Systems Design

OO DESIGN

Today's Plan

Look at Object Oriented Design

How to build models for developers

Best principles in OO Design

Review of OO Programs

In OO, programs are composed of Objects

Objects have:

- Data
- Logic

Objects described by a Class

Program consists of set of Objects

- They work together to solve the problem

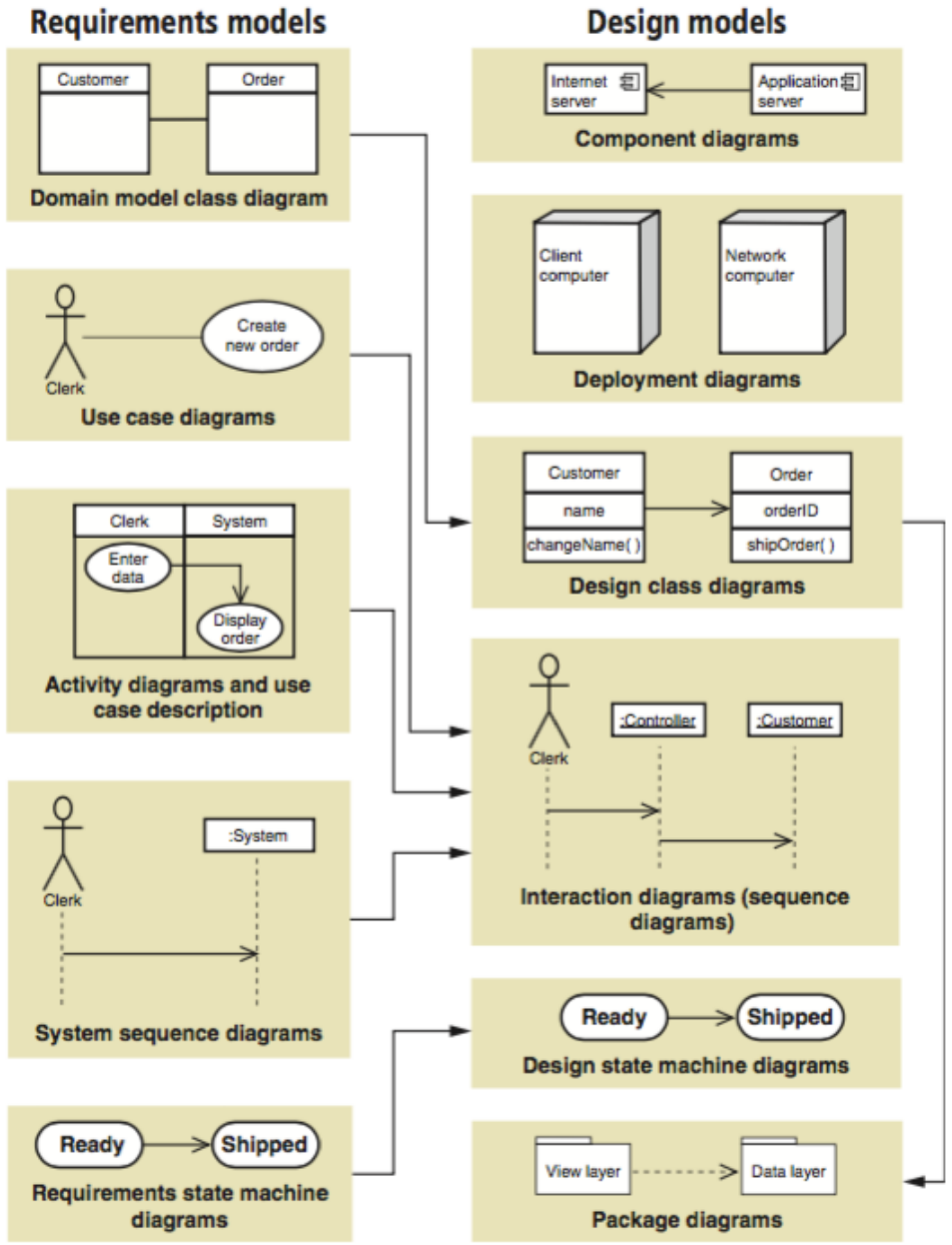
From Analysis Models to Design Models

Information about things

- Domain Model Class Diagrams
- Use case descriptions

Information about business processes

- Activity diagrams
- System sequence diagrams (SSDs)
- Flow Diagrams

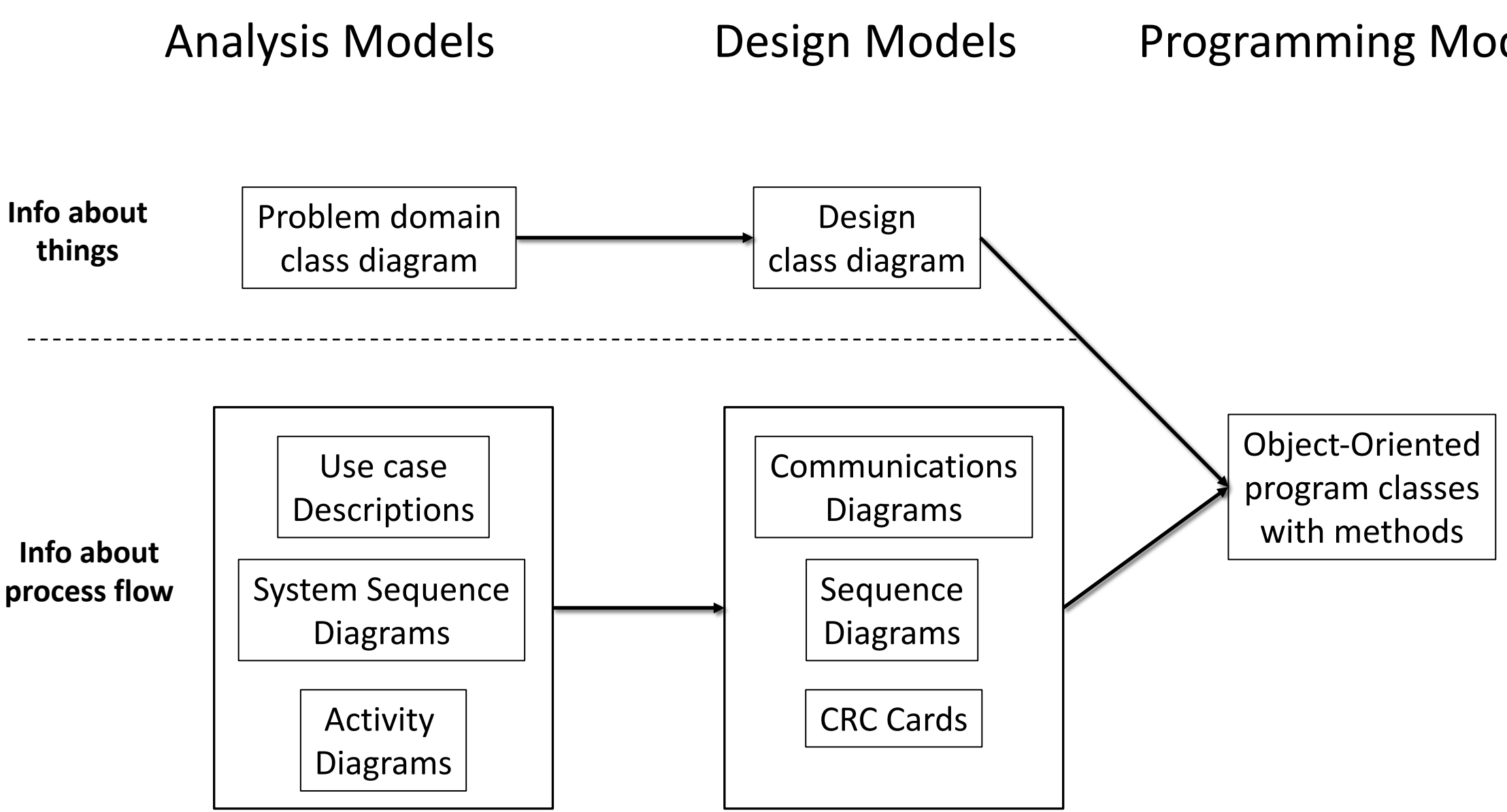

Requirements models
Design models
Customer
Order
Domain model class diagram
Internet server
Application server
Component diagrams
Client computer
Network computer
Deployment diagrams
Clerk
Create new order
Use case diagrams
Customer
name
changeName( )
Order
orderID
shipOrder( )
Design class diagrams
Clerk
System
Enter data
Display order
Activity diagrams and use case description
Clerk
:Controller
:Customer
Interaction diagrams (sequence diagrams)
Clerk
:System
System sequence diagrams
Ready
Shipped
Design state machine diagrams
Ready
Shipped
Requirements state machine diagrams
View layer
Data layer
Package diagrams

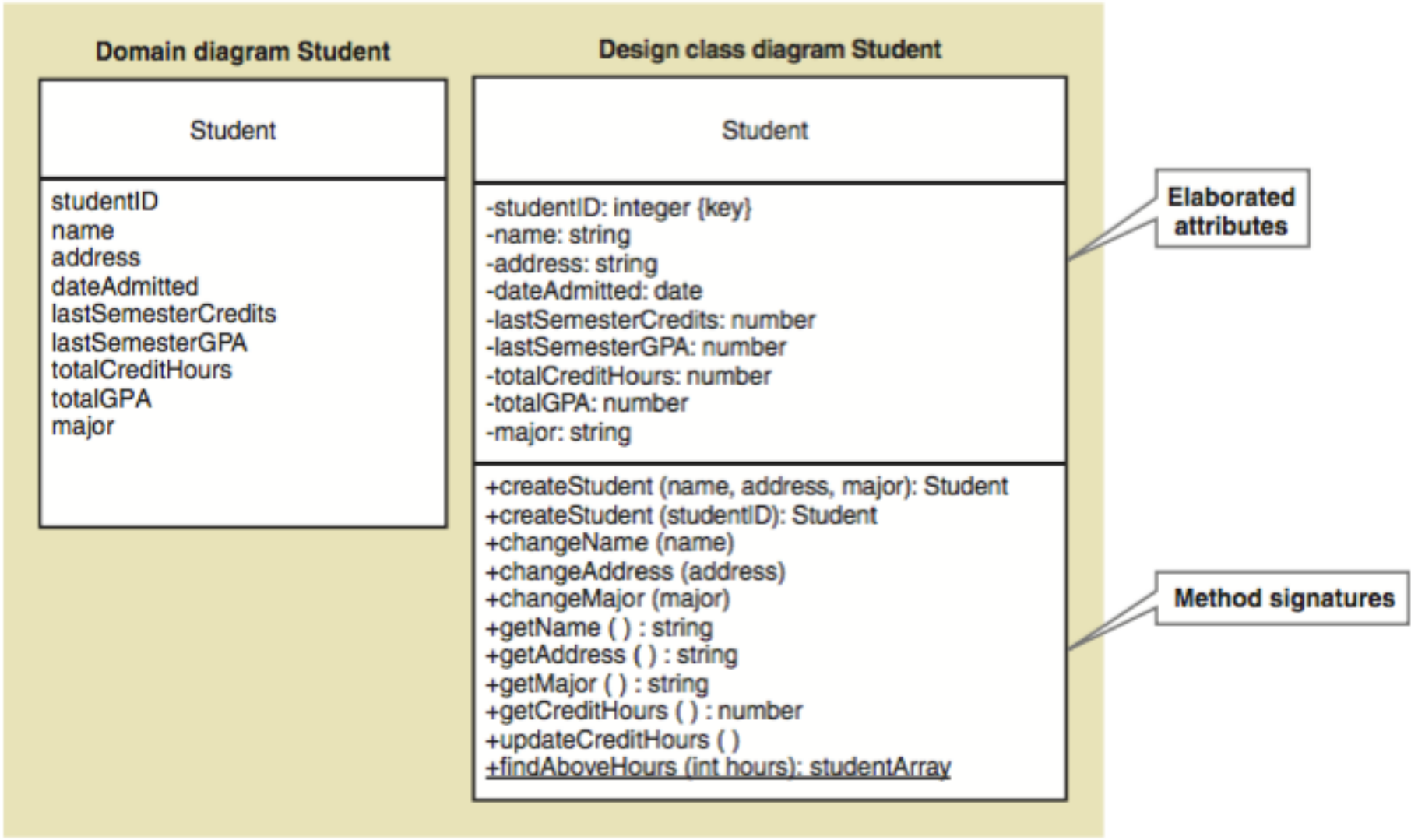
```java
public class Student{

    //attributes
    private int studentID;
    private String firstName;
    private String lastName;
    private String major;
    private float numberCredits;
    private float gpa;

    //constructors
    public Student(String inFirstName, String inLastName){
        firstName = inFirstName;
        lastName = inLastName;
    }
    public Student(int inStudentID){
        studentID = inStudentID;
    }
    //getters and setters
    public String getFullName(){
        return firstName + " " + lastName;
    }
    public void setFirstName(String inFirstName){
            firstName = inFirstName;
    }
    //and more here of course

    //processing methods
    public void updateGPA(){
        //access course records, and update gpa and credits
    }
    //and so forth
}
```


Analysis Models
Design Models
Programming Models
Info about things
Problem domain class diagram
Design class diagram
Info about process flow
Use case Descriptions
System Sequence Diagrams
Activity Diagrams
Communications Diagrams
Sequence Diagrams
CRC Cards
Object-Oriented program classes with methods

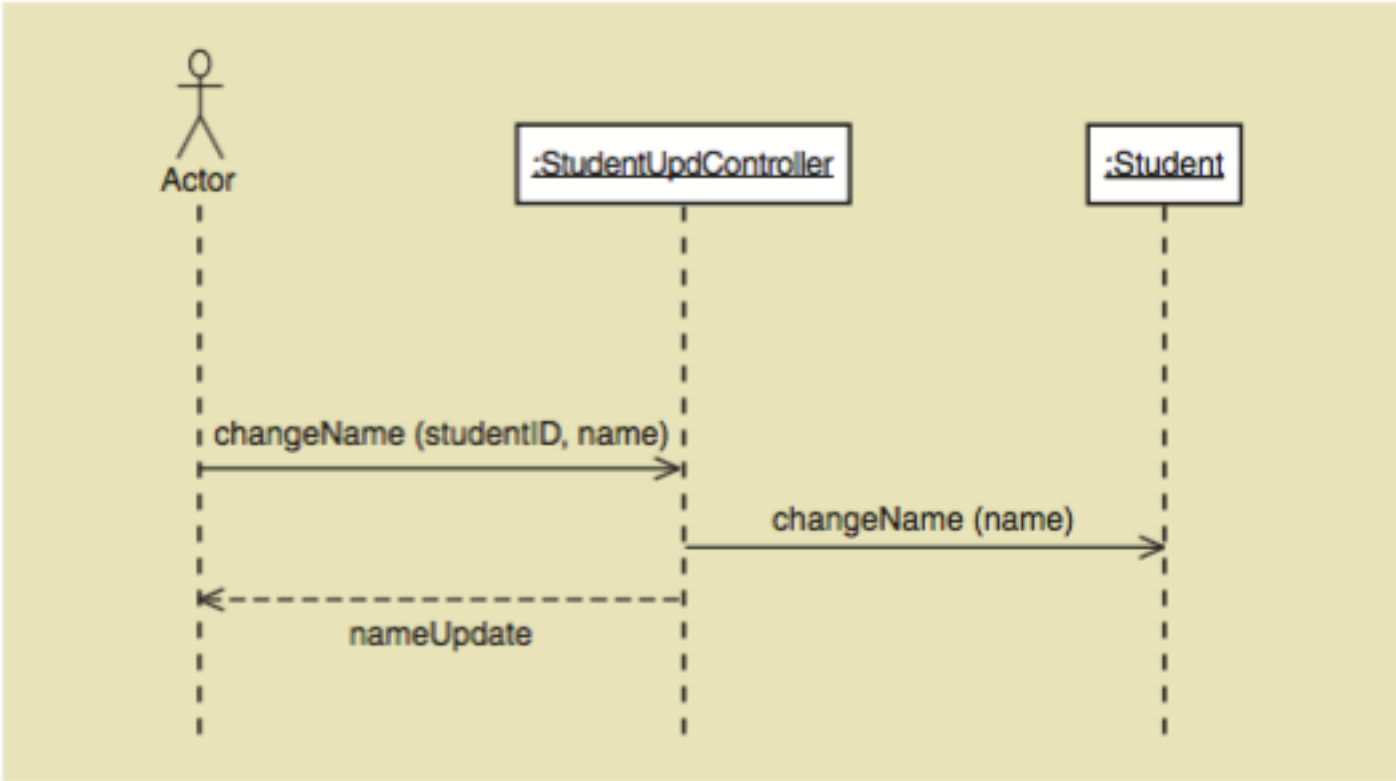

Domain diagram Student
Student
studentID
name
address
dateAdmitted
lastSemesterCredits
lastSemesterGPA
totalCreditHours
totalGPA
major
Design class diagram Student
Student
-studentID: integer {key}
-name: string
-address: string
-dateAdmitted: date
-lastSemesterCredits: number
-lastSemesterGPA: number
-totalCreditHours: number
-totalGPA: number
-major: string
+createStudent (name, address, major): Student
+createStudent (studentID): Student
+changeName (name)
+changeAddress (address)
+changeMajor (major)
+getName ( ) : string
+getAddress ( ) : string
+getMajor ( ) : string
+getCreditHours ( ) : number
+updateCreditHours ( )
+findAboveHours (int hours): studentArray
Elaborated attributes
Method signatures

