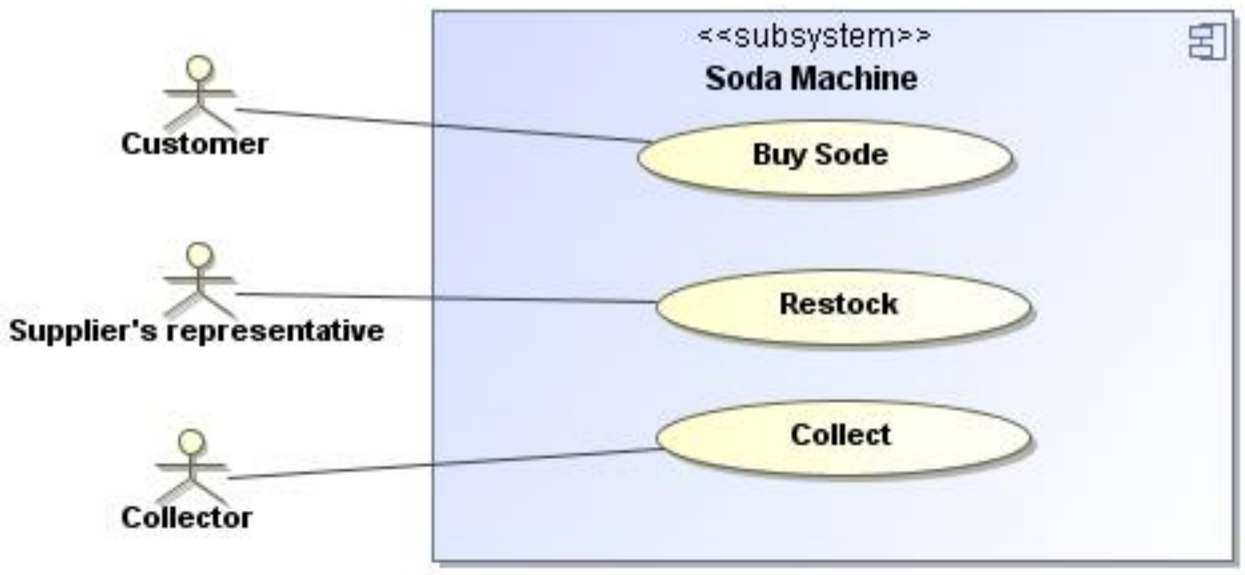
Supermarket Software cont…



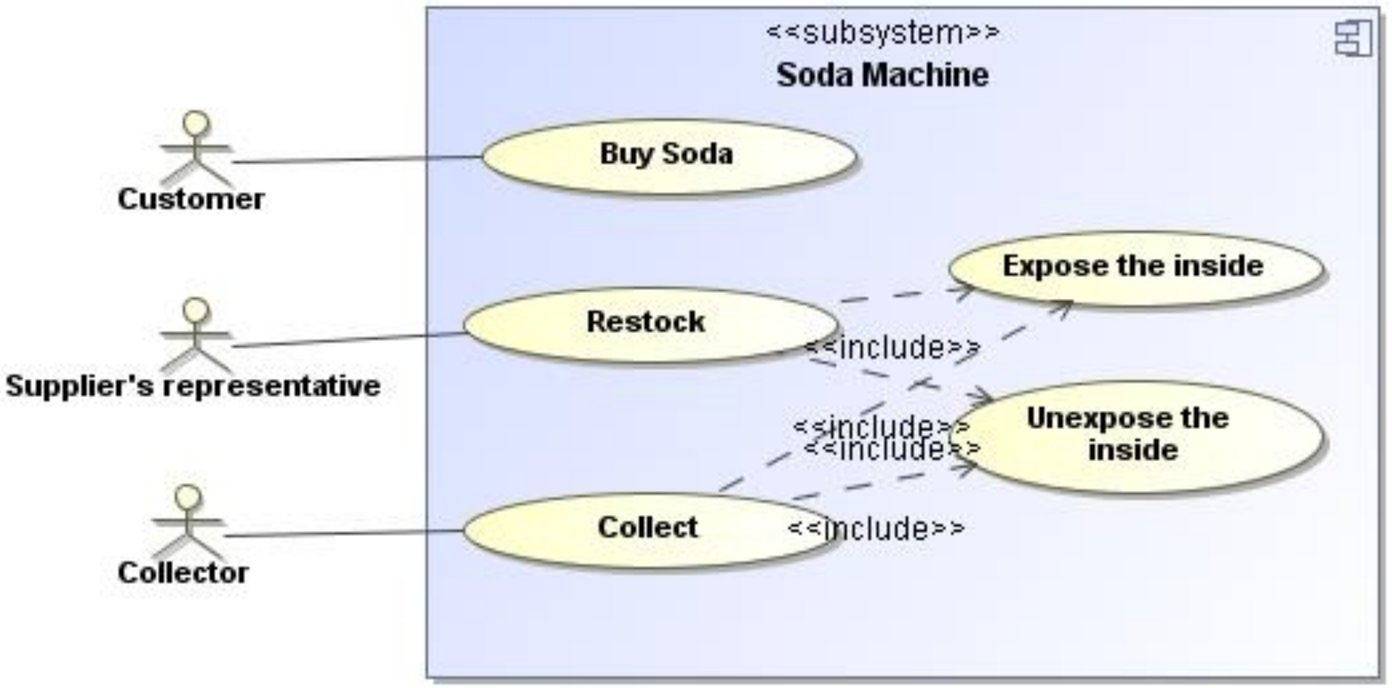
Use Case Packaging



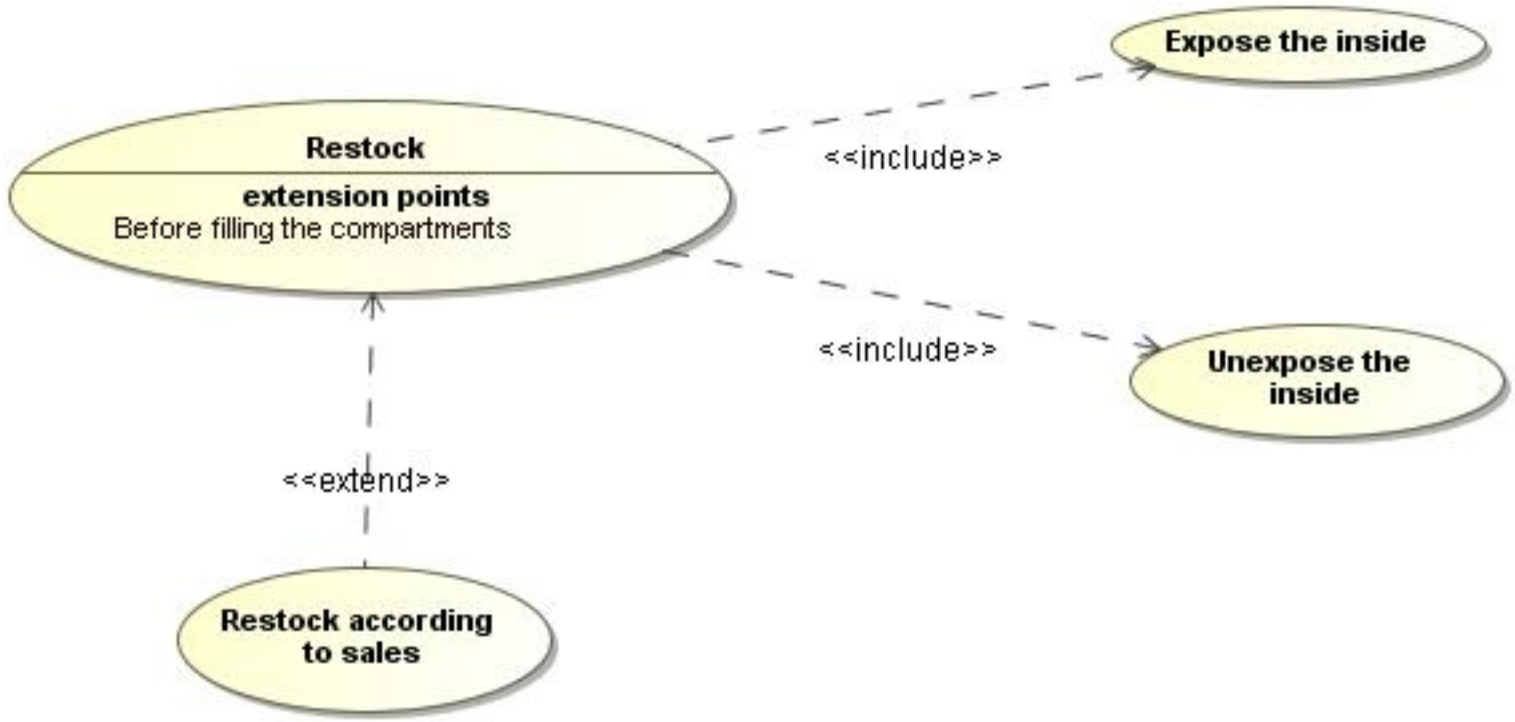
Soda Machine



Soda Machine cont…



Soda Machine cont…



The modeling processes: Choosing use cases on which to focus

- Often one use case (or a very small number) can be identified as *central* to the system
    - The entire system can be built around this particular use case
- There are other reasons for focusing on particular use cases:
    - Some use cases will represent a high *risk* because for some reason their implementation is problematic
    - Some use cases will have high political or commercial value

The benefits of basing software development on use cases

- They can
  - Help to define the *scope* of the system
  - Be used to *plan* the development process
  - Be used to both develop and validate the requirements
  - Form the basis for the definition of test cases
  - Be used to structure user manuals

Use cases must not be seen as a panacea

- The use cases themselves must be validated
  - Using the requirements validation methods.
- Some aspects of software are not covered by use case analysis.
- Innovative solutions may not be considered.

