Use case Analysis

Use-Cases: describing how the user will use the system

- A ***use case*** is a typical sequence of actions that a user performs in order to complete a given task
  - The objective of ***use case analysis*** is to model the system from the point of view of

    … how users interact with this system

    … when trying to achieve their objectives.

    It is one of the key activities in **requirements analysis**
- **First step in use case analysis is to determine the types of users or other systems that will use the facilities of this system. These are called *actors*.**
  - A *use case model* consists of
    - a set of use cases
    - an optional description or diagram indicating how they are related

Use cases

- **The second step in use case analysis is to determine the tasks that each actor will need to do with the system.**
- Each task is a **use case**
- A use case should
  - Cover the *full sequence of steps* from the beginning of a task until the end.
  - Describe the *user's interaction* with the system …
    - Not the computations the system performs.
  - Be written so as to be as *independent* as possible from any particular user interface design.
  - Only include actions in which the actor interacts with the computer.
    - Not actions a user does manually

Use case Example

- *List a minimal set of use cases for the following actors in a library system:*
  - *Borrower, Checkout Clerk, Librarian*

Borrower:

- ❑ Search for items by title.
- ❑ … by author.
- ❑ … by subject.
- ❑ Check the borrower's personal information and list of books currently borrowed.

Checkout Clerk:

- ❑ All the Borrower use cases, plus
- ❑ Check out an item for a borrower.
- ❑ Check in an item that has been returned.

Use case Example....

- ❑ Renew an item.
- ❑ Record that a fine has been paid.
- ❑ Add a new borrower.
- ❑ Update a borrower's personal information (address, telephone number etc.).

Librarian:

- ❑ All of the Borrower and Checkout Clerk use cases, plus
- ❑ Add a new item to the collection.
- ❑ Delete an item from the collection.
- ❑ Change the information the system has recorded about an item.

How to describe a single use case

- A**. Name**: Give a short, descriptive name to the use case.
- B. **Actors**: List the actors who can perform this use case.
- C. **Goals**: Explain what the actor or actors are trying to achieve.
- D. **Preconditions**: State of the system before the use case.
- E. **Summary**: Give a short informal description.
- F. **Related use cases**.
- G. **Steps**: Describe each step using a 2-column format. The left column showing the *actions taken by the actor*, and the right column showing *the system's responses*.
- H. **Post conditions**: State of the system in following completion.

- ***A and G are the most important***

Use Case Description: Example

Use case: Check out an item for a borrower

Actors: Checkout clerk (regularly), chief librarian (occasionally)

Goals: To help the borrower to borrow the item if they are allowed, and to ensure a proper record is entered of the loan.

Preconditions: The borrower must have a valid card and not owe any fines. The item must have a valid barcode and not be from the reference section.

Steps:

Actor actions	*System responses*
1. Scan item's barcode and barcode of the borrower's card.	2. Display confirmation that the loan is allowed.
3. Stamp item with the due date.	
4. Confirm that the loan is to be initiated.	5. Display confirmation that the loan has been recorded.

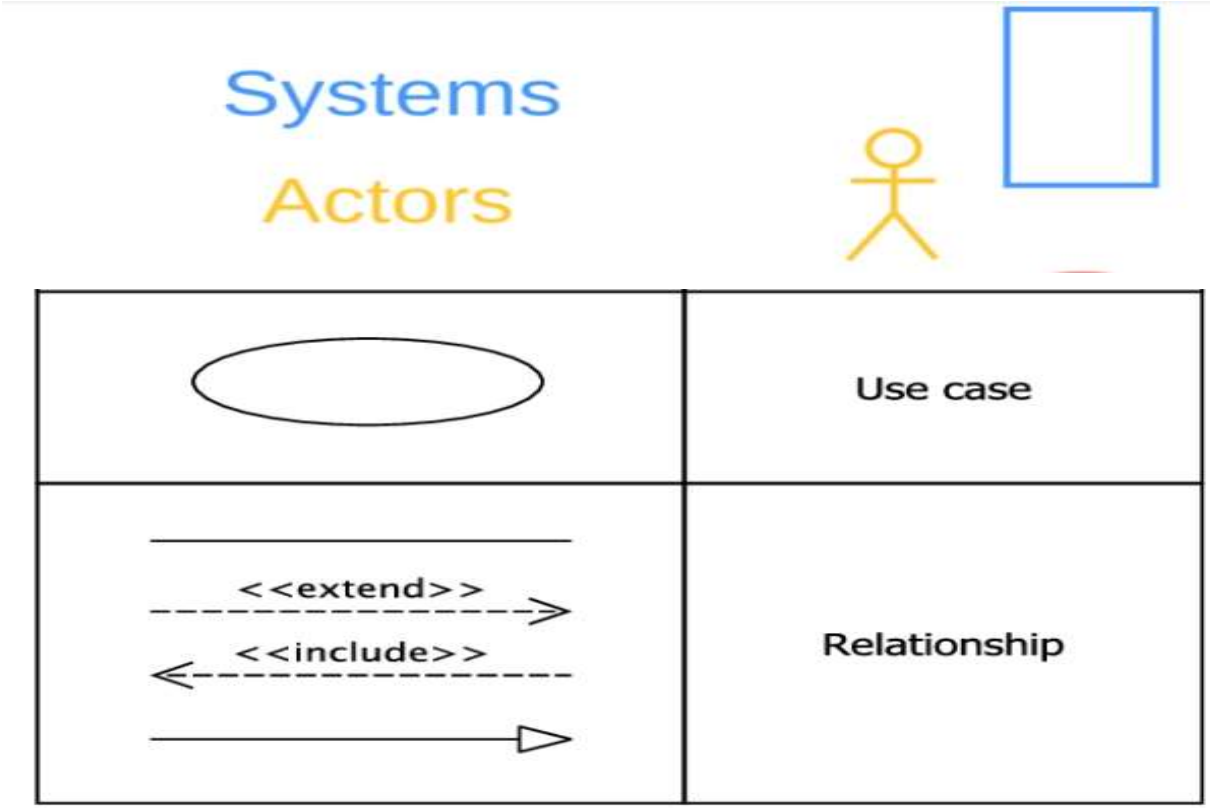
Postconditions: The system has a record of the fact that the item is borrowed, and the date it is due.

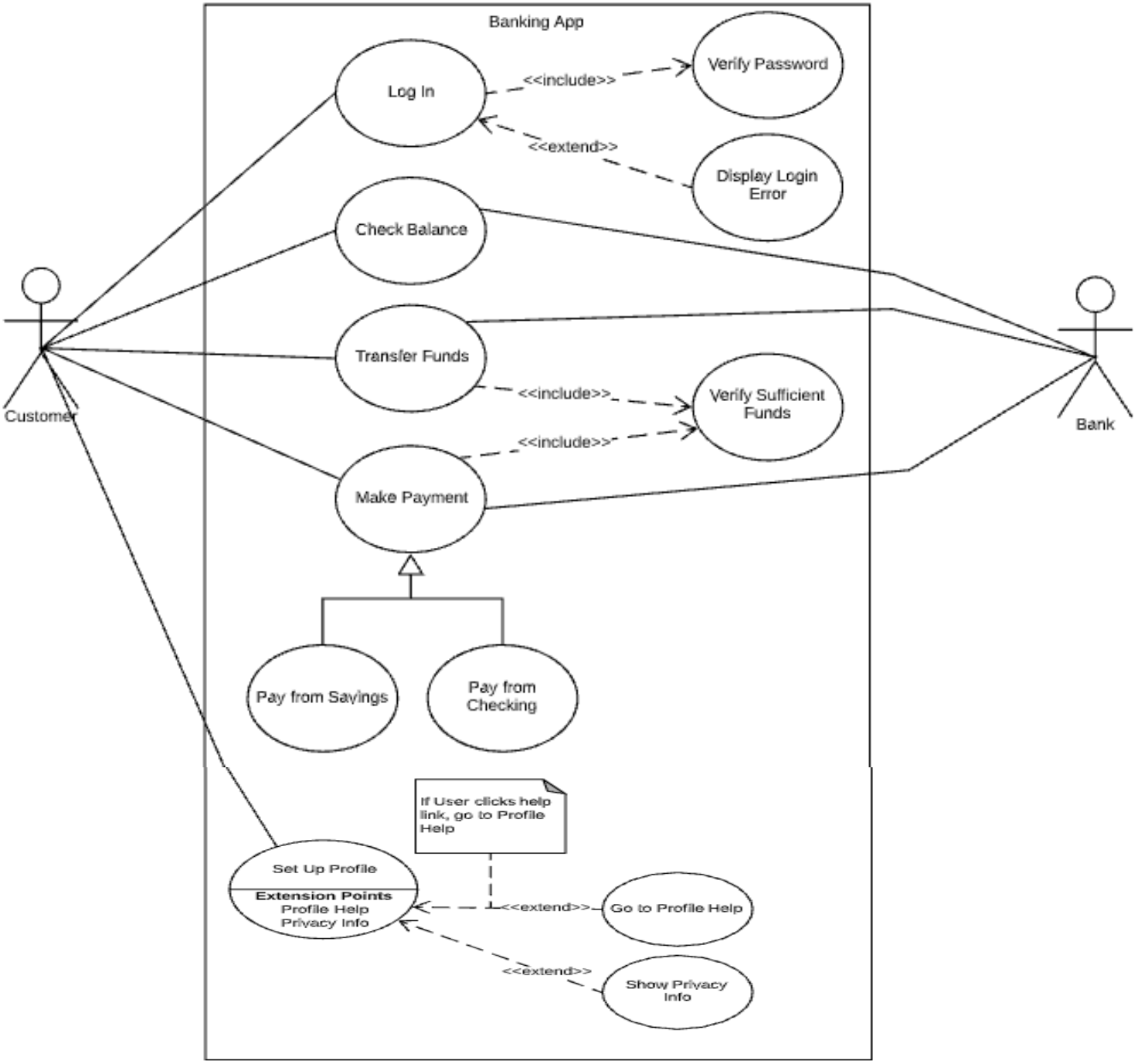
Use case diagrams

- Use case diagrams are UML's notation for showing the relationships among a set of use cases and actors.
- They help a software engineer to convey a high-level picture of the functionality of a system.
- there are two main symbols in use case diagrams:
  - *An actor is shown as a stick person*
  - *a use case is shown as an ellipse.*
- Lines indicate which actors perform which use cases.

Use case diagrams .....

Symbols used in Use case Diagram

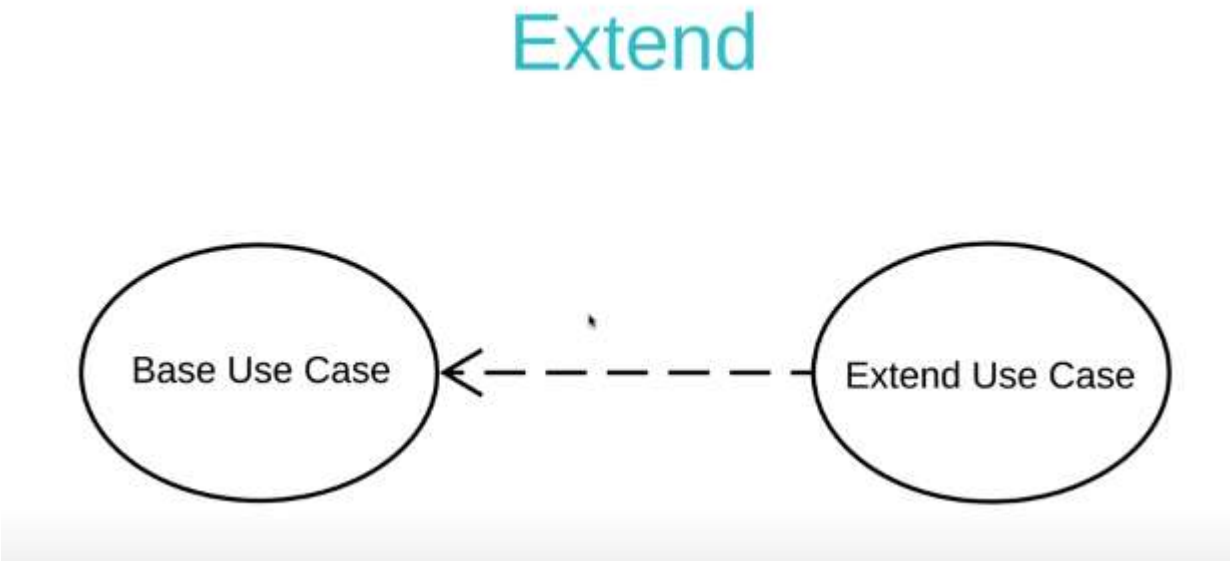



Banking App
Log In
Verify Password
<<include>>
<<extend>>
Display Login Error
Check Balance
Transfer Funds
<<include>>
Verify Sufficient Funds
<<include>>
Make Payment
Customer
Bank
Pay from Savings
Pay from Checking
If User clicks help link, go to Profile Help
Set Up Profile
Extension Points
Profile Help
Privacy Info
<<extend>>
Go to Profile Help
<<extend>>
Show Privacy Info

Use case diagrams....

Extensions

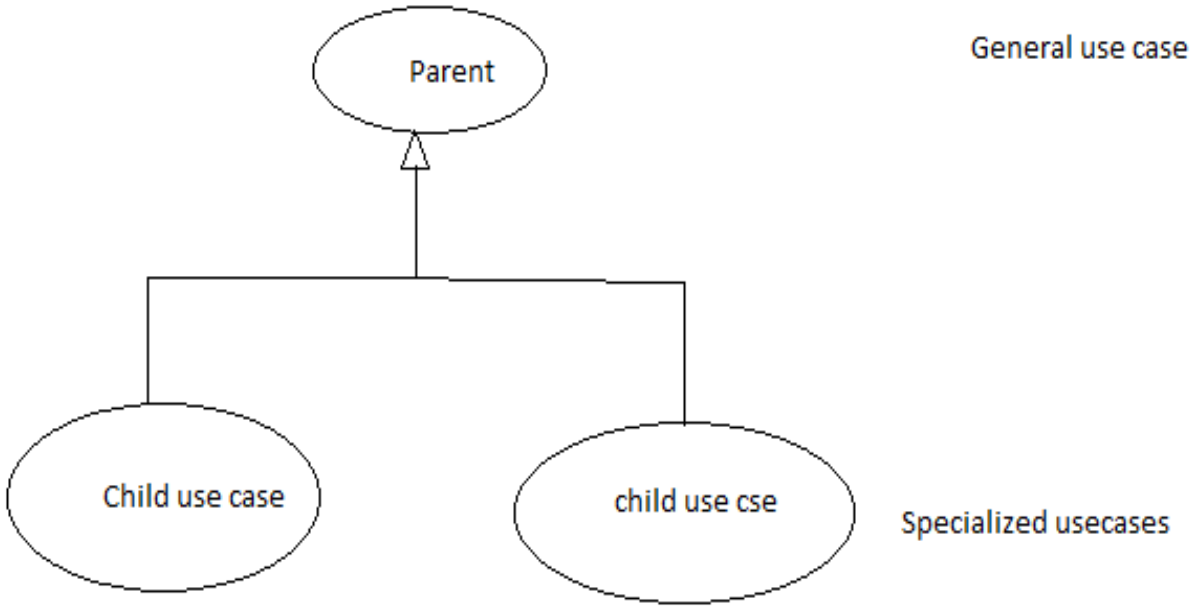
- Used to make *optional* interactions explicit or to handle *exceptional* cases.
- Keep the description of the basic use case simple.



Use case diagrams....

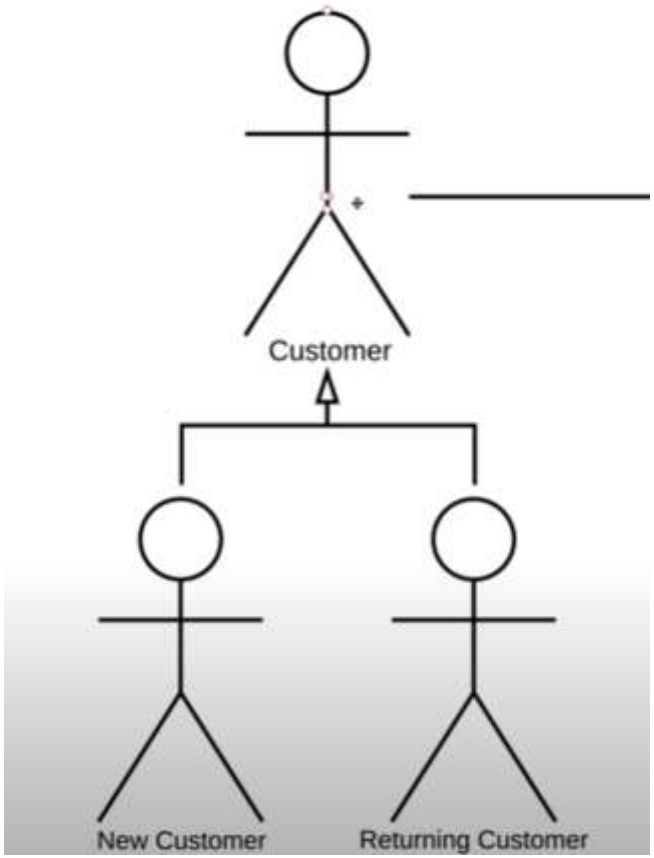
Generalizations

- **Also called inheritance**
- A generalized use case represents *several similar* use cases.
- One or more specializations provides details of the similar use cases.



