COSC 304
Introduction to Database Systems
Enhanced Entity-Relationship (EER) Modeling

Dr. Ramon Lawrence
University of British Columbia Okanagan
ramon.lawrence@ubc.ca
Enhanced Entity-Relationship Modeling

*Enhanced Entity-Relationship (EER) modeling* is an extension of ER modeling to include object-oriented concepts such as:

- superclasses and subclasses
- specialization and generalization
- aggregation and composition

These modeling constructs may allow more precise modeling of systems that are object-oriented in nature such as:

- CAD/CAM systems (Computer-Aided Design/Manufacturing)
- GIS (Geographical Information Systems)
Review: Superclasses and Subclasses

The object-oriented ideas of inheritance and superclasses and subclasses are taught during programming in an OO language such as Java.

A **superclass** is a general class that is extended by one or more subclasses.

A **subclass** is a more specific class that extends a superclass by inheriting its methods and attributes and then adding its own methods and attributes.

**Inheritance** is the process of a subclass inheriting all the methods and attributes of a superclass.
Superclasses and Subclasses Example

Java code:

```java
public class SavingsAccount extends BankAccount
```

UML class diagram:

Triangle points to superclass
**Superclasses and Subclasses Question**

**Question:** How many of the following statements are **true**?

1) D is a superclass.
2) D has 1 attribute.
3) B and C are subclasses of A.
4) B inherits V from D.
5) D inherits attribute X.

A) 0  B) 1  C) 2  D) 3  E) 4
When to use EER Modeling?

It is important to emphasize that many database projects do not need the object-oriented modeling features of EER modeling.

Remember the goal of conceptual modeling is to produce a model that is simple and easy to understand.

Do not introduce complicated subclass/superclass relationships if they are not needed.

Only use the EER modeling constructs if they offer a **significant advantage** over regular ER modeling.
When to use EER Modeling? (2)

EER modeling is especially useful when the domain being modeled is object-oriented in nature and the use of inheritance reduces the complexity of the design.

There are two common cases where EER modeling is useful instead of basic ER modeling:

1) When using *attribute inheritance* can reduce the use of nulls in a single entity relation (that contains multiple subclasses).

2) Subclasses can be used to explicitly model and name subsets of entity types that participate in their own relationships.
When to use EER Modeling?
Using Attribute Inheritance

Note that the title attribute indicates what job the employee does at the company. Consider if each job title had its own unique information that we would want to record such as:

- **EE, PR** - programming language used (lang), DB used (db)
- **SA, ME** - MBA? (MBA), bonus
When to use EER Modeling? Using Attribute Inheritance (2)

We could represent all these attributes in a single relation:

<table>
<thead>
<tr>
<th>eno</th>
<th>ename</th>
<th>bdate</th>
<th>title</th>
<th>salary</th>
<th>supereno</th>
<th>dno</th>
<th>lang</th>
<th>db</th>
<th>MBA</th>
<th>bonus</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>J. Doe</td>
<td>01-05-75</td>
<td>EE</td>
<td>30000</td>
<td>E2</td>
<td></td>
<td>C++</td>
<td>MySQL</td>
<td>N</td>
<td>2000</td>
</tr>
<tr>
<td>E2</td>
<td>M. Smith</td>
<td>06-04-66</td>
<td>SA</td>
<td>50000</td>
<td>E5</td>
<td>D3</td>
<td></td>
<td></td>
<td>N</td>
<td>3000</td>
</tr>
<tr>
<td>E3</td>
<td>A. Lee</td>
<td>07-05-66</td>
<td>ME</td>
<td>40000</td>
<td>E7</td>
<td>D2</td>
<td>Java</td>
<td>Oracle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E4</td>
<td>J. Miller</td>
<td>09-01-50</td>
<td>PR</td>
<td>20000</td>
<td>E6</td>
<td>D3</td>
<td></td>
<td></td>
<td>Y</td>
<td>4000</td>
</tr>
<tr>
<td>E5</td>
<td>B. Casey</td>
<td>12-25-71</td>
<td>SA</td>
<td>50000</td>
<td>E8</td>
<td>D3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E6</td>
<td>L. Chu</td>
<td>11-30-65</td>
<td>EE</td>
<td>30000</td>
<td>E7</td>
<td>D2</td>
<td>C++</td>
<td>DB2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E7</td>
<td>R. Davis</td>
<td>09-08-77</td>
<td>ME</td>
<td>40000</td>
<td>E8</td>
<td>D1</td>
<td></td>
<td></td>
<td>N</td>
<td>3000</td>
</tr>
<tr>
<td>E8</td>
<td>J. Jones</td>
<td>10-11-72</td>
<td>SA</td>
<td>50000</td>
<td>D1</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td>6000</td>
</tr>
</tbody>
</table>