Data Models

- **Data Model**: A set of concepts to describe the *structure* of a database, and certain *constraints* that the database should obey.
- **Data Model Operations**: Operations for specifying database retrievals and updates by referring to the concepts of the data model. Operations on the data model may include *basic operations* and *user-defined operations*.

Data Models

- Relational model
- Entity-Relationship data model (mainly for database design)
- Object-based data models (Object-oriented and Object-relational)
- Semistructured data model (XML)
- Other older models:
  - Network model
  - Hierarchical model

History of Data Models

- Relational Model: proposed in 1970 by E.F. Codd (IBM), first commercial system in 1981-82. Now in several commercial products (DB2, ORACLE, SQL Server, SYBASE, INFORMIX).
- Network Model: the first one to be implemented by Honeywell in 1964-65 (IDS System). Adopted heavily due to the support by CODASYL (CODASYL - DBTG report of 1971). Later implemented in a large variety of systems - IDMS (Cullinet - now CA), DMS 1100 (Unisys), IMAGE (H.P.), VAX -DBMS (Digital Equipment Corp.).
- Hierarchical Data Model: implemented in a joint effort by IBM and North American Rockwell around 1965. Resulted in the IMS family of systems. The most popular model. Other system based on this model: System 2k (SAS inc.)

History of Data Models

- Object-oriented Data Model(s): several models have been proposed for implementing in a database system. One set comprises models of persistent O-O Programming Languages such as C++ (e.g., in OBJECTSTORE or VERSANT), and Smalltalk (e.g., in GEMSTONE). Additionally, systems like $O_{2,}$ ORION (at MCC - then ITASCA), IRIS (at H.P.- used in Open OODB).
- Object-Relational Models: Most Recent Trend. Started with Informix Universal Server. Exemplified in the latest versions of Oracle-10i, DB2, and SQL Server etc. systems.

Example COMPANY Database

- Requirements of the Company (oversimplified for illustrative purposes)
  - The company is organized into DEPARTMENTs. Each department has a name, number and an employee who *manages* the department. We keep track of the start date of the department manager.
  - Each department *controls* a number of PROJECTs*.* Each project has a name, number and is located at a single location.
  - We store each EMPLOYEE's social security number, address, salary, sex, and birthdate. Each employee *works for* one department but may *work on* several projects. We keep track of the number of hours per week that an employee currently works on each project. We also keep track of the *direct supervisor* of each employee.
  - Each employee may *have* a number of DEPENDENTs. For each dependent, we keep track of their name, sex, birthdate, and relationship to employee.

ER Model Concepts

- Entities and Attributes
  - Entities are specific objects or things in the mini-world that are represented in the database. For example the EMPLOYEE John Smith, the Research DEPARTMENT, the ProductX PROJECT
  - Attributes are properties used to describe an entity. For example an EMPLOYEE entity may have a Name, SSN, Address, Sex, BirthDate
  - A specific entity will have a value for each of its attributes. For example a specific employee entity may have Name='John Smith', SSN='123456789', Address ='731, Fondren, Houston, TX', Sex='M', BirthDate='09-JAN-55'
  - Each attribute has a *value set* (or data type) associated with it – e.g. integer, string, subrange, enumerated type, …

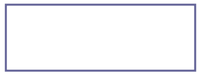
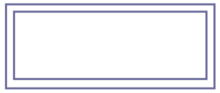
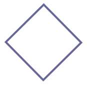
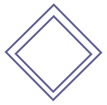
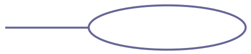
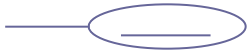
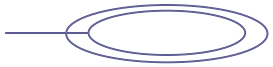
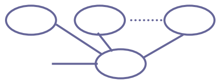
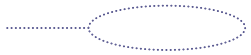
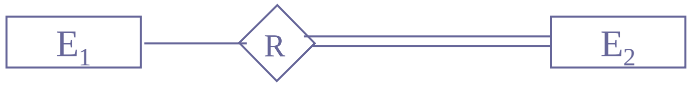
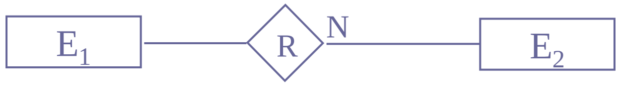
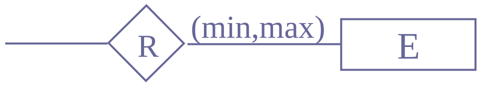
Types of Attributes

- Simple
  - Each entity has a single atomic value for the attribute. For example, SSN or Sex.
- Composite
  - The attribute may be composed of several components. For example, Address (Apt#, House#, Street, City, State, ZipCode, Country) or Name (FirstName, MiddleName, LastName). Composition may form a hierarchy where some components are themselves composite.
- Multi-valued
  - An entity may have multiple values for that attribute. For example, Color of a CAR or PreviousDegrees of a STUDENT. Denoted as {Color} or {PreviousDegrees}.
- In general, composite and multi-valued attributes may be nested arbitrarily to any number of levels although this is rare. For example, PreviousDegrees of a STUDENT is a composite multi-valued attribute denoted by {PreviousDegrees (College, Year, Degree, Field)}.

