

UML

Rodrigo García Carmona
Universidad San Pablo-CEU
Escuela Politécnica Superior



DATA MODELING WITH UML

DATA MODELING

- Data modelling is the way we represent data in order to manage it.
- Choosing the way we model data depends on the way we are going to use that data.
- Sample data models:
 - **Relational**: To implement DBMS. Very efficient.
 - **XML**: Tree-shaped. Human readable.
- These are **low-level** models, implemented with system concerns.
- As an alternative, we could use **high-level** models:
 - **E-R**: Entity-Relationship model. Already studied.
 - **UML**: *Unified Modelling Language*.
 - We will study it in this lesson. Most popular.
- High-level models are very easy to understand:
 - Can be depicted using “drawings”.
 - Can be translated to a low-level language later.

UML FEATURES

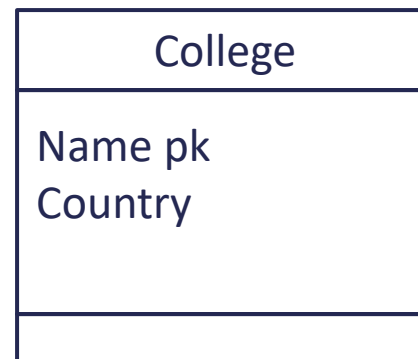
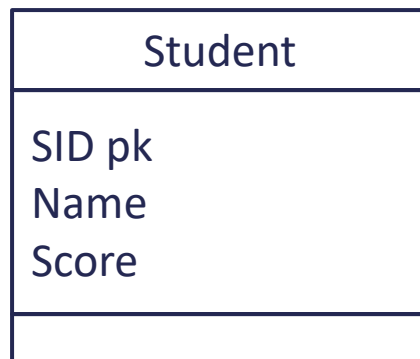
- Designed with object-oriented languages in mind.
- Useful for software architects and managers.
- Broad standard:
 - This course will only cover the part related with data modeling.
- Composed of several diagram types:
 - **Structural:** Static design and analysis.
 - Class, component, package, deployment...
 - **Behavioral:** Dynamic design and analysis.
 - Activity, sequence, state, use case...
 - In this course we will study **class diagrams**.
- **A class diagram can be easily translated to an object-oriented language code.**
- The way object orientation understands data is **very different** to the relational model's approach.

DATA MODELING WITH UML

- **7 core concepts:**
 - Classes.
 - Associations.
 - Association classes.
 - Composition.
 - Aggregation.
 - Inheritance (generalization and specialization).
 - Realization.
- Association, composition, aggregation, generalization, specialization and realization are subtypes of a generic idea called relation.
- **The UML “relation” should never be confused with the relational model “relation”.**
- We will study how to use UML to **model data**.

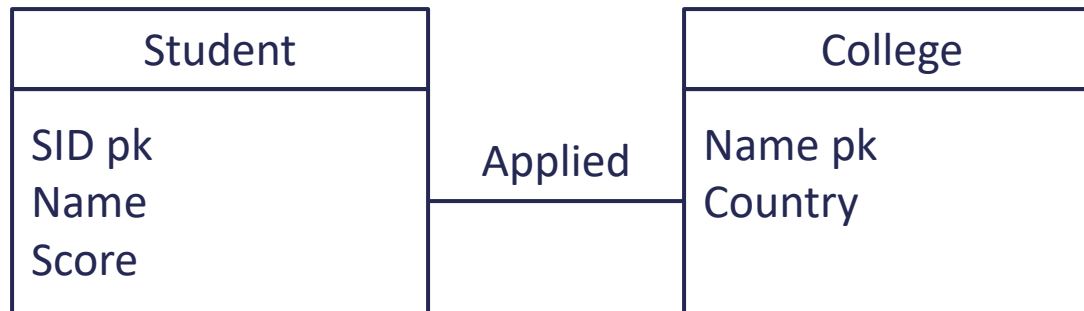
CLASSES

- Model a component type.
- Same as classes in object orientation. Made of **name**, **attributes** and **methods**.
- Similar to entities in E-R diagrams.
- **In order to model data** we will take into account the following:
 - We must add “**pk**” to the primary key.
 - There’s a school of thought that postulates that data shouldn’t have behavior. If followed, the method block should be omitted.




ASSOCIATIONS

- Generic relationships between instances (objects) of two classes.
- Represented with solid lines.
- Optionally, they could use a name as identifier.
- Similar to relationships in E-R diagrams.



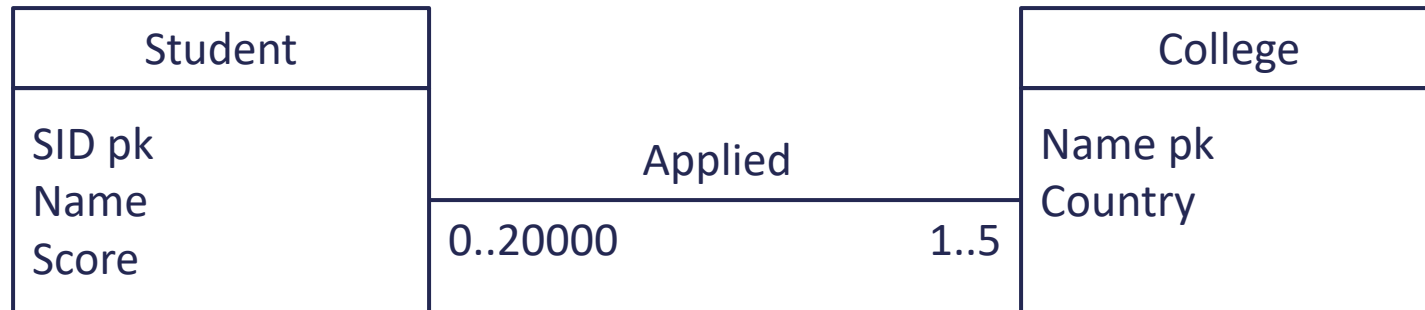
ASSOCIATIONS MULTIPLICITY

- The objects amount at each side of the association is shown with an interval.
- **Each side is treated independently.**
- The amount in one side depicts how the other side “sees” it.
- Expressed as follows: *minimum amount .. maximum amount*.
 - Minimum and maximum could be the same.
 - In this case we can remove the “..”
 - “*” indicates any amount.
- Examples:
 - 3..5: Between three and five. 
 - 1..1: Just one. Also written as “1”.
 - 0..*: Without limits. Also written as “*”.
 - 1..*: More than one but without upper limit.



