# Subject:

# Object-oriented analysis and design

## Chapter 3: Object-oriented Analysis

**Lê Văn Vinh, PhD**
Department of Software Engineering
Faculty of Information Technology
HCMC University of Technology and Education

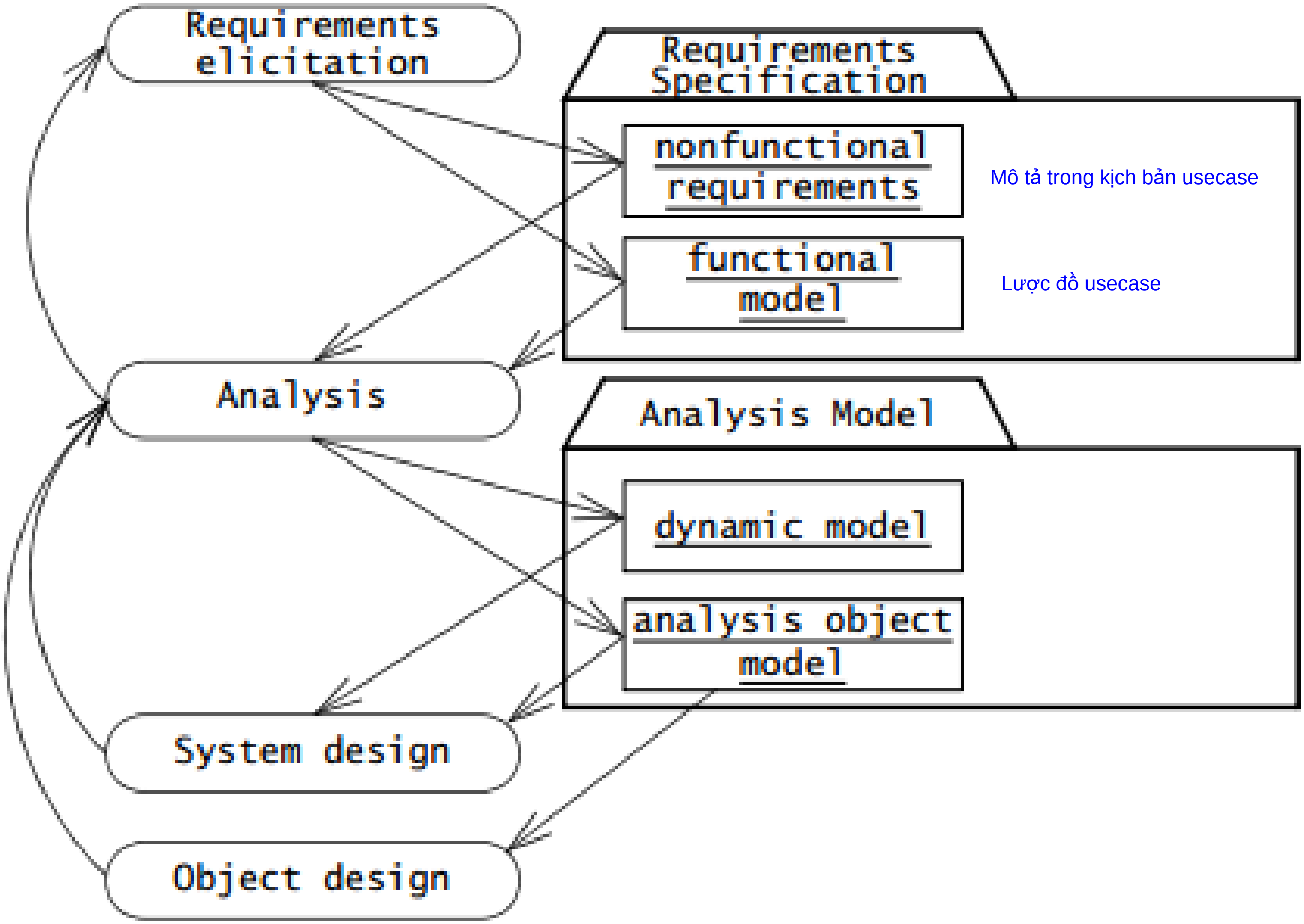
# Contents

**I. Overview of Analysis**

II. Analysis activities

III. Excercies

# I.Overview of Analysis



Mô tả trong kịch bản usecase

Lược đồ usecase

# I.Overview of Analysis

- **Analysis focuses on**
  - producing a model of the system, called the analysis model
- **Analysis model**
  - Functional model (usecases, scenarios)
  - Analysis Object model (class/object diagrams)
  - Dynamic model (state diagrams, sequence diagrams, communication diagrams)

# Contents

I. Overview of Analysis

**II. Analysis activities**

III. Excercies

# II. Analysis activities

1. **Identifying Objects**
2. Mapping Usecase to Objects (with Sequence diagrams)

   Practical: Usecase >> Sequence diagram >> Class diagram
   Report: Usecase >> Class diagram >> Sequence diagram
3. Identifying Class relationship
4. Identifying Attributes
5. Modeling State-dependent Behavior of Objects

# Analysis Object models

## Analysis object models

- Focus on idivididual concepts that are manipulated by the system.
- UML class diagram:
  - Classes
  - Attributes
  - Operations
- Common stereotypes:
  - M • Entity classes là những đối tượng thật mà hệ thống lưu trữ thông tin => phát sinh DB
  - V • Boudary classes
  - C • Control classes

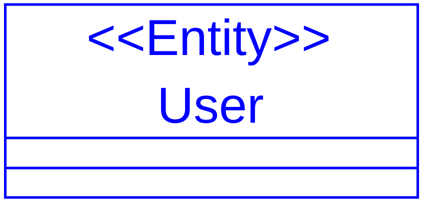
# Case study: “Login” usecase

| | |
|---|---|
| Name | Log In |
| Brief Description | A user of the System logs in to the System. |
| Actor(s) | Logged In User |
| Flow of Events | |
| Basic Flow | |
| This use case starts when a system user is not logged in to the system and goes to the login page.<br>1. The System prompts the user for a username and password or register new account<br>2. The user enters his/her username and password.<br>3. The system validates the entered username and password, making sure that the entered username is a valid username in the System, and that the required password is entered for the entered username.<br>4. The user is signed in and returned to the home page as a Logged In User.<br>5. The use case ends. | |
| Alternate Flows | |
| Title | Description |
| User Fails Authentication | If the User entered an invalid username and/or password, the following occurs:<br>1. The system describes the reasons why the User failed authentication.<br>2. The system presents the User with suggestions for changes necessary to allow the User to pass authentication.<br>3. The system prompts the User to re-enter the valid information.<br>4. The Basic Flow continues where the User enters new information (see step 2 of the Basic Flow). |
| Pre-Conditions | |
| Title | Description |
| (none) | |
| Post-Conditions | |
| Title | Description |
| Success | The User is authenticated and the system displays a home page based on the user type. |
| Failure | User is unable to log in for one or more reasons. |
| Extension Points | |
| None | |

# Entity Class

- An entity class models information and associated behavior that is generally long-lived (persistent)
  - It can reflect a real-life phenomenon
  - It may also be needed for the internal tasks of the system
  - The values of its attributes are often provided by an actor
  - The behavior is surroundings-independent
- Entity classes in the “Login” use case
  - Users

VD: Entity class: Môn học, khóa học, sinh viên

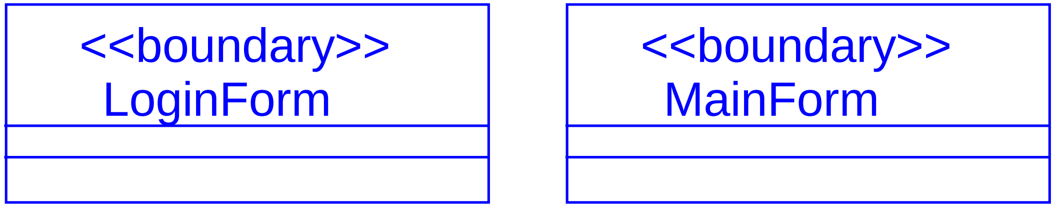

<<Entity>>
User

## How to identify Entiy classes:

- Terms that developers or users need to clarify in order to understand the use
- Real-world entities that the system needs to track (e.g., FieldOfficer, Dispatcher, Resource)
- Real-world activities that the system needs to track (e.g., EmergencyOperationsPlan
- Find common noun

# Boundary Class

- A boundary class models communication between the system’s surroundings and its inner workings
- Typical boundary classes
  - Windows (user interface)
  - Communication protocol (system interface)
  - Printer interface
  - Sensors
- In the “Login” scenario:



<<boundary>>
LoginForm

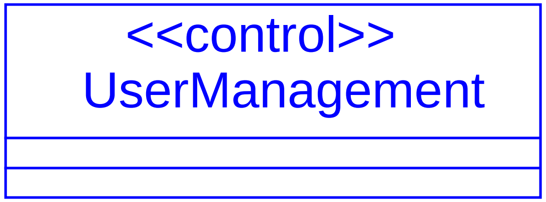
<<boundary>>
MainForm

# Boundary Class

- How to identify Boundary classes:
  - Identify user interface controls that the user needs to initiate the use case
  - Identify forms the users needs to enter data into the system
  - Identify notices and messages the system uses to respond to the user
  - *Always* use the end user’s terms for describing interfaces; do not use terms from the solution or implementation domains.

# Control Class

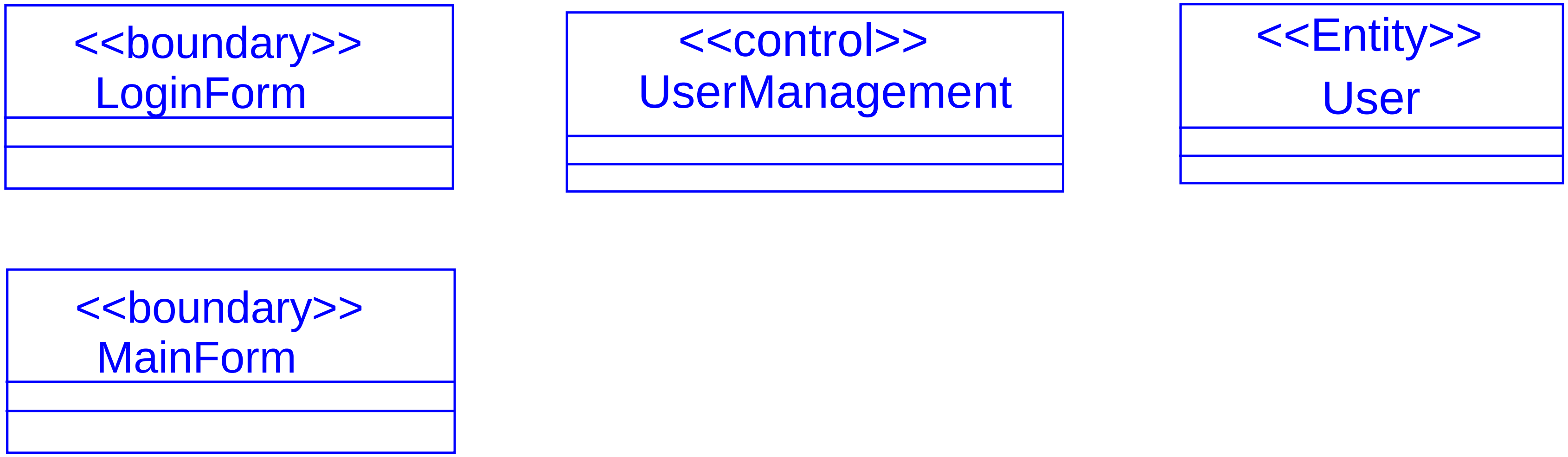
- Control objects are responsible for coordinating boundary and entity objects
- It is responsible for collecting information from the boundary objects and dispatching it to entity objects
- A control class models control behavior specific to one or more use cases
- A control class
  - Creates, initializes and deletes controlled objects
  - Controls the sequencing or coordination of execution of controlled objects
  - Controls concurrency issues for controlled classes
- In the “Login” scenario



<<control>>
UserManagement

# 1. Identify objects

- **Examples of “Login” usecase**



<<boundary>>
LoginForm

<<control>>
UserManagement

<<Entity>>
User

<<boundary>>
MainForm

# 1. Identify objects

- **Exercises**
  - Identify objects of the three following systems
    - Hotel management (Windows application)
    - Restaurant management (Windows application)
    - Online selling webstite

# II. Analysis activities

1. Identifying Objects
2. **Mapping Usecase to Objects (with Sequence diagrams)**
3. Identifying Class relationship
4. Identifying Attributes
5. Modeling State-dependent Behavior of Objects

# II.2.Mapping Usecase to Objects

- **Sequence diagrams**

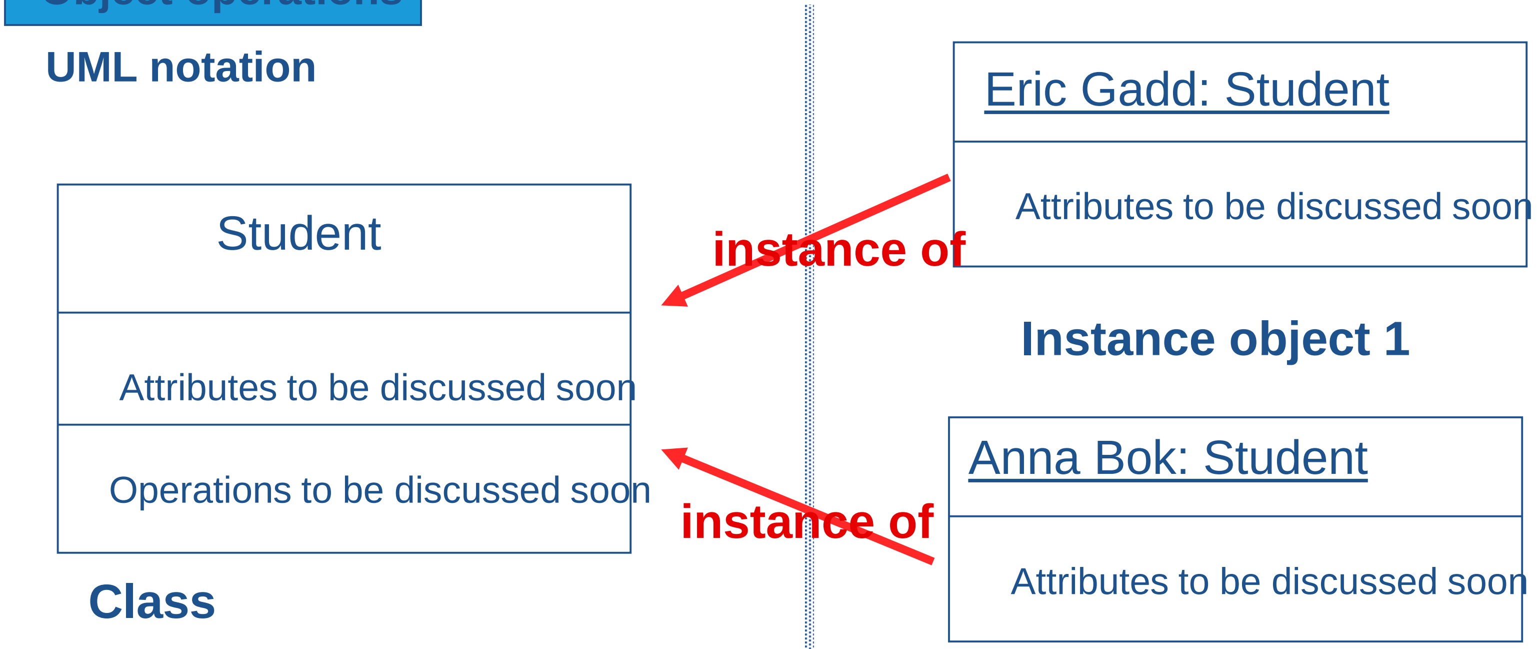
# Class vs Object

Class name

Object attributes

Object operations

**UML notation**



Student
Attributes to be discussed soon
Operations to be discussed soon
instance of
instance of
Eric Gadd: Student
Attributes to be discussed soon
Anna Bok: Student
Attributes to be discussed soon

**Class**

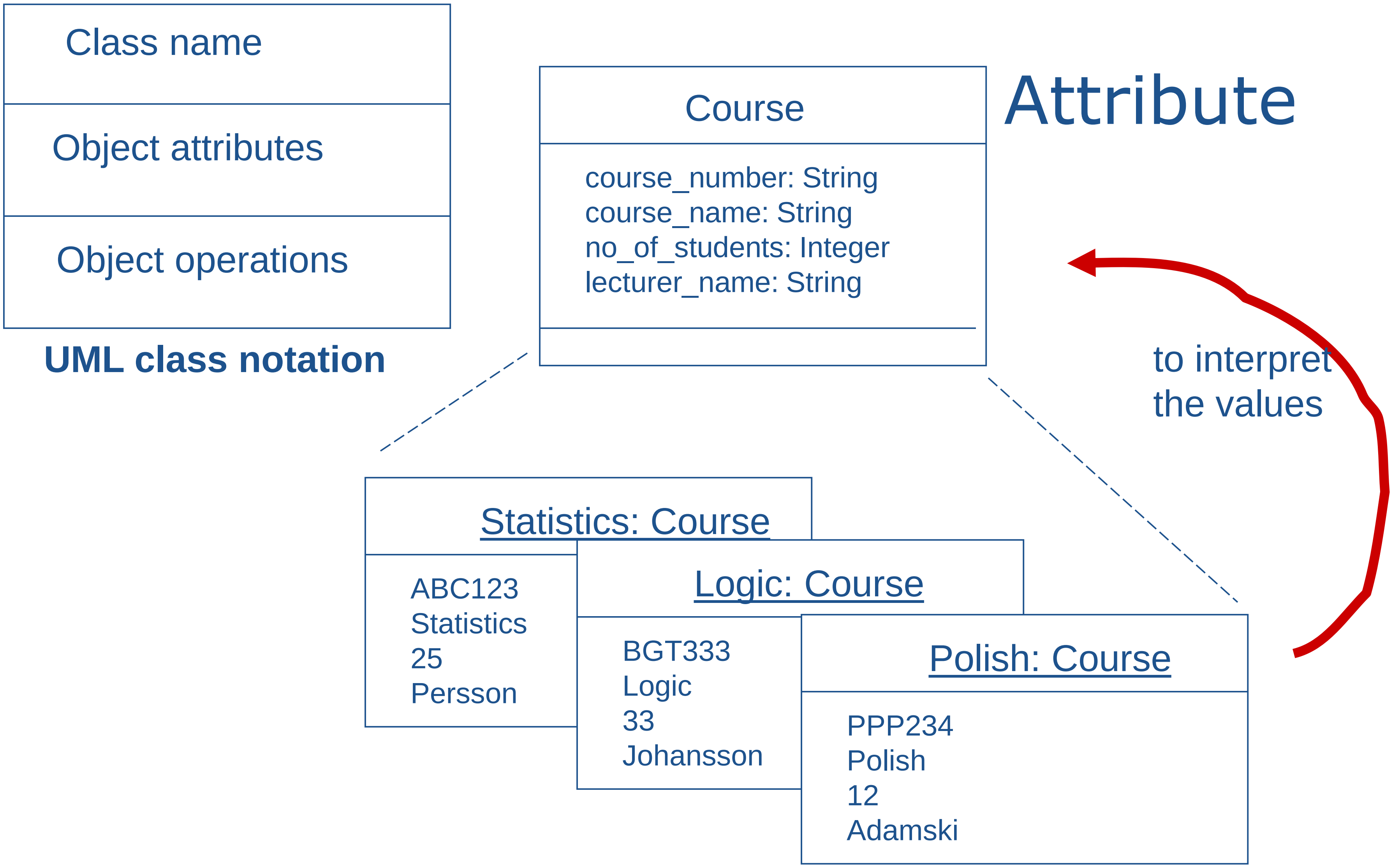
**Instance object 1**

**Instance object 2**

A class is the descriptor for a set of objects with the same attributes and operations.

Objects store data values for a certain instance of a student.

