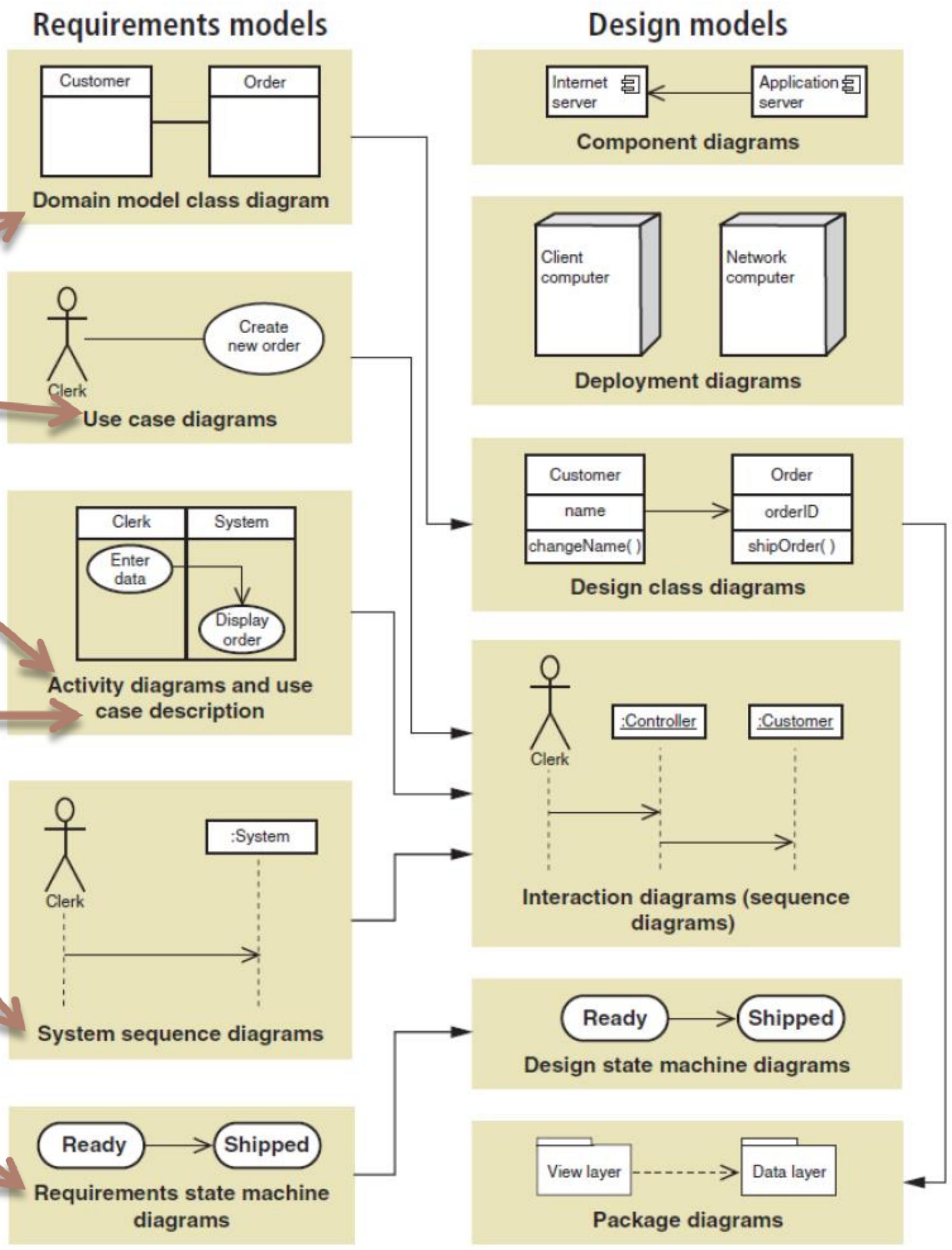
The Unified Modeling Language

Asst.Prof.Dr. Supakit Nootyaskool

IT-KMITL

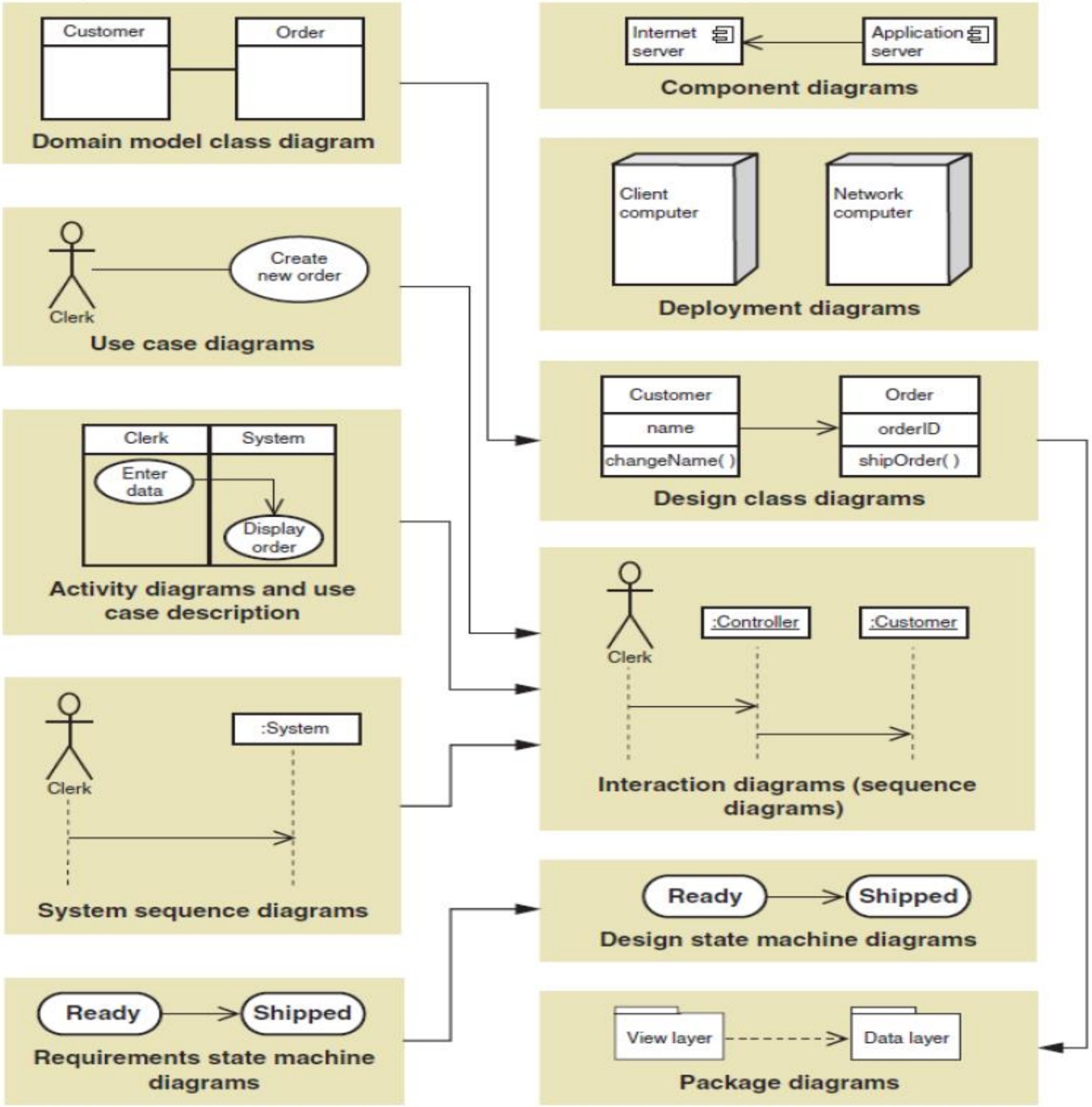
UML: requirement VS Design models

- Identify
  - All the classes or things
  - Elementary business process
  - Necessary step to carry out a use case
  - Describe document the internal workflow of each use case
  - Related activity diagram show message or data between user and system
  - Track all status of all condition requirement for a class.



Requirements models

Design models


Domain model class diagram

Use case diagrams

Activity diagrams and use case description

System sequence diagrams

Requirements state machine diagrams

Component diagrams

Deployment diagrams

Design class diagrams

Interaction diagrams (sequence diagrams)

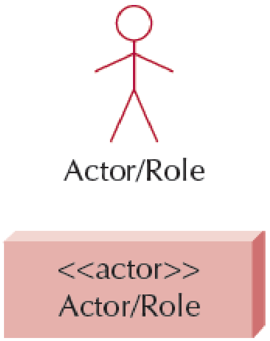
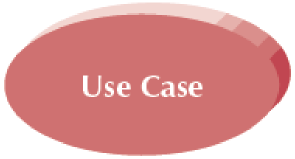
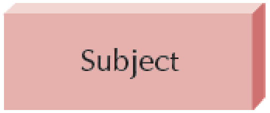
Design state machine diagrams

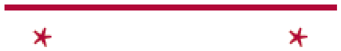
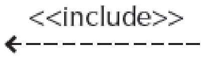
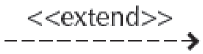
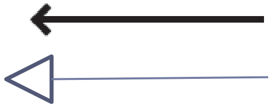
Package diagrams

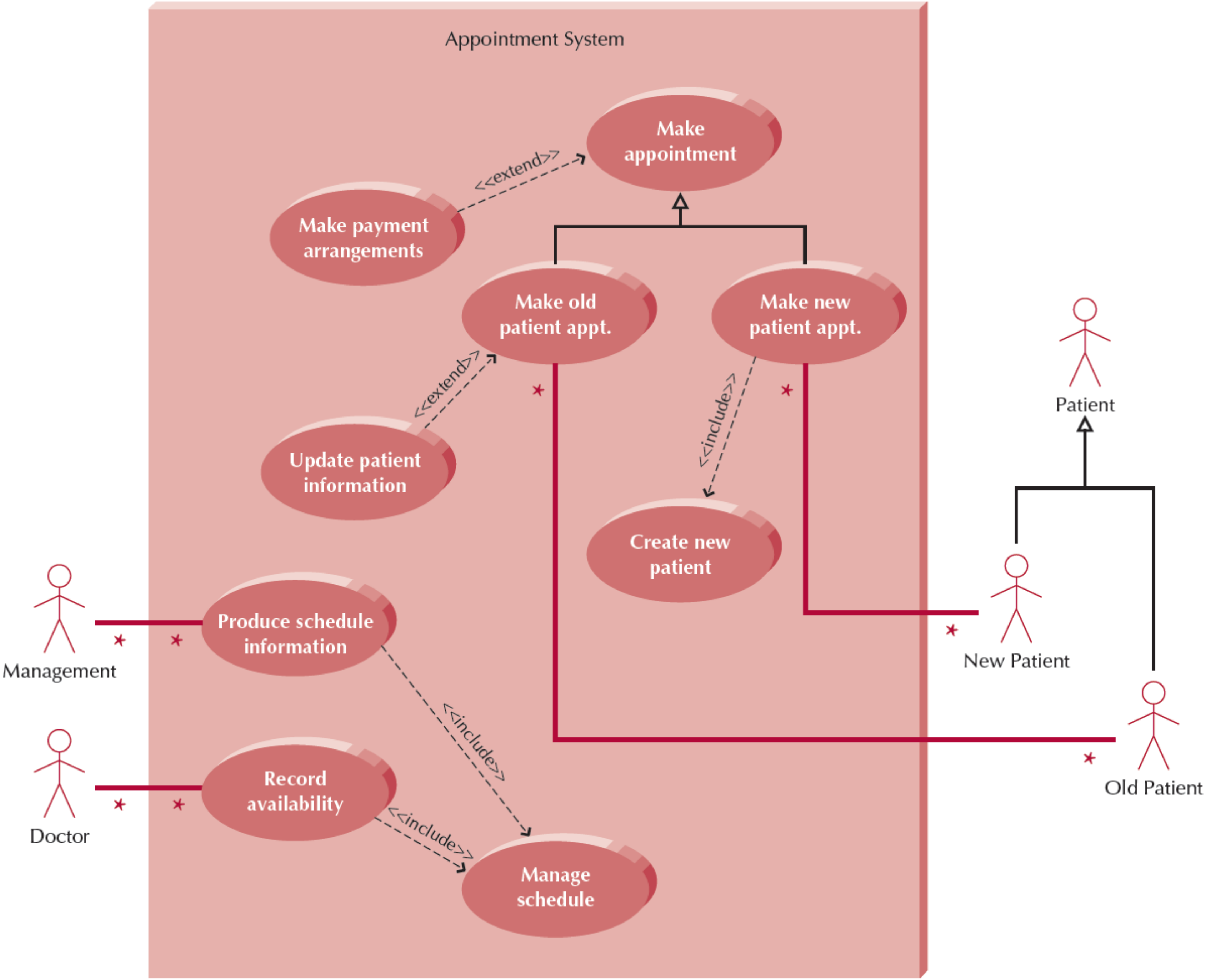
Use case diagram

Use case diagram

- The use case diagram is the UML model used to graphically show the use cases and their relationship to user.

An Actor: ■ Is a person or system that derives benefit from and is external to the subject ■ Is depicted as either a stick figure (default) or if a non-human actor is involved, as a rectangle with <<actor>> in it (alternative) ■ Is labeled with its role ■ Can be associated with other actors using a specialization/superclass association, denoted by an arrow with a hollow arrowhead ■ Are placed outside the subject boundary	
A Use Case: ■ Represents a major piece of system functionality ■ Can extend another use case ■ Can include another use case ■ Is placed inside the system boundary ■ Is labeled with a descriptive verb-noun phrase	
A Subject Boundary: ■ Includes the name of the subject inside or on top ■ Represents the scope of the subject, e.g., a system or an individual business process	

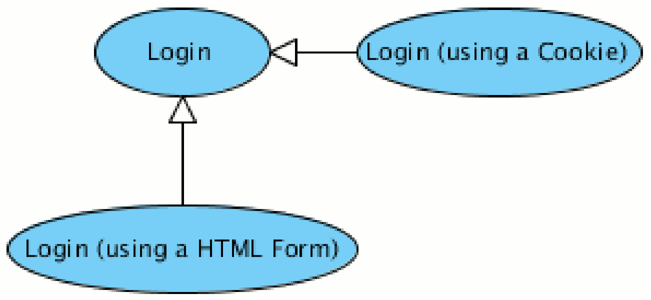
An Association Relationship: ■ Links an actor with the use case(s) with which it interacts	
An Include Relationship: ■ Represents the inclusion of the functionality of one use case within another ■ The arrow is drawn from the base use case to the included use case	
An Extend Relationship: ■ Represents the extension of the use case to include optional behavior ■ The arrow is drawn from the extension use case to the base use case	
A Generalization Relationship: ■ Represents a specialized use case to a more generalized one ■ The arrow is drawn from the specialized use case to the base use case	



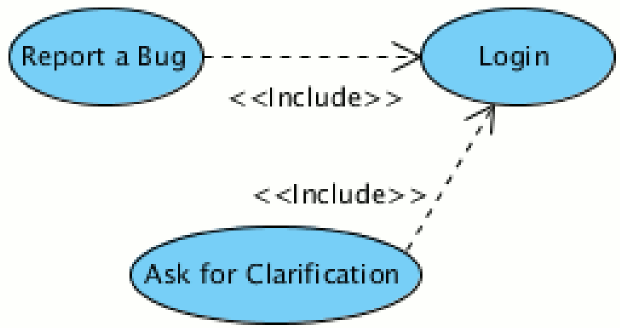
Appointment System
Make appointment
<<extend>>
Make payment arrangements
Make old patient appt.
Make new patient appt.
<<extend>>
<<include>>
Update patient information
Create new patient
Patient
*
*
New Patient
*
Management
*
*
Produce schedule information
<<include>>
Old Patient
*
Doctor
*
*
Record availability
<<include>>
Manage schedule

Symbols

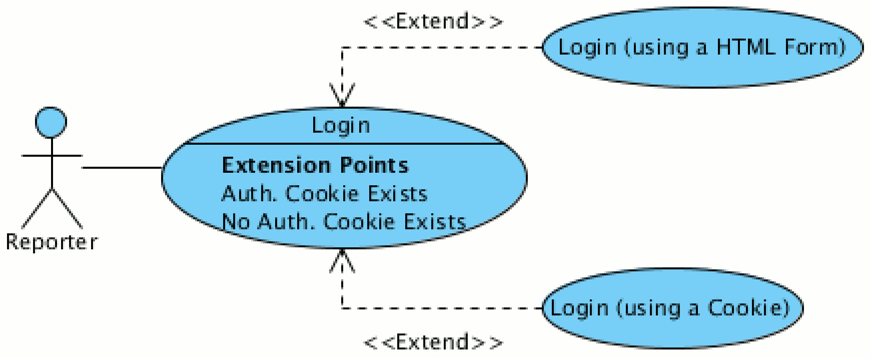
Generalization



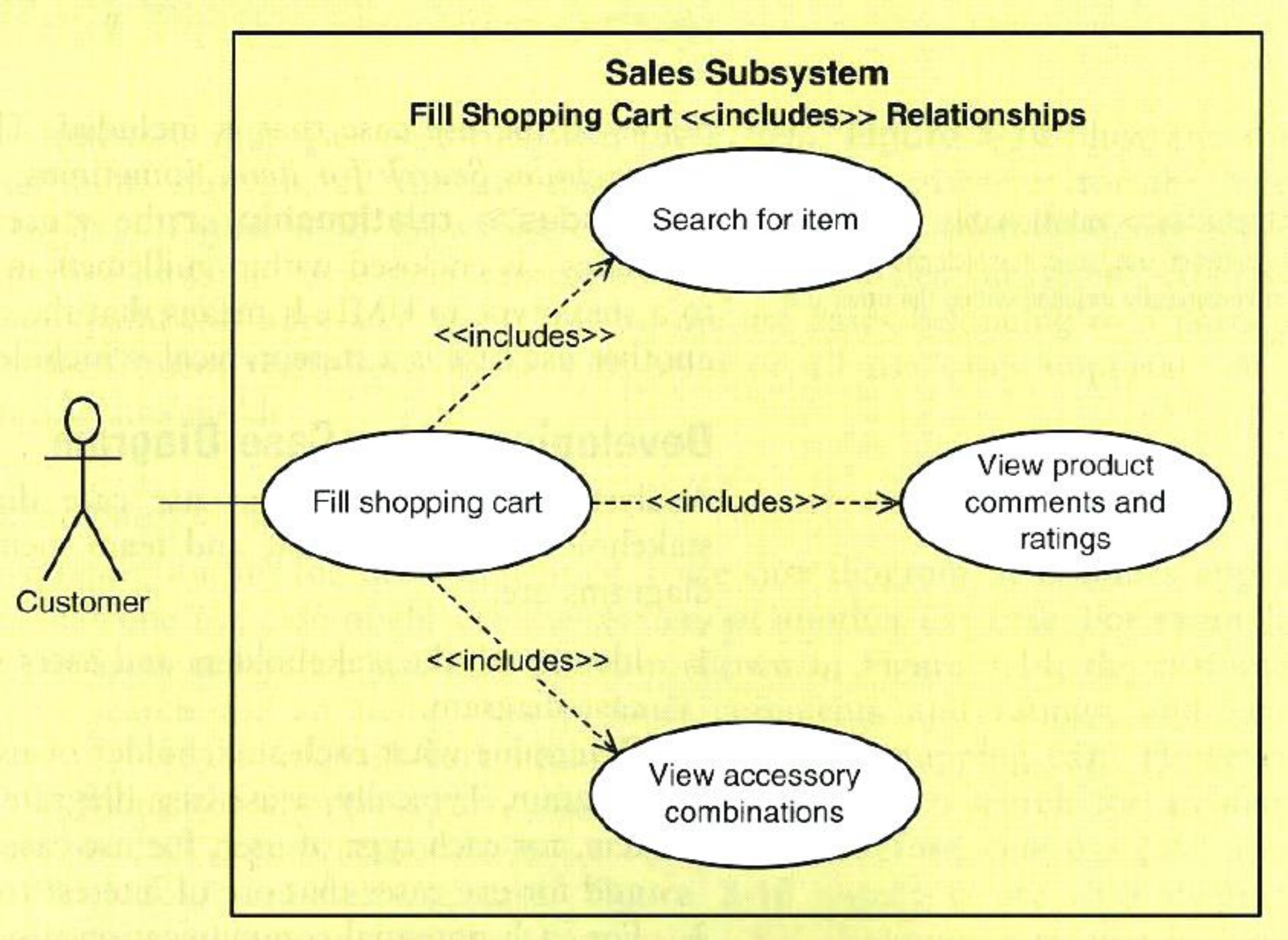
<<include>> relationship

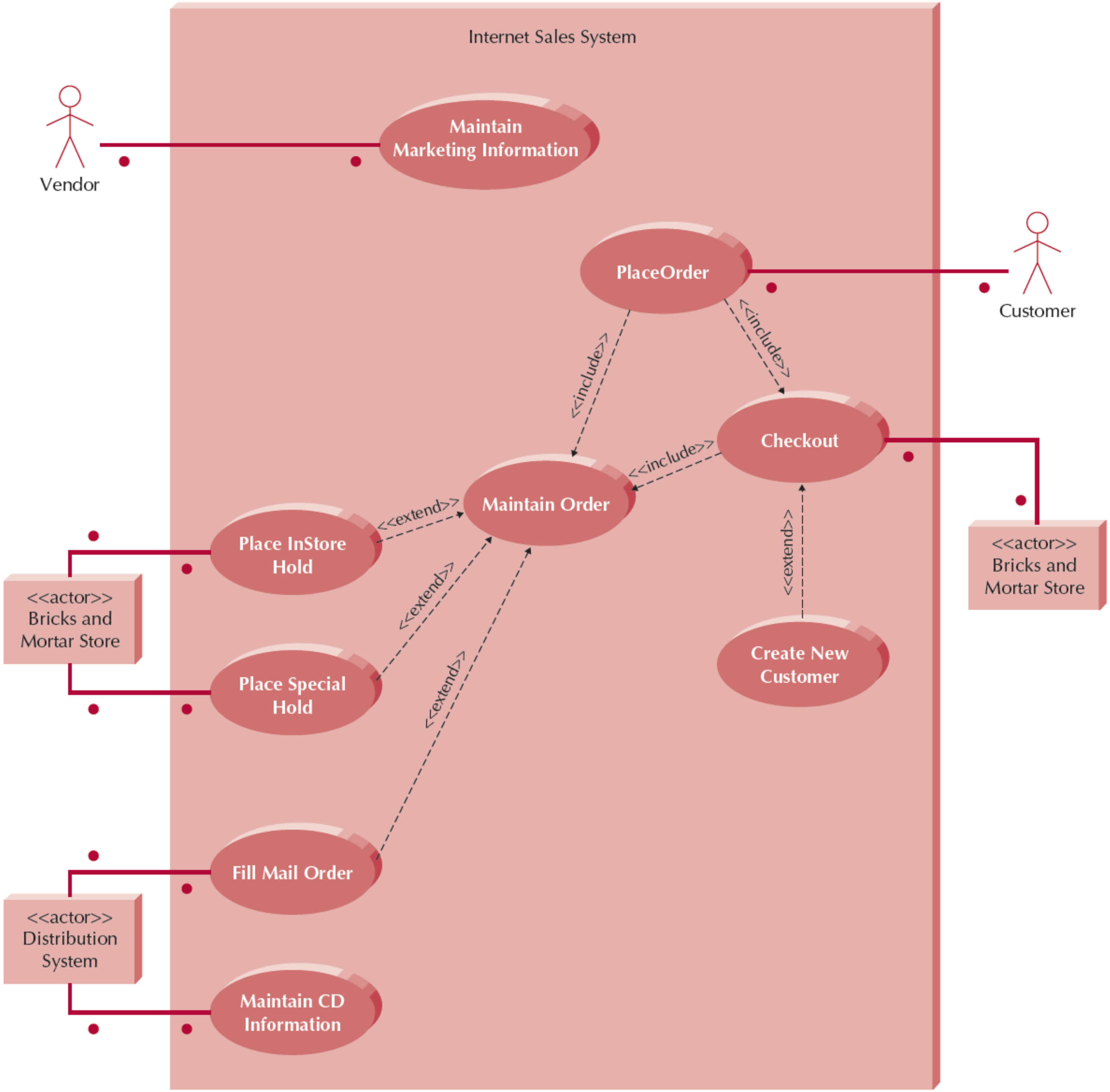


<<extend>> relationship



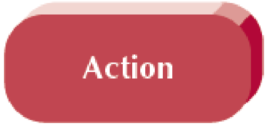
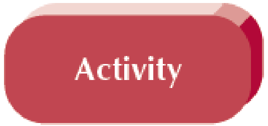
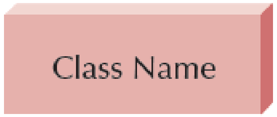
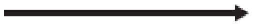
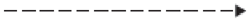
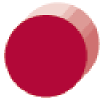
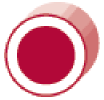
A use case diagram of the Fill shopping