Use case diagrams....

- Generalization of actors



Use case diagrams....

Inclusions

- Show the relationship between base use case and included use case.
- Every time the base use case is executed ,the included use case is executed well.
- Allow one to express *commonality* between several different use cases.

- Are included in other use cases
  - Even very different use cases can share sequence of actions.
  - Enable you to avoid repeating details in multiple use cases.

Include

The benefits of basing software development on use cases

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

Use cases must not be seen as a panacea

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

UML Diagrams

What is UML?

- At the end of the 1980s and the beginning of 1990s, the first object-oriented development processes appeared
- The proliferation of methods and notations tended to cause considerable confusion
- The **Unified Modelling Language is a standard graphical language** for modelling object oriented software
- UML was developed as a collaborative effort by James Rumbaugh, Grady Booch and Ivar Jacobson, each of whom had developed their own notation
- In 1997 the Object Management Group (OMG) started the process of UML standardization

UML diagrams

- Class diagrams
  - describe classes and their relationships
- Interaction diagrams
  - show the behaviour of systems in terms of how objects interact with each other
- State diagrams and activity diagrams
  - show how systems behave internally
- Component and deployment diagrams
  - show how the various components of systems are arranged logically and physically

UML features

- It has detailed *semantics*
- It has *extension* mechanisms
- It has an associated textual language
  - *Object Constraint Language* (OCL)

- The objective of UML is to assist in software development
  - It is not a *methodology*

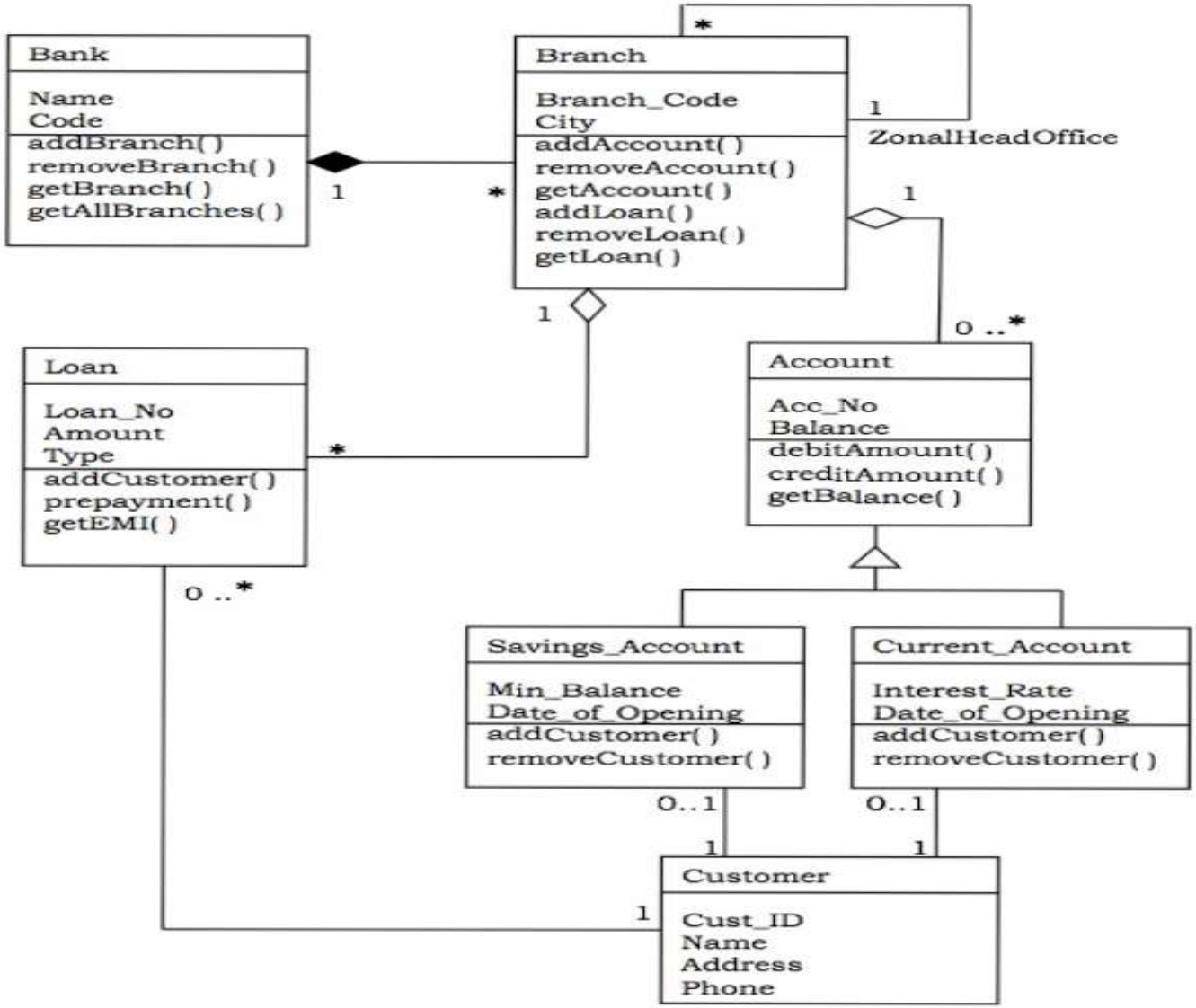
UML Class Diagram

- A type **of static structure diagram**
- In the design of a system**, a number of classes are identified and grouped together that helps to determine the static relations** between them.

Essentials of UML Class Diagrams

- *The main symbols shown on class diagrams are:*
  - *Classes*
    - represent the types of data themselves
  - *Associations*
    - represent linkages between instances of classes
  - *Attributes*
    - are simple data found in classes and their instances
  - *Operations*
    - represent the functions performed by the classes and their instances
  - *Generalizations*
    - group classes into inheritance hierarchies

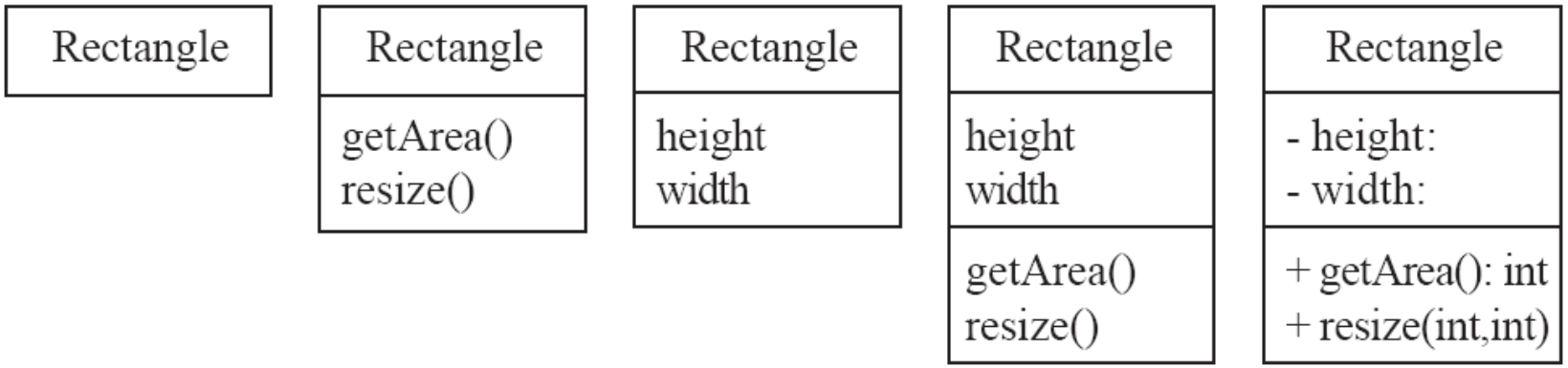
An Example


Bank
Name
Code
addBranch( )
removeBranch( )
getBranch( )
getAllBranches( )
Branch
Branch_Code
City
addAccount( )
removeAccount( )
getAccount( )
addLoan( )
removeLoan( )
getLoan( )
ZonalHeadOffice
Loan
Loan_No
Amount
Type
addCustomer( )
prepayment( )
getEMI( )
Account
Acc_No
Balance
debitAmount( )
creditAmount( )
getBalance( )
Savings_Account
Min_Balance
Date_of_Opening
addCustomer( )
removeCustomer( )
Current_Account
Interest_Rate
Date_of_Opening
addCustomer( )
removeCustomer( )
Customer
Cust_ID
Name
Address
Phone
1
*
0 ..*
0..1

Classes

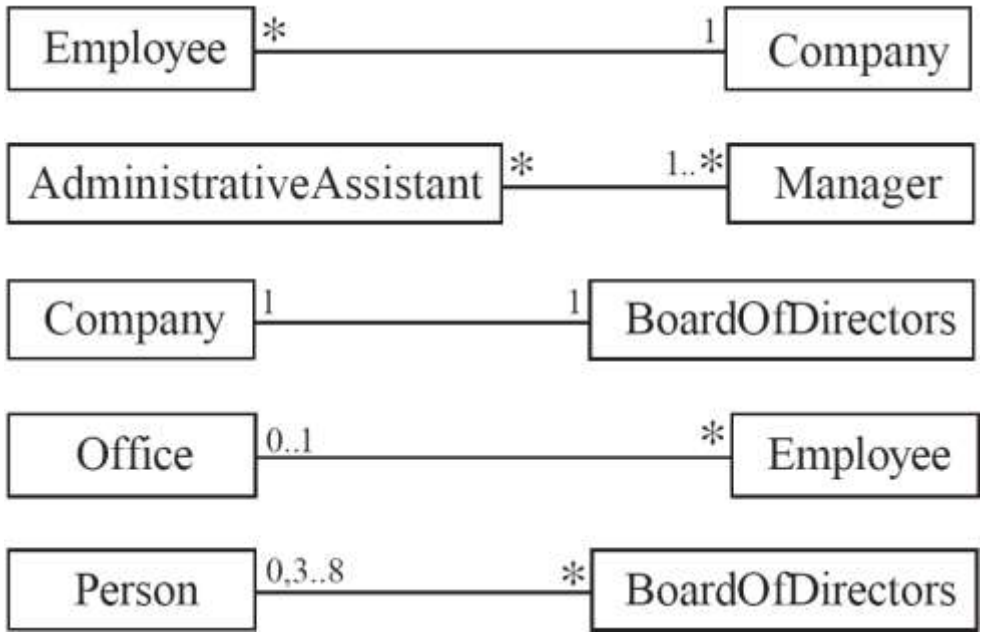
- A class is simply represented as a box with the name of the class inside
  - The diagram may also show the attributes and operations
  - The complete signature of an operation is:

    operationName(parameterName: parameterType ...): returnType



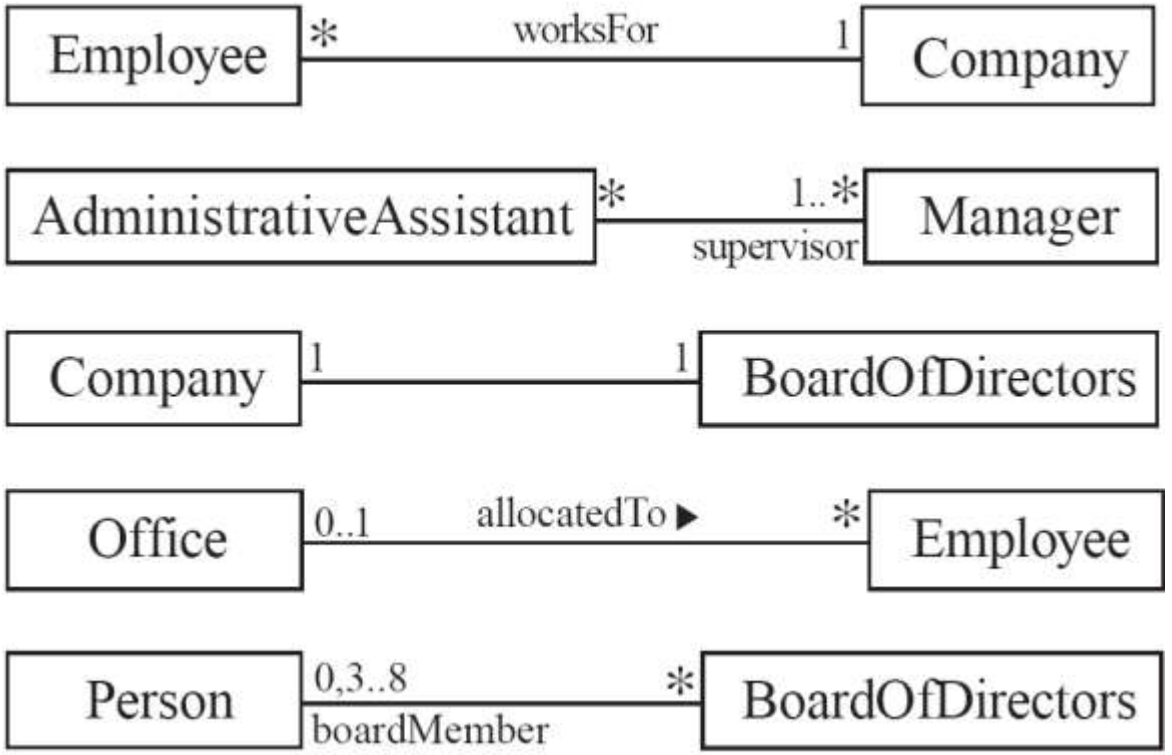
Associations and Multiplicity

- An *association* is used to show how two classes are related to each other
  - Symbols indicating *multiplicity* are shown at each end of the association



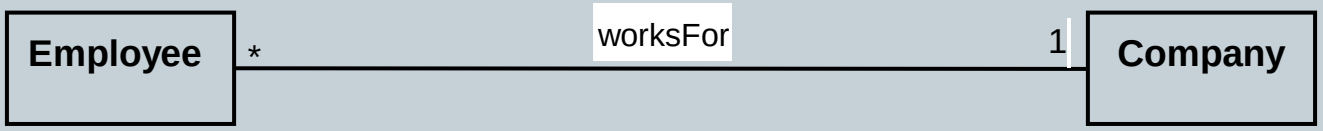
Labelling associations

- Each association can be labelled, to make explicit the nature of the association



Analyzing and validating associations

- **Many-to-one**
  - A company has many employees,
  - An employee can only work for one company.
  - A company can have zero employees
    - E.g. a ‘shell’ company
  - It is not possible to be an employee unless you work for a company



Analyzing and validating associations

- **Many-to-many**
  - An assistant can work for many managers
  - A manager can have many assistants
  - Assistants can work in pools
  - Managers can have a group of assistants
  - Some managers might have zero assistants.
  - Is it possible for an assistant to have, perhaps temporarily, zero managers?