# Class vs Object



Class name

Object attributes

Object operations

**UML class notation**

Course

course_number: String
course_name: String
no_of_students: Integer
lecturer_name: String

Attribute

to interpret
the values

Statistics: Course

ABC123
Statistics
25
Persson

Logic: Course

BGT333
Logic
33
Johansson

Polish: Course

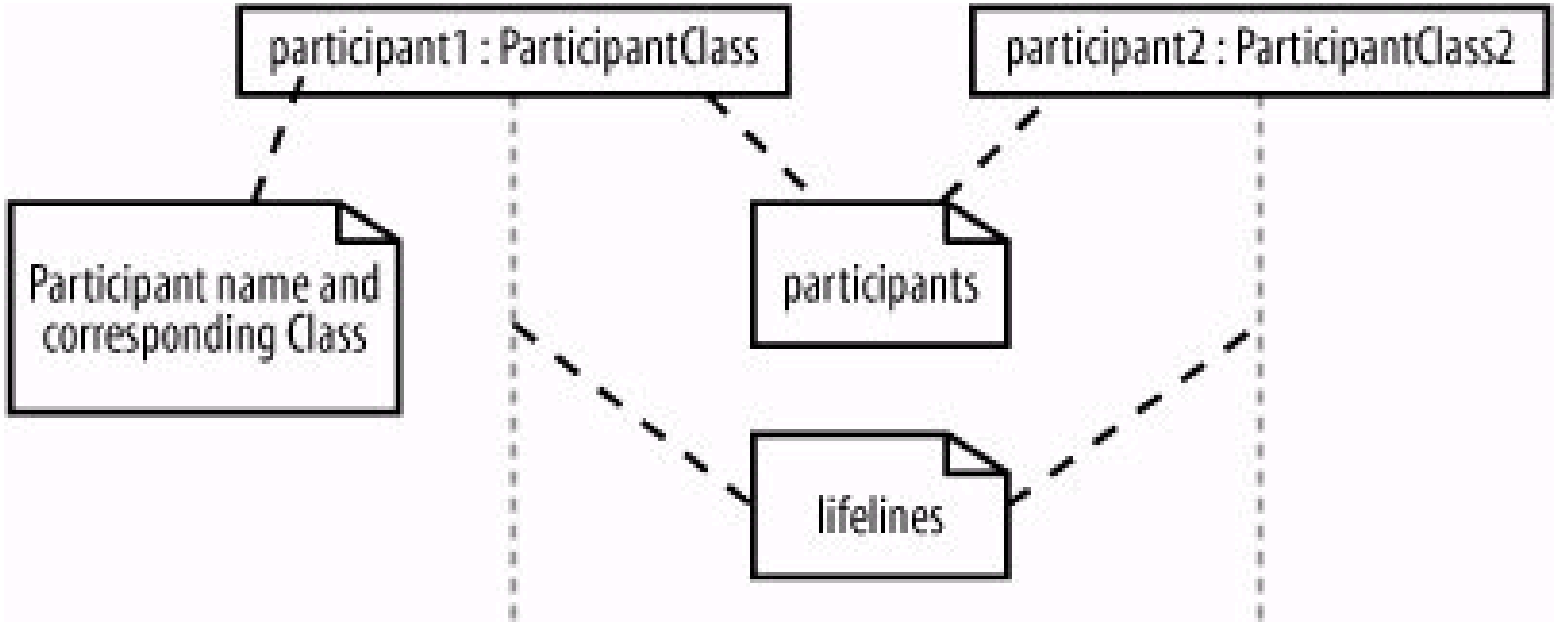
PPP234
Polish
12
Adamski

# What is a Sequence Diagram?

- A sequence diagram shows object interactions arranged in time sequence
- The diagram shows
  - The objects participating in the interaction
  - The sequence of messages exchanged
- A sequence diagram contains:
  - Objects with their “lifelines”
  - Messages exchanged between objects in ordered sequence
  - Focus of control (optional)

# Participants

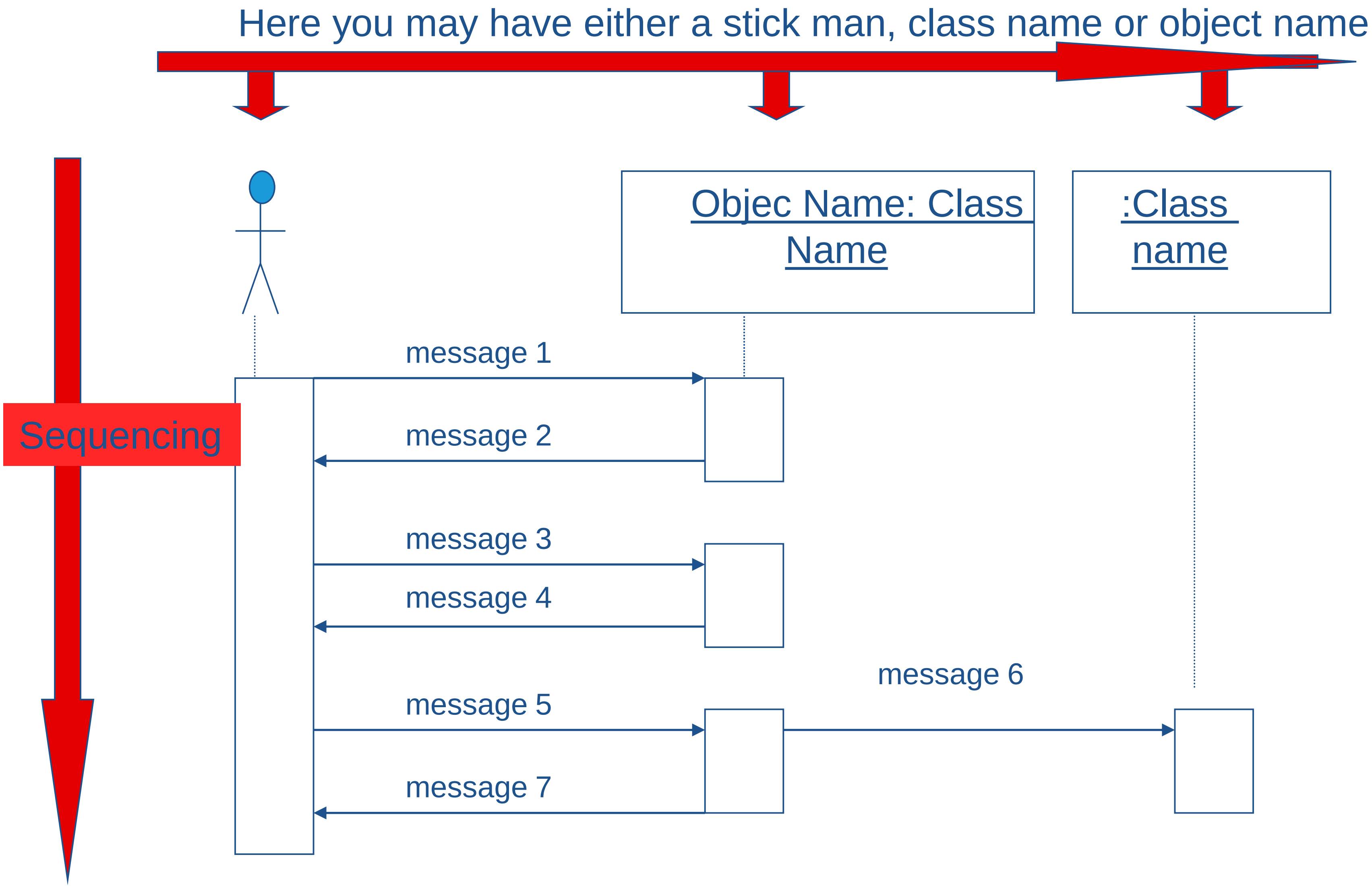
- **Participants in sequence diagrams are objects**


participant1 : ParticipantClass
participant2 : ParticipantClass2
Participant name and corresponding Class
participants
lifelines

# Naming Objects in Sequence Diagrams

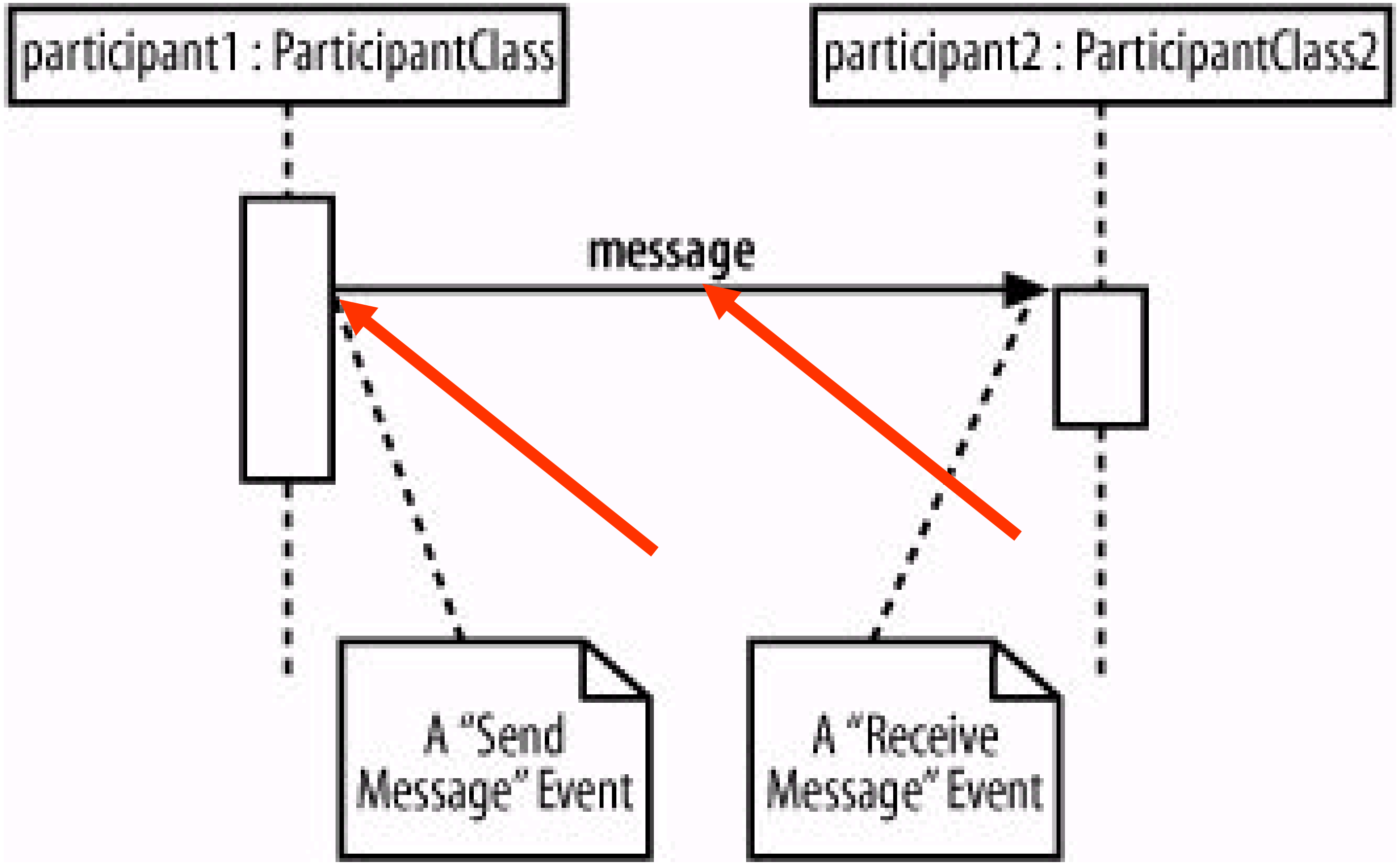
| | |
|---|---|
| admin | Object name, not assigned to a specific class |
| :BlogEntry | Object without a name, class name only |
| admin:Administrator | Object name with class |
| b [10]:BlogEntry | Array of 10 objects |
| :CMS ref cmsInteraction | Reference to another sequence diagram |

# Naming Objects in Sequence Diagrams



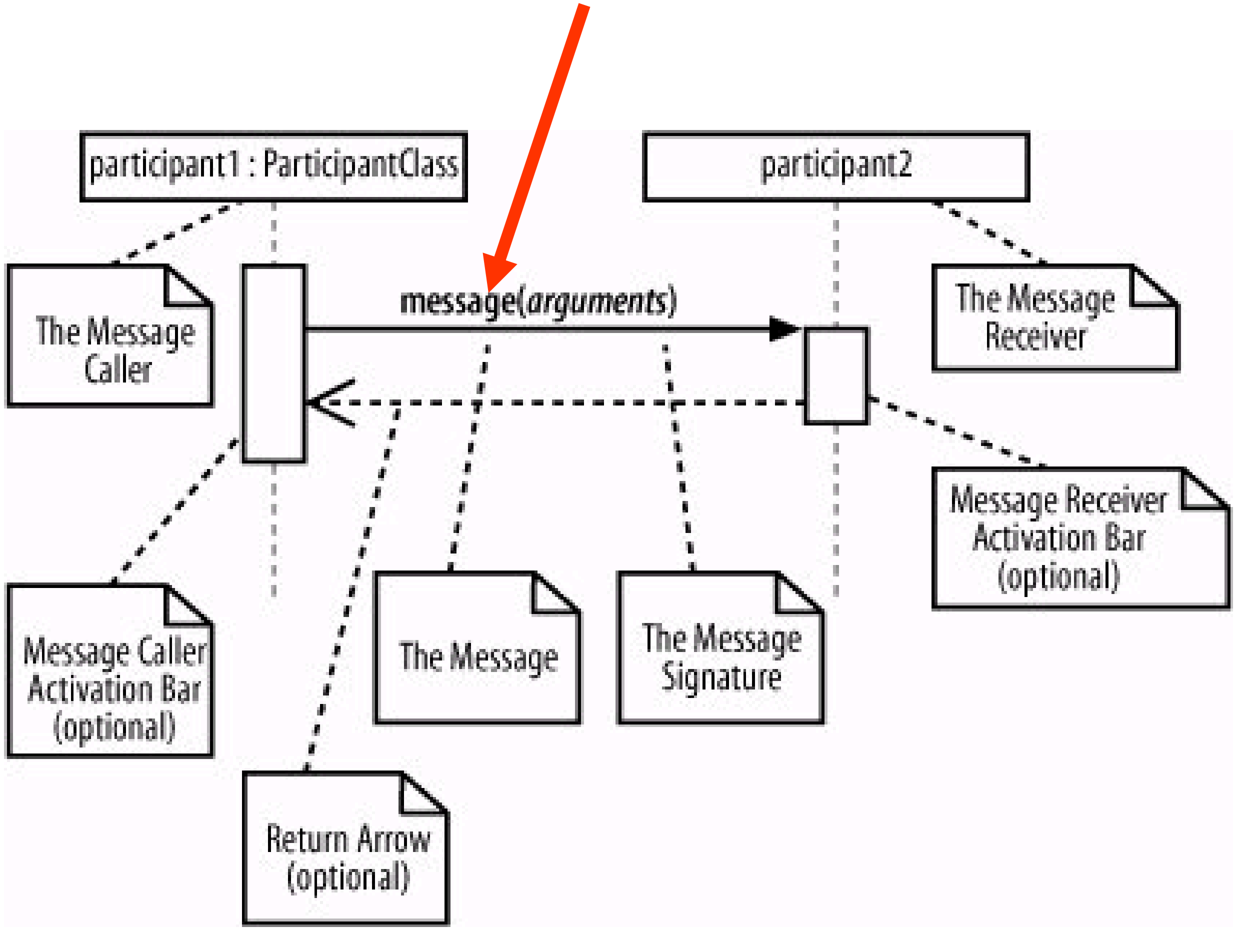
Here you may have either a stick man, class name or object name

# Events and Messages


participant1 : ParticipantClass
participant2 : ParticipantClass2
message
A "Send Message" Event
A "Receive Message" Event

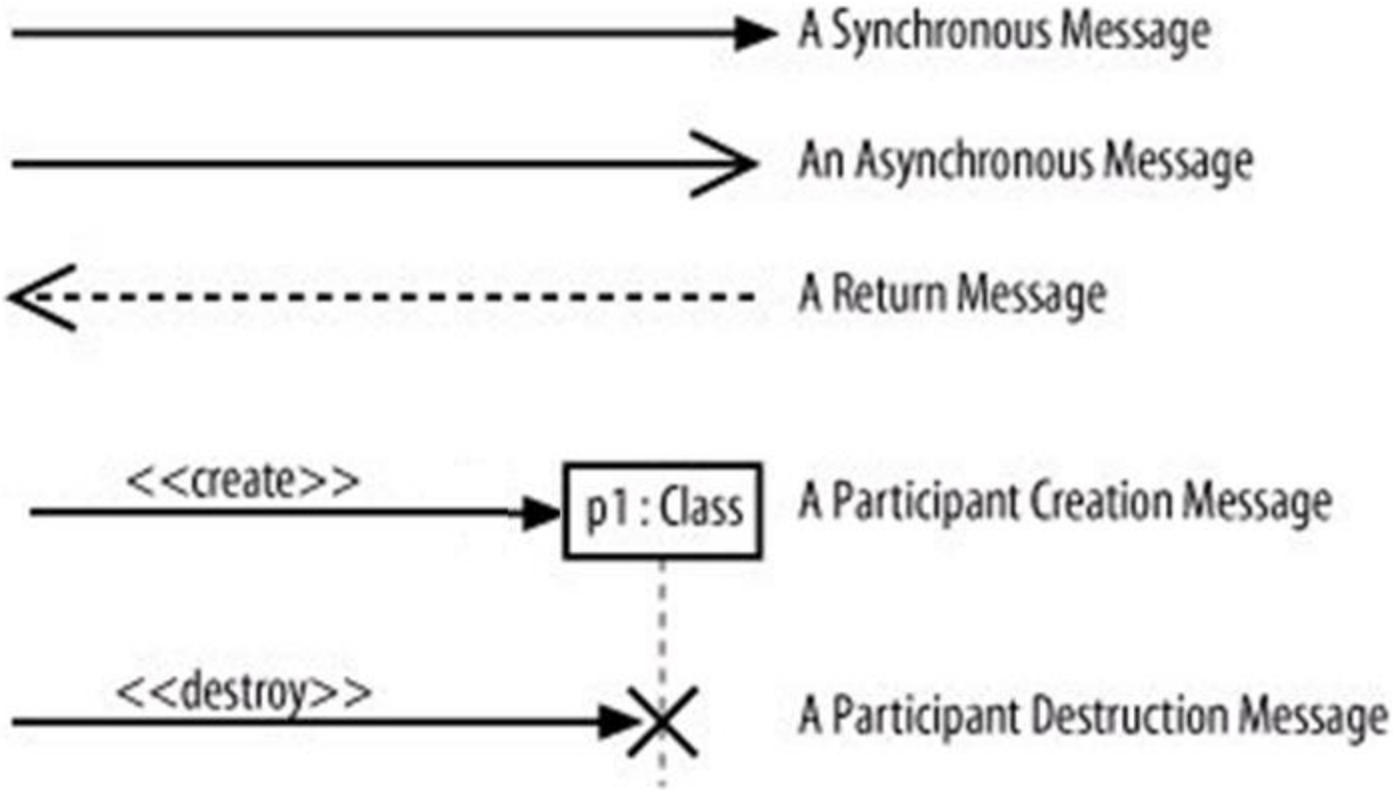
# Events and Messages (2)

- attribute=message_name(arguments):return_type



# Message arrows

- **There are 5 types of message arrows:**



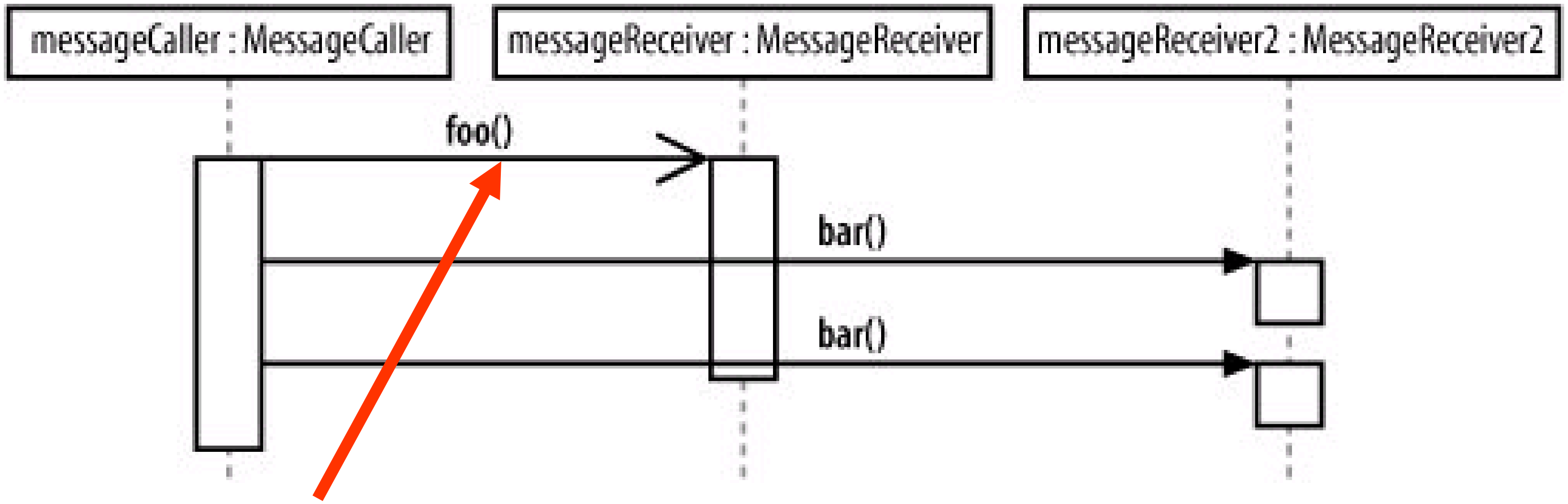
# Message arrows

- **Synchronous message and asynchronous message:**

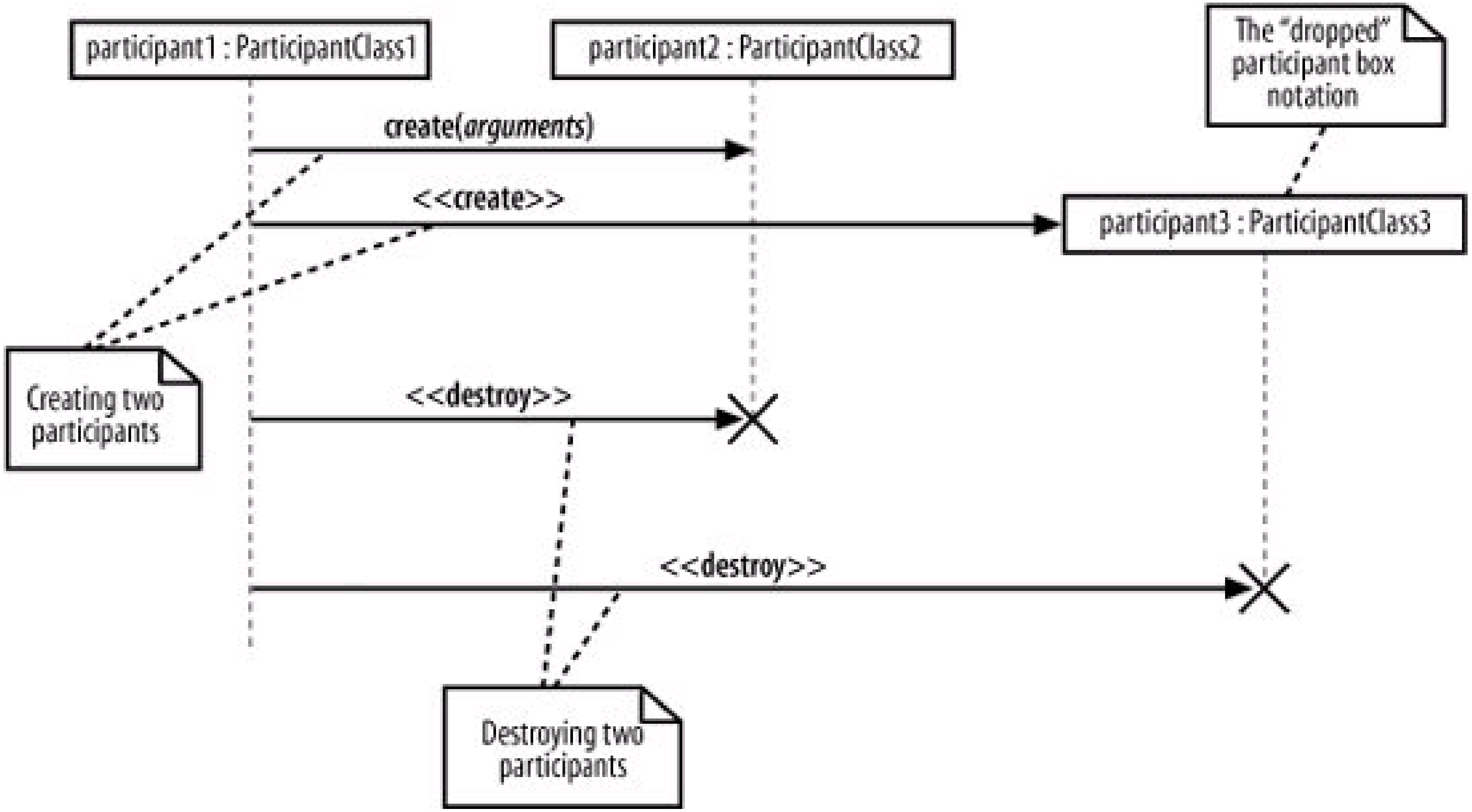
  If a caller sends a **synchronous message**, it must wait until the message is done, such as invoking a subroutine.
  If a caller sends an **asynchronous message**, it can continue processing and doesn’t have to wait for a response