Sales Subsystem
Fill Shopping Cart <<includes>> Relationships
Search for item
<<includes>>
Fill shopping cart
<<includes>>
View product comments and ratings
Customer
<<includes>>
View accessory combinations

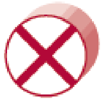
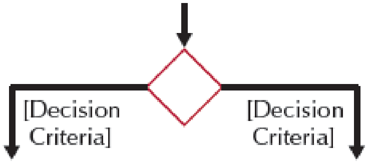
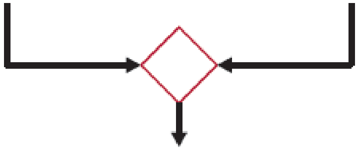
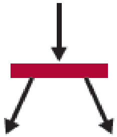
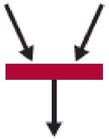
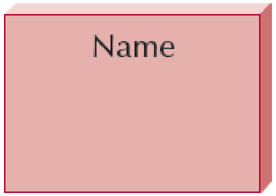

Internet Sales System
Vendor
Maintain Marketing Information
PlaceOrder
Customer
<<include>>
<<include>>
Checkout
<<include>>
Maintain Order
<<extend>>
Place InStore Hold
<<extend>>
Place Special Hold
<<extend>>
Fill Mail Order
Maintain CD Information
<<extend>>
Create New Customer
<<actor>> Bricks and Mortar Store
<<actor>> Bricks and Mortar Store
<<actor>> Distribution System

Activity diagram

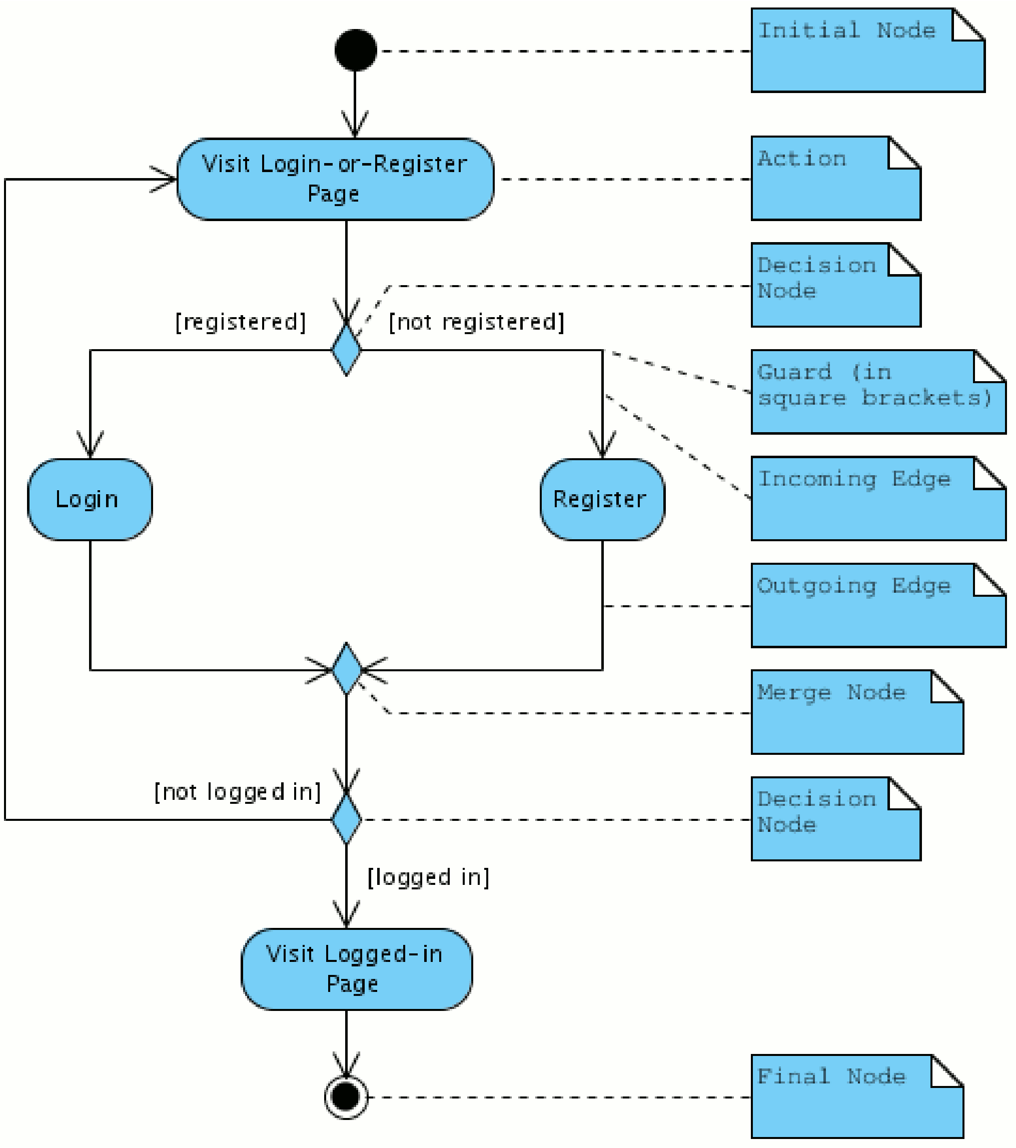
Syntax for an Activity diagram (1)

An Action: ■ Is a simple, non-decomposable piece of behavior ■ Is labeled by its name	
An Activity: ■ Is used to represent a set of actions ■ Is labeled by its name	
An Object Node: ■ Is used to represent an object that is connected to a set of Object Flows ■ Is labeled by its class name	
A Control Flow: ■ Shows the sequence of execution	
An Object Flow: ■ Shows the flow of an object from one activity (or action) to another activity (or action)	
An Initial Node: ■ Portrays the beginning of a set of actions or activities	
A Final-Activity Node: ■ Is used to stop all control flows and object flows in an activity (or action)	

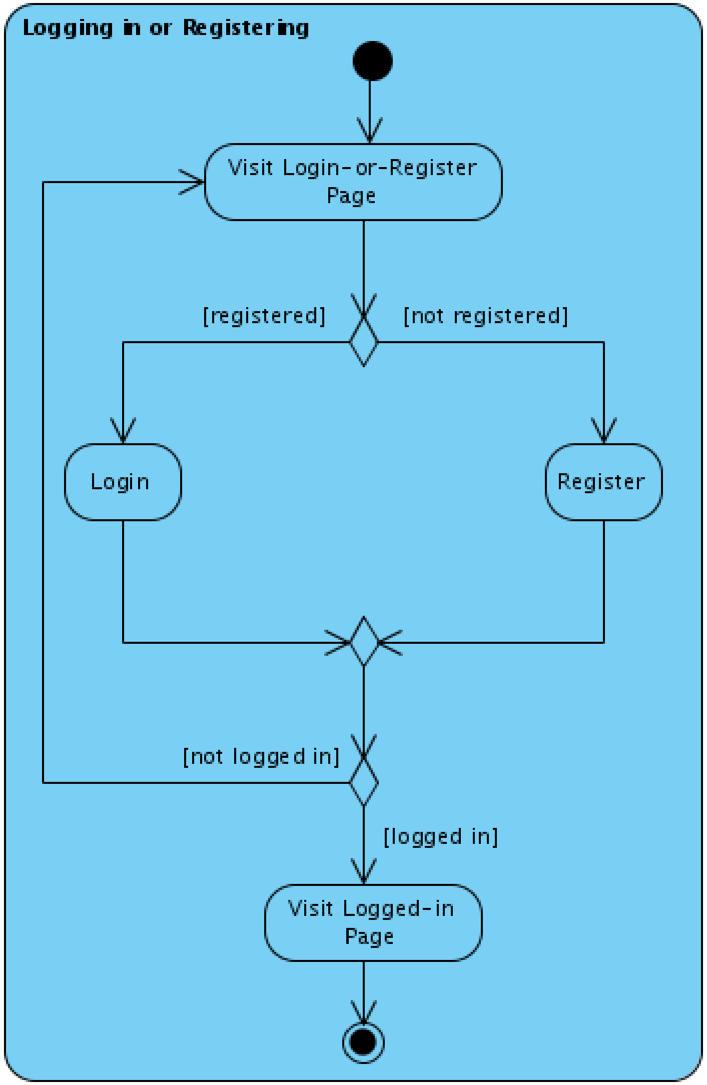
Syntax for an Activity diagram (1)

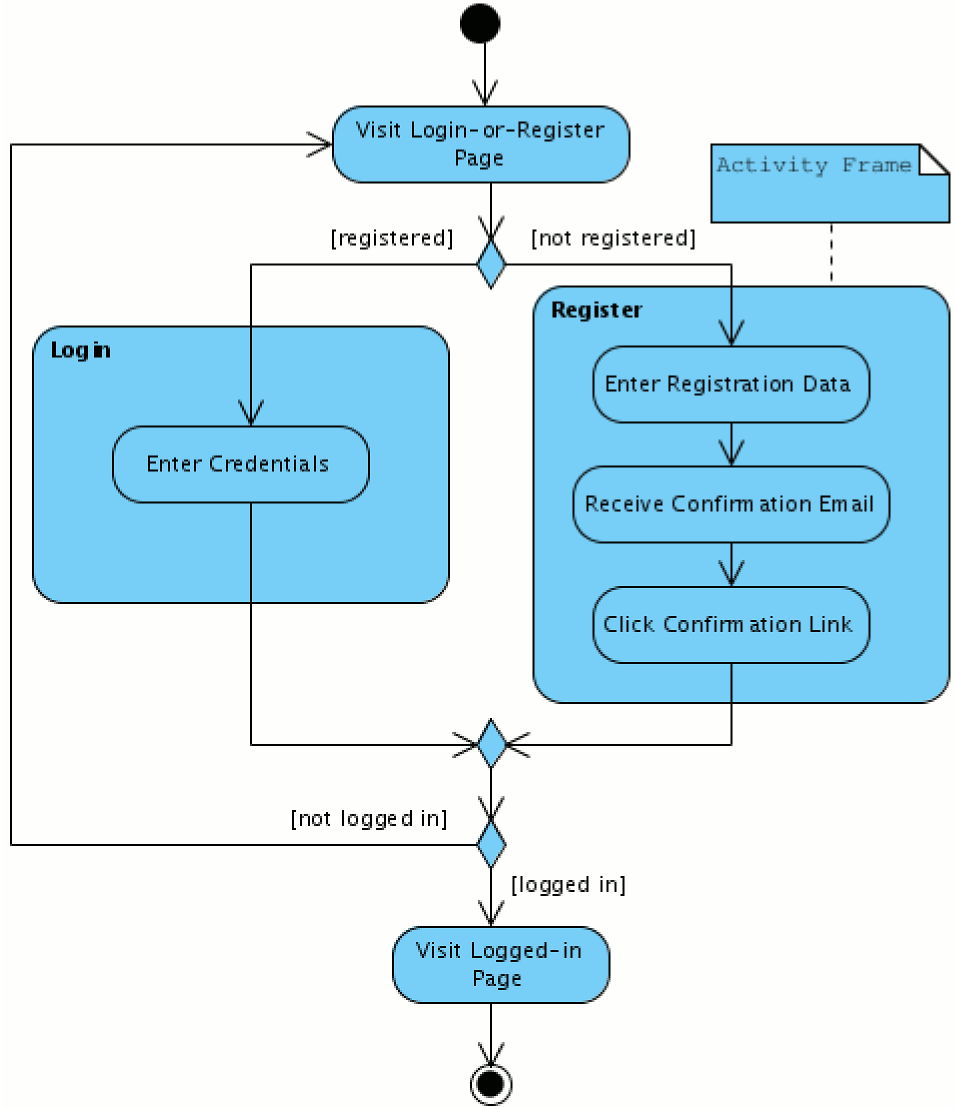
A Final-Flow Node: ■ Is used to stop a specific control flow or object flow	
A Decision Node: ■ Is used to represent a test condition to ensure that the control flow or object flow only goes down one path ■ Is labeled with the decision criteria to continue down the specific path	
A Merge Node: ■ Is used to bring back together different decision paths that were created using a decision-node	
A Fork Node: ■ Is used to split behavior into a set of parallel or concurrent flows of activities (or actions)	
A Join Node: ■ Is used to bring back together a set of parallel or concurrent flows of activities (or actions)	
A Swimlane: ■ Is used to break up an activity diagram into rows and columns to assign the individual activities (or actions) to the individuals or objects that are responsible for executing the activity (or action) ■ Is labeled with the name of the individual or object responsible	

