System Analysis and Design

(IT3120E)

Quang Nhat Nguyen

quang.nguyennhat@hust.edu.vn

Hanoi University of Science and Technology

School of Information and Communication Technology

Academic year 2022-2023

Content:

Structure analysis

- Goal of the structure analysis
- Objects and classes
- Determining domain classes
- Determining classes involving in the use cases
- Illustrative example exercise

Goal of the structure analysis

- <u>Preliminary</u> determine the main classes that make up the system
  - This is <u>not the complete (i.e., final) version</u> of the classes

Objects and classes

- Definition and representation of objects and classes
- Attributes
- Operations
- Relations
  - Dependency
  - Generalization
  - Association
- Class diagram and Object diagram

Definition of object and class (1)

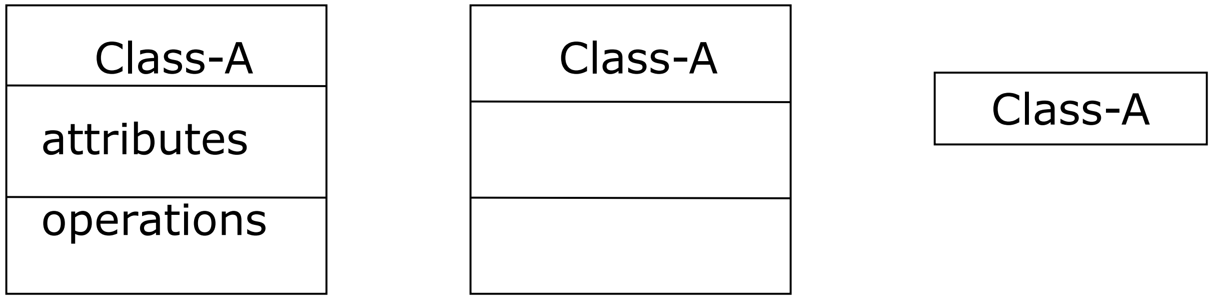
- An **object** (in IT) is an abstract representation of a (physical or conceptual) entity that has a clear **identity** and boundary in the real world, including the **state** and **behavior** of that entity, aiming at simulating or controlling that entity
  - The **state** of an object is represented by a set of **attributes**. At a time, each attribute of the object has a certain value.
  - The **behavior** of an object is represented by a set of **operations**, which are services it can perform when requested by another object.
  - The **identifier** of an object is what distinguishes it from another

Definition of object and class (2)

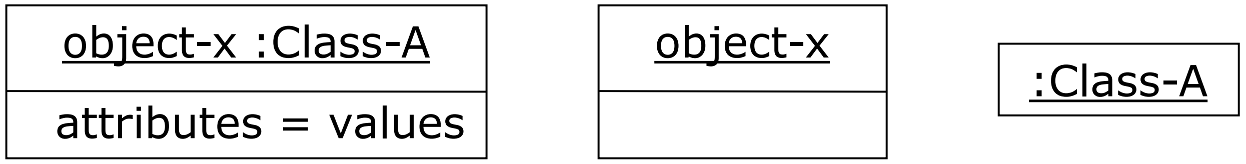
- A **class** is a description of a set of objects that share the same attributes, operations, relations, constraints, and semantics
- A class is a type, and each object of the class is an instance

Representation of objects and classes

- Representation of classes



- Representation of objects



Attributes (1)

- An **attribute** is a named property of a class that takes a value for each object of that class at a time
- Syntax of an attribute:

```
[Visibility] [/] Name [: Type]
[Multiplicity] [= Initial value]
[{Property-string}]
```

Attributes (2)

```
[visibility] [/] name [: type] [multiplicity]
[= initial value] [{property-string}]
```

- **Visibility** tells how the attribute is seen and used by other classes
  - **Private**, denoted by '**-**', if the attribute is not accessible from any other class
  - **Protected**, denoted by '**#**', if the attribute is only accessible from classes that inherit the current class
  - **Package**, denoted by '**~**', if the attribute is accessible from classes belonging to the same (i.e., the most inner) package of the current class
  - **Public**, denoted by '**+**', if the attribute is accessible from any other class

Attributes (3)

`[Visibility] [/] Name [:` **`Type`**`] [`**`Multiplicity`**`]`
`[= Initial value] [{Property-string}]`

- **Type** is the type of the attribute's values
  - Basic types like Integer, Real, Boolean
  - Structured types like Point, Area, Enumeration
  - Type is another class
- **Multiplicity** is the number of possible values assigned to the attribute
  - Example: [0..1] shows that the attribute is optional (i.e., it may take no or one value)

Attributes (4)

`[Visibility] [/] Name [: Type] [Multiplicity] [`**`= Initial value`**`] [`**`{Property-string}`**`]`

- **Initial value** is the default value assigned to the attribute when an object is instantiated from that class
- **Property-string** defines the values that can be assigned to the attribute, usually used for enumeration-typed attributes

- Example: **status: Status = unpaid {unpaid, paid}**

Attributes (5)

- An attribute may have **class scope** if it reflects the general characteristics of the class
  - Class-scoped attributes must be underlined
  - Example: Attribute <u>`number-of-invoices`</u> of the class `Invoice`
- An attribute is **derived**, if its value is calculated from the values of other attributes of the class
  - Derived attributes must include a slash '/' at the beginning
  - Example: `/age` (while the attribute `/birthday` exists)

Operations (1)

- An operation is a service that an object can respond to when requested (via a message)
- Operations are implemented as methods
- Syntax of an operation:

```
  [Visibility] Name [(Parameter list)] [:
 Return type] [{Property-string}]
```

  - **Visibility** is defined the same as for attributes
  - **Parameter list** is a list of parameters, separated by commas, each of the form:

```
    [Direction] Name : Type [= Default value]
```

Operations (2)

  - **Direction** may be either of `in`, `out`, `inout` or `return` depending on the parameter is: may not be modified (`in`), may be modified to communicate information to the caller (`out`), may be modified (`inout`), or to return the result to the caller (`return`)
  - **Default value** is the value to be used when the corresponding parameter is missing in the call

- **Property-string** includes pre-conditions, post-conditions, effects on object state, etc.

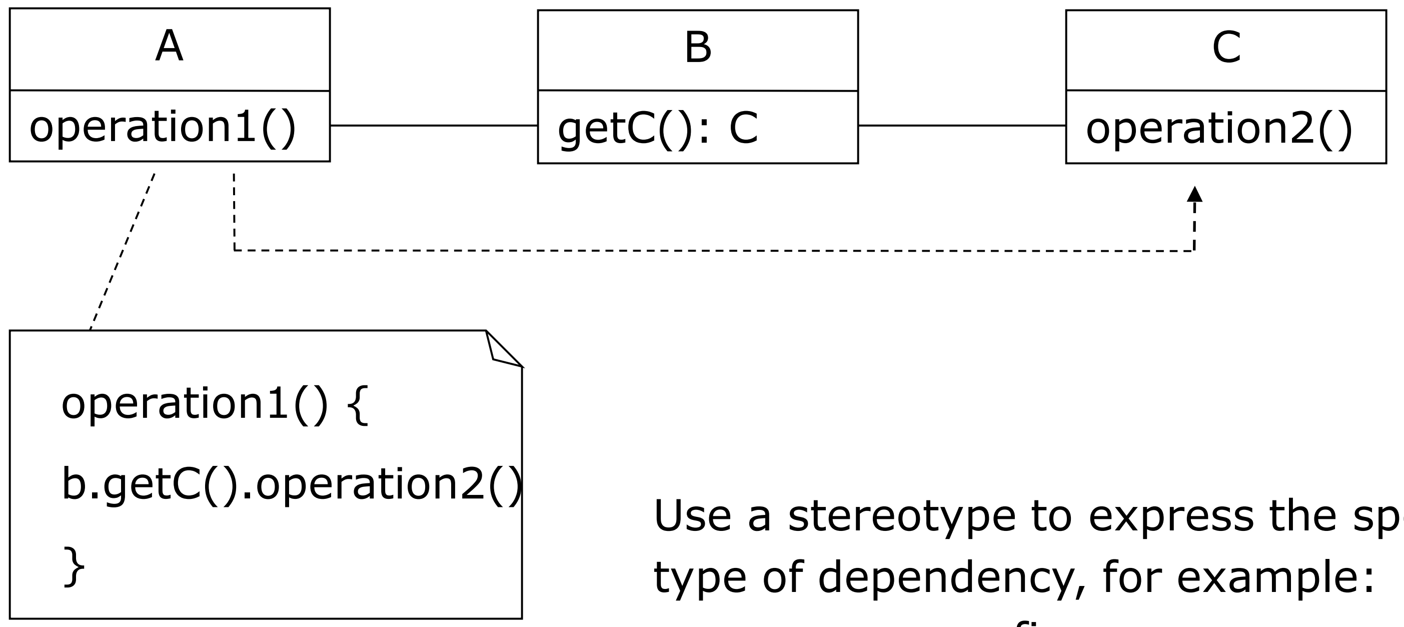
Relations

- There are three possible relationships between classes:
  - Dependency
  - Generalization
  - Association

Dependency relation (1)

- Dependency relation is used to express **a (dependent) class that is affected by any changes in another (independent) one**
  - The opposite direction is not necessarily
- Usually, the dependent class needs to use the independent one to specify or implement it
- UML represents the dependency relation with a dashed arrow from the dependent class to the independent one

Dependency relation (2)



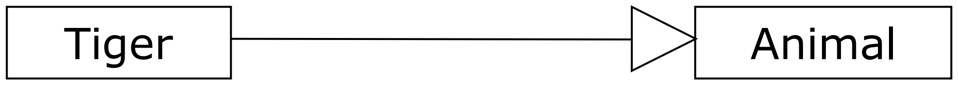
Use a stereotype to express the specific type of dependency, for example: <<use>>, <<refine>>

Generalization relation (1)

- **Generalization** is the extraction of common characteristics of many classes to form a simpler (i.e., more generalize) class
  - Called a super-class
- On the contrary, **specialization** is the enhancement (i.e., addition) that adds some new characteristics from a given class, forming a more specific class
  - Called a sub-class
- A class can have no, one or more super-class(es)
  - A class with only one super-class is called simple inheritance
  - A class with multiple super-classes is called multiple inheritance
- A class that has no super-class and has sub-classes is called a root class (i.e., base class)