# Message arrows - Async

## Example:

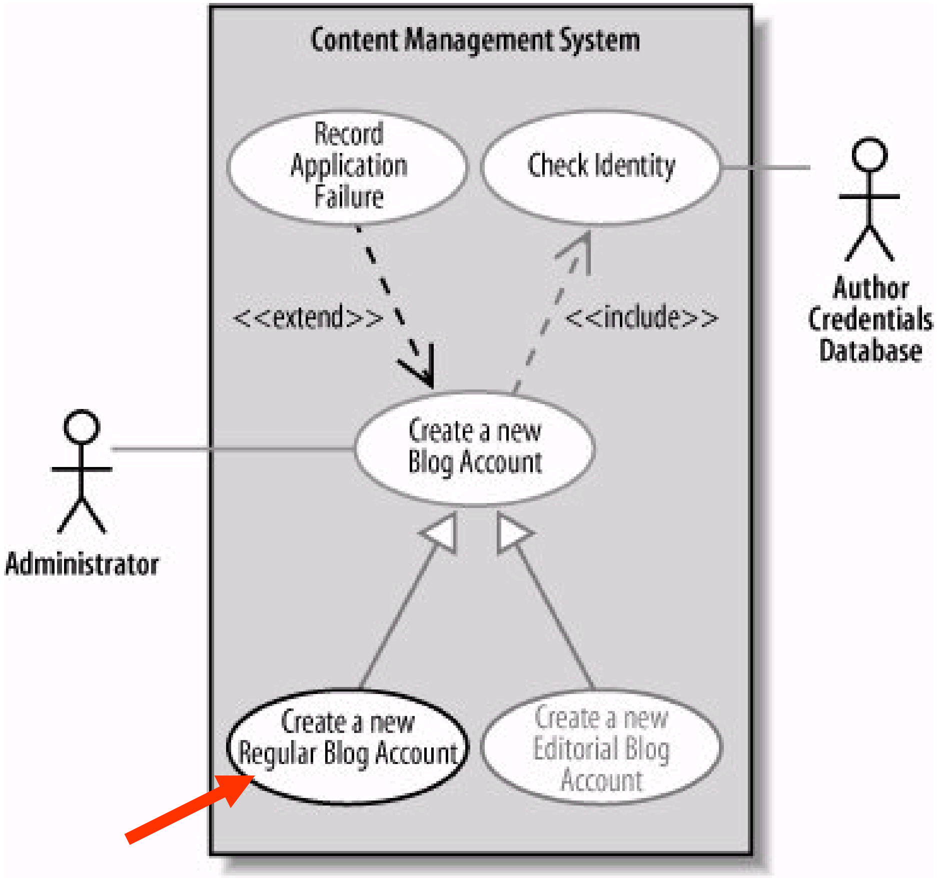

messageCaller : MessageCaller
messageReceiver : MessageReceiver
messageReceiver2 : MessageReceiver2
foo()
bar()
bar()

# <<create>> and <<destroy>>


participant1 : ParticipantClass1
participant2 : ParticipantClass2
The "dropped" participant box notation
create(*arguments*)
<<create>>
participant3 : ParticipantClass3
Creating two participants
<<destroy>>
<<destroy>>
Destroying two participants

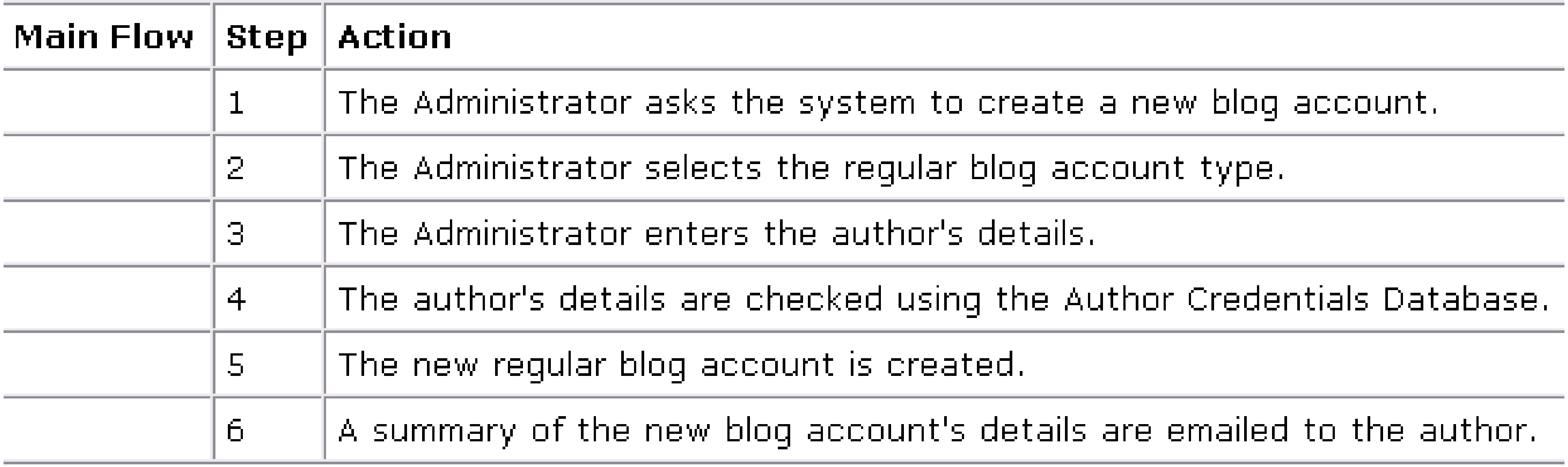
# Realization of use case

- Each use case may have 1 or more sequence diagram
- Sequence diagram describe the flows of events


Content Management System
Record Application Failure
Check Identity
Author Credentials Database
<<extend>>
<<include>>
Administrator
Create a new Blog Account
Create a new Regular Blog Account
Create a new Editorial Blog Account

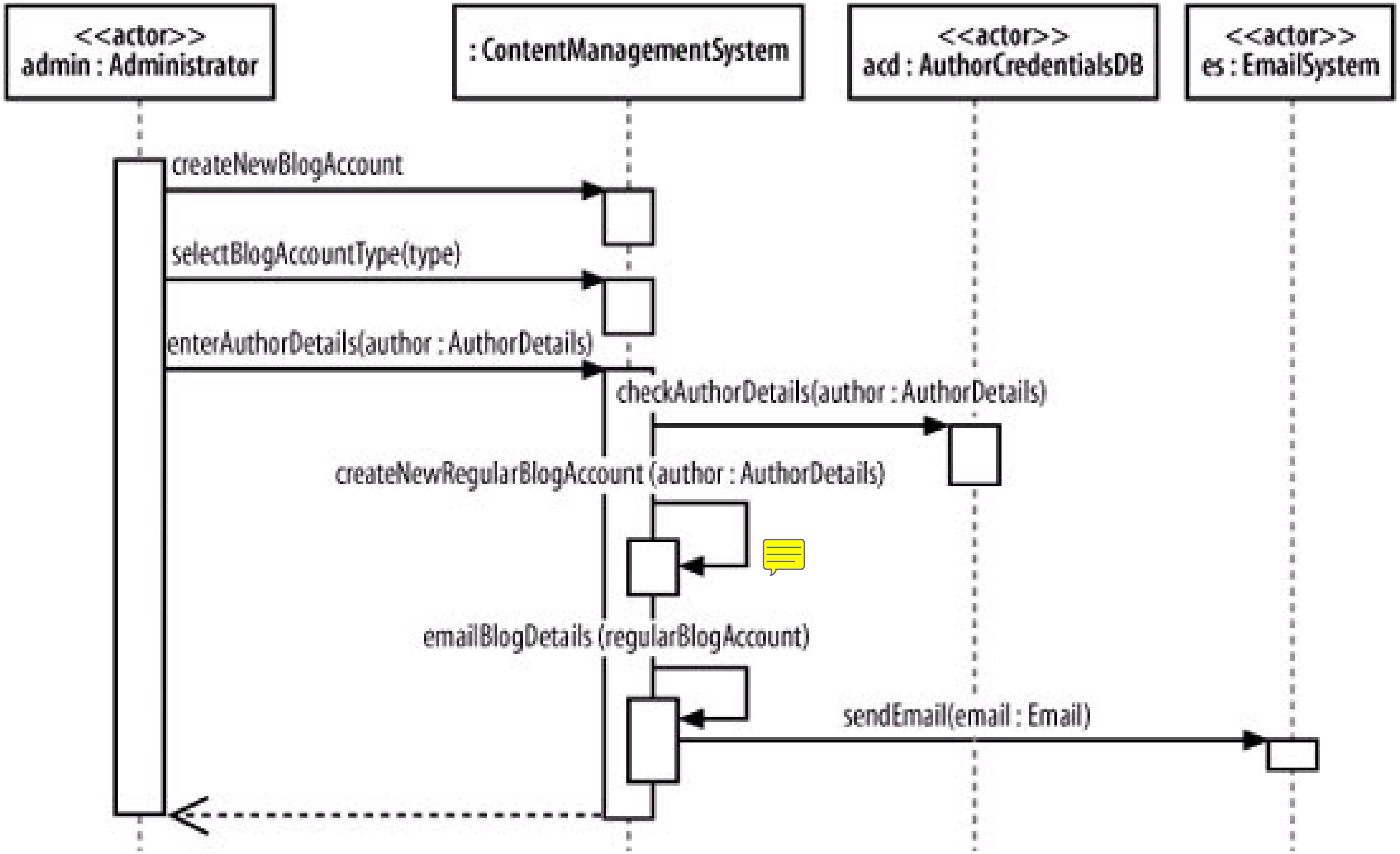
# Realization of use case (2)

## Create new Regular account use case

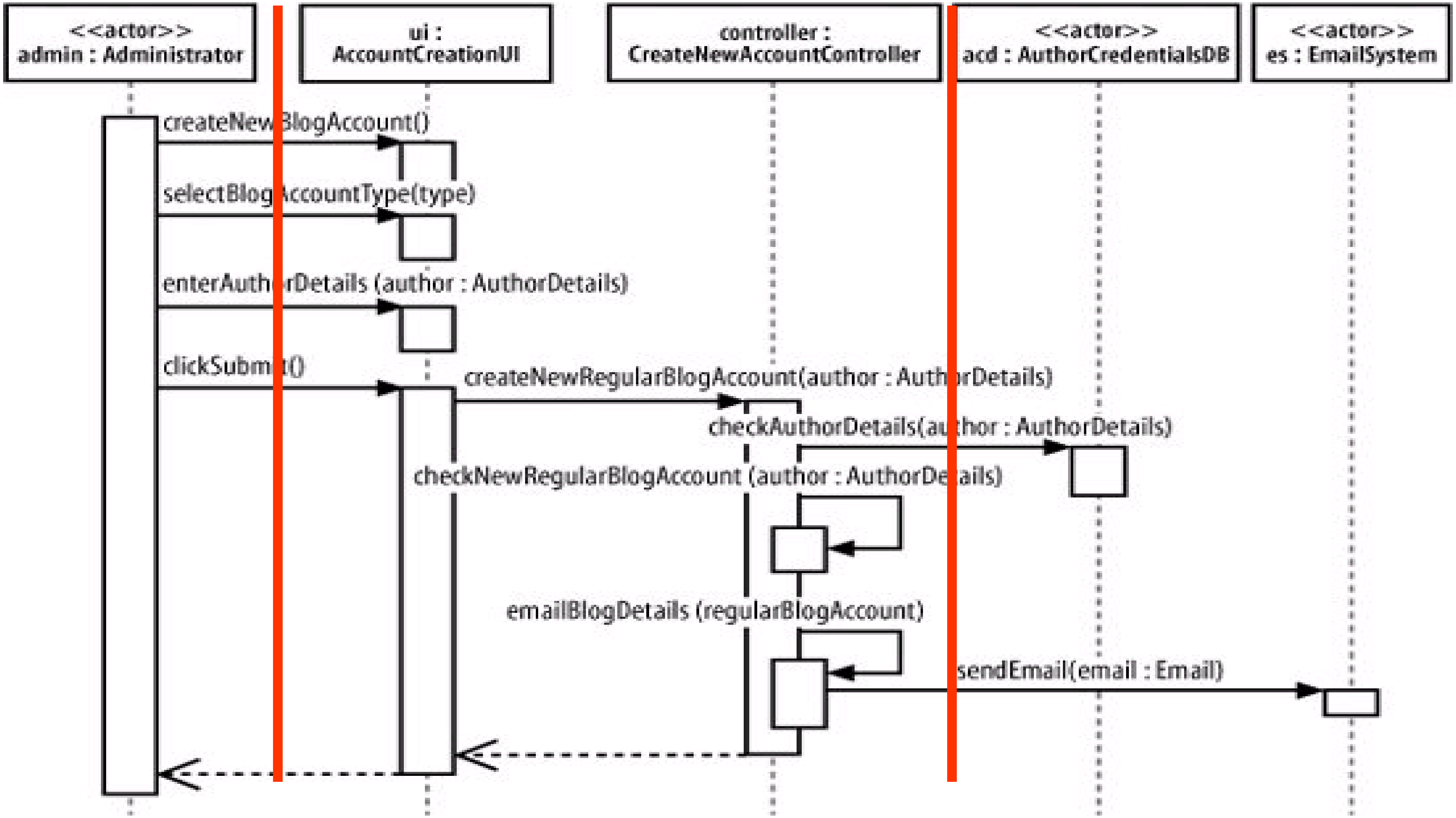


| Main Flow | Step | Action |
|---|---|---|
| | 1 | The Administrator asks the system to create a new blog account. |
| | 2 | The Administrator selects the regular blog account type. |
| | 3 | The Administrator enters the author's details. |
| | 4 | The author's details are checked using the Author Credentials Database. |
| | 5 | The new regular blog account is created. |
| | 6 | A summary of the new blog account's details are emailed to the author. |

# Top level sequence diagram

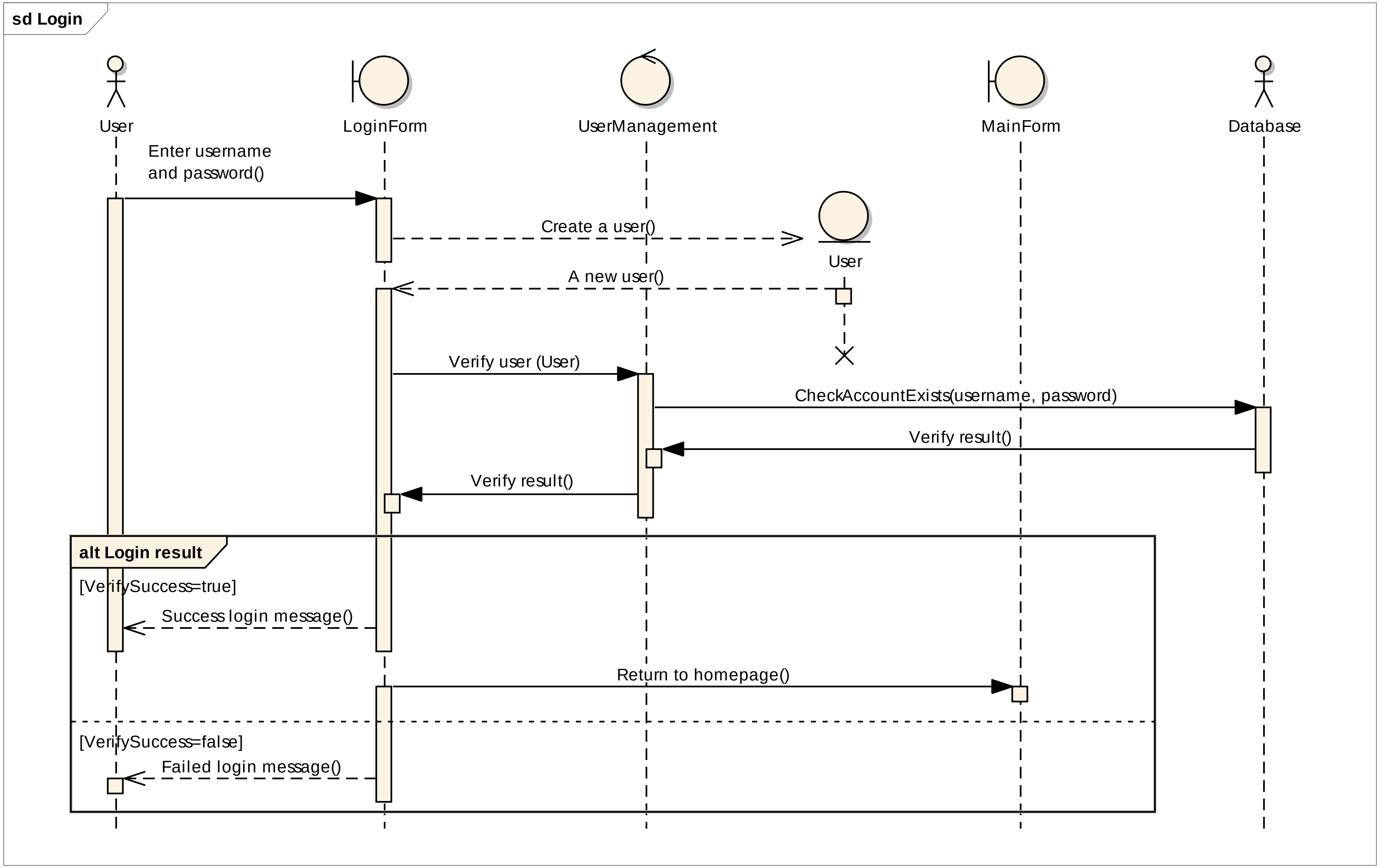

<<actor>>
admin : Administrator
: ContentManagementSystem
<<actor>>
acd : AuthorCredentialsDB
<<actor>>
es : EmailSystem
createNewBlogAccount
selectBlogAccountType(type)
enterAuthorDetails(author : AuthorDetails)
checkAuthorDetails(author : AuthorDetails)
createNewRegularBlogAccount (author : AuthorDetails)
emailBlogDetails (regularBlogAccount)
sendEmail(email : Email)

# More details


<<actor>>
admin : Administrator
ui :
AccountCreationUI
controller :
CreateNewAccountController
<<actor>>
acd : AuthorCredentialsDB
<<actor>>
es : EmailSystem
createNewBlogAccount()
selectBlogAccountType(type)
enterAuthorDetails (author : AuthorDetails)
clickSubmit()
createNewRegularBlogAccount(author : AuthorDetails)
checkAuthorDetails(author : AuthorDetails)
checkNewRegularBlogAccount (author : AuthorDetails)
emailBlogDetails (regularBlogAccount)
sendEmail(email : Email)