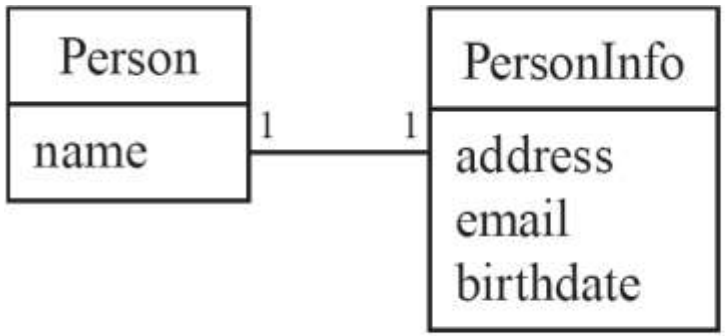
Analyzing and validating associations

- **One-to-one**
  - For each company, there is exactly one board of directors
  - A board is the board of only one company
  - A company must always have a board
  - A board must always be of some company

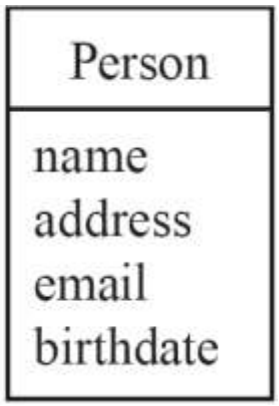
Analyzing and validating associations

- Avoid unnecessary one-to-one associations

Avoid this

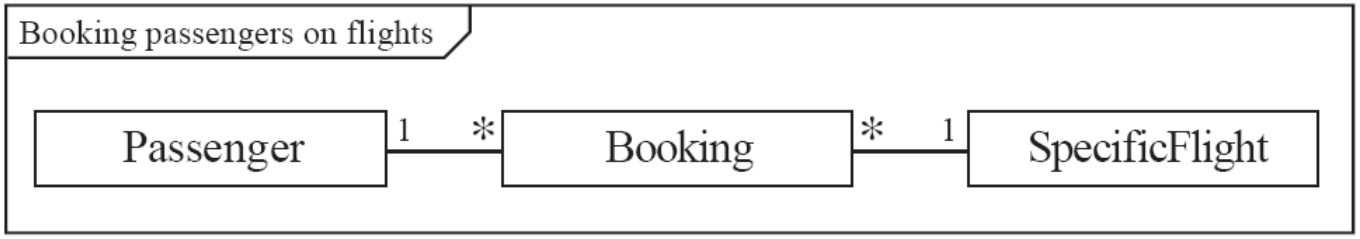


do this



A more complex example

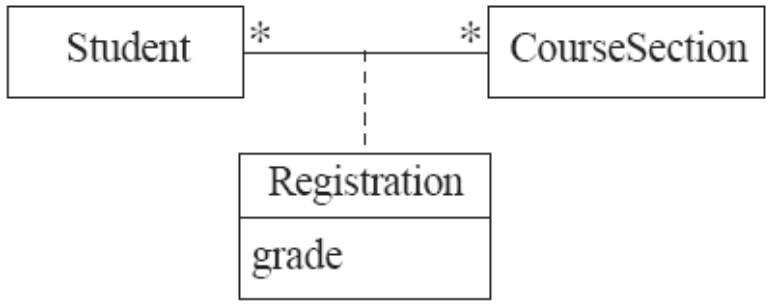
- A booking is always for exactly one passenger
  - no booking with zero passengers
  - a booking could *never* involve more than one passenger.
- A Passenger can have any number of Bookings
  - a passenger could have no bookings at all
  - a passenger could have more than one booking

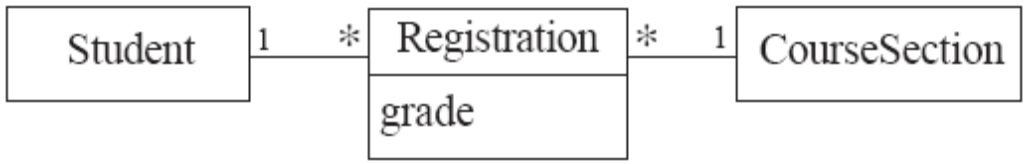


- The *frame* around this diagram is an optional feature that any UML 2.0 may possess.

Association classes

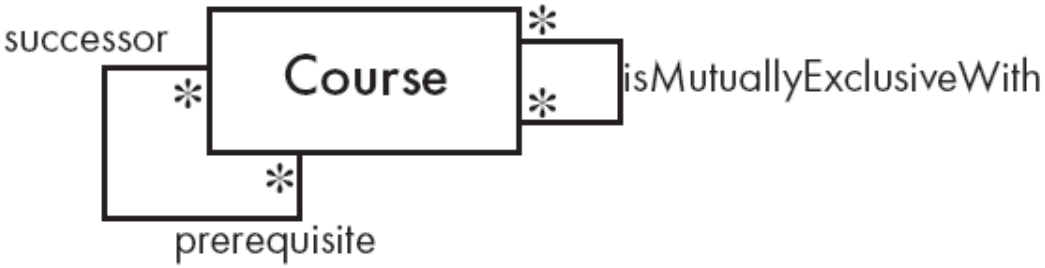
- Sometimes, an attribute that concerns two associated classes cannot be placed in either of the classes
- The following are equivalent





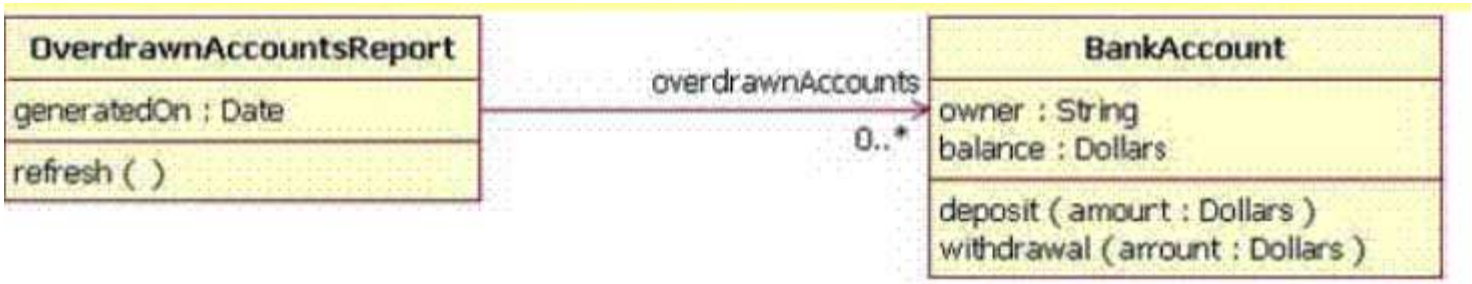
Reflexive associations

- It is possible for an association to connect a class to itself



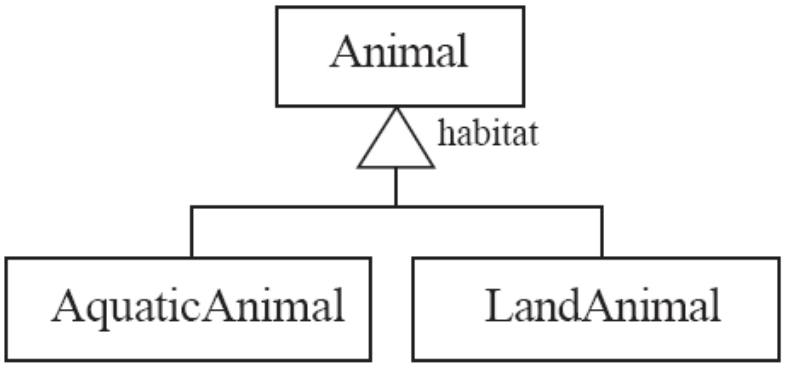
Directionality in associations

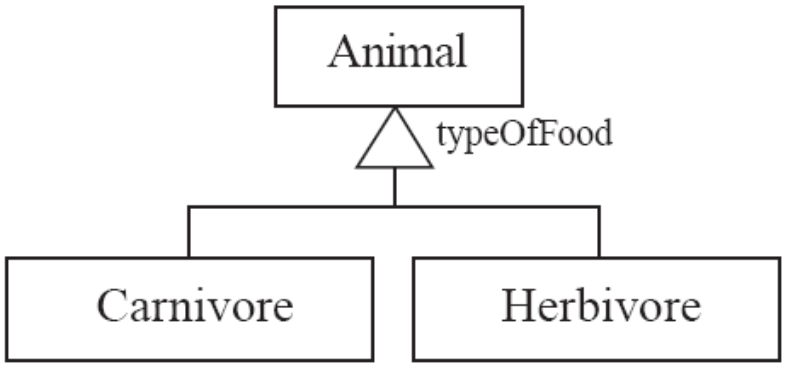
- Associations are by default *bi-directional*
- It is possible to limit the direction of an association by adding an arrow at one end



Generalization

- Specializing a superclass into two or more subclasses
  - A *generalization set* is a labeled group of generalizations with a common superclass
  - The label (sometimes called the *discriminator*) describes the criteria used in the specialization

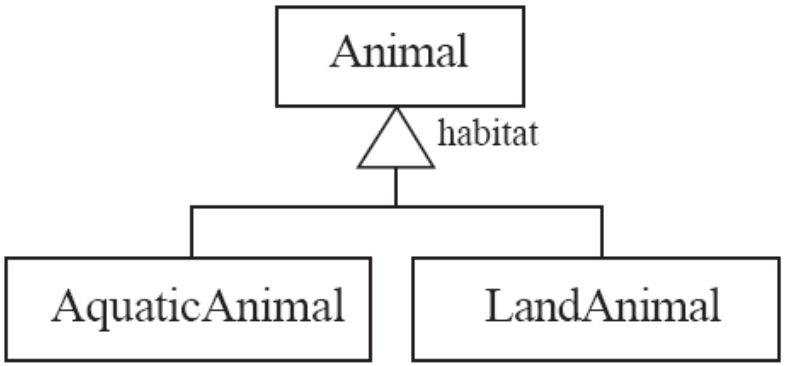


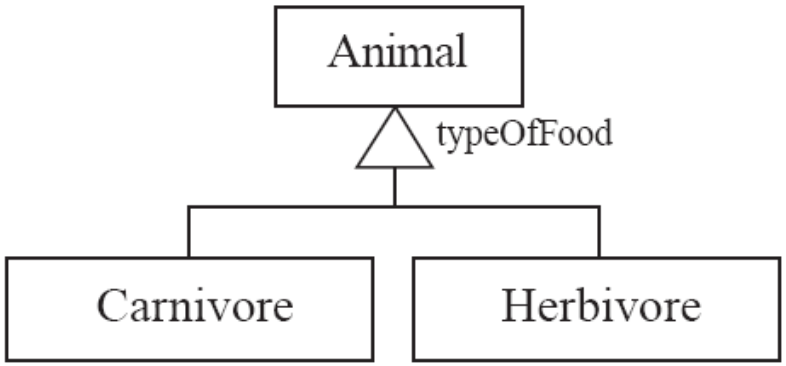


UML – Class Diagram

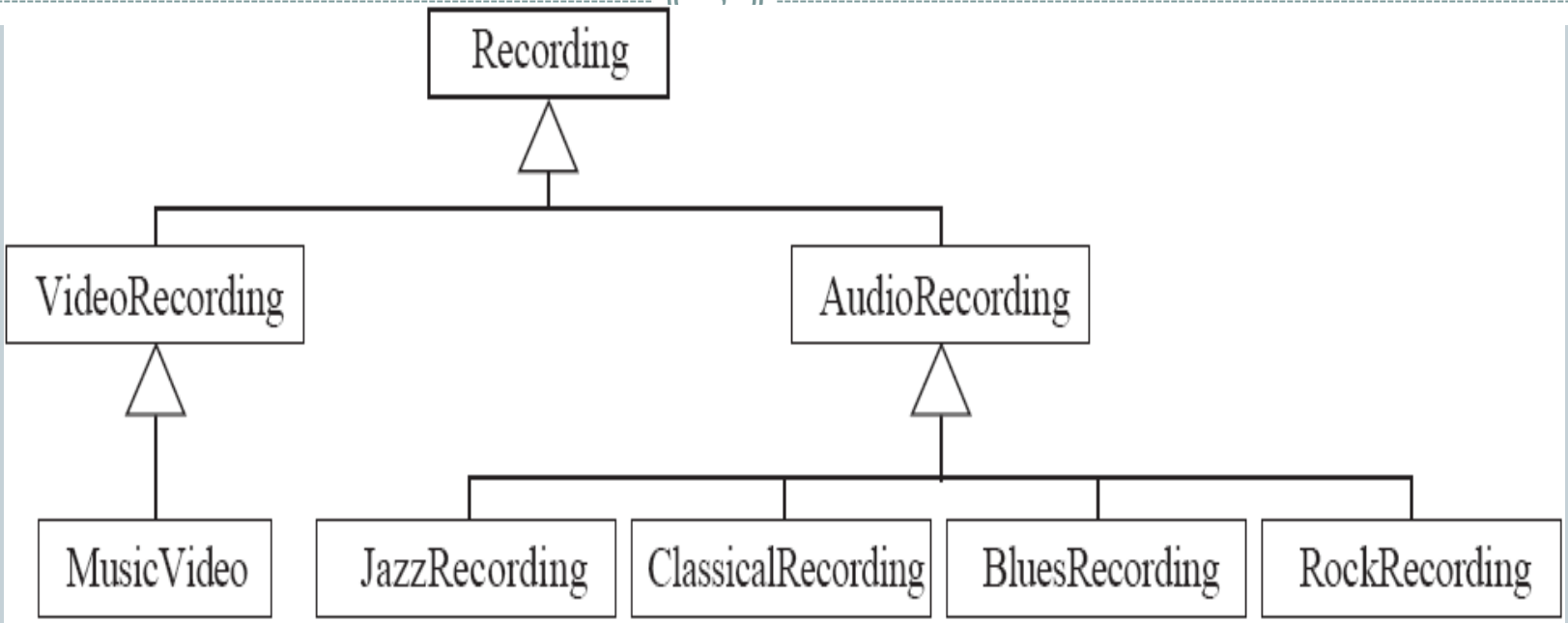
Generalization

- Specializing a superclass into two or more subclasses
  - A *generalization set* is a labeled group of generalizations with a common superclass
  - The label (sometimes called the *discriminator*) describes the criteria used in the specialization