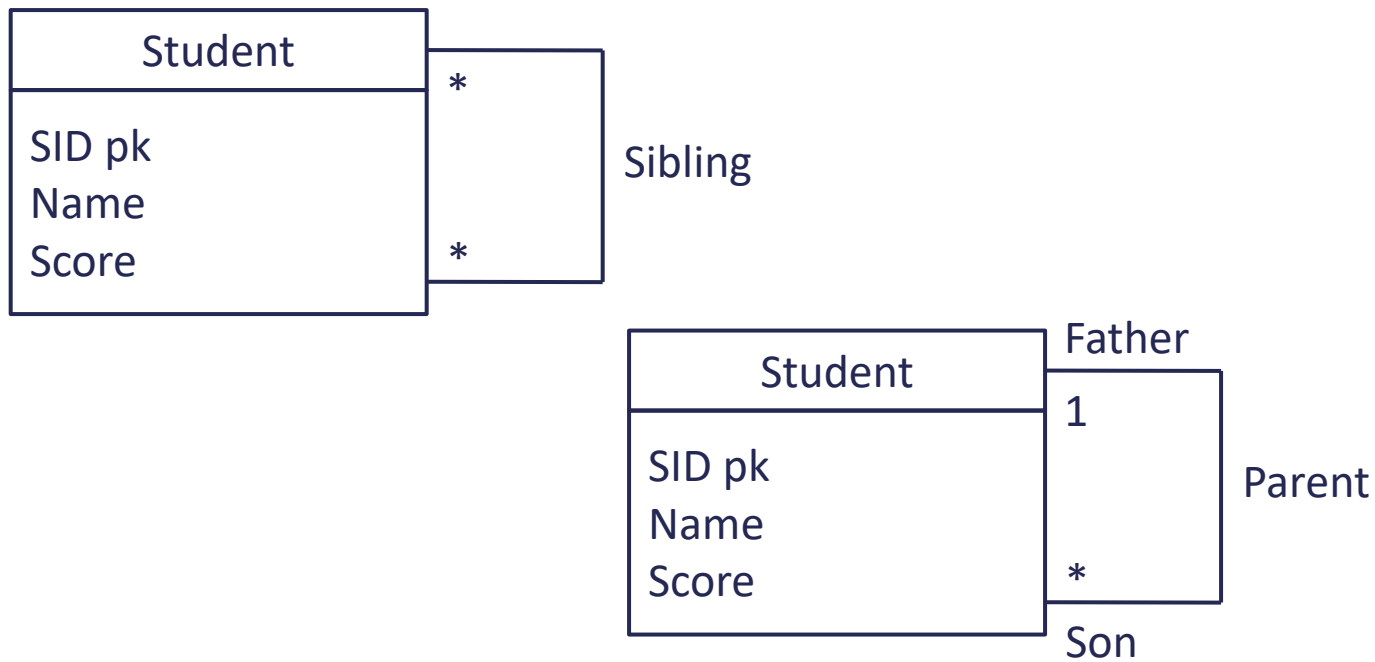
ASSOCIATION EXAMPLE

- Students can apply to up to 5 universities, and they must apply to at least 1.
- A college can't have more than 20.000 applicants at any given moment.



ASSOCIATION WITH THE SAME CLASS

- An association can have the same class in both ends.
- If the association is not symmetric it's a good idea to indicate which roll each end fulfills.

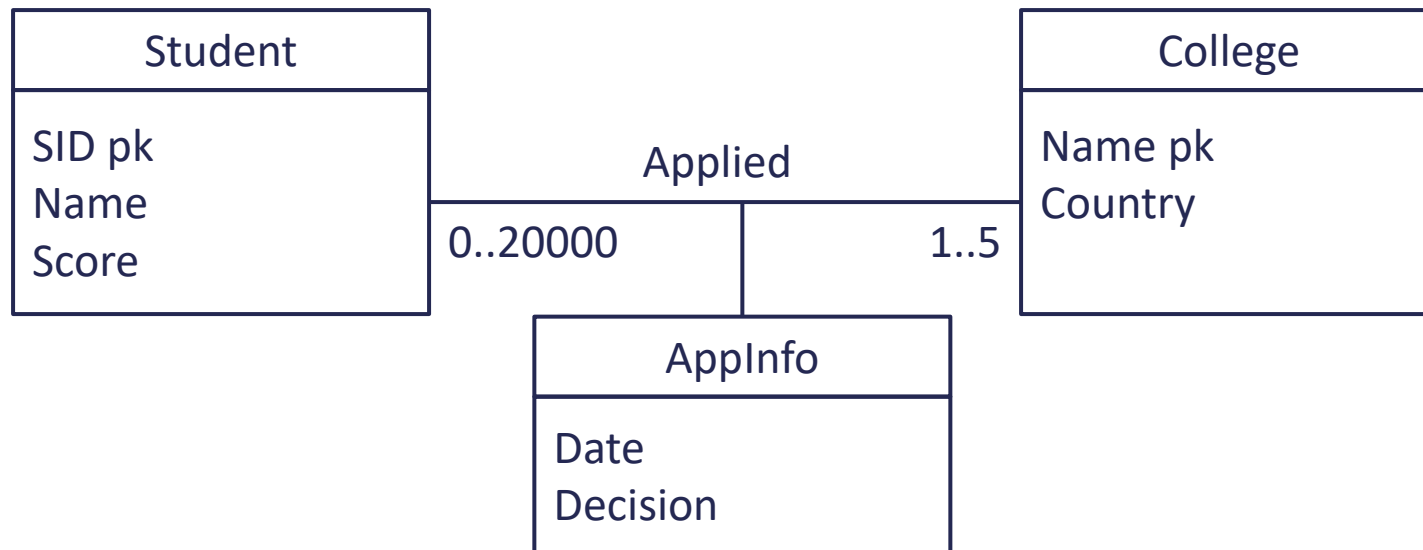


ASSOCIATION TYPES ATTENDING TO ITS MULTIPLICITY

- Associations can be **divided in three special subtypes** attending to its multiplicity:
 - *One-to-One*:
 - 0..1 in both ends.
 - *Many-to-One*:
 - 0..1 in one end and 0..* in the other.
 - *Many-to-Many*:
 - 0..* in both ends.
- These subtypes are called **complete** if:
 - *Complete One-to-One*: 1 in both ends.
 - *Complete Many-to-One*: 1 in one end and 1..* in the other.
 - *Complete Many-to-Many*: 1..* in both ends.

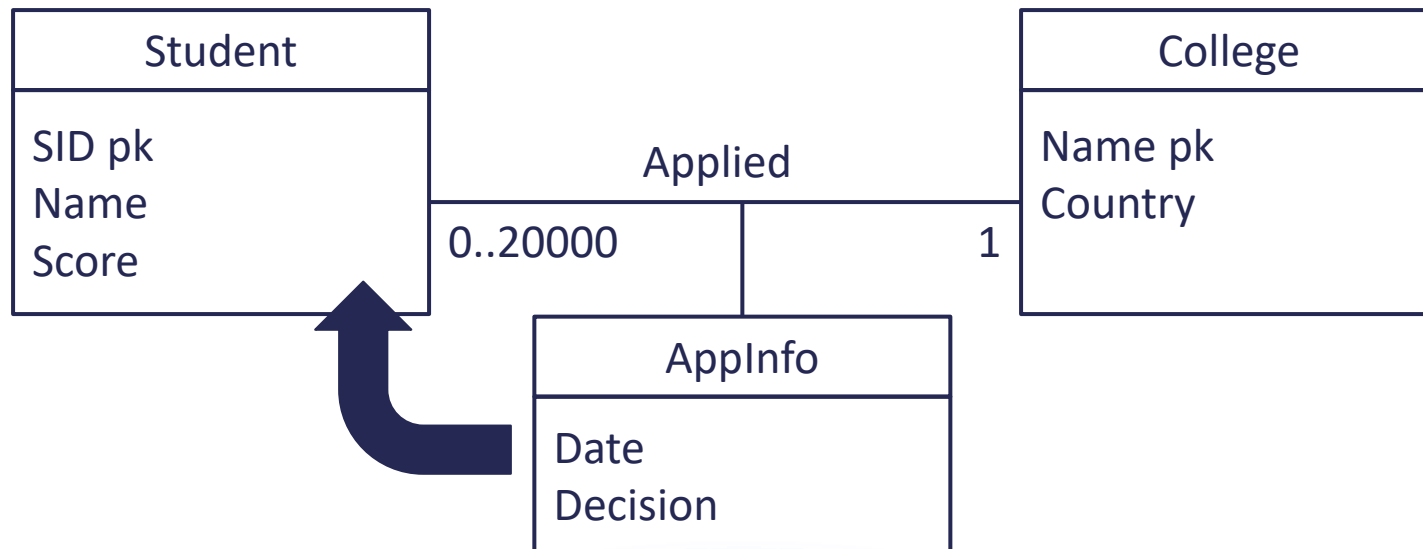
ASSOCIATION CLASSES

- Classes that don't represent objects, but characteristics of a relationship between two other classes.
- They provide extra detail.
- They have attributes but no primary key.