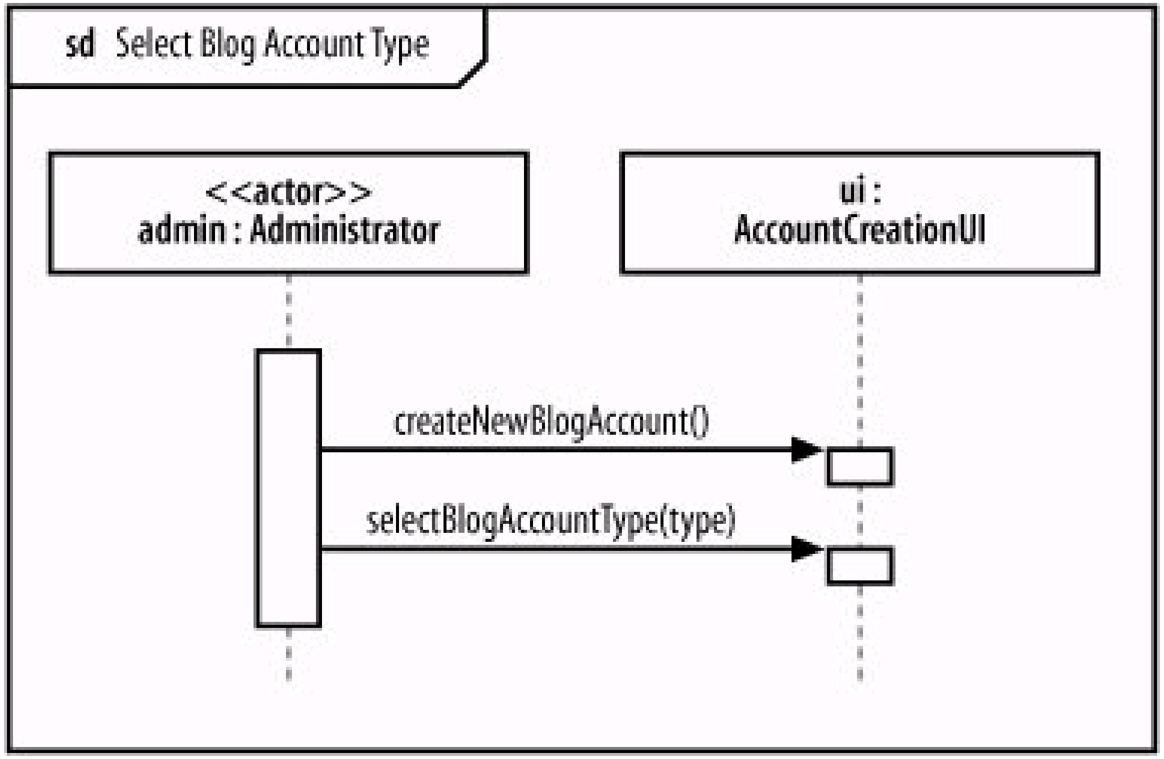
# 2. Mapping Usecase to Objects

## Example of "Login" usecase

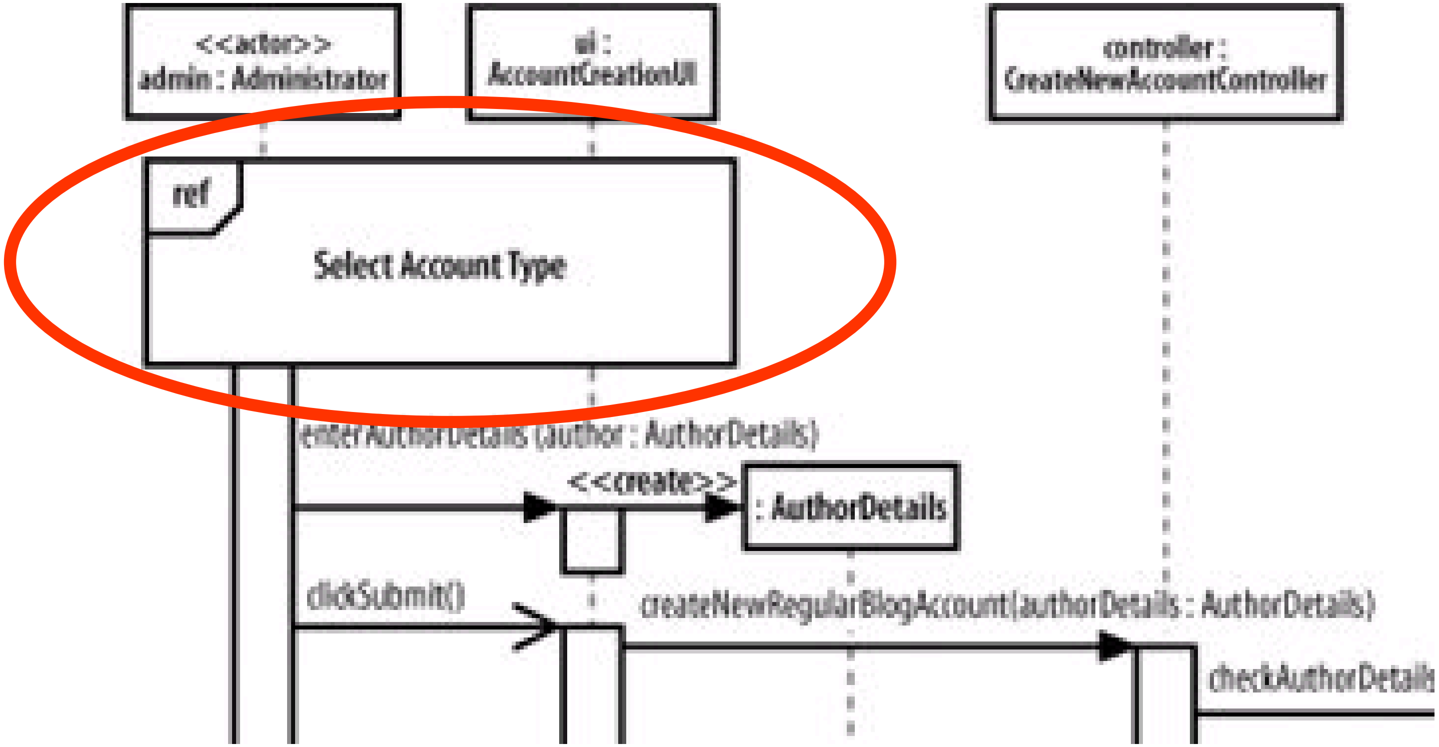


# Fragments in sequence diagram

- **A sequence diagram may be broken into fragments**



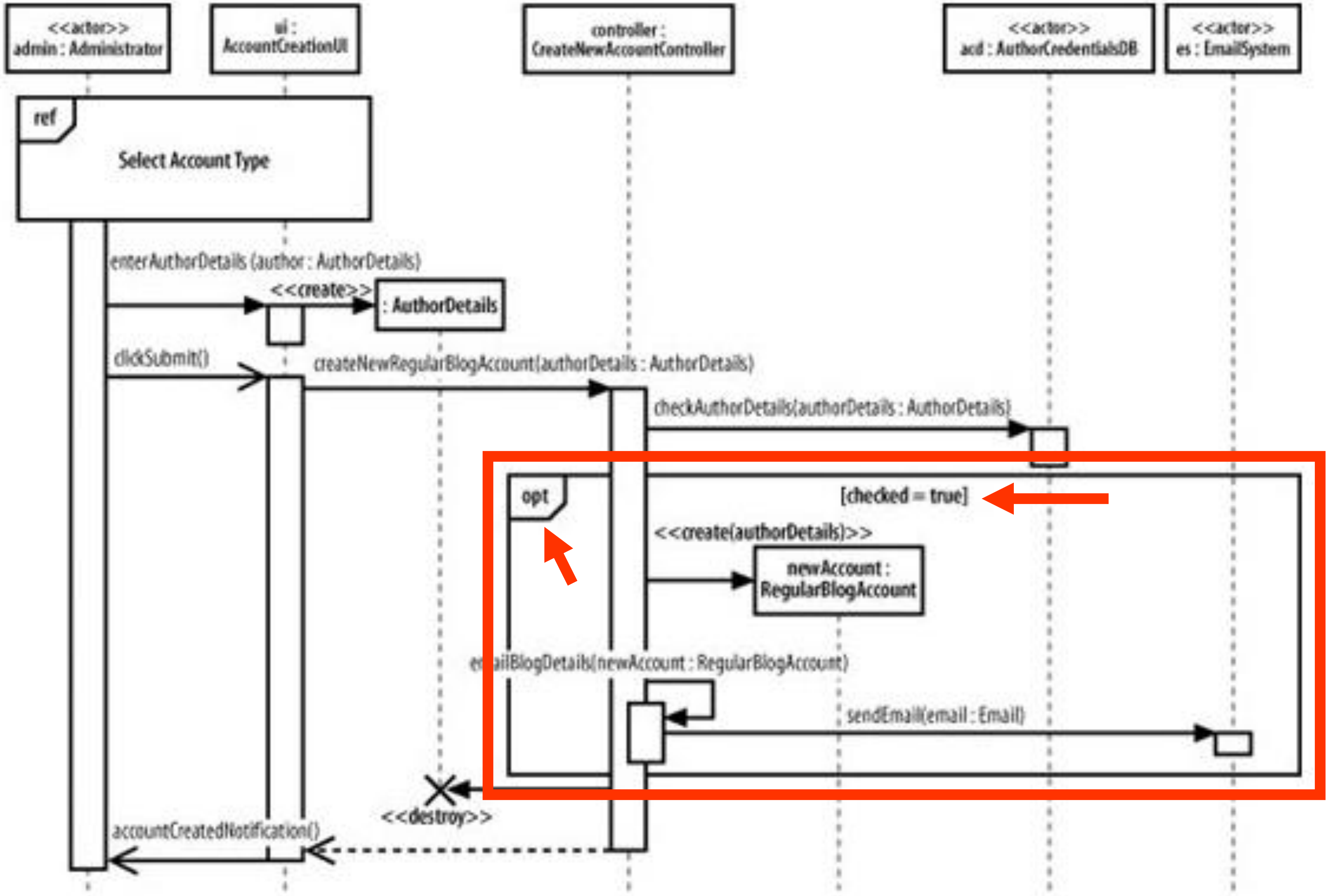
# Using fragments



...

...

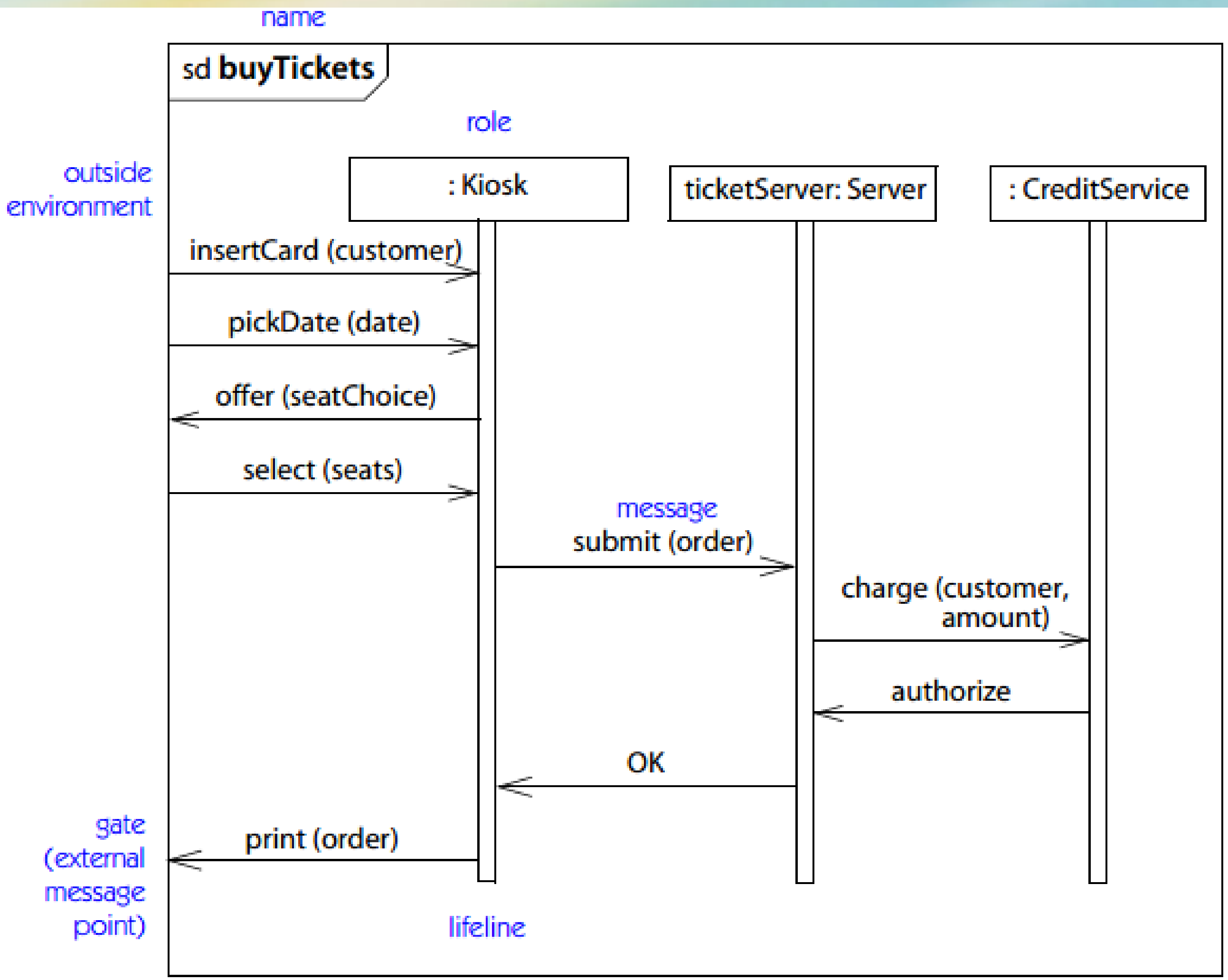
# Using fragments (2)

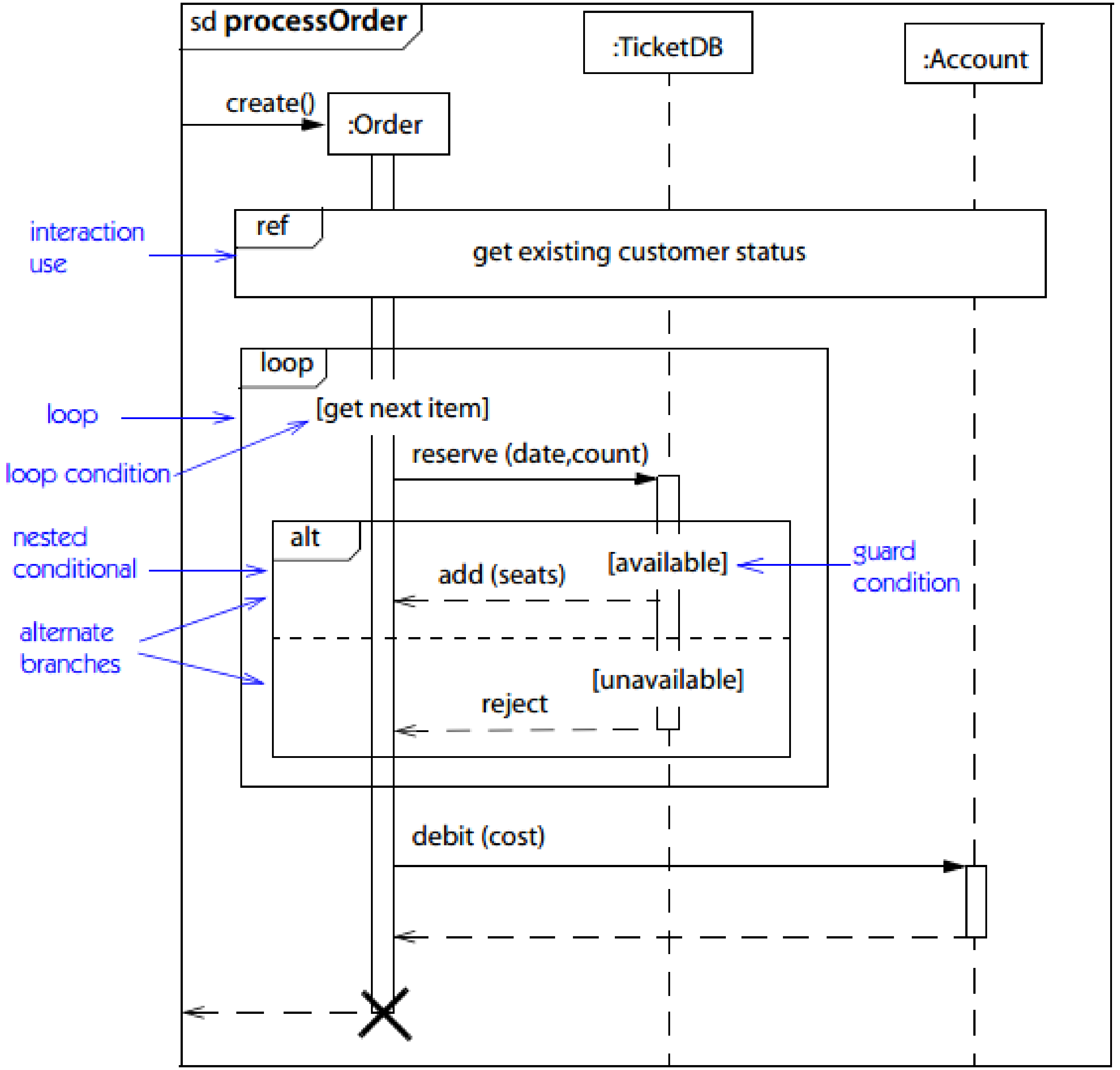

<<actor>>
admin : Administrator
ui :
AccountCreationUI
controller :
CreateNewAccountController
<<actor>>
acd : AuthorCredentialsDB
<<actor>>
es : EmailSystem
ref
Select Account Type
enterAuthorDetails (author : AuthorDetails)
<<create>>
: AuthorDetails
clickSubmit()
createNewRegularBlogAccount(authorDetails : AuthorDetails)
checkAuthorDetails(authorDetails : AuthorDetails)
opt
[checked = true]
<<create(authorDetails)>>
newAccount :
RegularBlogAccount
emailBlogDetails(newAccount : RegularBlogAccount)
sendEmail(email : Email)
<<destroy>>
accountCreatedNotification()

# Some fragment types

- ref: an interaction defined elsewhere
- loop: repeat the interactions in the fragment many times
- alt [guard conditions]: execute the corresponding set of interactions, base on which guard condition is true
- opt [check]: optional fragment, executed only if the check value is true
- par: the interactions can be executed in parallel

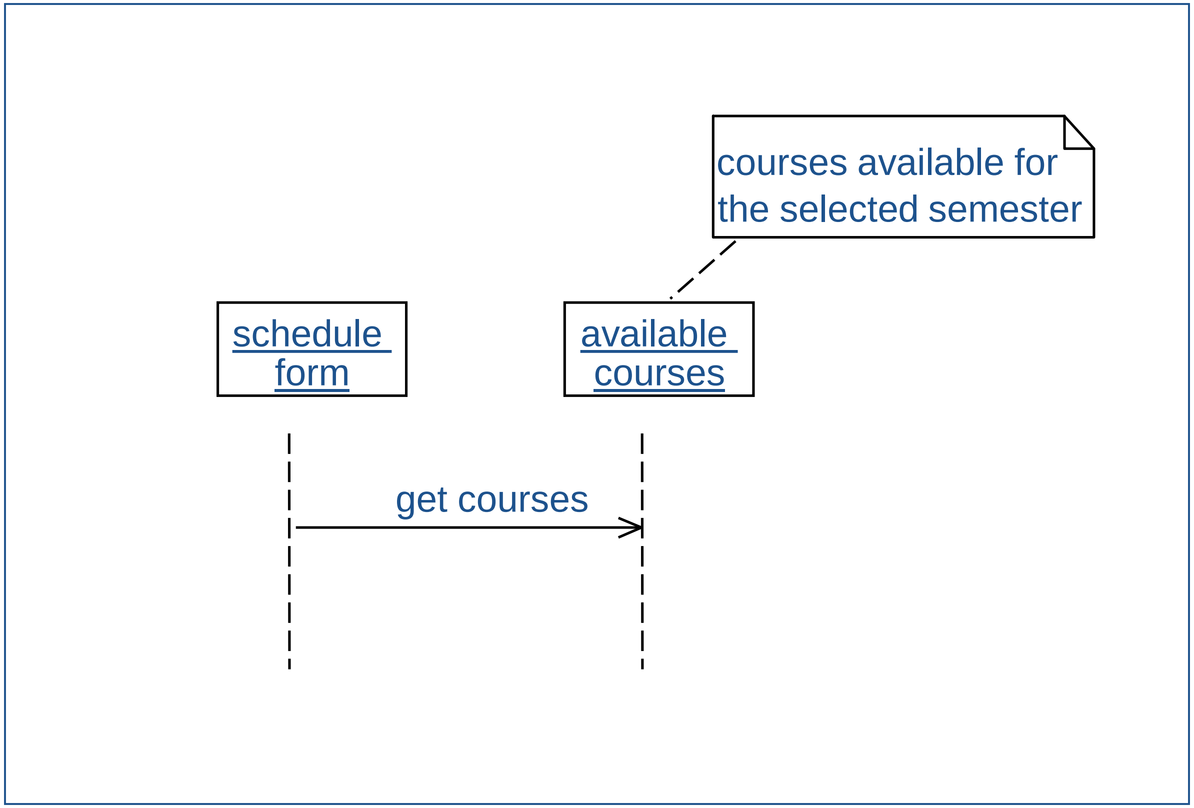
# Examples


name
sd buyTickets
role
outside environment
: Kiosk
ticketServer: Server
: CreditService
insertCard (customer)
pickDate (date)
offer (seatChoice)
select (seats)
message
submit (order)
charge (customer, amount)
authorize
OK
gate (external message point)
print (order)
lifeline


sd processOrder
:TicketDB
:Account
create()
:Order
interaction use
ref
get existing customer status
loop
loop
[get next item]
loop condition
reserve (date,count)
nested conditional
alt
add (seats)
[available]
guard condition
alternate branches
[unavailable]
reject
debit (cost)

# Notes

- Notes may be added to add more information to the diagram



# II. Analysis activities

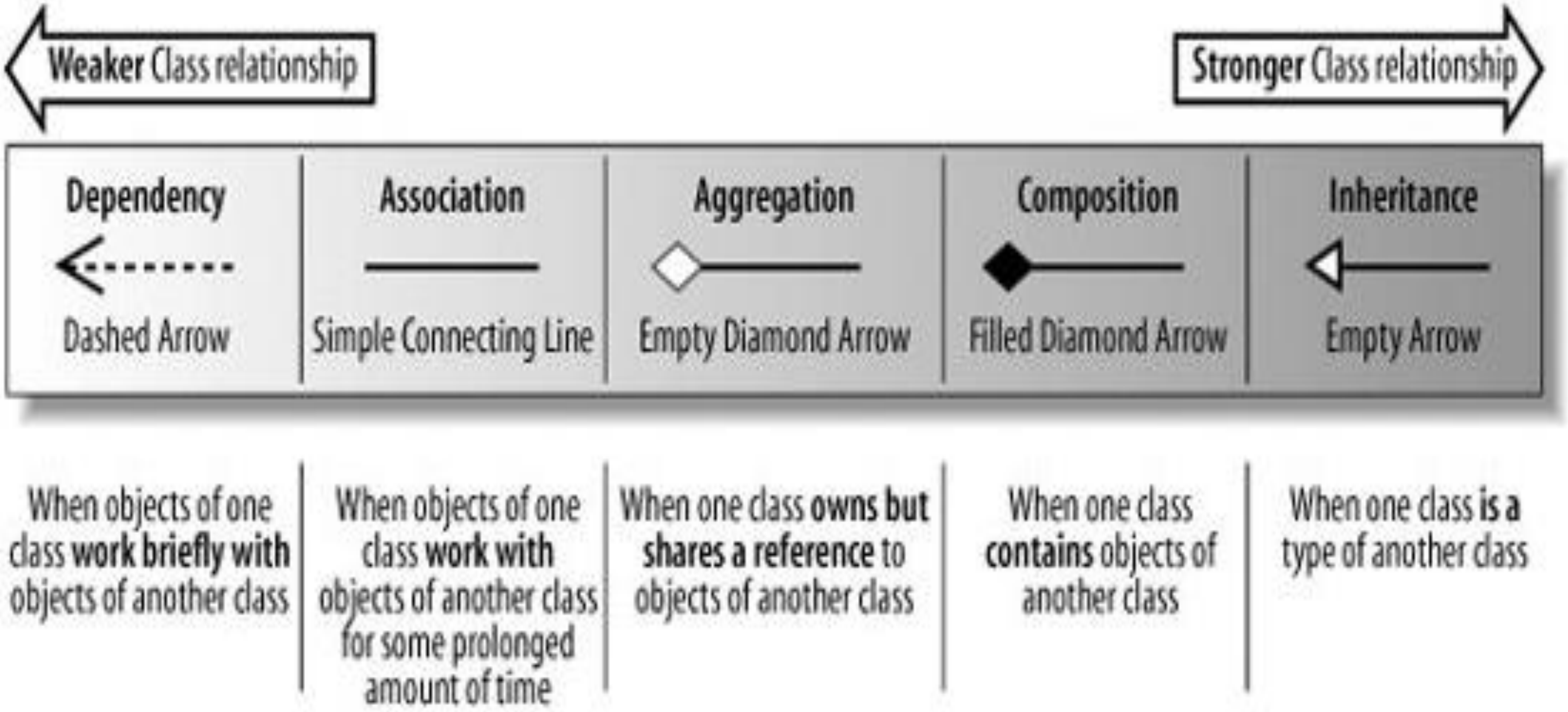
1. Identifying Objects
2. Mapping Usecase to Objects (with Sequence diagrams)
3. **Identifying Class relationship**
4. Identifying Attributes
5. Modeling State-dependent Behavior of Objects