Generalization relation (2)

- The term “**inheritance**” is commonly used in programming languages to describe a sub-class that *has all the attributes, operations, and relations* described in its super-class
  - Sub-classes can add new (own) attributes, operations, and relations
  - A sub-class can re-define (i.e., overwrite) an operation of the supper-class: **Polymorphism**
- Representation of generalization relation:



Generalization relation (3)

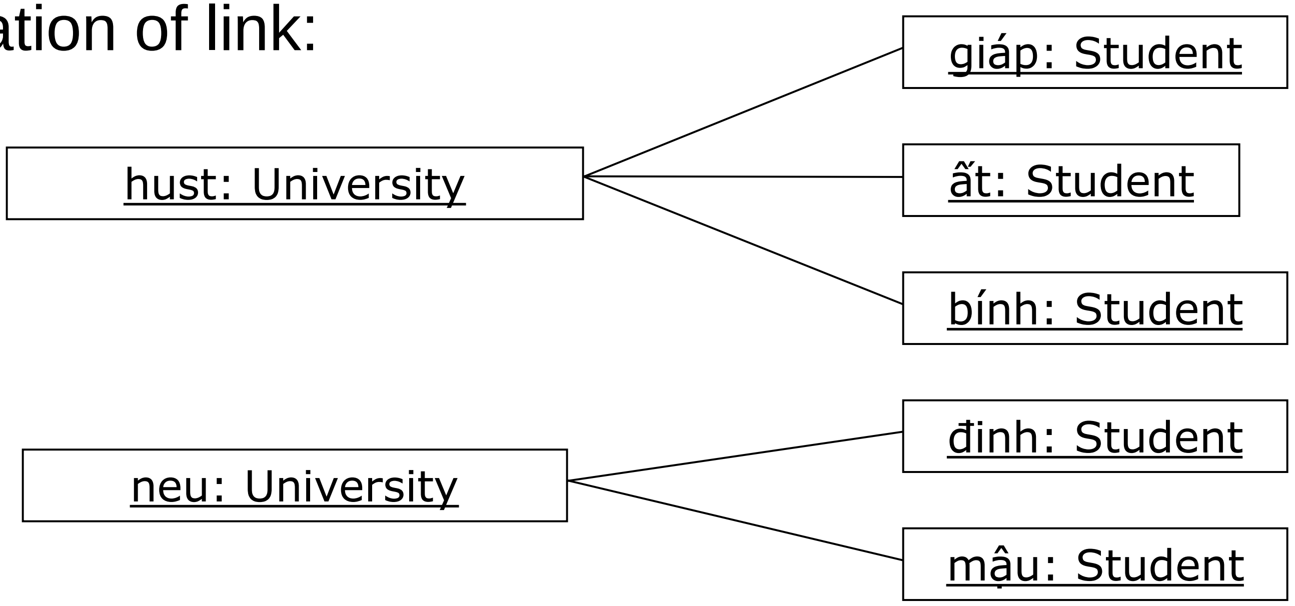
- Through generalization, we can make **abstract classes**, i.e., those classes that have no instance objects, but are only used to describe common characteristics of sub-classes
- An abstract class usually contains **abstract operations**, i.e., those operations with only title (i.e., name) and without implementation
  - Abstract operations must be redefined (i.e., overwritten) and implemented in sub-classes
  - The name of the abstract class and the title of the abstract operation must be italicized and may include the property string **{abstract}**
- Example: `Animal` is an abstract class generalized from the classes `Horse`, `Tiger`, `Bat`. It has an abstract operation `sleep()`. This abstract operation will be specifically implemented in the sub-classes `Horse`, `Tiger`, `Bat` in different ways, but keeping the same name `sleep()`

Association relation (1)

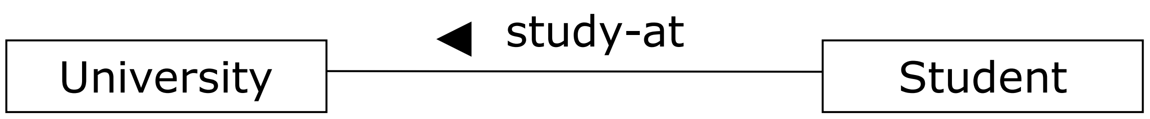
- **Link** represents some actual relation **between instances (objects)** of two classes
  - Example: wife-husband link, teacher-student link, motorbike-owner link, customer-invoice link, etc.
- **Association** is a relation **between two classes**, consisting of a set of links of the same type (i.e., of the same meaning) between instances of those two classes
  - This is a relation (in the mathematical sense) between two sets (two classes)

Association relation (2)

- Representation of link:

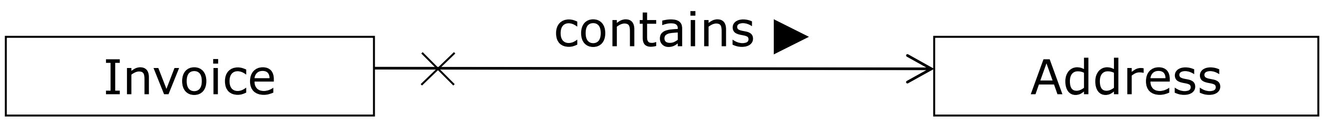


- Representation of association:



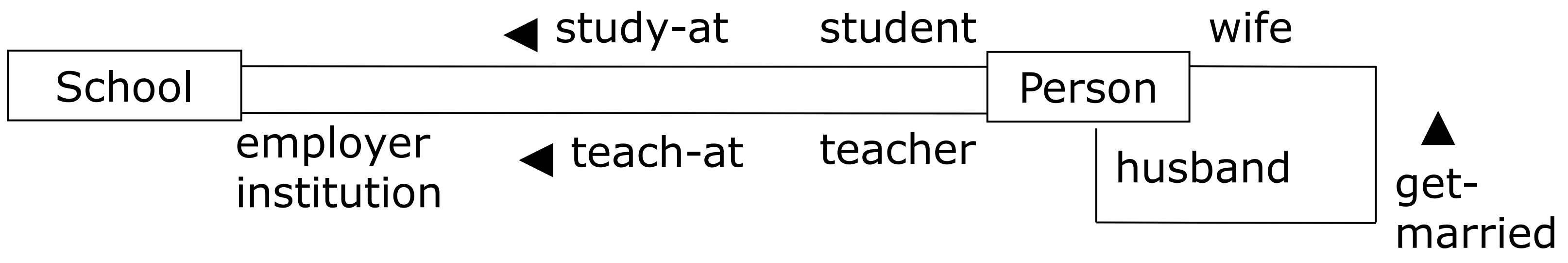
Association relation (3)

- **Navigation** on an association
  - A link between two objects (of two classes in an association) means that the objects "know each other"
    - By the link, from one object we can find the other
  - Navigation may be bi-directional (i.e., from both ends) on an association
  - Navigation may be restricted in one direction
    - Then we add an arrow to the navigated end and a slash to the unnavigated end



Association relation (4)

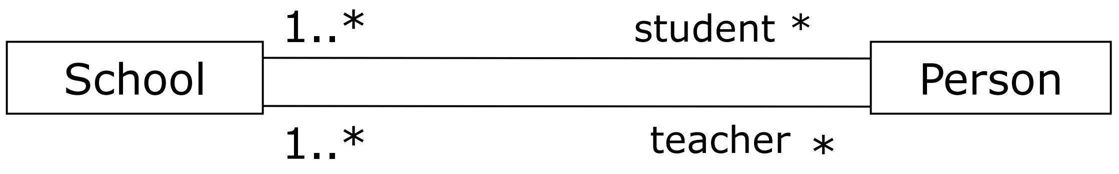
- **Role**
  - In an association between two classes, each class plays a different role
  - The role name (with the first letter in lowercase) can be appended to each end of the link (thus the role is also called the name of an association end)
  - In terms of meaning, a role represents a subset of objects of the corresponding class



Association relation (5)

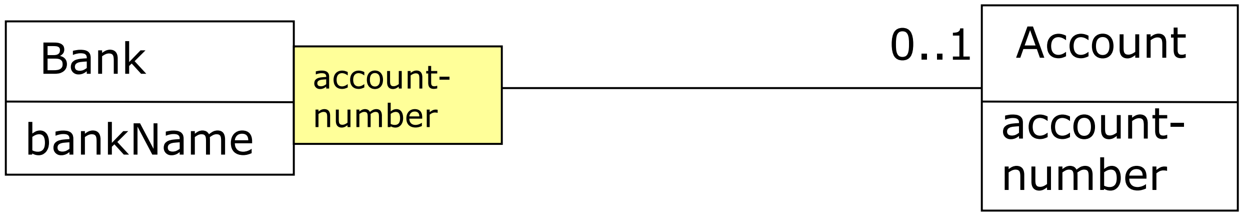
- **Multiplicity**
  - Each end of the association may also contain a multiplicity to indicate the (minimum and maximum) number of instances of that end participating in the association with <u>one</u> instance at the other end
  - Commonly used values of multiplicity:

1	1 and only one
0..1	0 or 1
m..n	From m to n (m and n are natural numbers)
0..* or *	From 0 to many
1..*	From 1 to many



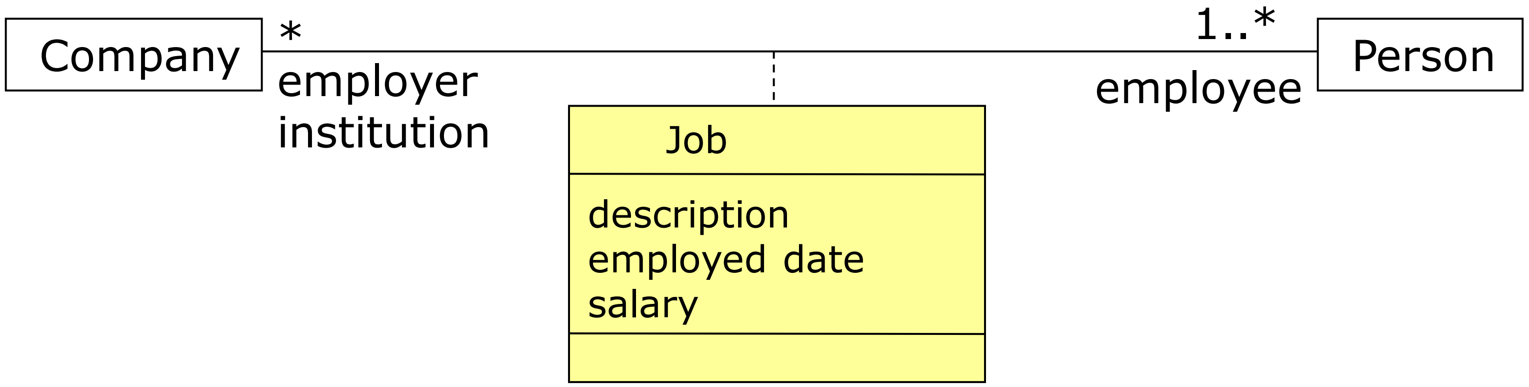
Association relation (6)