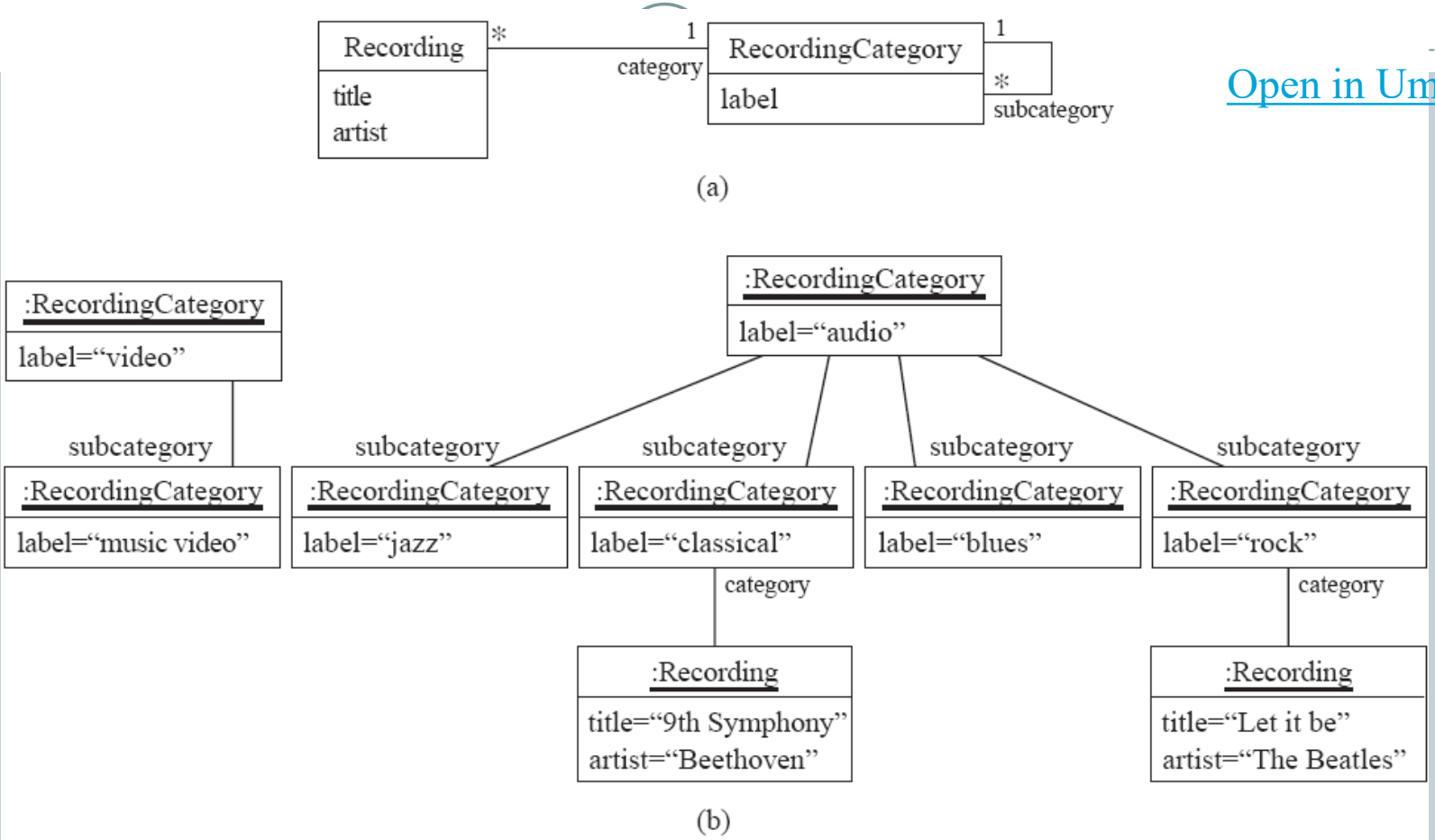


Avoiding unnecessary generalizations



Inappropriate hierarchy of classes, which should be instances

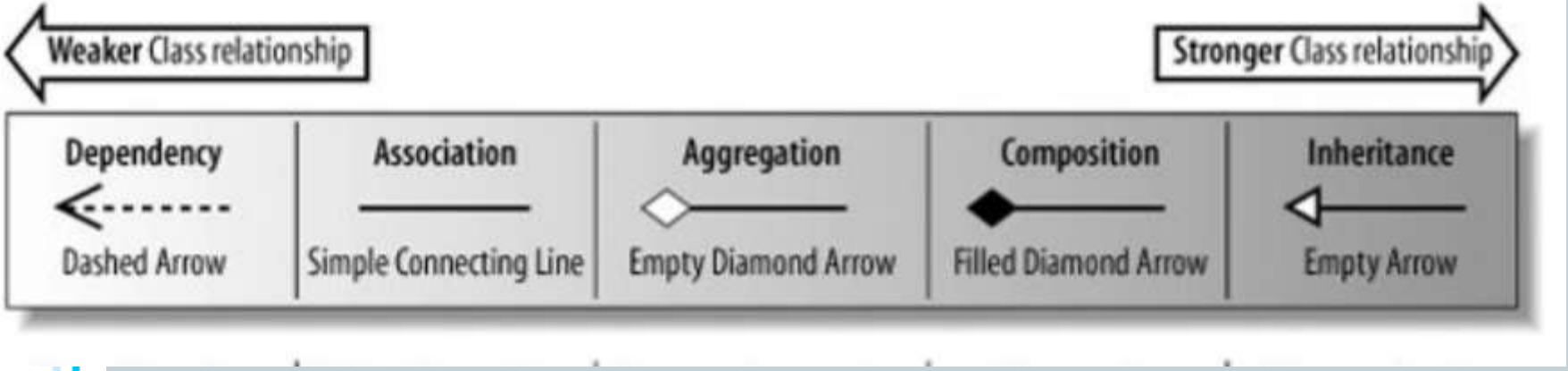
Avoiding unnecessary generalizations (cont)



Open in Umple

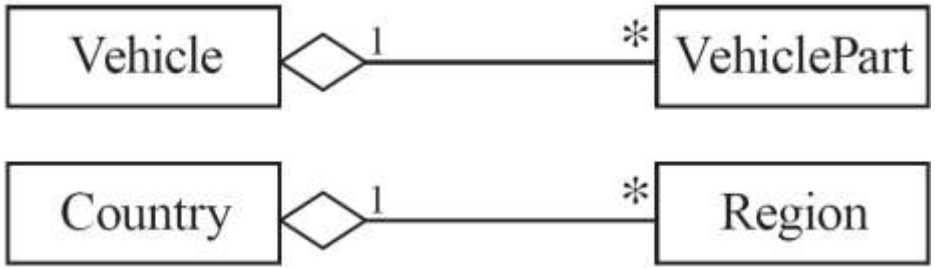
Improved class diagram, with its corresponding instance diagram

Relationships



More Advanced Features: Aggregation

- Aggregations are special associations that represent 'part-whole' relationships.
  - The 'whole' side is often called the *assembly* or the *aggregate*
  - This symbol is a shorthand notation association named `isPartOf`



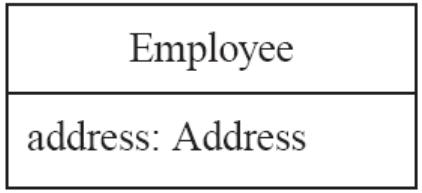
When to use an aggregation

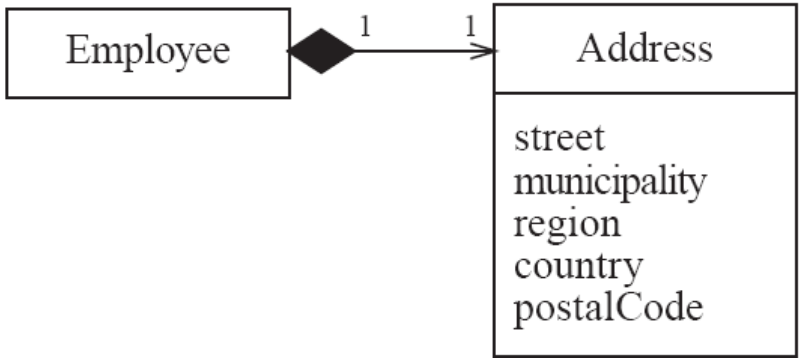
- As a general rule, you can mark an association as an aggregation if the following are true:
  - You can state that
    - the parts ‘are part of’ the aggregate
    - or the aggregate ‘is composed of’ the parts
  - When something owns or controls the aggregate, then they also own or control the parts

Composition

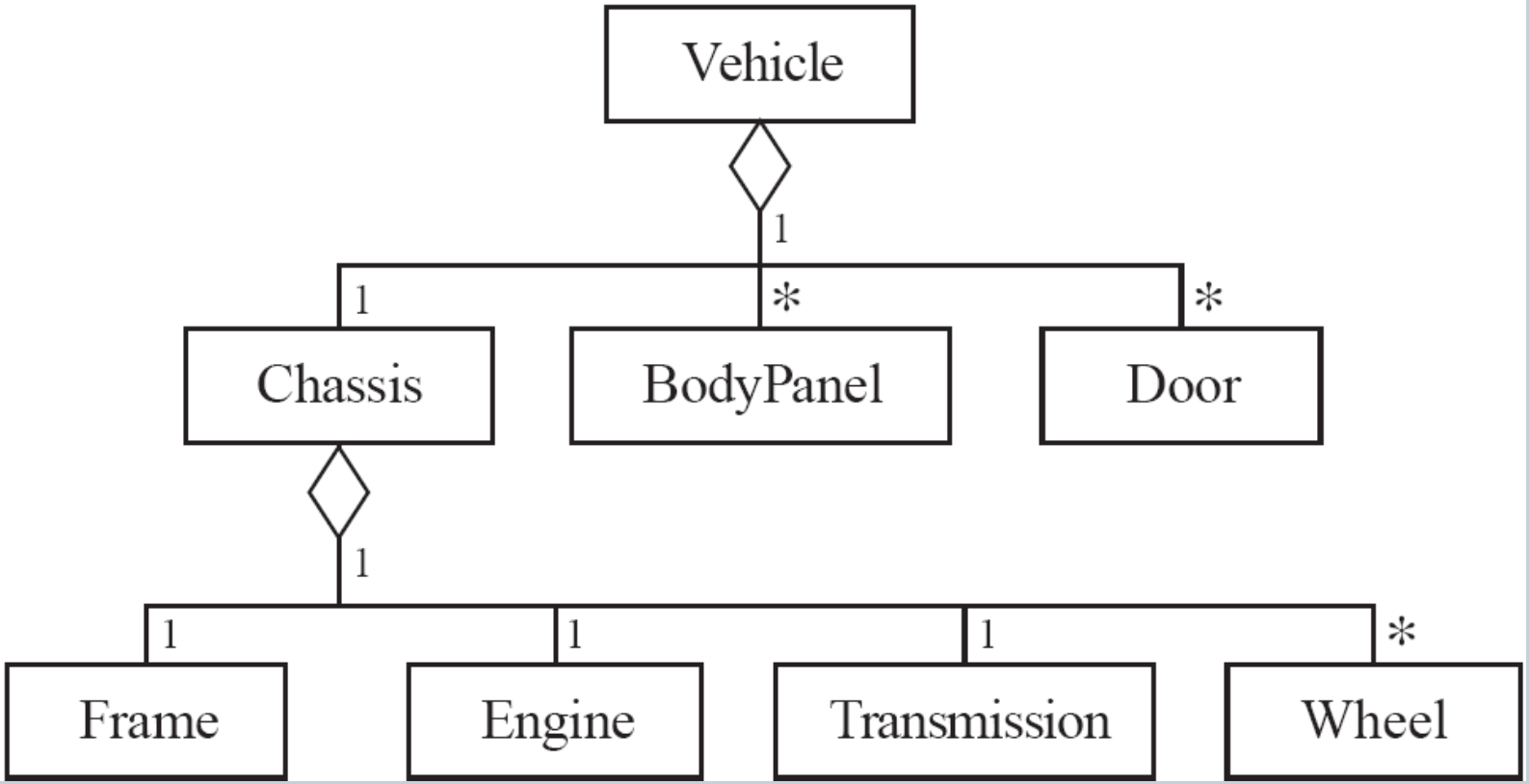
- A *composition* is a strong kind of aggregation
  - if the aggregate is destroyed, then the parts are destroyed as well

- Two alternatives for addresses





Aggregation hierarchy



Propagation

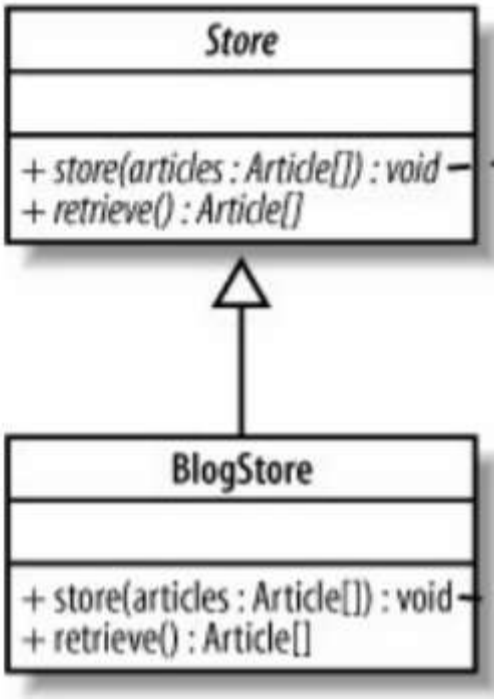
- A mechanism where an operation in an aggregate is implemented by having the aggregate perform that operation on its parts
- At the same time, properties of the parts are often propagated back to the aggregate
- Propagation is to aggregation as inheritance is to generalization.
  - The major difference is:
    - inheritance is an implicit mechanism

Abstract class

- When the concrete implementation of methods are left for the subclasses.
- Can contain both abstract and non-abstract methods

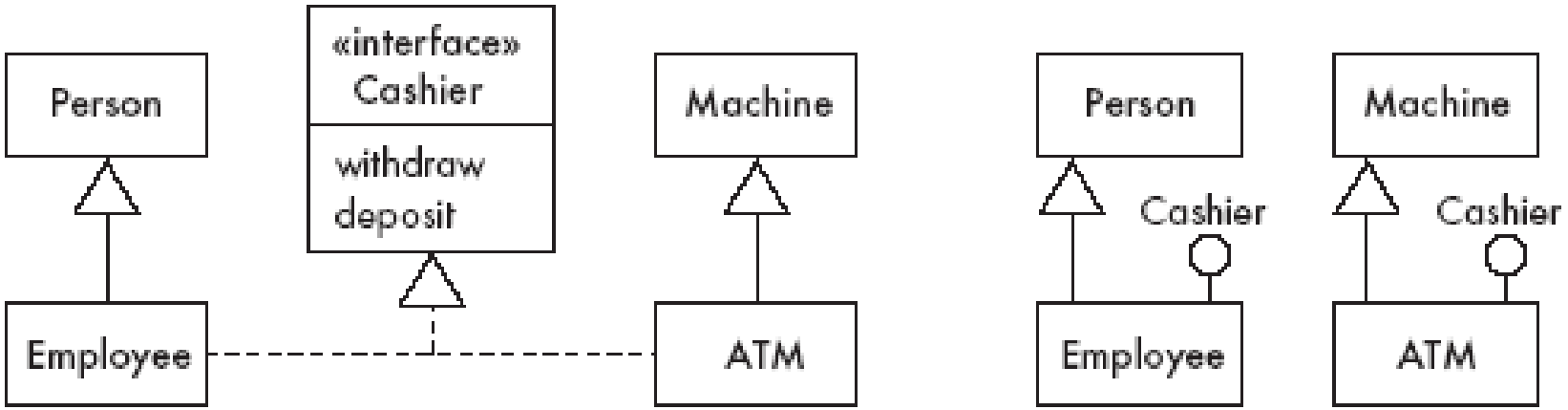

```
public abstract class Store {
    public abstract void store(Article[] articles);
    public abstract Article[] retrieve(  );
}
```

Abstract class



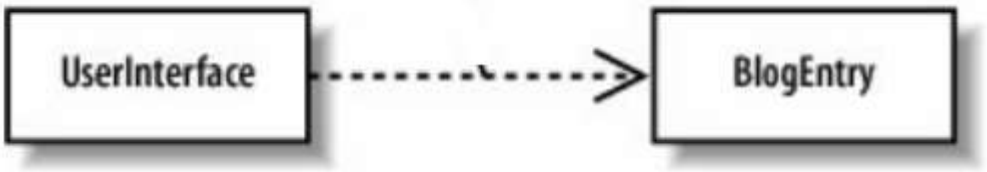
Interfaces

- An *interface* is similar to a class, except it lacks instance variables and implemented methods
- An interface describes a *portion of the visible behaviour* of a set of objects.



Dependency

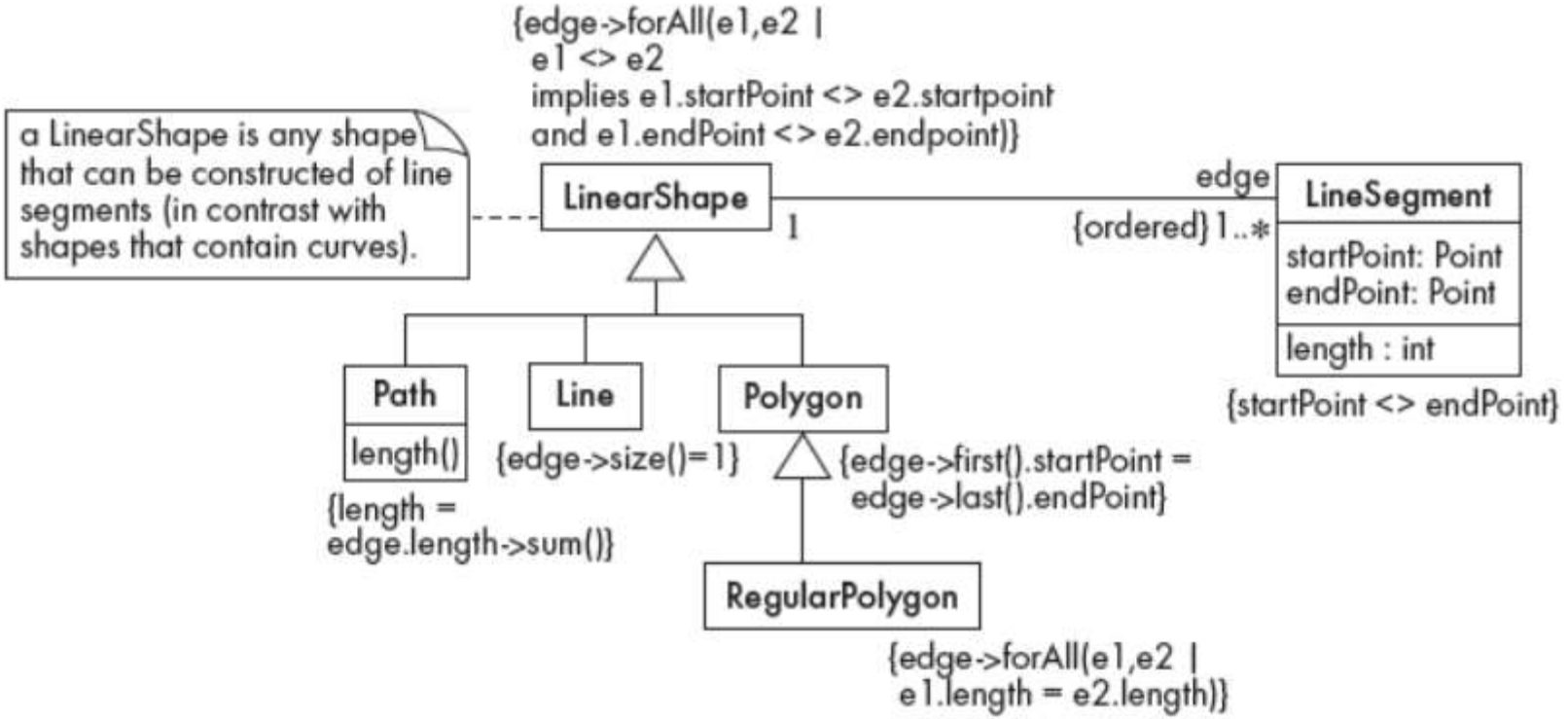
- A class needs to know about the other class in order use it's objects
- When the UserInterface wants to display, it accesses BlogEntry