Creating a Design Class Diagram

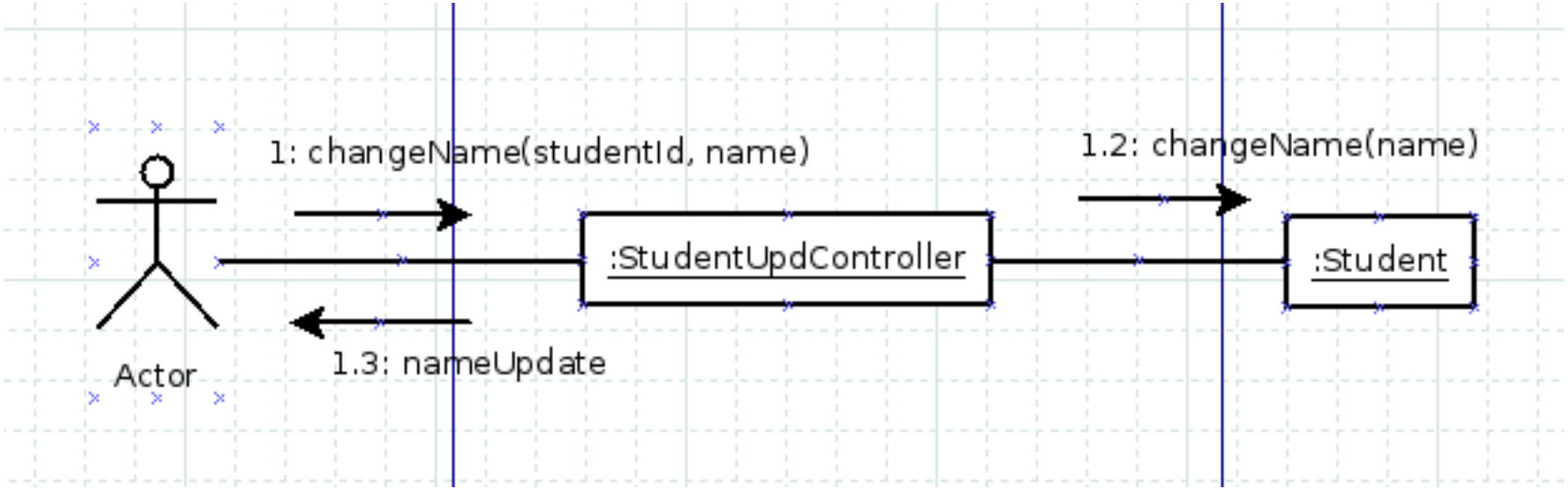
Transform Analysis Models into:

- CRC cards
- UML interaction model
  - Communication diagram
  - Sequence diagram

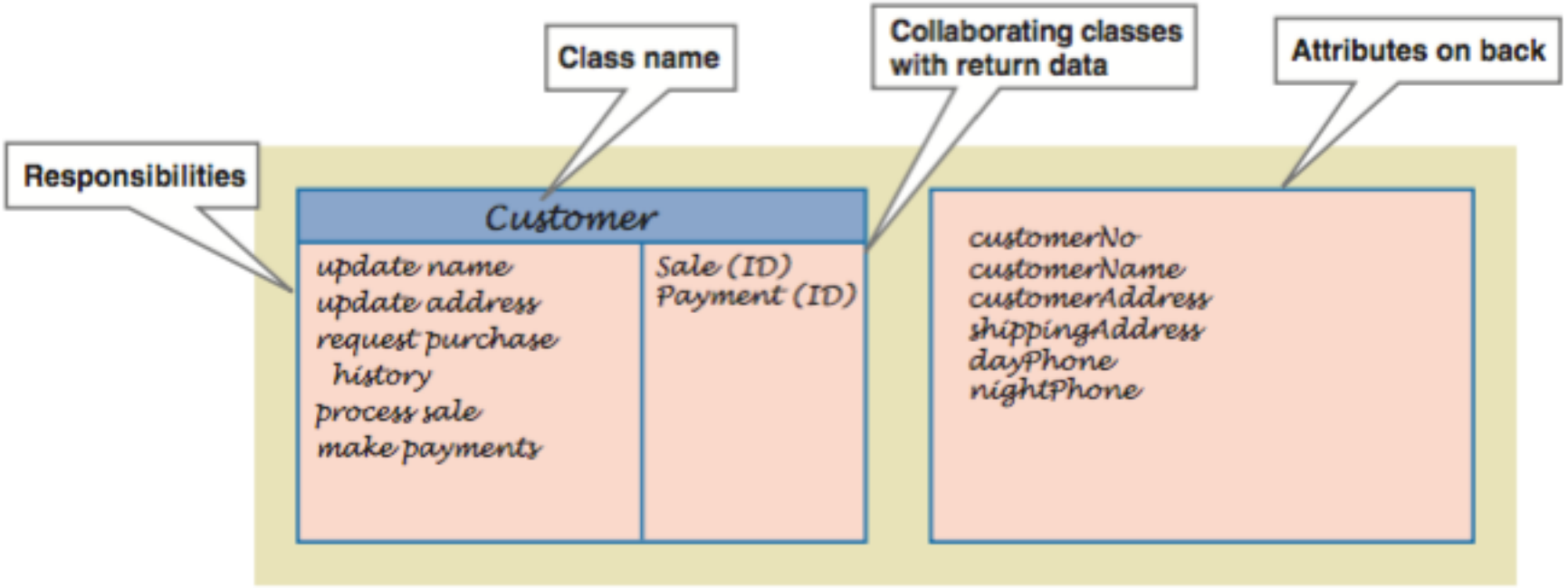
Sequence Diagram



Communication Diagram



CRC Cards


Class name
Collaborating classes with return data
Attributes on back
Responsibilities
Customer
update name
update address
request purchase history
process sale
make payments
Sale (ID)
Payment (ID)
customerNo
customerName
customerAddress
shippingAddress
dayPhone
nightPhone

How to do OO Design

OO Design is Use Case driven

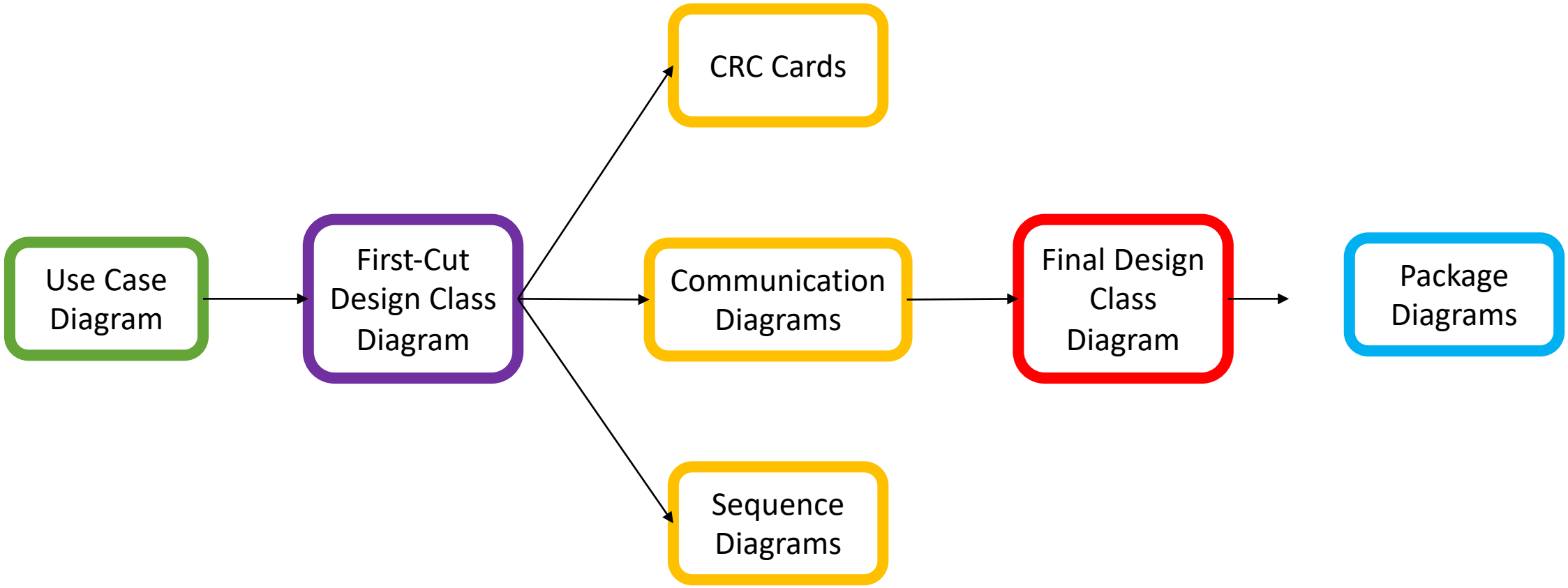
- One use case at a time:
  - Models are constructed and updated

OO is a rigorous process

OO Design is time consuming

- Need to decide whether or how to do it, carefully

Steps of OO Design



Creating the Design Class Diagram

The following processes are done *for each use case*

Start with Analysis Models

- In particular, the Domain Model Class Diagram
- Need to add:
  - Types and visibility of attributes
  - Methods, parameters, and return types
  - Classes outside the user's domain

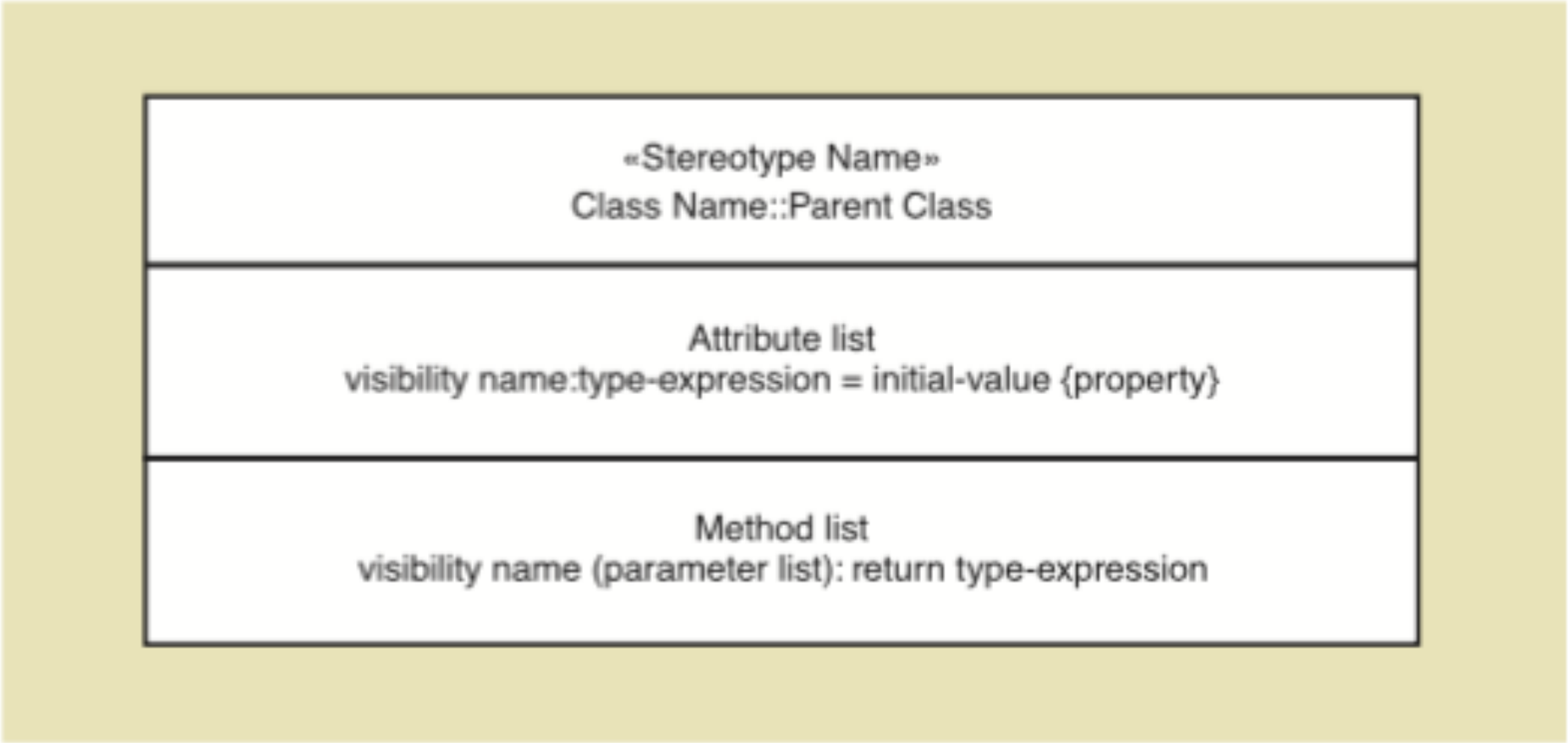
Design Class Stereotypes

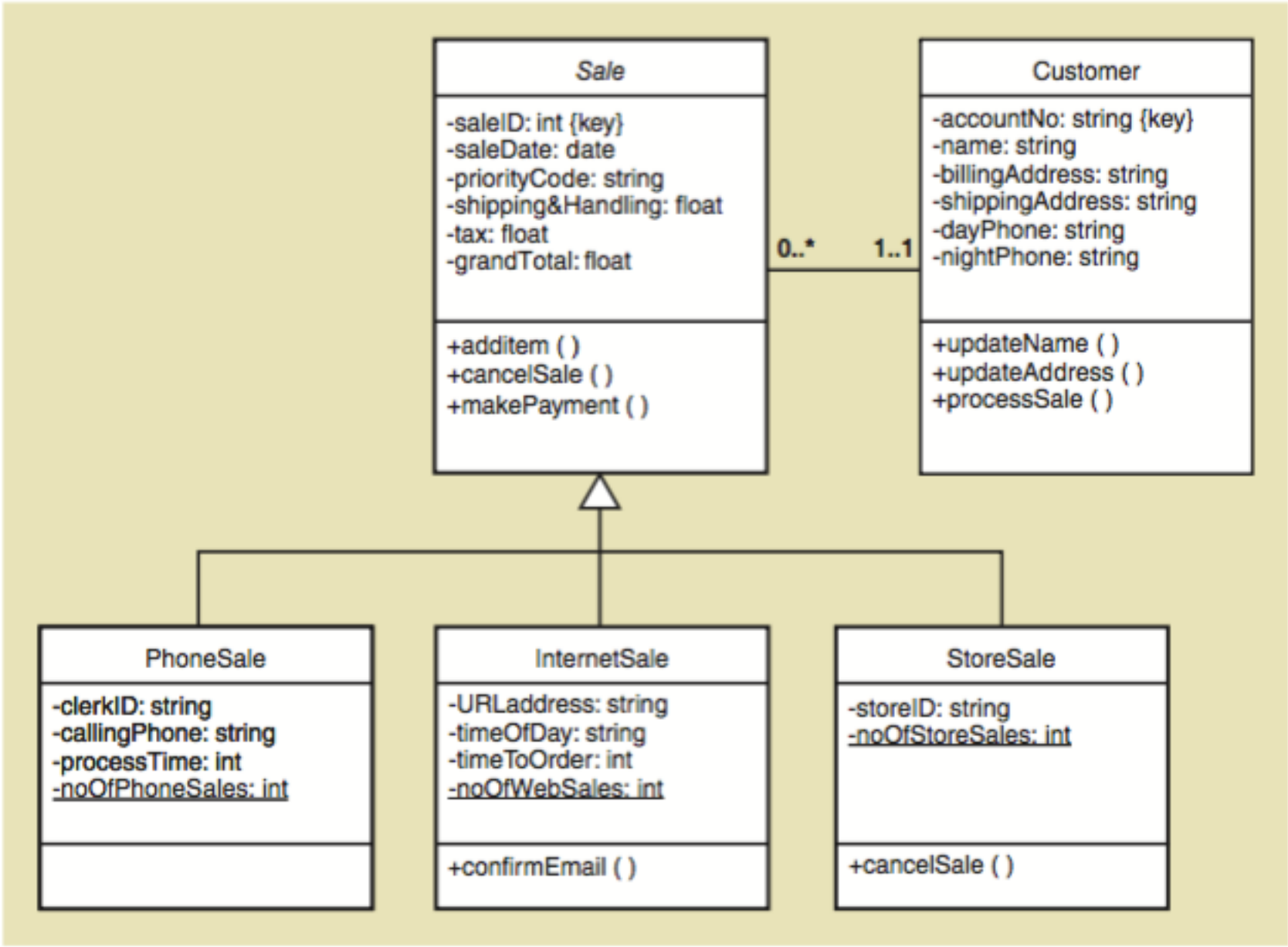
Definition: A design class stereotype is a way of categorizing a model element by its characteristics, indicated by guillemets (<< >>)