- **Qualifier**
  - Search problem: Given an object at one end of the association, find an object or set of objects connected to it at the other end
  - To reduce the number of objects found, we can limit the search area to (values of) certain attributes
  - These attributes are called **qualifiers**
    - Such attribute is shown in a small box attached to the end (the class) of the association, where the navigation originates
  - Thus, the qualifier is applied to 1-to-many or many-to-many associations, to reduce from many to 1 (or 0..1), or to reduce from many to many (but the number is less)



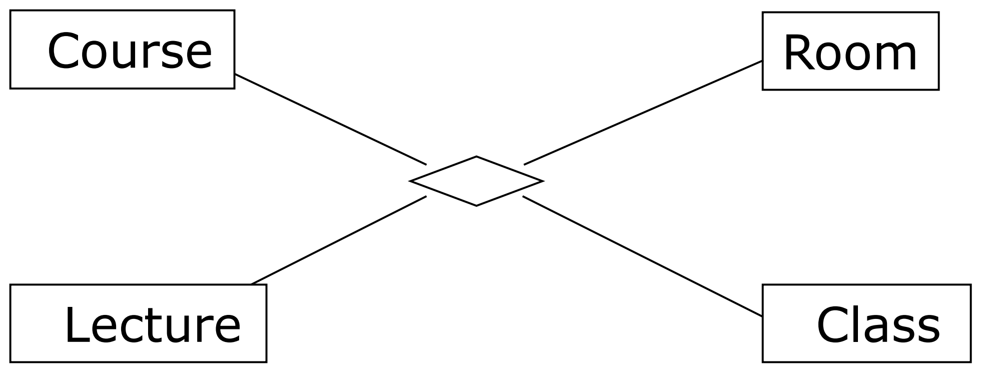
Association relation (7)

- **Association class**
  - An association itself may also need to have specific attributes
    - UML supports this by **association classes**
  - Association class is a class as usual (with attributes, operations, and associations), but the name-box can be optionally named or left blank, and it is attached to the association by a dashed line



Association relation (8)

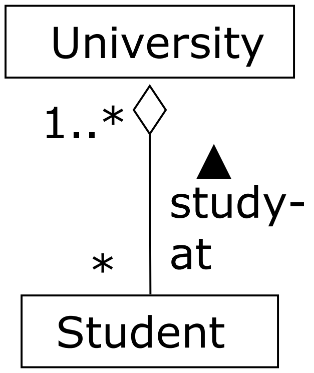
- **N-ary association**
  - There can be an association between more than two classes, in the sense of a plural (i.e., a tuple-n) relation
  - An n-ary association is represented by a small rhombus (diamond) symbol, connected by solid lines to the participating classes and may also have an association class attached



Association relation (9)

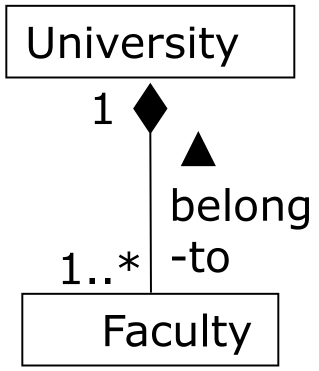
- **Aggregation**
  - In many association relations, two parties (i.e., classes) are considered equal, neither side is emphasized more than the other
  - However, sometimes we want to model a "whole-part" relation between a class of "*whole*" objects and a class of "*part*" objects. This is **aggregation** association, represented by attaching an empty rhombus (diamond) symbol to the end at the "whole" side of the association

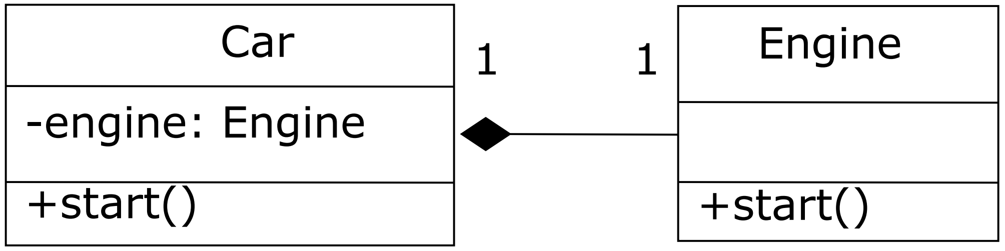
    Example: Class *Student* (i.e., the "part" class) has an aggregation association with class *University* (i.e., the "whole" class); and an object of class *Student* has the "*study-at*" relation with multiple objects of class *University*



Association relation (10)

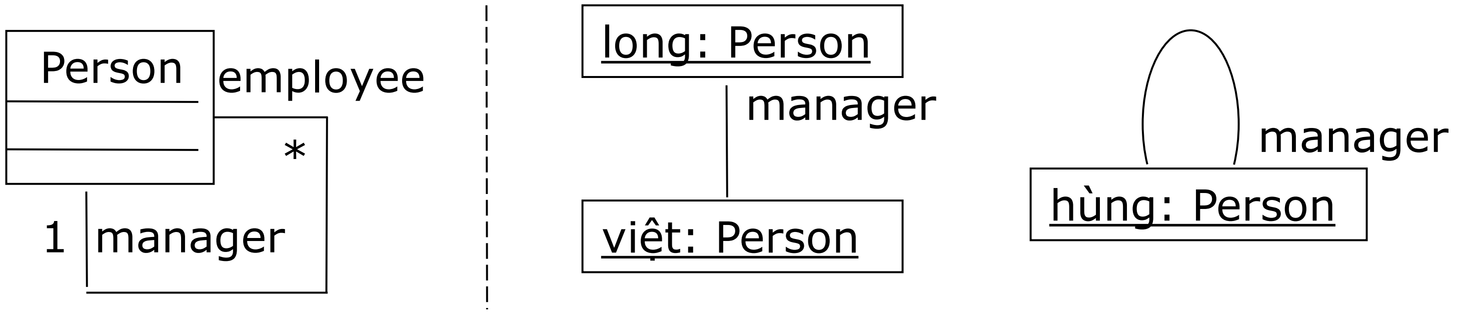
- **Composition**
  - A **composition** is a type of an aggregation with stronger ownership relation, in that a "part" class belongs only to a single "whole" class and the "whole" class is responsible for creating and destroying the "part" class
  - When the "whole" class is destroyed, the "part" class is also (automatically) destroyed
  - Composition is represented by attaching a solid rhombus (diamond) to the end at the "whole" side of the association





Class diagram and Object diagram

- A **class diagram** is a diagram that shows a set of classes and the relations (association, aggregation, composition, generalization, dependency) between them
- Class diagrams are used to model static structures of the system, including all declared elements
- An **object diagram** represents the static structure of a class diagram in a specific way
  - Objects instead of classes, Links instead of associations



Determining domain classes

- Goal and execution process
- Identifying domain concepts
- Adding associations and attributes
- Generalizing classes

Goal and execution process

- Starting from the concepts of things in the application domain, we abstract them into classes called **domain classes**
- These domain classes are often only used to reflect and simulate real-world things, so their role is often just to store and provide information about those things
- Execution process:
  - Identifying domain concepts
  - Determining associations and attributes
  - Generalizing concepts

Identifying domain concepts (1)

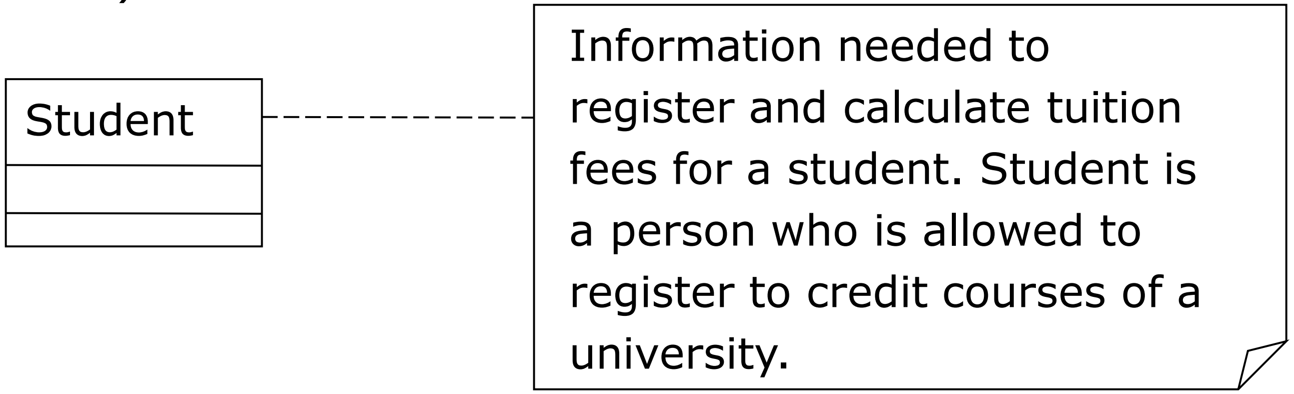
- Search sources
  - Domain concepts are concepts about things (concrete or abstract) used by users and business experts when they discuss about that domain
  - To search for domain concepts, we rely on:
    - Knowledge of the business domain
    - Interviews with users and business experts
    - The document that describes an overview of the system and needs
    - The documents that describe the use cases (created in the previous step of the software development process)

Identifying domain concepts (2)

- How to identify concepts?
  - Read the system description document (i.e., describing requirements):
    - **Nouns** can be **objects** or **attributes**,
    - **Verbs** can be **operations**
  - Based on that, identify the following objects:
    - **Entity** (e.g., motorbike, airplane, sensor, etc.)
    - **Role** (e.g., teaching, monitoring, etc.)
    - **Event** (e.g., landing, interrupting, motorbike registration, etc.)
    - **Interaction** (e.g., lending, discussion, etc.)
    - **Organization** (e.g., company, faculty, class, etc.)