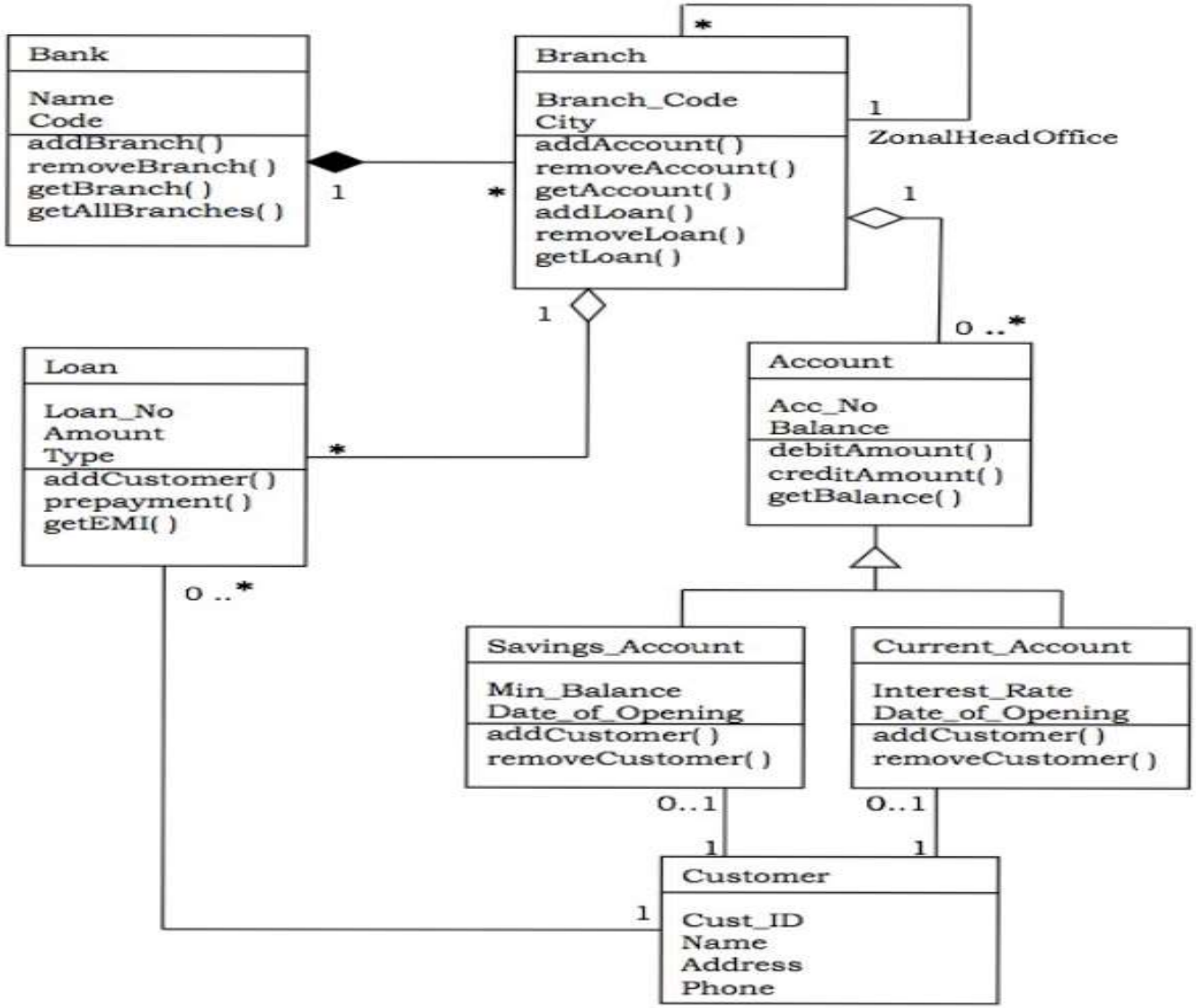
- Dependency implies only that the classes can work together, so is the weakest relationship

Notes and descriptive text

- **Descriptive text and other diagrams**
  - Embed your diagrams in a larger document
  - Text can explain aspects of the system using any notation you like
  - Highlight and expand on important features, and give rationale
- **Notes**:
  - A note is a small block of text embedded *in* a UML diagram
  - It acts like a comment in a programming language
- Constraints:
  - A constraint is like a note, except that it is written in a formal language that can be interpreted by a computer
  - Recommended language is Object Constraint Language


{edge->forAll(e1,e2 |
e1 <> e2
implies e1.startPoint <> e2.startpoint
and e1.endPoint <> e2.endpoint)}
a LinearShape is any shape that can be constructed of line segments (in contrast with shapes that contain curves).
LinearShape
edge
1
{ordered} 1..*
LineSegment
startPoint: Point
endPoint: Point
length : int
{startPoint <> endPoint}
Path
length()
{length = edge.length->sum()}
Line
{edge->size()=1}
Polygon
{edge->first().startPoint = edge->last().endPoint}
RegularPolygon
{edge->forAll(e1,e2 |
e1.length = e2.length)}

An Example


Bank
Name
Code
addBranch( )
removeBranch( )
getBranch( )
getAllBranches( )
Branch
Branch_Code
City
addAccount( )
removeAccount( )
getAccount( )
addLoan( )
removeLoan( )
getLoan( )
ZonalHeadOffice
Loan
Loan_No
Amount
Type
addCustomer( )
prepayment( )
getEMI( )
Account
Acc_No
Balance
debitAmount( )
creditAmount( )
getBalance( )
Savings_Account
Min_Balance
Date_of_Opening
addCustomer( )
removeCustomer( )
Current_Account
Interest_Rate
Date_of_Opening
addCustomer( )
removeCustomer( )
Customer
Cust_ID
Name
Address
Phone
1
*
0 ..*
0..1

Suggested sequence of activities

- Identify a first set of candidate **classes**
- Add **associations** and **attributes**
- Find **generalizations**
- List the main **responsibilities** of each class
- Decide on specific **operations**
- **Iterate** over the entire process until the model is satisfactory
  - Add or delete classes, associations, attributes, generalizations, responsibilities or operations
  - Identify interfaces

- *Don't be too disorganized. Don't be too rigid either.*

A simple technique for discovering domain classes

- Look at a source material such as a description of requirements
- Extract the *nouns* and *noun phrases*
- Eliminate nouns that:
  - are redundant
  - represent instances
  - are vague or highly general
  - not needed in the application
- Pay attention to classes in a domain model that represent *types of users* or other actors

Identifying associations and attributes

- Start with classes you think are most **central** and important
- Decide on the clear and obvious data it must contain and its relationships to other classes.
- Work outwards towards the classes that are less important.
- Avoid adding many associations and attributes to a class
  - A system is simpler if it manipulates less information

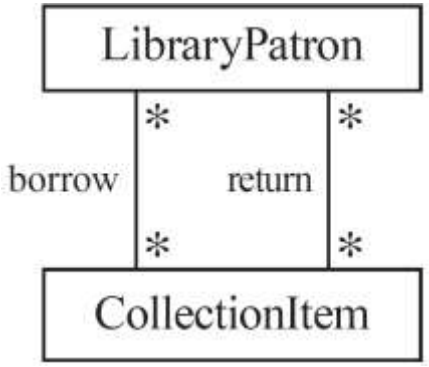
Tips about identifying and specifying valid associations

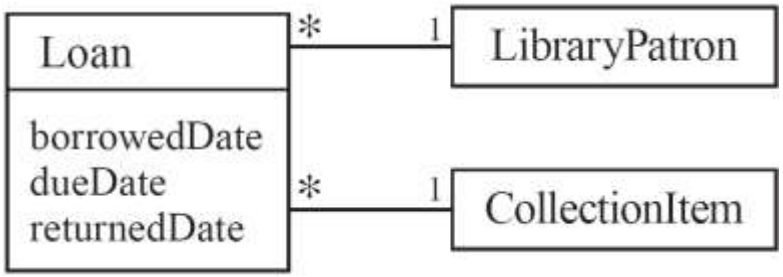
- An association should exist if a class
  - *possesses*
  - *controls*
  - *is connected to*
  - *is related to*
  - *is a part of*
  - *has as parts*
  - *is a member of*, or
  - *has as members*

  some other class in your model
- Specify the multiplicity at both ends
- Label it clearly.

Actions versus associations

- A common mistake is to represent *actions* as if they were associations



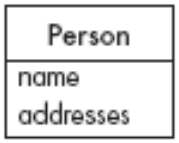


Identifying attributes

- Look for information that must be maintained about each class
- Several nouns rejected as classes, may now become attributes
- An attribute should generally contain a simple value
  - E.g. string, number

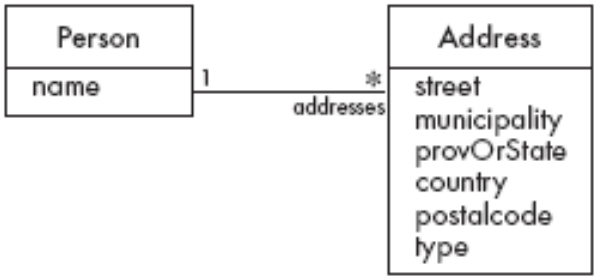
Tips about identifying and specifying valid attributes

- It is not good to have many duplicate attributes
- If a subset of a class's attributes form a coherent group, then create a distinct class containing these attributes



Bad, due to a plural attribute

Bad, due to too many attributes, and the inability to add more addresses



Good solution. The type indicates whether it is a home address, business address etc.

Identifying generalizations and interfaces

- There are two ways to identify generalizations:
  - bottom-up
    - Group together similar classes creating a new superclass
  - top-down
    - Look for more general classes first, specialize them if needed
- Create an *interface*, instead of a superclass if
  - The classes are very dissimilar except for having a few operations in common
  - One or more of the classes already have their own superclasses
  - Different implementations of the same class might be available

Allocating responsibilities to classes

- A *responsibility* is something that the system is required to do.
  - Each functional requirement must be attributed to one of the classes
    - All the responsibilities of a given class should be *clearly related*.
    - If a class has too many responsibilities, consider *splitting* it into distinct classes
    - If a class has no responsibilities attached to it, then it is probably *useless*
    - When a responsibility cannot be attributed to any of the existing classes, then a *new class* should be created
  - To determine responsibilities
    - Perform use case analysis
    - Look for verbs and nouns describing *actions* in the system description

Categories of responsibilities

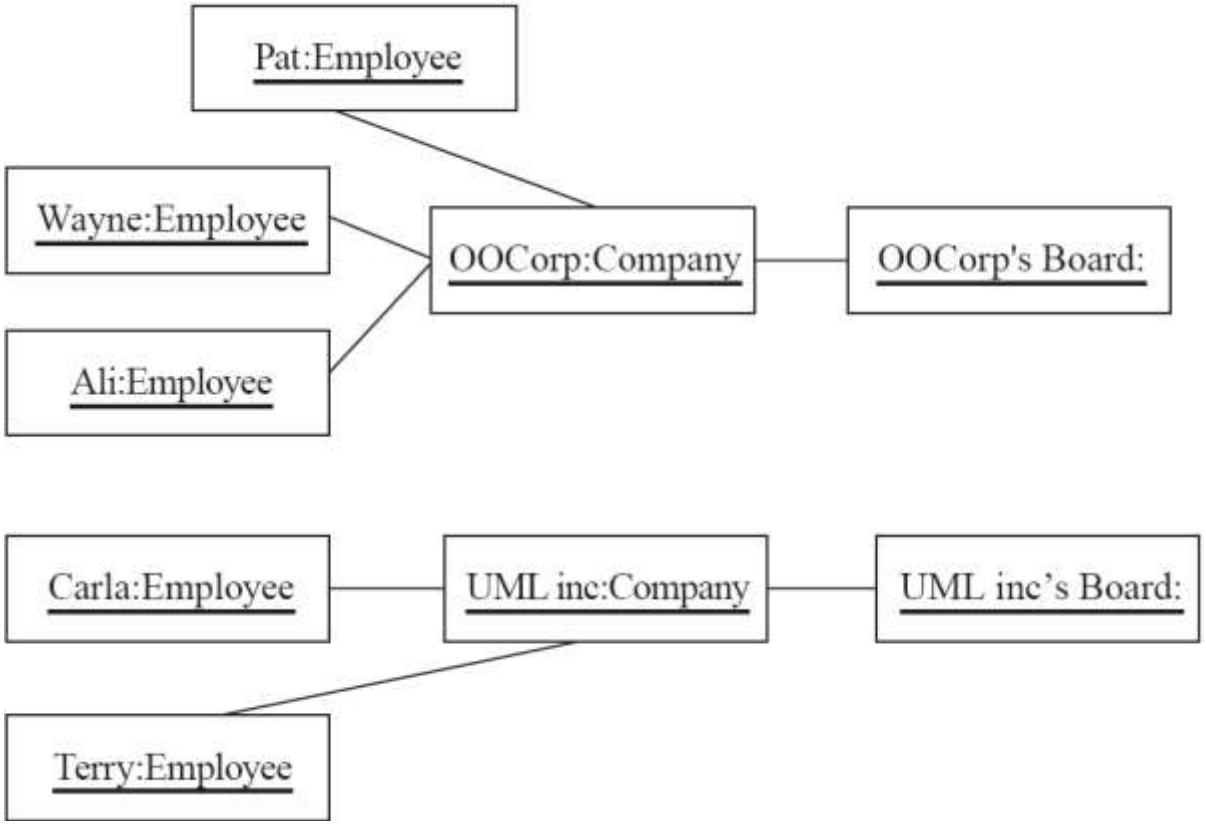
- Setting and getting the values of attributes
- Creating and initializing new instances
- Loading to and saving from persistent storage
- Destroying instances
- Adding and deleting links of associations
- Copying, converting, transforming, transmitting or outputting
- Computing numerical results
- Navigating and searching
- Other specialized work

Identifying operations

- Operations are needed to realize the responsibilities of each class
  - There may be several operations per responsibility
  - The main operations that implement a responsibility are normally declared **public**
  - Other methods that collaborate to perform the responsibility must be as private as possible

Object Diagrams

- A *link* is an instance of an association
  - In the same way that we say an object is an instance of a class



Associations versus generalizations in object diagrams

- Associations describe the relationships that will exist between *instances* at run time.
  - When you show an instance diagram generated from a class diagram, there will be an instance of *both* classes joined by an association
- Generalizations describe relationships between *classes* in class diagrams.
  - They do not appear in instance diagrams at all.
  - An instance of any class should also be considered to be an instance of each of that class's superclasses

Modelling Interactions and Behaviour
Sequence Diagram

Interaction Diagrams

- Interaction diagrams are used to model the dynamic aspects of a software system
  - They help you to visualize how the system runs.
  - An interaction diagram is often built from a use case and a class diagram.
    - The objective is to show how a set of objects accomplish the required interactions with an actor.

Interactions and messages

- Interaction diagrams show how a set of actors and objects communicate with each other to perform:
  - The steps of a use case, or
  - The steps of some other piece of functionality.
- The set of steps, taken together, is called **an *interaction***.
- Interaction diagrams can show several different types of communication.
  - E.g. method calls, messages send over the network
  - These are all referred to as *messages*.