Standard Design Class Stereotypes:

- Entity class
- Boundary or view class
- Control classes
- Data Access Class
- Persistent class

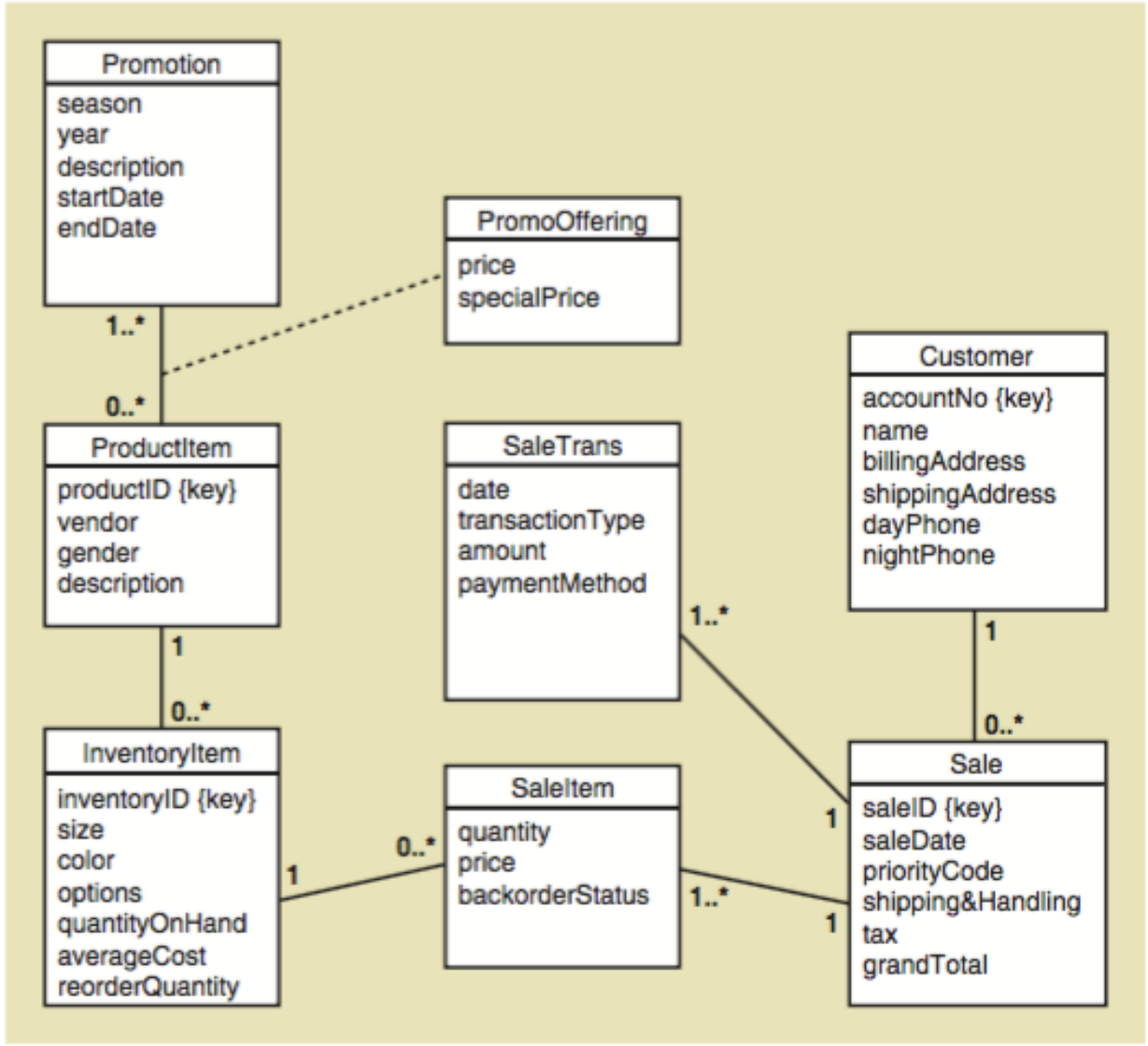
Design Class Notation

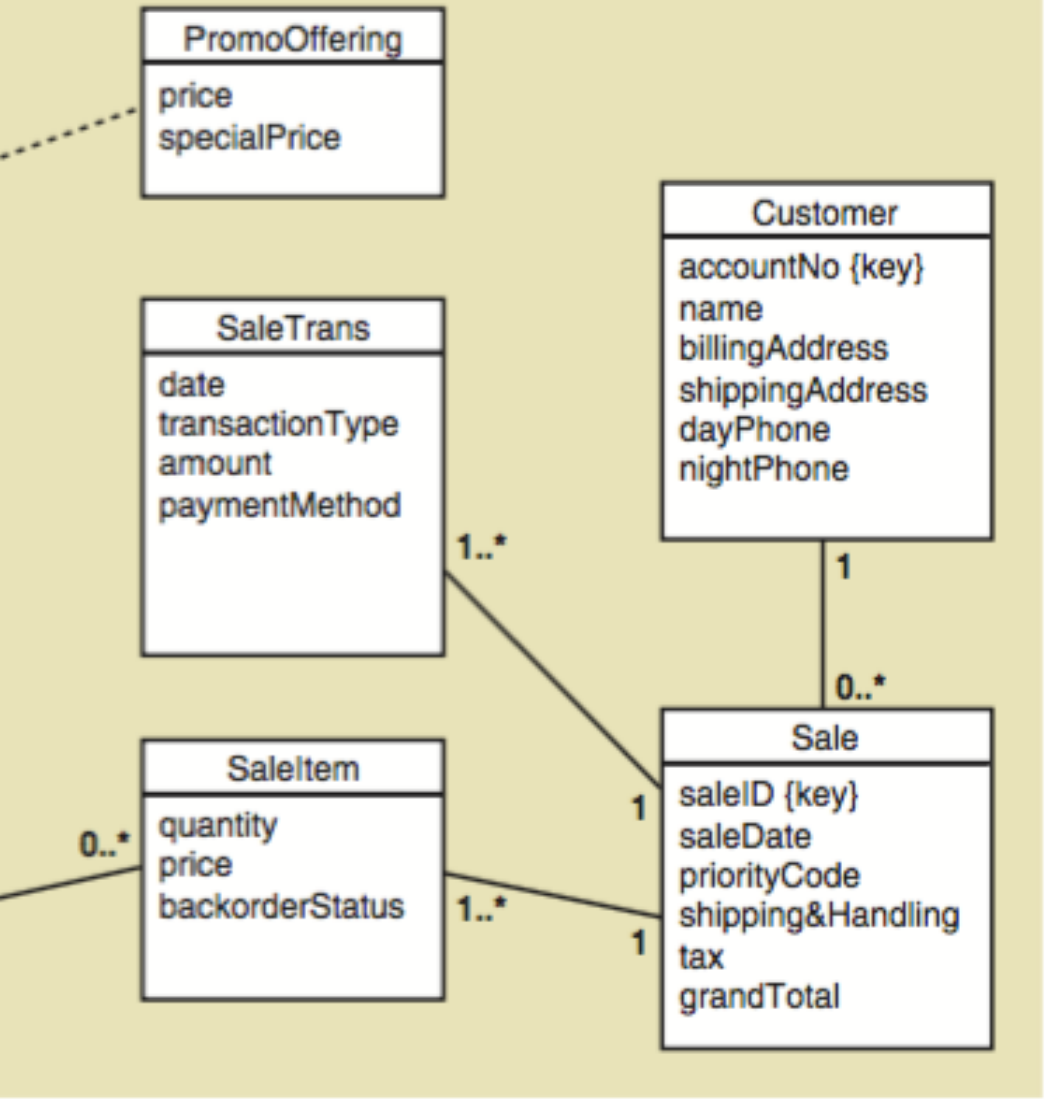


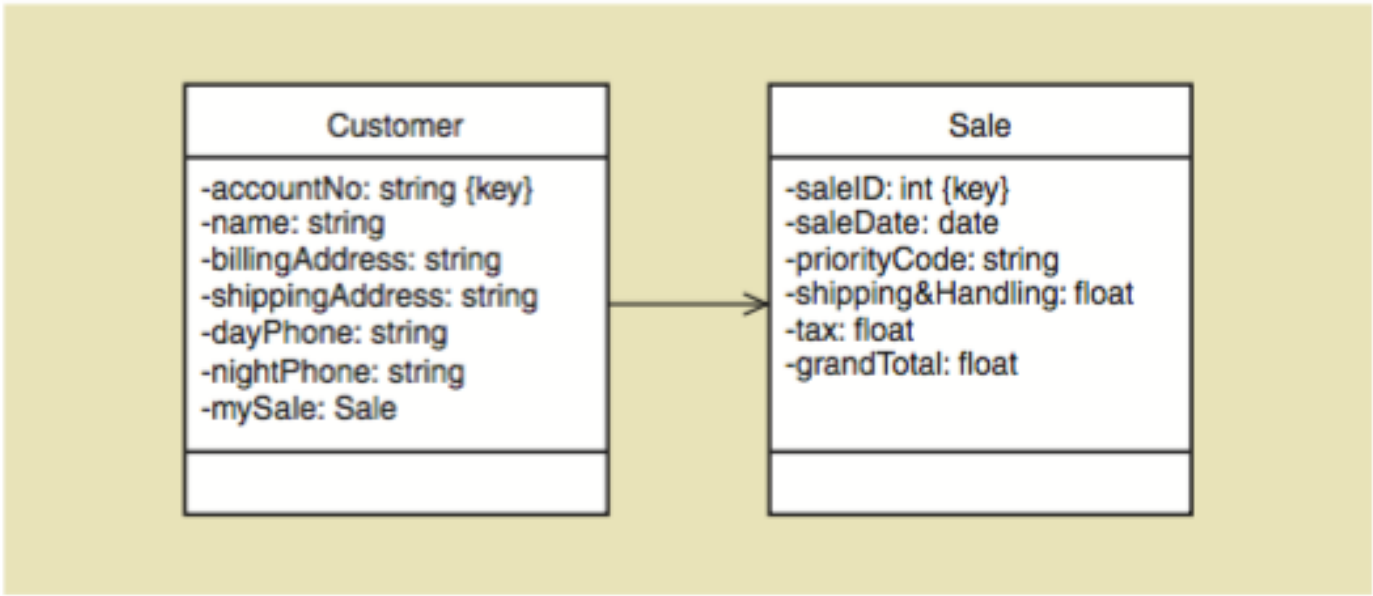

Sale
-saleID: int {key}
-saleDate: date
-priorityCode: string
-shipping&Handling: float
-tax: float
-grandTotal: float
+additem ( )
+cancelSale ( )
+makePayment ( )
0..*
1..1
Customer
-accountNo: string {key}
-name: string
-billingAddress: string
-shippingAddress: string
-dayPhone: string
-nightPhone: string
+updateName ( )
+updateAddress ( )
+processSale ( )
PhoneSale
-clerkID: string
-callingPhone: string
-processTime: int
-noOfPhoneSales: int
InternetSale
-URLaddress: string
-timeOfDay: string
-timeToOrder: int
-noOfWebSales: int
+confirmEmail ( )
StoreSale
-storeID: string
-noOfStoreSales: int
+cancelSale ( )

Developing First-Cut Design Class Diagram

1. Elaborate attributes
2. Navigation visibility
   - A design principle in which one object has a reference to another object and thus can interact with it


Promotion
season
year
description
startDate
endDate
PromoOffering
price
specialPrice
1..*
0..*
ProductItem
productID {key}
vendor
gender
description
SaleTrans
date
transactionType
amount
paymentMethod
Customer
accountNo {key}
name
billingAddress
shippingAddress
dayPhone
nightPhone
1
0..*
1..*
1
0..*
InventoryItem
inventoryID {key}
size
color
options
quantityOnHand
averageCost
reorderQuantity
SaleItem
quantity
price
backorderStatus
Sale
saleID {key}
saleDate
priorityCode
shipping&Handling
tax
grandTotal
1
0..*
1..*
1


PromoOffering
price
specialPrice
Customer
accountNo {key}
name
billingAddress
shippingAddress
dayPhone
nightPhone
SaleTrans
date
transactionType
amount
paymentMethod
1..*
1
0..*
SaleItem
quantity
price
backorderStatus
0..*
1..*
1
Sale
saleID {key}
saleDate
priorityCode
shipping&Handling
tax
grandTotal


Customer
-accountNo: string {key}
-name: string
-billingAddress: string
-shippingAddress: string
-dayPhone: string
-nightPhone: string
-mySale: Sale
Sale
-saleID: int {key}
-saleDate: date
-priorityCode: string
-shipping&Handling: float
-tax: float
-grandTotal: float

