Advanced Software
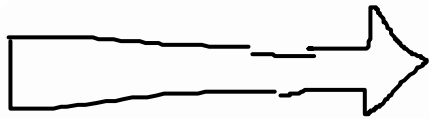# CSE608
# Lect 5
Sequence Diagrams

Dr. Islam El-Maddah

Ain shams university

Faculty of engineering

Dynamic behaviors

- Class diagrams represent *static* relationships. Why?

- What about modeling *dynamic* behavior? → *execution runtime*

- **Interaction** diagrams model how groups of object collaborate to perform some behavior

  - Typically captures the behavior of a single use case

handler

GUI → Control → info
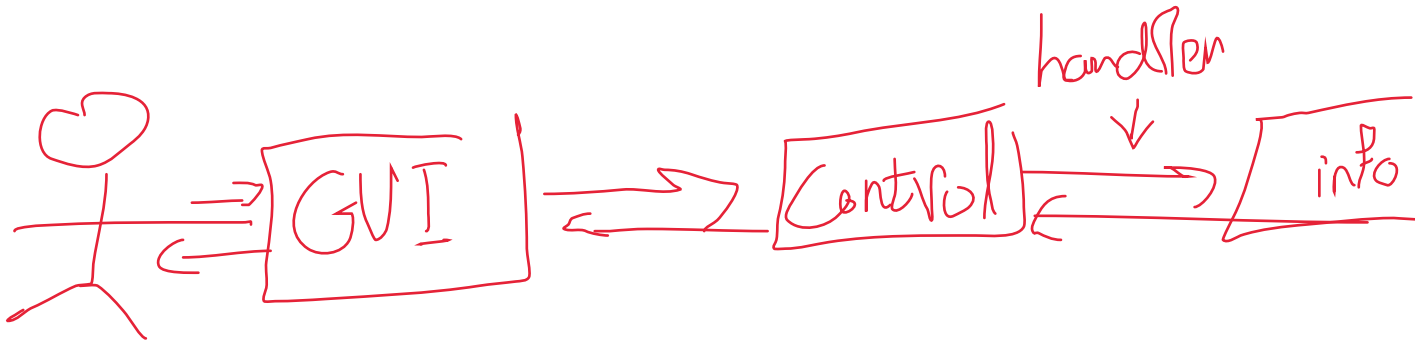
Dynamic behaviors

Use Case: Order Entry
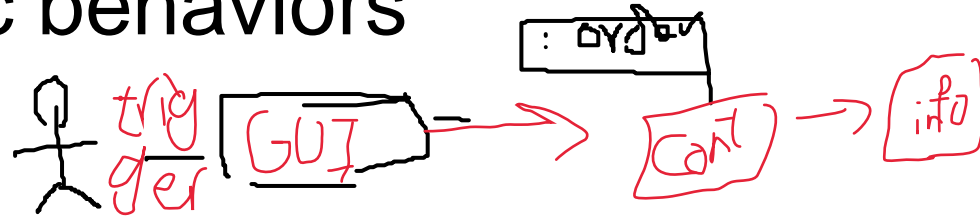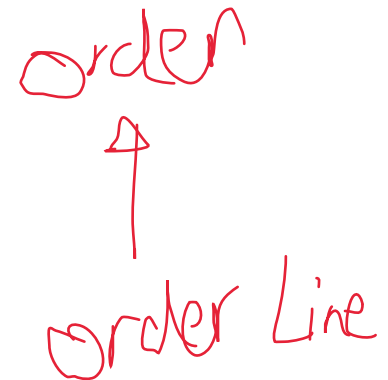
1) An Order Entry window sends a "prepare" message to an Order

2) The Order sends "prepare" to each Order Line on the Order

3) Each Order Line checks the given Stock Item → happy ending

4) Remove appropriate quantity of Stock Item from stock

5) Create a deliver item

Alternative: Insufficient Stock

3a) if Stock Item falls below reorder level

   then Stock Item requests reorder

Sequence diagrams

- Vertical line is called an object's **lifeline**

  - Represents an object's life during interaction

- Object deletion denoted by X, ending a lifeline

  - Horizontal arrow is a message between two objects

- Order of messages sequences top to bottom

- Messages labeled with message name

  - Optionally arguments and control information

- Control information may express conditions:

  - such as [hasStock], or iteration

- Returns (dashed lines) are optional

  - Use them to add clarity

System Sequence Diagram (SSD)

For a use case scenario, an SSD shows:

- The System (as a black box)