Elements found in interaction diagrams

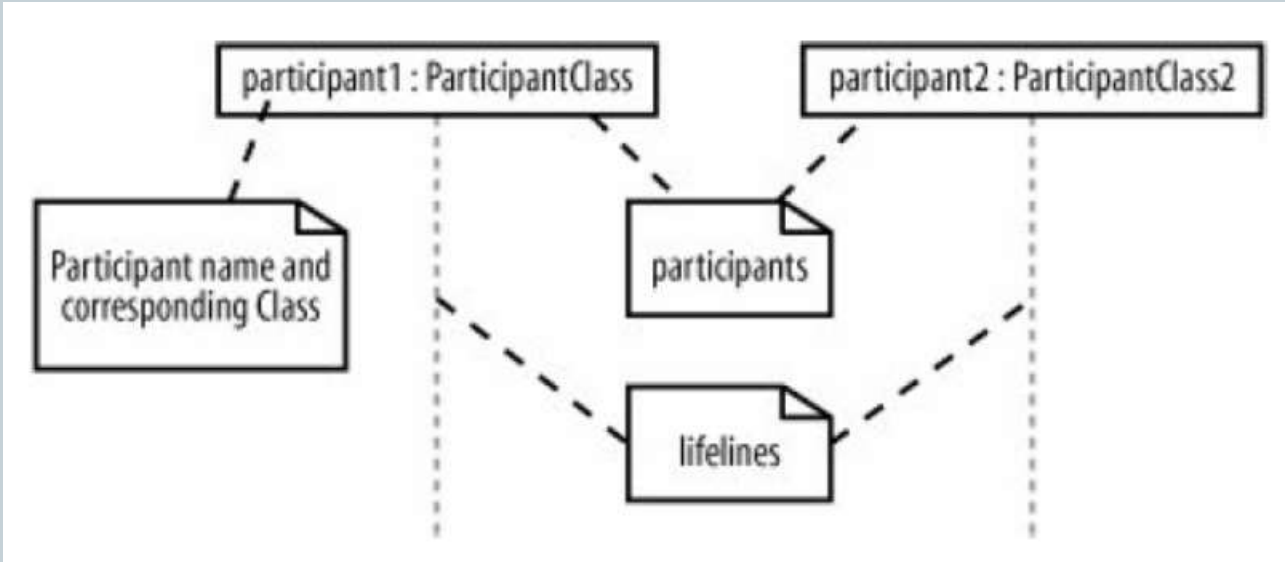
- Instances of classes
  - Shown as boxes with the class and object identifier underlined
- Actors
  - Use the stick-person symbol as in use case diagrams
- Messages
  - Shown as arrows from actor to object, or from object to object

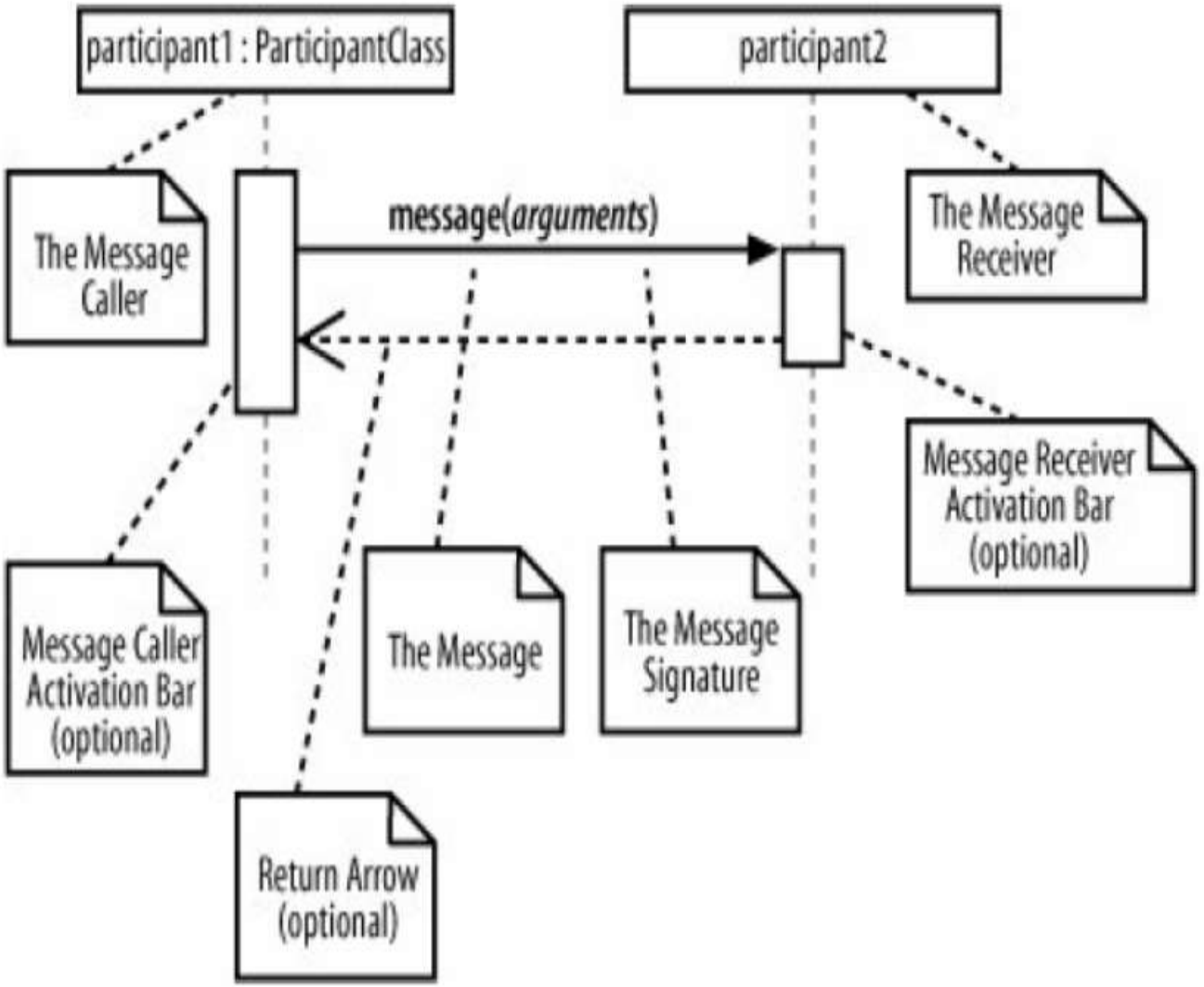
Creating interaction diagrams

- You should develop a class diagram and a use case model before starting to create an interaction diagram.
  - Important interaction diagrams:
    - *Sequence diagrams*
    - *Communication diagrams*
    - *Timing diagrams*

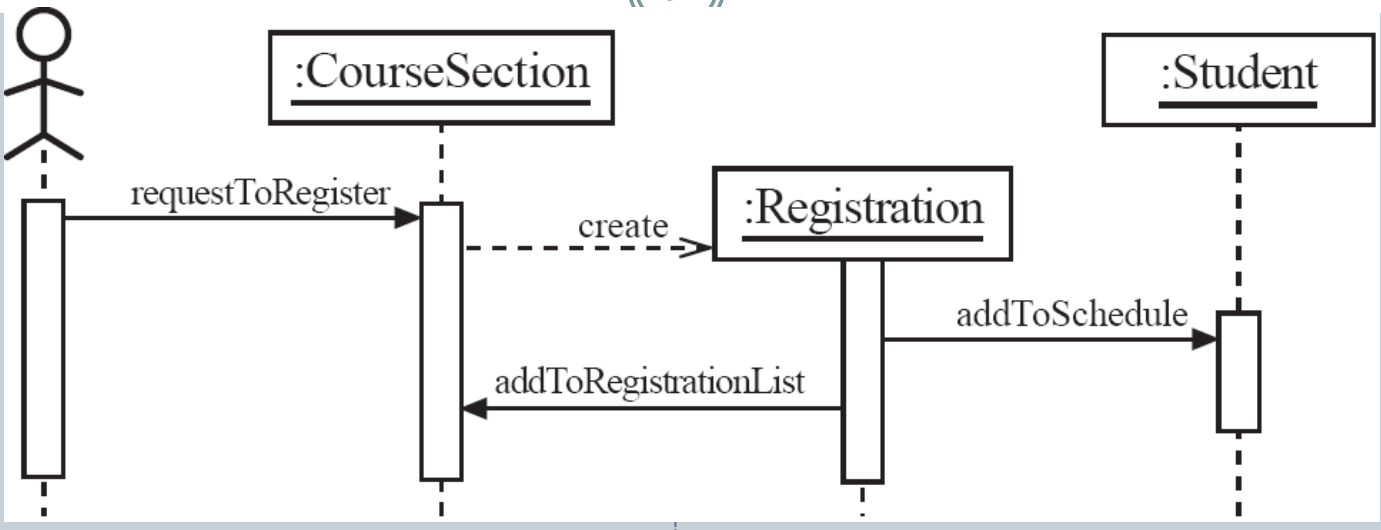
Sequence diagrams

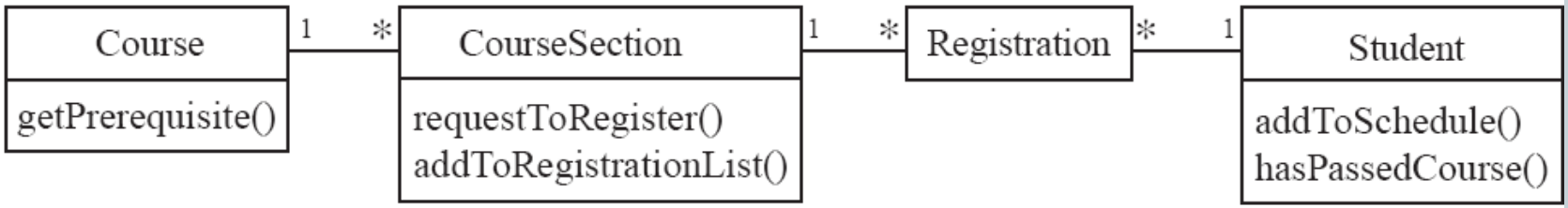
- A sequence diagram shows the sequence of messages exchanged by the set of objects performing a certain task
  - The objects are arranged horizontally across the diagram.
  - An actor that initiates the interaction is often shown on the left.
  - The vertical dimension represents time.
  - A vertical line, called a *lifeline*, is attached to each object or actor.
  - The lifeline becomes a broad box, called an *activation box* during the *live activation* period.
  - A message is represented as an arrow between activation boxes of the sender and receiver.
    - A message is labelled and can have an argument list and a return value.


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

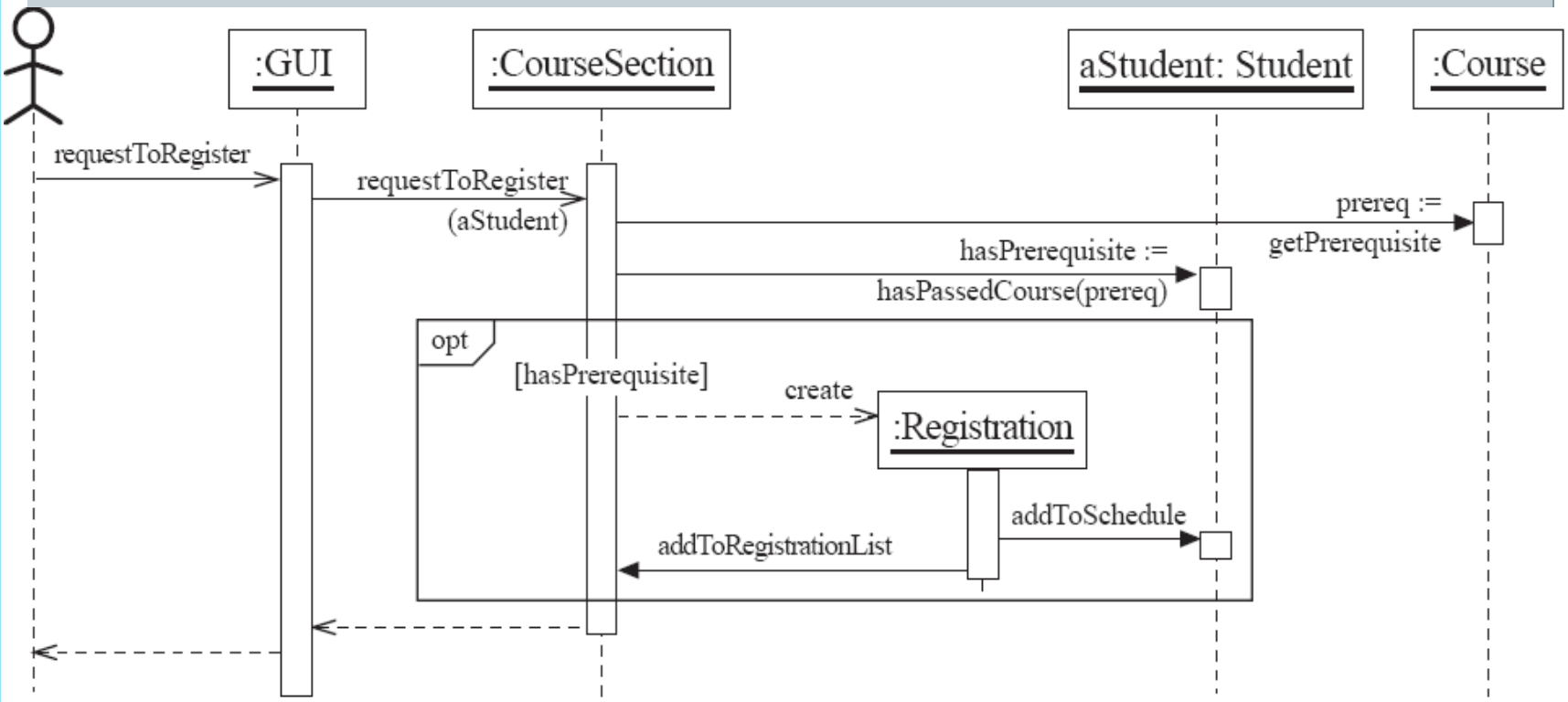

participant1 : ParticipantClass
participant2
The Message Caller
message(arguments)
The Message Receiver
Message Receiver Activation Bar (optional)
Message Caller Activation Bar (optional)
The Message
The Message Signature
Return Arrow (optional)