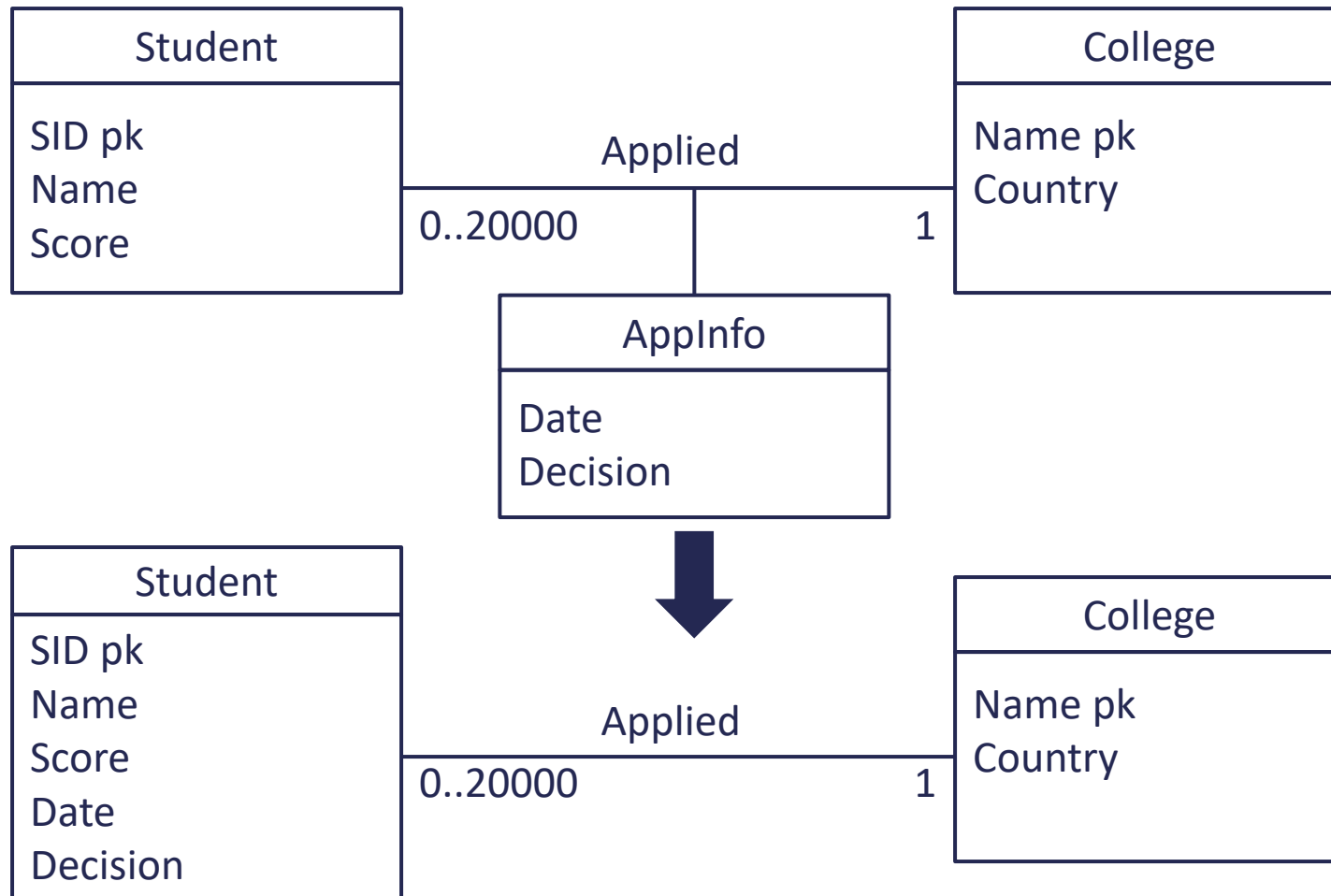


REMOVE ASSOCIATION CLASSES

- Sometimes association classes are not really needed:
 - If one end's multiplicity is 0..1 or 1.
 - Specially recommended if the multiplicity is 1.
- We can put the association class' attributes inside one of the two classes linked by it.



ASSOCIATION CLASS REMOVAL EXAMPLE



COMPOSITION AND AGGREGATION

- Association's special cases. Both represent a relationship between the parts and the whole.
- **Aggregation:**
 - *Many-to-One* association's special case.
 - The parts and the whole don't need each other.
 - They make sense by themselves. The whole **"uses"** the parts.
 - An empty diamond is used instead of 0..1.
 - If parts are not explicitly numbered, an * is assumed.
 - Parts must have "pk".
- **Composition:**
 - *Many-to-One* association's special case.
 - The parts and the whole need each other.
 - They **don't** make sense by themselves. The whole **"owns"** the parts.
 - A filled diamond is used instead of 1.
 - If parts are not explicitly numbered, an * is assumed.
 - Although parts don't need to have "pk", it's **strongly recommended**.

COMPOSITION AND AGGREGATION EXAMPLES

- Aggregation:



- Composition:



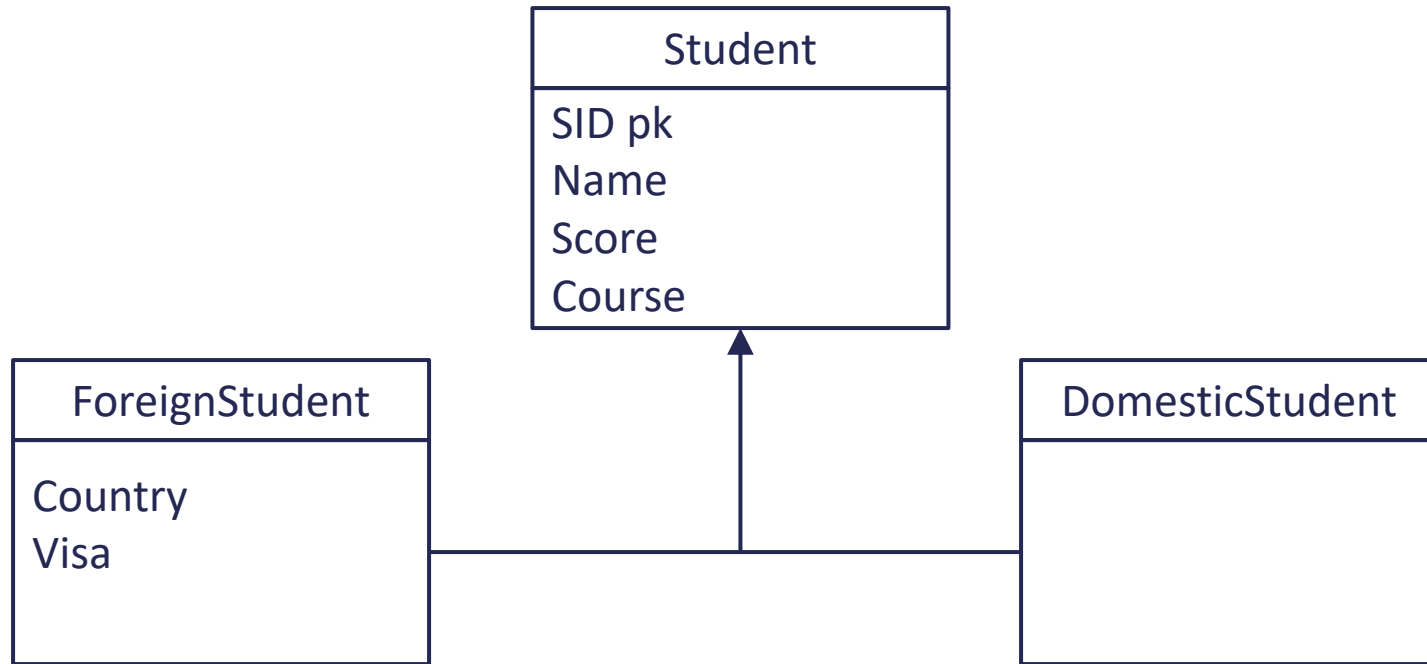
INHERITANCE

- Inheritance between two classes is depicted using a “solid-headed” arrow.
- Inheritance is made up of two relationships:
 - **Generalization:** The **superclass** (or parent class) is the more generic version.
 - The class pointed by the arrow.
 - **Specialization:** The **subclass** (or child class) is the more specific version.
- Multiplicity is not specified here.
 - We are dealing with objects, not classes.
- The subclass owns all attributes, associations, compositions and aggregations of its superclass.
- Several arrows can be put together for clarity’s sake.
- Subclasses don’t need “pk”.

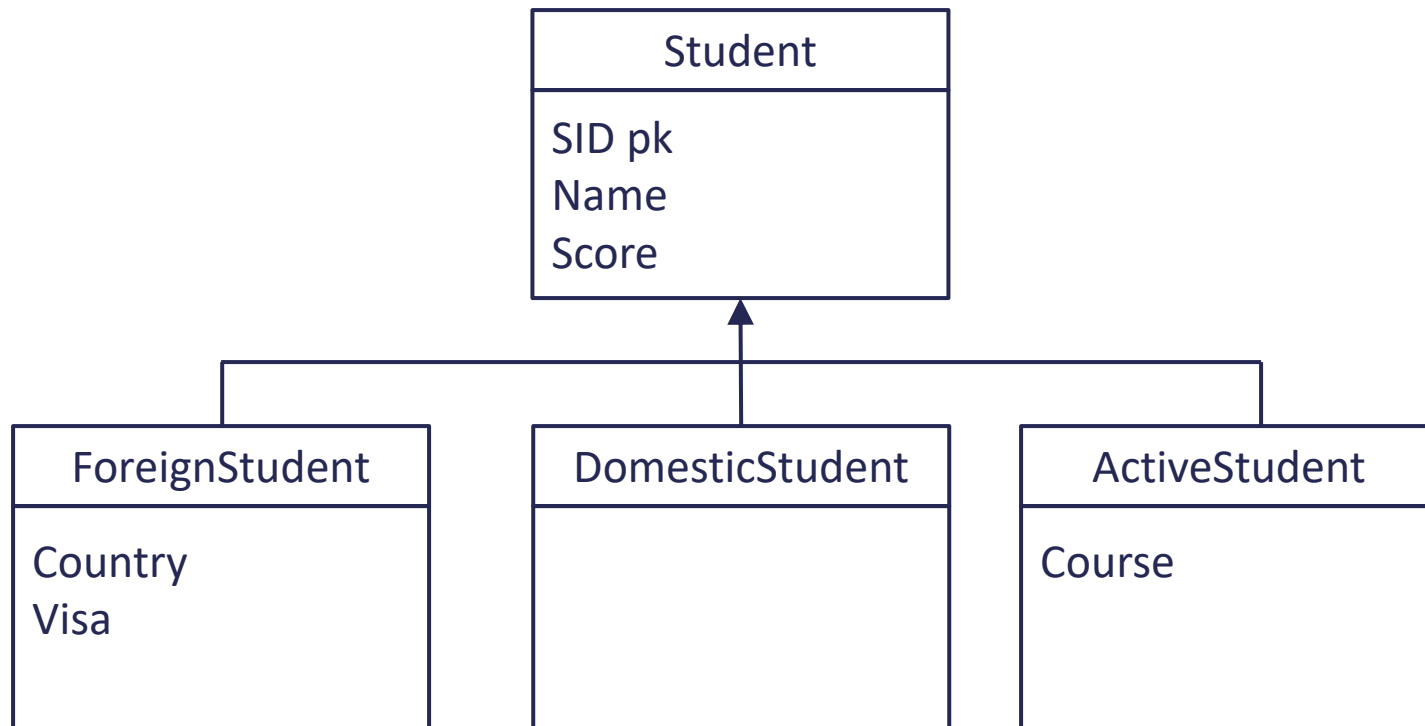
INHERITANCE TYPES

- There's a two-dimension classification for inheritance:
- **Completeness:**
 - **Complete:** Every object instance of a superclass is also instance of at least one of its subclasses.
 - **Incomplete (partial):** An object can be an instance of the superclass without being an instance of one of its subclasses.
- **Exclusivity:**
 - **Disjoint (exclusive):** If an object is an instance of a subclass it can't be an instance of another subclass of the same superclass.
 - **Overlapping:** An object can be an instance of several subclasses of the same superclass at the same time.
- Completeness and exclusivity are depicted in UML diagrams using brackets: “{}”

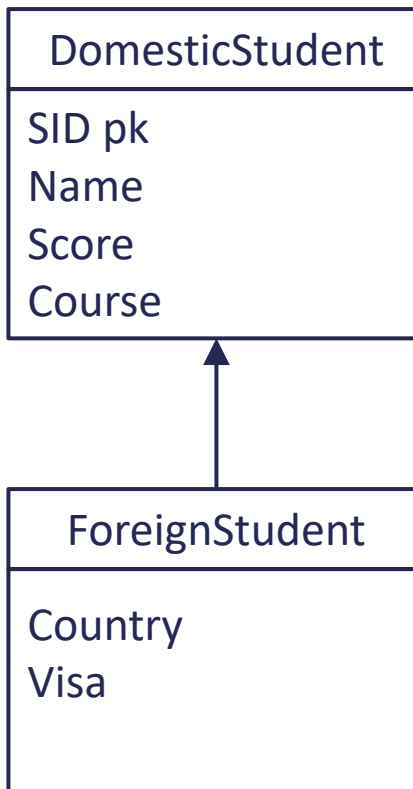
COMPLETE AND DISJOINT INHERITANCE EXAMPLE