# Relationships

- **Types of relationship**
  - Association (Kết hợp)
  - Aggregation (Thu nạp)
  - Composition (Hợp thành)
  - Dependency (Phụ thuộc)
  - Generalization/Inheritance (Kế thừa)
  - Realization (Hiện thực hóa)

# Class relationships

- **Dependency (“uses”)**
- **Association (“uses”)**
- **Aggregation (“has”)**
- **Composition (“has”)**
- **Inheritance (“is”)**

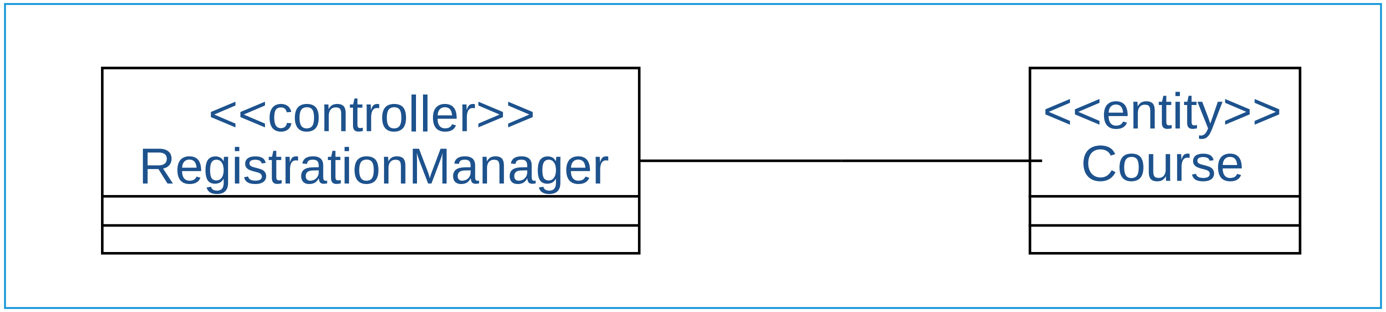


**Weaker** Class relationship ⟵ ⟶ **Stronger** Class relationship

| Dependency | Association | Aggregation | Composition | Inheritance |
|---|---|---|---|---|
| Dashed Arrow | Simple Connecting Line | Empty Diamond Arrow | Filled Diamond Arrow | Empty Arrow |
| When objects of one class **work briefly with** objects of another class | When objects of one class **work with** objects of another class for some prolonged amount of time | When one class **owns but shares a reference** to objects of another class | When one class **contains** objects of another class | When one class **is a** type of another class |

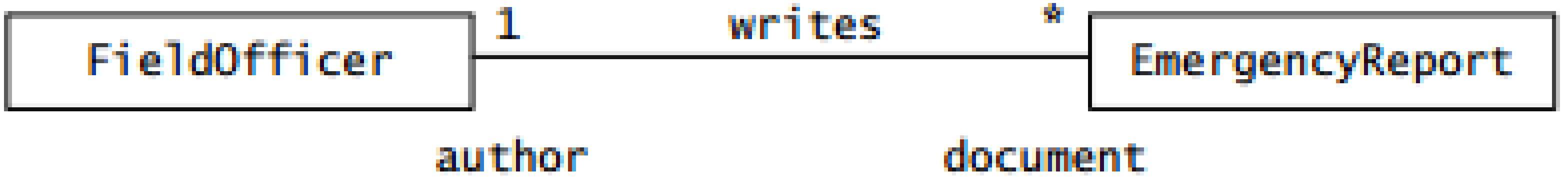
# Associations

- An association is a bi-directional semantic connection between classes
  - This implies that there is a link between objects in the associated classes
- Associations are represented on class diagrams by a line connecting the associated classes
- Data may flow in either direction or both directions across a link



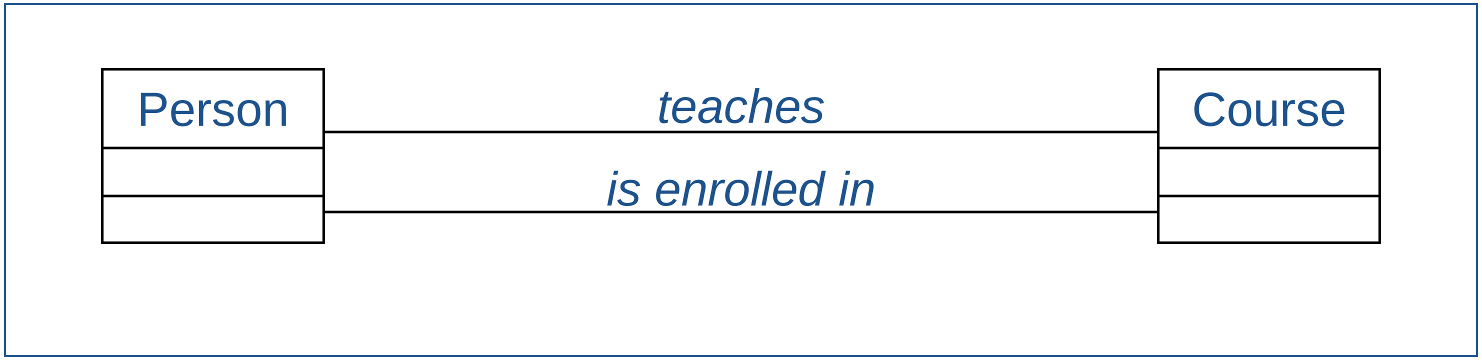
# Associations

- To clarify its meaning, an association may be named
- A role denotes the purpose or capacity wherein one class associates with another



# Multiple Associations

- More than one association may exist between two classes
- If there is more than one association between two classes then they MUST be named



# Multiplicity for Associations

- Multiplicity  is the number of instances of one class related to ONE instance of the other class
- For each association, there are two multiplicity decisions to make: one for each end of the association
- For example, in the connection between Person playing the role of the teacher and Course
  - For each instance of Person, many (i.e., zero or more) Courses may be taught
  - For each instance of Course, exactly one Person is the teacher

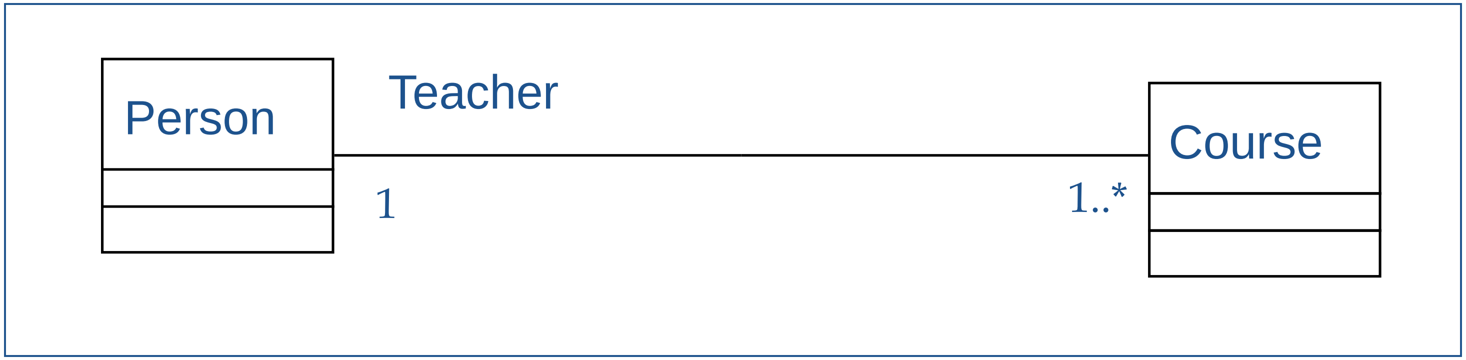
# Multiplicity Indicators

- Each end of an association contains a multiplicity indicator
  - Indicates the number of objects participating in the relationship

| | |
|---|---|
| Many | * |
| Exactly one | 1 |
| Zero or more | 0..* |
| One or more | 1..* |
| Zero or one | 0..1 |
| Specified range | 2..4 |

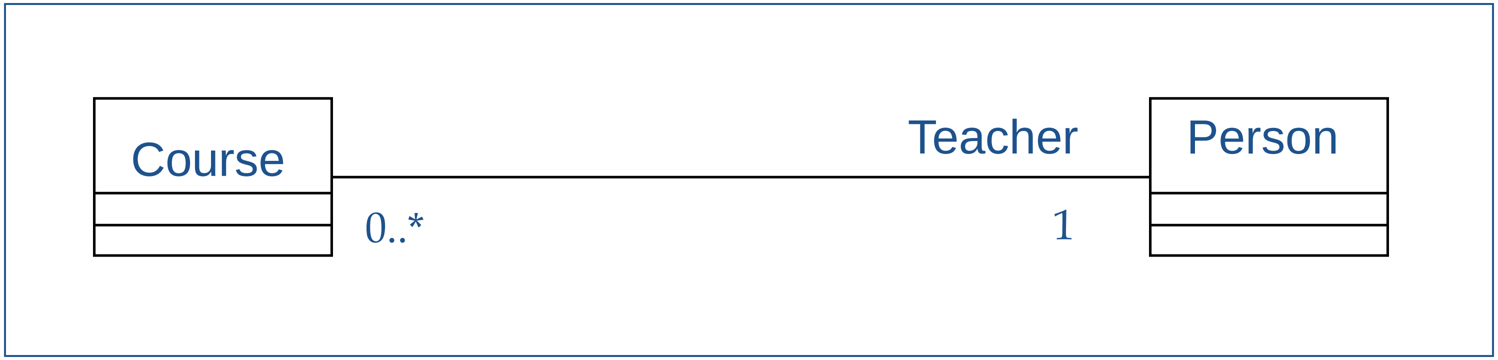
# Example: Multiplicity

- Multiplicity decisions expose many hidden assumptions about the problem being modeled
  - Can a teacher be on sabbatical?
  - Can a course have two teachers?



# What Does Multiplicity Mean?

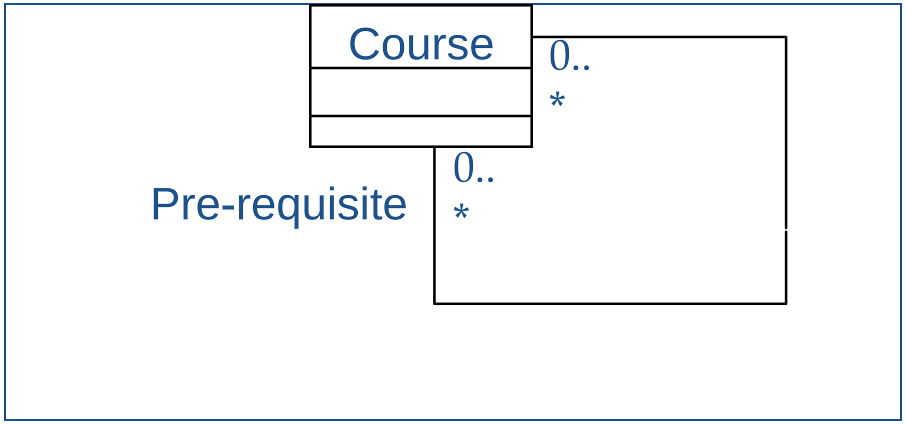
- Multiplicity answers two questions
  - Is the association mandatory or optional?
  - What is the minimum and maximum number of instances that can be linked to one instance?



What does this diagram tell you?

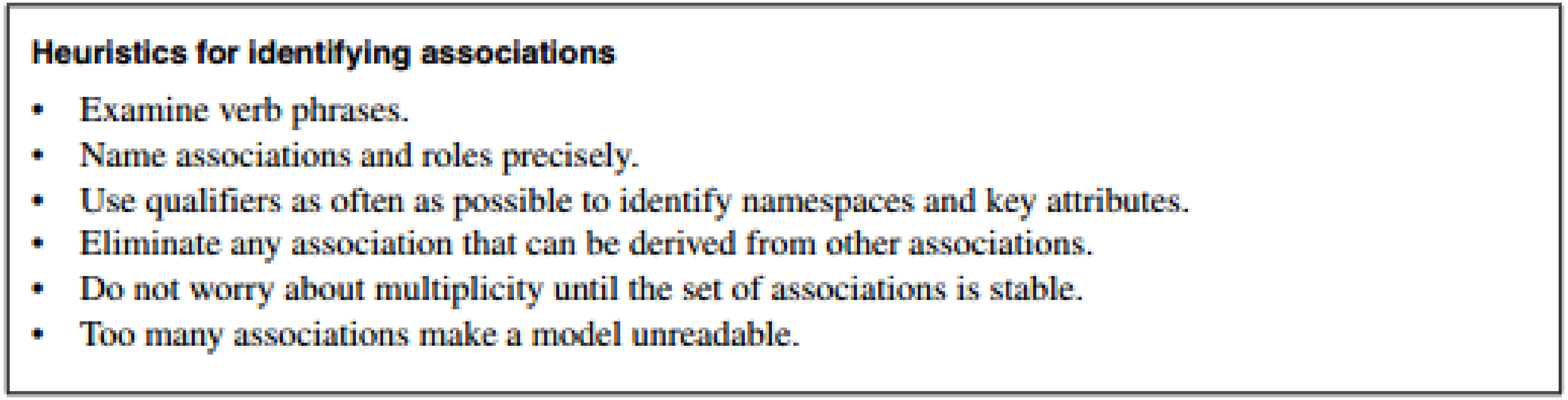
# Reflexive Associations

- In a reflexive association, objects in the same class are related
  - Indicates that multiple objects in the same class collaborate together in some way



A course may have many pre-requisites
A course may be a pre-requisite for many other courses

# Associations



**Heuristics for identifying associations**

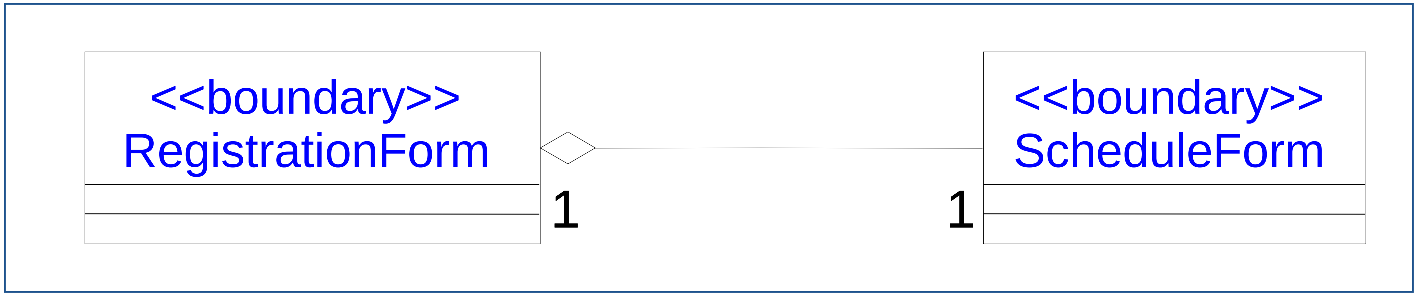
- Examine verb phrases.
- Name associations and roles precisely.
- Use qualifiers as often as possible to identify namespaces and key attributes.
- Eliminate any association that can be derived from other associations.
- Do not worry about multiplicity until the set of associations is stable.
- Too many associations make a model unreadable.

# Relationships

- **Types of relationship**
  - Association (Kết hợp)
  - **Aggregation (Thu nạp)**
  - Composition (Hợp thành)
  - Dependency (Phụ thuộc)
  - Generalization/Inheritance (Kế thừa)
  - Realization (Hiện thực hóa)

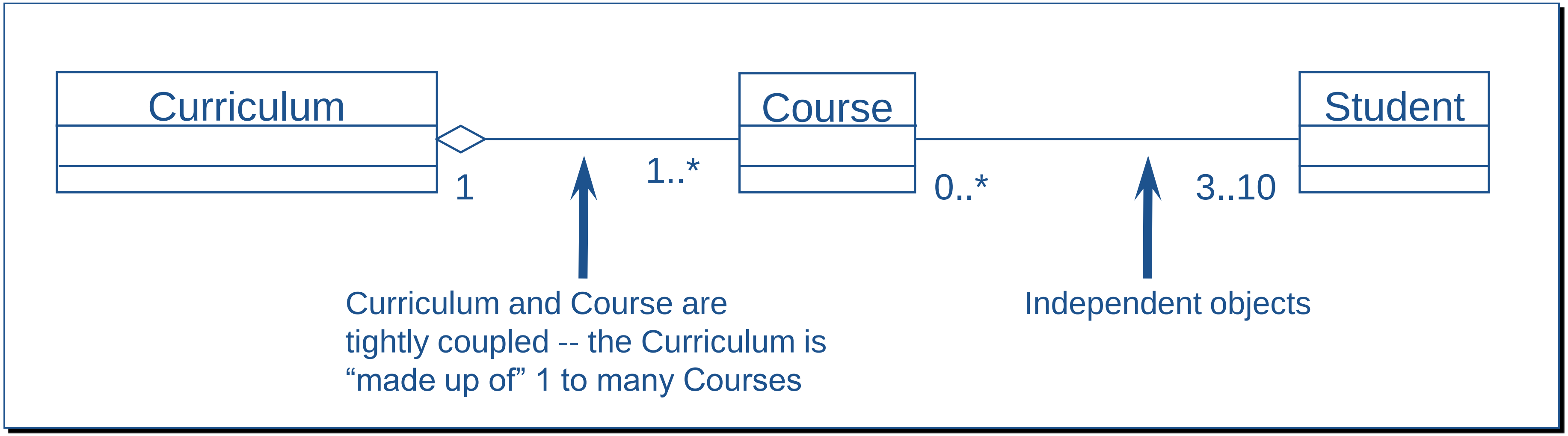
# Aggregation

- Aggregation is a specialized form of association in which a whole is related to its part(s)
  - Aggregation is known as a “part-of” or containment relationship
- An aggregation is represented as an association with a diamond next to the class denoting the aggregate (whole)
- Multiplicity is represented in the same manner as other associations



# Association or Aggregation?

- If two objects are tightly bound by a whole-part relationship
  - The relationship is an aggregation
- If two objects are usually considered as independent, even though they are often linked
  - The relationship is an association

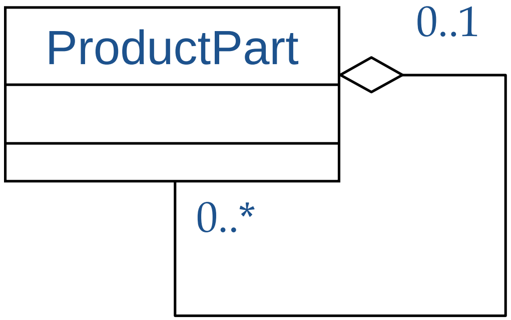


Curriculum and Course are tightly coupled -- the Curriculum is “made up of” 1 to many Courses

Independent objects

# Reflexive Aggregates

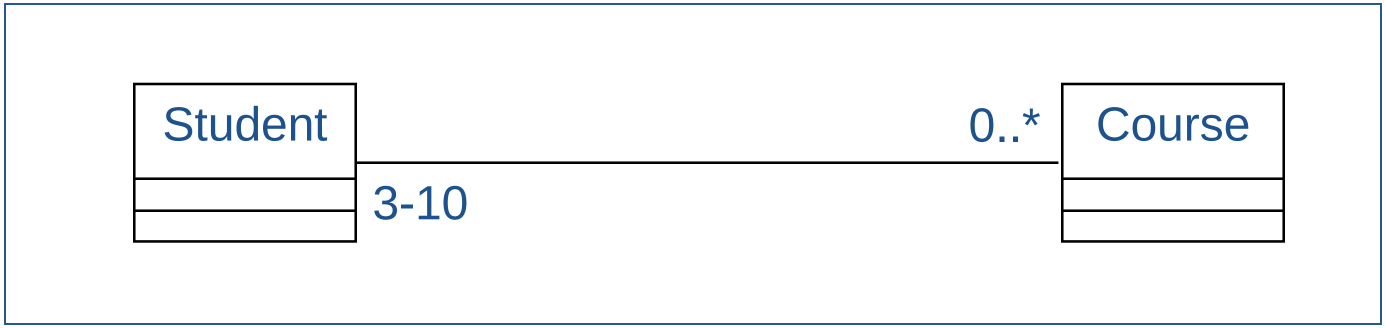
- Aggregates can also be reflexive
  - Classic bill of materials type problem
- This indicates a recursive relationship



One ProductPart object contains
zero or more ProductPart objects

# Association Classes

- We wish to track the grades for all courses a student has taken
- The relationship between student and Course is a many-to-many relationship
- Where do we place the attribute grade?