Note the wasted space as attributes that do not apply to a particular subclass are NULL.
When to use EER Modeling? Using Attribute Inheritance (3)

A better solution would be to make two subclasses of Employee called Developer and Manager:

<table>
<thead>
<tr>
<th>Employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>eno {PK}</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>bdate</td>
</tr>
<tr>
<td>title</td>
</tr>
<tr>
<td>salary</td>
</tr>
<tr>
<td>supereno</td>
</tr>
<tr>
<td>dno</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Developer</th>
</tr>
</thead>
<tbody>
<tr>
<td>lang</td>
</tr>
<tr>
<td>db</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBA</td>
</tr>
<tr>
<td>bonus</td>
</tr>
</tbody>
</table>
When to use EER Modeling?
Using Attribute Inheritance (4)

Resulting relations:

**Employee Relation**

<table>
<thead>
<tr>
<th>eno</th>
<th>ename</th>
<th>bdate</th>
<th>title</th>
<th>salary</th>
<th>supereno</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>J. Doe</td>
<td>01-05-75</td>
<td>EE</td>
<td>30000</td>
<td>E2</td>
<td>null</td>
</tr>
<tr>
<td>E2</td>
<td>M. Smith</td>
<td>06-04-66</td>
<td>SA</td>
<td>50000</td>
<td>E5</td>
<td>D3</td>
</tr>
<tr>
<td>E3</td>
<td>A. Lee</td>
<td>07-05-66</td>
<td>ME</td>
<td>40000</td>
<td>E7</td>
<td>D2</td>
</tr>
<tr>
<td>E4</td>
<td>J. Miller</td>
<td>09-01-50</td>
<td>PR</td>
<td>20000</td>
<td>E6</td>
<td>D3</td>
</tr>
<tr>
<td>E5</td>
<td>B. Casey</td>
<td>12-25-71</td>
<td>SA</td>
<td>50000</td>
<td>E8</td>
<td>D3</td>
</tr>
<tr>
<td>E6</td>
<td>L. Chu</td>
<td>11-30-65</td>
<td>EE</td>
<td>30000</td>
<td>E7</td>
<td>D2</td>
</tr>
<tr>
<td>E7</td>
<td>R. Davis</td>
<td>09-08-77</td>
<td>ME</td>
<td>40000</td>
<td>E8</td>
<td>D1</td>
</tr>
<tr>
<td>E8</td>
<td>J. Jones</td>
<td>10-11-72</td>
<td>SA</td>
<td>50000</td>
<td>null</td>
<td>D1</td>
</tr>
</tbody>
</table>

**Developer Relation**

<table>
<thead>
<tr>
<th>eno</th>
<th>lang</th>
<th>db</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>C++</td>
<td>MySQL</td>
</tr>
<tr>
<td>E4</td>
<td>Java</td>
<td>Oracle</td>
</tr>
<tr>
<td>E6</td>
<td>C++</td>
<td>DB2</td>
</tr>
</tbody>
</table>

**Manager Relation**

<table>
<thead>
<tr>
<th>eno</th>
<th>MBA</th>
<th>bonus</th>
</tr>
</thead>
<tbody>
<tr>
<td>E2</td>
<td>N</td>
<td>2000</td>
</tr>
<tr>
<td>E3</td>
<td>N</td>
<td>3000</td>
</tr>
<tr>
<td>E5</td>
<td>Y</td>
<td>4000</td>
</tr>
<tr>
<td>E7</td>
<td>N</td>
<td>3000</td>
</tr>
<tr>
<td>E8</td>
<td>Y</td>
<td>6000</td>
</tr>
</tbody>
</table>
Generalization and Specialization

Subclasses and superclasses are created by using either generalization or specialization.

**Specialization** is the process of creating more specialized subclasses of an existing superclass.

- Top-down process: Start with a general class and then subdivide it into more specialized classes.
  - The specialized classes may contain their own attributes. Attributes common to all subclasses remain in the superclass.