Identifying domain concepts (3)

- Name and assign responsibility
  - Next is naming and assigning the **responsibility** to each of the newly defined classes
  - **Responsibility** describes the role and use purpose of the class, not the structure of that class
    - Although latter responsibility will allow us to define the structure (i.e., properties and associations) and the behavior (i.e., operations) of the class


Student
Information needed to register and calculate tuition fees for a student. Student is a person who is allowed to register to credit courses of a university.

Identifying domain concepts (4)

- Name and assign responsibility (…*continued*)
  - Naming and assigning responsibility to a class also **help to check if the class selection is reasonable**:
    - If you can choose a name and assign clear and concrete responsibility, then the class selection is good;
    - If you can choose a name, but the responsibility is the same as (or very similar to) the responsibility of another class, then you should combine those two classes into one;
    - If you can choose a name, but the responsibility is too long (too complex), then you should split that class into several ones;
    - Difficult to choose a reasonable name or describe the responsibility, then you should analyze more deeply to choose the appropriate representations

Determining associations and attributes (1)

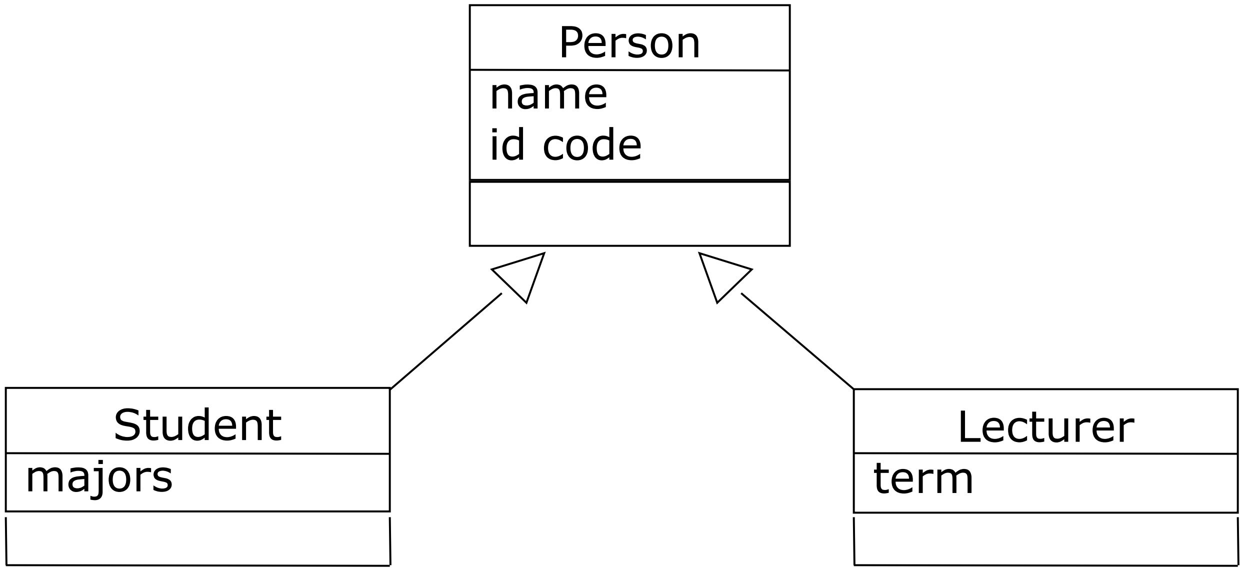
- First, many attributes and associations of domain classes can be directly determined:
  - From the document describing the system and needs,
  - From the statements of domain experts and users, and
  - From the responsibilities of the classes defined in the previous step
- Later, we will add more associations and properties, as well as add operations to classes, when we do the analysis of interaction and behavior of the system

Determining associations and attributes (2)

- Example: From the description of the responsibility of the class "Student", we have:
  - The sentence "Information needed to register and calculate tuition fees for a student" help us infer some attributes such as name, student code, address, etc. of the class “Student”
  - The sentence "Student is a person who is allowed to register to credit courses of a university" help us infer that there is an association between the class “Student” and the class “CreditCourse” with the association name maybe “register"

Generalizing classes

- In order to optimize the class representation model, we seek to extract the common parts between classes to form a more general class
- For example, the classes “Student” and “Lecturer” have common attributes (e.g., name, id code, etc.) => We may need to define a more generalized class (e.g., class “Person”)



Determining classes involving in the use cases

- Goal of determining classes involving in the use cases
- Procedure (of steps) to help determine classes participating in the use cases
- Create a class diagram for each use case

Goal of determining classes involving in the use cases

- In the previous step, we just **studied the domain, but not the current software application** => **The classes determined are just domain classes**
  - *These domain classes are the same for every software application of the domain*!
- The specifics of a software application are in its use cases. So, **to study the current software application, we have to analyze the structure and behavior of these use cases in depth.**
- A use case is viewed as a collaboration of several objects, including domain classes and application-specific classes
- The purpose of this step is to analyze the static structure of the use cases. We will have to do the following tasks:
  1. Determine classes involving in the use cases
  2. Add relations between classes to create a class diagram for each use case

Determining classes involving in the use cases (1)

- The use cases are studied (analyzed) to determine the classes that participate in each of these use cases
- The classes that participate in the use case are called the **analysis classes**, including 3 types:
  - *Boundary* class
  - *Control* class
  - *Entity* class

Determining classes involving in the use cases (2)

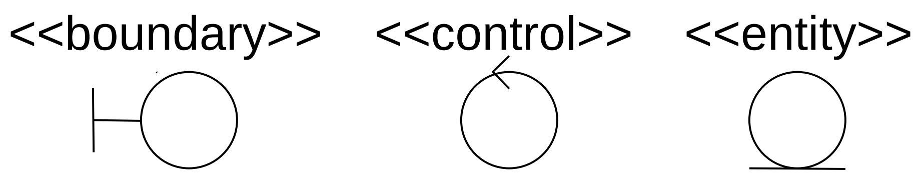
- **Boundary classes**, (a.k.a. dialog classes):
  - These are classes that communicate (exchange) information between the actors and the system:
    - Typically, they represent the screens for communicating with the users, allowing information to be collected or results to be displayed
    - They may also represent the (hardware/software) interfaces between the system and the devices that it controls or collects information
  - For each pair of (actor, use case), there must be at least one boundary class
    - This main boundary class may in turn need auxiliary boundary classes to delegate some of its overwhelming responsibilities
  - Usually, boundary objects have the same lifecycle as with its associated use cases

Determining classes involving in the use cases (3)

- **Control classes**:
  - These are the classes that manage and control the progress in a use case, i.e., it is the "engine" that makes the use case progress
  - Control classes <u>contain business rules</u> and <u>are intermediate between boundary classes and entity classes</u>, allowing from the screen (user interface) to manipulate the information contained in the entity classes
  - For each use case, we need to create at least one control class
  - A control class, when moved to the design phase, is not necessarily going to exist as a real class, since its task can be dispersed into other classes, but during the analysis phase, it is necessary to have it to ensure that functions or behaviors in the use cases are not left out

Determining classes involving in the use cases (4)

- **Entity classes**:
  - These are business classes, determined directly from the description of the domain, and will be confirmed if they appear in the use cases
  - Entity classes are persistent classes, i.e., classes whose data and relations are retained (usually in databases or in files) after their use cases finish
  - It is confirmed that an entity participate in a use case if the information contained in that entity class is mentioned in the use case
- Representation:



Creating a class diagram for each use case (1)

- Goal and requirements:
  - The ultimate goal of Step 4 in the RUP process (i.e., Determine classes participating in the use cases) is to draw a class diagram for each use case, to reflect the static structure of the collaboration (between classes)
  - The diagram of classes participating in the use case will be the foundation on which the interactions between classes take place, which we will explore in depth in the next step

Creating a class diagram for each use case (2)

- The diagram of classes participating in a use case must include all the determined classes and the necessary structural elements (*attributes*, *operations*, *associations*) of each class
  - The *attributes* must store enough information about the objects needed in the collaboration
  - The *operations* provide the required service capabilities of each class engaging in the collaboration
  - The *associations* create links between the classes, allowing them to know each other and communicate with each other in the collaboration

Creating a class diagram for each use case (3)

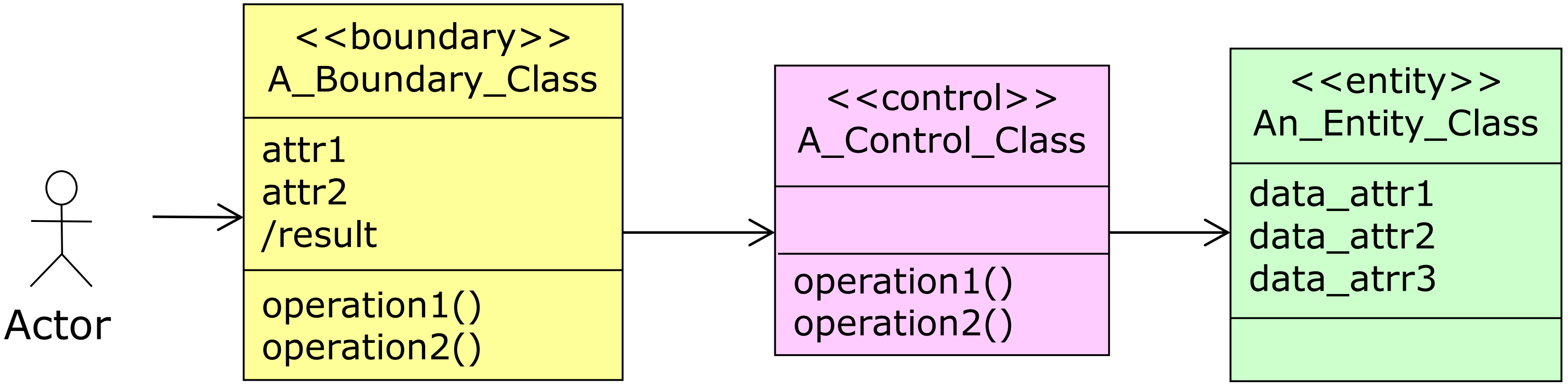
- Adding the attributes and operations:
  - The entity classes temporarily have only attributes. These attributes describe the persistent information of the system
  - The control classes temporarily have only operations. These operations represent the business processing logics, business rules, and behaviors of the system
  - The boundary classes have both attributes and operations:
    - The attributes describe fields to collect information or to output results. The results are distinguished by the notation of derived attributes
    - The operations represent actions that the user performs on the interface screen

Creating a class diagram for each use case (4)

- Adding the associations:
  - Boundary classes are only associated with control classes or with other boundary classes. Usually, an association is one-way from a boundary class to a control class, unless the control class again creates a new dialogue (e.g., a page displaying the results).
  - Entity classes are only associated with control classes or with other entity classes. An association of an entity class with a control class is always one-way (from the control class to the entity class).
  - Control classes may be associated with all types of classes (including also with other control classes)

Creating a class diagram for each use case (5)

- Adding the actors:
  - Finally, we add the actors to the class diagram
  - An actor is connected to only one (or several) boundary class(es)
    - An actor is not associated with any control or entity class!