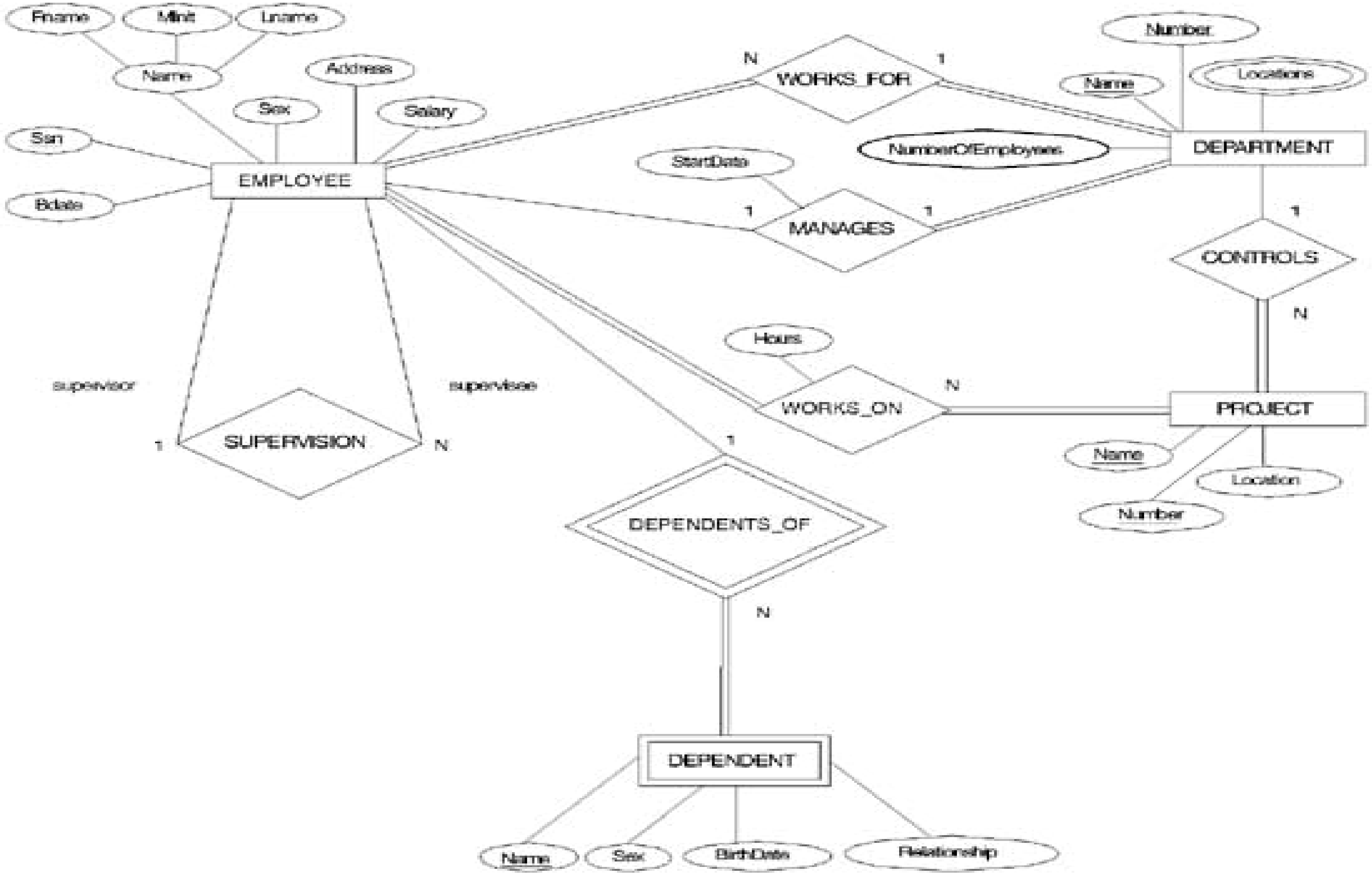
Entity Types and Key Attributes

- Entities with the same basic attributes are grouped or typed into an entity type. For example, the EMPLOYEE entity type or the PROJECT entity type.
- An attribute of an entity type for which each entity must have a unique value is called a key attribute of the entity type. For example, SSN of EMPLOYEE.
- A key attribute may be composite. For example, VehicleTagNumber is a key of the CAR entity type with components (Number, State).
- An entity type may have more than one key. For example, the CAR entity type may have two keys:
  - Chessis number and
  - License_plate number.

SUMMARY OF ER-DIAGRAM NOTATION FOR ER SCHEMAS

Symbol	Meaning
	ENTITY TYPE
	WEAK ENTITY TYPE
	RELATIONSHIP TYPE
	IDENTIFYING RELATIONSHIP TYPE
	ATTRIBUTE
	KEY ATTRIBUTE
	MULTIVALUED ATTRIBUTE
	COMPOSITE ATTRIBUTE
	DERIVED ATTRIBUTE
	TOTAL PARTICIPATION OF E_2 IN R
	CARDINALITY RATIO 1:N FOR E_1:E_2 IN R
	STRUCTURAL CONSTRAINT (min, max) ON PARTICIPATION OF E IN R

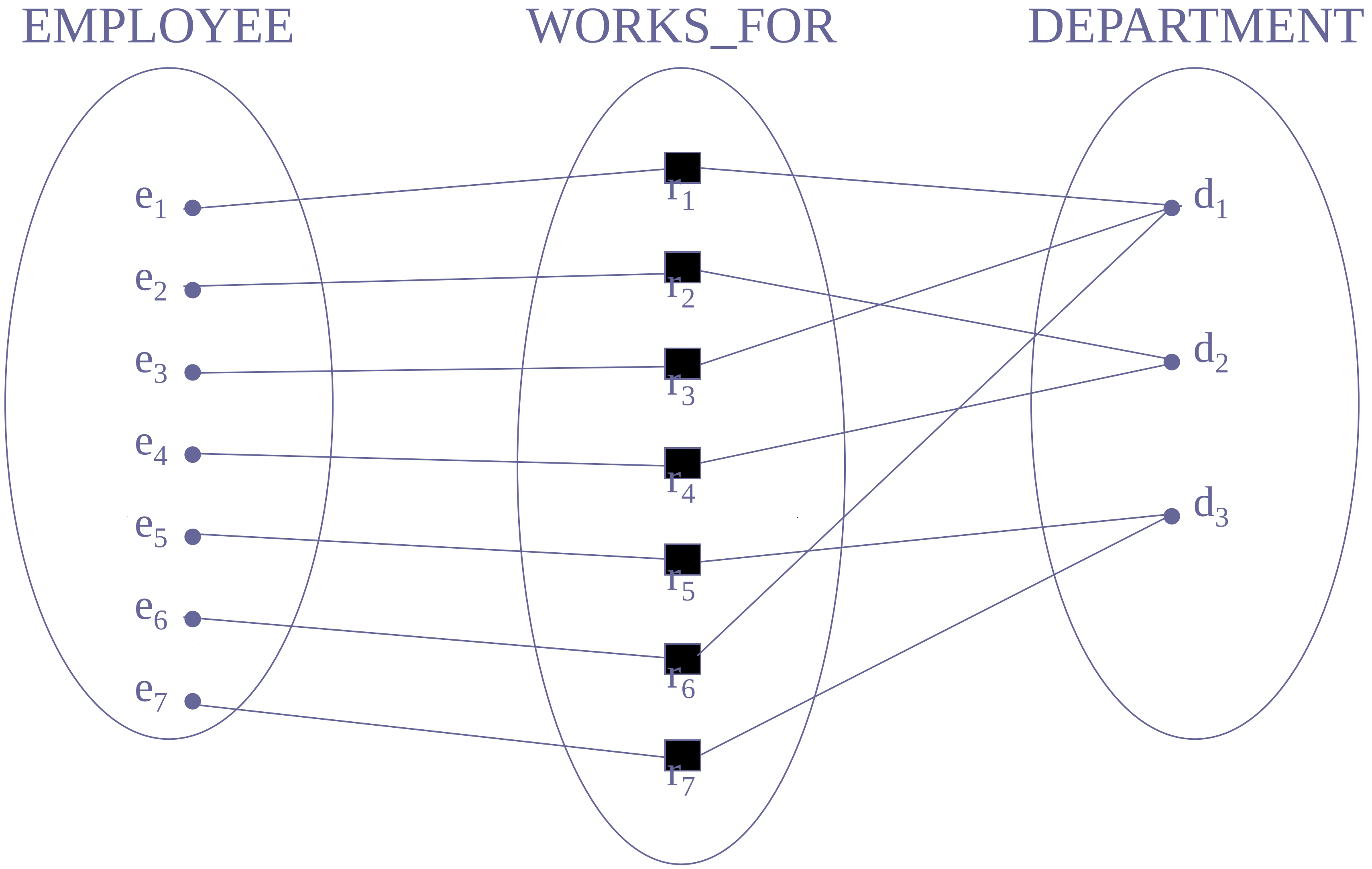
ER DIAGRAM – Entity Types are: EMPLOYEE, DEPARTMENT, PROJECT, DEPENDENT



Relationships and Relationship Types

- A relationship relates two or more distinct entities with a specific meaning. For example, EMPLOYEE John Smith works on the ProductX PROJECT or EMPLOYEE Franklin Wong manages the Research DEPARTMENT.
- Relationships of the same type are grouped or typed into a relationship type. For example, the WORKS_ON relationship type in which EMPLOYEEs and PROJECTs participate, or the MANAGES relationship type in which EMPLOYEEs and DEPARTMENTs participate.
- The degree of a relationship type is the number of participating entity types. Both MANAGES and WORKS_ON are binary relationships.