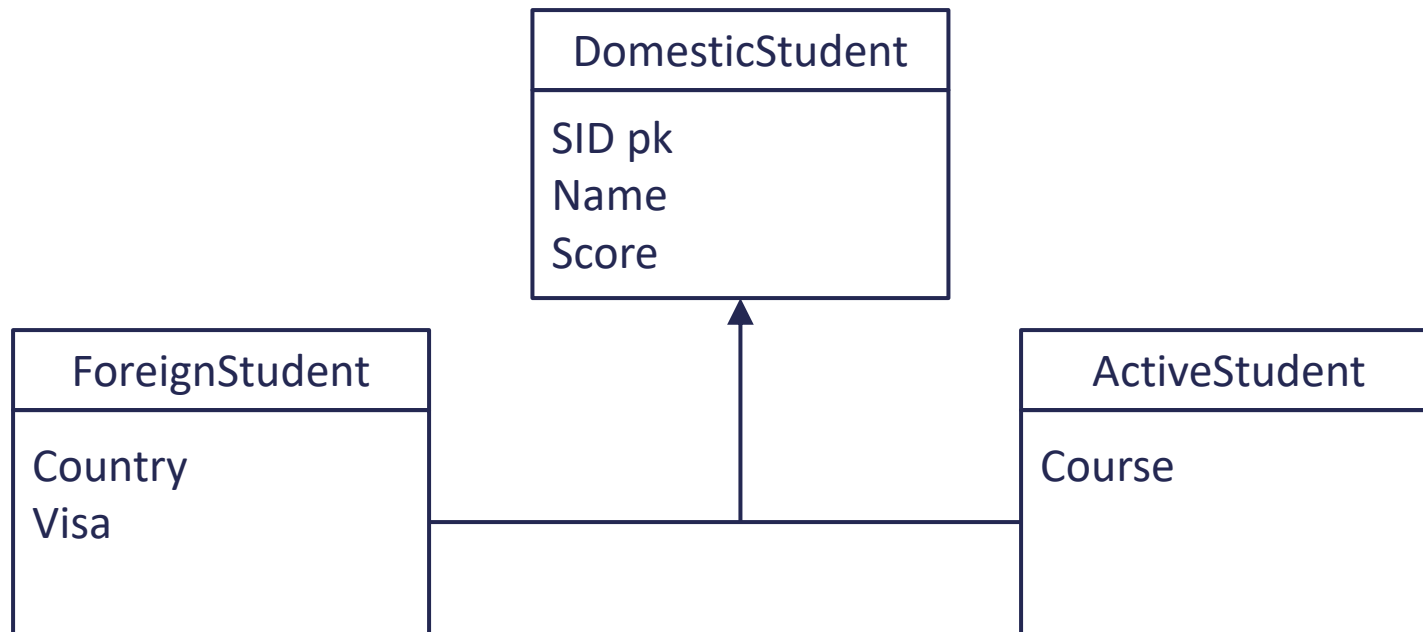
COMPLETE AND OVERLAPPING INHERITANCE EXAMPLE



INCOMPLETE AND DISJOINT INHERITANCE EXAMPLE



INCOMPLETE AND OVERLAPPING INHERITANCE EXAMPLE



REALIZATION: INTERFACES

- To understand realization we must understand the idea of an **interface**.
- UML defines an interface as a special kind of class that:
 - Defines a functionality that other class must implement.
 - **Doesn't provide any functionality by itself.**
 - Is a “contract” that must be fulfilled by the implementing class.
 - Doesn't have attributes.
- We can view an interface as a “template” to build classes.
- It's represented as a class with the `<<interface>>` keyword.
- In the “data shouldn't have behavior” school of thought **interfaces aren't used:**
 - Interfaces are designed to define a behavior.
 - Interfaces doesn't have attributes, only methods.
 - Interfaces can't be instantiated.

REALIZATION

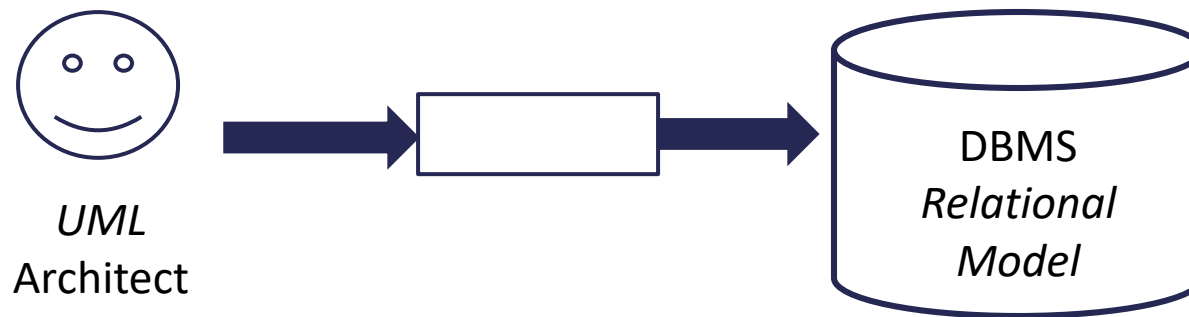
- Realization is a relationship established between an interface and the class that implements it.
- It's depicted using a dotted arrow with an "empty head".
- Is conceptually similar to the complete disjoint inheritance.



UML TO RELATIONAL MODEL TRANSLATION

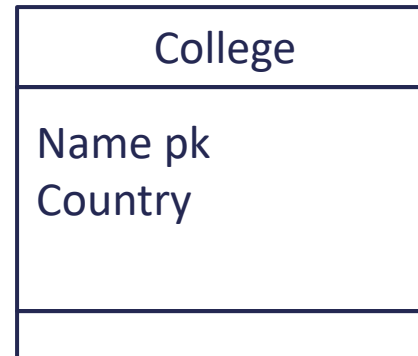
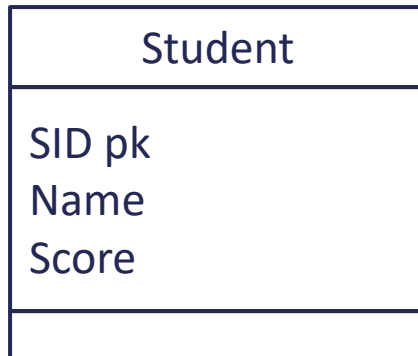
UML TO RELATIONAL MODEL TRANSLATION

- UML is easily understood by humans.
- But it's still necessary to translate UML to a relation model.
 - The relational model is used by most DBMS.
 - Provides a higher efficiency.
- This translation can be semiautomated.
 - As long as all "standard" classes have a "pk".



CLASSES TRANSLATION

- Each class is a table.
- The “pk” attribute is the primary key.
- Each attribute is a column.

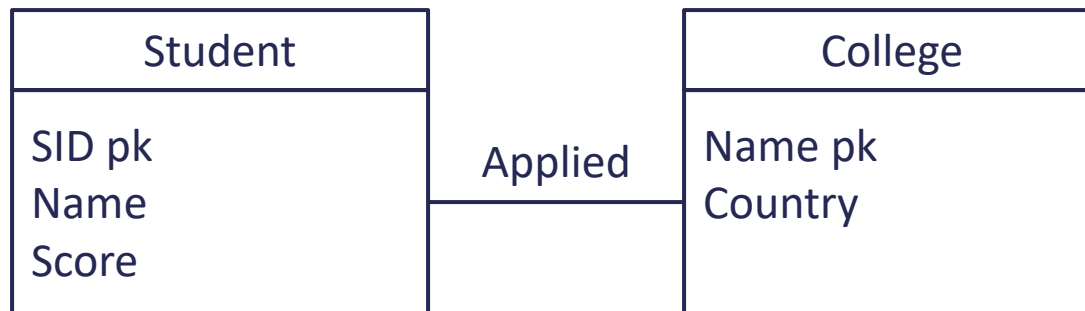


Student(SID, Name, Score)

College(Name, Country)

ASSOCIATIONS TRANSLATION

- Each association is a table.
- This table has one column for each related class, pointing to its “pk”. These columns are foreign keys.