Illustrative example exercise (1)

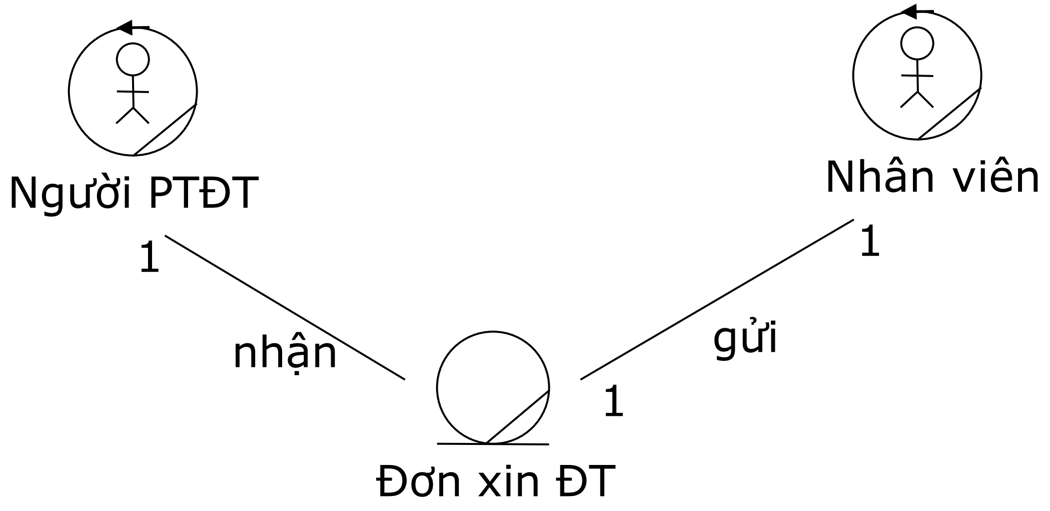
Mô hình hóa cấu trúc tĩnh được thực hiện dựa vào văn bản phát biểu bài toán. Phát biểu đó được biên tập lại có lược bớt như sau:

1) QTĐT bắt đầu khi người phụ trách đào tạo (PTĐT) nhận được một đề nghị đi đào tạo từ một nhân viên (NV).
2) Người PTĐT xem xét đề nghị này và đưa ra trả lời đồng ý hay không đồng ý.
3) Nếu đồng ý, người PTĐT tìm trong một danh sách các lớp đào tạo để chọn một lớp đào tạo phù hợp.
4) Người PTĐT thông báo nội dung của lớp đào tạo cho NV đã xin đi đào tạo, cùng với một danh sách các kỳ học sẽ mở tới đây.
5) Khi người NV đã chọn kỳ học, người PTĐT gửi một yêu cầu đăng ký cho NV đó tới cơ sở đào tạo.
6) Người PTĐT kiểm tra lại hoá đơn mà cơ sở đào tạo gửi tới, trước khi chuyển cho kế toán trả tiền.

Illustrative example exercise (2)

Bước 8: *MHH câu phát biểu thứ 1, sử dụng các biểu tượng của Jacobson*.
"*QTĐT bắt đầu khi người phụ trách đào tạo (PTĐT) nhận được một đề nghị đi đào tạo từ một nhân viên (NV)*".

- Chú ý các danh từ
- QTĐT đã được xác định từ Bước 1 (của bài 5 – Phân tích chức năng) là một quy trình nghiệp vụ, vậy không phải là lớp
- Còn lại các danh từ: "người PTĐT", "đề nghị ĐT", "nhân viên" sẽ được mô hình hóa thành các lớp

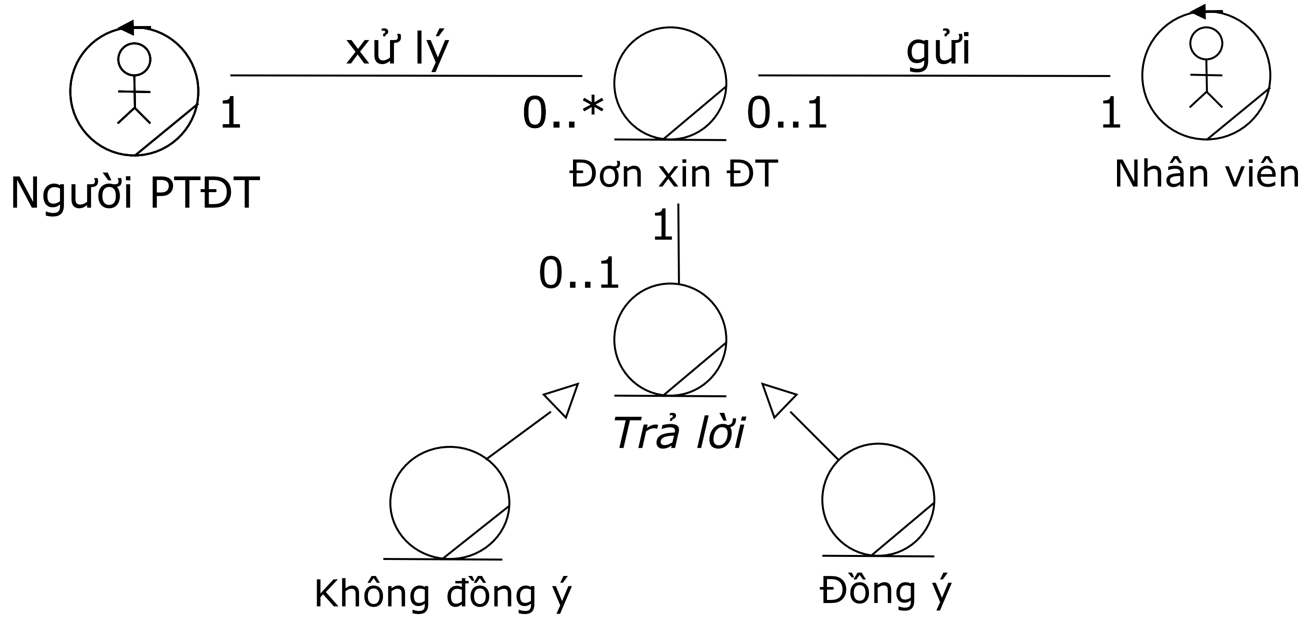


Illustrative example exercise (3)

Bước 9: *MHH câu phát biểu thứ 2.*

"*Người PTĐT xem xét đề nghị này và đưa ra trả lời đồng ý hay không đồng ý*".

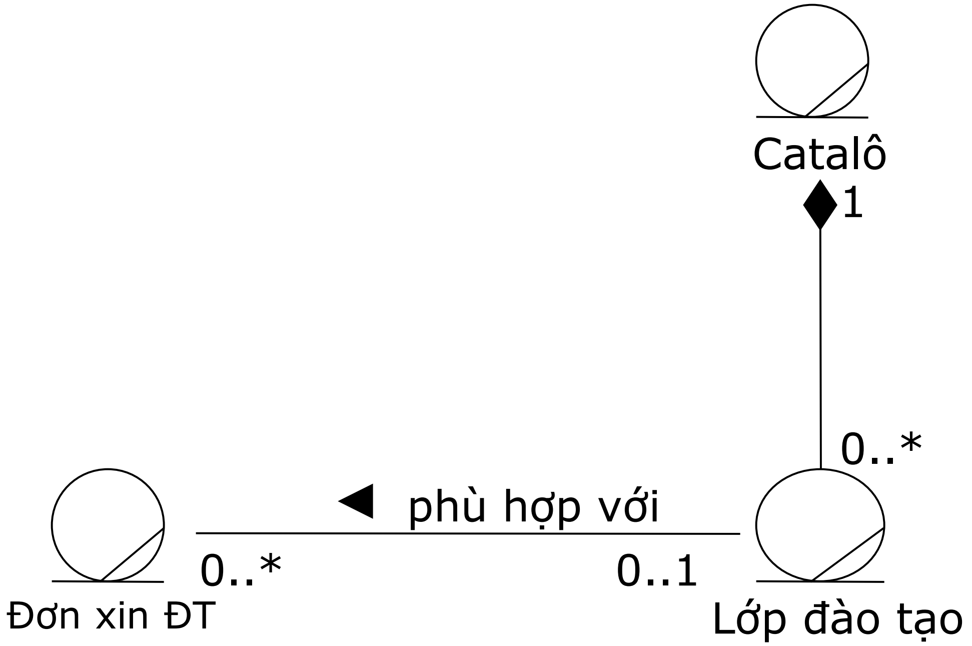
- Mở rộng mô hình trên, có chỉnh sửa vài chi tiết cho thích hợp hơn và thêm một thực thể trừu tượng (Trả lời) cùng với 2 thực thể chuyên biệt (Đồng ý, Không đồng ý)



Illustrative example exercise (4)