Adding Navigation Visibility

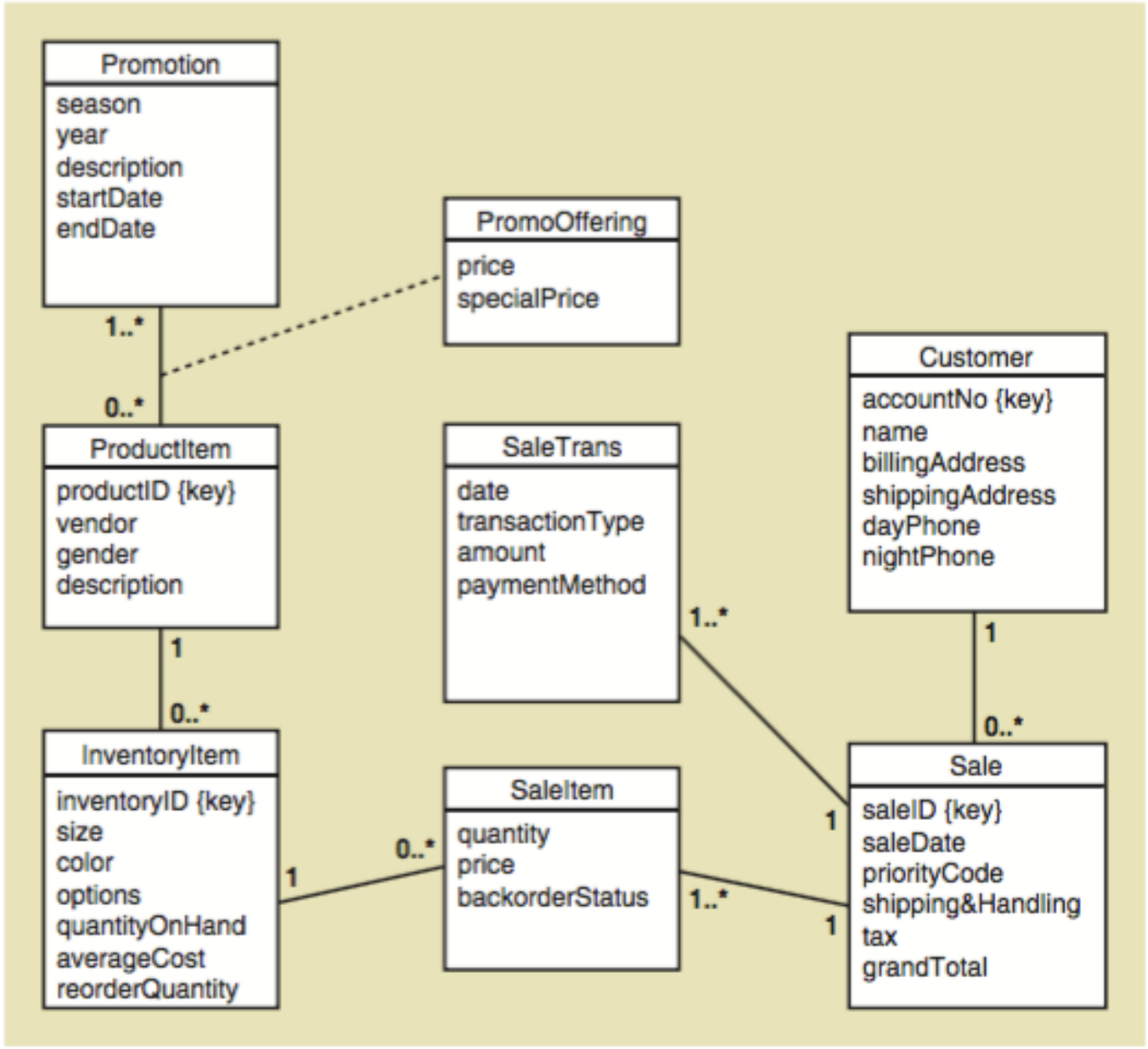
Identify:

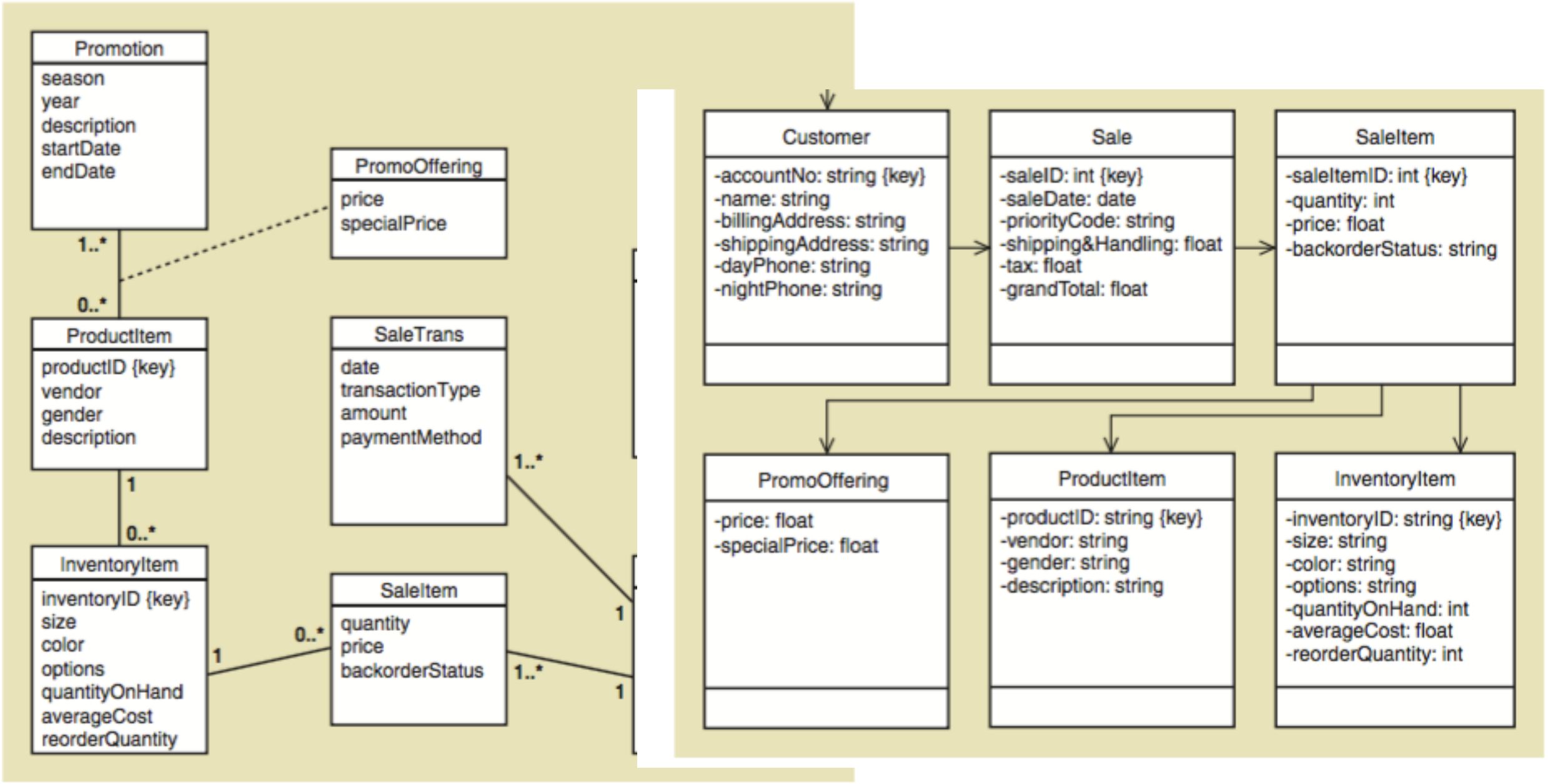
- 1 to many relationships
  - If it is a superior/subordinate relationship, add visibility from superior to subordinate
  - Ex. From Sale to SaleItem
- For Mandatory associations
  - Where one can't live without the other

    Add navigation from independent to dependent:

    Ex. From Customer to Sale

Figure Out Where Navigation Visibility Must be Added

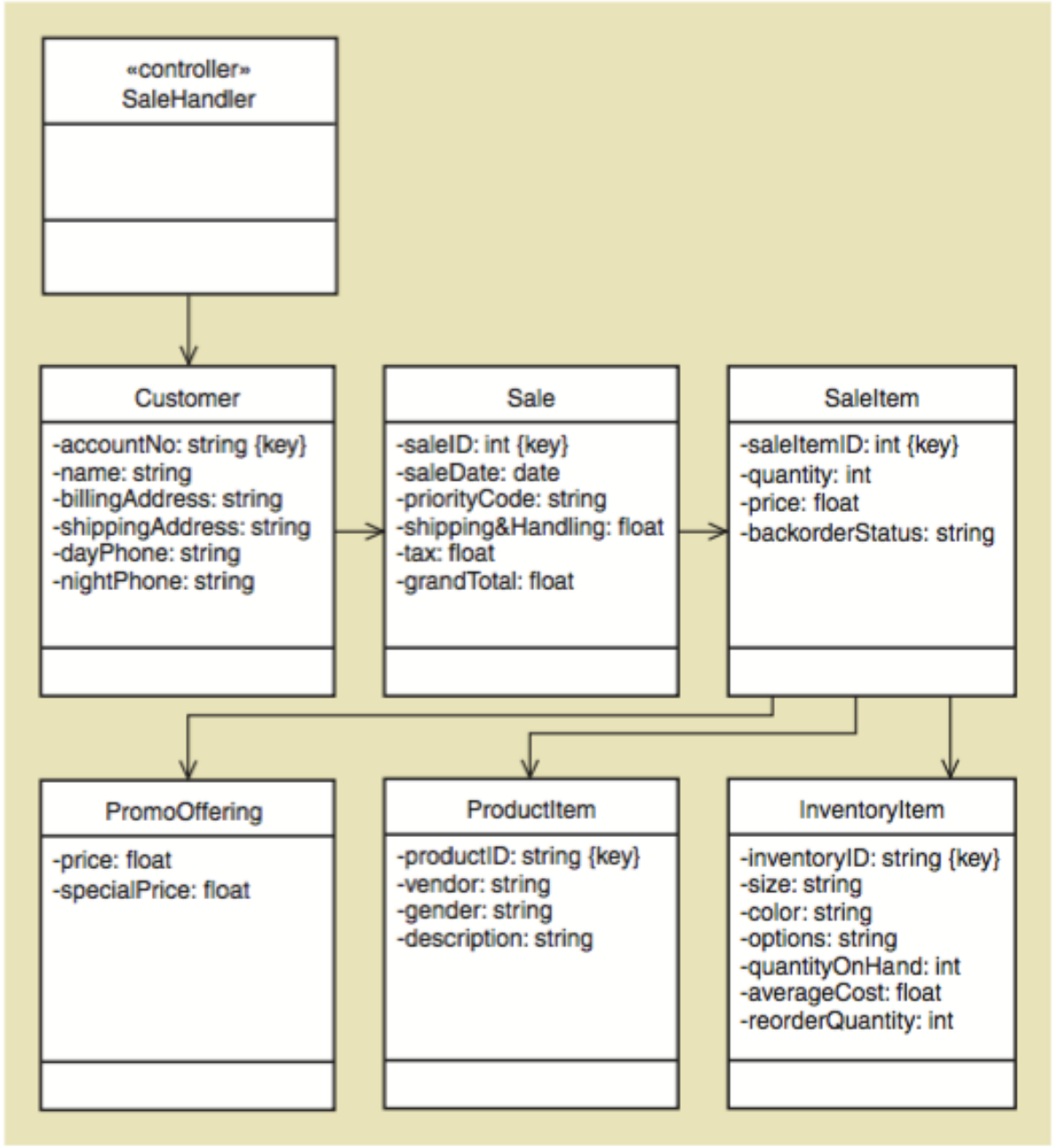



Promotion
season
year
description
startDate
endDate
PromoOffering
price
specialPrice
1..*
0..*
ProductItem
productID {key}
vendor
gender
description
SaleTrans
date
transactionType
amount
paymentMethod
1
0..*
InventoryItem
inventoryID {key}
size
color
options
quantityOnHand
averageCost
reorderQuantity
1
0..*
SaleItem
quantity
price
backorderStatus
1..*
1
1..*
1
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

Building a First Cut Design Diagram

Next step:

- Add a controller class to act as a switch board between input screens and programming logic classes


«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

Fundamental Principles of Good Design

Object Responsibility

Separation of Responsibilities/Concerns

Protection from Variations

Indirection

Coupling

Cohesion

Object Responsibility

Definition: Object Responsibility is a design principle in which objects are responsible for carrying out system processing

Separation of Concern

Definition: Separation of Concern is a design principle that recommends segregating classes into separate packages or groups based on the primary focus of processing responsibility

Protection from Variation

Definition: Protection from Variation is a design principle in which parts of a system that are unlikely to change, are separated form those that will

Indirection

Definition: Indirection is a design principle in which an intermediate class is placed between two classes to decouple them, while keeping them linked.

Coupling

Definition: Coupling is a qualitative measure of how closely the classes in a design class diagram are linked

Cohesion

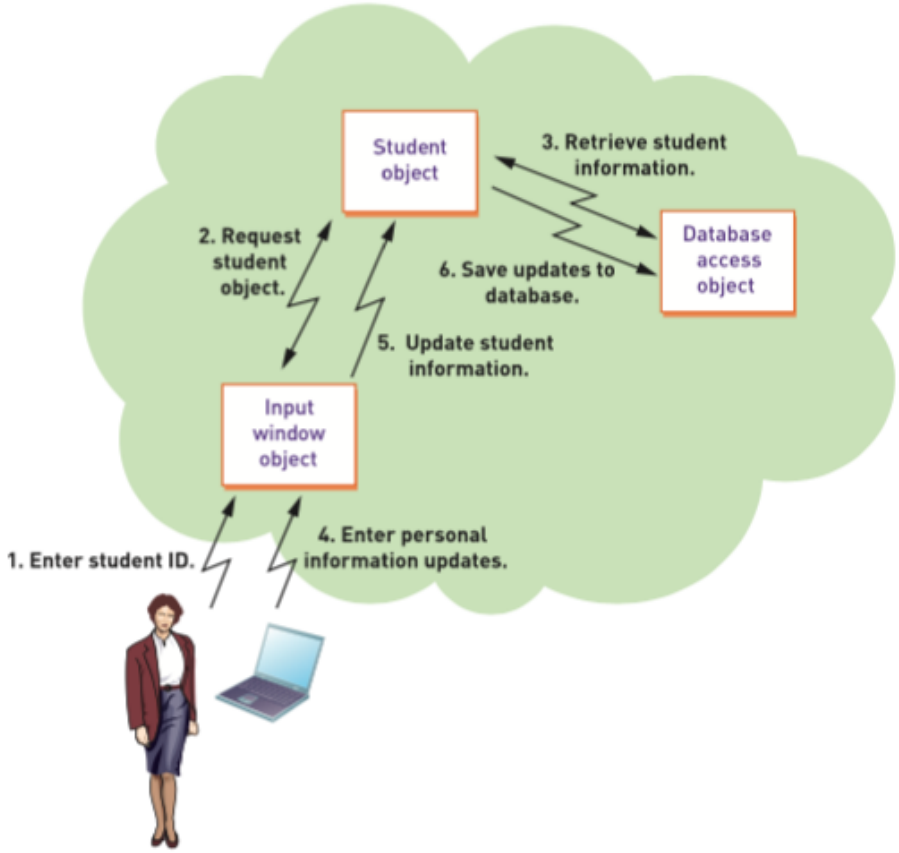
Definition: Cohesion is an other qualitative measure of the focus or unity of purpose within a single class

Advanced OO Design

Last time:

- How to build Design Class Diagrams
- Used CRC Cards to:
  - Add collaborators
  - Add functionality
  - Initial look at *multi-layer classes*
    - controllers

Designing Multilayer Systems



Techniques

For simple projects:

- CRC Cards

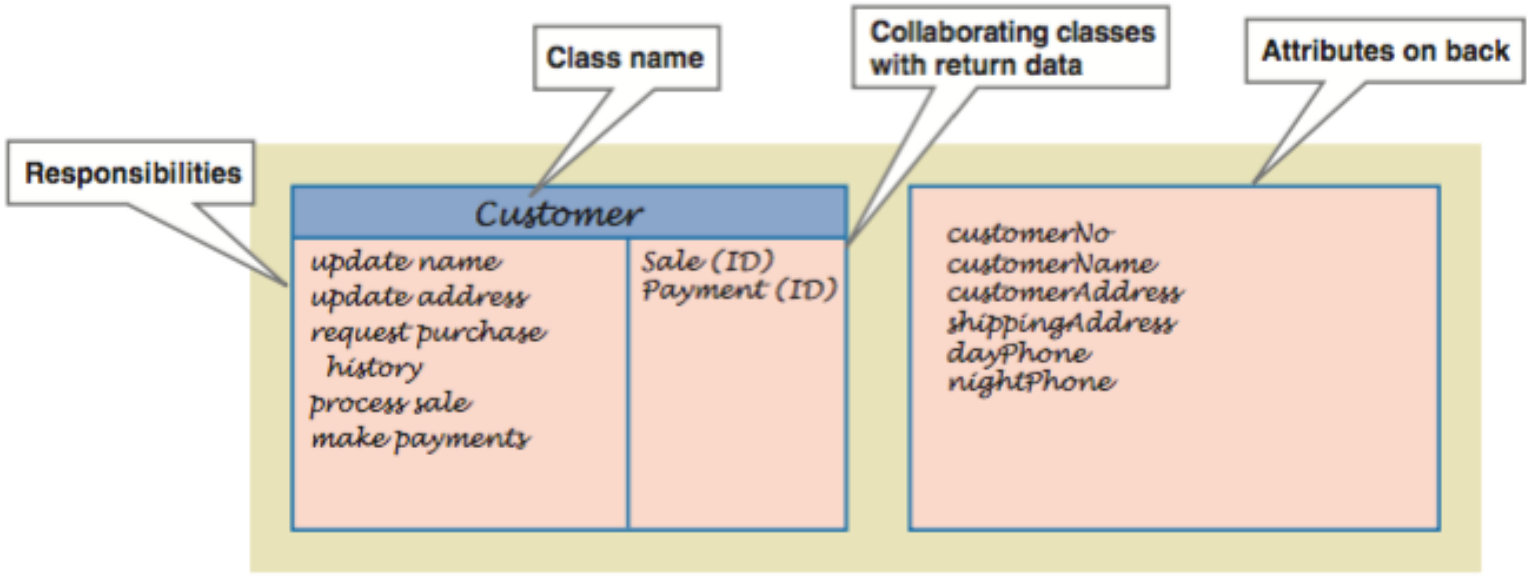
For more complex projects:

- Communications Diagrams
- Sequence Diagrams

CRC Cards

Create Class Responsibility Collaboration (CRC) Cards

- A brainstorming design technique
- For designing interactions in use cases
- Assign responsibilities and collaboration classes



The process

Brainstorming, so done in a group

Need:

- Domain Model Class Diagrams
- Use Case Diagrams
- List of Use Cases
- Activity Diagrams
- SSD
- Use Case Descriptions

CRC Cards: The process

1. Select a use case
    1. Start with the use case controller card
2. Identify 1st domain class that has responsibility for this use case
3. Identify other collaborating classes
    1. Which have needed data?
    2. Which need to be updated

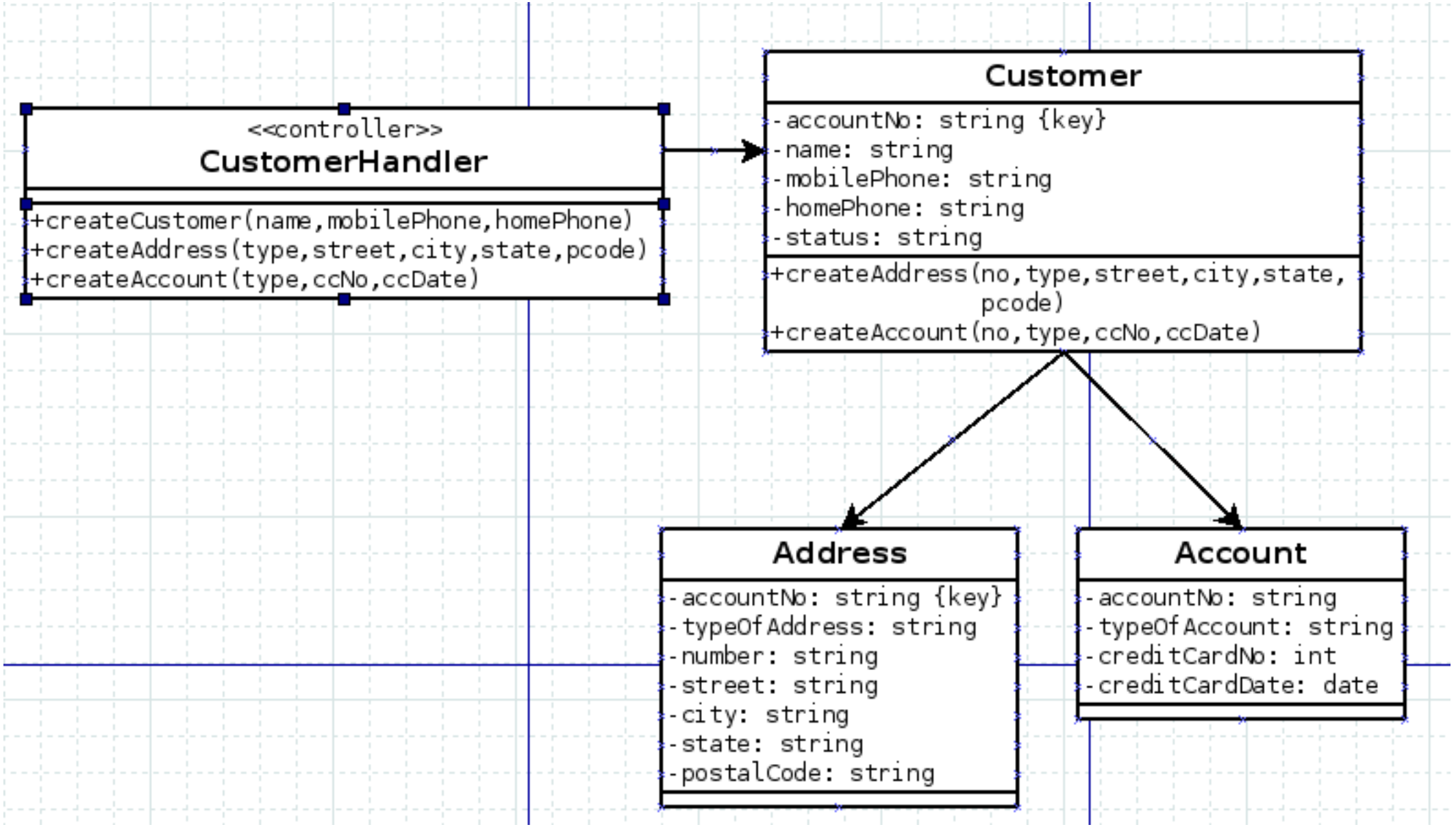
CRC Cards: Example

Use Case: Create Customer Account (see hand out)

Create the CRC Cards

- Start with CustomerHandler controller class
- Identify primary class responsible for creating a new customer account
- Look for other messages
  - Add collaborators, create classes, update cards
- Add user-interface classes
- Add db access classes

Revise the Design Class Diagram


<<controller>>
CustomerHandler
+createCustomer(name,mobilePhone,homePhone)
+createAddress(type,street,city,state,pcode)
+createAccount(type,ccNo,ccDate)
Customer
-accountNo: string {key}
-name: string
-mobilePhone: string
-homePhone: string
-status: string
+createAddress(no,type,street,city,state,
pcode)
+createAccount(no,type,ccNo,ccDate)
Address
-accountNo: string {key}
-typeOfAddress: string
-number: string
-street: string
-city: string
-state: string
-postalCode: string
Account
-accountNo: string
-typeOfAccount: string
-creditCardNo: int
-creditCardDate: date

Use Case Realization

Two techniques:

- Communication Diagrams
- Sequence Diagrams

Use Case Realization

Definition: Use case realization is the process of elaborating the detailed design with interaction diagrams for a particular use case

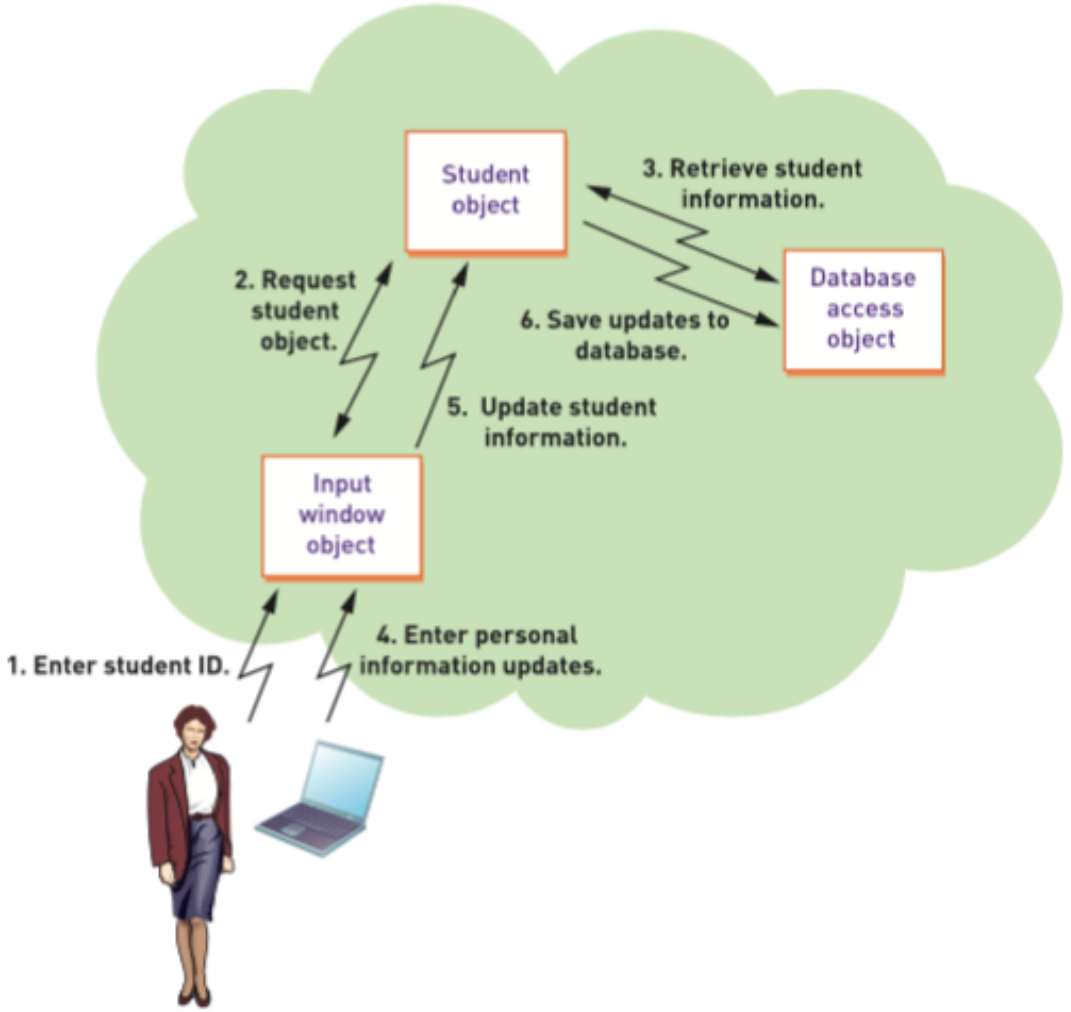
Use Case Controller

With CRC Cards

- Added controller class to our model

Part of the Model View Controller (MVC) Design Pattern

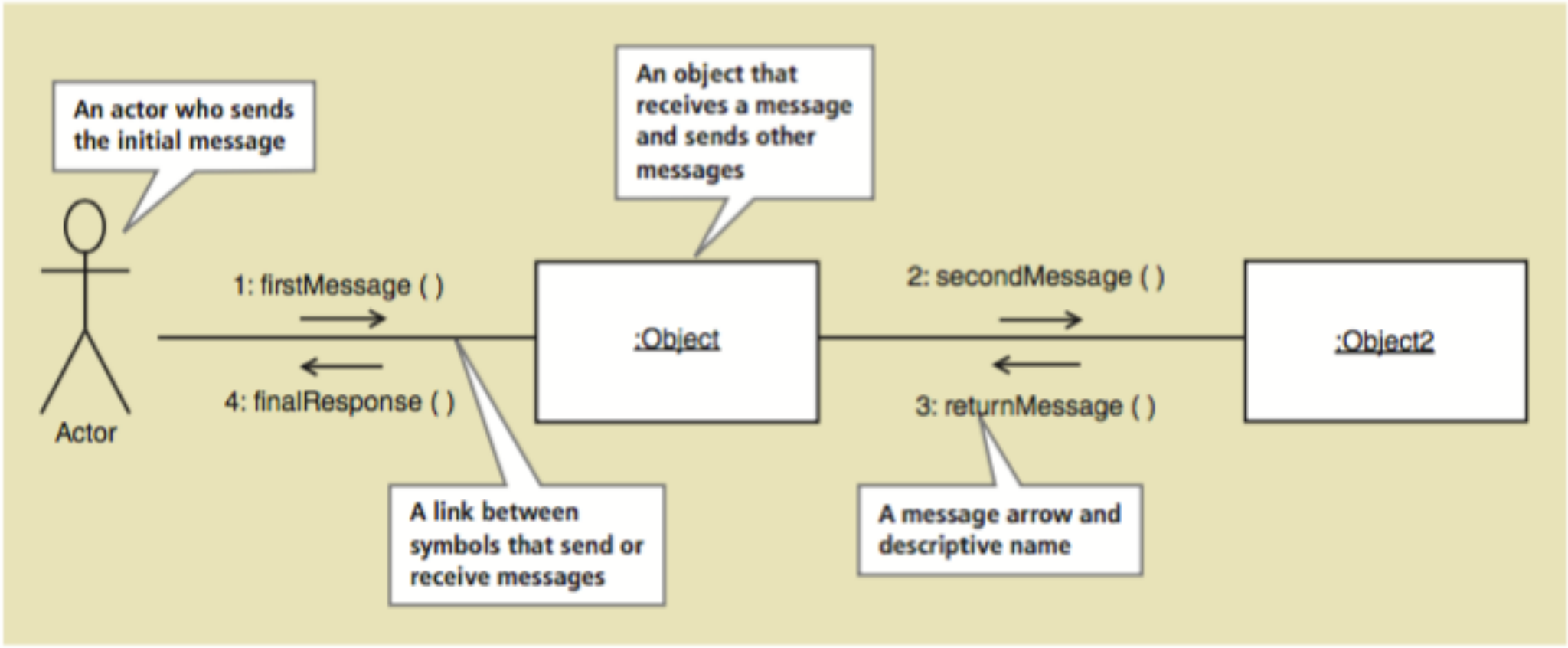
- Very popular
- Many IDEs add them automagically



Use Case Realization with Communication Diagrams

1. Overview of Communication Diagrams
2. How to do OO Design with Communication Diagrams

Communication Diagrams



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

T/F condition
- Message sent if true
- optional

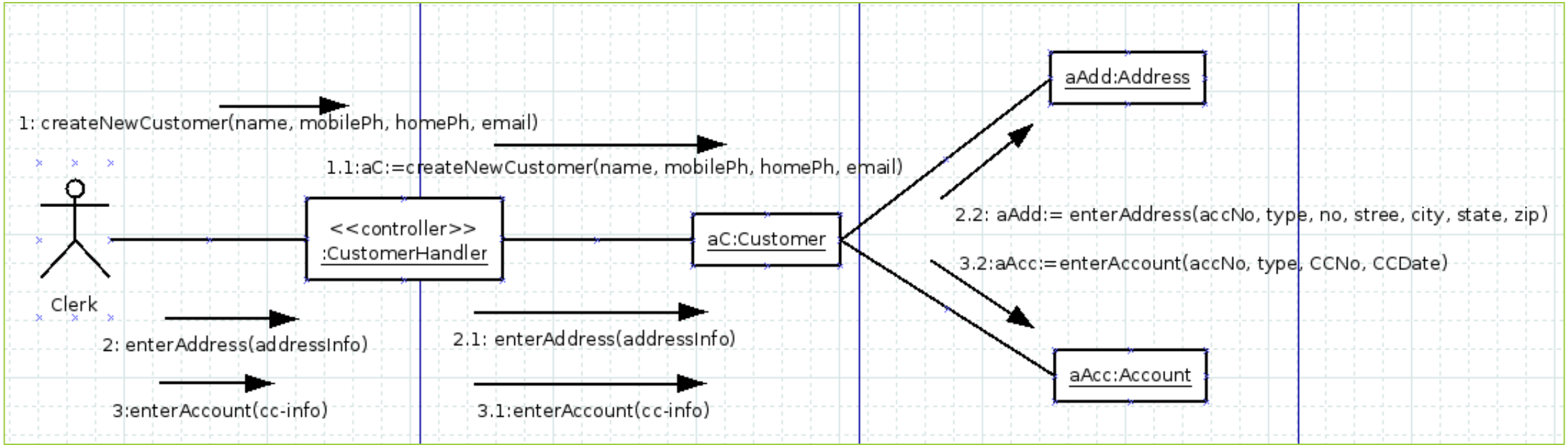
Sequence Number
- Indicates order of messages
- Hierarchical dot-notation to show dependency and sequence

Return Value
- Similar to method return value
- Can be shown as separate message, or return value