Student(SID, Name, Score)

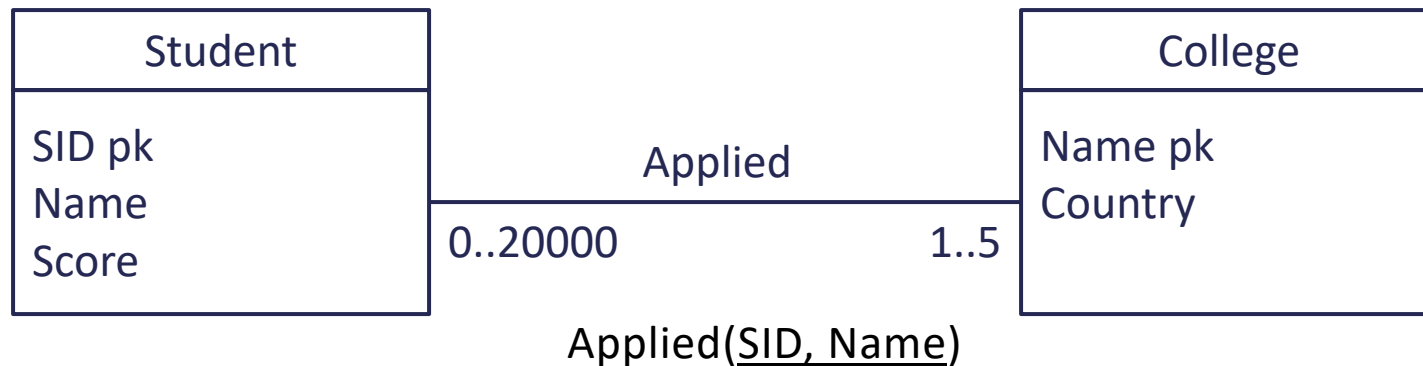
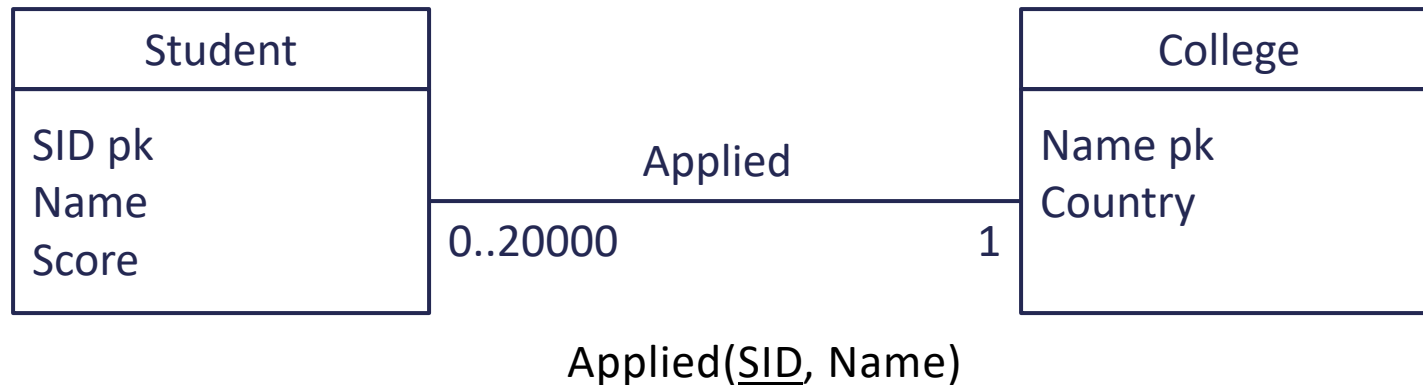
College(Name, Country)

Applied(SID, Name)

ASSOCIATIONS TRANSLATION: PRIMARY KEYS

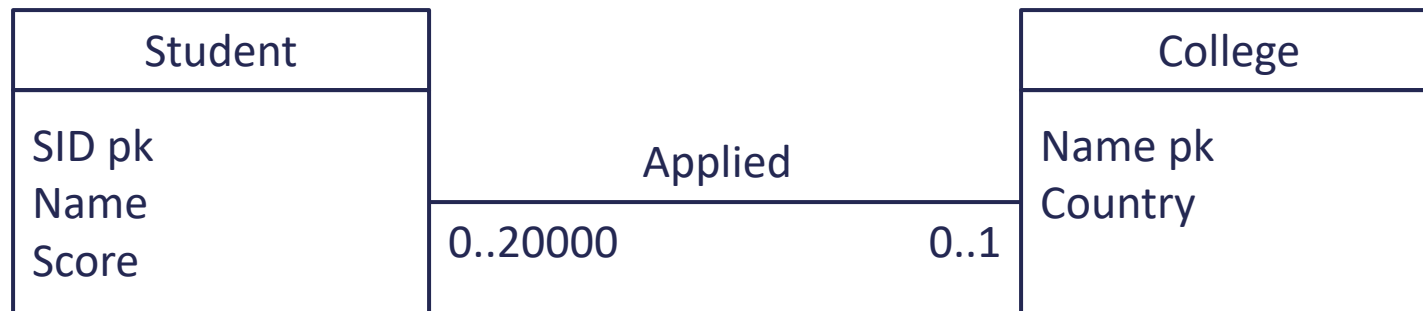
- Which primary key we choose for the association's table depends on the association's multiplicity:
 - **Both ends are 0..1 or 1:**
 - The primary key may be the "pk" of any of the ends' classes.
 - This table is not really needed (as we'll see later).
 - **Only one end is 0..1 or 1:**
 - The primary key must be the "pk" of the end that doesn't have the 0..1 or 1 multiplicity.
 - This table is not really needed (as we'll see later).
 - **Neither end is 0..1 or 1:**
 - The primary key must be the combination of both "pk".

ASSOCIATIONS TRANSLATION EXAMPLES: PRIMARY KEYS



UNNEEDED TABLES

- It's not always necessary to create a new table to represent an association.
- Sometimes the association's data can be merged to one of the two classes' tables.
- This is feasible when at least one of the ends have a 0..1 or 1 multiplicity.
 - If multiplicity is 0..1 the table must support NULL values for that attribute.



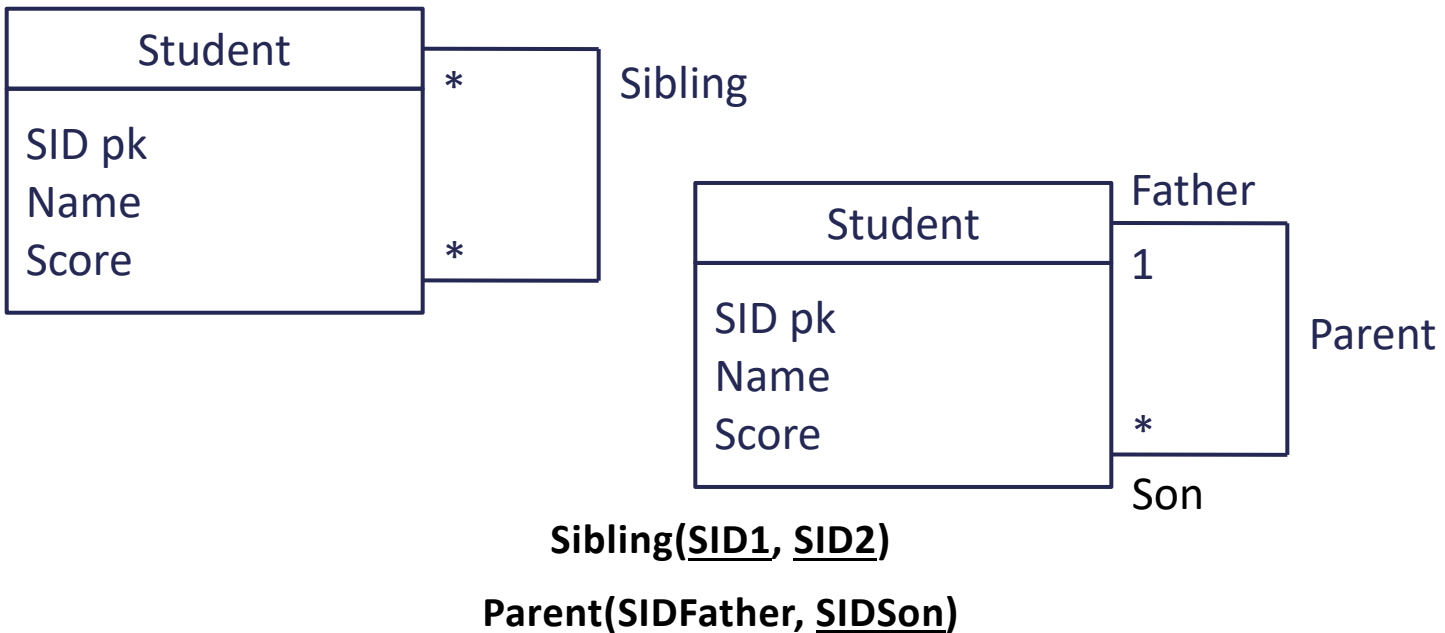
Student(SID, Name, Score, **College.Name)**