Introduction Activity Diagram

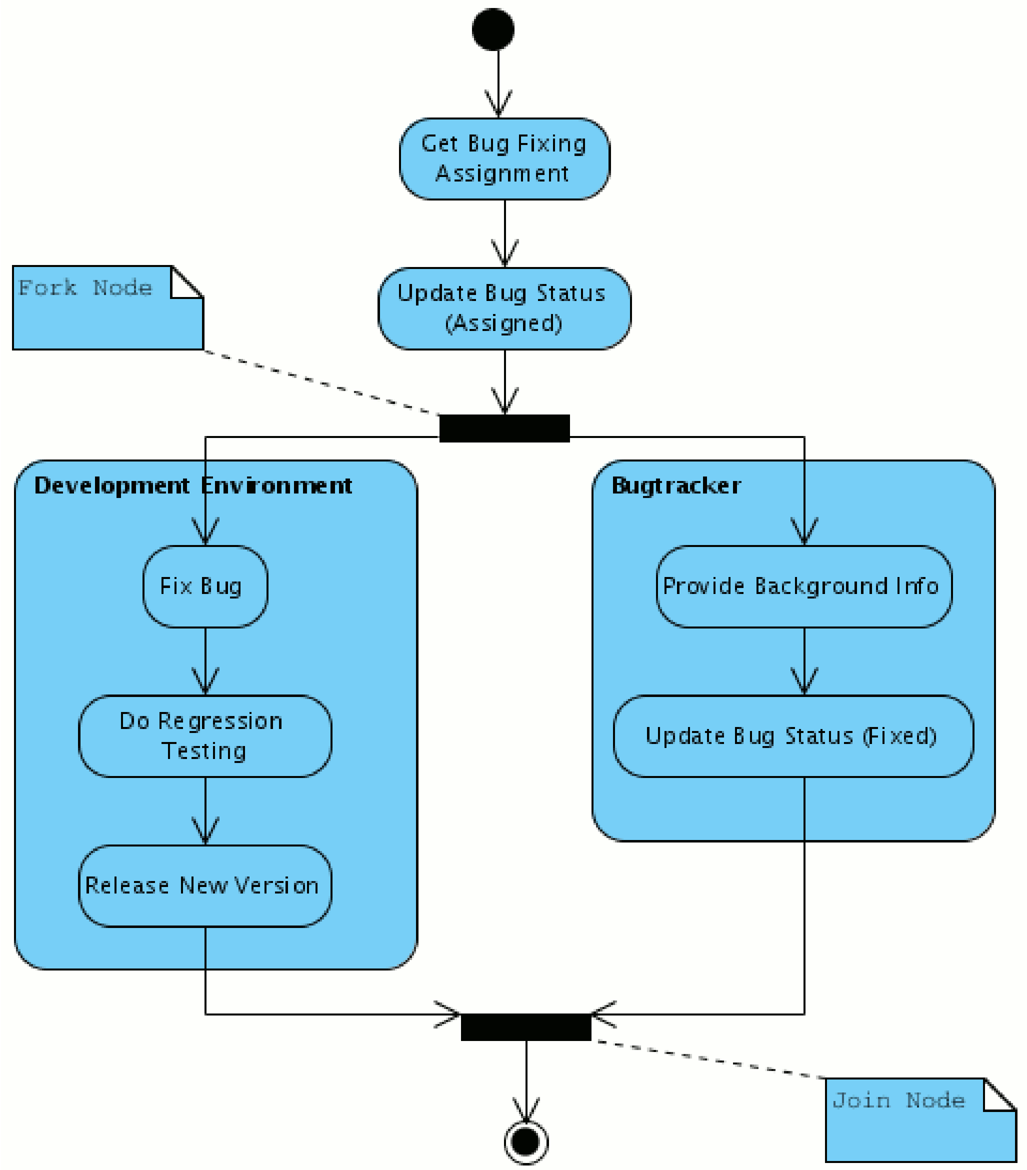


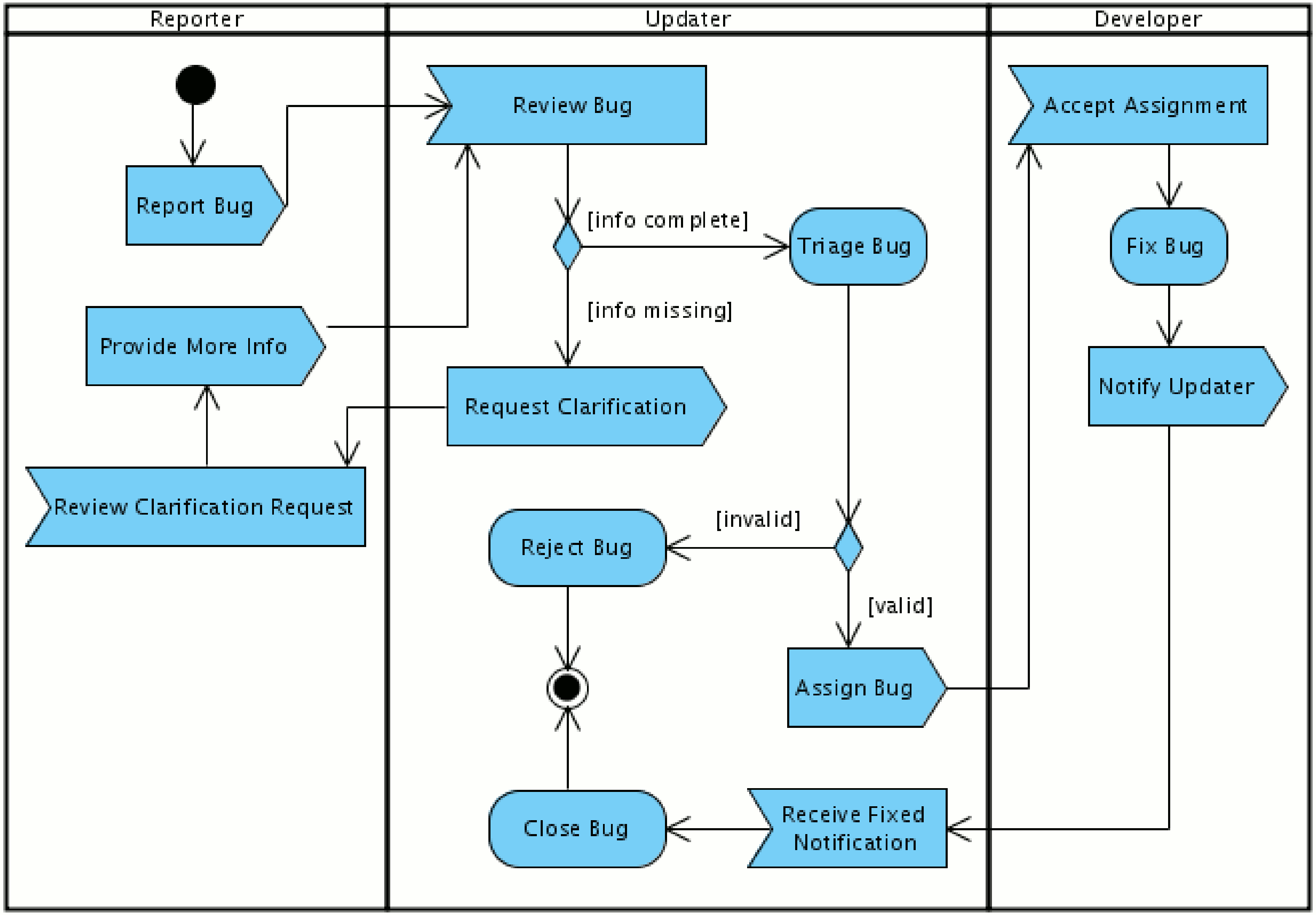
Activity diagram: Action and Activity



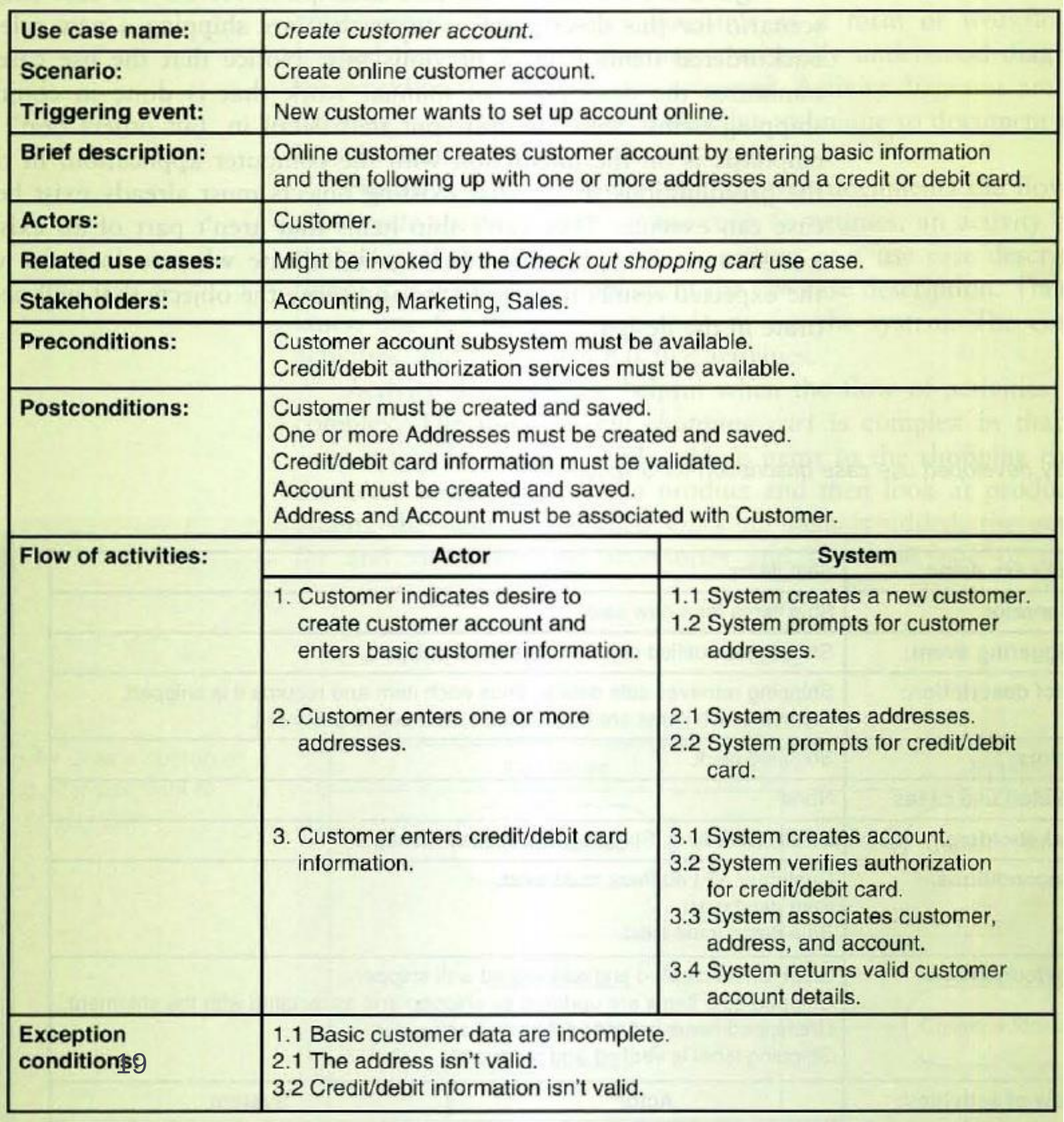


Parallel activity: Fork/Join Node




Reporter
Updater
Developer
Review Bug
Accept Assignment
Report Bug
[info complete]
Triage Bug
Fix Bug
Provide More Info
[info missing]
Request Clarification
Notify Updater
Review Clarification Request
Reject Bug
[invalid]
[valid]
Assign Bug
Close Bug
Receive Fixed Notification

Use case description

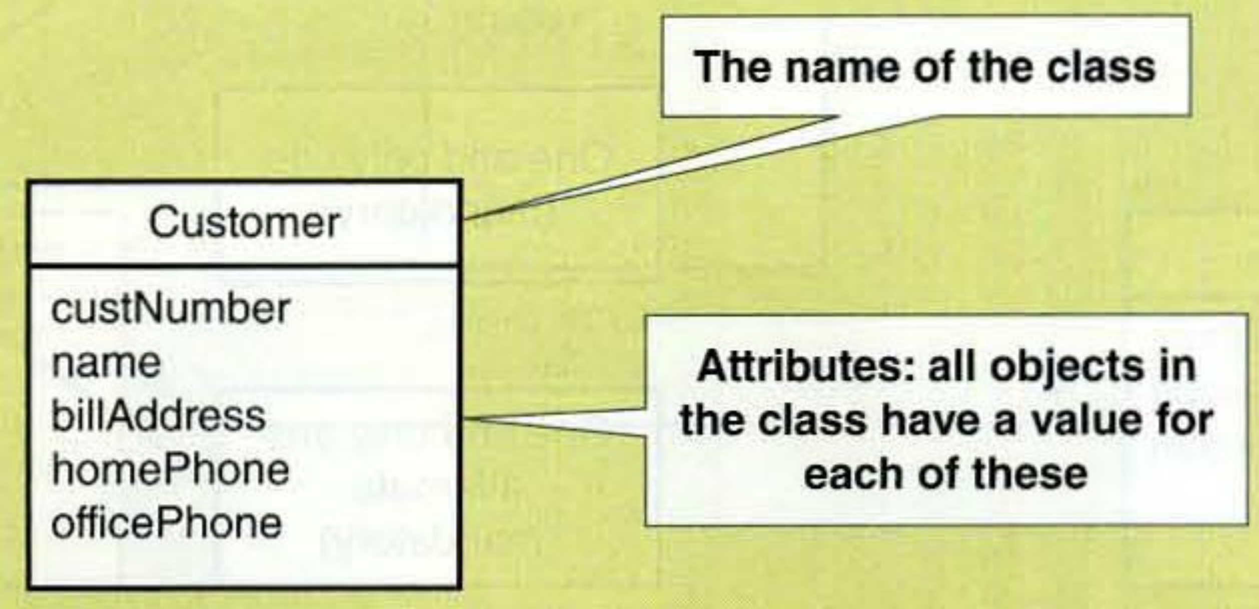


Use case name:	*Create customer account.*	
Scenario:	Create online customer account.	
Triggering event:	New customer wants to set up account online.	
Brief description:	Online customer creates customer account by entering basic information and then following up with one or more addresses and a credit or debit card.	
Actors:	Customer.	
Related use cases:	Might be invoked by the *Check out shopping cart* use case.	
Stakeholders:	Accounting, Marketing, Sales.	
Preconditions:	Customer account subsystem must be available. Credit/debit authorization services must be available.	
Postconditions:	Customer must be created and saved. One or more Addresses must be created and saved. Credit/debit card information must be validated. Account must be created and saved. Address and Account must be associated with Customer.	
Flow of activities:	**Actor**	**System**
	1. Customer indicates desire to create customer account and enters basic customer information.	1.1 System creates a new customer. 1.2 System prompts for customer addresses.
	2. Customer enters one or more addresses.	2.1 System creates addresses. 2.2 System prompts for credit/debit card.
	3. Customer enters credit/debit card information.	3.1 System creates account. 3.2 System verifies authorization for credit/debit card. 3.3 System associates customer, address, and account. 3.4 System returns valid customer account details.
Exception conditions:	1.1 Basic customer data are incomplete. 2.1 The address isn't valid. 3.2 Credit/debit information isn't valid.	

Domain model class diagram

4.3 The Domain Model Class Diagram

- **Class** is category or classification used to describe a collection of object.
  - **Object** is member belongs to a class
- **Domain class** is the classes that describe thing in the problem domain.



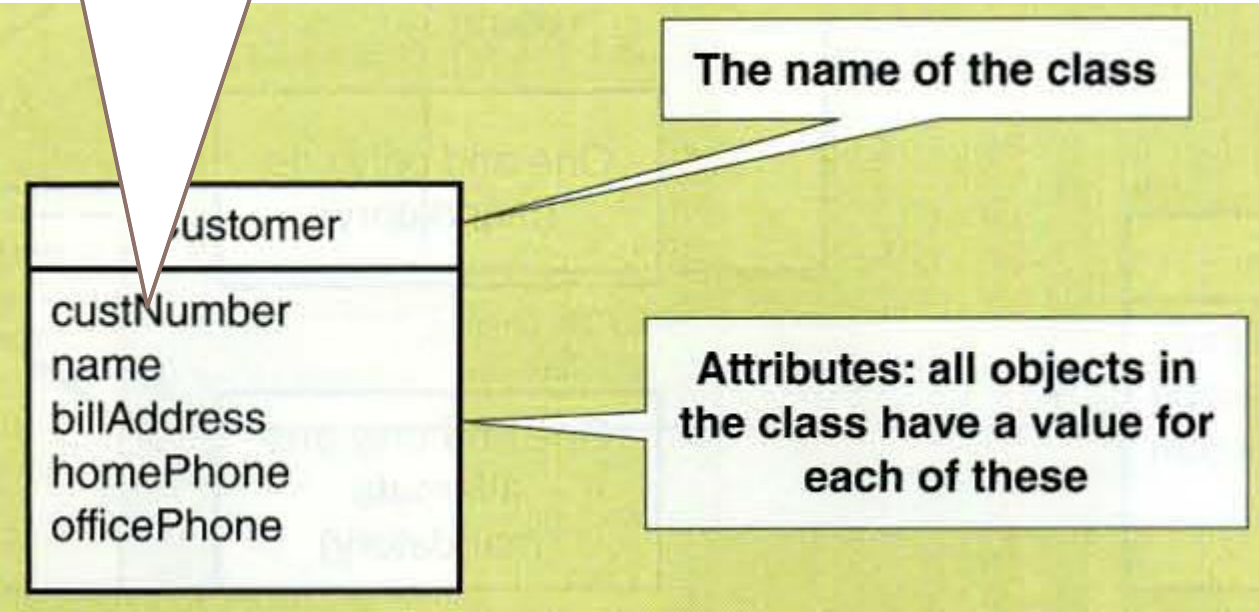
The name of the class

Attributes: all objects in the class have a value for each of these

4.3 The Domain Model Class Diagram

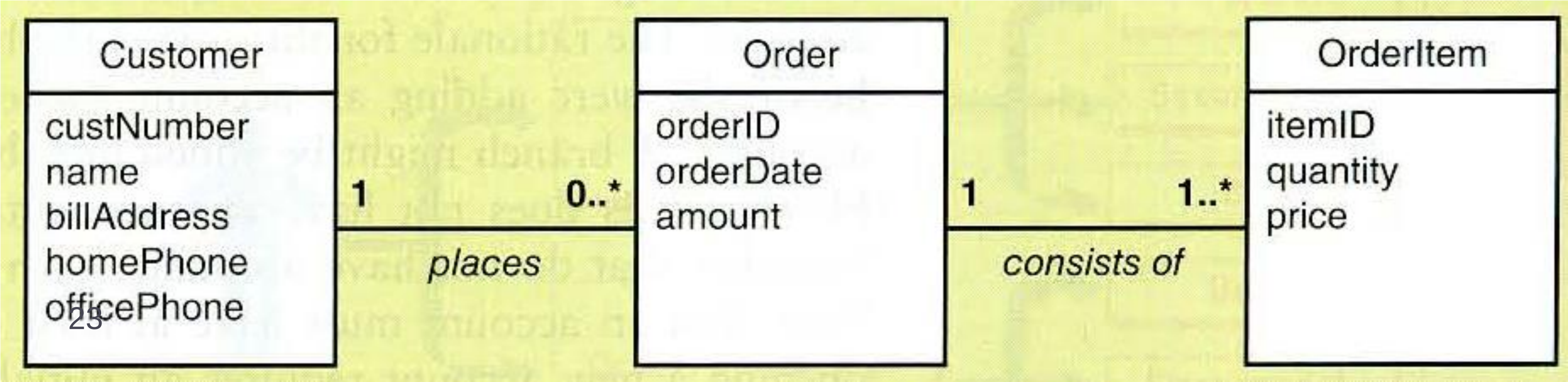
- **Clas** n used to describe a collection of o
  - **O** ass
- **Dor** t describe thing in the problem dom

Camelback notation *or* ***camelCase notation*** *are concatenation words to a single word by the first character of each word typing capitalized.*



4.3 The Domain Model Class Diagram

- **Class diagram (UML)** is used to show class object for a system.
- **Domain model class diagram** is one type of UML class diagram that shows the things in the users 'problem domain.

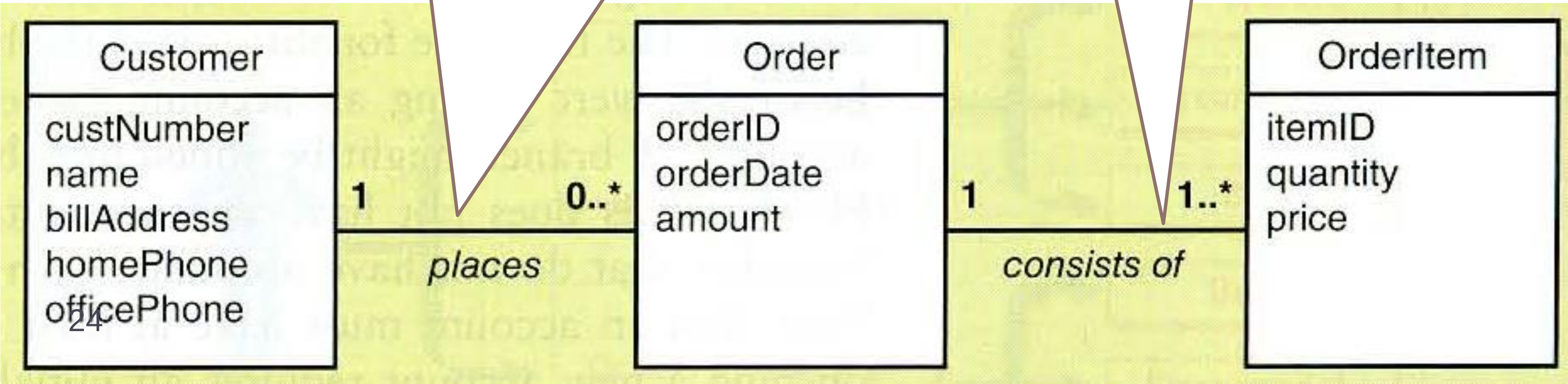


4.3 The Domain Model Class Diagram

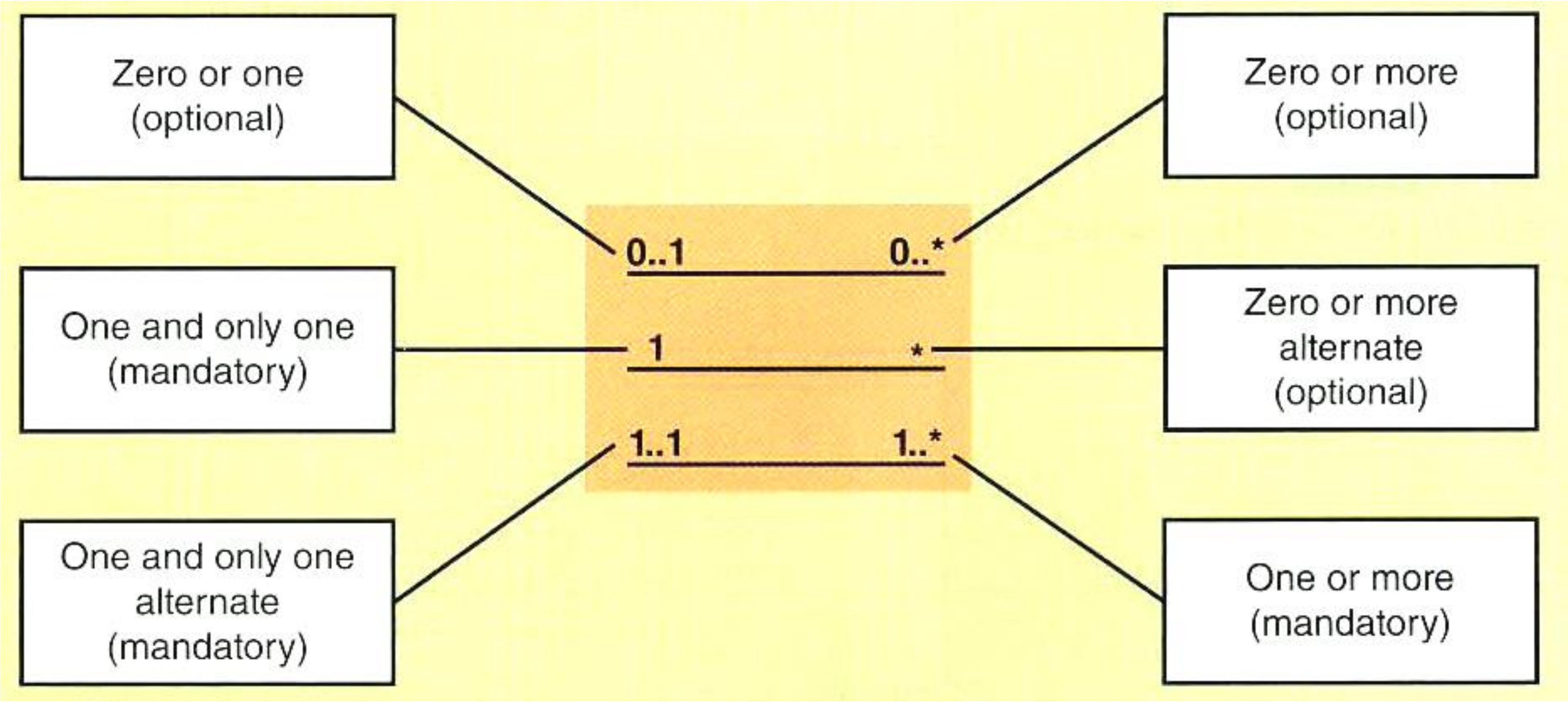
- **Class diagram (UML** a system.
- **Domain model class diagram** is one ty ML class diagram that shows the things in the users m domain.