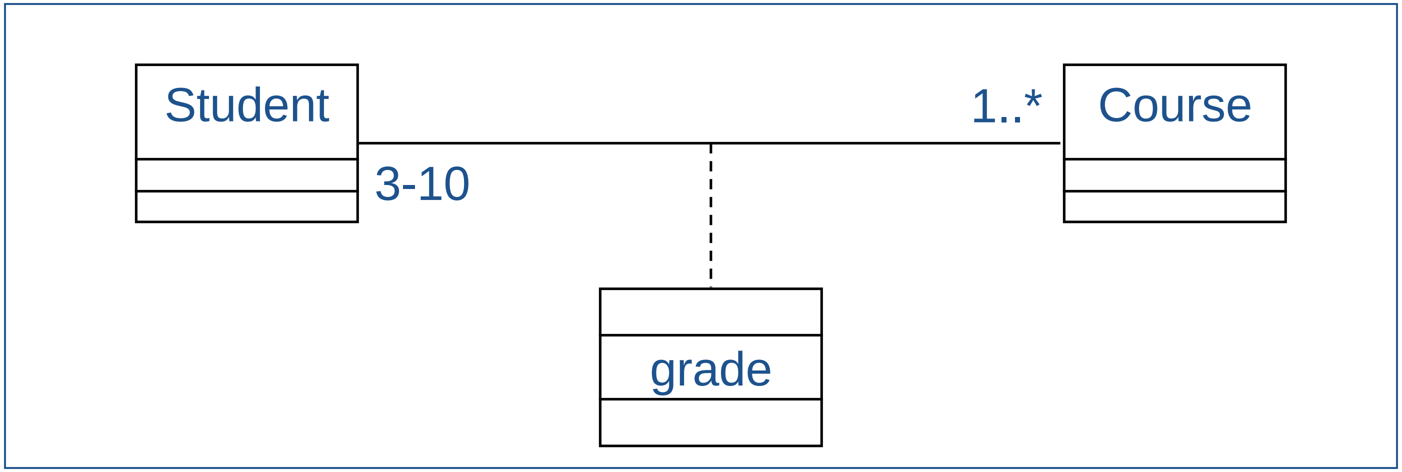


# Association Classes (cont.)

- The attribute grade cannot be placed in the Course class because there are (potentially) many links to many Student objects
- The attribute grade cannot be placed in the Student class because there are (potentially) many links to many Course objects
- Therefore, the attribute really belongs to the individual Student-Course link
- An association class is used to hold the link information

# Drawing Association Classes

- Create an association class using the class icon
- Connect the class icon to the association line using a dashed line
- The association class may include multiple properties of the association
- Only one association class is permitted per association

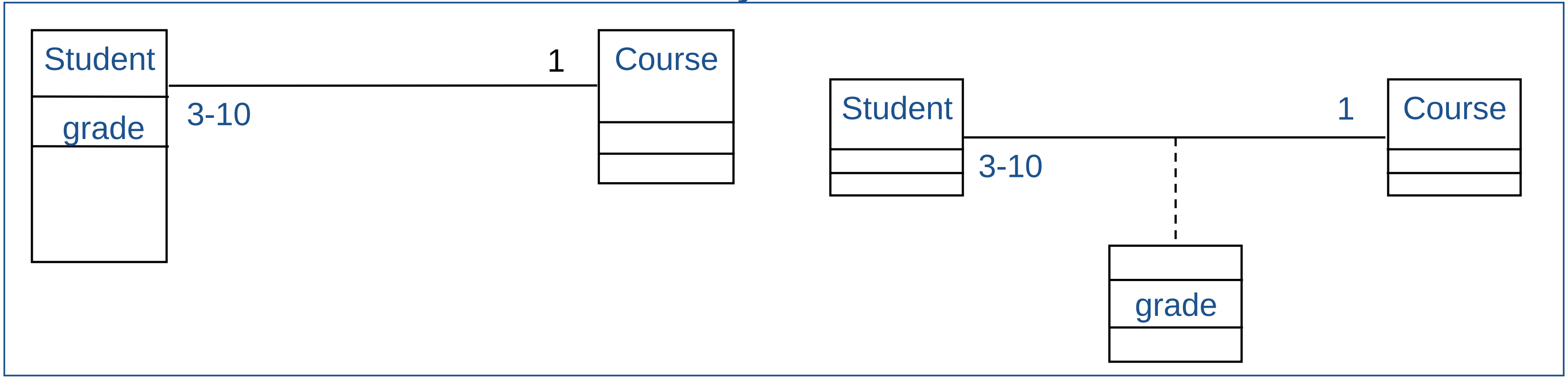


# Association Classes and Multiplicity

- Association classes are often used for many-to-many associations
- If the multiplicity at either end of an association is “to-one”
  - The attribute may be placed within the class on the many side of the relationship

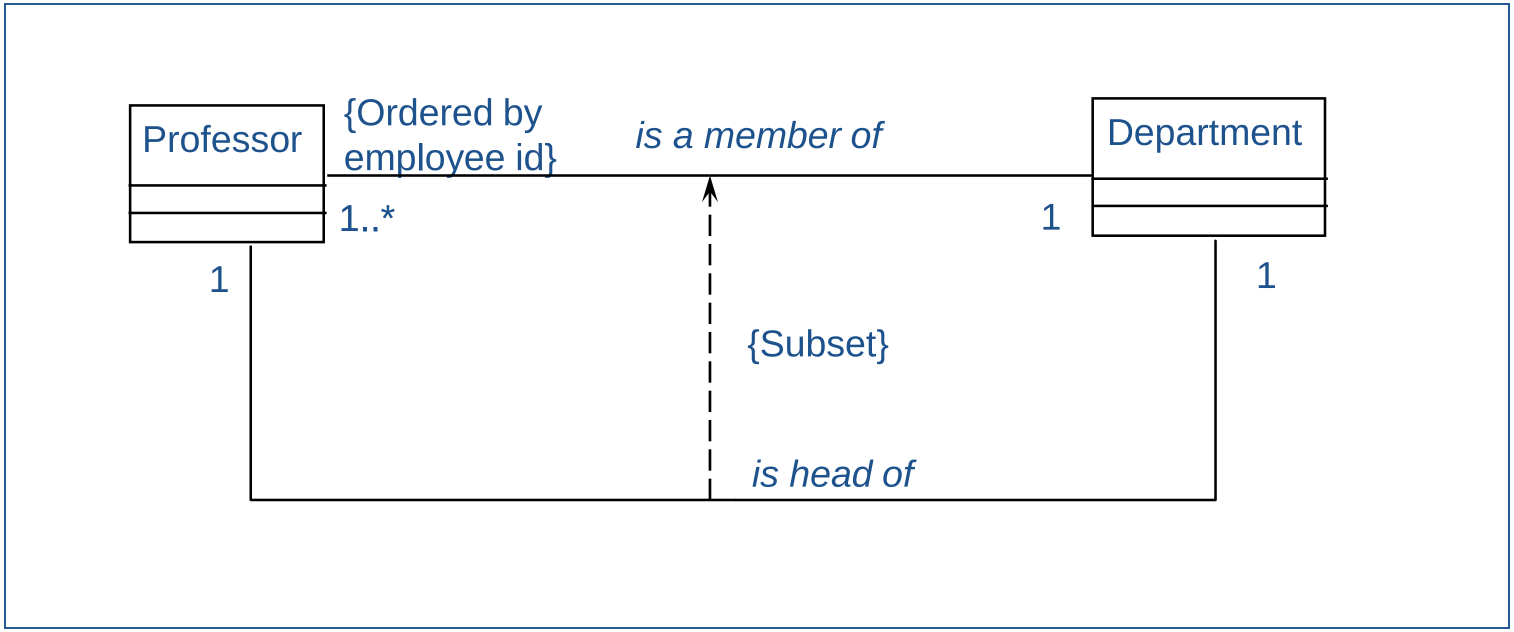
OR

  - An association class may still be used



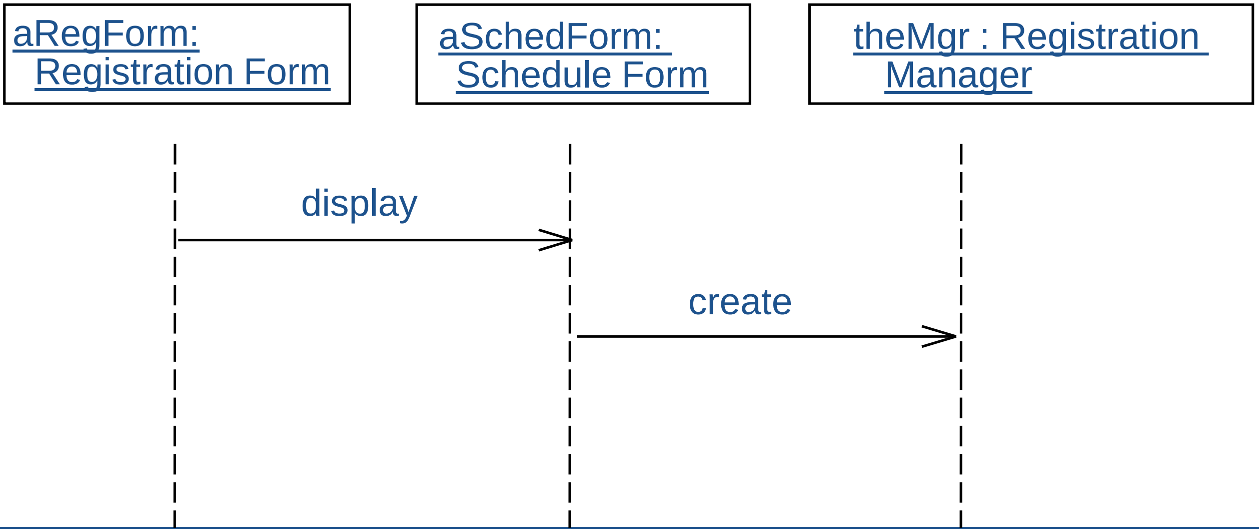
# Constraints

- A constraint is the expression of some condition that must be preserved
  - A constraint is shown within curly braces



# Finding Associations and Aggregations

- Scenarios may be examined to determine if a relationship should exist between two classes
  - Two objects can communicate only if they “know” one another
- Associations and/or aggregations provide a pathway for communication

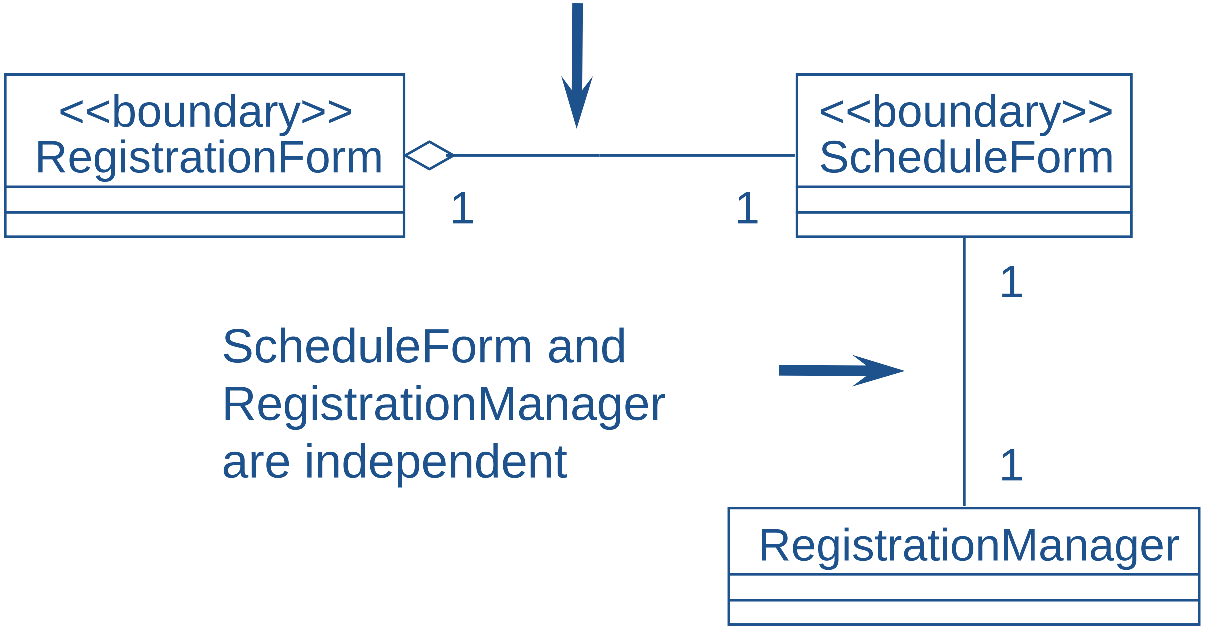

aRegForm: Registration Form
aSchedForm: Schedule Form
theMgr : Registration Manager
display
create

RegistrationForm “talks” to ScheduleForm

ScheduleForm “talks” to RegistrationManager

# Association or Aggregation?

RegistrationForm and ScheduleForm are tightly coupled  -- a ScheduleForm is “part of” the RegistrationForm



ScheduleForm and RegistrationManager are independent

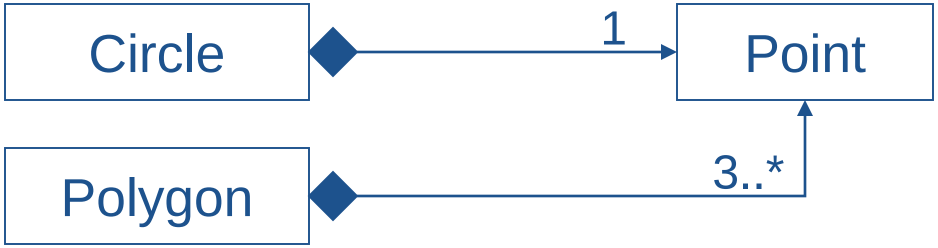
# Relationships

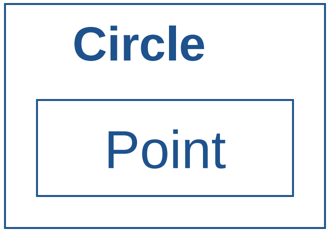
- **Types of relationship**
  - Association (Kết hợp)
  - Aggregation (Thu nạp)
  - **Composition (Hợp thành)**
  - Dependency (Phụ thuộc)
  - Generalization/Inheritance (Kế thừa)
  - Realization (Hiện thực hóa)

# Composition

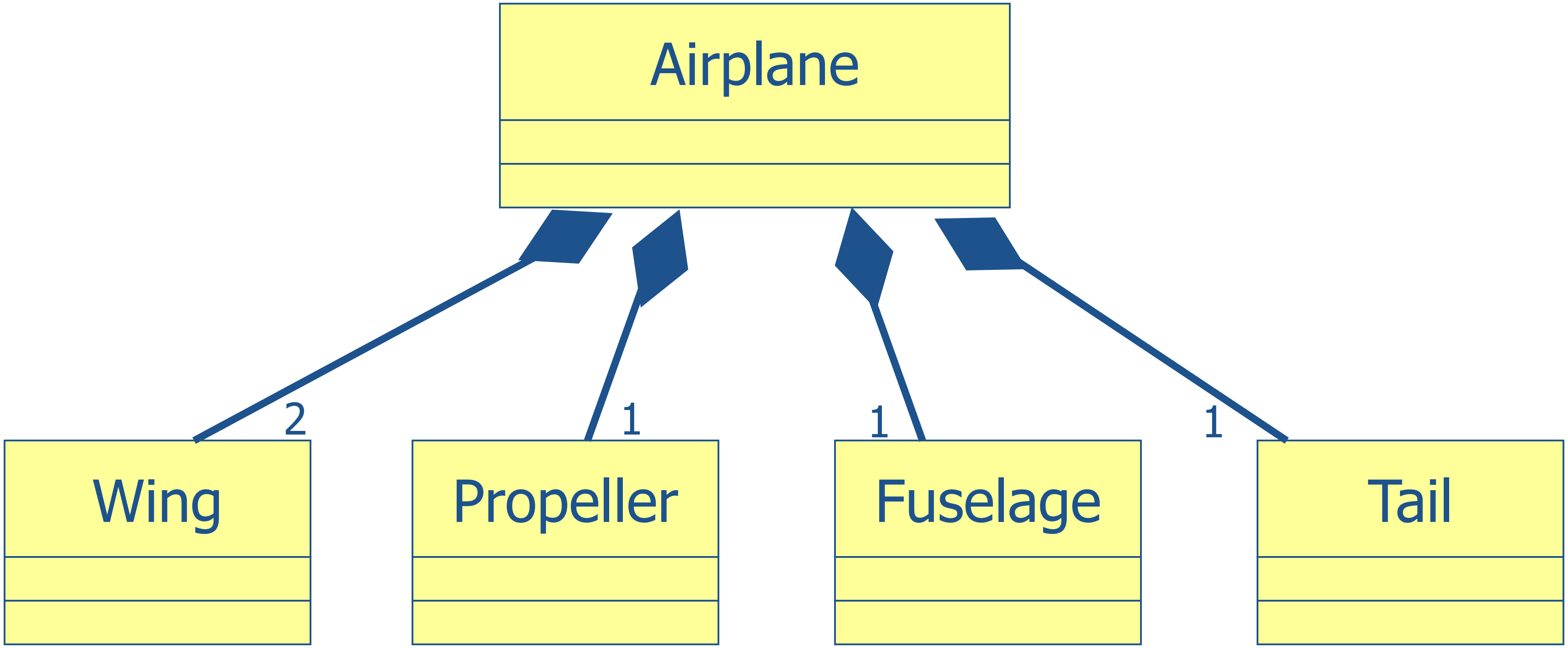
## A strong form of aggregation

- A **composition aggregation** indicates that the existence of the parts depends on the whole
- The whole is the sole owner of its part.
  - The part object may belong to only one whole
- Multiplicity on the whole side must be zero or one.
- The life time of the part is dependent upon the whole.
  - The composite must manage the creation and destruction of its parts.





# Composition



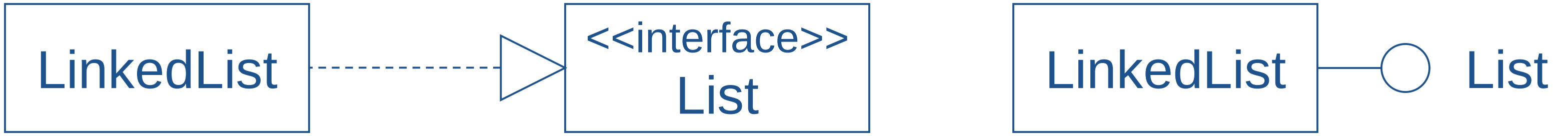
# Relationships

## Types of relationship

- Association (Kết hợp)
- Aggregation (Thu nạp)
- Composition (Hợp thành)
- Dependency (Phụ thuộc)
- Generalization/Inheritance (Kế thừa)
- **Realization (Hiện thực hóa)**

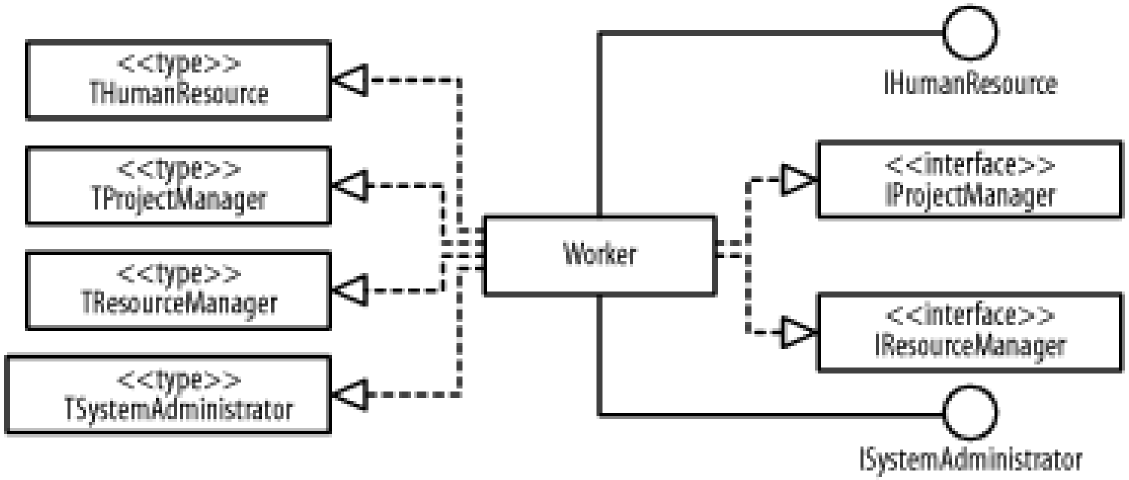
# Realization

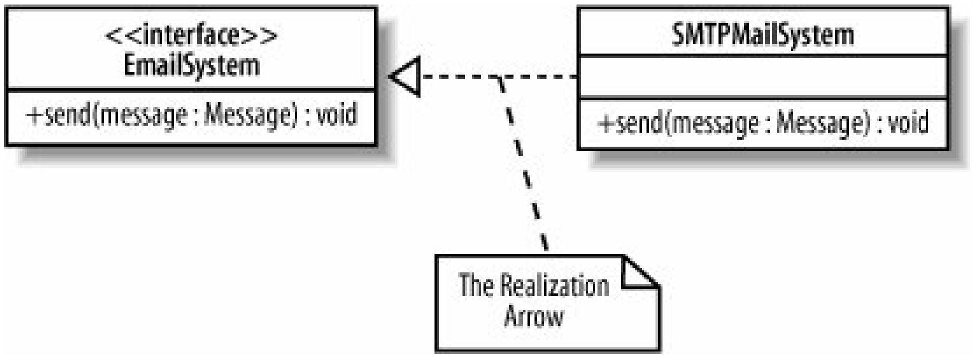
- A realization relationship indicates that one class implements a behavior specified by another class (an interface or protocol).
- An interface can be realized by many classes.
- A class may realize many interfaces



# Realization

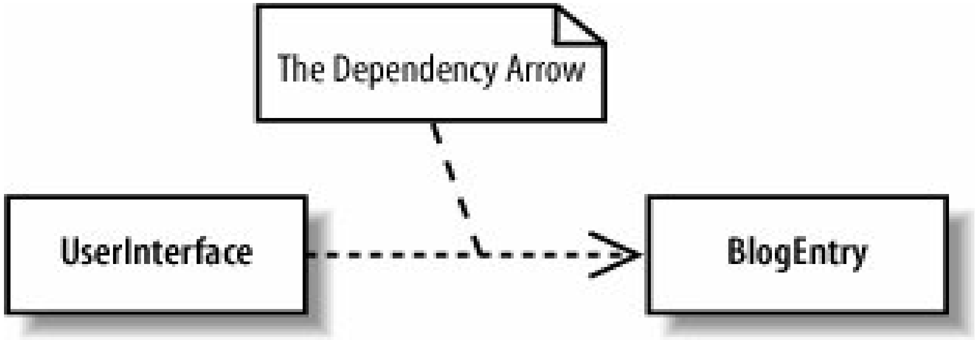
Figure 3-34. Realizations for the Worker class




<<interface>>
EmailSystem
+send(message : Message) : void
SMTPMailSystem
+send(message : Message) : void
The Realization
Arrow

# Dependency

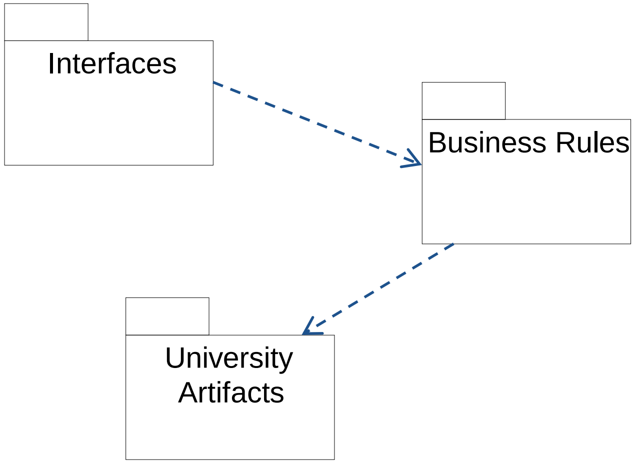
- **A dependency indicates a semantic relation between two or more classes in which a change in one may force changes in the other although there is no explicit association between them.**