College(Name, Country)

~~Applied(SID, Name)~~

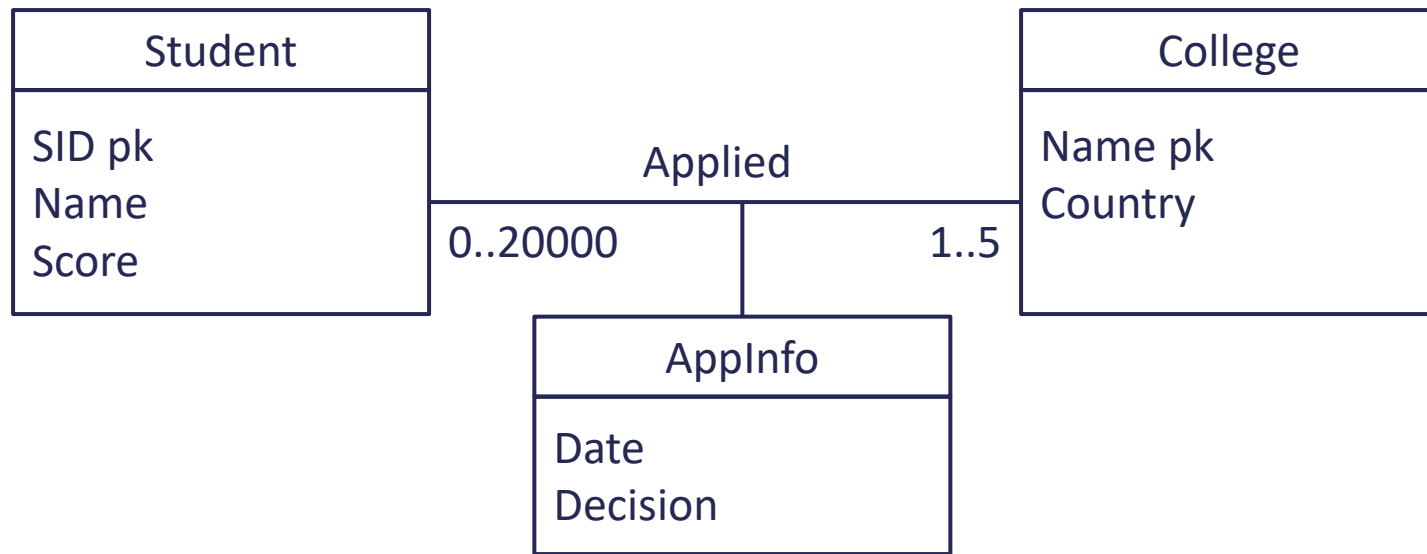
SAME CLASS ASSOCIATIONS TRANSLATION

- The table that represents the association features two instances of the class' table "pk".
- As explained before, some tables could not be really needed, and its contents could be merged into the class.



ASSOCIATION CLASSES TRANSLATION

- We add the association class' attributes to the table that represents the association.



Student(SID, Name, Score)

College(Name, Country)

Applied(SID, Name, Date, Decision)

AGGREGATION AND COMPOSITION TRANSLATION

- The relational model's semantics doesn't support UML aggregation and composition.
 - **They are considered standard associations.**
- Aggregation is translated as an association with 0..1 multiplicity in one end.
 - This table must support NULL attributes.
- Composition is translated as an association with 1 multiplicity in one end.

AGGREGATION AND COMPOSITION

TRANSLATION EXAMPLES

- Agregation: Professor(PID, Name, Course, College.Name)



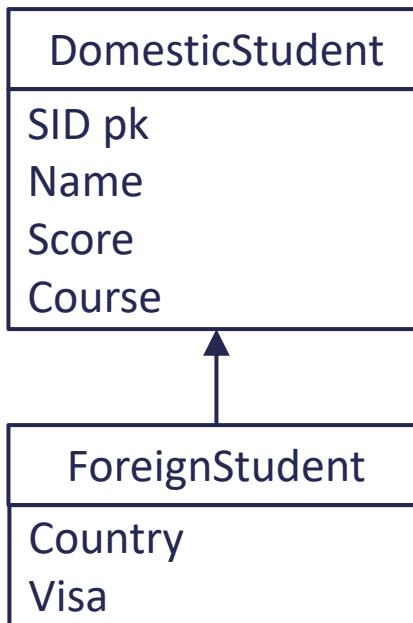
- Composition: Department(Name, Building, College.Name)



INHERITANCE TRANSLATION

- The relational model doesn't feature inheritance. We are forced to "improvise" a solution.
- There are three feasible approaches:
 - **One table for the superclass and one table for each subclass:** Each subclass' table contains the parent's "pk" and the new attributes.
 - Useful for disjoint and incomplete inheritance.
 - **One table for each subclass:** Each subclass' table contains all attributes of its class and the superclass.
 - Useful for disjoint and complete inheritance.
 - **One table for all classes:** The table contains all attributes of the superclass and all subclasses.
 - Useful for overlapping inheritance.
- Which is the best option depends on the specific situation we're in.