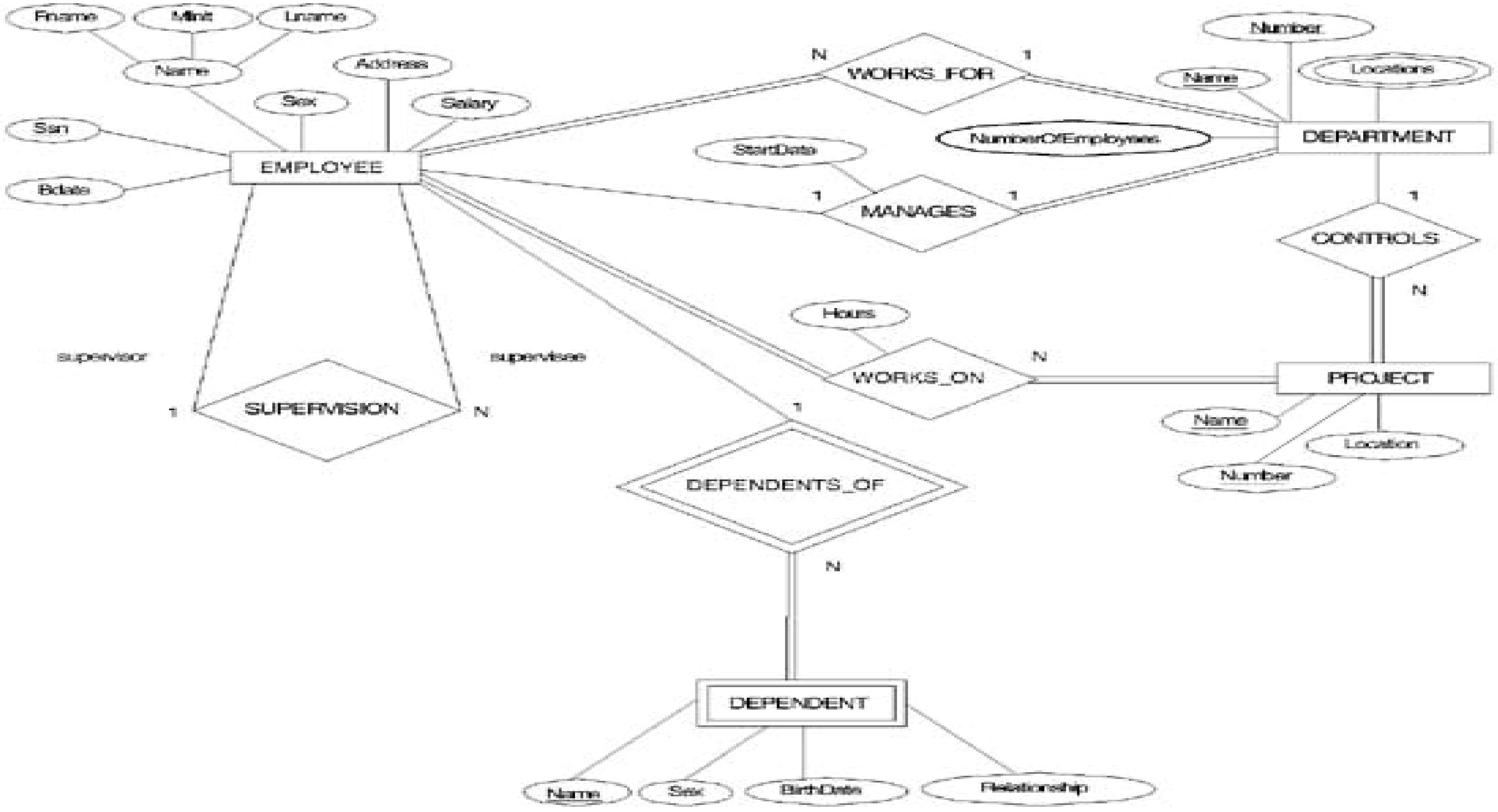
Example relationship instances of the WORKS_FOR relationship between EMPLOYEE and DEPARTMENT



Relationships and Relationship Types

- More than one relationship type can exist with the same participating entity types. For example, MANAGES and WORKS_FOR are distinct relationships between EMPLOYEE and DEPARTMENT, but with different meanings and different relationship instances.

ER DIAGRAM – Relationship Types are: WORKS_FOR, MANAGES, WORKS_ON, CONTROLS, SUPERVISION, DEPENDENTS_OF



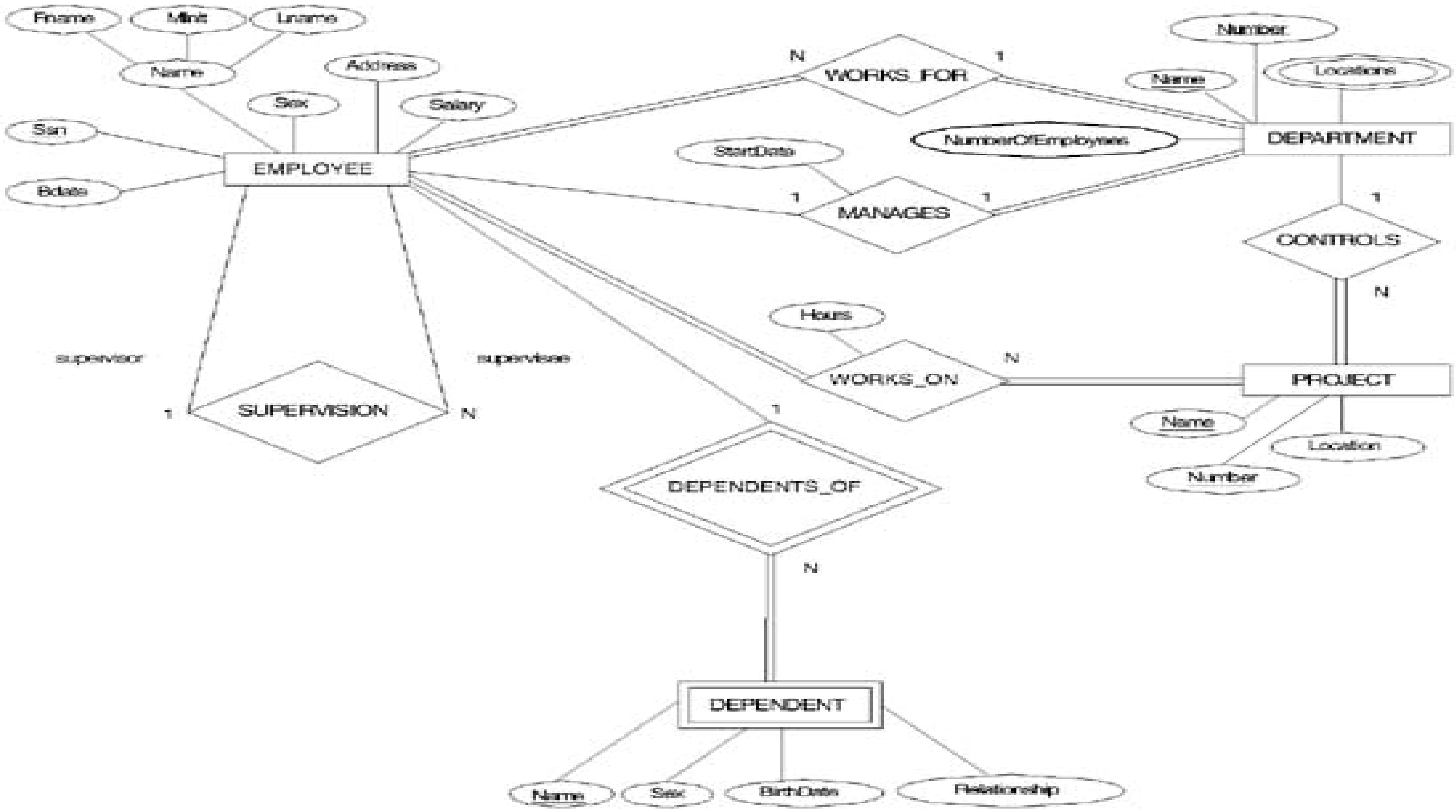
Weak Entity Types

- An entity that does not have a key attribute
- A weak entity must participate in an identifying relationship type with an owner or identifying entity type
- Entities are identified by the combination of:
  - A partial key of the weak entity type
  - The particular entity they are related to in the identifying entity type

Example:

Suppose that a DEPENDENT entity is identified by the dependent's first name and birhtdate, *and* the specific EMPLOYEE that the dependent is related to. DEPENDENT is a weak entity type with EMPLOYEE as its identifying entity type via the identifying relationship type DEPENDENT_OF