# Dependency

## Package Relationships

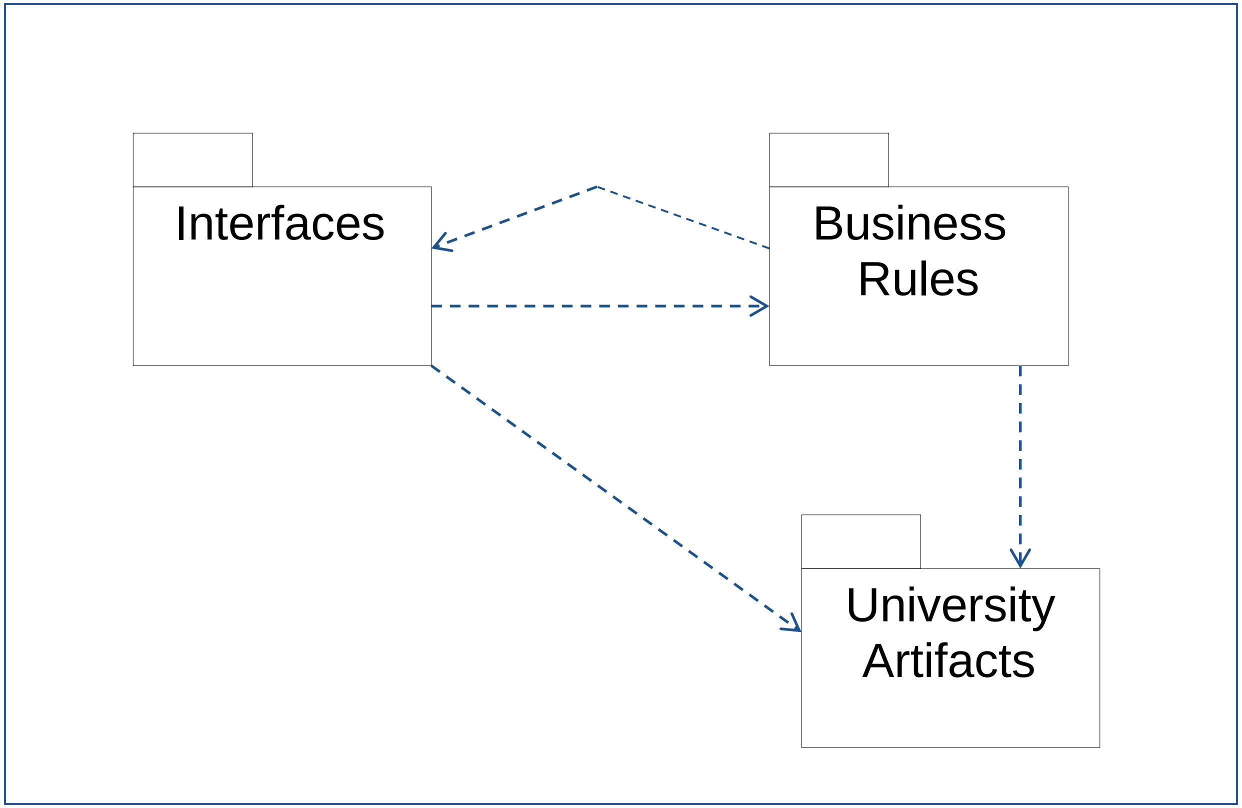
- Packages are related to one another using a dependency relationship
- If a class in one package “talks” to a class in another package then a dependency relationship is added at the package level



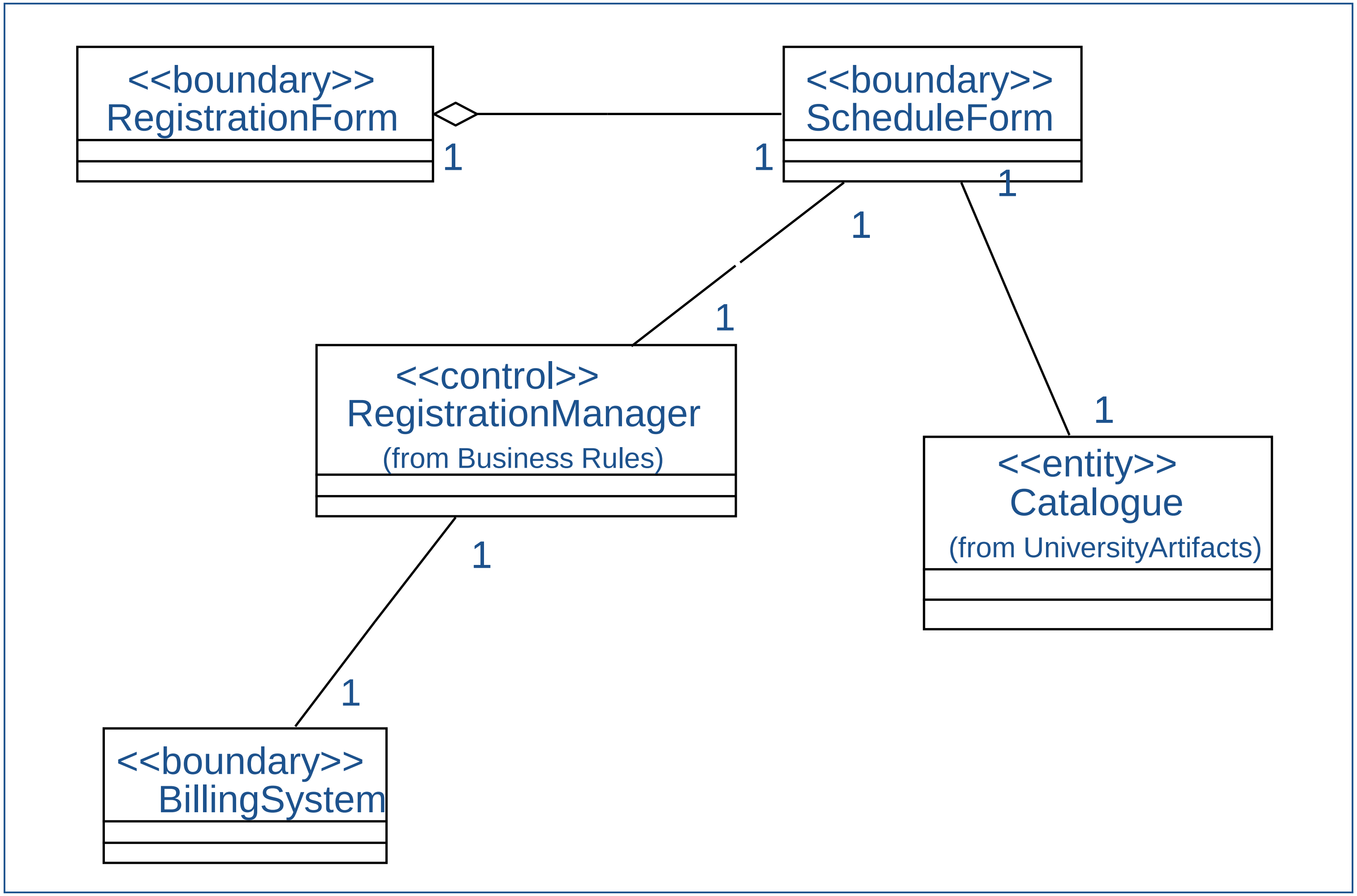
# Relationships During Analysis and Design

- During analysis, establish connections (associations and aggregations) among classes
  - These connections exist because of the nature of the classes, not because of a specific implementation
  - Make an initial estimate of multiplicity in order to expose hidden assumptions
- Class diagrams are updated to show the added relationships
- During design:
  - Multiplicity estimates are refined and updated
  - Associations and aggregations are evaluated and refined
  - Package relationships are re-evaluated and refined
  - Class diagrams are matured

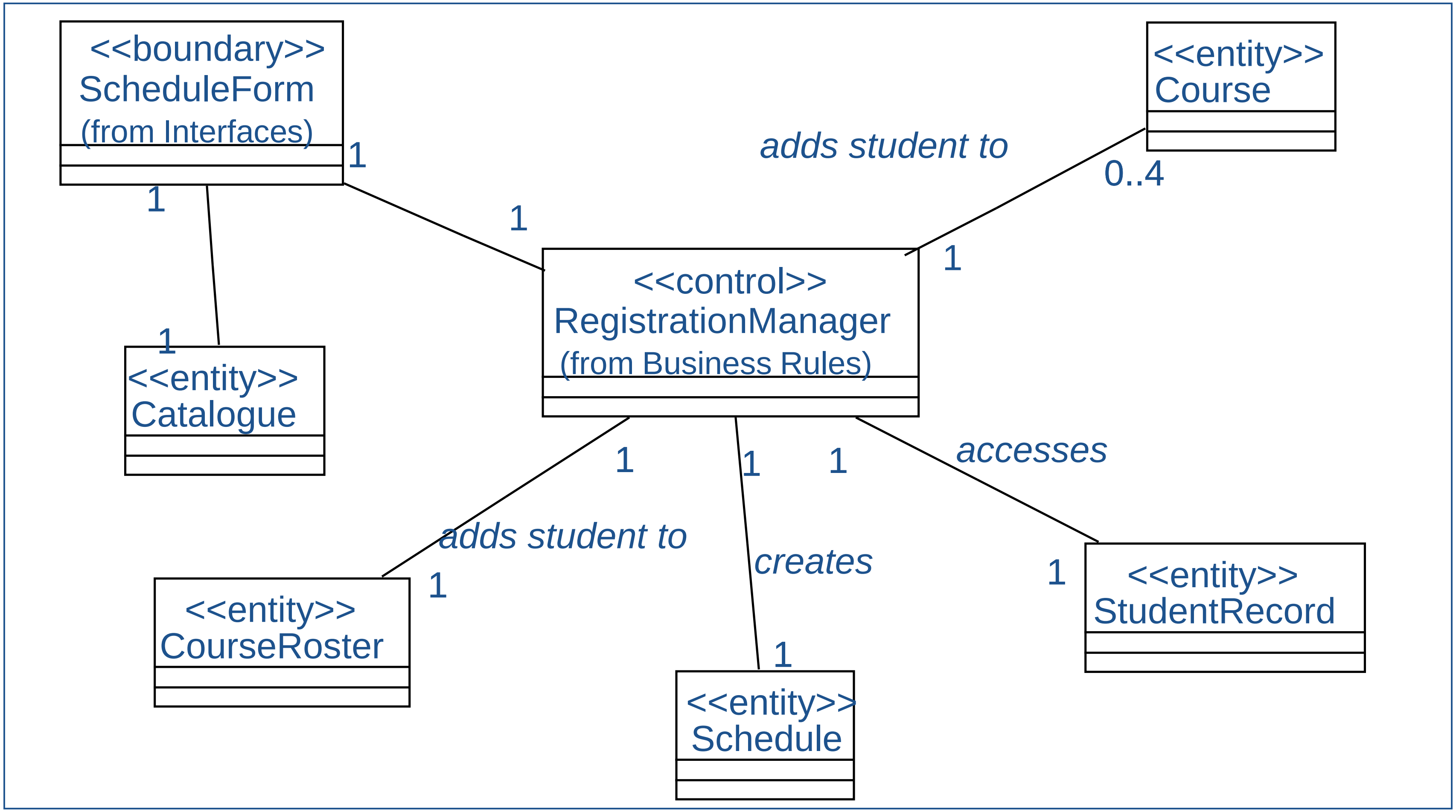
# Updated Main Class Diagram for the Registration System



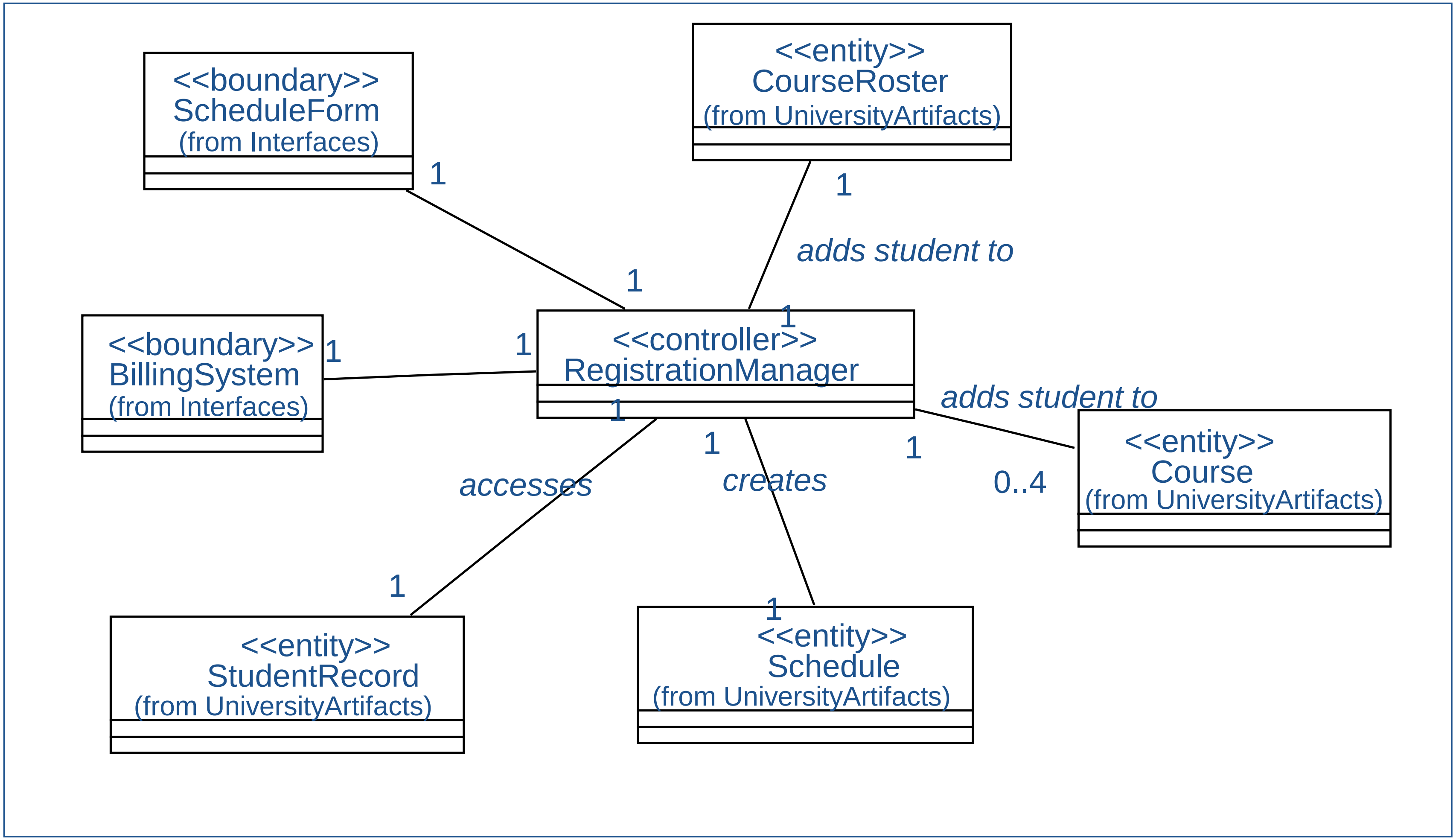
# Updated Interfaces Main Class Diagram



# Updated UniversityArtifacts Main Class Diagram



# Updated Business Rules Main Class Diagram



# Relationships

## Types of relationship

- Association (Kết hợp)
- Aggregation (Thu nạp)
- Composition (Hợp thành)
- Dependency (Phụ thuộc)
- **Generalization/Inheritance (Kế thừa)**
- Realization (Hiện thực hóa)

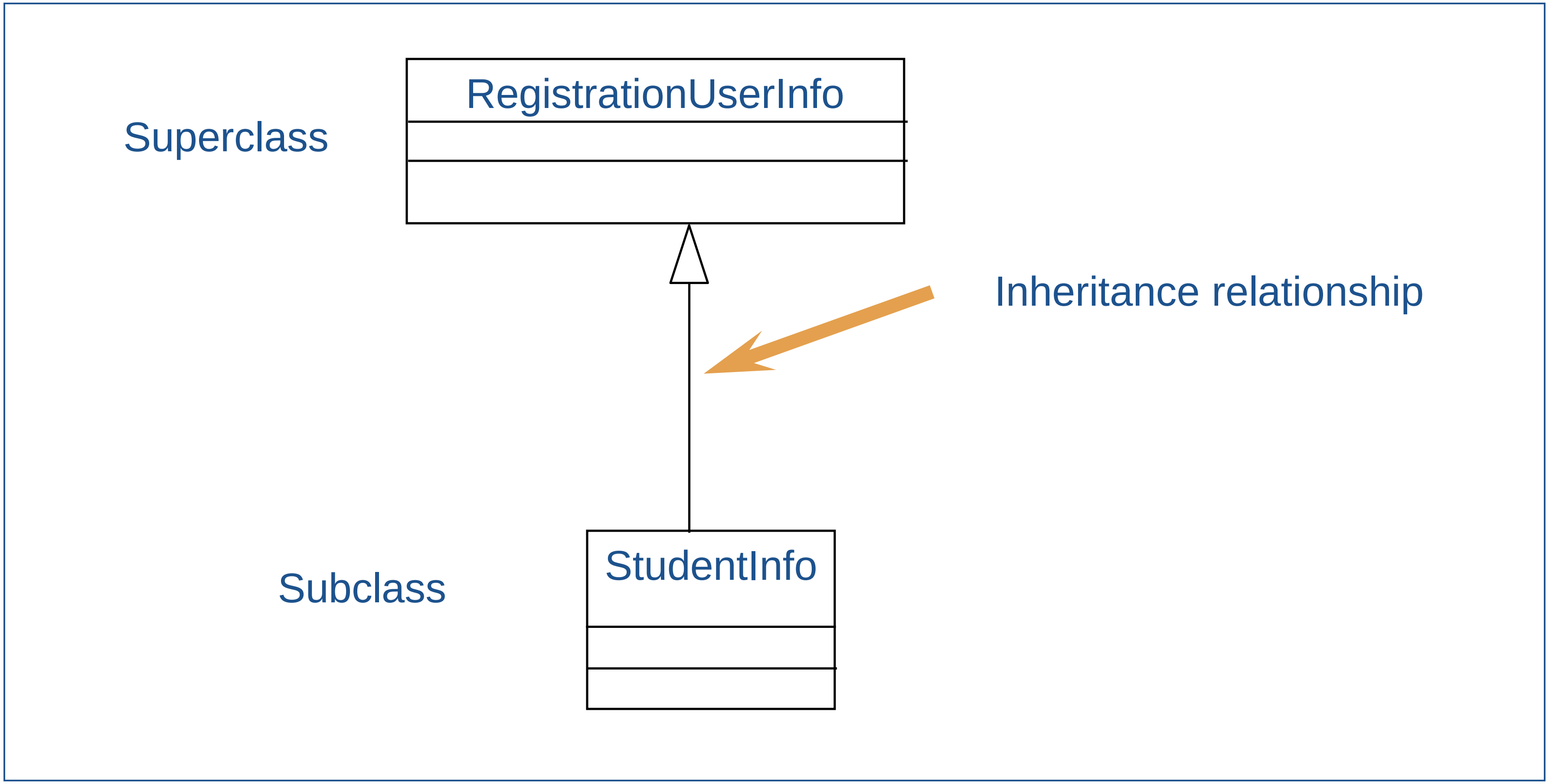
# Bài tập

- **Một nhóm 2 SV**
  - Chọn 1 nghiệp vụ cụ thể, vẽ các lược đồ sau:
    - 1 Usecase diagram (viết scenario): có ít nhất 1 usecase
    - 1 Sequence diagram
    - 1 Class diagrams
    - *Không làm chức năng đăng ký, đăng nhập, đăng xuất*
    - *Viết ra giấy (nộp vào cuối buổi)*

# Inheritance

- Inheritance defines a relationship among classes where one class shares the structure and/or behavior of one or more classes
- Inheritance defines a hierarchy of abstractions in which a subclass inherits from one or more superclasses
  - With single inheritance, the subclass inherits from only one superclass
  - With multiple inheritance, the subclass inherits from more than one superclass
- Inheritance is an “is a” or “kind of” relationship

# Drawing an Inheritance Hierarchy

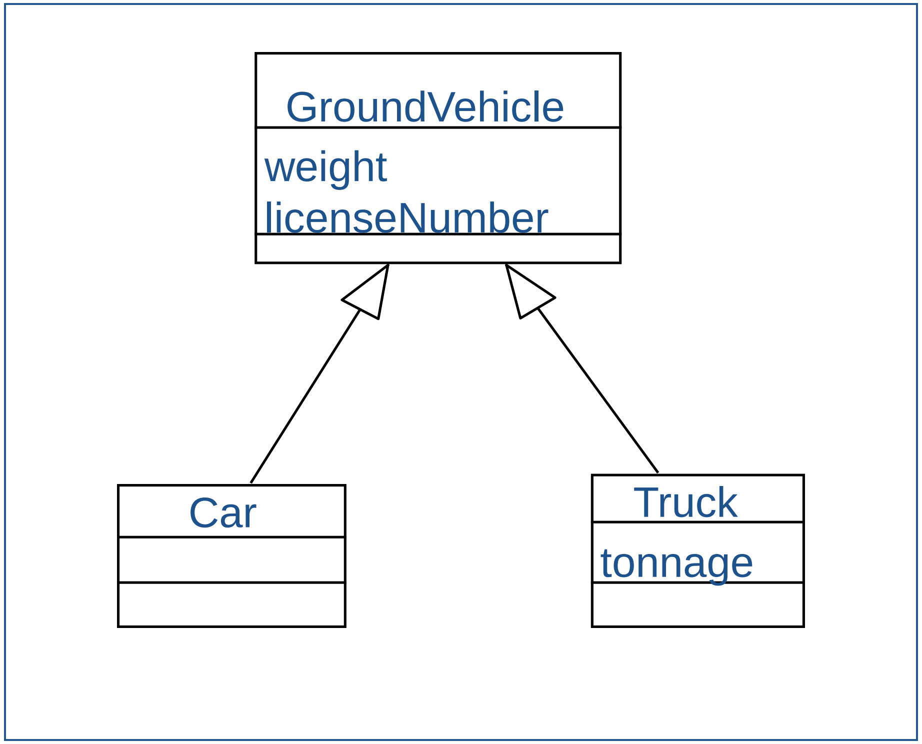


# What Gets Inherited?

- A subclass inherits its parent’s:
  - Attributes
  - Operations
  - Relationships
- A subclass may:
  - Add additional attributes, operations, relationships
  - Redefine inherited operations (use caution!)