- The external actors that interact with System

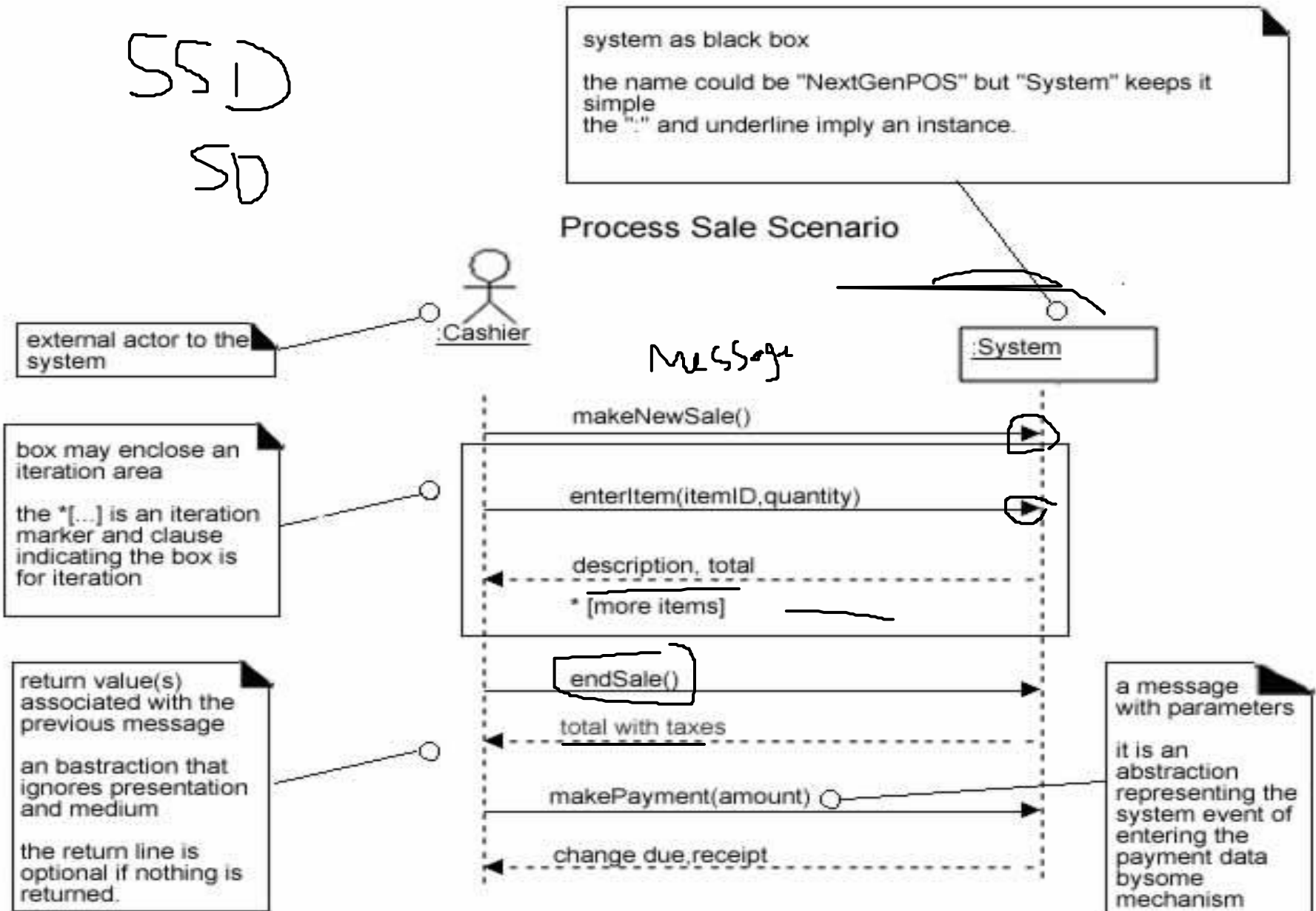- The System events that the actors generate

- SSD shows operations of the System in response to events, in temporal order

- Develop SSDs for the main success scenario of a selected use case, then frequent and salient alternative scenarios

SSD for Process Sale scenario

(Larman, page 175)

SSD
SD

system as black box

the name could be "NextGenPOS" but "System" keeps it simple
the ":" and underline imply an instance.