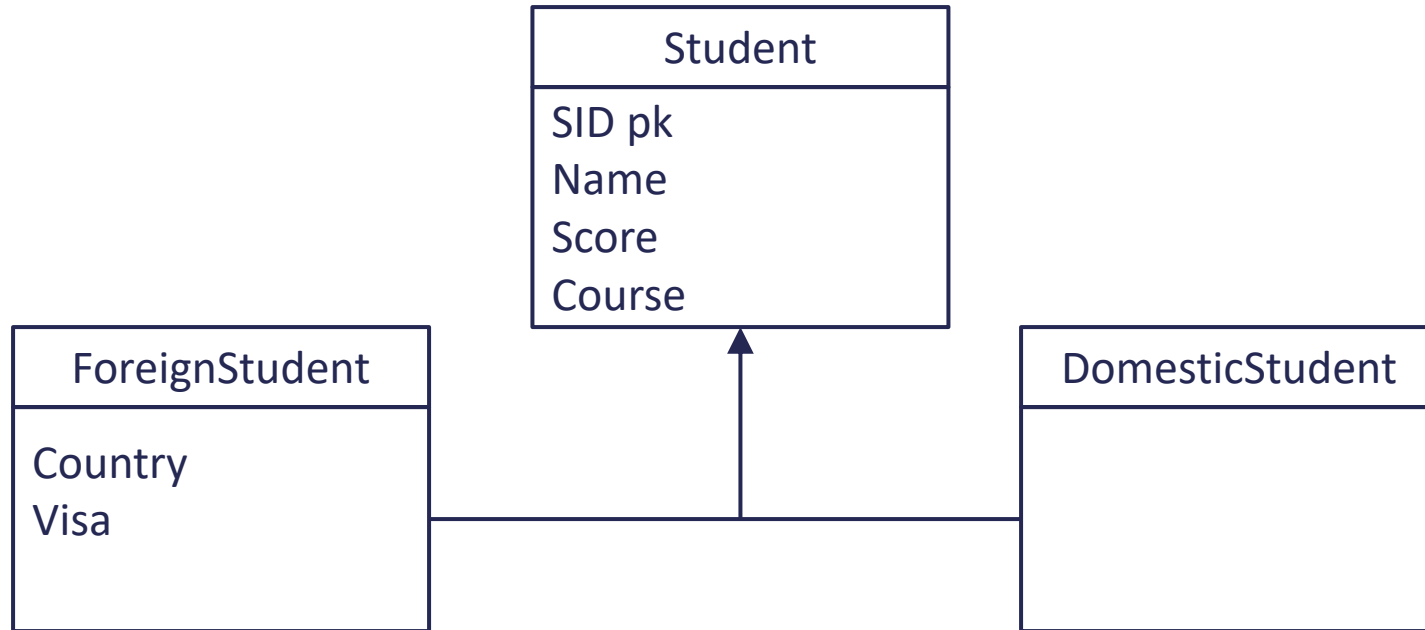
DISJOINT AND INCOMPLETE INHERITANCE TRANSLATION EXAMPLE



DomesticStudent(SID, Name, Score, Course)

ForeignStudent(SID, Country, Visa)

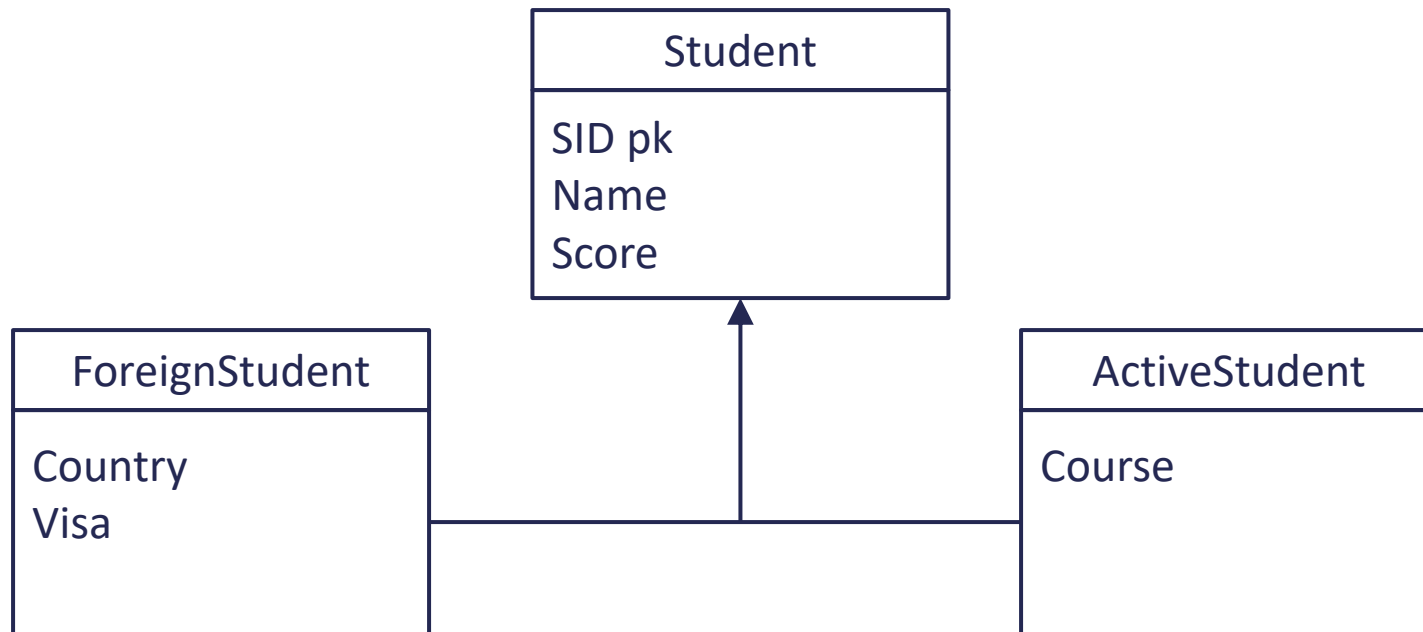
DISJOINT AND COMPLETE INHERITANCE TRANSLATION EXAMPLE



ForeignStudent(SID, Name, Score, Course, Country, Visa)

DomesticStudent(SID, Name, Score, Course)

OVERLAPPING INHERITANCE TRANSLATION EXAMPLE



Student(SID, Name, Score, Country, Visa, Course)

UML AND JAVA

UML AND JAVA

- Since UML was designed to represent object-oriented data, its translation to Java is easy.
- **Most correspondences are direct:**
 - UML classes as Java classes.
 - Associations as using objects from other classes.
 - Composition as an association created in the constructor.
 - Aggregation as an association with the proper multiplicity.
 - Inheritance as Java inheritance through “extends”.
 - Realization as Java inheritance through “implements”.
- There’s software that can **automatically generate** skeleton Java code from UML and *vice versa*.
 - One of them is the EMF framework.

OBJECT RELATIONAL MAPPING

- **ORM** (Object Relational Mapping) is a way to translate objects into relational tables, and vice-versa.
 - Follows the same translations explained in this unit, but with objects of a programming language instead of UML class diagrams.
 - Realizes the CRUD (Create, Read, Update, Delete) operations.
 - **Identity** ($a==b$) and **equivalence** ($a.equals(b)$) are **not** the same.
- **JPA** (Java Persistence API) is a Java ORM framework.
 - It's included with the JDK since version 1.6.
 - It's a set of interfaces that need to be implemented by a particular solution. We'll use **EclipseLink**.
 - It's configured with XML documents and annotations.
 - Classes that represent data are called **entities** that are controlled by an **entity manager**.
 - Accepts SQL and its own SQL-like object-oriented query language: JPL

ORM PROBLEMS (I)

- ORM tends to makes life easier... but it's not easy:
polymorphism, inheritance, association vs. composition...
- It's said that **ORM is the Vietnam war of Computer Science**
- ORM isn't a silver bullet. It introduces it's own **set of problems**:
 - **Who owns the schema?**
 - The relational model
 - The object-oriented model
 - Both? *Dual schema problem*: Both views need to be kept updated
 - **Identity issues:**
 - Identity vs. equality
 - Several sessions against the same DBMS
 - Isolation and concurrency:
CAP (consistency, availability, partitioning) theorem

ORM PROBLEMS (2)

- ORM isn't a silver bullet. It introduces its own set of problems:
 - **Retrieval mechanism concern.** Alternatives:
 - **Query by example:** Sample object
 - **Query by API:** Query objects
 - **Query by language:** SQL-like language.
 - **Partial object problem:**
 - **Object orientation:** All object fields are loaded.
 - **Relational model:** Only the desired columns are loaded.
 - **Compromise?:** Lazy loading