Bước 10: *MHH câu phát biểu thứ 3.*

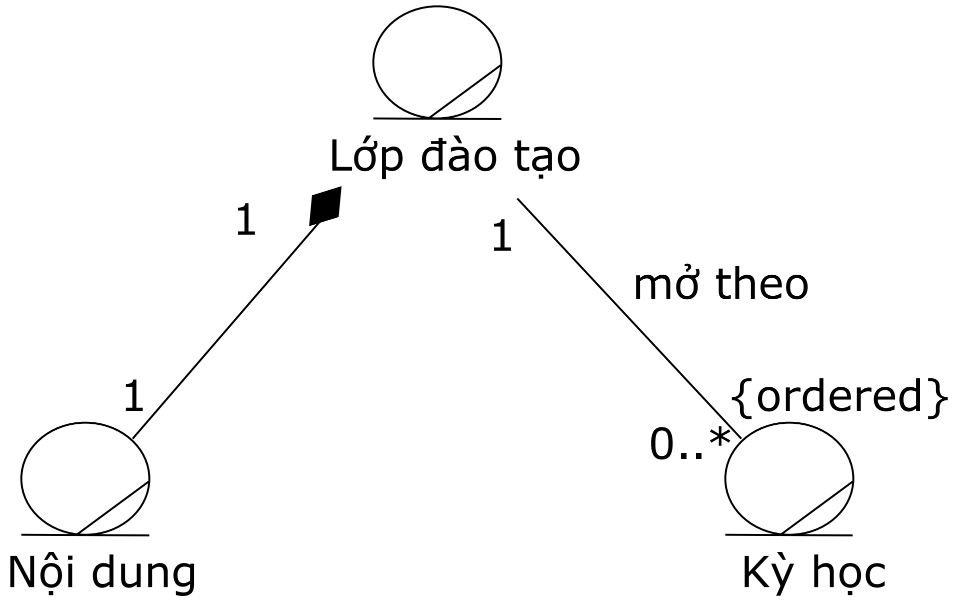
"*Nếu đồng ý, người PTĐT tìm trong một danh sách các cơ sở đào tạo (Catalô) một lớp đào tạo phù hợp*".



Illustrative example exercise (5)

Bước 11: *MHH câu phát biểu thứ 4.*

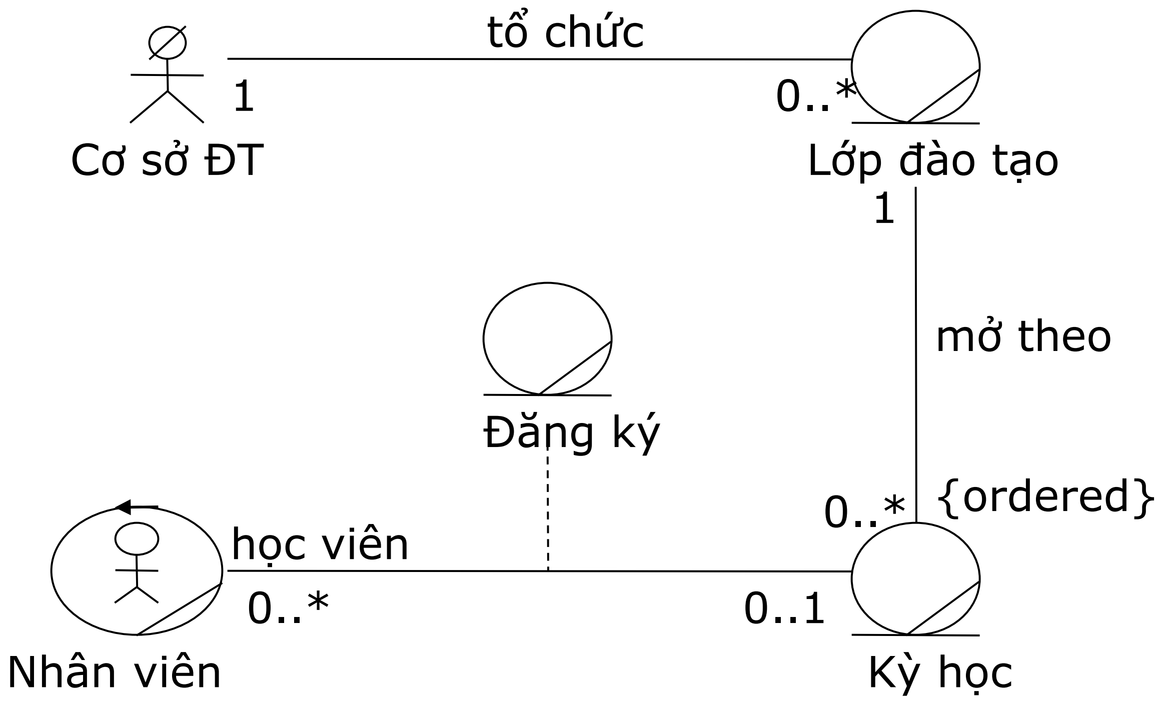
"*Người PTĐT thông báo nội dung của lớp đào tạo cho NV đã xin đi đào tạo, cùng với một danh sách các kỳ học sẽ mở tới đây*".



Illustrative example exercise (6)

Bước 12: *MHH câu phát biểu thứ 5.*

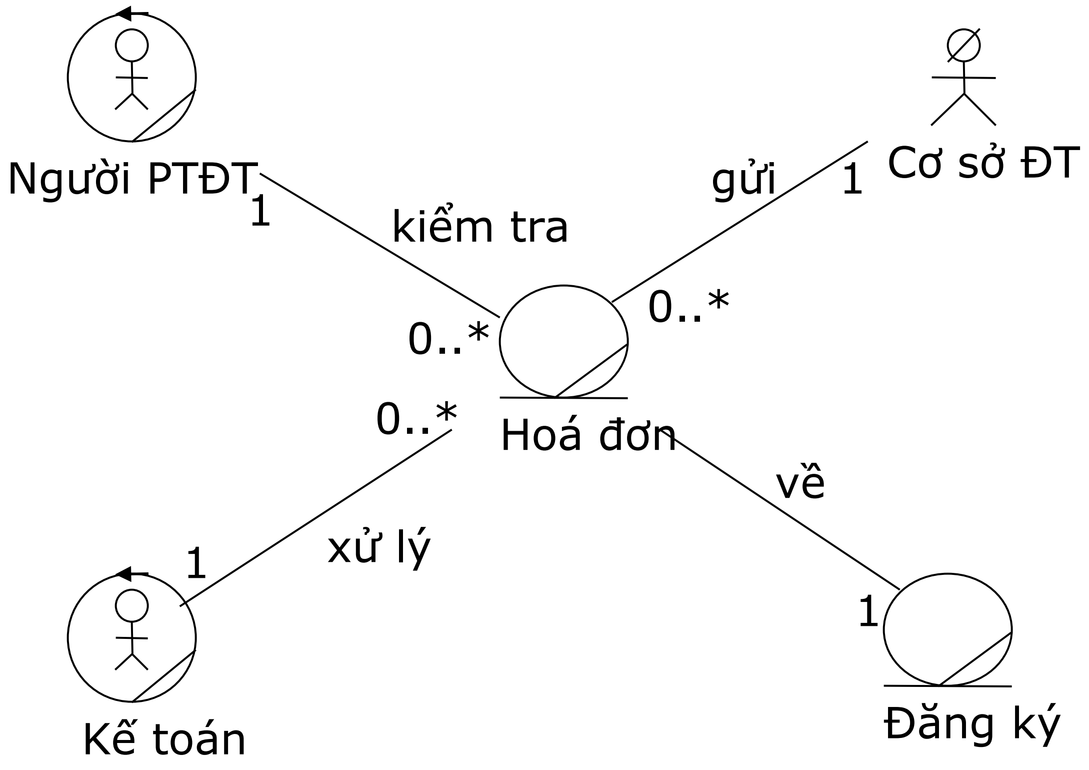
"*Khi người NV đã chọn kỳ học, người PTĐT gửi một yêu cầu đăng ký cho NV đó tới cơ sở đào tạo*".



Illustrative example exercise (7)

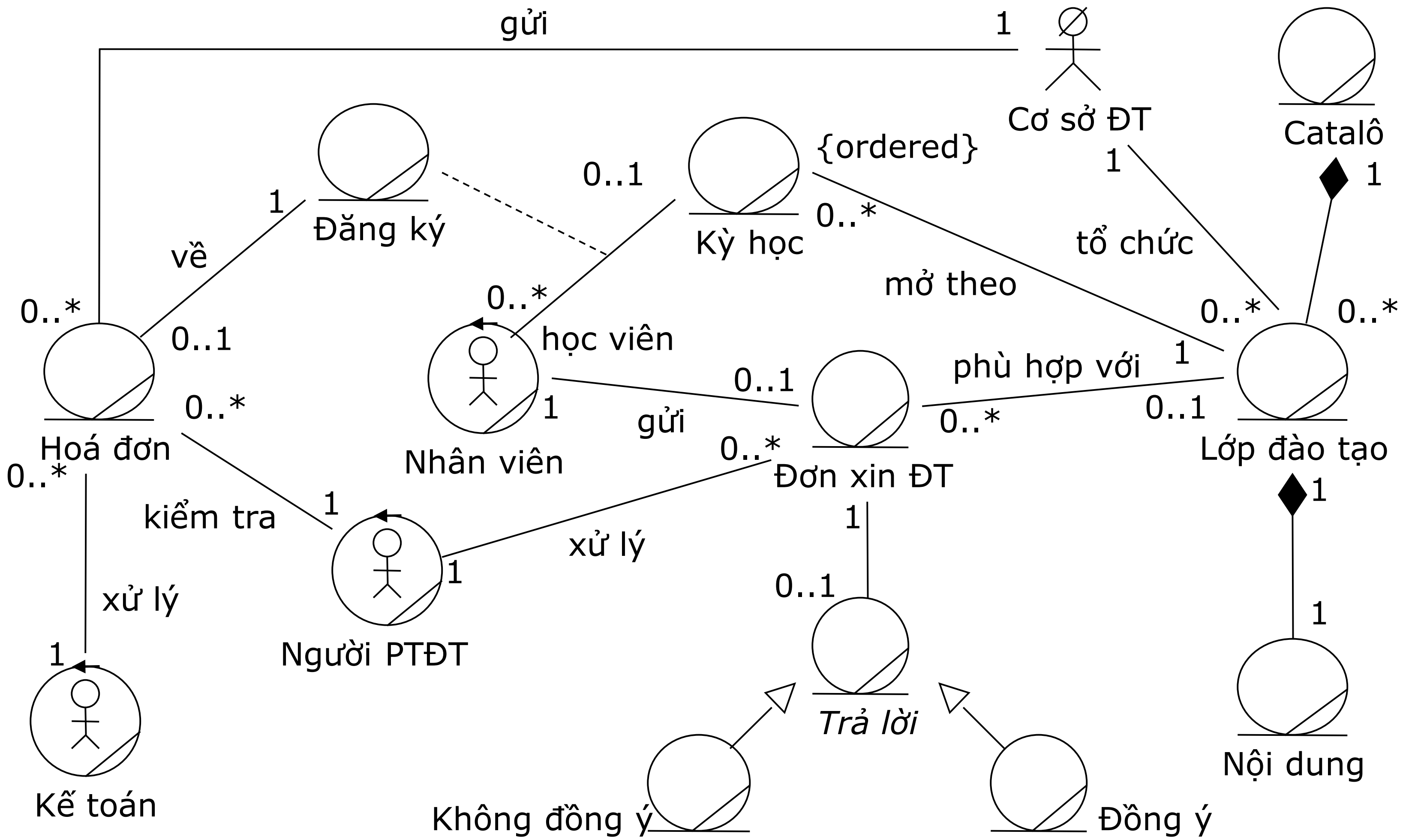
Bước 13: *MHH câu phát biểu thứ 6.*

"*Người PTĐT kiểm tra lại hoá đơn mà cơ sở đào tạo gửi tới, trước khi chuyển cho kế toán trả tiền*".