Process Sale Scenario

:Cashier

Message

:System

external actor to the system

box may enclose an iteration area

the *[...] is an iteration marker and clause indicating the box is for iteration

makeNewSale()

enterItem(itemID,quantity)

description, total

* [more items]

return value(s) associated with the previous message

an bastraction that ignores presentation and medium

the return line is optional if nothing is returned.

endSale()

total with taxes

makePayment(amount)

change due,receipt

a message with parameters

it is an abstraction representing the system event of entering the payment data bysome mechanism
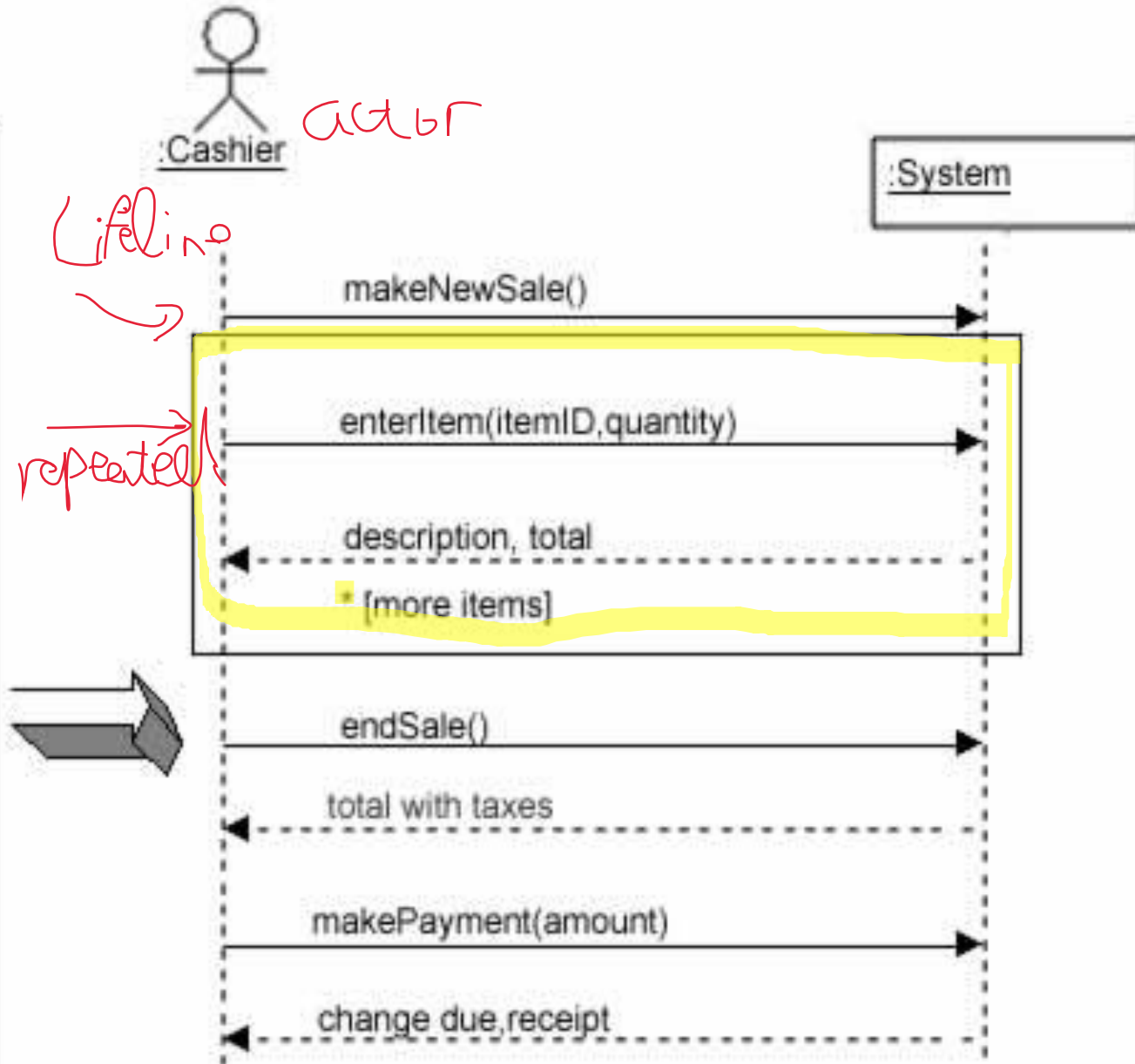
From Use Case to Sequence System Diagram

How to construct an SSD from a use case:

1. Draw System as black box on right side

2. For each actor that directly operates on the System, draw a stick figure and a lifeline.

3. For each System events that each actor generates in use case, draw a message.

4. Optionally, include use case text to left of diagram.

Example: use cases to SSD

Simple Cash-only Process Sale scenario:

1. customer arrives at aPOS check out with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters a new item identifier.
4. System records new sale line item and presents item description, price and running total.