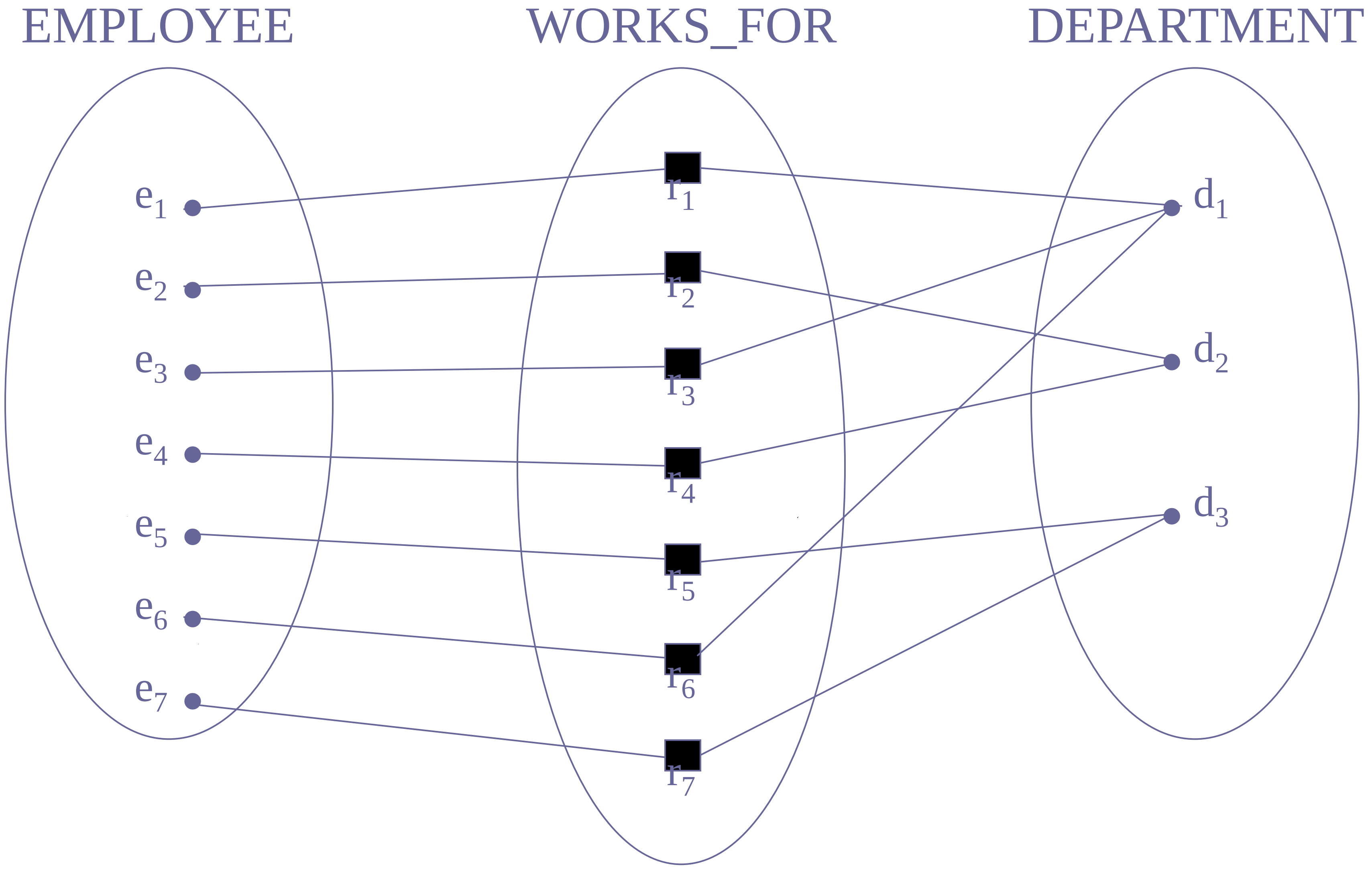
Weak Entity Type is: DEPENDENT Identifying Relationship is: DEPENDENTS_OF



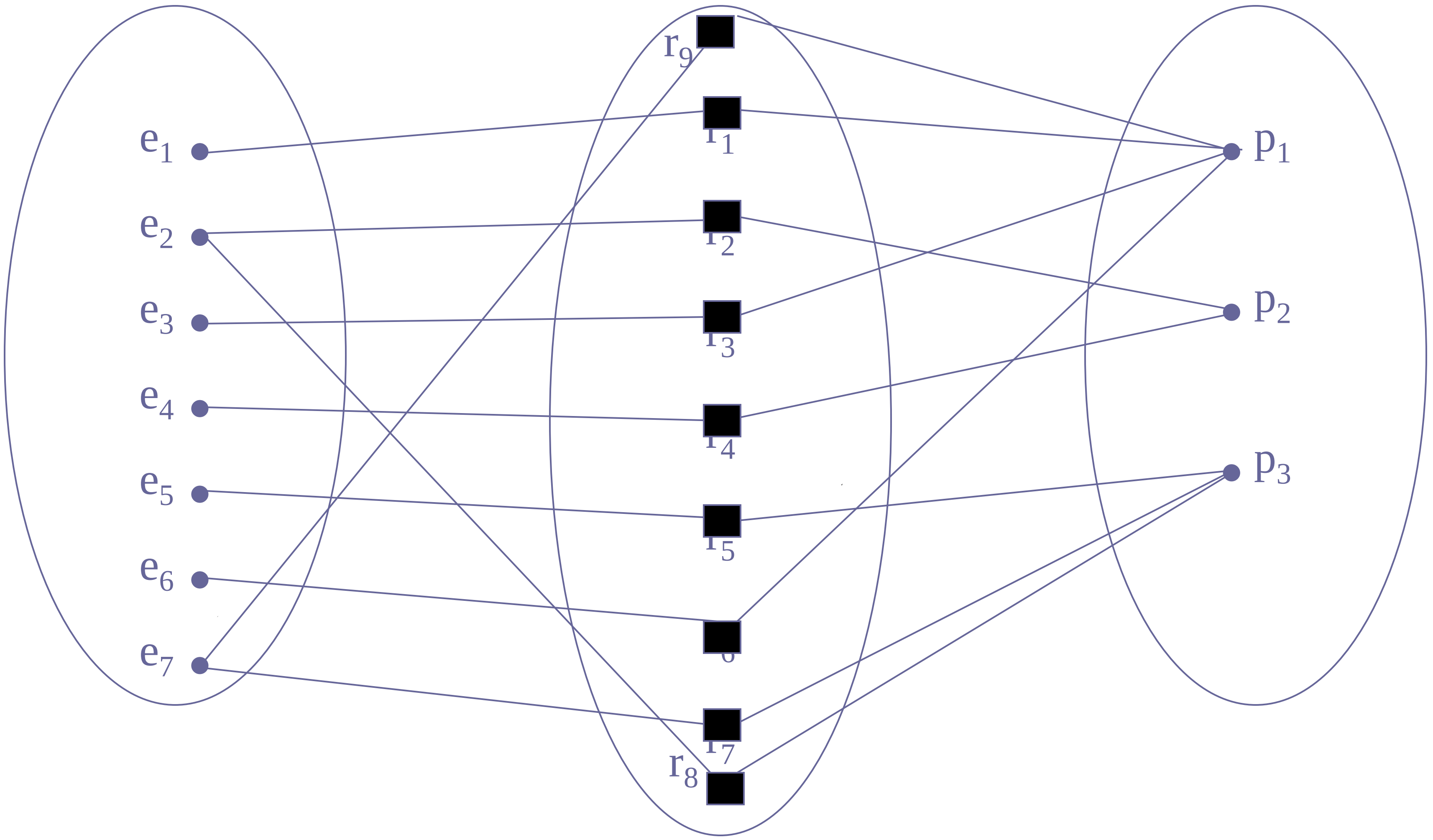
Constraints on Relationships

- Constraints on Relationship Types
  - (Also known as ratio constraints)
  - Maximum Cardinality
    - One-to-one (1:1)
    - One-to-many (1:N) or Many-to-one (N:1)
    - Many-to-many
  - Minimum Cardinality (also called participation constraint or existence dependency constraints)
    - zero (optional participation, not existence-dependent)
    - one or more (mandatory, existence-dependent)