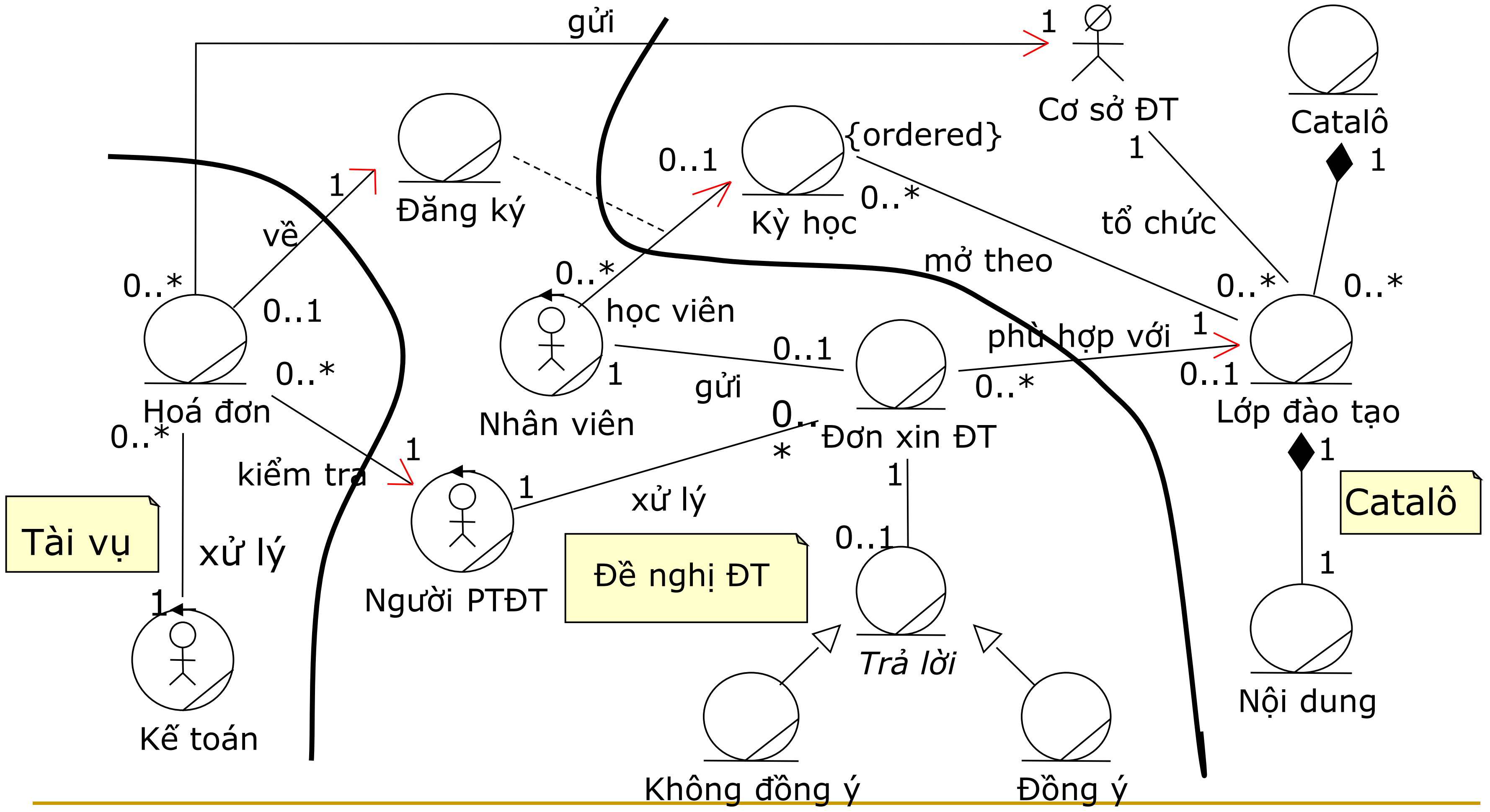
Illustrative example exercise (8)

- **Bước 14:** *Tổng hợp các kết quả trên vào một biểu đồ lớp nghiệp vụ (lĩnh vực)*.



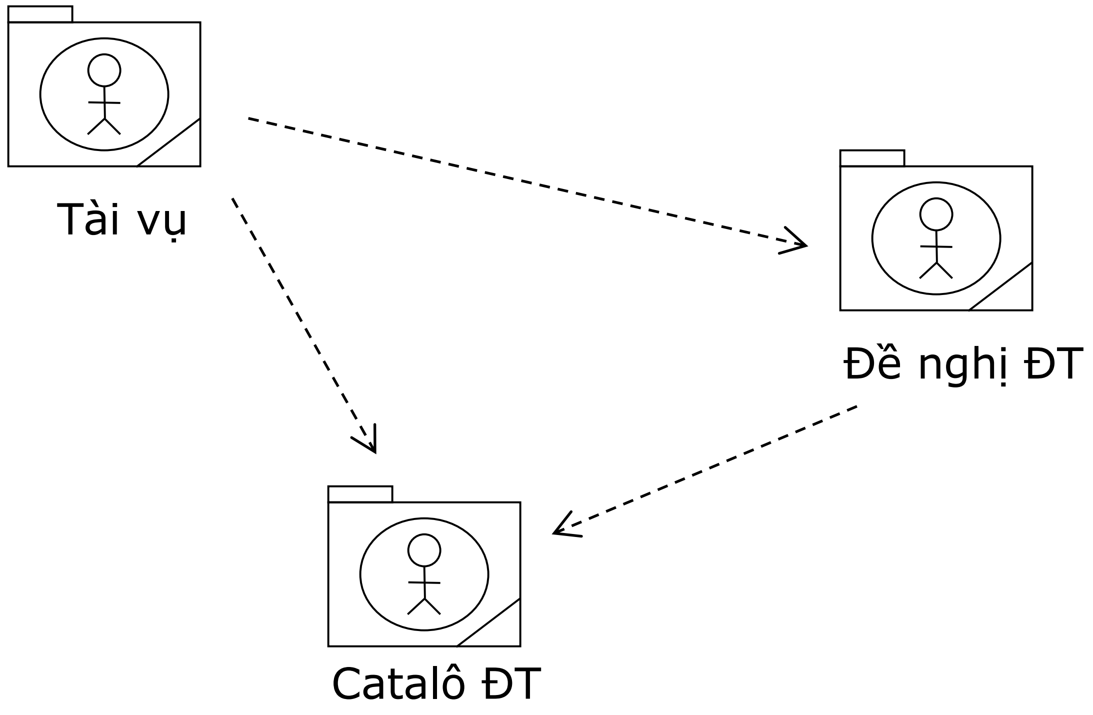
Illustrative example exercise (9)

- **Bước 15:** *Chia cắt mô hình thành các gói theo các đơn vị nghiệp vụ và sửa lại các liên kết cho nhẹ bớt sự phụ thuộc giữa các gói.*



Illustrative example exercise (10)

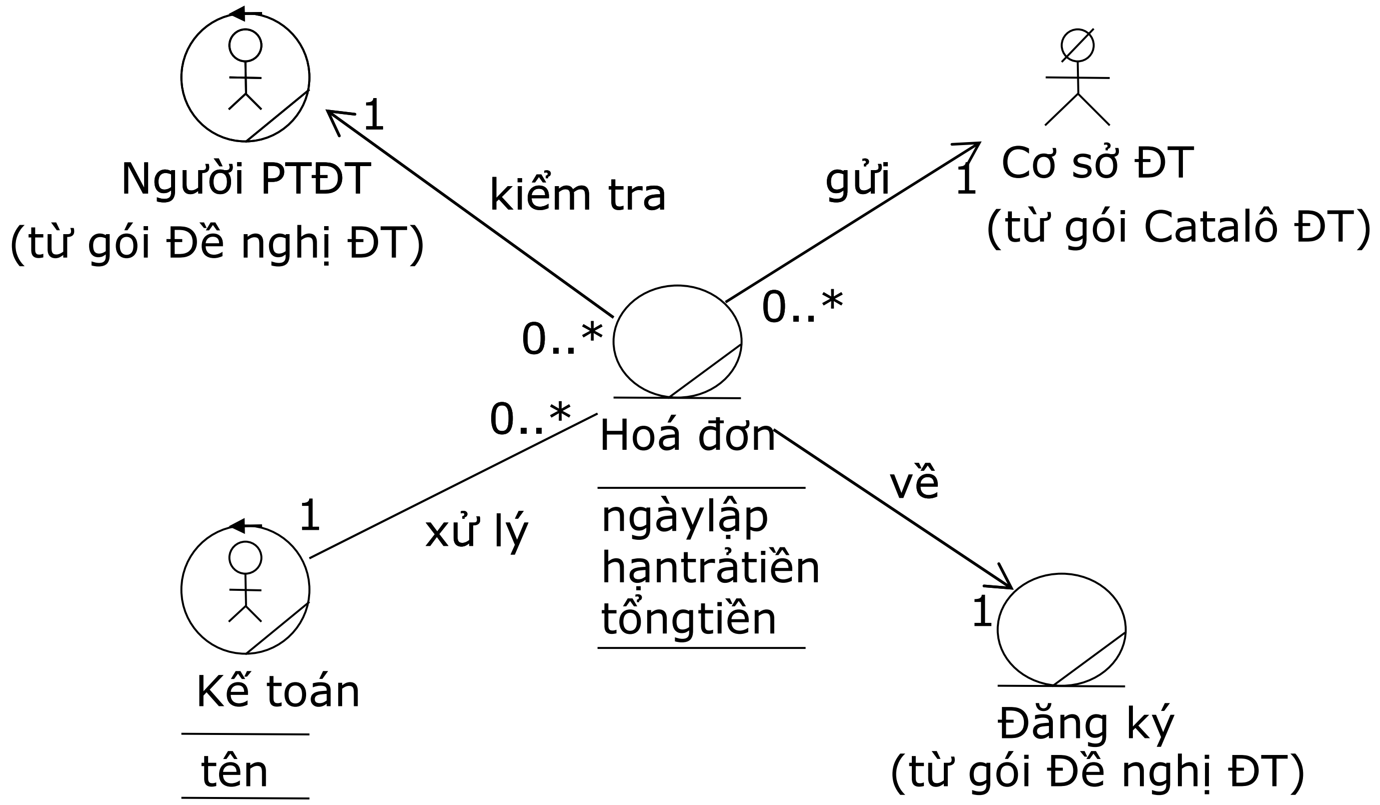
- Biểu đồ gói thu được như sau (các phụ thuộc là một chiều):