# Inheriting Attributes

- Attributes are defined at the highest level in the inheritance hierarchy at which they are applicable
- Subclasses of a class inherit all attributes
- Each subclass may add additional attributes



A truck has three attributes:

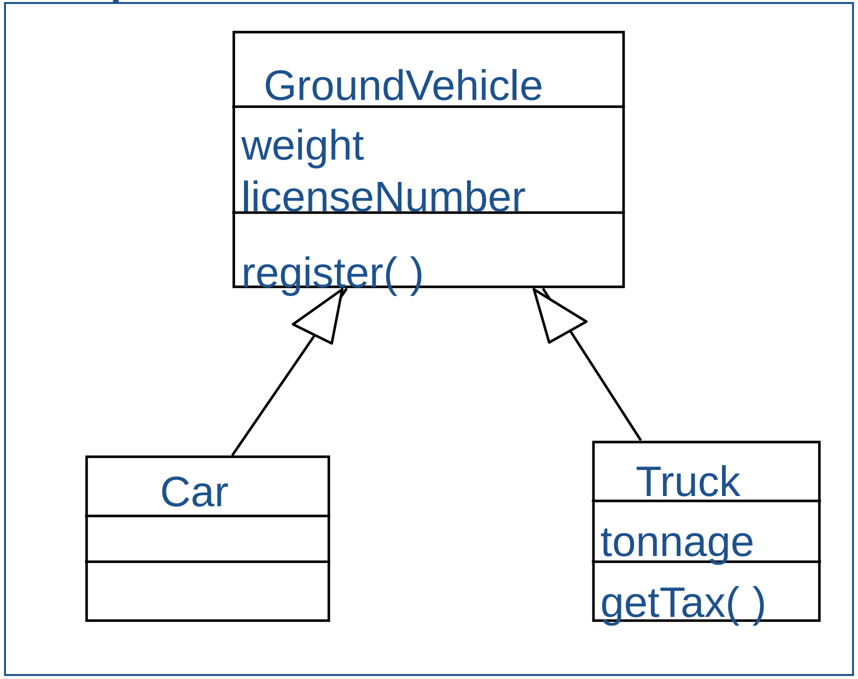
licenseNumber

weight

tonnage

# Inheriting Operations

- Operations are defined at the highest level in the inheritance hierarchy at which they are applicable
- Subclasses of a class inherit all operations
- Each subclass may augment or redefine inherited operations



A truck has three attributes:

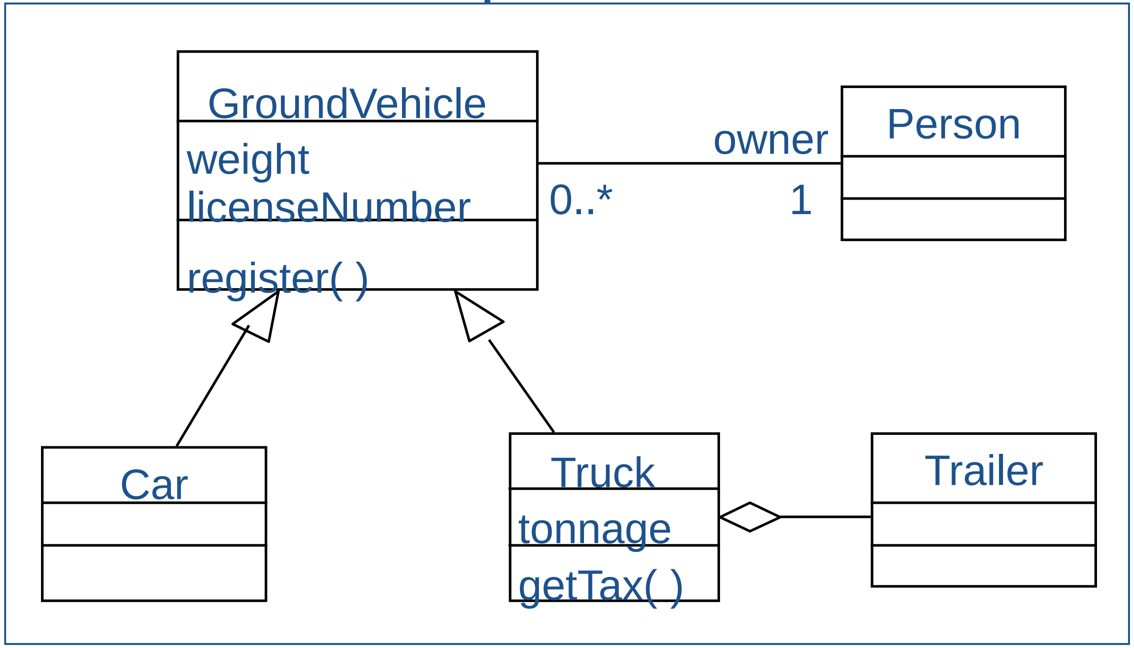
licenseNumber
weight
tonnage

and two operations:

register()
getTax()

# Inheriting Relationships

- Relationships are also inherited and should be defined at the highest level in the inheritance hierarchy at which they are applicable
- Subclasses of a class inherit all relationships
- Each subclass may also participate in additional relationships

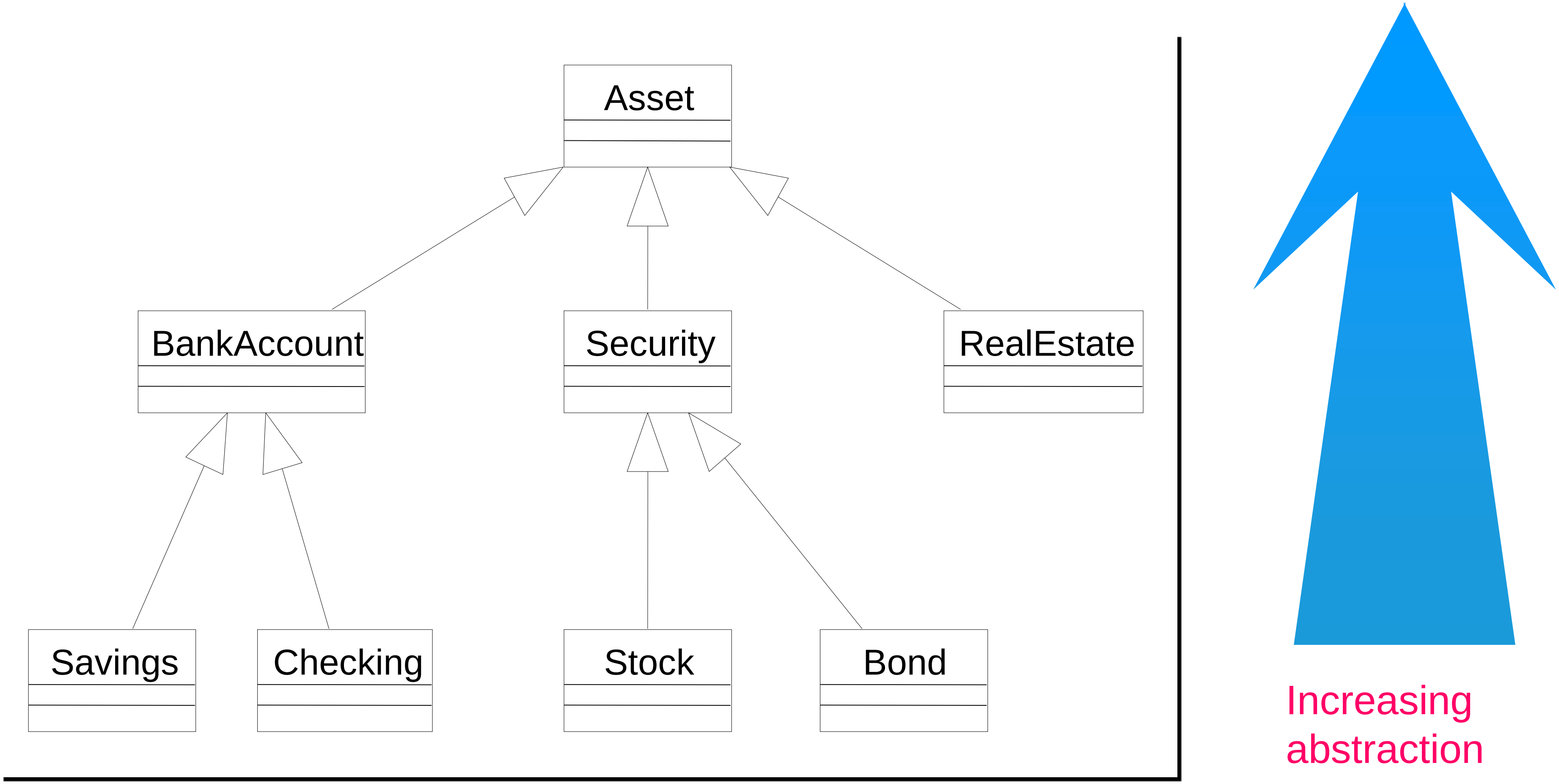


car is related to an owner
A truck is related to an owner
A truck also has a trailer

# Generalization of Classes

- Generalization provides the capability to create superclasses that encapsulate structure and/or behavior common to several subclasses
- Generalization procedure
  - Identifying similarity of structure/behavior among several classes
  - Creating a superclass to encapsulate the common structure/behavior
  - The original classes are subclassed off of the new superclass
- Superclasses are more abstract than their subclasses

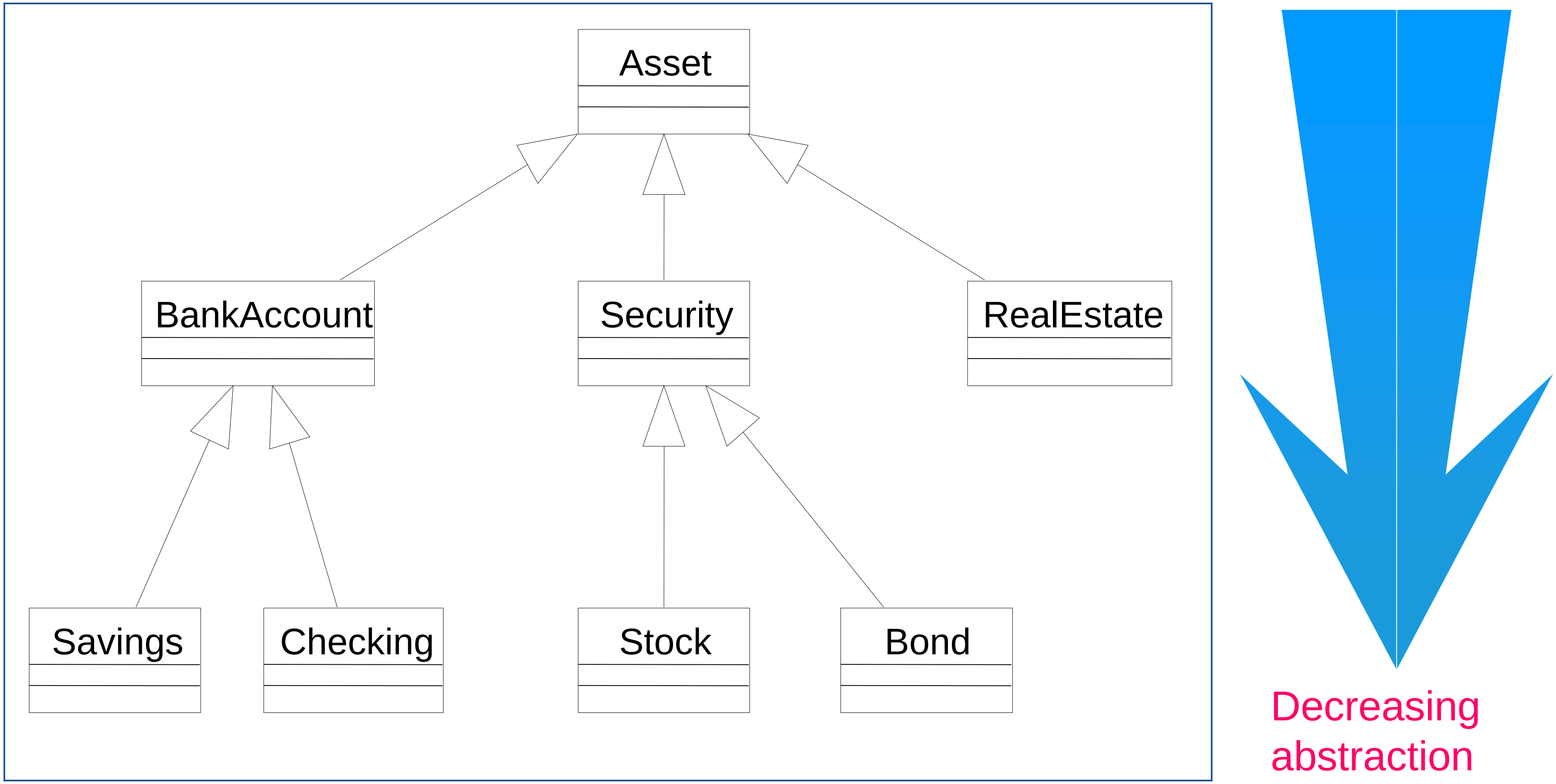
# Example of Generalization



# Specialization of Classes

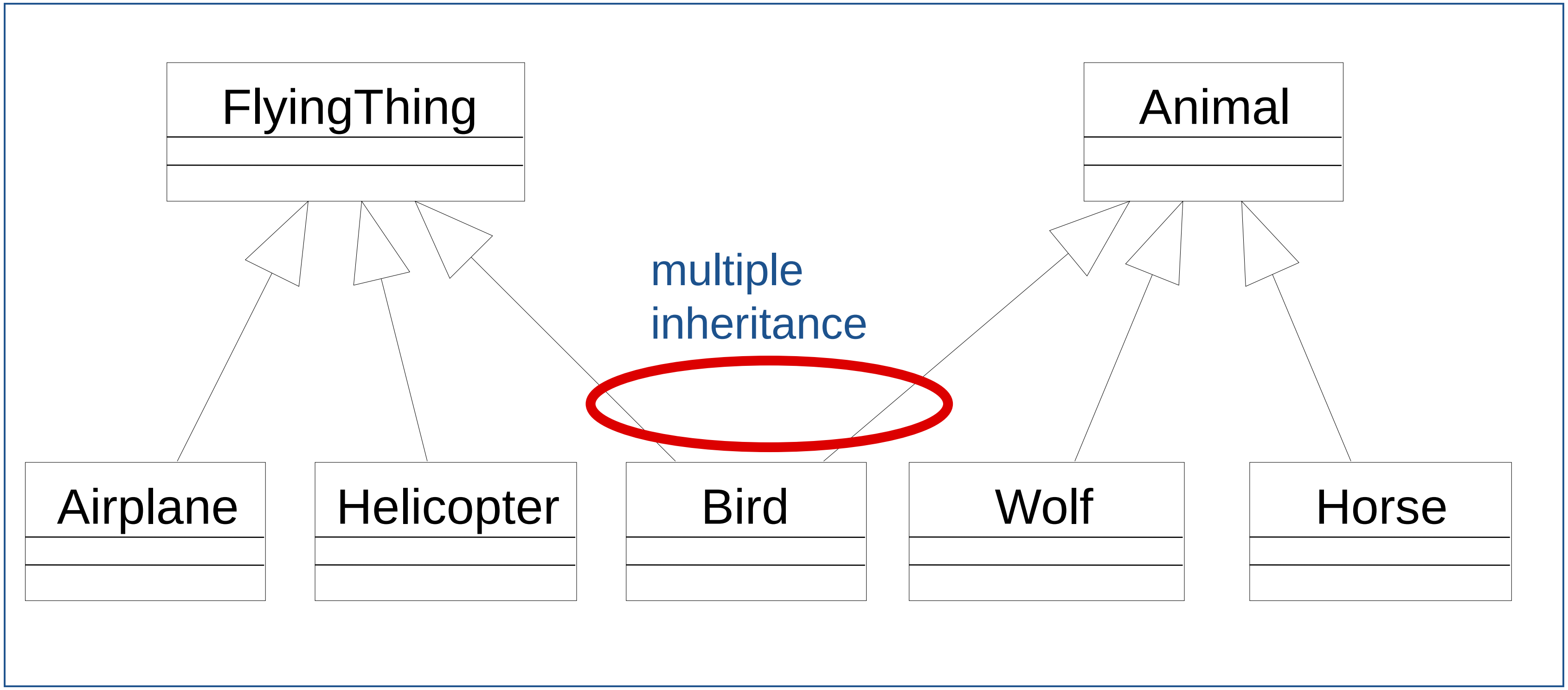
- Specialization provides the capability to create subclasses that represent refinements in which structure and/or behavior from the superclass are added or modified
- Specialization procedure
  - Noticing that some instances exhibit specialized structure or behavior
  - Creating subclasses to group instances according to their specialization
- Subclasses are less abstract than their superclasses

# Example of Specialization



Decreasing abstraction

# Multiple Inheritance



Bird inherits from both FlyingThing and Animal

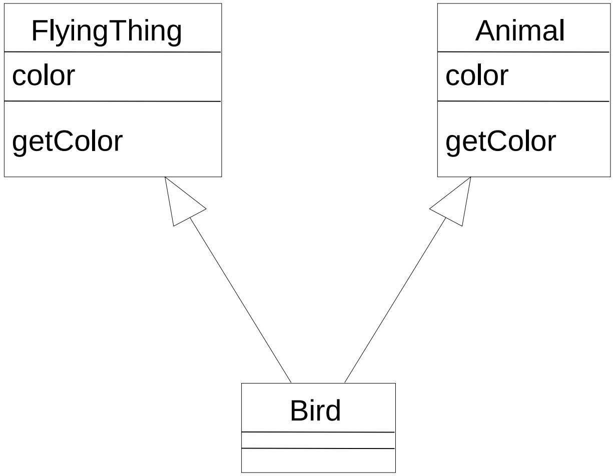
# Multiple Inheritance Concepts

- Conceptually straightforward and necessary for modeling the real world accurately
- In practice, may lead to difficulties in implementation
  - Not all object-oriented programming languages support multiple inheritance directly

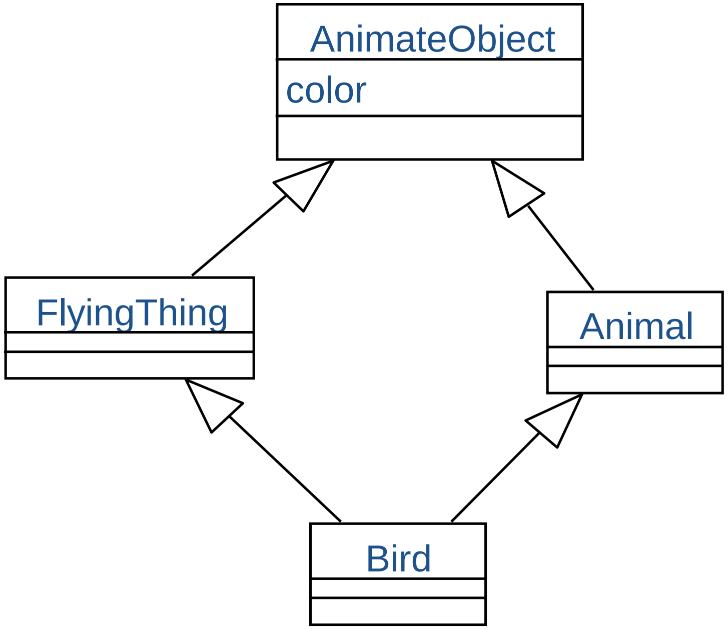
Use multiple inheritance only when needed,
and
always with caution !

# Multiple Inheritance: Problems

Name clashes on attributes or operations



Repeated inheritance



Each programming language/environment chooses ways to resolve these difficulties

# Finding Inheritance

- It is important to evaluate all classes for possible inheritance
  - Look for common behavior (operations) and state (attributes) in classes
- Addition technique
  - Add new operations/attributes to the subclass(es)
- Modification technique
  - Redefine operations
    - Must be careful not to change the semantics

# Inheritance vs. Aggregation

- Inheritance and aggregation are often confused
  - Inheritance represents an “is-a” or “kind-of” relationship
  - Aggregation represents a “part-of” relationship

The keywords “is a” and “part of” will help determine the correct relationship

# Inheritance vs. Aggregation

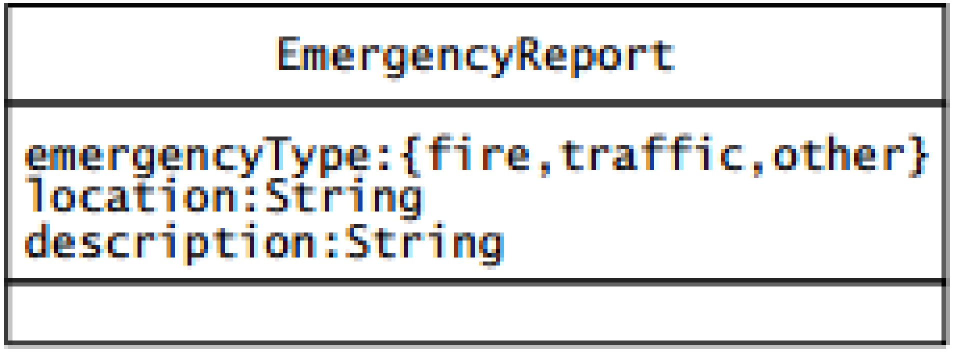
| Inheritance | Aggregation |
|---|---|
| Keywords “is a” | Keywords “has a” |
| One object | Relates objects in different classes |
| Represented by an arrow | Represented by a diamond |

# II. Analysis activities

1. Identifying Objects
2. Mapping Usecase to Objects (with Sequence diagrams)
3. Identifying Class relationship
4. **Identifying Attributes**
5. Modeling State-dependent Behavior of Objects

# 4. Identifying Attributes

- Attributes are properties of individual objects.
- Atribute has:
  - Name
  - type



EmergencyReport

emergencyType:{fire,traffic,other}
location:String
description:String