An order must consist 1 to N Items

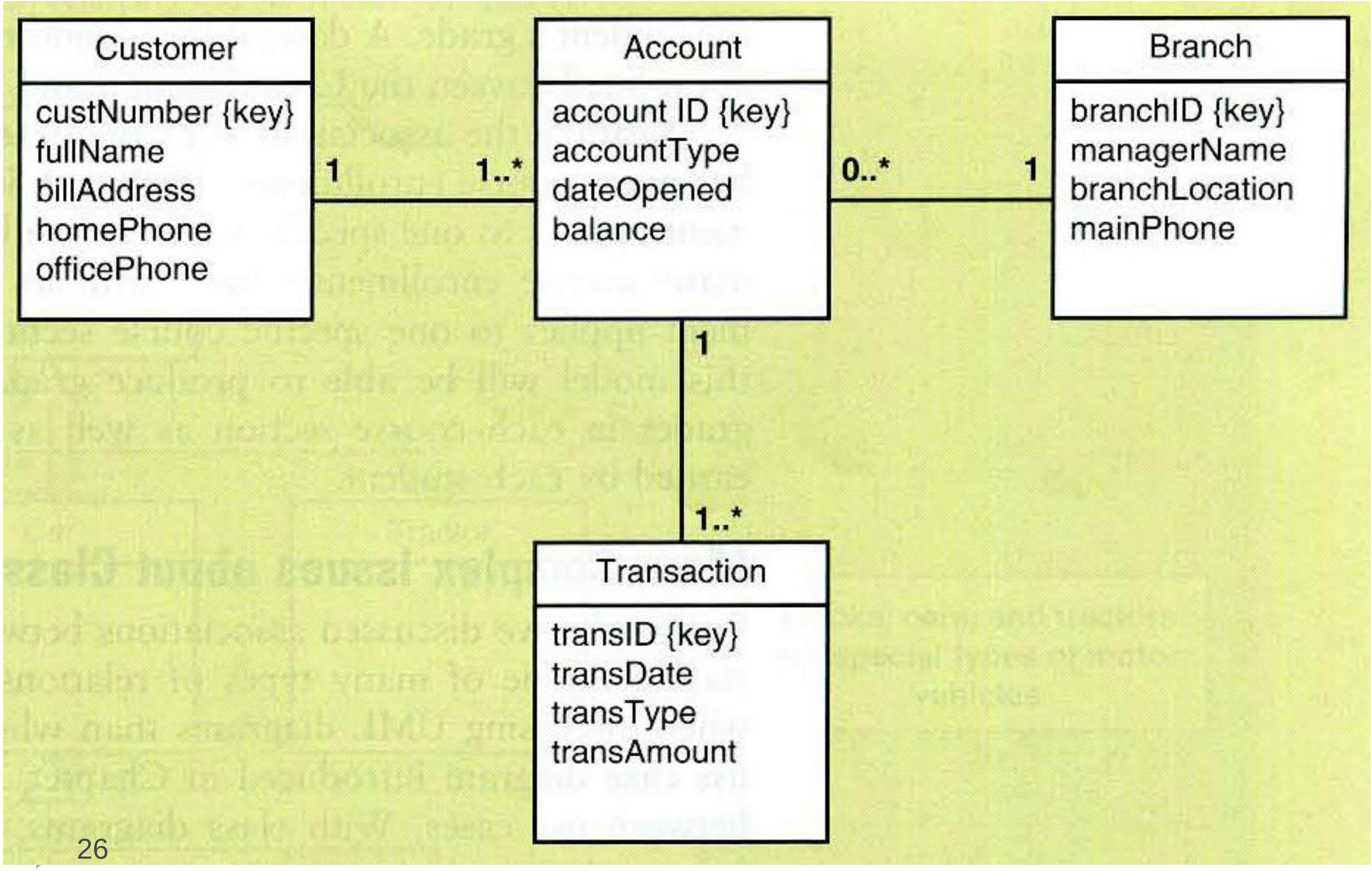
A customer can place 0 to N orders



4.3.1 The Domain Model Class Diagram Notation



4.3.1 The Domain Model Class Diagram Example: A bank system



Customer

custNumber {key}
fullName
billAddress
homePhone
officePhone

Account

account ID {key}
accountType
dateOpened
balance

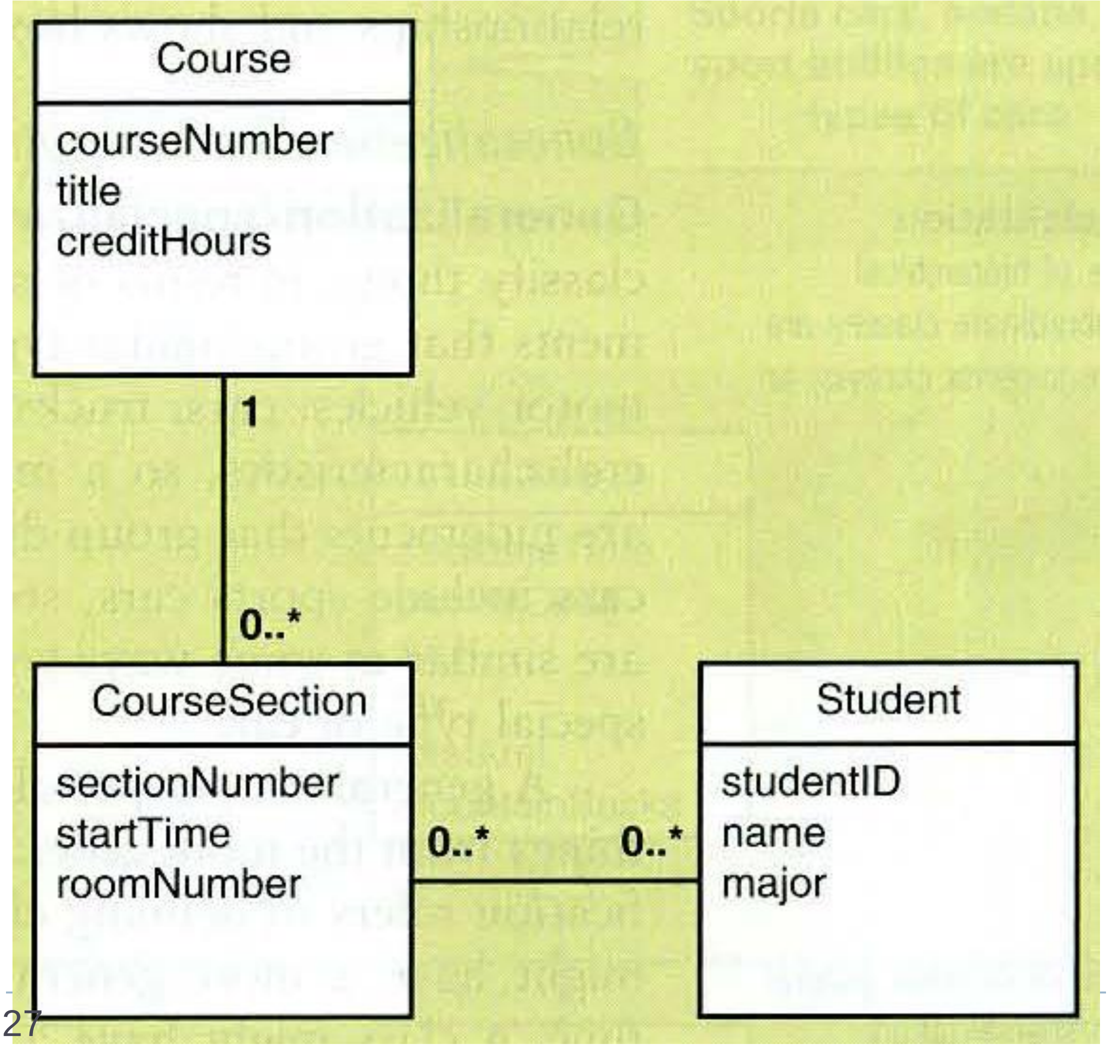
Branch

branchID {key}
managerName
branchLocation
mainPhone

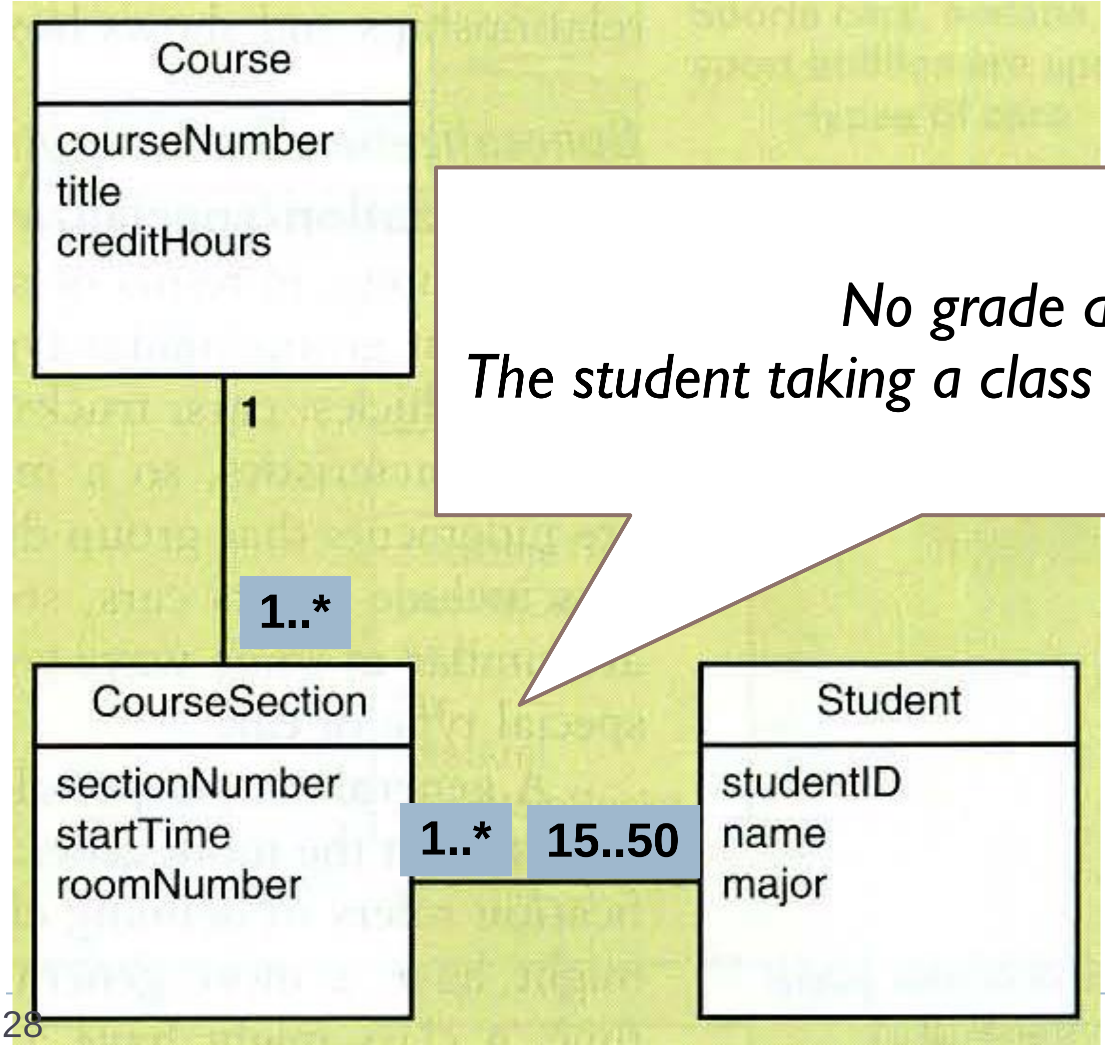
Transaction

transID {key}
transDate
transType
transAmount

4.3.1 The Domain Model Class Diagram Example: A university course enrollment



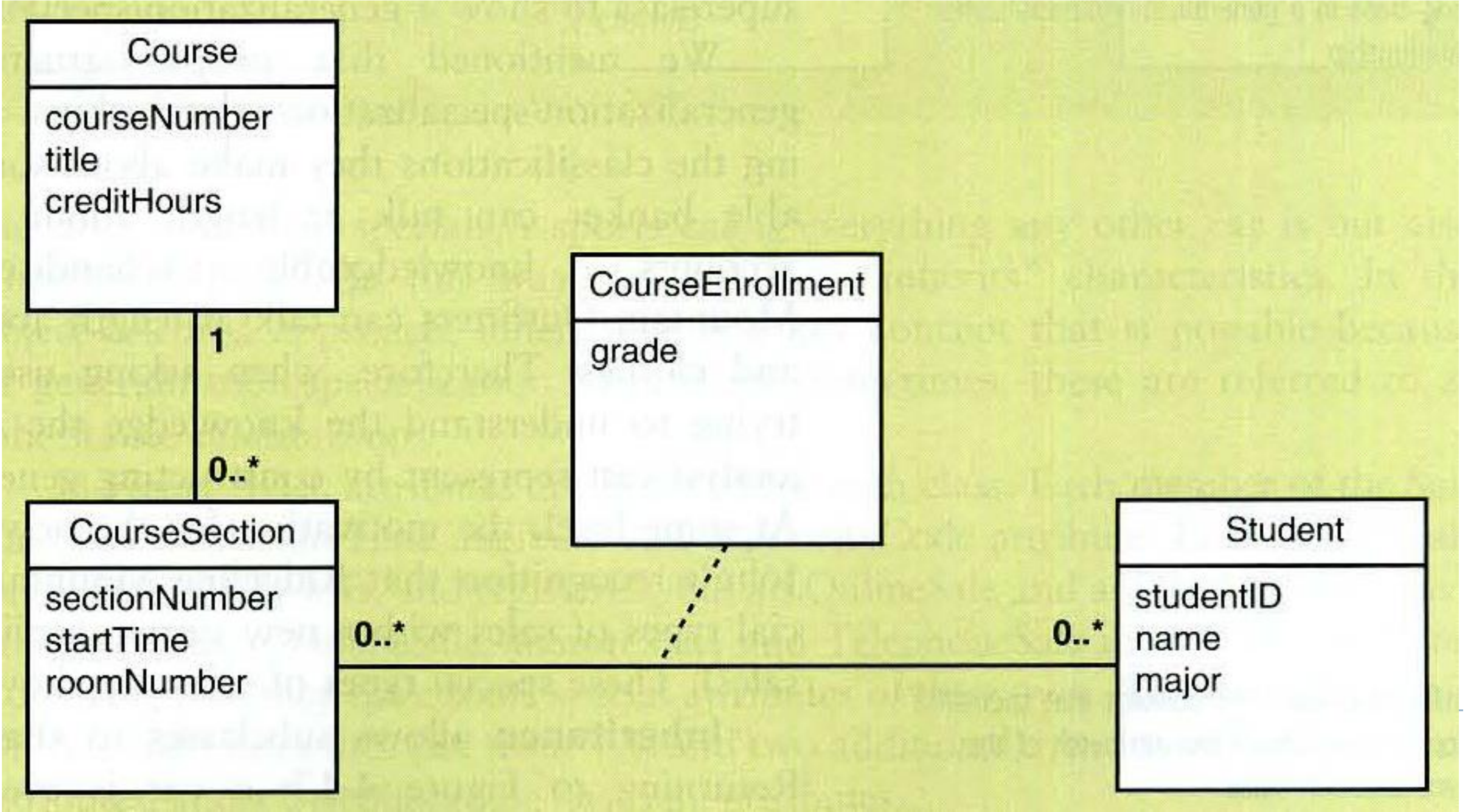
4.3.1 The Domain Model Class Diagram Example: A university course enrollment



No grade data,
The student taking a class don't have a grade.

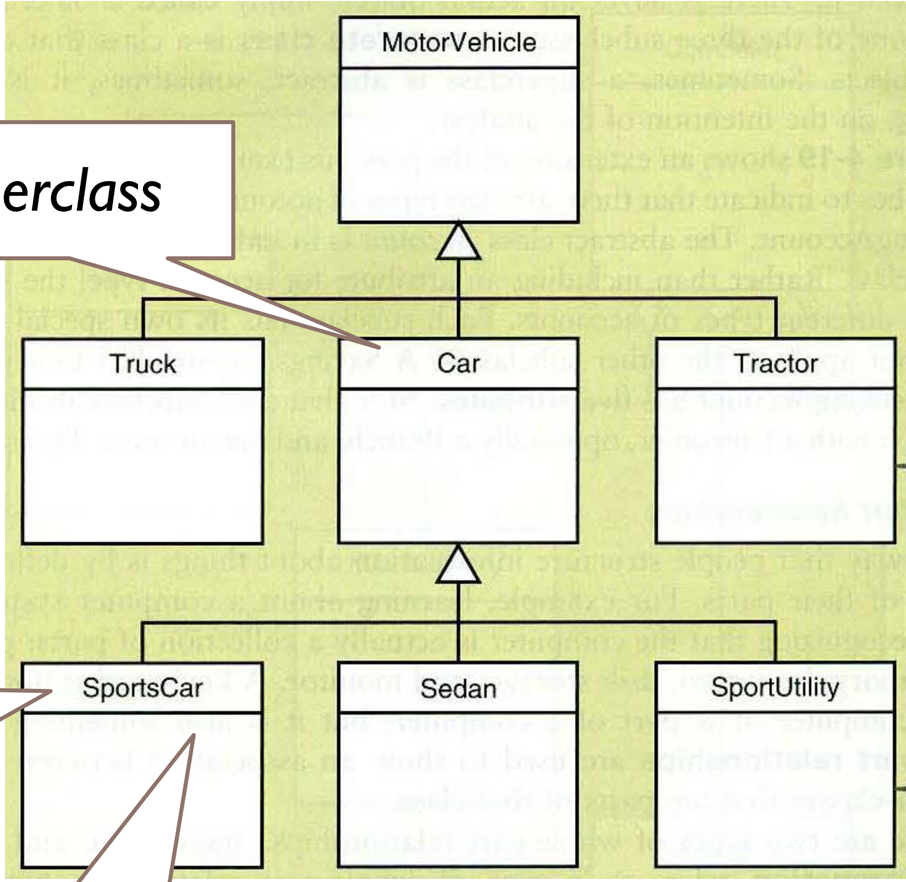
4.3.1 The Domain Model Class Diagram Example: A university course enrollment (2)

- Association class
  - The solution is to add a domain class represent association between two classes.



4.3.2 More complex issue about classes of objects

- Generalization relationship
  - Group similar types of things
- Specialization relationship
  - Group different types of things



Superclass

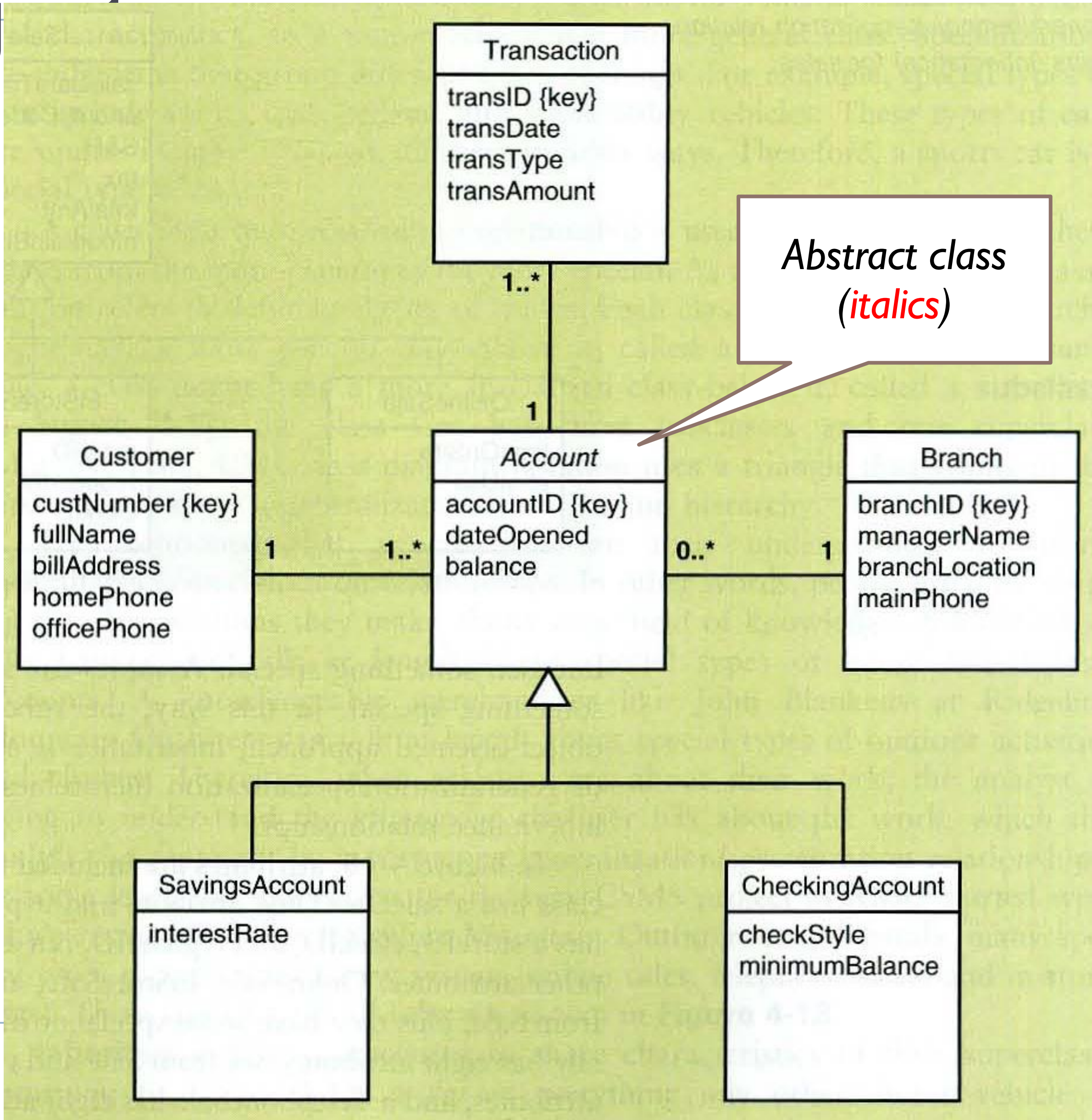
Specialize,
Sport car is different from tractor and truck, is car.

Subclass

4.3.2 More complex issue about classes of obj

- **Abstract class** is a class that subclass can inherit from it.

- **Concrete class** is a class that does have actual object.



Transaction
transID {key}
transDate
transType
transAmount

Customer
custNumber {key}
fullName
billAddress
homePhone
officePhone

Account
accountID {key}
dateOpened
balance

Branch
branchID {key}
managerName
branchLocation
mainPhone

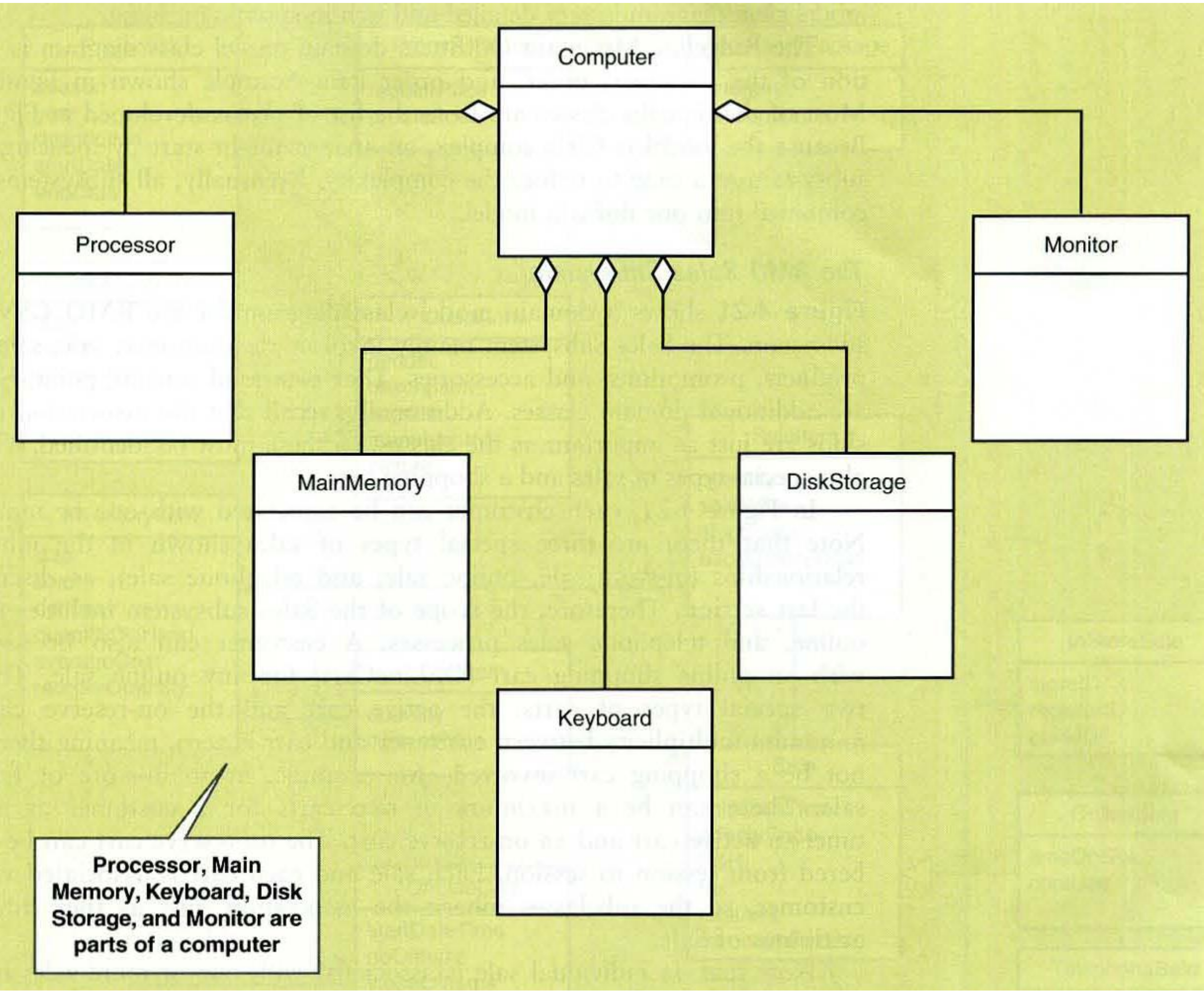
SavingsAccount
interestRate

CheckingAccount
checkStyle
minimumBalance

Abstract class (italics)