Many-to-one (N:1) RELATIONSHIP



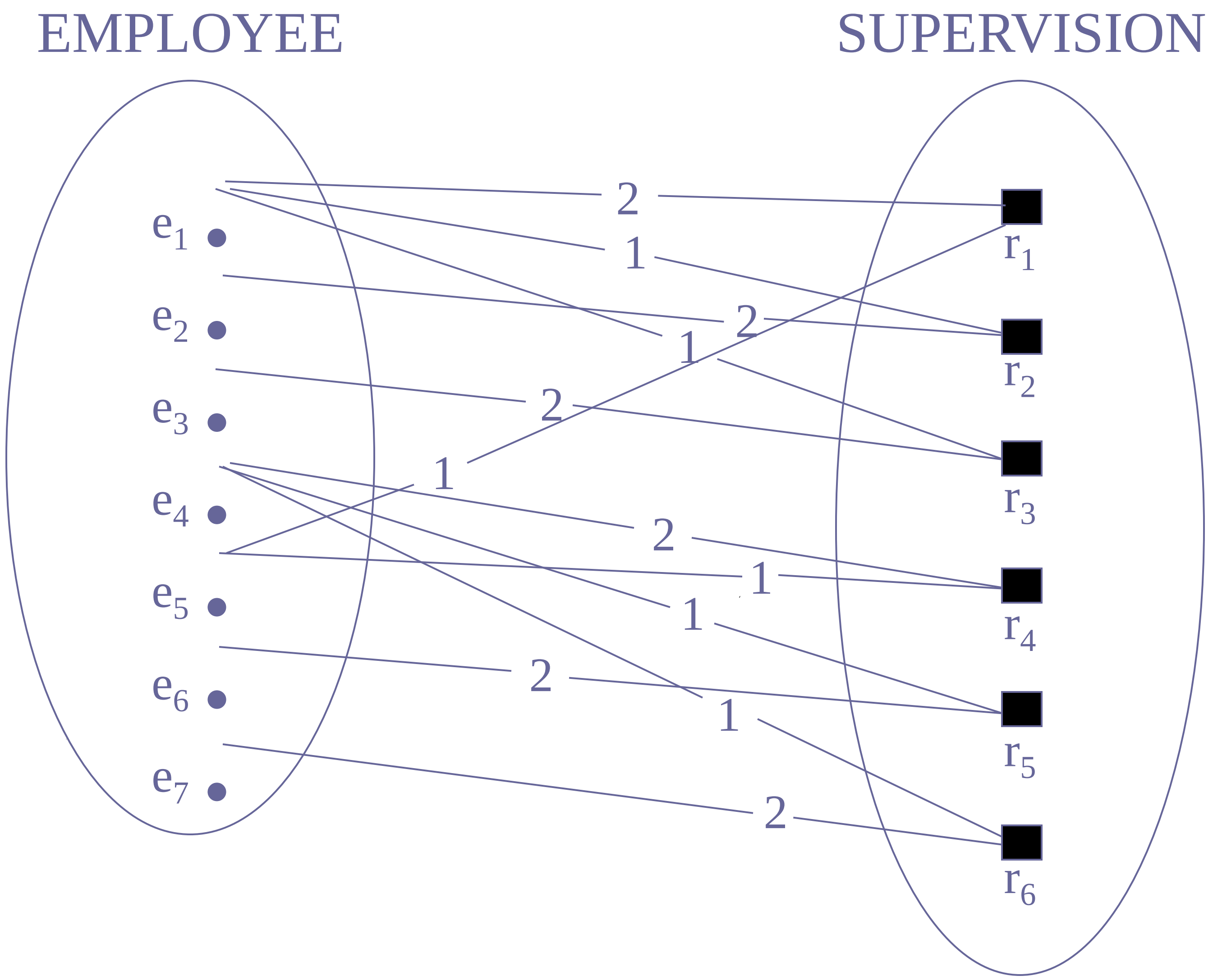
Many-to-many (M:N) RELATIONSHIP



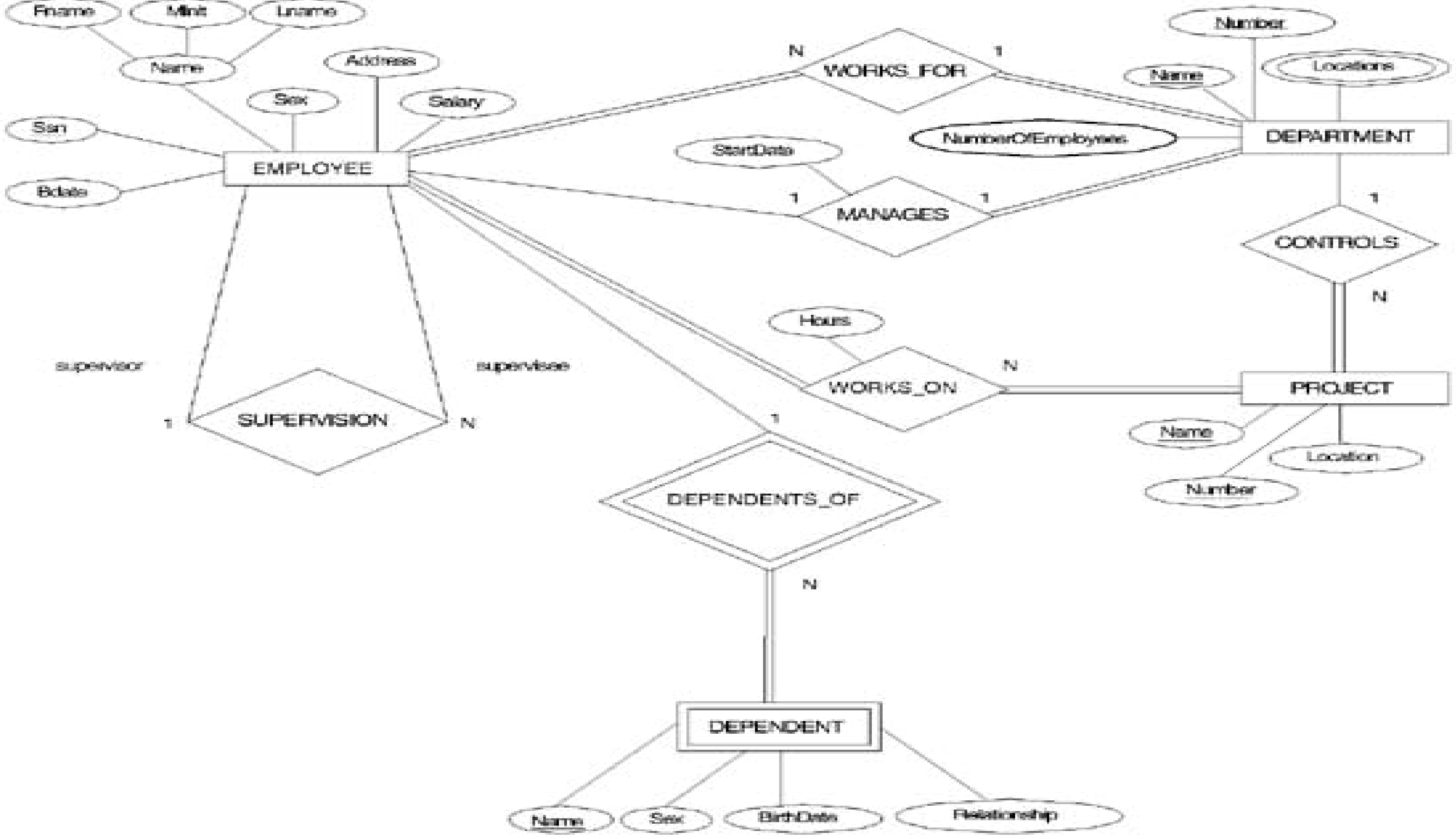
Relationships and Relationship Types

- We can also have a **recursive** relationship type.
- Both participations are same entity type in different roles.
- For example, SUPERVISION relationships between EMPLOYEE (in role of supervisor or boss) and (another) EMPLOYEE (in role of subordinate or worker).
- In following figure, first role participation labeled with 1 and second role participation labeled with 2.
- In ER diagram, need to display role names to distinguish participations.

A RECURSIVE RELATIONSHIP SUPERVISION



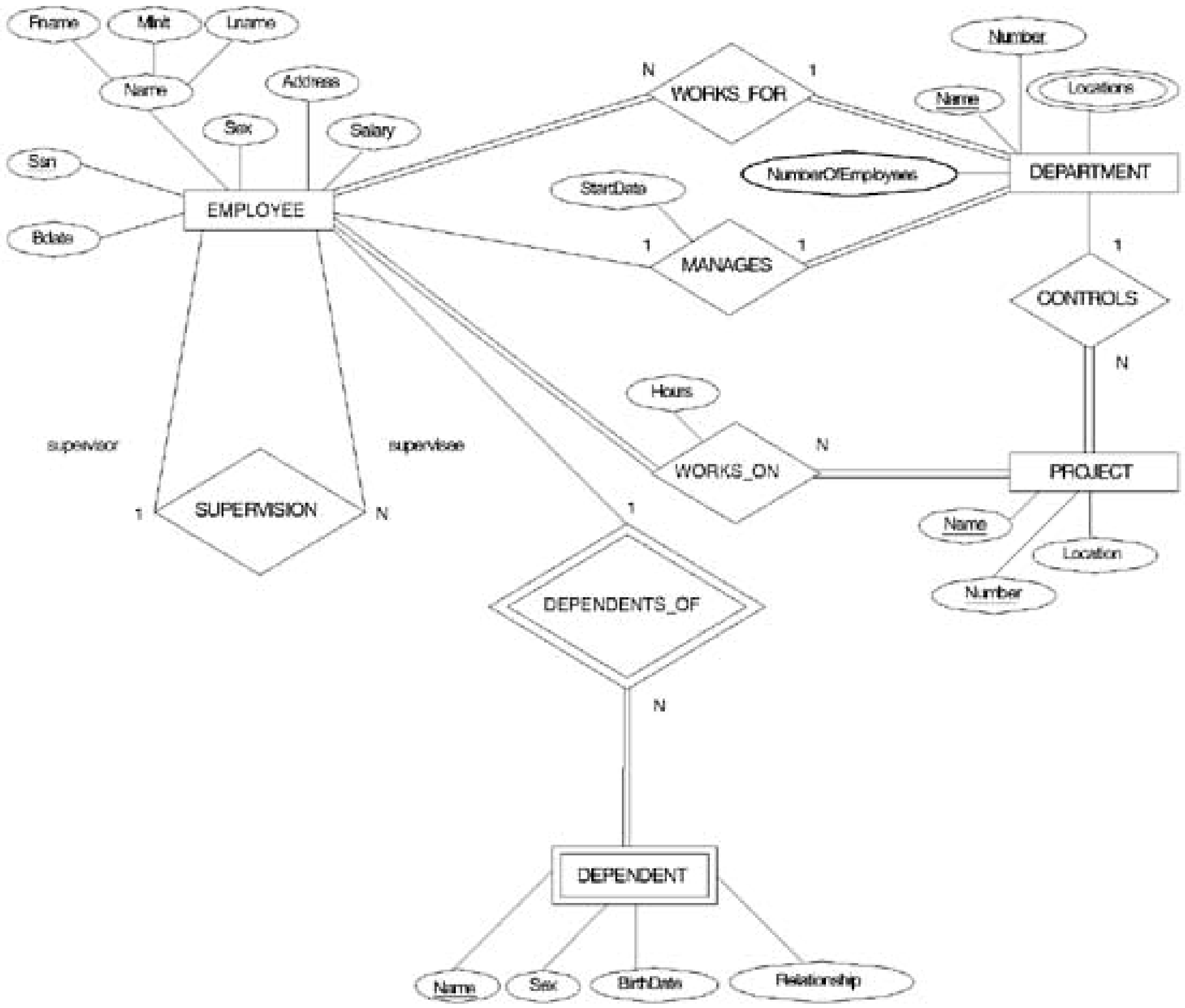
Recursive Relationship Type is: SUPERVISION (participation role names are shown)