**Generalization** is the process of creating a more general superclass from existing subclasses.

- Bottom-up process: Start with specialized classes and try to determine a general class that contains the attributes common to all of them.
Specialization Example

General class - specialize into subclasses

Employee
- eno {PK}
- name
- bdate
- title
- salary
- supereno
- dno
- lang
- db
- MBA
- bonus

Employee superclass

Manager
- MBA
- bonus

Developer
- lang
- db

Indicates specialization/generalization

subclasses
Generalization Example

<table>
<thead>
<tr>
<th>Developer</th>
<th>Employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>number {PK}</td>
<td>eno {PK}</td>
</tr>
<tr>
<td>developerName</td>
<td>name</td>
</tr>
<tr>
<td>birthDate</td>
<td>bdate</td>
</tr>
<tr>
<td>title</td>
<td>title</td>
</tr>
<tr>
<td>salary</td>
<td>salary</td>
</tr>
<tr>
<td>supereno</td>
<td>supereno</td>
</tr>
<tr>
<td>dno</td>
<td>dno</td>
</tr>
<tr>
<td>lang</td>
<td>MBA</td>
</tr>
<tr>
<td>db</td>
<td>bonus</td>
</tr>
</tbody>
</table>

Specific classes - generalize to create superclass

Indicates specialization/generalization

subclasses
Constraints on Generalization and Specialization

There are two types of constraints associated with generalization and specialization:

- **Participation constraint** - determines if every member in a superclass must participate as a member of one of its subclasses.
  - It may be **optional** for a superclass member to be a member of one of its subclasses, or it may be **mandatory** that a superclass member be a member of one of its subclasses.

- **Disjoint constraint** - determines if a member of a superclass can be a member of one or more than one of its subclasses.
  - If a superclass object may be a member of only one of its subclasses this is denoted by **OR** (subclasses are disjoint).
  - Otherwise, **AND** is used to indicate that it may be in more than one of its subclasses.
An employee must be either a developer or a manager, but cannot be both.

<table>
<thead>
<tr>
<th>Employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>eno {PK}</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>bdate</td>
</tr>
<tr>
<td>title</td>
</tr>
<tr>
<td>salary</td>
</tr>
<tr>
<td>supereno</td>
</tr>
<tr>
<td>dno</td>
</tr>
</tbody>
</table>

{Mandatory, OR}

- **Developer**
  - lang
  - db

- **Manager**
  - MBA
  - bonus

Participation constraint (must be developer or manager)

Disjoint constraint (cannot be both a developer and a manager)
An employee may specialize as a developer or manager. An employee may be both a manager and developer.

Employee
- eno {PK}
- name
- bdate
- title
- salary
- supereno
- dno

Participation constraint (may be developer or manager)

Disjoint constraint (can be both a developer and a manager)

Developer
- lang
- db

Manager
- MBA
- bonus
General Predicate Constraints

*Predicate-defined constraints* specify when an object participates in a subclass using a certain rule.

- For example, a subclass called *RichEmployees* can be defined with a membership predicate such as *salary > 100000*.

*Attribute-defined subclasses* are a particular type of predicate-defined constraint where the value of an attribute(s) determines if an object is a member of a subclass.

- For example, the *title* field could be used as a *defining attribute* for the *Developer* and *Manager* subclasses.
  
  $\Rightarrow$ *Emp is in Developer if title = 'EE' or 'PR'*
  
  $\Rightarrow$ *Emp is in Manager if title = 'ME' or 'SA'*
Note: What is the participation and the disjoint constraints for superclass Employee (with subclasses Manager and Supervisor) given these instances?
Relationship Constraints vs. Inheritance Constraints

There is a parallel between relationship constraints on associations/relationships and inheritance constraints on superclasses and subclasses.

- Minimum # of occurrences – called participation constraint in both cases
- Maximum # of occurrences – called cardinality constraint for relationships and disjoint constraint for subclasses

Possible combinations:

<table>
<thead>
<tr>
<th>Subclass Constraints</th>
<th>Relationship Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional, AND</td>
<td>0..*</td>
</tr>
<tr>
<td>Optional, OR</td>
<td>0..1</td>
</tr>
<tr>
<td>Mandatory, AND</td>
<td>1..*</td>
</tr>
<tr>
<td>Mandatory, OR</td>
<td>1..1</td>
</tr>
</tbody>
</table>
**EER Question**

**Question:** How many of the following statements are true?

1) Generalization is a bottom-up process.
2) In an UML diagram, the inheritance arrow points towards the superclass.
3) OPTIONAL and MANDATORY are possible choices for the participation constraint.
4) If the disjoint constraint is AND, a given object can be a member of multiple subclasses.
5) If the participation constraint is OPTIONAL, the disjoint constraint must be AND.