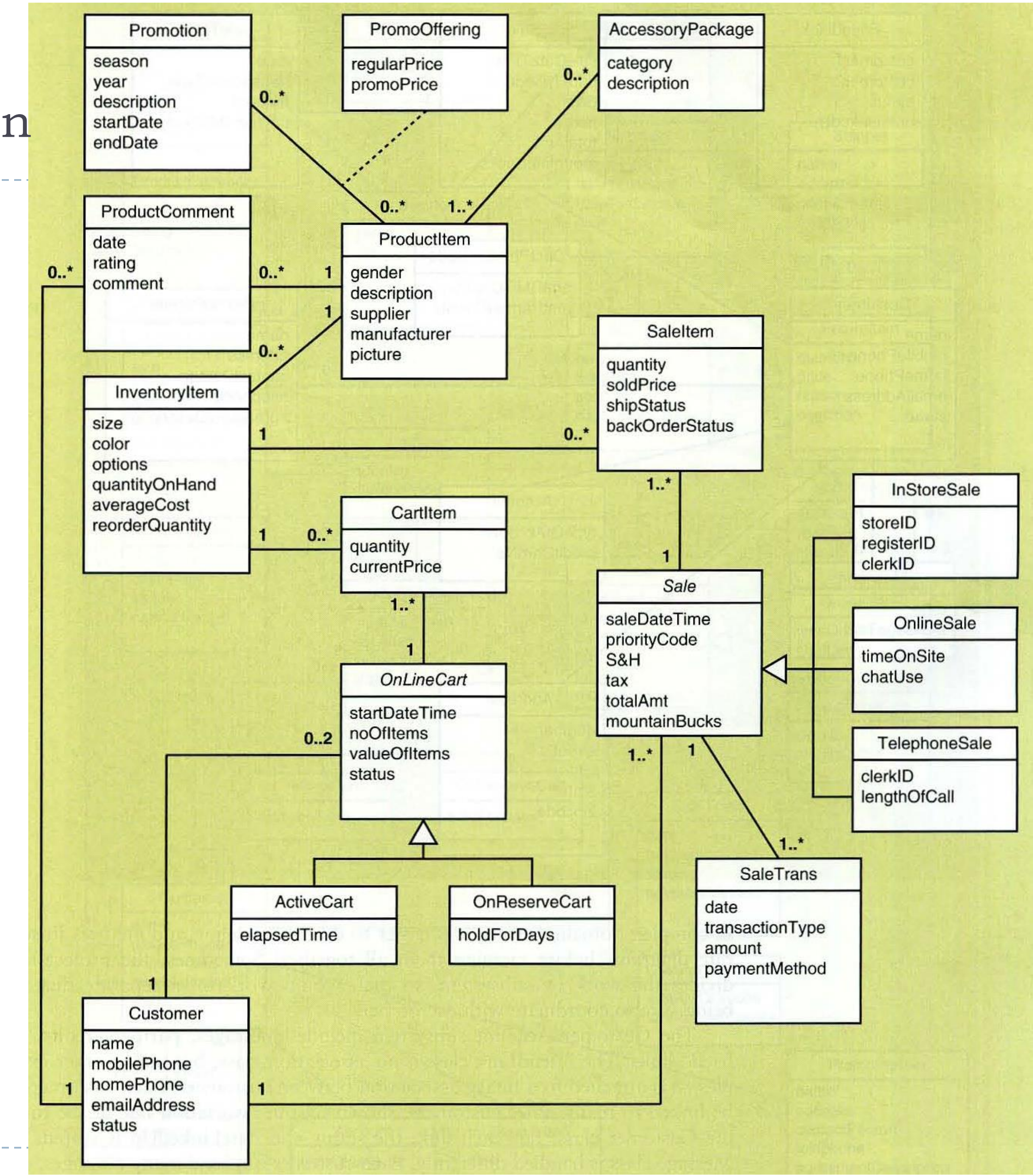
4.3.2 More complex issue about classes of objects

- Whole part relationship
  - Aggregation
  - Composition

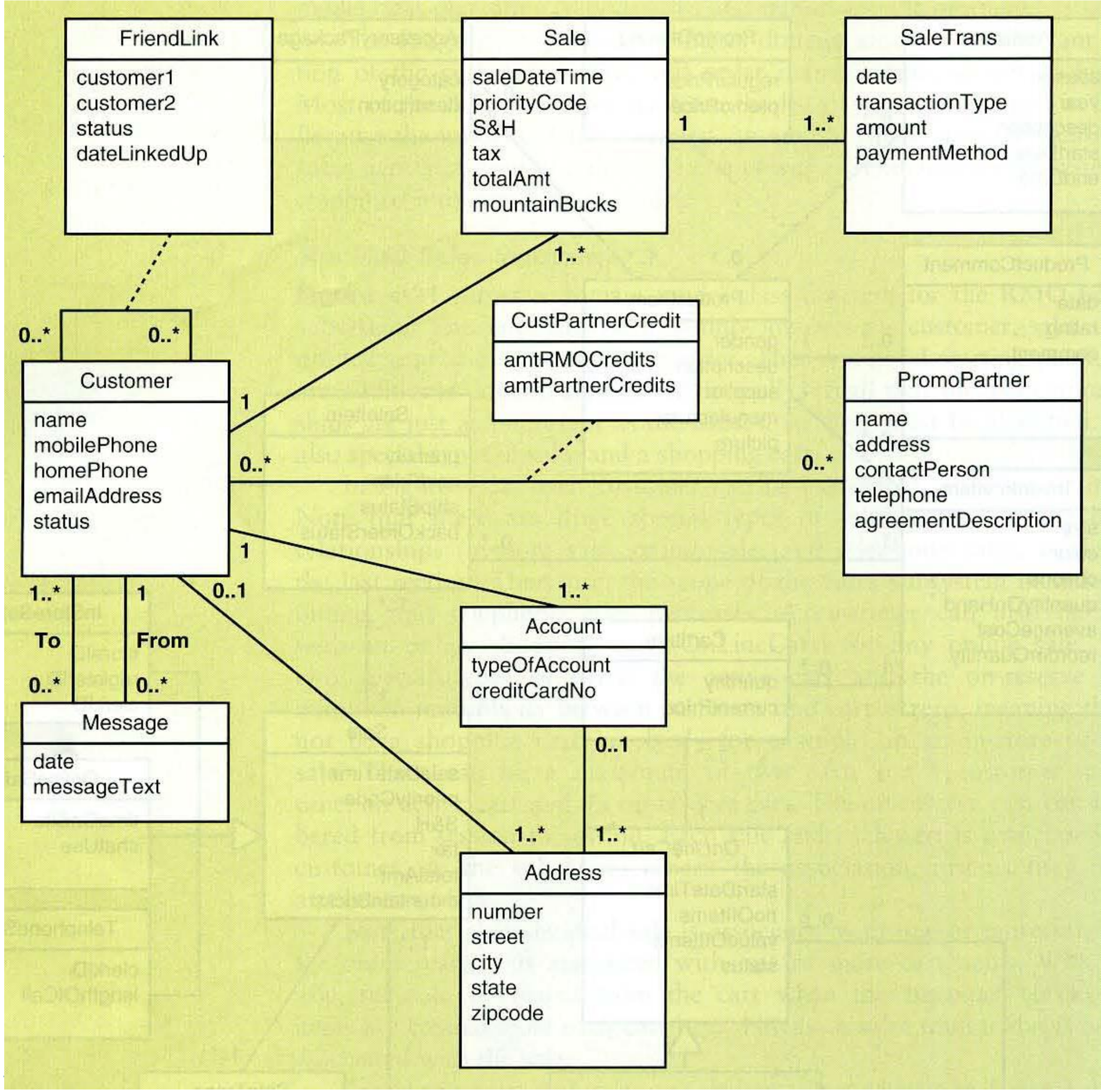


Processor, Main Memory, Keyboard, Disk Storage, and Monitor are parts of a computer

4.3.3 RMO Example: Domain Model Class Diagram (Sales subsystem)



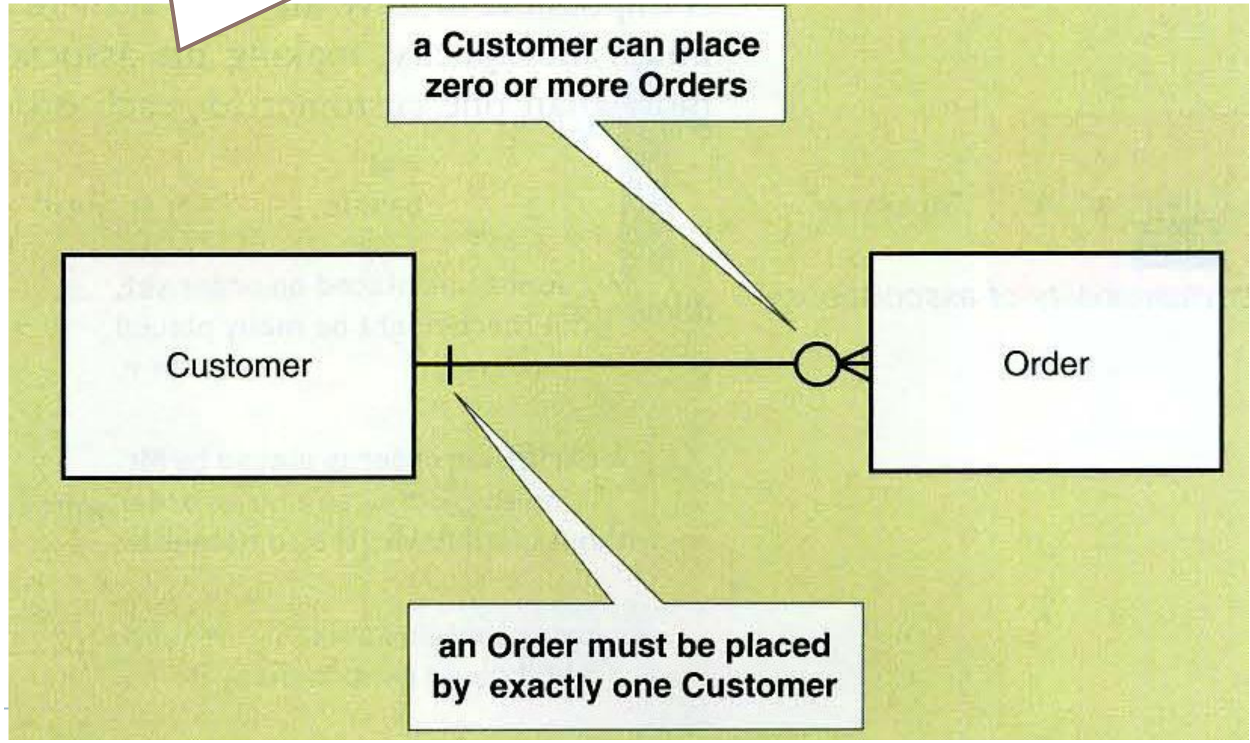
4.3.3 RMO Example: Domain Model Class Diagram (Customer account subsystem)



Entity Relationship Diagram

4.2.1 ERD Notation

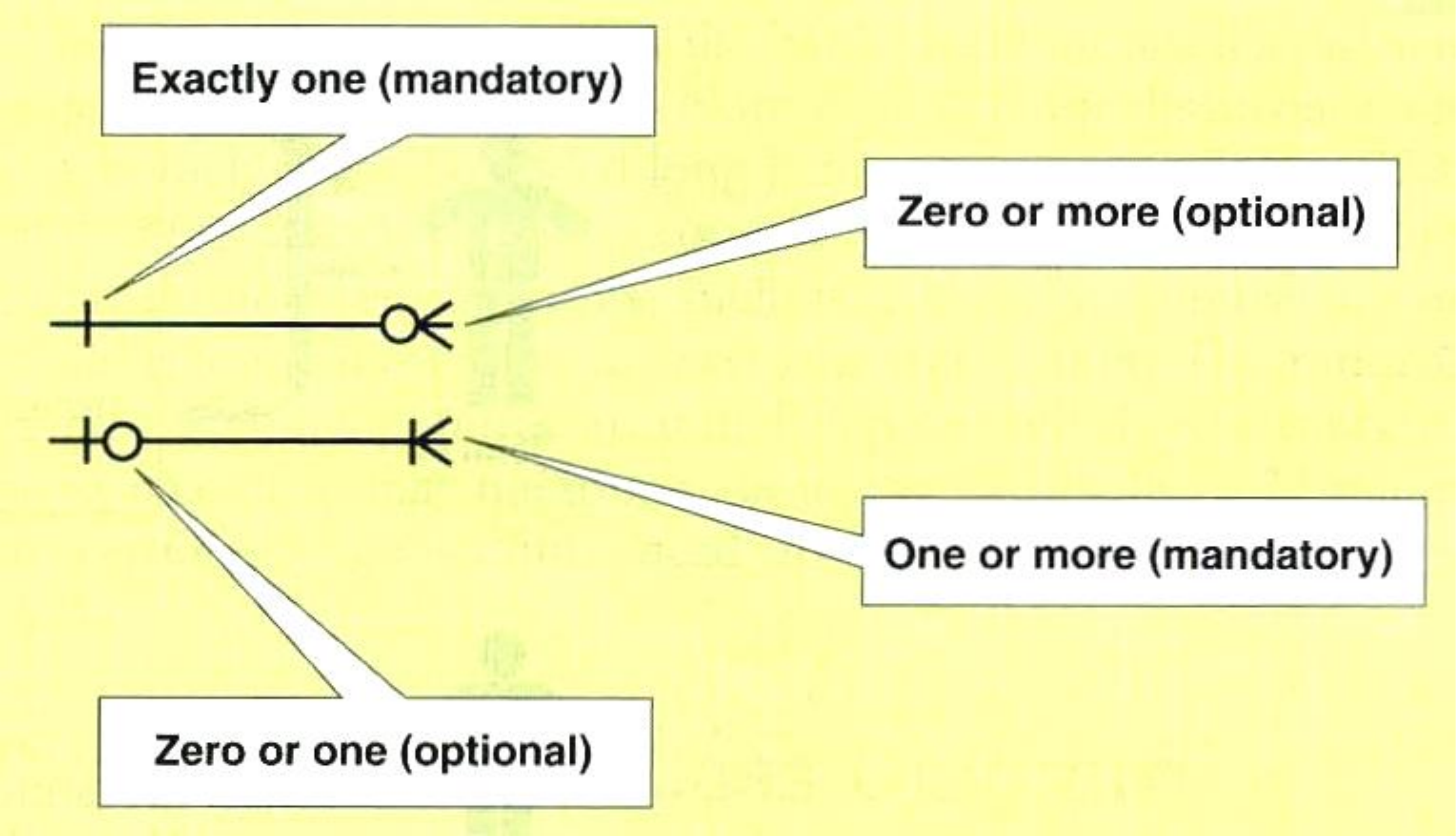
Two data entities (Customer and Order)



a Customer can place zero or more Orders

an Order must be placed by exactly one Customer

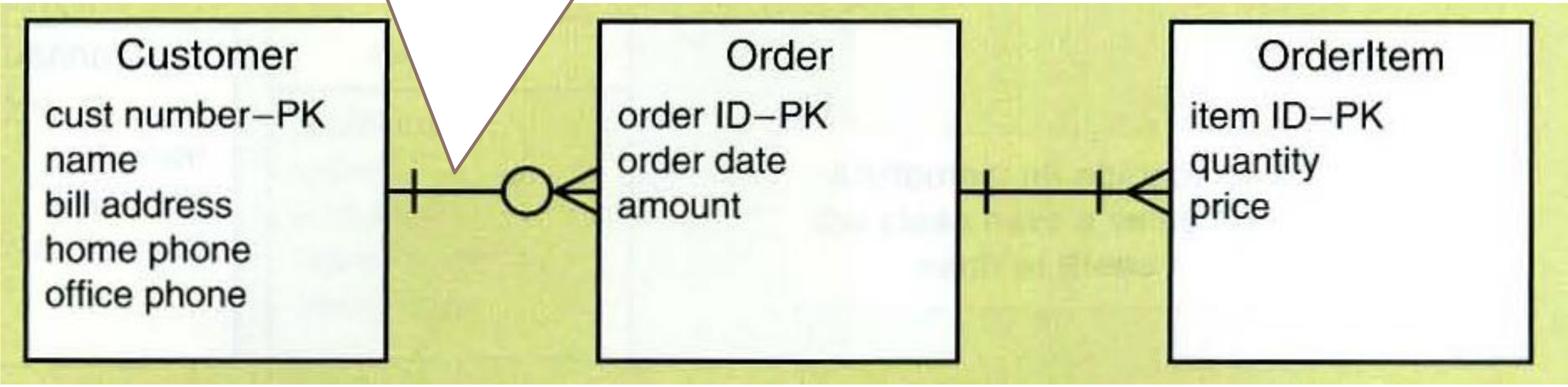
4.2.1 ERD Notation



4.2.1 ERD Notation: Example

- ERD with attributes shown

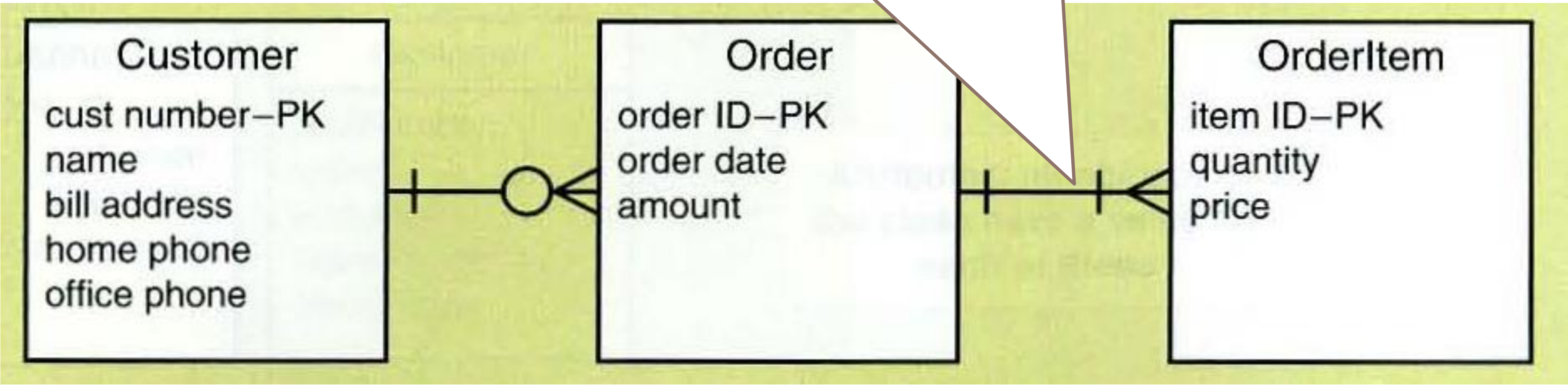
A customer takes many orders.
A customer don’t take an order.



4.2.1 ERD Notation: Example

- ERD with attributes shown

An order has minimum one item.
An order has many items.



Customer

cust number–PK
name
bill address
home phone
office phone

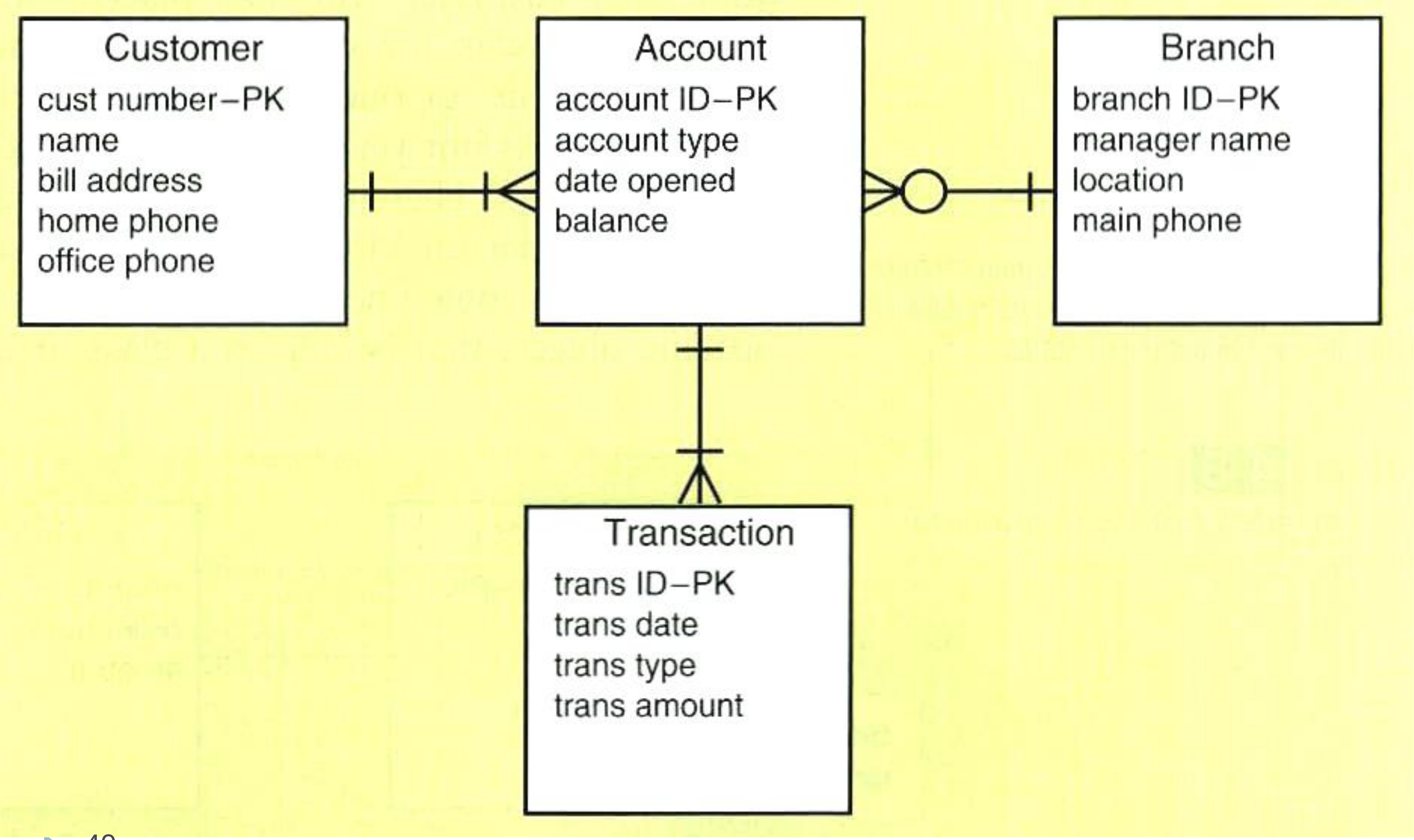
Order

order ID–PK
order date
amount

OrderItem

item ID–PK
quantity
price

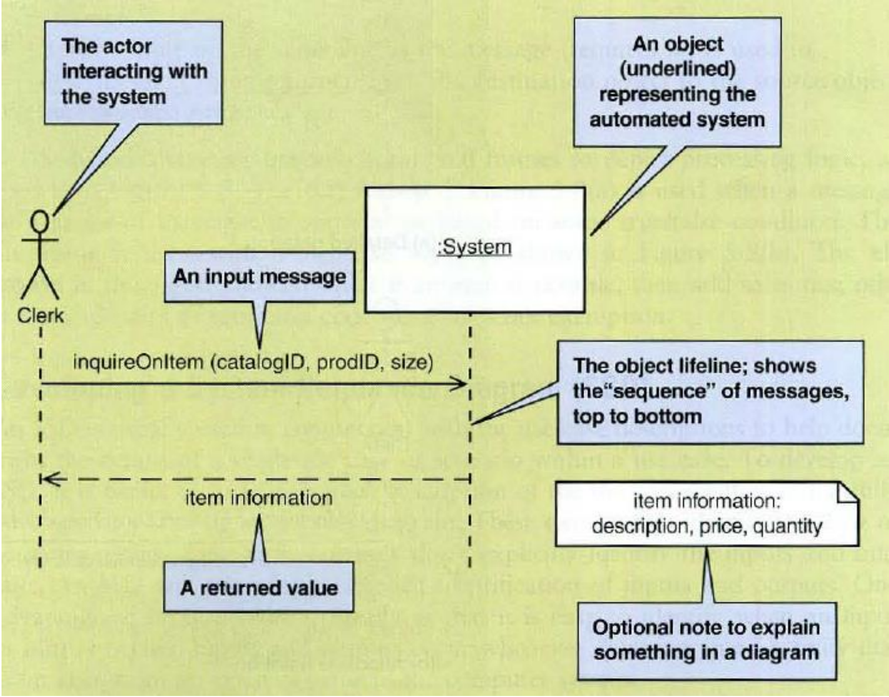
4.2.1 ERD Notation: Example many branch of bank