Attributes of Relationship types

- A relationship type can have attributes; for example, HoursPerWeek of WORKS_ON; its value for each relationship instance describes the number of hours per week that an EMPLOYEE works on a PROJECT.

Attribute of a Relationship Type is: Hours of WORKS_ON



Structural Constraints – one way to express semantics of relationships

<u>Structural constraints on relationships:</u>

- **Cardinality ratio** (of a binary relationship): 1:1, 1:N, N:1, or M:N

  SHOWN BY PLACING APPROPRIATE NUMBER ON THE LINK.

- **Participation constraint** (on each participating entity type): total (called *existence dependency*) or partial.

  SHOWN BY DOUBLE LINING THE LINK

NOTE: These are easy to specify <u>for Binary Relationship</u> <u>Types</u>.

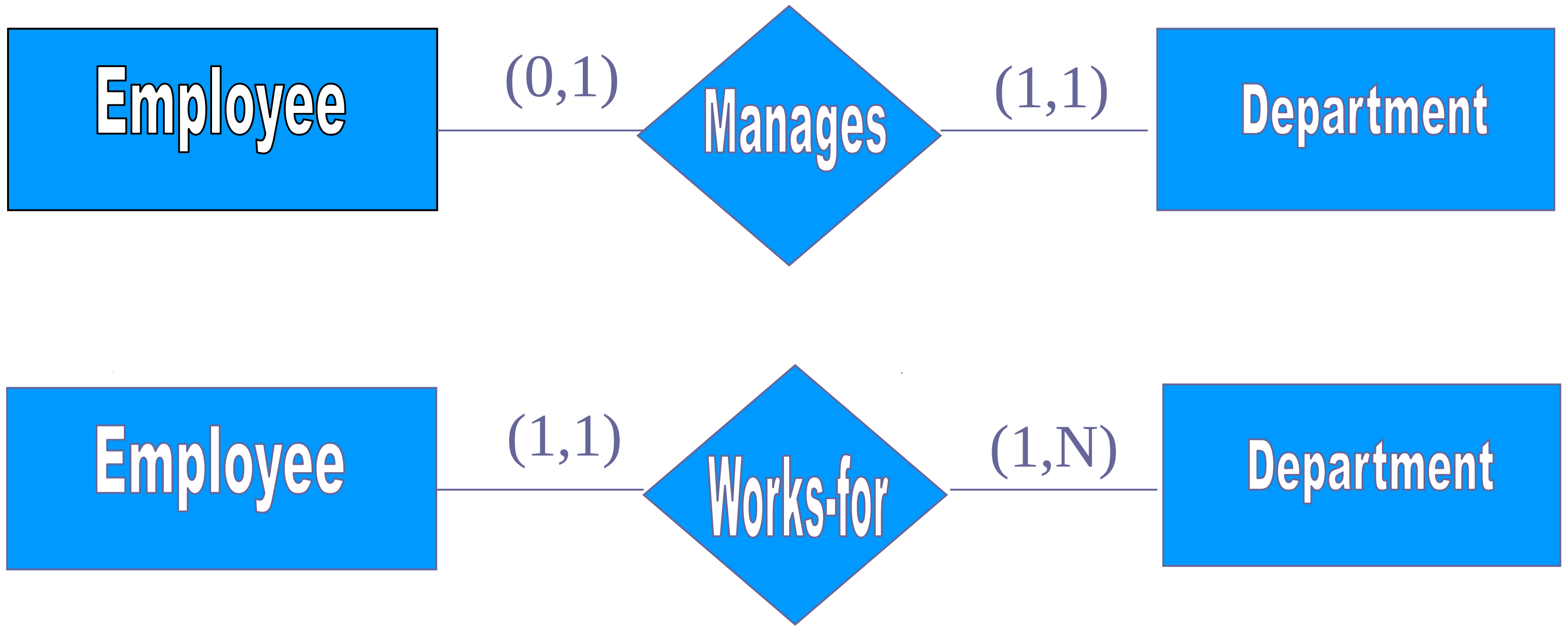
Alternative (min, max) notation for relationship structural constraints:

- Specified on *each participation* of an entity type E in a relationship type R
- Specifies that each entity e in E participates in *at least* min and *at most* max relationship instances in R
- Default(no constraint): min=0, max=n
- Must have min≤max, min≥0, max ≥1
- Derived from the knowledge of mini-world constraints

Examples:

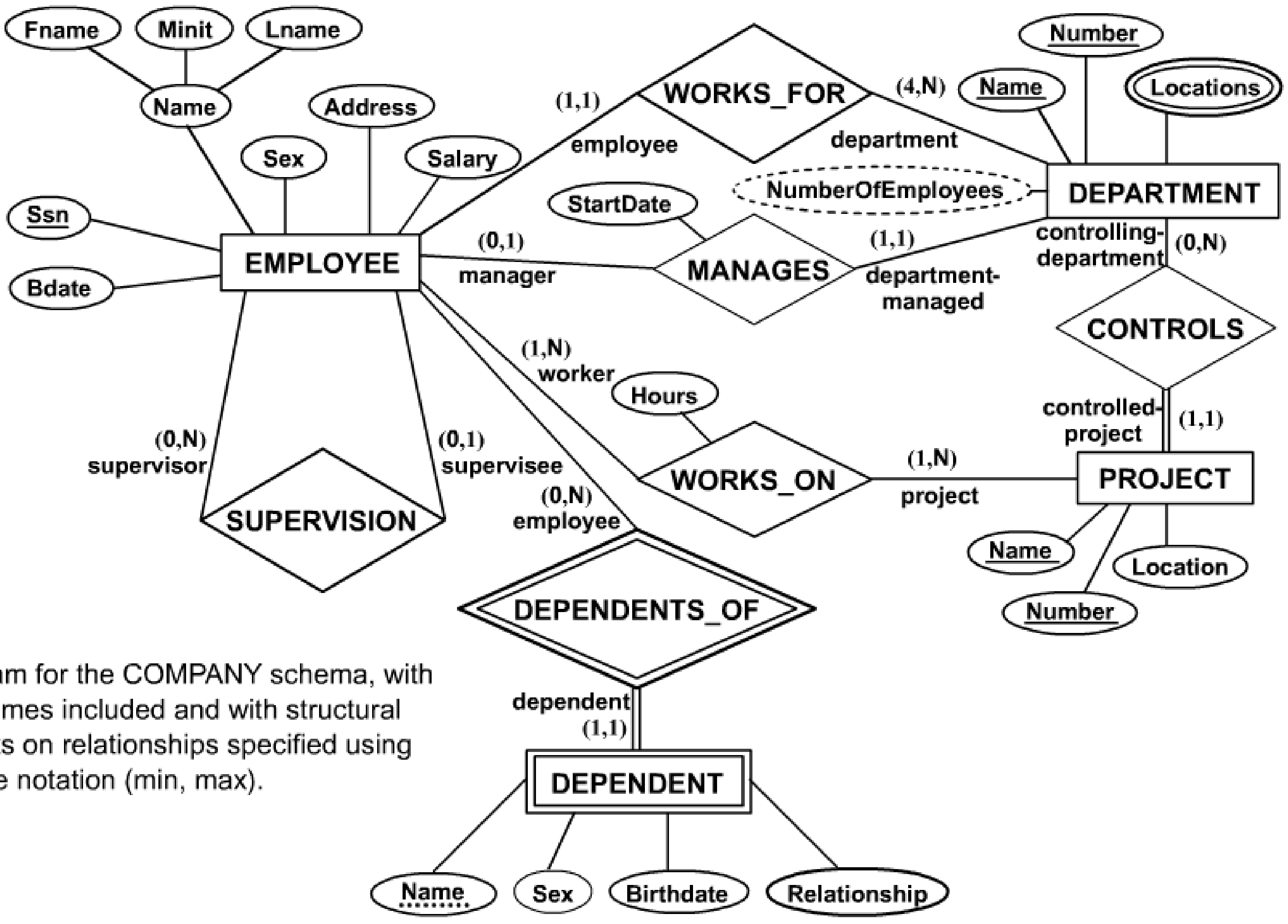
- A department has *exactly one* manager and an employee can manage *at most one* department.
  – Specify (0,1) for participation of EMPLOYEE in MANAGES
  – Specify (1,1) for participation of DEPARTMENT in MANAGES
- An employee can work for *exactly one* department but a department can have *any number of employees*.
  – Specify (1,1) for participation of EMPLOYEE in WORKS_FOR
  – Specify (0,n) for participation of DEPARTMENT in WORKS_FOR