A) 0    B) 1    C) 2    D) 3    E) 4
Multiple Inheritance

If each class only has one superclass, then the class diagram is said to be a *specialization* or *type hierarchy*.

If a class may have more than one superclass, then the class diagram is said to be a *specialization* or *type lattice*.

Although multiple inheritance is powerful, it should be avoided if possible.
Multiple Inheritance Example

Employee

- eno {PK}
- name
- ...

{Optional, OR}

Developer

- lang
- db

Manager

- MBA
- bonus

{Optional}

DeveloperManager

- project
- projectDueDate
Aggregation represents a 'HAS-A' or 'IS-PART-OF' relationship between entity types. One entity type is the whole, the other is the part.

Example:

<table>
<thead>
<tr>
<th>Employee</th>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>eno {PK}</td>
<td>number {PK}</td>
</tr>
<tr>
<td>name</td>
<td>name</td>
</tr>
<tr>
<td>bdate</td>
<td></td>
</tr>
<tr>
<td>title</td>
<td></td>
</tr>
<tr>
<td>salary</td>
<td></td>
</tr>
<tr>
<td>supereno</td>
<td></td>
</tr>
<tr>
<td>dno</td>
<td></td>
</tr>
</tbody>
</table>

Diamond on the side of the whole entity. Department has employees.
**Composition**

*Composition* is a stronger form of aggregation where the part cannot exist without its containing whole entity type and the part can only be part of one entity type.

Example:

<table>
<thead>
<tr>
<th>Project</th>
<th>Has</th>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>number {PK}</td>
<td>0..*</td>
<td>number {PK}</td>
</tr>
<tr>
<td>name</td>
<td></td>
<td>name</td>
</tr>
<tr>
<td>budget</td>
<td></td>
<td></td>
</tr>
<tr>
<td>location [1..3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/totalEmp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Filled diamond on the side of the whole entity to indicate composition. Project must have only one department.

Note: The min-max constraint on the whole side of the relationship (in this case department) must always be 1..1 when modeling composition. Why?
Original ER Model Example

- **Employee**
  - number {PK}
  - name
  - address
  - state
  - city
  - street
  - title
  - salary

- **Department**
  - number {PK}
  - name

- **Project**
  - number {PK}
  - name
  - budget
  - location [1..3]

- Supervises
  - 0..1
  - Supervisor

- Manages
  - 0..1
  - 0..*

- Has
  - 0..*
  - 0..1

- WorksOn
  - 0..*
  - 0..*

- **Supervisee**
  - 0..*

- **Supervisor**
  - 0..1

- **Responsibility**
  - hours

- **Has**
  - 0..1
  - 0..*

- **Has**
  - 0..*
  - 0..*
EER Model Example

Employee
- number {PK}
- name
- address
- state
- city
- street
- title
- salary

Supervisor
- 0..1
- {Optional, AND}

Manager
- Manages
- 0..1

Department
- number {PK}
- name
- Has
- 0..*

Project
- number {PK}
- name
- budget
- location [1..3]
- /totalEmp

WorksOn
- 0..*
- 0..*

Supervises
- 0..*

Manages
- 0..1

Responsibility
- hours
Conclusion

The **Enhanced Entity-Relationship** (EER) model allows for object-oriented design features to be captured. **Generalization** and **specialization** are two complementary processes for constructing superclasses and subclasses. **Participation** and **disjoint constraints** apply to subclasses.

- Participation of a superclass may be **mandatory** or **optional** in a subclass.
- A superclass may only be a member of one subclass (disjoint constraint indicated by OR) or multiple (indicated by AND).

Aggregation and composition are used to model **HAS-A** or **PART-OF** relationships. The features of EER modeling are rarely needed in most database design projects.
Objectives

Given an EER diagram, recognize the subclasses, superclasses, and constraints using the notation.

Explain the difference between the participation constraint and the disjoint constraint.

Explain the difference between aggregation and composition.

Given an EER diagram, list the attributes of each class including attributes inherited from superclasses.