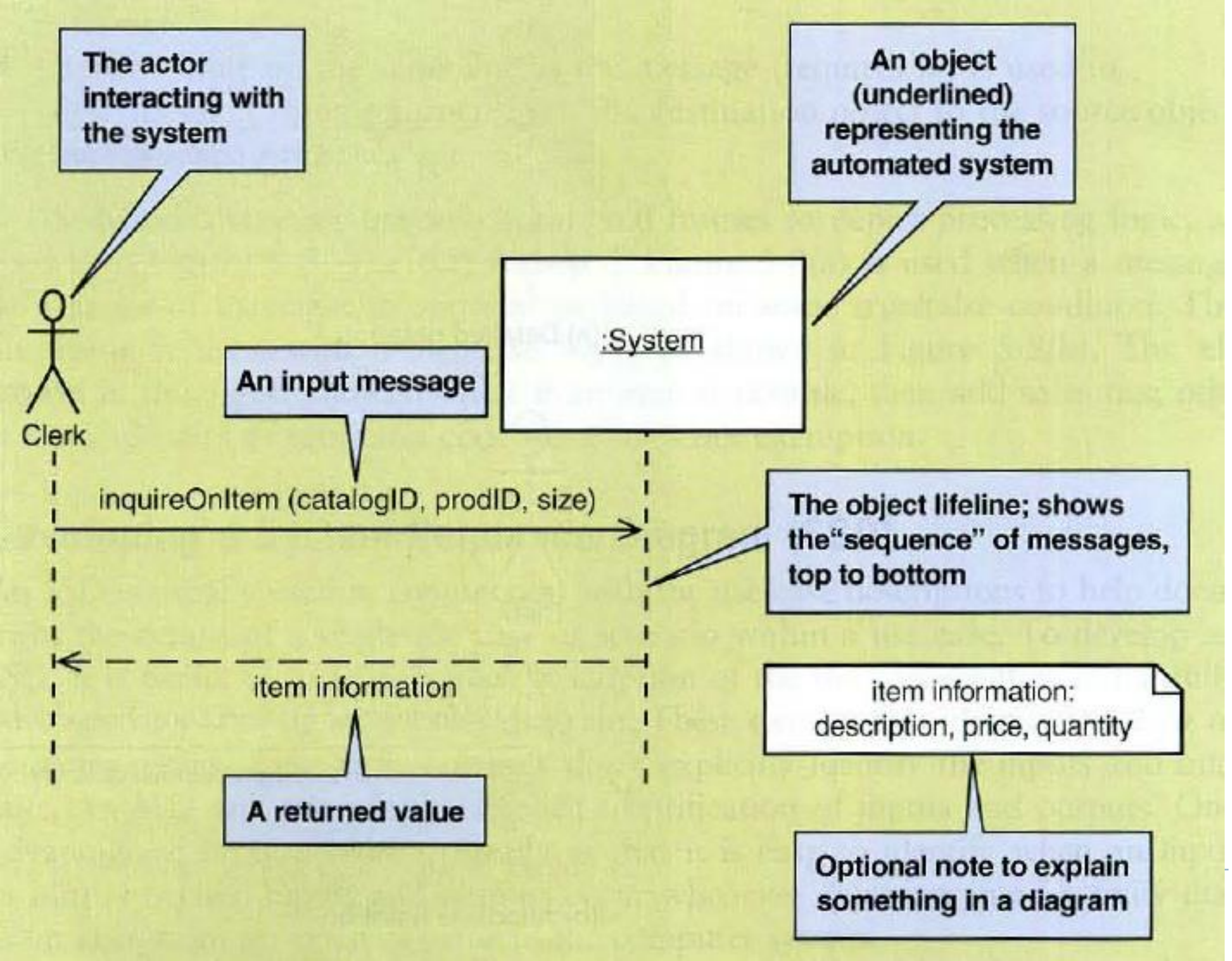
System sequence diagram

5.3 SSD Indentifying I/O

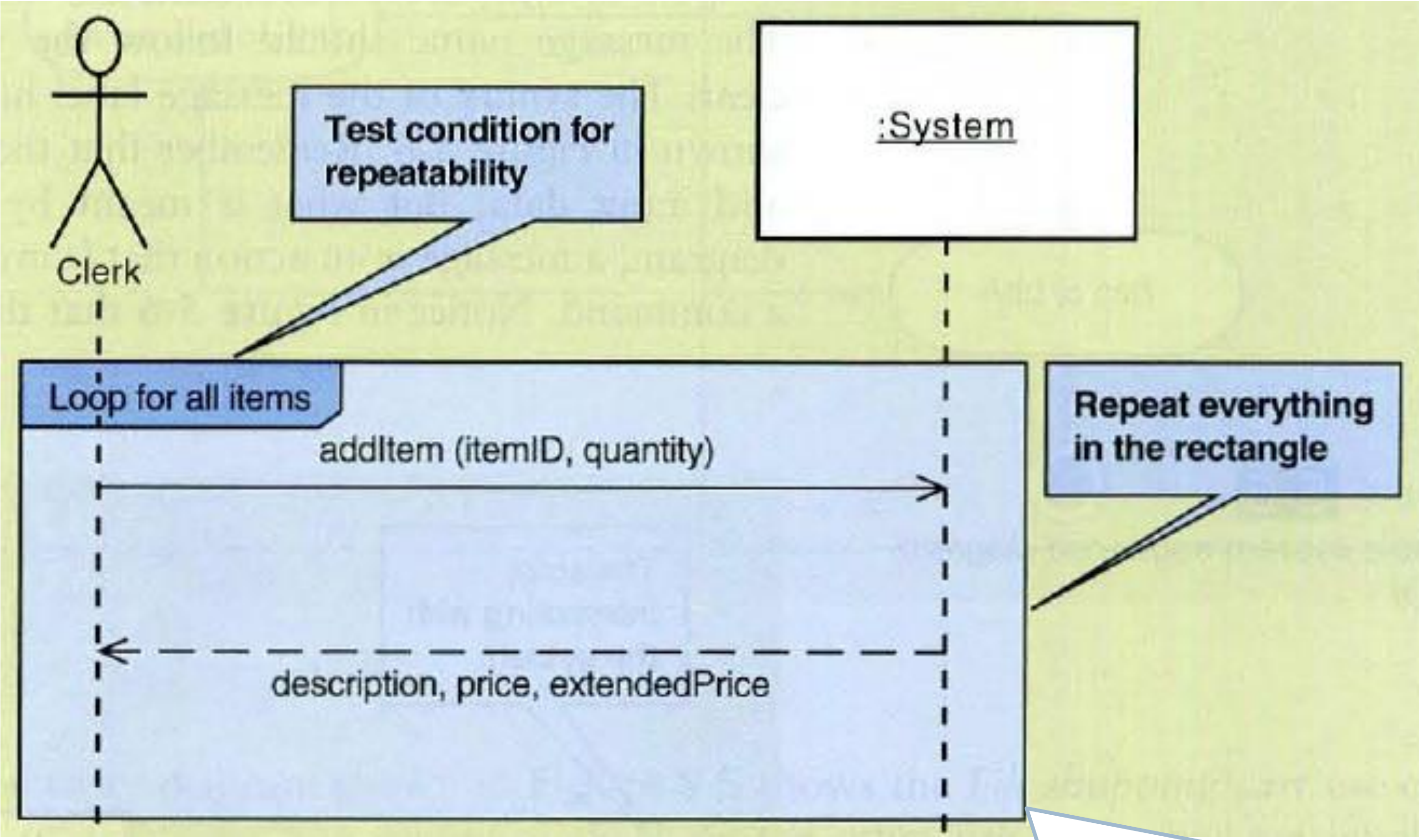
- System Sequence Diagram (SSD)
  - Uses to describe the flow of information into and out of the automatic system.
  - Show the sequence message inform diagram between an external actor and the system.
  - SSD is type of **Interaction diagram**


The actor interacting with the system
An object (underlined) representing the automated system
:System
An input message
Clerk
inquireOnItem (catalogID, prodID, size)
The object lifeline; shows the"sequence" of messages, top to bottom
item information
item information: description, price, quantity
A returned value
Optional note to explain something in a diagram

5.3 SSD Indentifying I/O (2)


The actor interacting with the system
An object (underlined) representing the automated system
:System
Clerk
An input message
inquireOnItem (catalogID, prodID, size)
The object lifeline; shows the“sequence” of messages, top to bottom
item information
item information: description, price, quantity
A returned value
Optional note to explain something in a diagram

5.3 SSD Indentifying I/O (3): loop frame

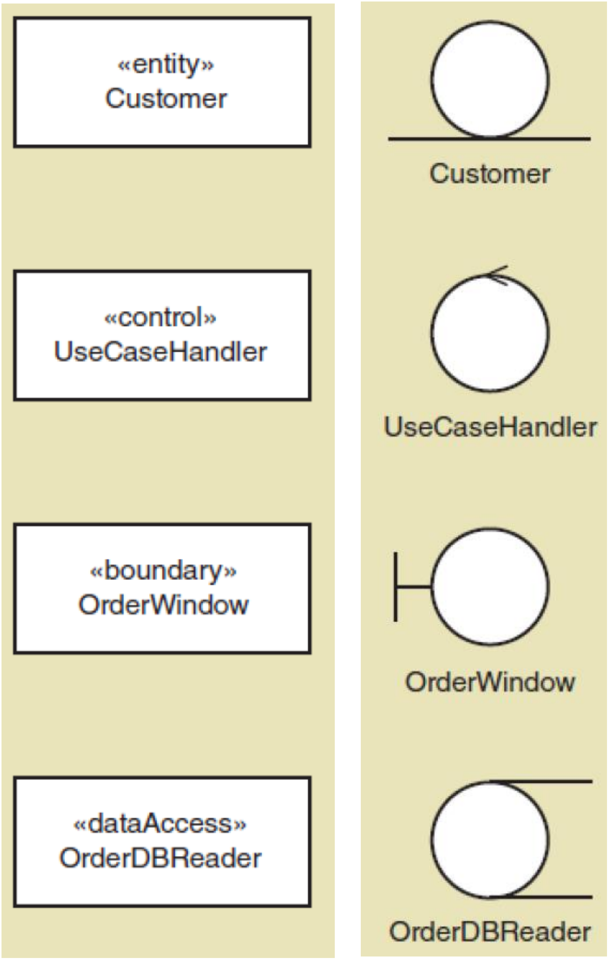

Clerk
:System
Test condition for repeatability
Loop for all items
addItem (itemID, quantity)
description, price, extendedPrice
Repeat everything in the rectangle

Loop frame is the repeating operation operating multiple times between an actor and a system

Class diagram

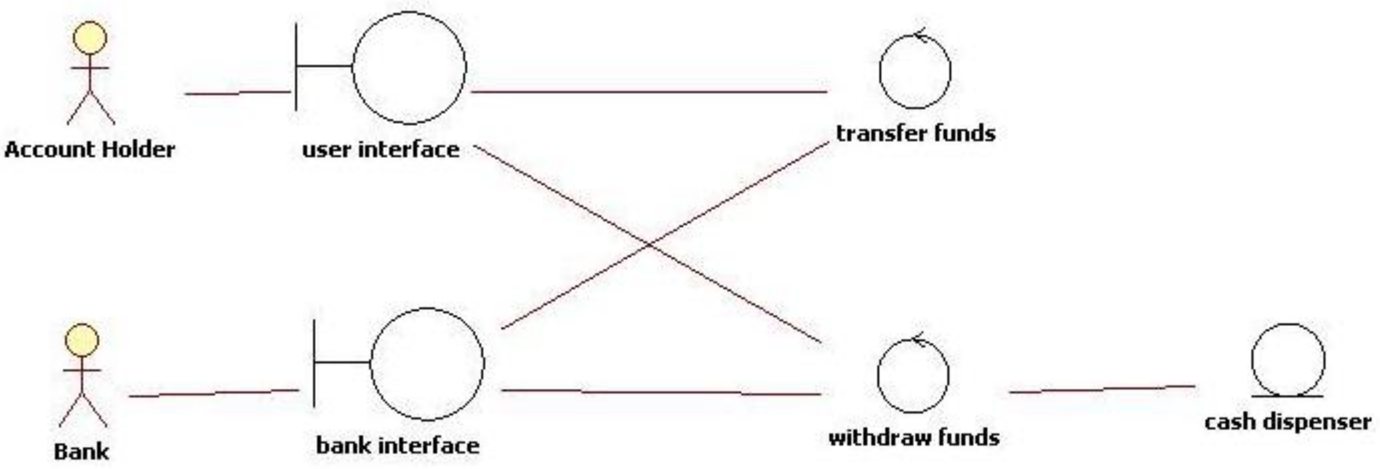
10.4.1 Design class symbols (2)

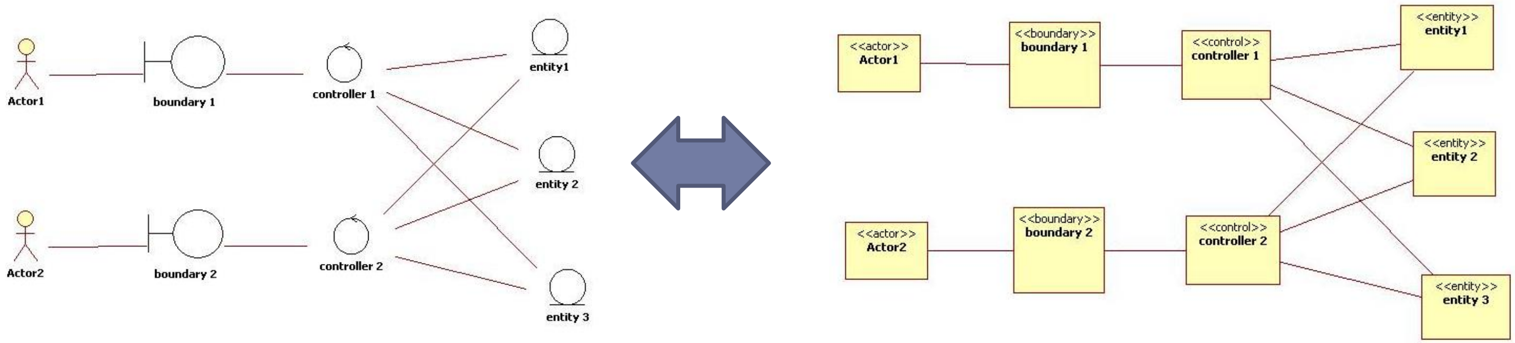
- entity class a design identifier for a problem domain class (usually persistent)
- control class a class that mediates between boundary classes and entity classes, acting as a switchboard between the view layer and domain layer
- boundary class or view class a class that exists on a system's automation boundary, such as an input window form or Web page
- data access class a class that is used to retrieve data from and send data to a database



Example UML design class symbols

ATM

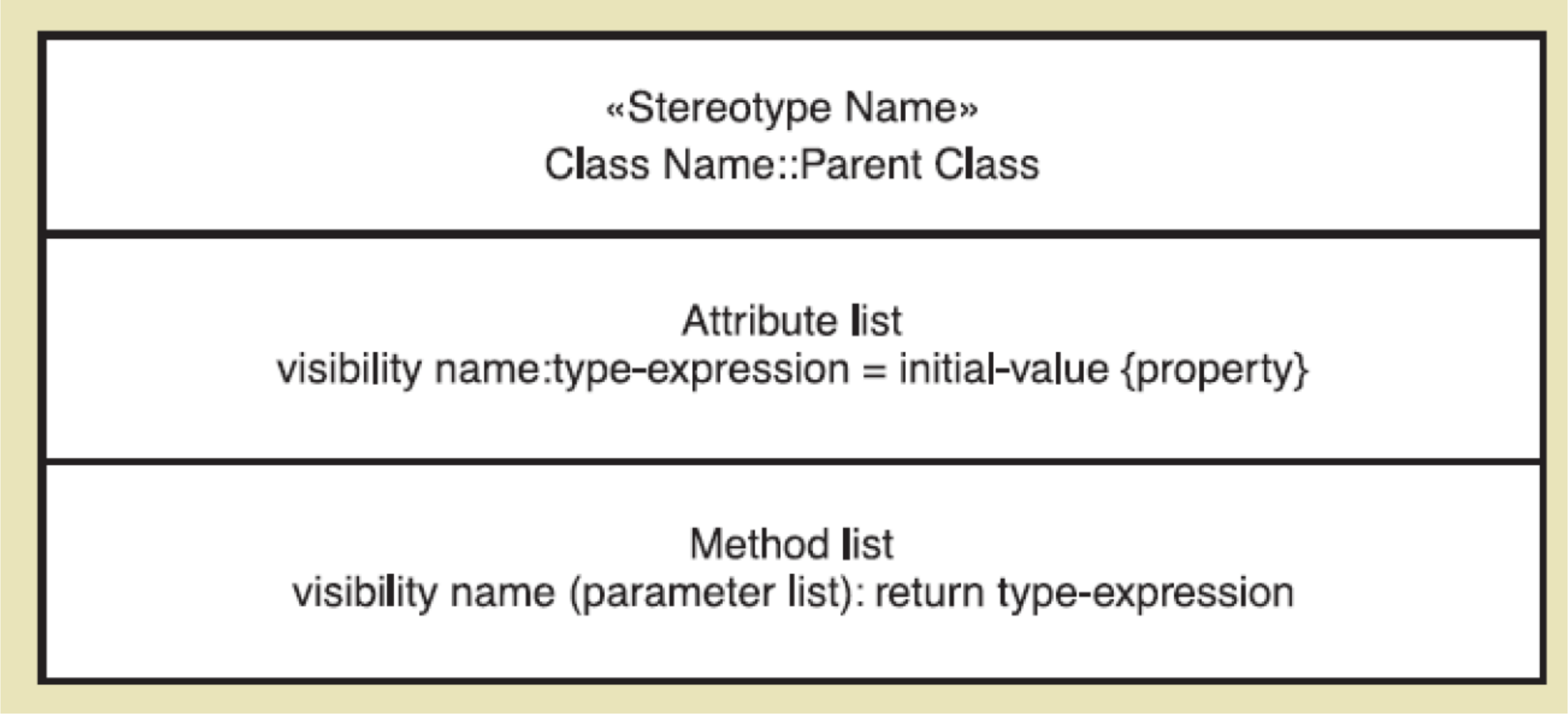




Ref: http://www.cs.sjsu.edu/~pearce/modules/patterns/enterprise/ecb/ecb.htm

10.4.2
Notation for a design class

- Syntax for name, attributes, and methods



«Stereotype Name»
Class Name::Parent Class

Attribute list
visibility name:type-expression = initial-value {property}

Method list
visibility name (parameter list): return type-expression

10.4.2
Notation for design classes

- Attributes
  - Visibility—indicates (+ or -) whether an attribute can be accessed directly by another object.
    - private (-) not visibility
    - public (+) visibility
  - Attribute name—Lower case *camelback* notation
  - Type expression—class, string, integer, double, date
  - Initial value—if applicable the default value
  - Property—if applicable, such as {key}
  - Examples:

- accountNo: String {key}

- startingJobCode: integer = 01

10.4.2
Notation for design classes

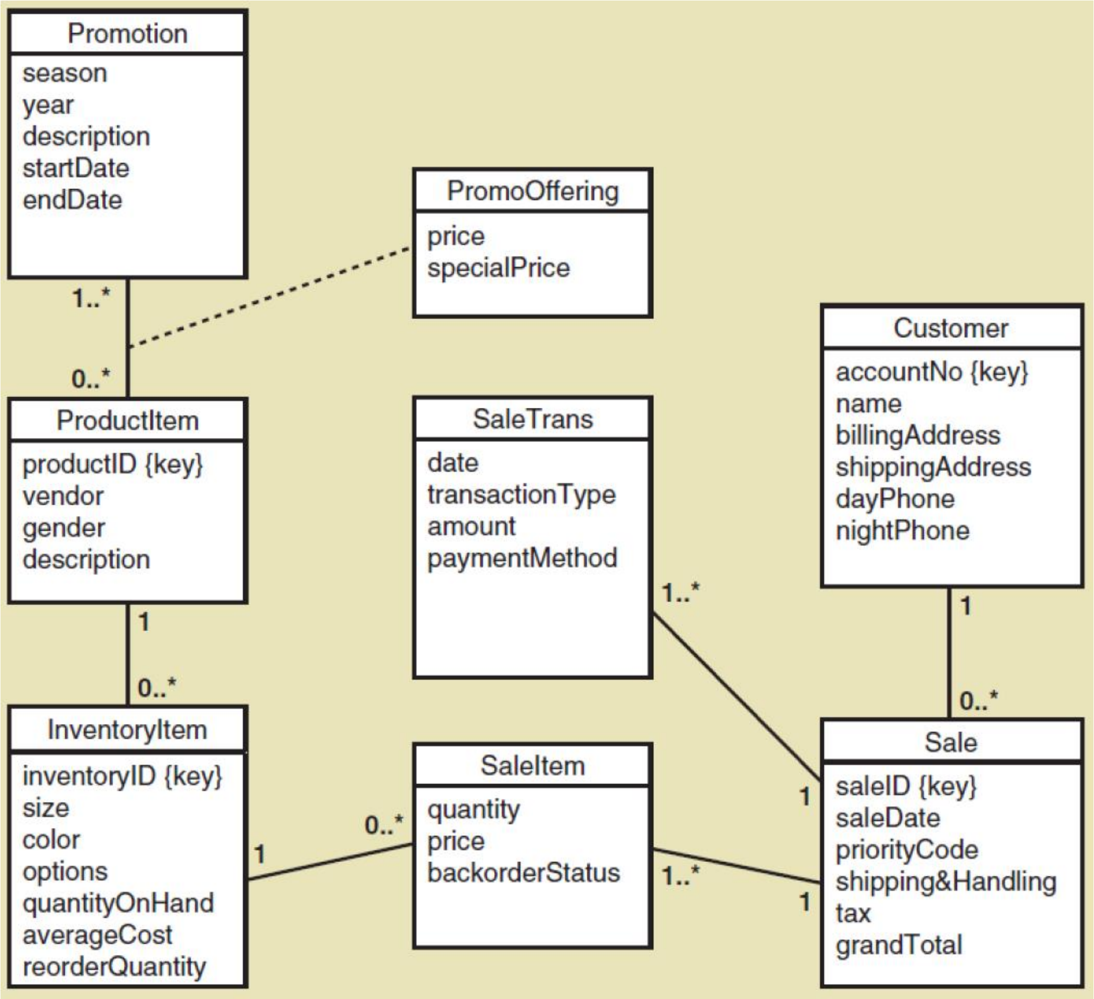
- Methods
  - Visibility—indicates (+ or -) whether an method can be invoked by another object.
    - private (-) not visibility
    - public (+) visibility
  - Method name—Lower case *camelback*, verb-noun
  - Parameters—variables passed to a method
  - Return type—the type of the data returned
  - Examples:

+setName(fName, lName) : void (void is usually let off)

+getName(): string (what is returned is a string)

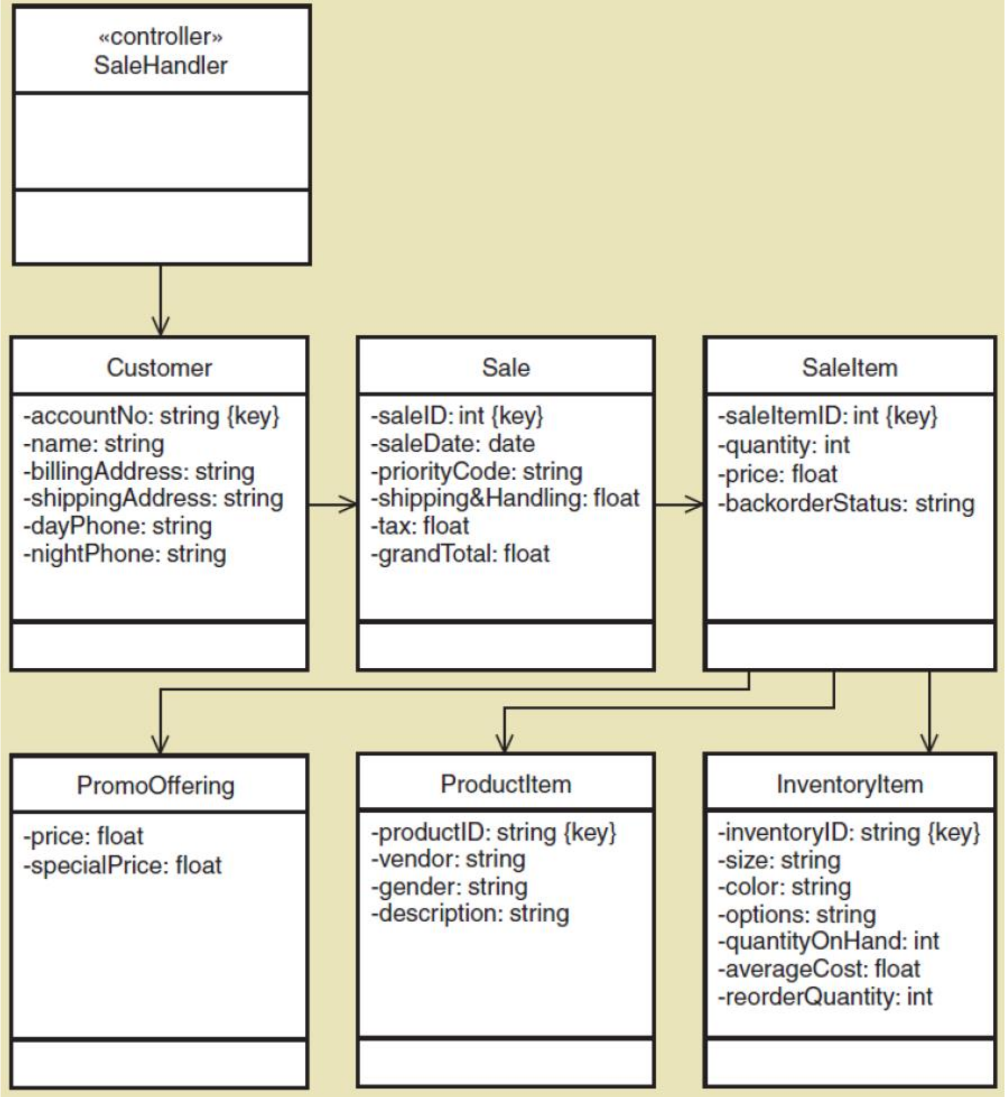
-checkValidity(date) : int (assuming int is a returned code)

Start with domain class diagram RMO sales subsystem



Create first cut design class diagram

- Use case create phone sale with controller added



«controller»
SaleHandler

Customer

-accountNo: string {key}
-name: string
-billingAddress: string
-shippingAddress: string
-dayPhone: string
-nightPhone: string

Sale

-saleID: int {key}
-saleDate: date
-priorityCode: string
-shipping&Handling: float
-tax: float
-grandTotal: float

SaleItem

-saleItemID: int {key}
-quantity: int
-price: float
-backorderStatus: string

PromoOffering

-price: float
-specialPrice: float

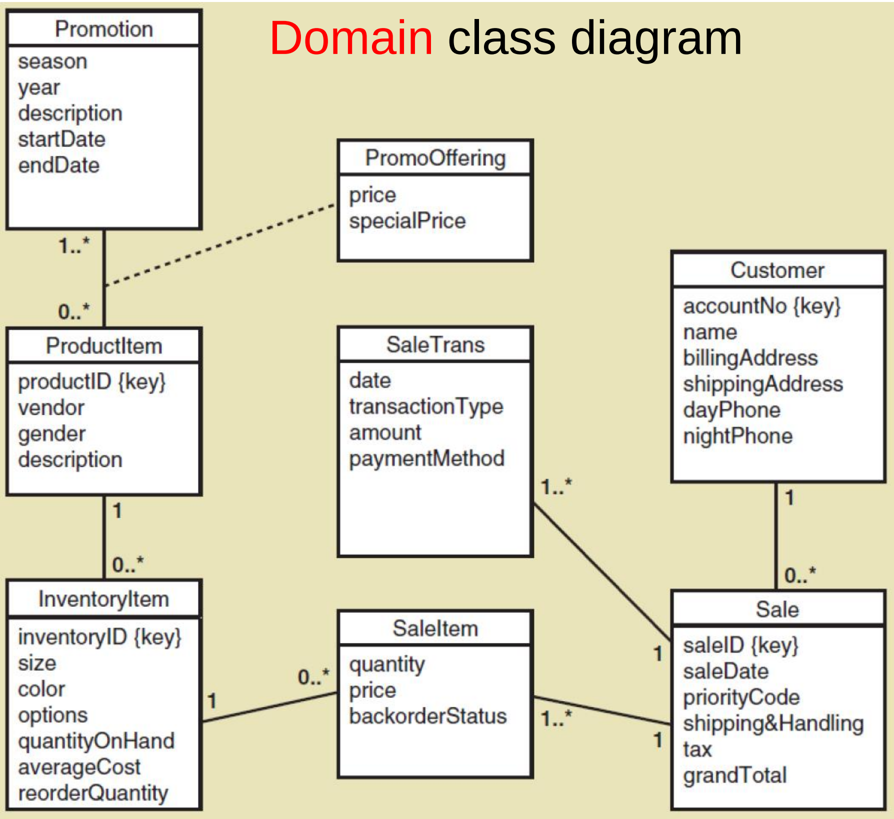
ProductItem

-productID: string {key}
-vendor: string
-gender: string
-description: string

InventoryItem

-inventoryID: string {key}
-size: string
-color: string
-options: string
-quantityOnHand: int
-averageCost: float
-reorderQuantity: int

Domain class diagram



Promotion
- season
- year
- description
- startDate
- endDate

PromoOffering
- price
- specialPrice

ProductItem
- productID {key}
- vendor
- gender
- description

SaleTrans
- date
- transactionType
- amount
- paymentMethod

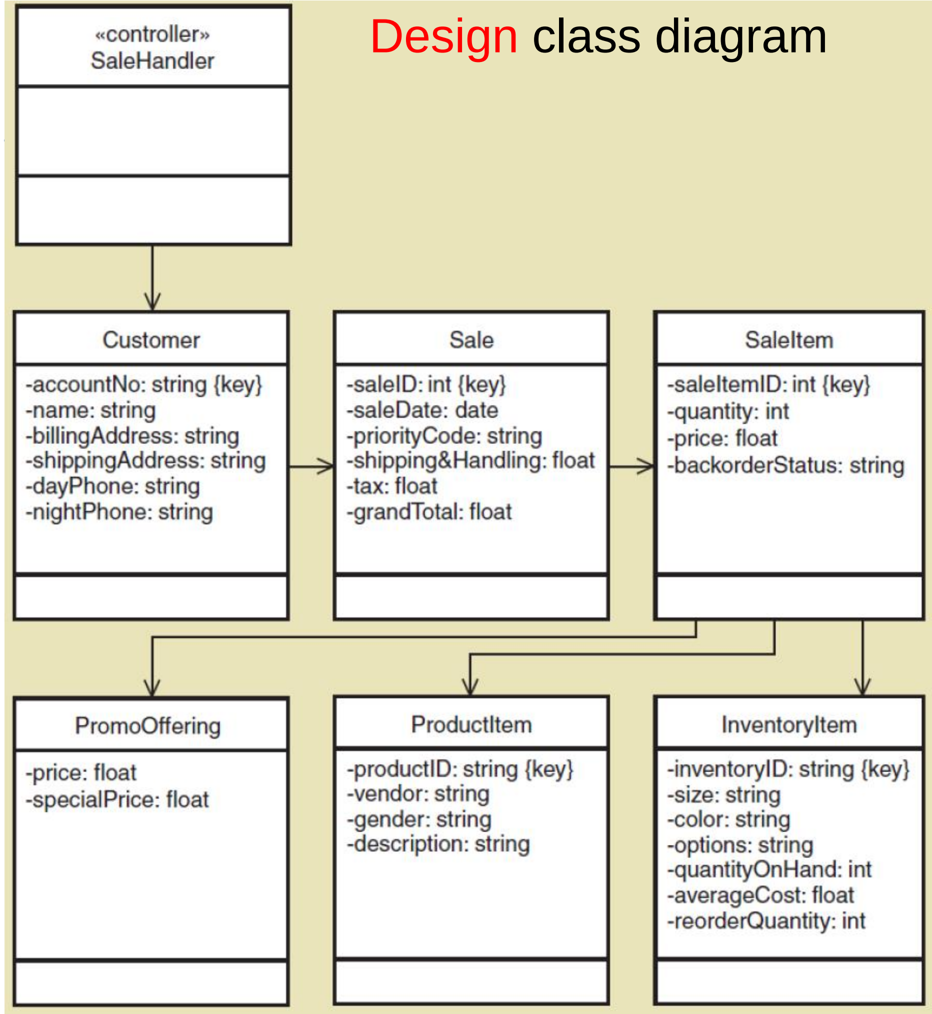
Customer
- accountNo {key}
- name
- billingAddress
- shippingAddress
- dayPhone
- nightPhone

InventoryItem
- inventoryID {key}
- size
- color
- options
- quantityOnHand
- averageCost
- reorderQuantity

SaleItem
- quantity
- price
- backorderStatus

Sale
- saleID {key}
- saleDate
- priorityCode
- shipping&Handling
- tax
- grandTotal

Design class diagram



«controller»
SaleHandler

Customer
- -accountNo: string {key}
- -name: string
- -billingAddress: string
- -shippingAddress: string
- -dayPhone: string
- -nightPhone: string

Sale
- -saleID: int {key}
- -saleDate: date
- -priorityCode: string
- -shipping&Handling: float
- -tax: float
- -grandTotal: float

SaleItem
- -saleItemID: int {key}
- -quantity: int
- -price: float
- -backorderStatus: string

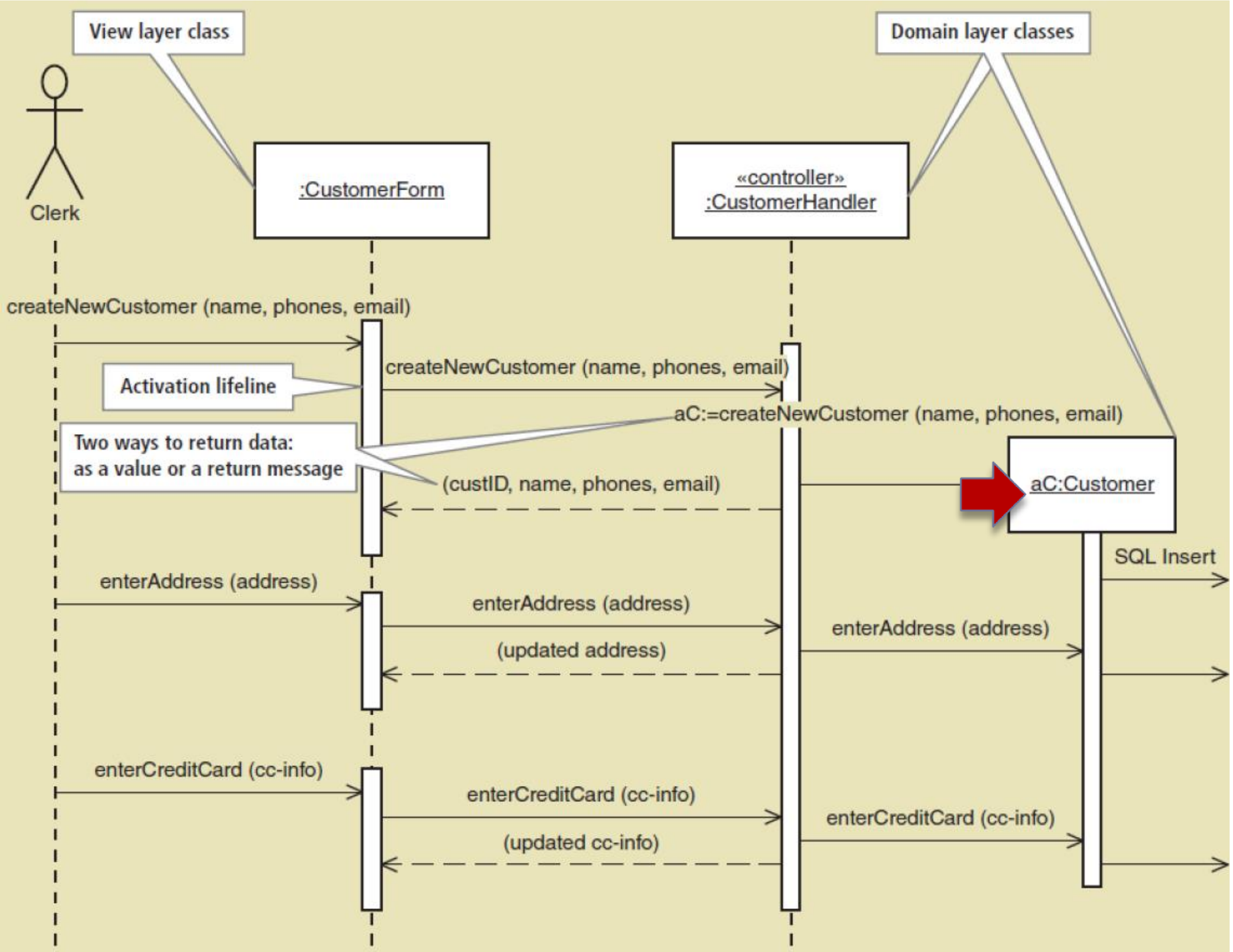
PromoOffering
- -price: float
- -specialPrice: float

ProductItem
- -productID: string {key}
- -vendor: string
- -gender: string
- -description: string

InventoryItem
- -inventoryID: string {key}
- -size: string
- -color: string
- -options: string
- -quantityOnHand: int
- -averageCost: float
- -reorderQuantity: int

Sequence diagram

11.2.1 Sequence diagram: Example, two-level details design

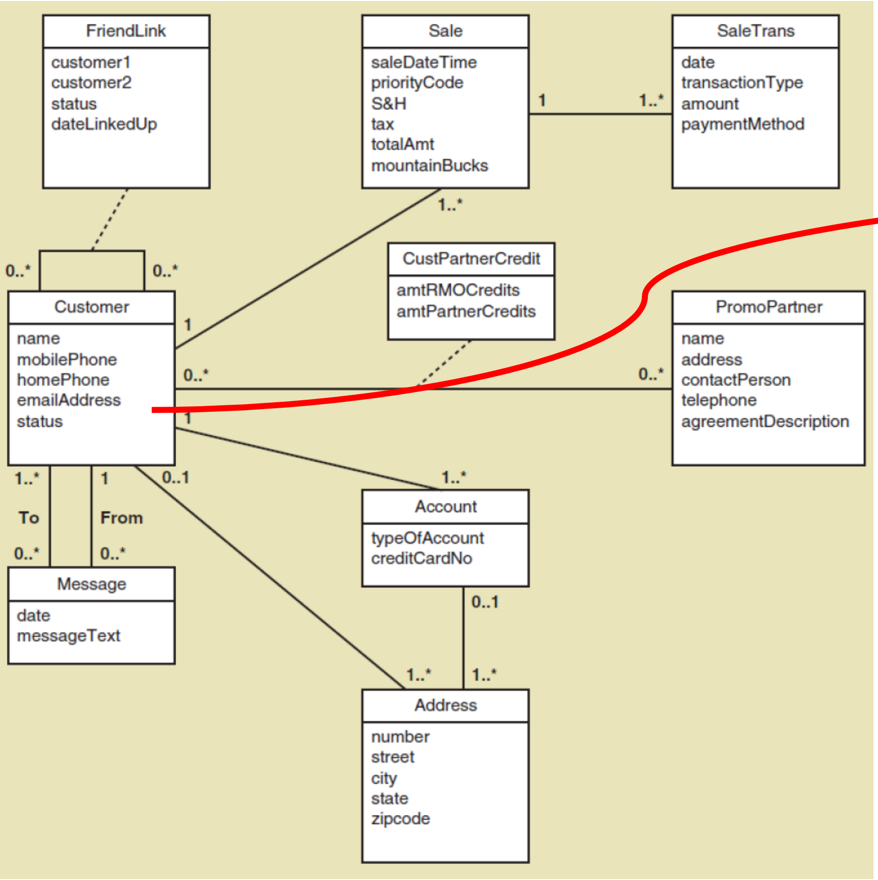

View layer class
Domain layer classes
Clerk
:CustomerForm
«controller»
:CustomerHandler
createNewCustomer (name, phones, email)
createNewCustomer (name, phones, email)
Activation lifeline
aC:=createNewCustomer (name, phones, email)
Two ways to return data:
as a value or a return message
(custID, name, phones, email)
aC:Customer
SQL Insert
enterAddress (address)
enterAddress (address)
enterAddress (address)
(updated address)
enterCreditCard (cc-info)
enterCreditCard (cc-info)
enterCreditCard (cc-info)
(updated cc-info)

Note of expanded sequence diagram

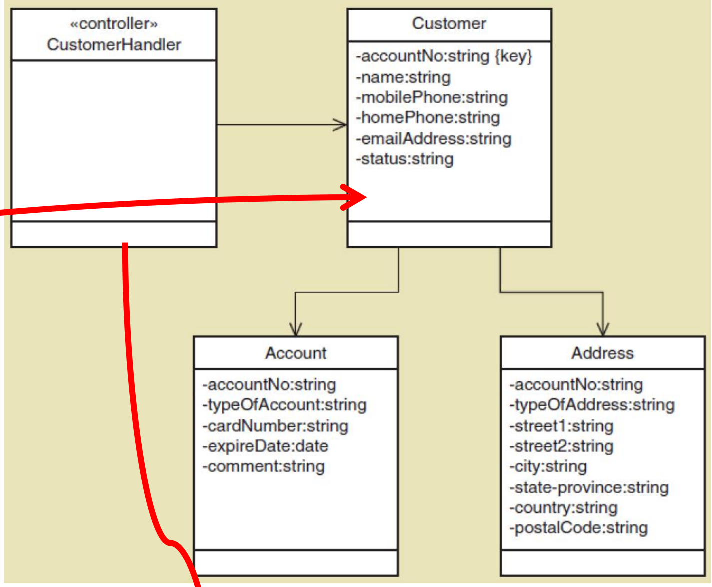
- This is a two layer architecture, as the domain class Customer knows about the database and executes SQL statements for data access
- Three layer design would add a data access class to handle the database resulting in higher cohesiveness and loose coupling
- Note :
  - CustomerForm is an object of the CustomerForm class,
  - :CustomerHandler is an object of the CustomerHandler class playing the role of a controller stereotype (both underlined because they are objects)
  - aC:Customer is an object of the Customer class known by reference variable named aC