Message Name
- Should describe service
- := indicates return value given

Parameter list

Communication Diagram for Create Customer Account Use Case

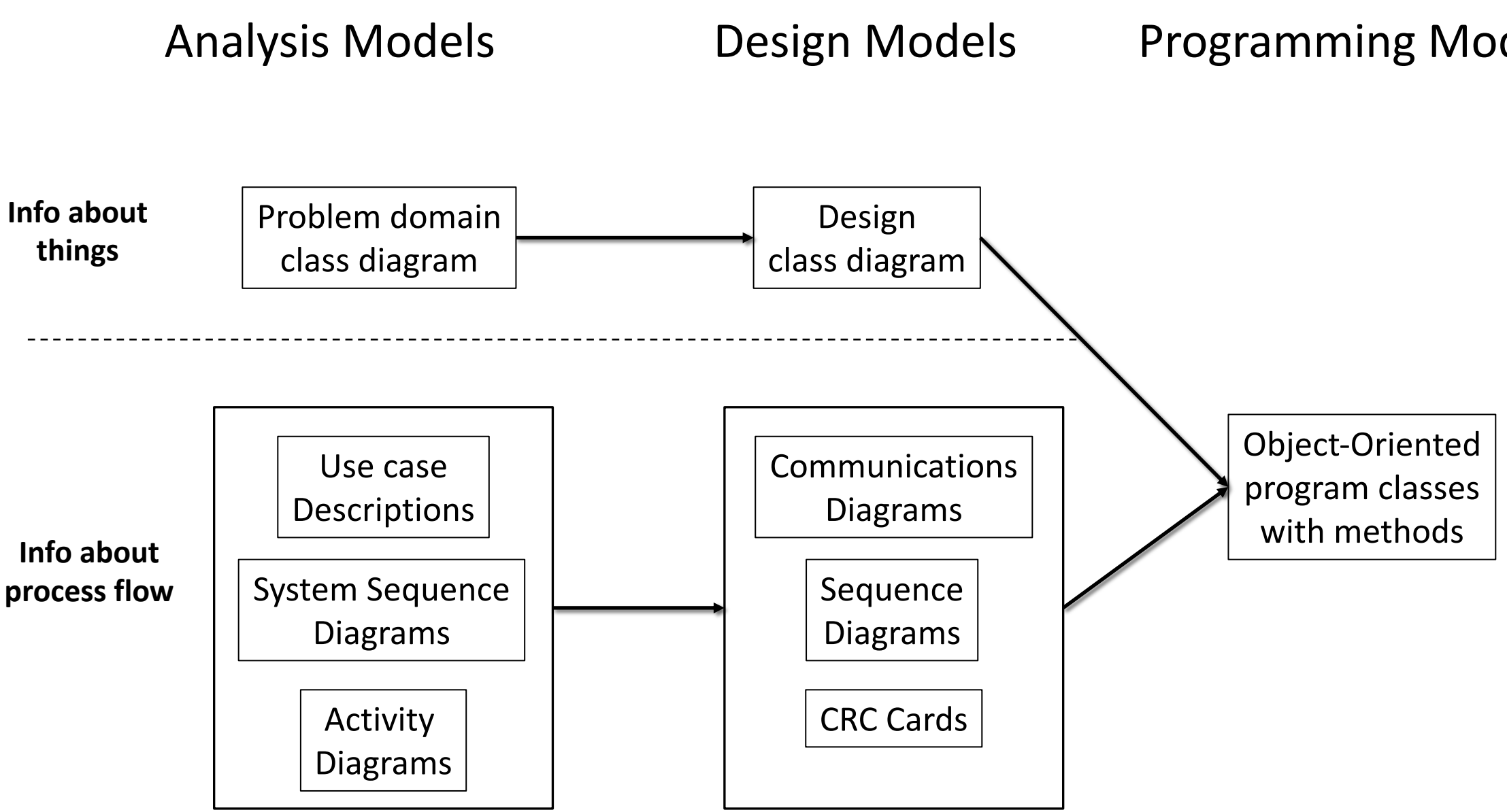


Doing OO Design using Communication Diagrams

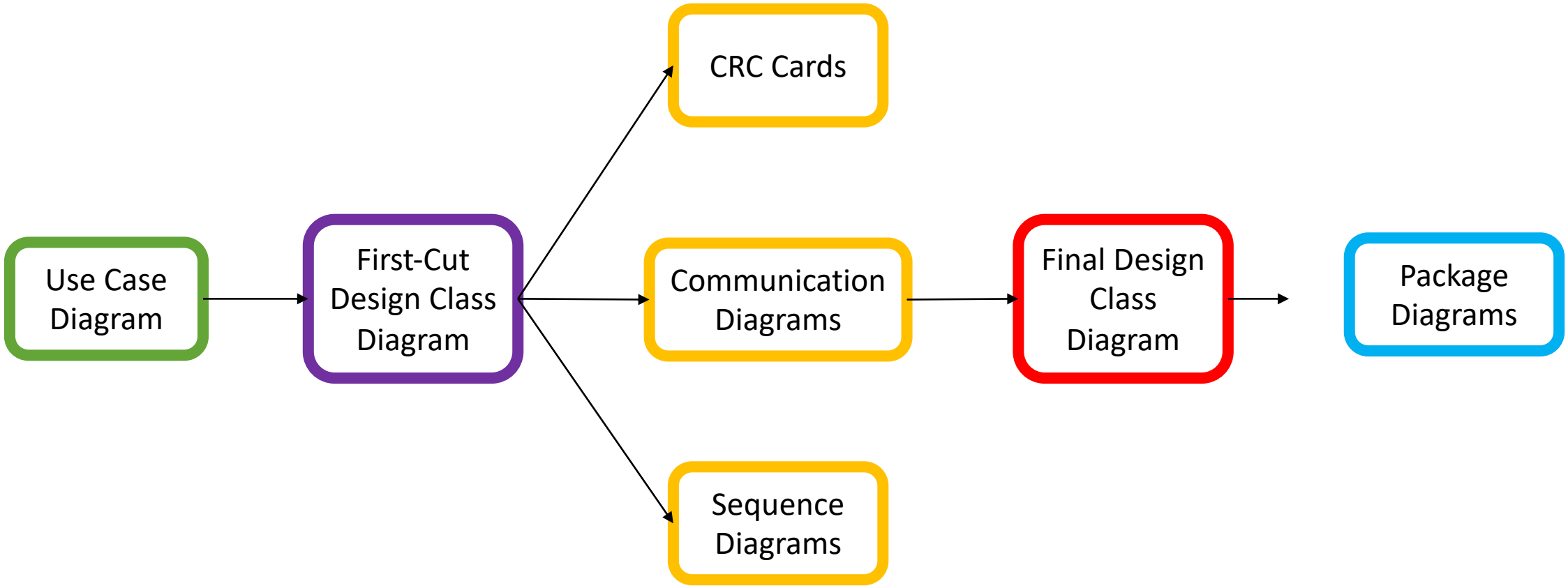
Input Models

Extending Input Messages

Final Design Class Diagrams


Analysis Models
Design Models
Programming Models
Info about things
Problem domain class diagram
Design class diagram
Info about process flow
Use case Descriptions
System Sequence Diagrams
Activity Diagrams
Communications Diagrams
Sequence Diagrams
CRC Cards
Object-Oriented program classes with methods

Steps of OO Design



Extending Input Messages

For each input message:

1. Identify classes needed to execute message, and put them on diagram
2. Starting with input messages, identify each message needed:
   - Check navigation visibility
   - Decide which object takes care of service
3. Name each message
   - Should reflect service requested
   - Parameters
   - Return values

Create New Customer Use Case

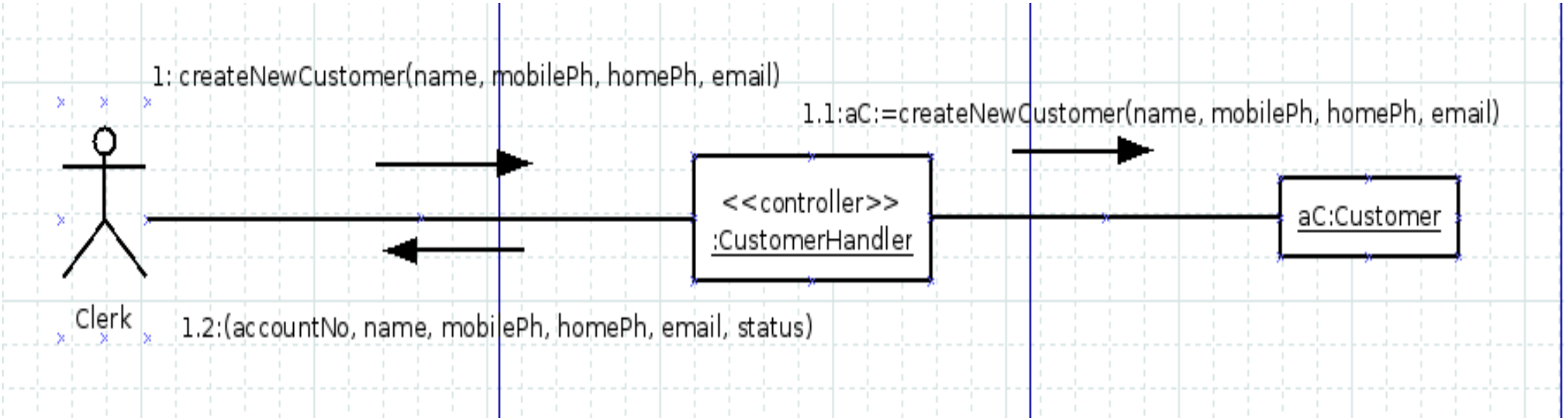
1: What classes are needed?

2: What messages are needed?

- Who should handle them?

3: Name messages

createNewCustomer message extended to all objects



enterAddress Message

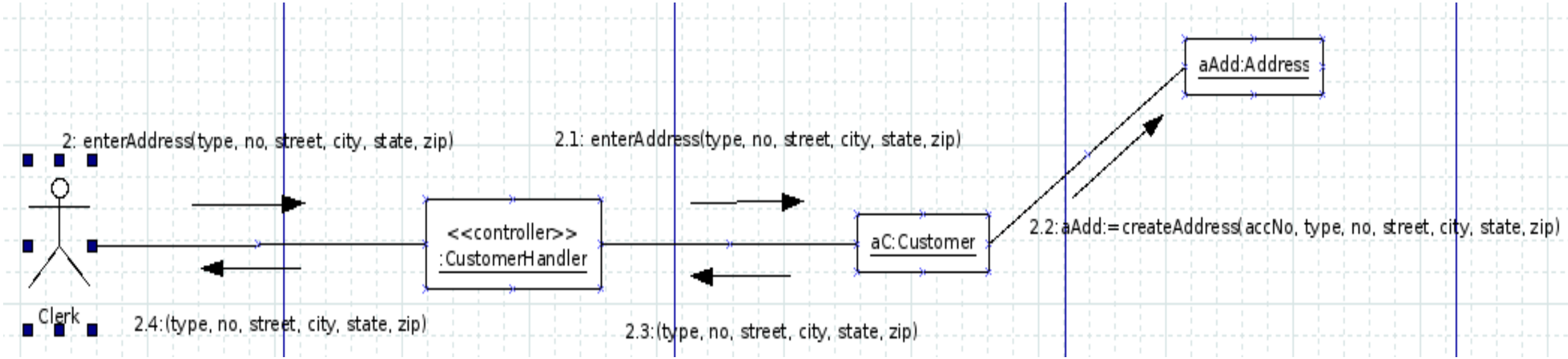
1: What classes are needed?

2: What messages are needed?

- Who should handle them?

3: Name messages

enterAddress message extended to all objects



enterAccount Message

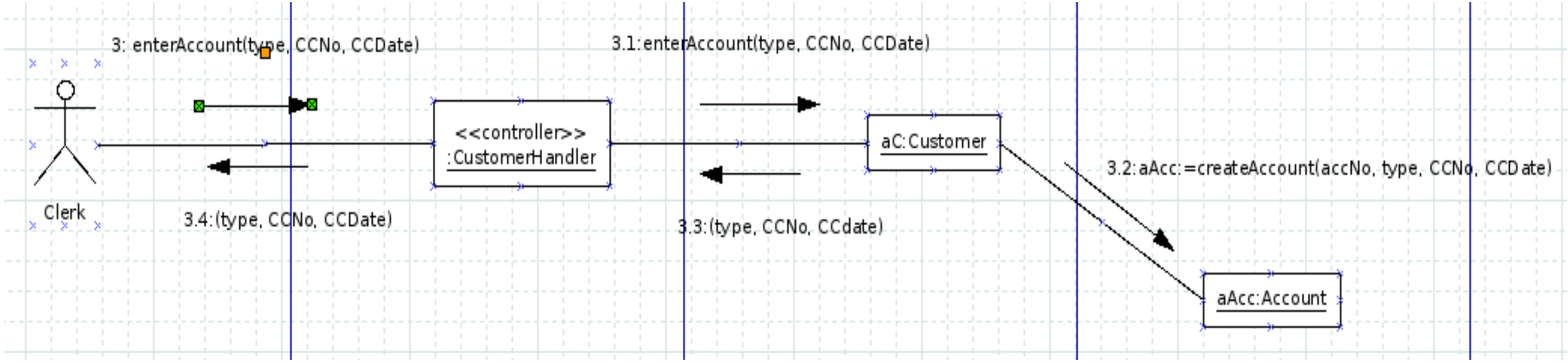
1: What classes are needed?

2: What messages are needed?

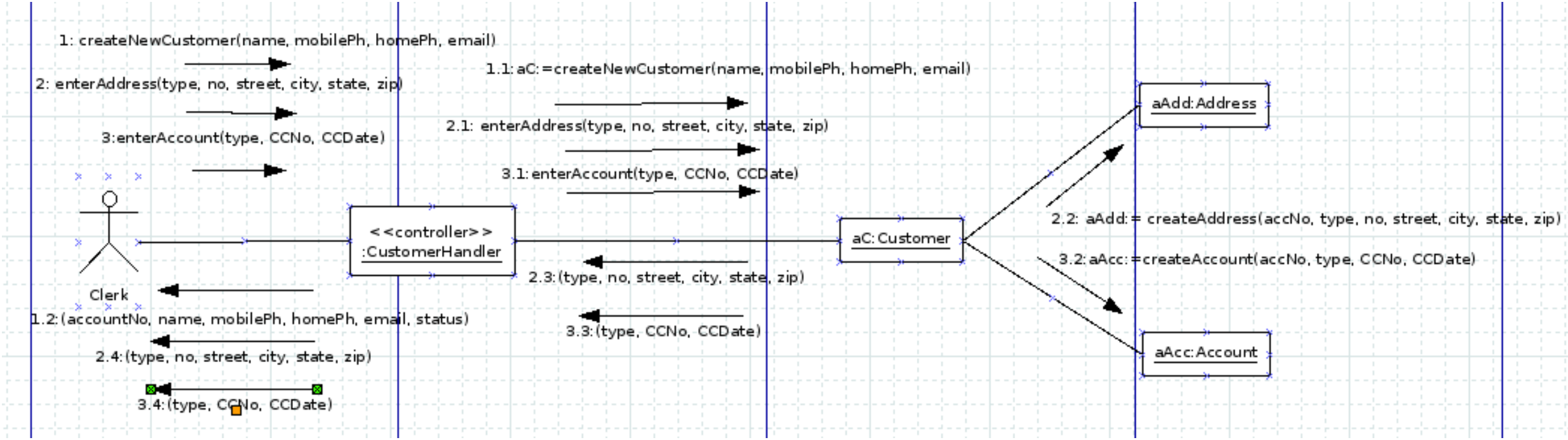
- Who should handle them?