The (min,max) notation relationship constraints



COMPANY ER Schema Diagram using (min, max) notation

Alternative ER Notations

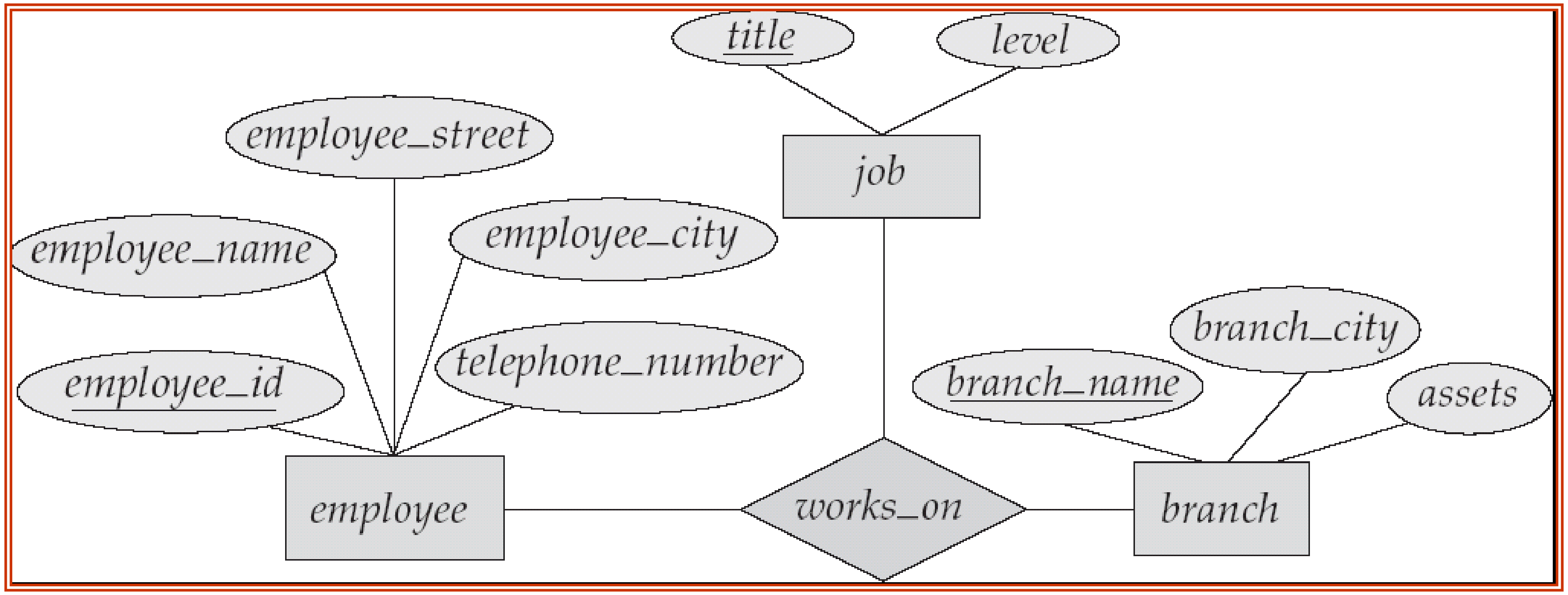


ER diagram for the COMPANY schema, with all role names included and with structural constraints on relationships specified using alternative notation (min, max).

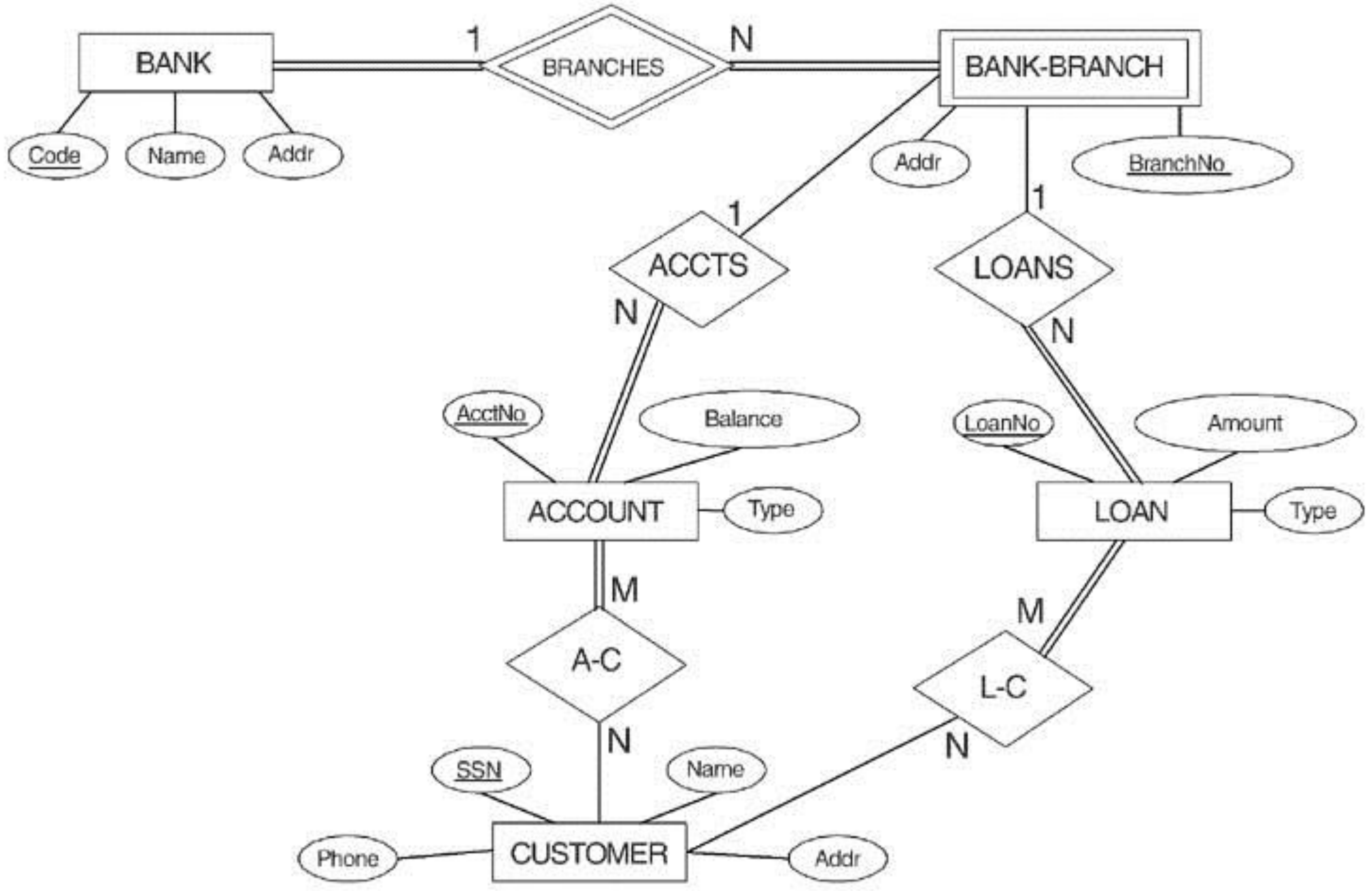
Relationships of Higher Degree

- Relationship types of degree 2 are called **binary**
- Relationship types of degree 3 are called **ternary** and of degree n are called **n-ary**
- In general, an n-ary relationship *is not* equivalent to n binary relationships
- Higher-order relationships discussed further.

E-R Diagram with a Ternary Relationship



ER DIAGRAM FOR A BANK DATABASE



This image cannot currently be displayed.

End of Chapter 4

Questions?