Illustrative example exercise (11)

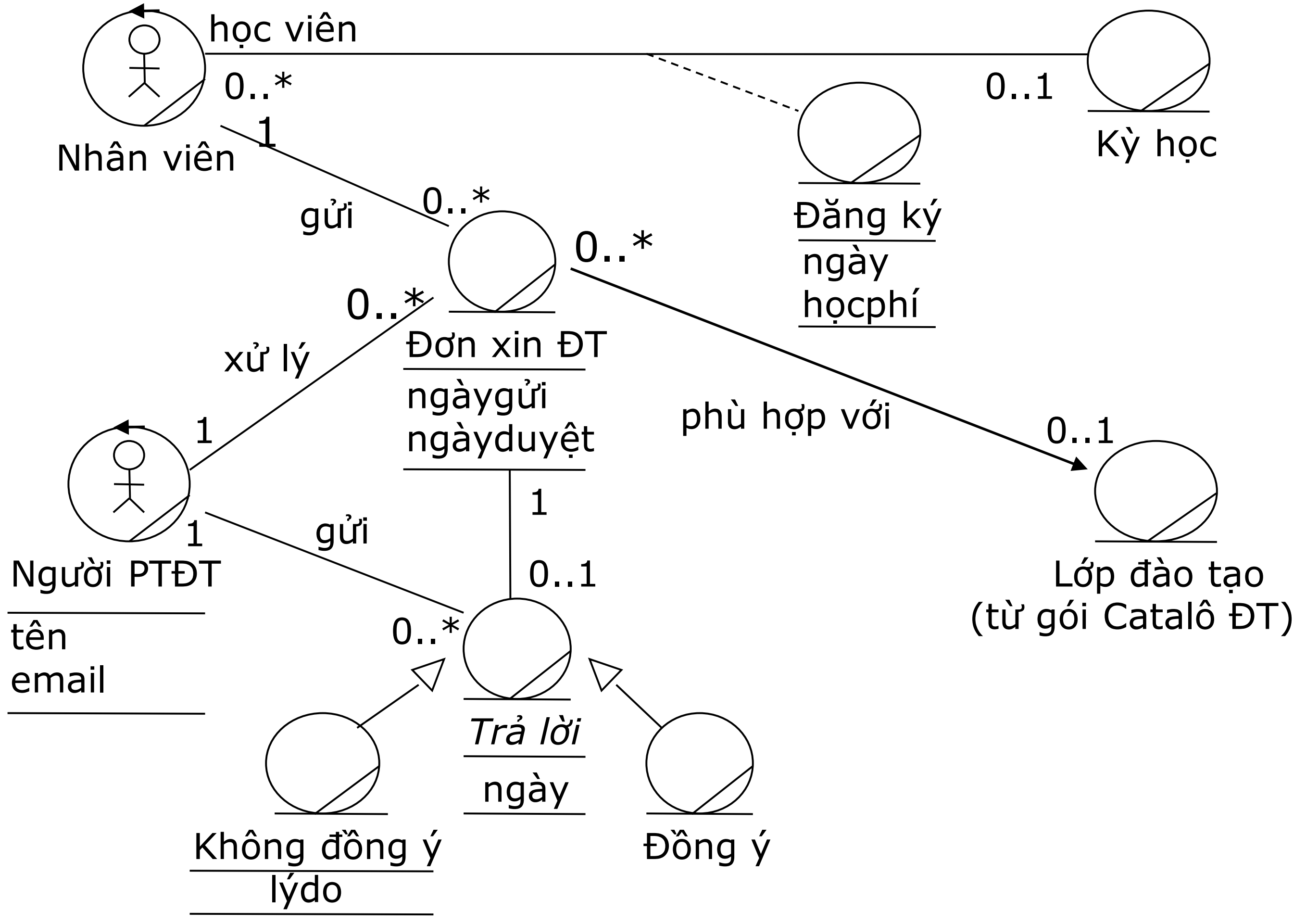
Bước 16: *Thêm các thuộc tính trường cửu (persistent) vào các lớp (vẽ riêng cho từng gói)*

- Biểu đồ lớp của gói "Tài vụ":



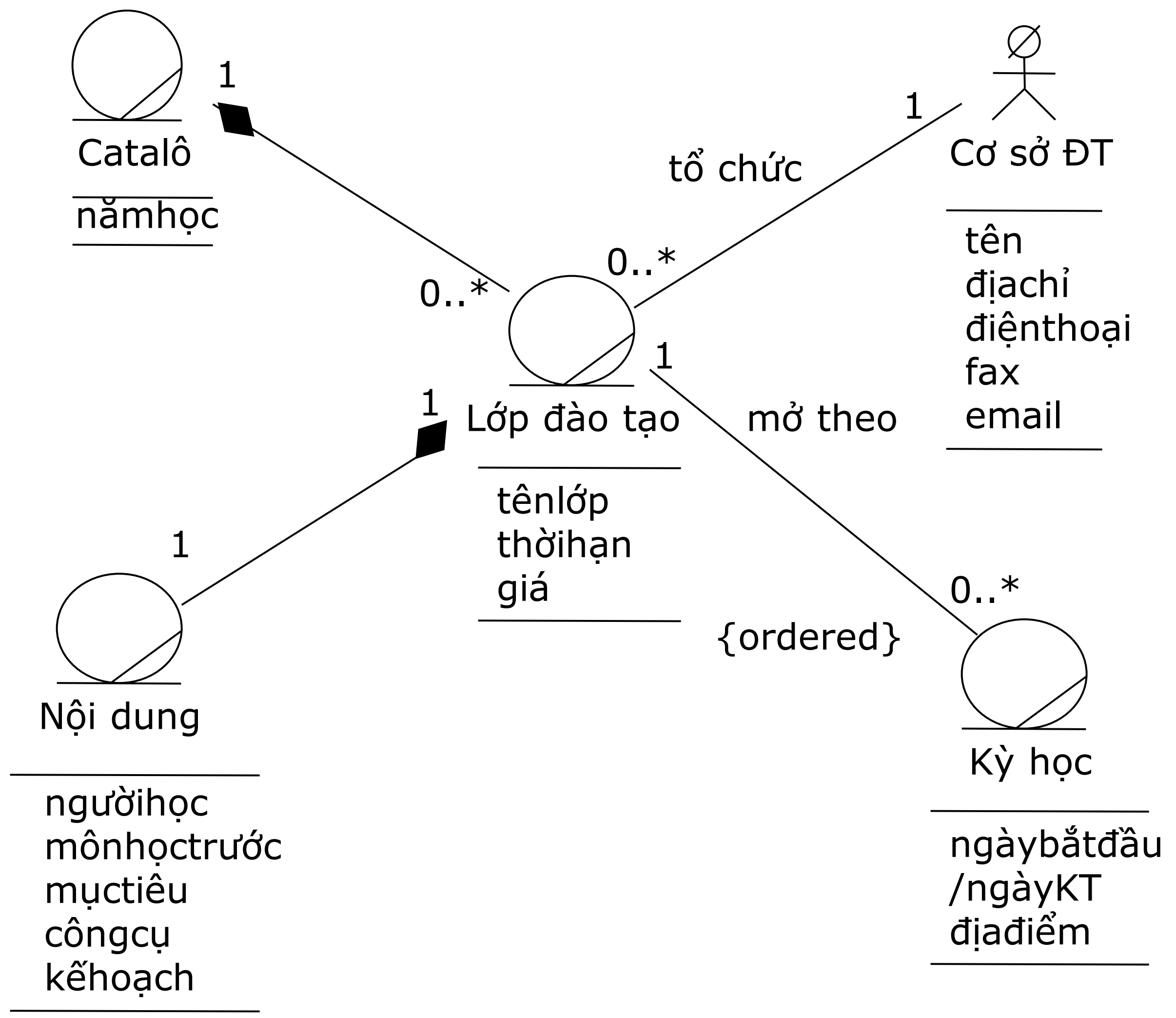
Illustrative example exercise (12)

- Biểu đồ lớp của gói "Đề nghị ĐT":



Illustrative example exercise (13)

- Biểu đồ lớp của gói "Catalô ĐT":



Illustrative example exercise (14)

- Đến đây thì ta đã hoàn thành việc phát hiện <u>các lớp lĩnh vực (lớp thực thể)</u> của ứng dụng
- Việc phát hiện các lớp biên và các lớp điều khiển cho mỗi ca sử dụng:
  - Xem đó là bài tập về nhà!
- Việc phát hiện các lớp biên và các lớp điều khiển sẽ tiếp tục được cập nhật khi chúng ta tiến hành việc Phân tích hành vi (tương tác, ứng xử)
  - Sẽ được trình bày trong các bài học tiếp theo!