Sequence diagrams – an example

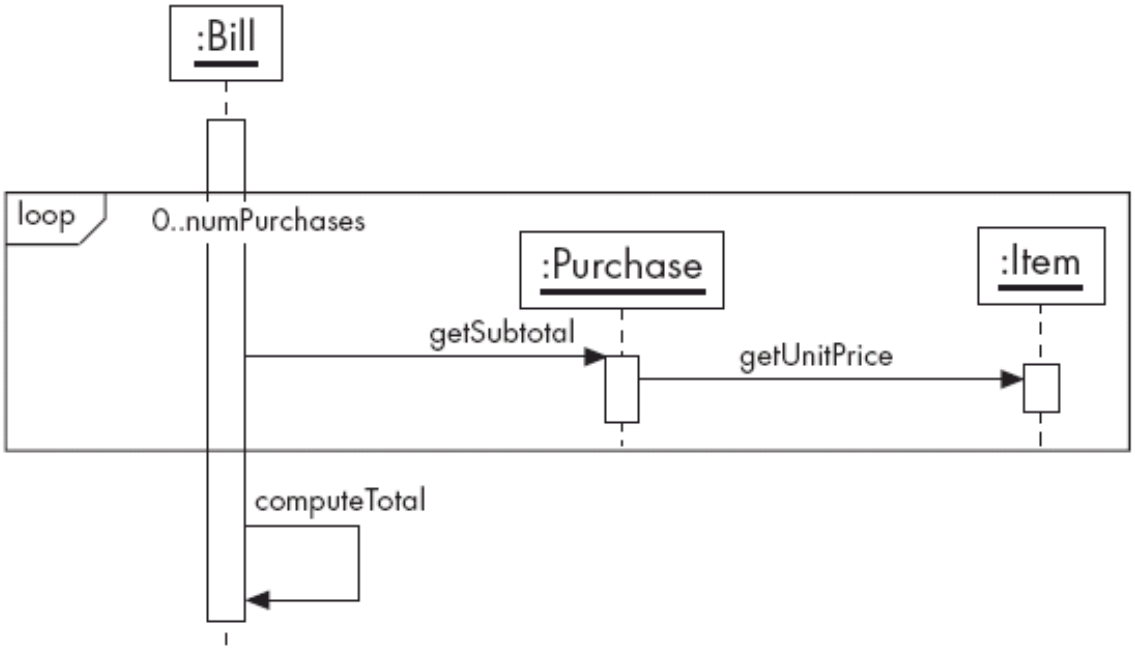


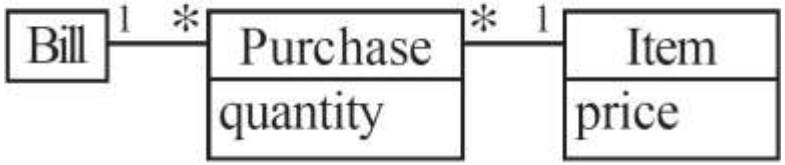


Sequence diagrams – same example, more details



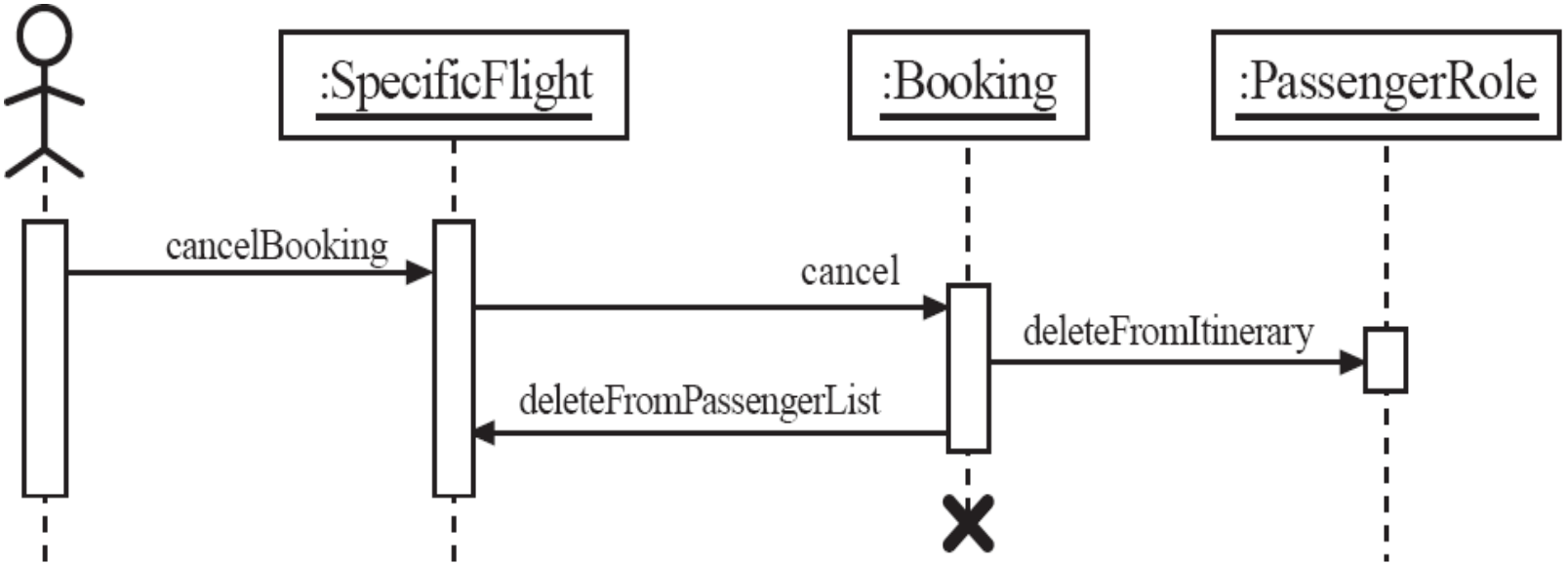
Sequence diagrams – an example with replicated messages





Sequence diagrams – an example with object deletion

- If an object's life ends, this is shown with an X at the end of the lifeline



Quiz

- On 2nd March 2021, after the lecture
  - Class Diagram
  - Sequence Diagram

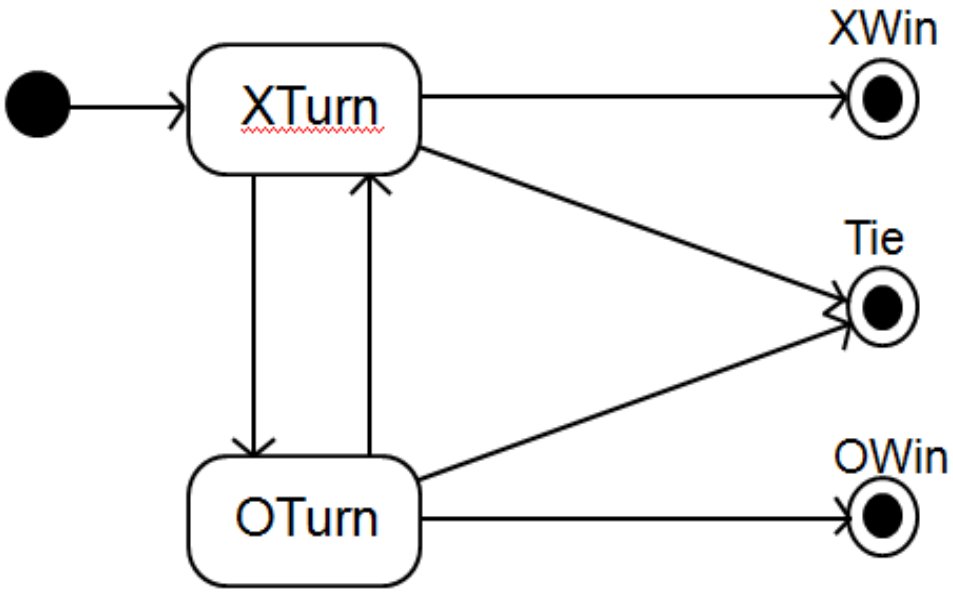
STATE DIAGRAM AND ACTIVITY DIAGRRAMS

State Diagrams

- **A state diagram describes the behaviour of a *system*, some *part* of a system, or an *individual object*.**
  - At any given point in time, the system or object is in a certain ***state***.
    - Being in a state means that it is will behave in a *specific way* in response to any events that occur.
  - Some events will cause the system to change state.
    - In the new state, the system will behave in a different way to events.
  - A state diagram is a directed graph where the nodes are states and the arcs are transitions.

State diagrams – an example

- tic-tac-toe game



States

- At any given point in time, the system is in one state.
- It will remain in this state until an event occurs that causes it to change state.
- A state is represented by a **rounded rectangle** containing the name of the state.
- Special states:
  - A black circle represents the ***start state***
  - A circle with a ring around it represents an ***end state***

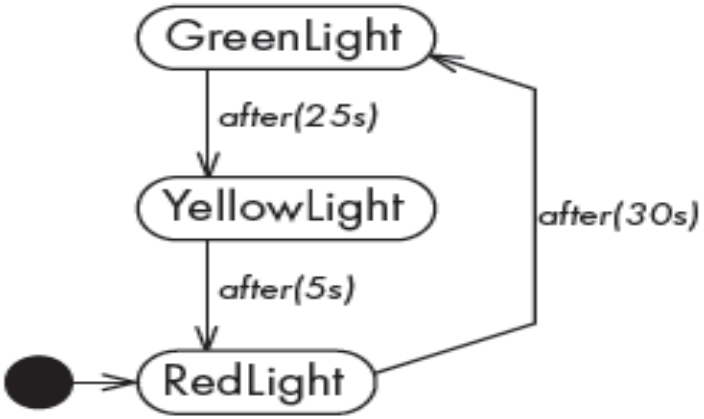
Transitions

- A transition represents a change of state in response to an event.
  - It is considered to occur instantaneously.

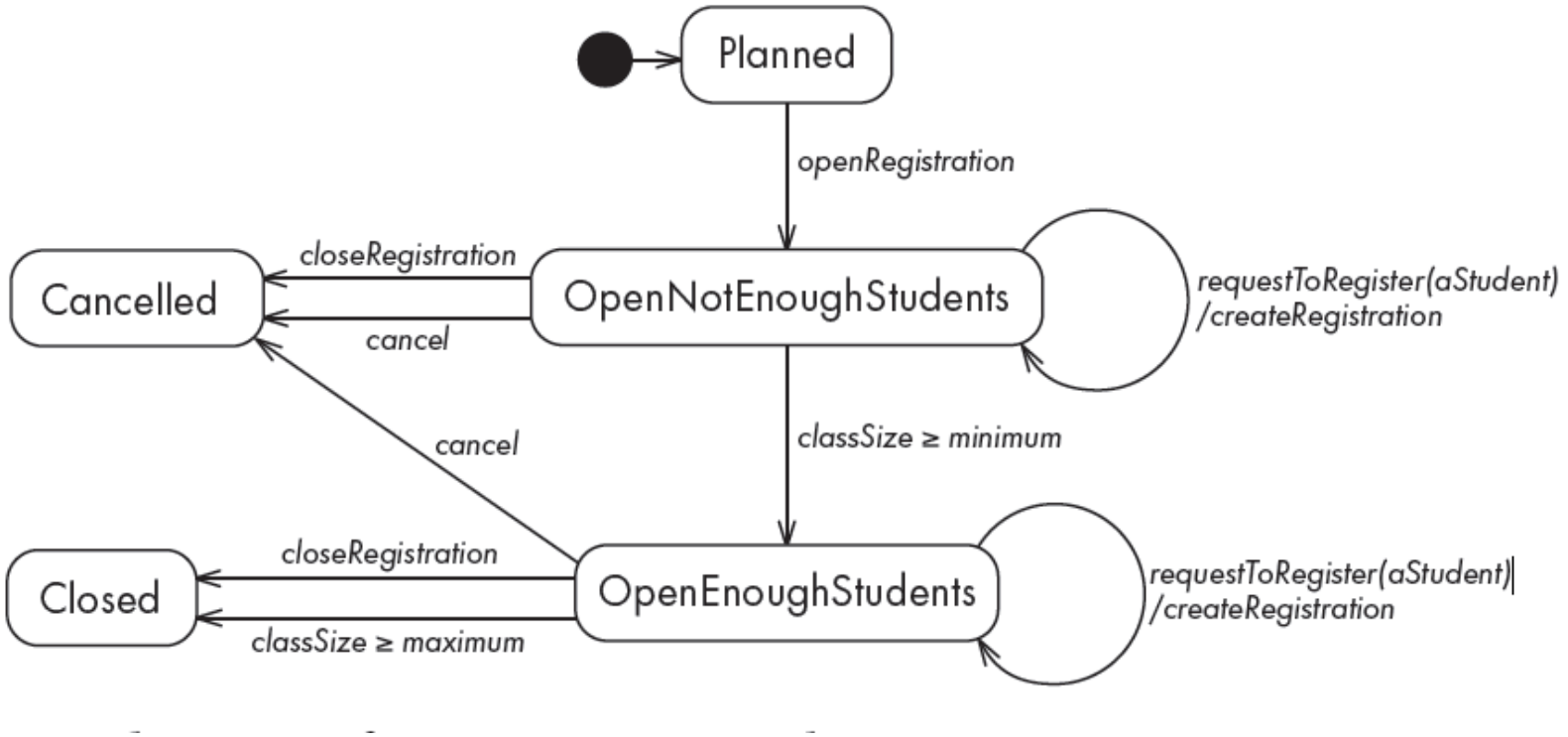
- The label on each transition is the event that causes the change of state.
- A transition is rendered as a **solid directed line**.

State diagrams – an example of transitions with time-outs and conditions

State diagrams of a simple traffic light, illustrating elapsed-time transitions



State diagrams – an example with conditional transitions



State diagram of a CourseSection class

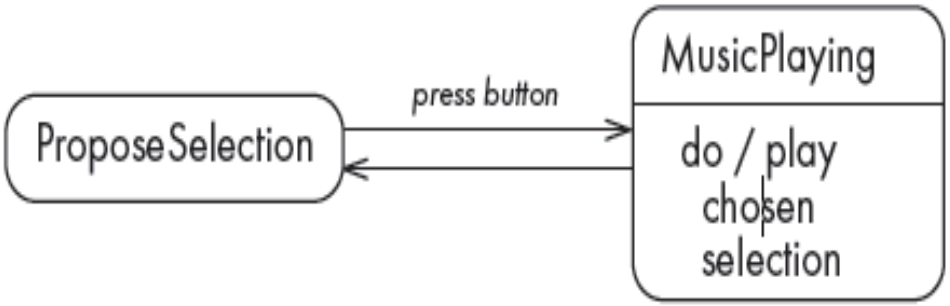
Activities in state diagrams

- An *activity* is something that takes place while the system is *in* a state.
  - It takes a period of time.
  - The system may take a transition out of the state in response to completion of the activity,
  - Some other outgoing transition may result in:
    - The interruption of the activity, and
    - An early exit from the state.

Activity representation

- An activity is shown textually within a state box by the word '**do**' followed by a '**/**' symbol, and a ***description*** of what is to be done.
- When you have details such as actions in a state, you draw a horizontal line above them to separate them from the state name.

State diagram – an example with activity



State diagram for a jukebox, illustrating an activity in a state

Actions in state diagrams

- An *action* is something that takes place effectively *instantaneously*
  - When a particular transition is taken,
  - Upon entry into a particular state, or
  - Upon exit from a particular state

- ***An action should consume no noticeable amount of time***

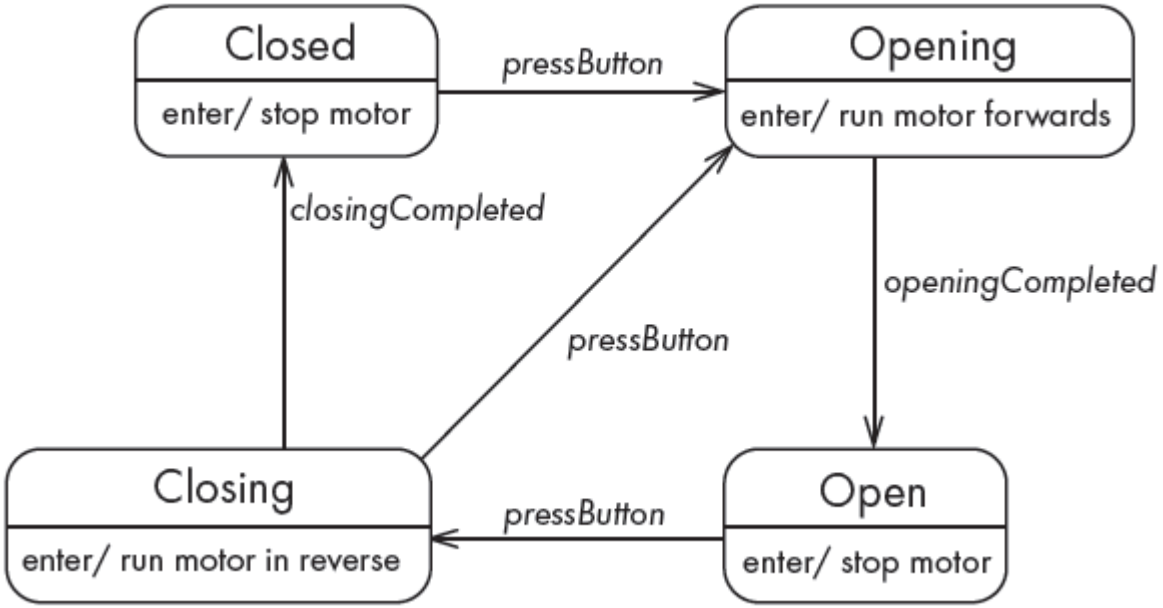
- It should be something simple, such as sending a message, starting a hardware device or setting a variable.