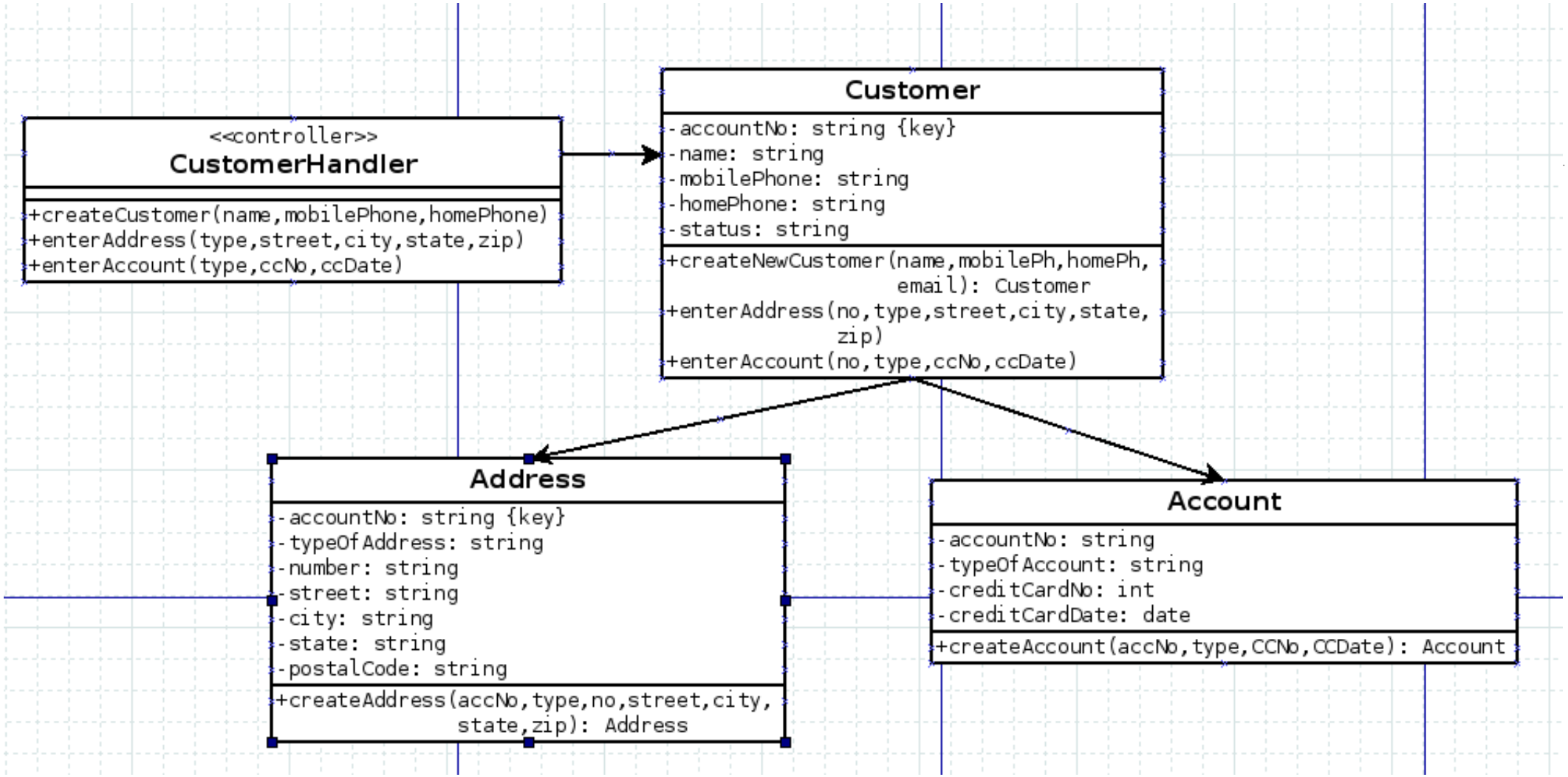
3: Name messages

enterAccount message extended to all objects

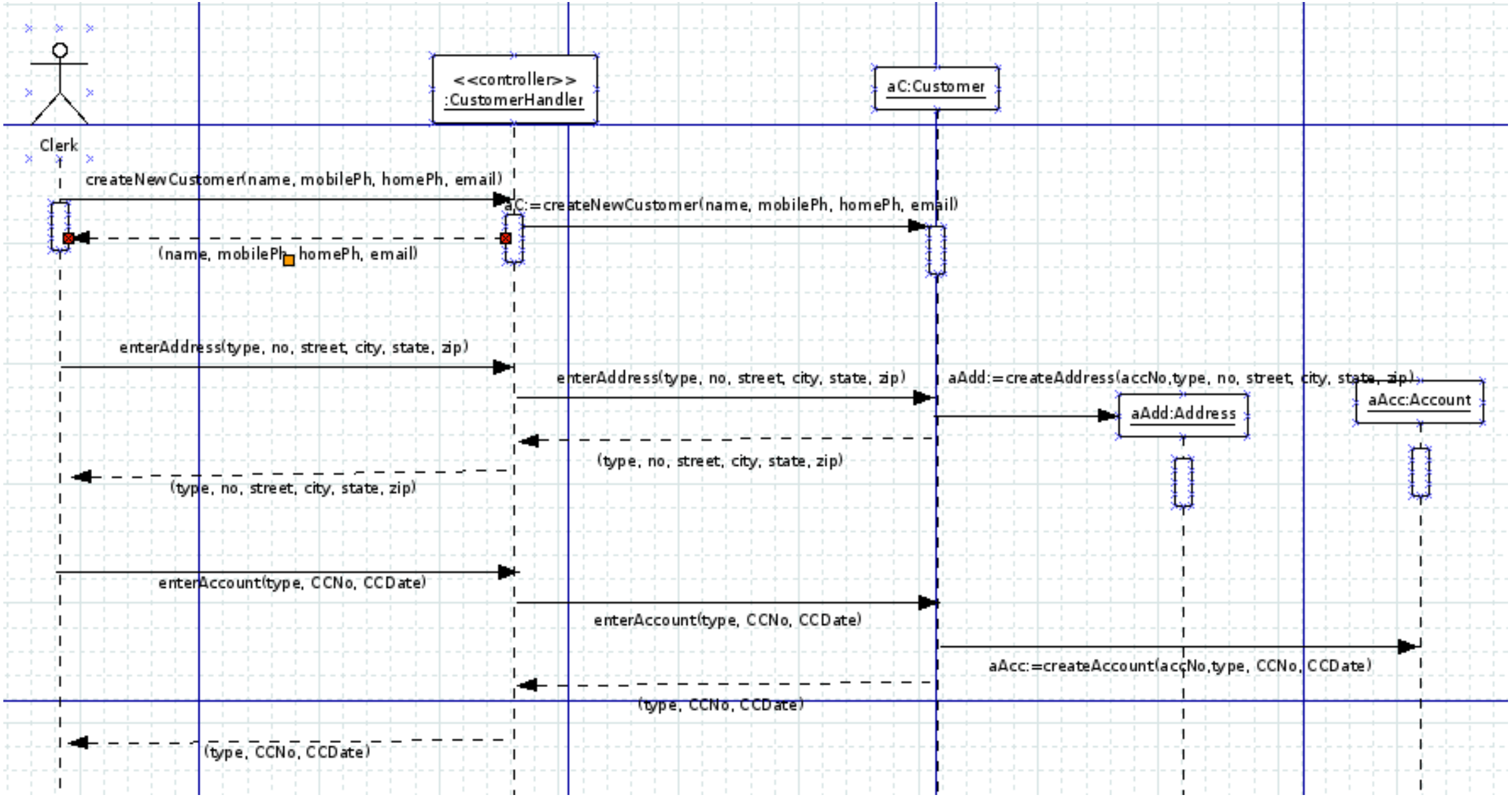


Final Communication Diagram for Create Customer Account use case




<<controller>>
CustomerHandler
+createCustomer(name,mobilePhone,homePhone)
+enterAddress(type,street,city,state,zip)
+enterAccount(type,ccNo,ccDate)
Customer
-accountNo: string {key}
-name: string
-mobilePhone: string
-homePhone: string
-status: string
+createNewCustomer(name,mobilePh,homePh, email): Customer
+enterAddress(no,type,street,city,state, zip)
+enterAccount(no,type,ccNo,ccDate)
Address
-accountNo: string {key}
-typeOfAddress: string
-number: string
-street: string
-city: string
-state: string
-postalCode: string
+createAddress(accNo,type,no,street,city, state,zip): Address
Account
-accountNo: string
-typeOfAccount: string
-creditCardNo: int
-creditCardDate: date
+createAccount(accNo,type,CCNo,CCDate): Account

Use Case Realization with Sequence Diagrams

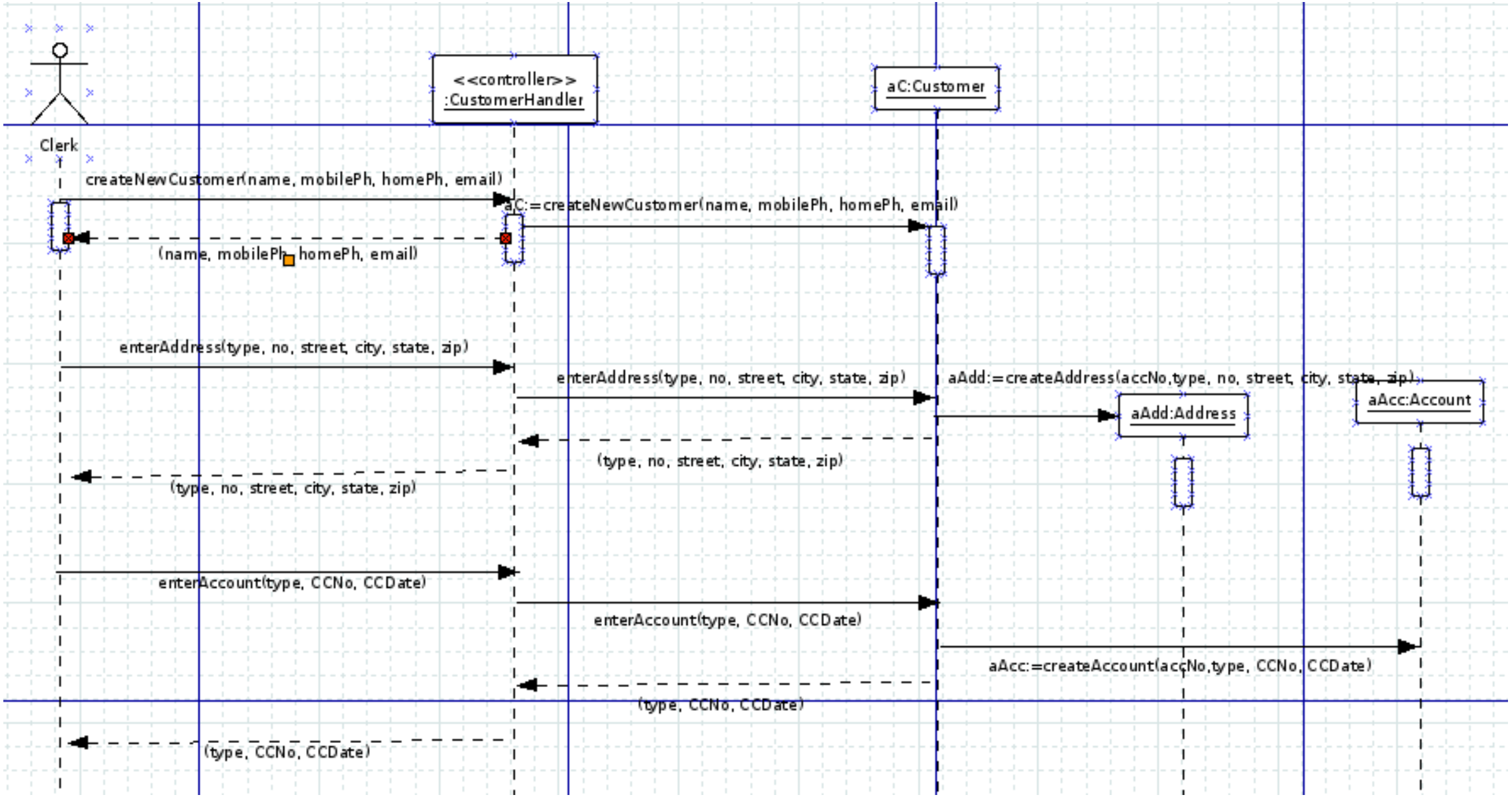

Clerk
<<controller>>
:CustomerHandler
aC:Customer
createNewCustomer(name, mobilePh, homePh, email)
aC:=createNewCustomer(name, mobilePh, homePh, email)
(name, mobilePh, homePh, email)
enterAddress(type, no, street, city, state, zip)
enterAddress(type, no, street, city, state, zip)
aAdd:=createAddress(accNo,type, no, street, city, state, zip)
aAdd:Address
aAcc:Account
(type, no, street, city, state, zip)
(type, no, street, city, state, zip)
enterAccount(type, CCNo, CCDate)
enterAccount(type, CCNo, CCDate)
aAcc:=createAccount(accNo,type, CCNo, CCDate)
(type, CCNo, CCDate)
(type, CCNo, CCDate)

Design Process

Pick a use case

Start with SSD and First Cut Domain Model Class Diagram

- For each input message:
  - What internal classes needed?
  - Extend all messages


Clerk
<<controller>>
:CustomerHandler
aC:Customer
createNewCustomer(name, mobilePh, homePh, email)
aC:=createNewCustomer(name, mobilePh, homePh, email)
(name, mobilePh, homePh, email)
enterAddress(type, no, street, city, state, zip)
enterAddress(type, no, street, city, state, zip)
aAdd:=createAddress(accNo,type, no, street, city, state, zip)
aAdd:Address
aAcc:Account
(type, no, street, city, state, zip)
(type, no, street, city, state, zip)
enterAccount(type, CCNo, CCDate)
enterAccount(type, CCNo, CCDate)
aAcc:=createAccount(accNo,type, CCNo, CCDate)
(type, CCNo, CCDate)
(type, CCNo, CCDate)

Practice: Fill Shopping Cart use case

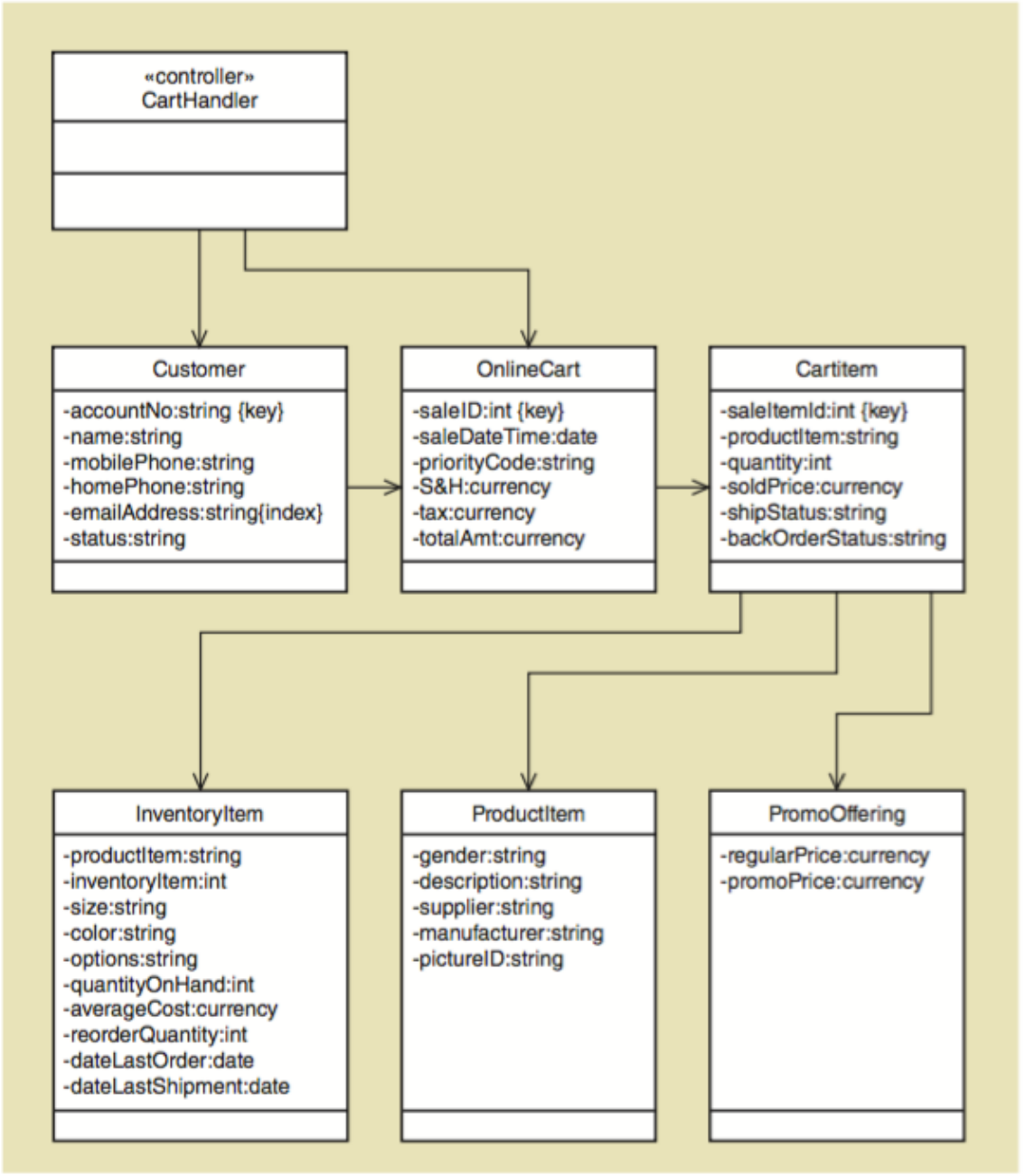
Start with Activity Diagram and SSD

Two messages:

- Add item
- Add accessory

Steps

1: what classes are needed?