11.2.2 First-cut sequence diagram: Example create *customer account* Use case

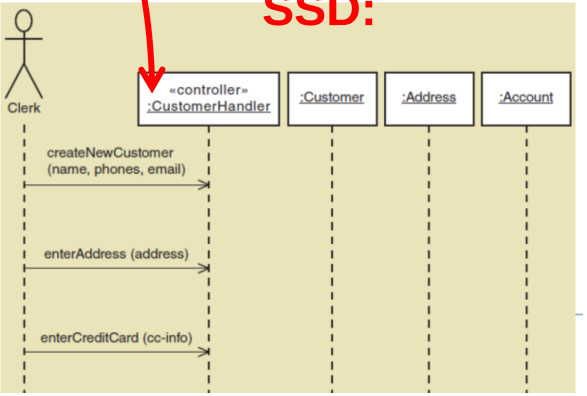
Domain model: (Chap4)
Customer account system



Design class diagram:
Create customer account use case

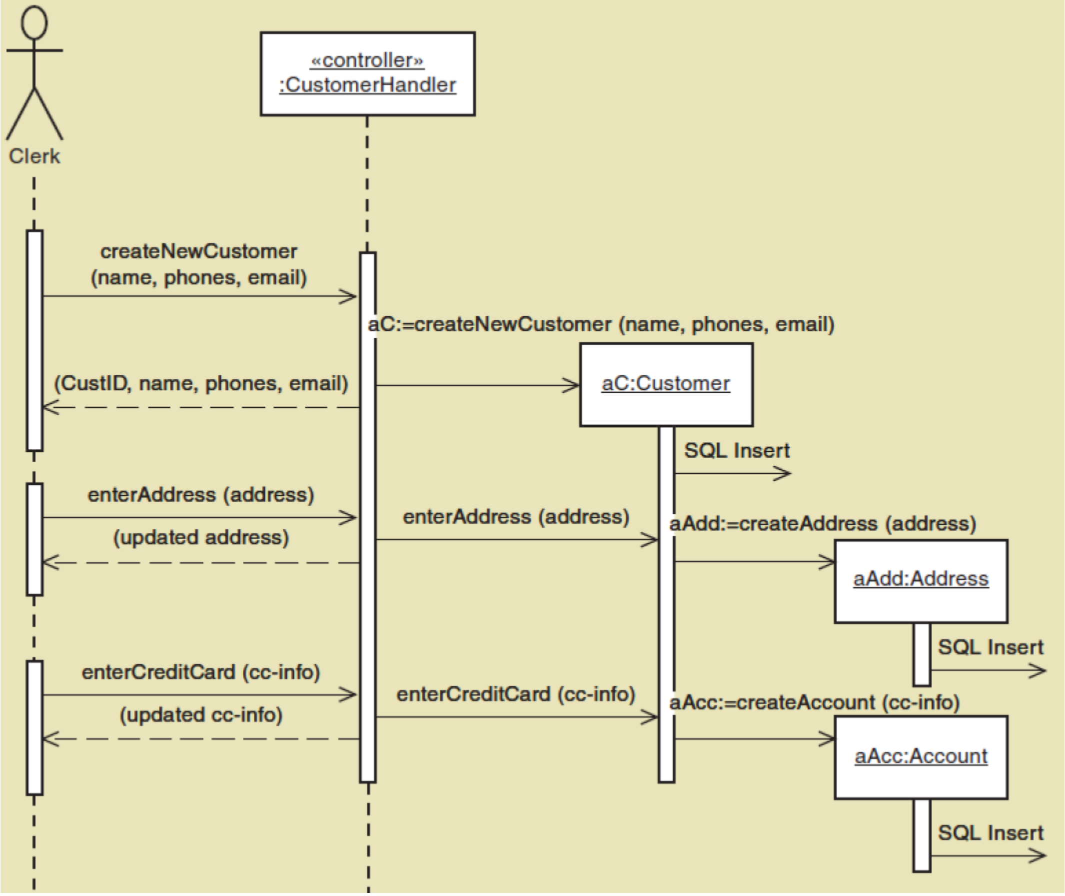


SSD:



11.2.2 First-cut sequence diagram: Example create *customer account* Use case

- Add messages and activation to complete collaboration
- This is just the domain layer
- These domain classes handle data access, so this is a two layer architecture



Clerk
«controller»
:CustomerHandler
createNewCustomer (name, phones, email)
aC:=createNewCustomer (name, phones, email)
(CustID, name, phones, email)
aC:Customer
SQL Insert
enterAddress (address)
(updated address)
enterAddress (address)
aAdd:=createAddress (address)
aAdd:Address
SQL Insert
enterCreditCard (cc-info)
(updated cc-info)
enterCreditCard (cc-info)
aAcc:=createAccount (cc-info)
aAcc:Account
SQL Insert

11.2.3 Guideline and Assumptions for first-cut sequence diagram development

- Perfect technology assumption—First encountered for use cases. We don't include messages such as the user having to log on.
- Perfect memory assumption—We have assumed that the necessary objects were in memory and available for the use case. In multilayer design to follow, we do include the steps necessary to create objects in memory.
- Perfect solution assumption—The first-cut sequence diagram assumes no exception conditions.
- Separation of responsibilities—Design principle that recommends segregating classes into separate components based on the primary focus, such as user interface, domain, and data access

11.2.4 Developing a multilayer design

Problem in domain classes

- Persistent classes is the problem on complex business logic that some class contains the mechanism for storing and retrieving data from a database.

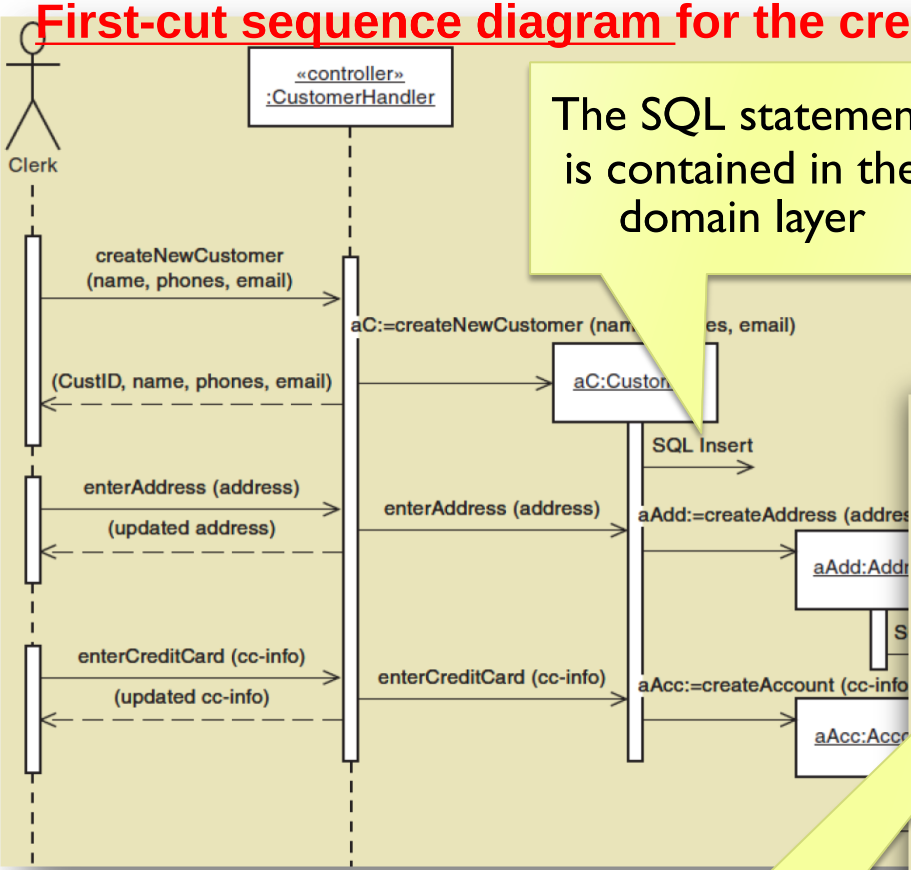
Solving

- Apply separate layer is the separate connection to database and SQL from the domain classes.

The Multilayer design has three-layers design use concept of separation responsibility

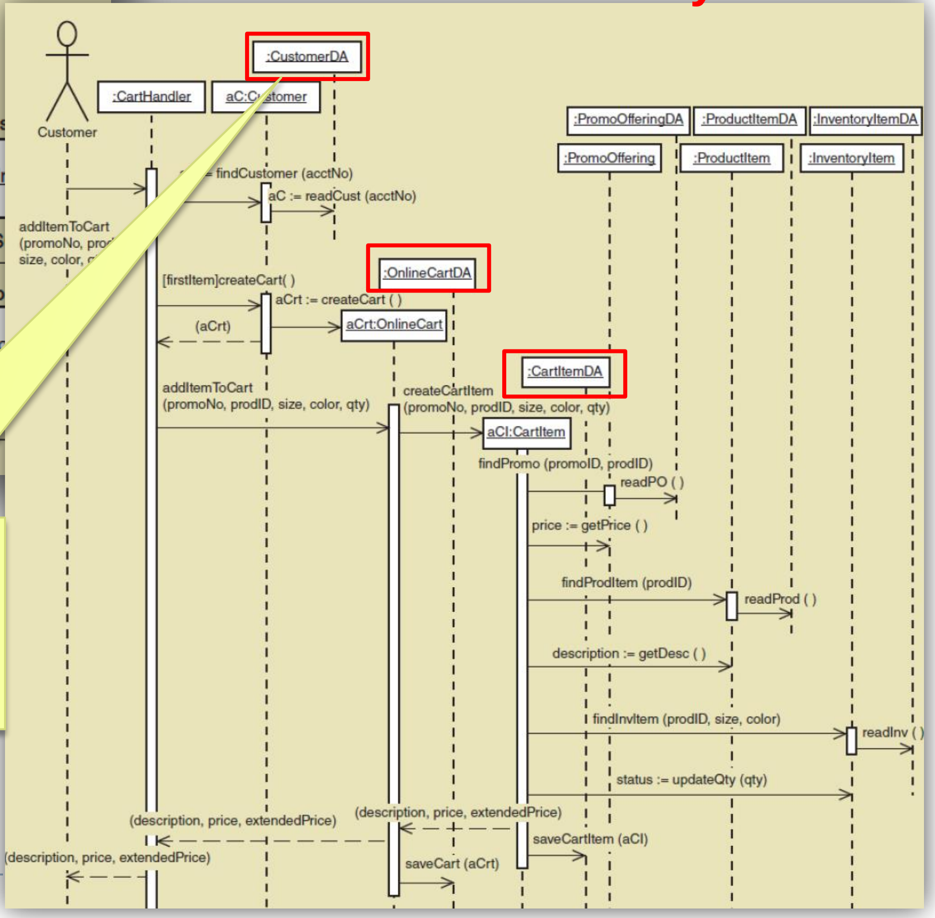
1) View layer
   - □ Get input data or commands
   - □ Show output or command responding
2) Domain layer
3) Data access layer

Fig11-8
First-cut sequence diagram for the create customer account use case



The SQL statement is contained in the domain layer

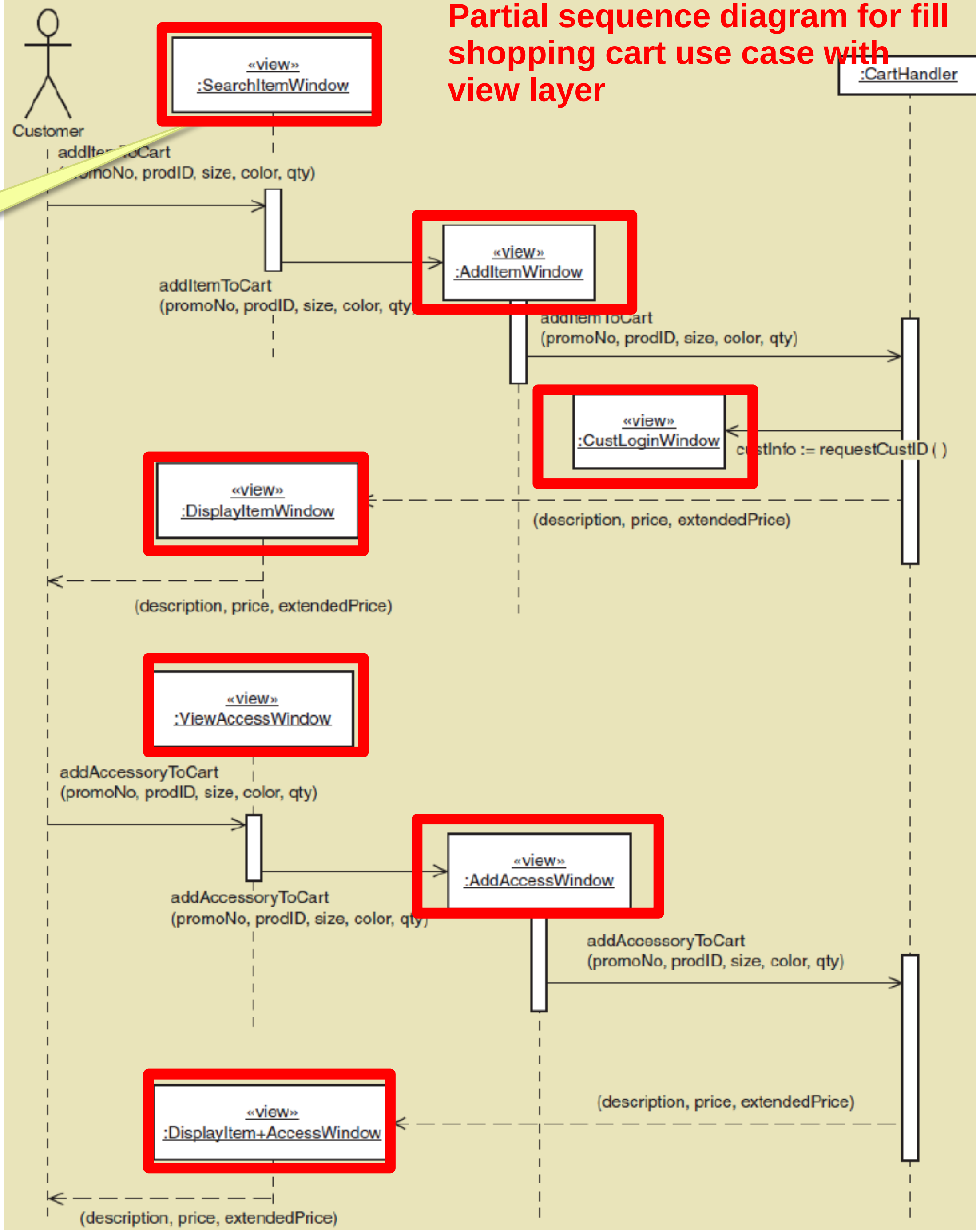
Fig11-13
Sequence diagram for fill shopping cart use case with data access layer



Data access layer
:CustomerDA
:CartItemDA
:OnlineCartDA

11.2.4 Developing a multilayer design: View layer

Fig11-14
Partial sequence diagram for fill shopping cart use case with view layer

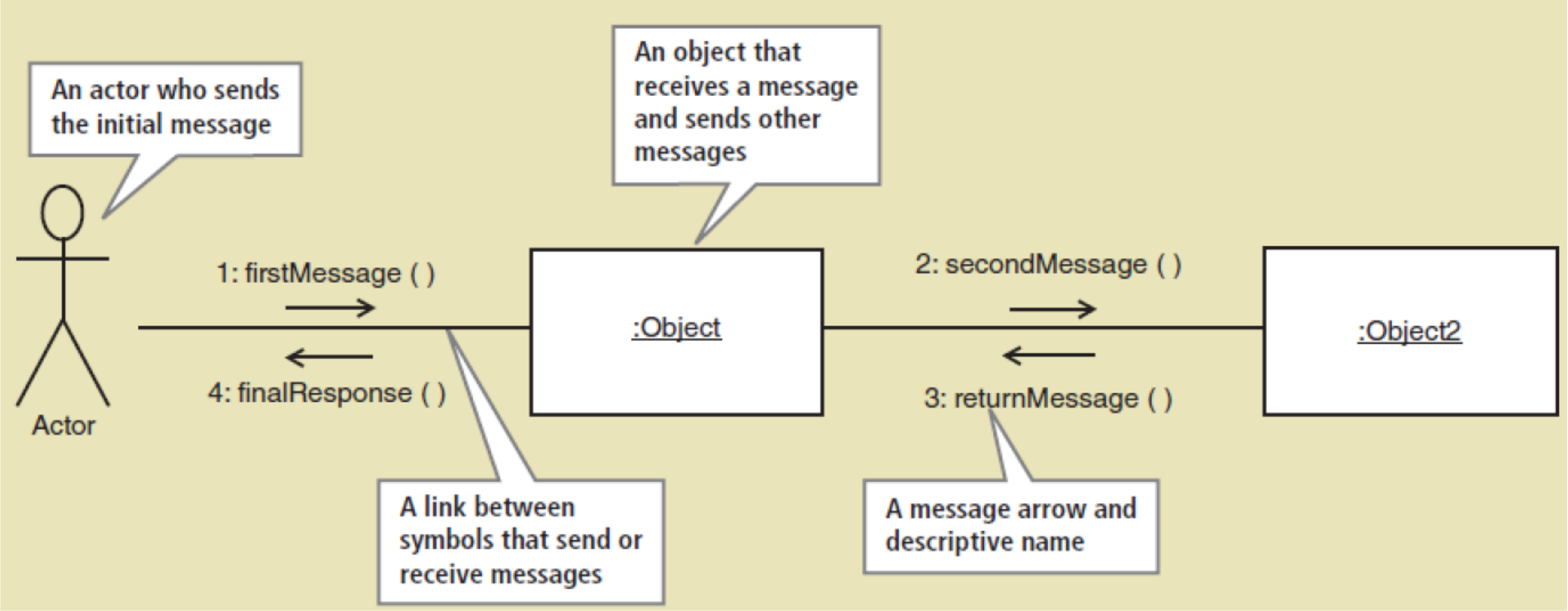

Customer
«view»
:SearchItemWindow
:CartHandler
addItemToCart
(promoNo, prodID, size, color, qty)
«view»
:AddItemWindow
addItemToCart
(promoNo, prodID, size, color, qty)
addItemToCart
(promoNo, prodID, size, color, qty)
«view»
:CustLoginWindow
custInfo := requestCustID ( )
«view»
:DisplayItemWindow
(description, price, extendedPrice)
(description, price, extendedPrice)
«view»
:ViewAccessWindow
addAccessoryToCart
(promoNo, prodID, size, color, qty)
«view»
:AddAccessWindow
addAccessoryToCart
(promoNo, prodID, size, color, qty)
addAccessoryToCart
(promoNo, prodID, size, color, qty)
«view»
:DisplayItem+AccessWindow
(description, price, extendedPrice)
(description, price, extendedPrice)

View layer

Communication diagram

11.3 Designing communication diagrams

- Shows the same information as a sequence diagram
- Symbols used in a communication diagram:

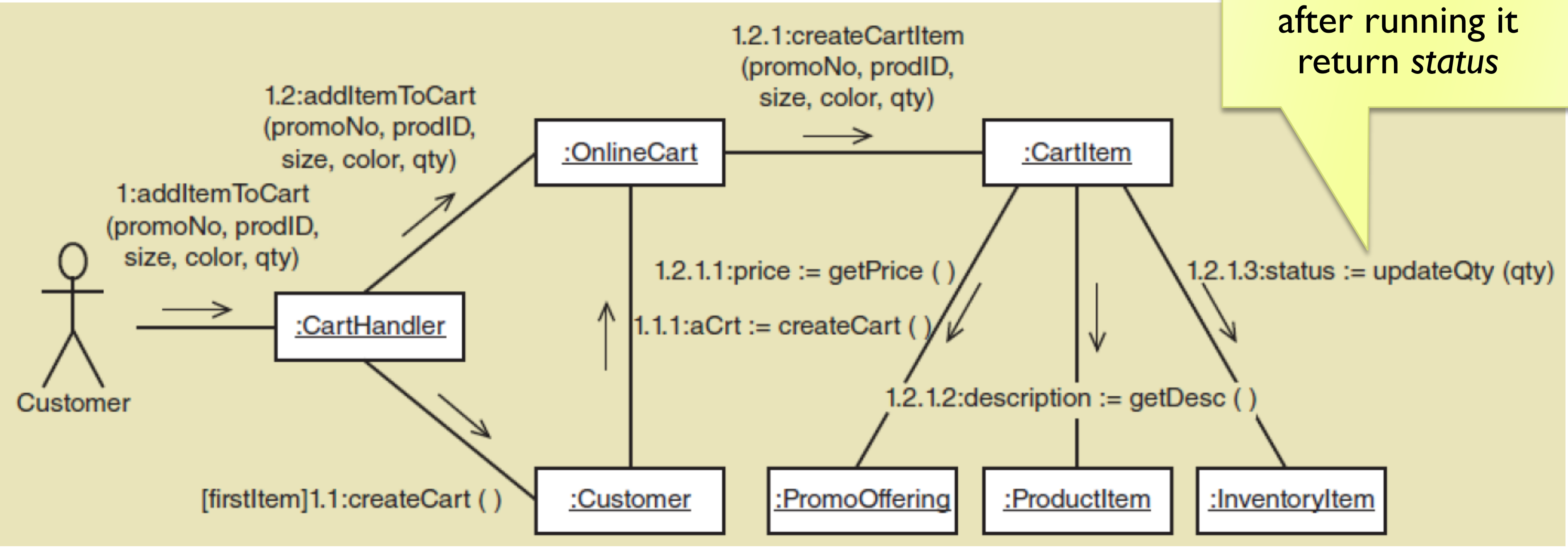


An actor who sends the initial message
An object that receives a message and sends other messages
Actor
1: firstMessage ( )
4: finalResponse ( )
:Object
2: secondMessage ( )
3: returnMessage ( )
:Object2
A link between symbols that send or receive messages
A message arrow and descriptive name

[true/false condition] sequence-number; return-value := **message-name**(parameter-list)

11.3 Designing communication diagrams Example Fill Shopping Cart use case

- This diagram should math the domain layer sequence diagram shown earlier.
- Many people prefer them for brainstorming


1.2.1:createCartItem (promoNo, prodID, size, color, qty)
1.2:addItemToCart (promoNo, prodID, size, color, qty)
1:addItemToCart (promoNo, prodID, size, color, qty)
:OnlineCart
:CartItem
:CartHandler
Customer
1.2.1.1:price := getPrice ( )
1.1.1:aCrt := createCart ( )
1.2.1.3:status := updateQty (qty)
1.2.1.2:description := getDesc ( )
[firstItem]1.1:createCart ( )
:Customer
:PromoOffering
:ProductItem
:InventoryItem

The method *UpdateQty* gets number *qty* and after running it return *status*