Representation of action

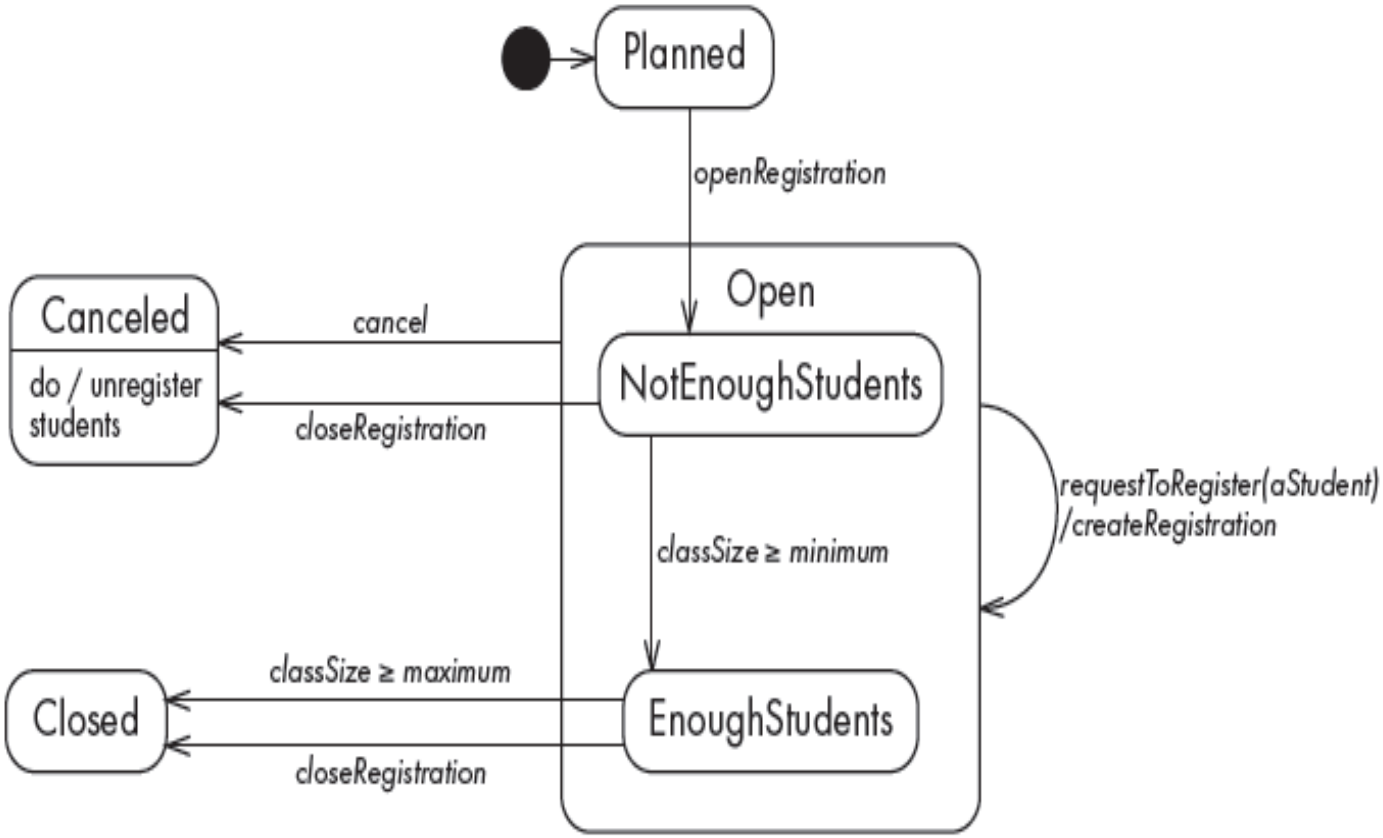
- An action is always shown preceded by a slash (‘/’) symbol.
- If the action is to be performed during a transition, then the syntax is **event/action**.
- If the action is to be performed when entering or exiting a state, then it is written in the state box with the notation **enter/action** or **exit/action**.

State diagram – an example with actions

- **State diagram for a garage door opener, showing actions triggered by entry into a state**



State diagram – an example with substates



A version of the course section example from Figure 8.14, showing the effect of nested states

Activity Diagrams

- **Activity diagram** is another important behavioural diagram in **UML** diagram to describe dynamic aspects of the system.
- The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario.
  - Activity diagram is essentially an advanced version of flow chart that modelling the flow from one activity to another activity.
  - An *activity diagram* is like ***a state diagram***.
    - Except most transitions are caused by *internal* events, such as the completion of a computation.

Activity Diagrams.....

- An activity diagram
  - Can be used to understand the flow of work that an object or component performs.
  - Can also be used to visualize the interrelation and interaction between different use cases.
  - Is most often associated with several classes.

- One of the strengths of activity diagrams is the representation of ***concurrent*** **activities**.

Activity Diagram Notation

- Activity diagram uses ***rounded rectangles*** to imply a specific system function
- ***Arrows*** to represent flow through the system.
- ***Decision diamonds*** to depict a branching decision (each arrow emanating from the diamond is labelled).
- ***Solid horizontal lines*** to indicate that parallel activities are occurring.

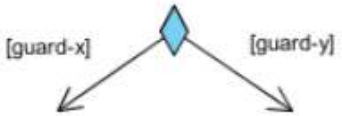
Activity Diagram Notation .....

Activity Is used to represent a set of actions	
Control Flow Shows the sequence of execution	
Initial Node Portrays the beginning of a set of actions or activities	
Activity Final Node Stop all control flows and object flows in an activity (or action)	

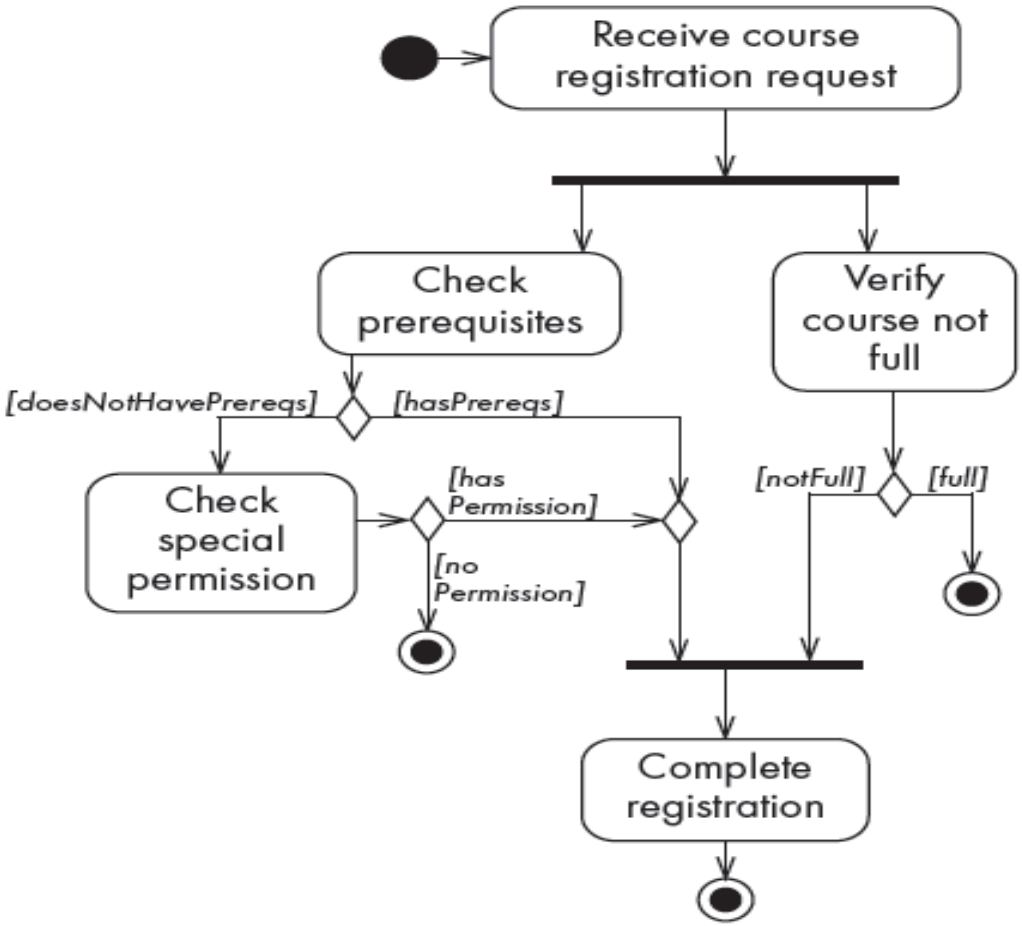
Decision nodes and merge nodes

- An activity diagram has two types of nodes for branching within a single thread. These are represented as small ***diamonds***:
  - ***Decision node***
    - *has one incoming transition and multiple outgoing transitions* each with a Boolean guard in square brackets. Exactly one of the outgoing transitions will be taken.
  - ***Merge node***
    - *has two incoming transitions and one outgoing transition. It is* used to bring together paths that had been split by decision nodes.

Activity Diagram Notation .....

Decision Node Represent a test condition to ensure that the control flow or object flow only goes down one path	
Merge Node Bring back together different decision paths that were created using a decision-node.	

Activity diagrams – an example



Activity diagram of the registration process

Representing concurrency

- Concurrency is shown using **forks, joins and rendezvous.**
  - A ***fork*** has one incoming transition and multiple outgoing transitions.
    - The execution splits into two concurrent threads.
  - A ***rendezvous*** has multiple incoming and multiple outgoing transitions.
    - Once all the incoming transitions occur all the outgoing transitions may occur.

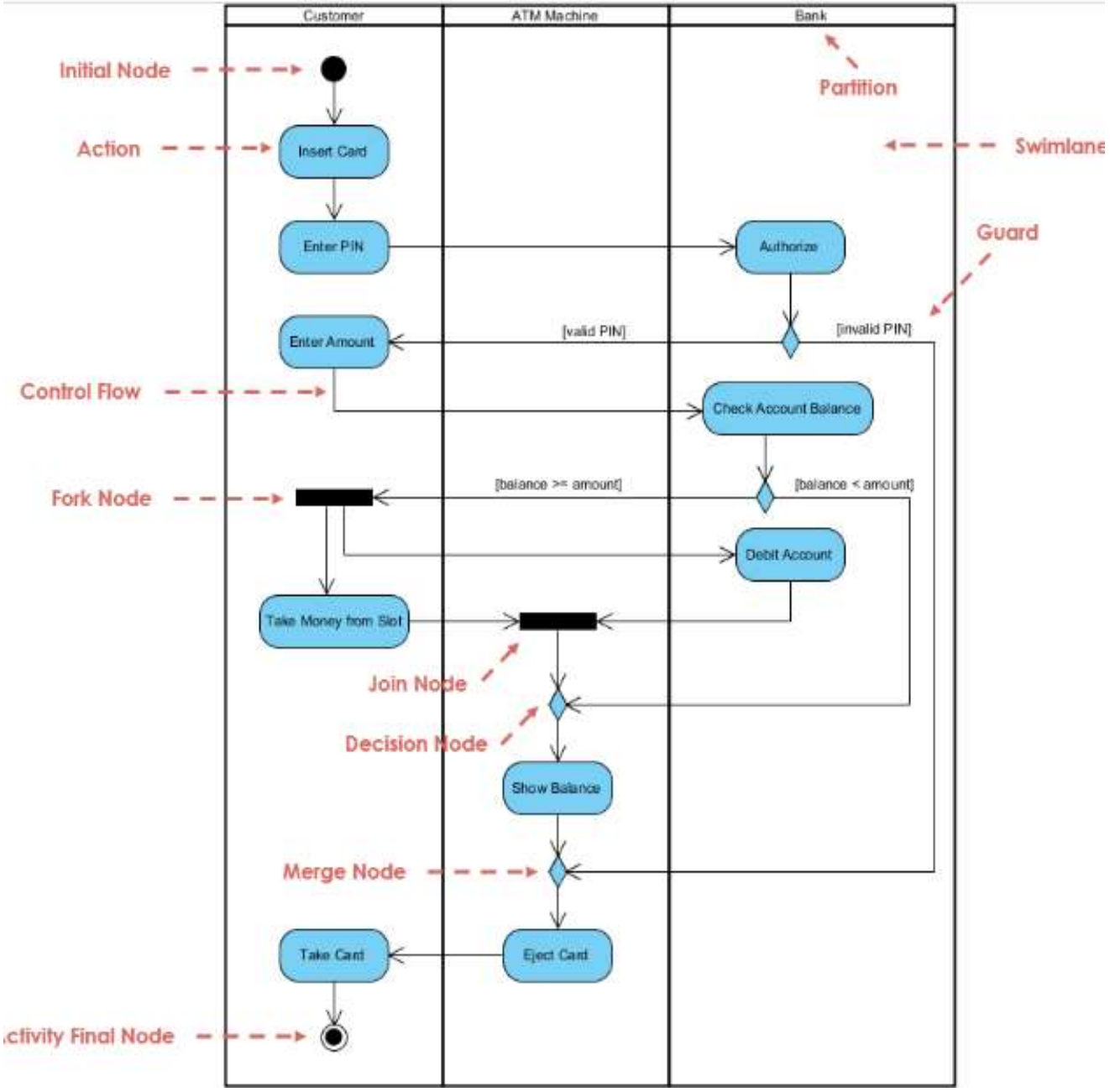
Representing concurrency

- A ***join*** has multiple incoming transitions and one outgoing transition.
  - The outgoing transition will be taken when all incoming transitions have occurred.
  - The incoming transitions must be triggered in separate threads.
  - If one incoming transition occurs, a wait condition occurs at the join until the other transitions occur.

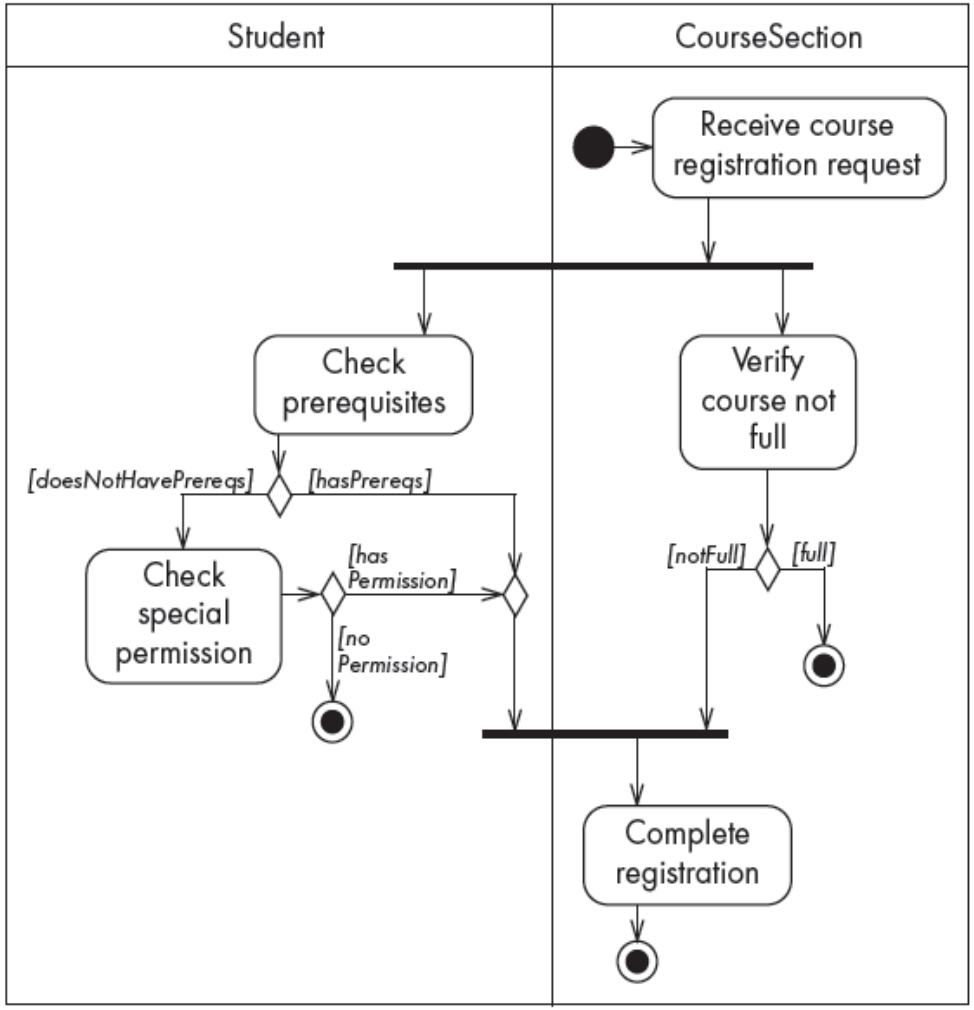
Swimlanes

- Activity diagrams are most often associated with several classes.
- The partition of activities among the existing classes can be explicitly shown in an activity diagram by the introduction of ***swimlanes.***
  - *Allows* you to represent the flow of activities described by the use case
  - indicate which actor or analysis class has responsibility for the action described by an activity rectangle.

- Responsibilities are represented as parallel segments that divide the diagram vertically, like the lanes in a swimming pool.


Customer
ATM Machine
Bank
Initial Node
Partition
Action
Insert Card
Swimlane
Guard
Enter PIN
Authorize
[valid PIN]
[invalid PIN]
Enter Amount
Control Flow
Check Account Balance
[balance >= amount]
[balance < amount]
Fork Node
Debit Account
Take Money from Slot
Join Node
Decision Node
Show Balance
Merge Node
Take Card
Eject Card
ctivity Final Node

Activity diagrams – an example with swimlanes



Activity diagram with swimlanes