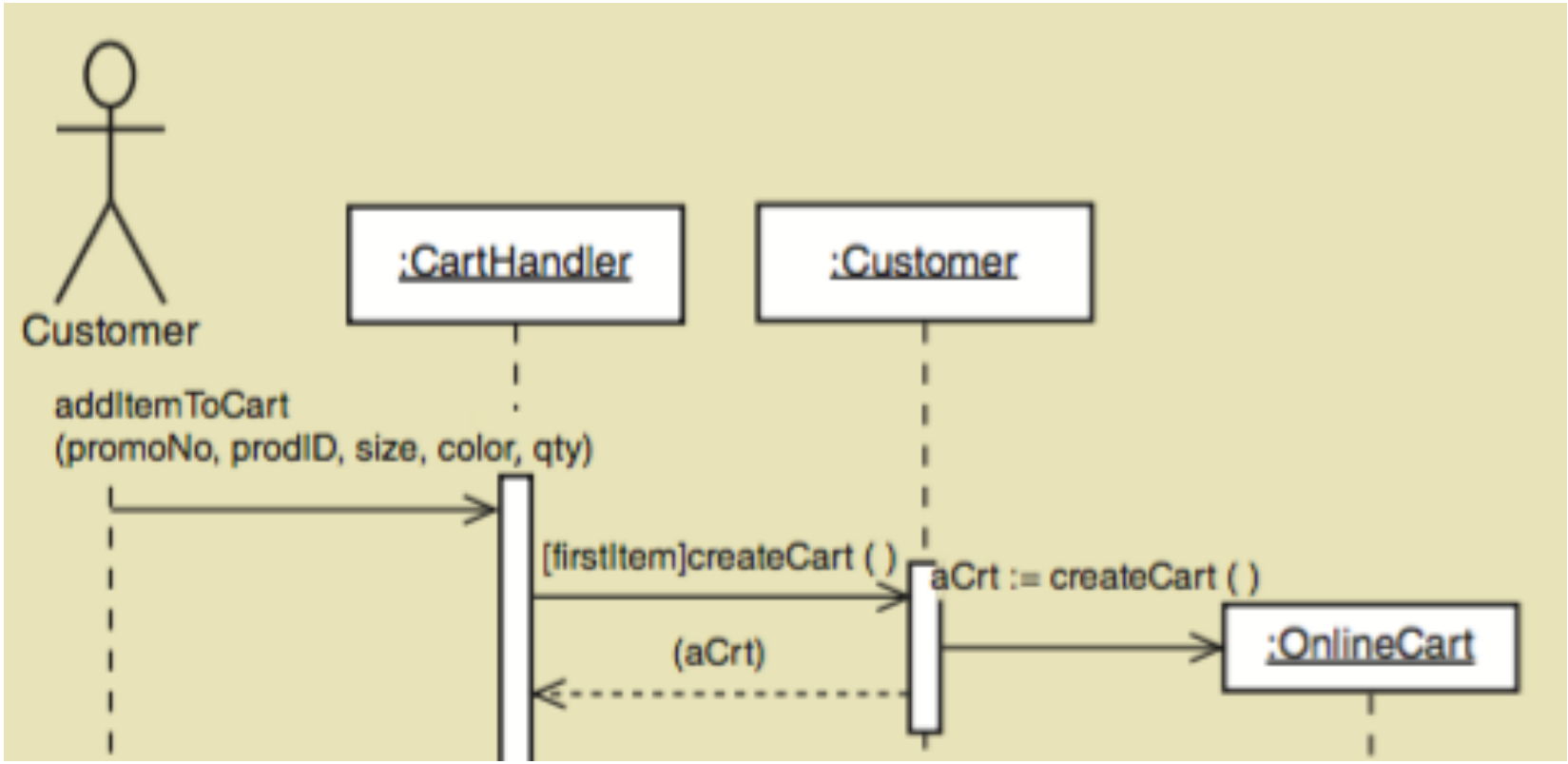

«controller»
CartHandler
Customer
-accountNo:string {key}
-name:string
-mobilePhone:string
-homePhone:string
-emailAddress:string{index}
-status:string
OnlineCart
-saleID:int {key}
-saleDateTime:date
-priorityCode:string
-S&H:currency
-tax:currency
-totalAmt:currency
CartItem
-saleItemId:int {key}
-productItem:string
-quantity:int
-soldPrice:currency
-shipStatus:string
-backOrderStatus:string
InventoryItem
-productItem:string
-inventoryItem:int
-size:string
-color:string
-options:string
-quantityOnHand:int
-averageCost:currency
-reorderQuantity:int
-dateLastOrder:date
-dateLastShipment:date
ProductItem
-gender:string
-description:string
-supplier:string
-manufacturer:string
-pictureID:string
PromoOffering
-regularPrice:currency
-promoPrice:currency

Next Step

2: Extend Input messages

- Same as we did for communication diagram
- Start with addItemToCart

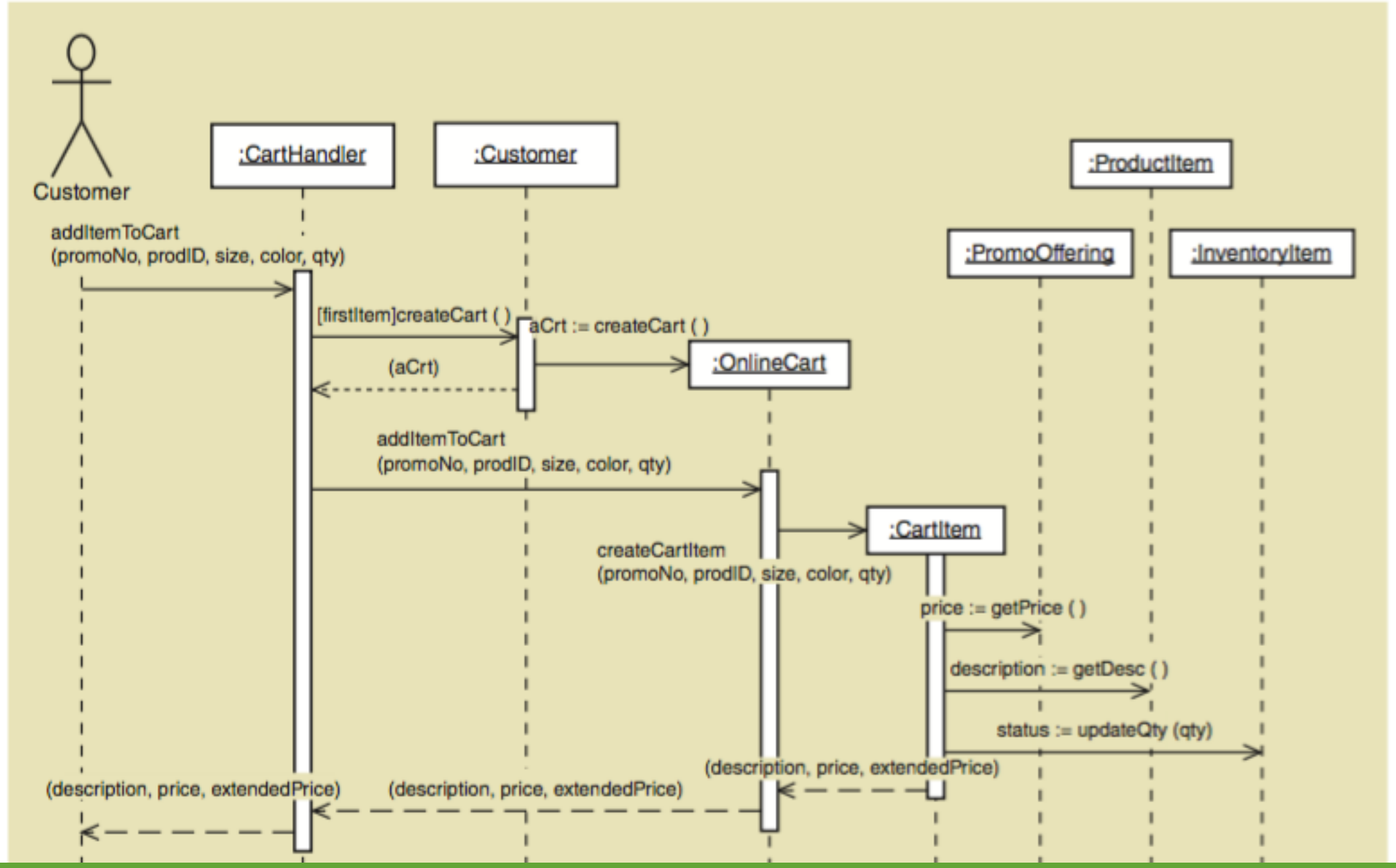
First Step in extending addItem message



Next Step: add item to cart

What steps need to be completed to add and item?

- Create a cart item
  - Get the item's price
  - Get the item's description
  - Check if the item is available


Customer
:CartHandler
:Customer
:ProductItem
:PromoOffering
:InventoryItem
addItemToCart
(promoNo, prodID, size, color, qty)
[firstItem]createCart ( )
aCrt := createCart ( )
:OnlineCart
(aCrt)
addItemToCart
(promoNo, prodID, size, color, qty)
:CartItem
createCartItem
(promoNo, prodID, size, color, qty)
price := getPrice ( )
description := getDesc ( )
status := updateQty (qty)
(description, price, extendedPrice)
(description, price, extendedPrice)
(description, price, extendedPrice)

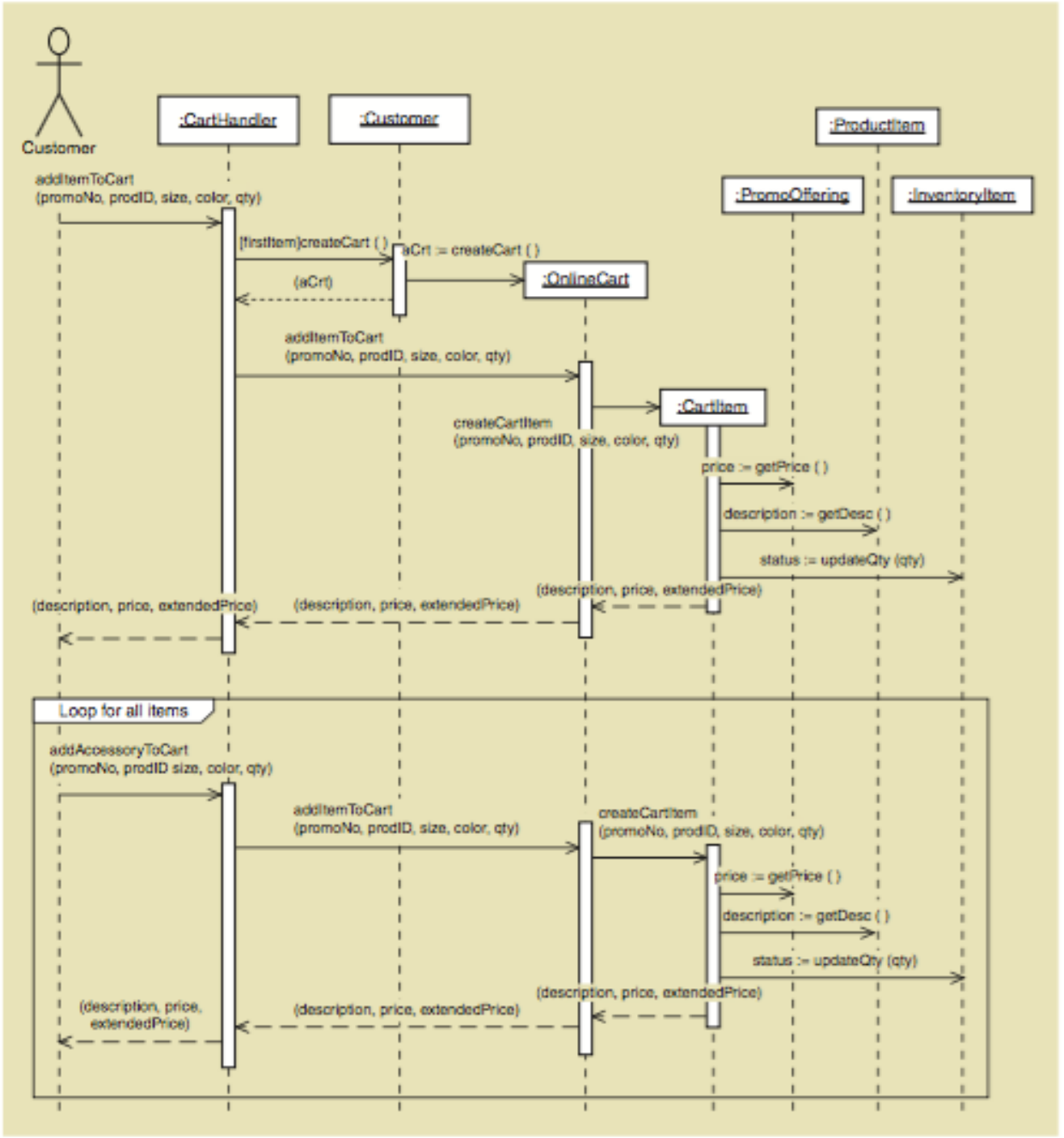
AddAccessoryItem

This is identical to addItem, except

- Associated with an item
- Can happen more than once

What messages are needed?

- All the same as for addItem


Customer
:CartHandler
:Customer
:ProductItem
:PromoOffering
:InventoryItem
addItemToCart
(promoNo, prodID, size, color, qty)
[firstItem]createCart ( )
aCrt := createCart ( )
:OnlineCart
(aCrt)
addItemToCart
(promoNo, prodID, size, color, qty)
:CartItem
createCartItem
(promoNo, prodID, size, color, qty)
price := getPrice ( )
description := getDesc ( )
status := updateQty (qty)
(description, price, extendedPrice)
(description, price, extendedPrice)
(description, price, extendedPrice)
Loop for all items
addAccessoryToCart
(promoNo, prodID size, color, qty)
addItemToCart
(promoNo, prodID, size, color, qty)
createCartItem
(promoNo, prodID, size, color, qty)
price := getPrice ( )
description := getDesc ( )
status := updateQty (qty)
(description, price, extendedPrice)
(description, price, extendedPrice)
(description, price,
extendedPrice)

Guidelines and Assumptions

Guidelines:

- For each input message, determine:
  - all internal messages
  - the message's objective
  - What information is needed
    - Which classes need it (destination)
    - Which classes have it (source)

Guidelines and Assumptions

Guidelines:

- For each input message, identify:
  - All classes that are needed or affected
    - Look for pre-/post- conditions
    - Classes that are created
    - Classes that are updated
    - Classes holding information

Guidelines and Assumptions

Guidelines:

- For each input message, flesh out:
  - Iteration
  - Conditions
  - Return values
  - Parameters

Guidelines and Assumptions

Assumptions

- Perfect Technology
- Perfect Memory
- Perfect Solution

Summary

We've seen:

- How to create design class diagrams
- Which Analysis Models to use
- What information is extracted from Analysis Models
- Principles of Object Oriented Design

Readings

Text Chapter 14 Deploying the New System

https://en.wikipedia.org/wiki/Unit_testing

https://en.wikipedia.org/wiki/Integration_testing

https://www.teamgantt.com/blog/post-mortem-meeting-template-and-tips

https://blog.lucidmeetings.com/blog/how-to-lead-a-successful-project-retrospective-meeting