Cashier repeats steps steps 3-4 until indicates done.

5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.

....

:Cashier — *actor*

Lifeline

repeated

:System

makeNewSale()

enterItem(itemID,quantity)

description, total

*[more items]

endSale()

total with taxes

makePayment(amount)

change due, receipt

Identifying the right Actor

- In the process Sale example, does the customer interact directly with the POS system?

- Who does?

- Cashier interacts with the system directly

- Cashier is the generator of the system events

- Why is this an important observation?

Naming System events & operations

- System events and associated system operations should be expressed at the level of intent

- Rather than physical input medium or UI widget

- Start operation names with verb (from use case)

- Which is better, scanBarCode or enterItem?

SSDs and the Glossary in parallel

- Why is updating the glossary important when developing the SSD?

- New terms used in SSDs may need explanation, especially if they are not derived from use cases

- A glossary is less formal, easier to maintain and more intuitive to discuss with external parties such as customers

SSDs within the Unified Process

Create System Sequence Diagrams during Elaboration in order to:

- Identify System events and major operations

- Write System operation contracts (Contracts describe detailed system behavior)

- Support better estimates

- Remember, there is a season for everything: it is not necessary to create SSDs for all scenarios of all use cases, at least not at the same time

Concurrency in Sequence Diagrams

- Concurrent processes:

  - UML 1: **asynchronous** messages as horizontal lines with **half** arrow heads

  - UML 2 makes this distinction by not filling an arrowhead

- After setting up Transaction Coordinator, invoke concurrent Transaction Checkers

  - If a check fails, kill all Transaction Checker processes

- Note use of comments in margin

  - *When is this a good idea?*

Collaboration diagrams

- Objects are rectangular icons

  - e.g., Order Entry Window, Order, etc.

- Messages are arrows between icons

  - e.g., prepare()

- Numbers on messages indicate sequence

  - Also spatial layout helps show flow

- *Which do you prefer: sequence or collaboration diagrams?*

- Fowler now admits he doesn't use collaboration diagrams

  - Interaction diagrams show flow clearly,
    but are awkward when modeling alternatives

- UML notation for control logic has changed in UML 2
  but Fowler isn't impressed

Learning Objectives

- Explain the different types of objects and layers in a design

- Develop sequence diagrams for use case realization

- Develop communication diagrams for detailed design

- Develop updated design class diagrams

- Develop multilayer subsystem packages

- Explain design patterns and recognize various specific patterns

Overview

- Primary focus of this chapter is how to develop detailed sequence diagrams to design use cases

  - The first-cut sequence diagram focuses only on the problem domain classes

  - The complete multi-layer design includes the data access layer and the view layer

- Design Patterns are an important concept that is becoming more important for system development

Design Patterns and the Use Case Controller

- Design pattern

  - A standard solution template to a design requirement that facilitates the use of good design principles

- Use case controller pattern

  - Design requirement is to identify which problem domain class should receive input messages from the user interface for a use case

Design Patterns and the Use Case Controller (continued)

- Solution is to choose a class to serve as a collection point for all incoming messages for the use case. Controller acts as intermediary between outside world and internal system

- Artifact – a class invented by a system designer to handle a needed system function, such as a controller class

Use Case Controller Pattern

| Name:	Controller
Problem:	Domain classes have the responsibility of processing use cases. However, since there can be many domain classes, which one(s) should be responsible for receiving the input messages? User-interface classes become very complex if they have visibility to all of the domain classes. How can the coupling between the user-interface classes and the domain classes be reduced?
Solution:	Assign the responsibility for receiving input messages to a class that receives all input messages and acts as a switchboard to forward them to the correct domain class. There are several ways to implement this solution: (a) Have a single class that represents the entire system, or (b) Have a class for each use case or related group of use cases to act as a use case handler.
Example:	The RMO order-entry subsystem accepts inputs from an OrderWindow. These input messages are passed to an OrderHandler, which acts as the switchboard to forward the message to the correct problem domain class. Collaberation Diagram Other examples of the controller can be found for each RMO subsystem.
Benefits and Consequences:	Coupling between the view layer and the domain layer is reduced. The controller provides a layer of indirection. The controller is closely coupled to many domain classes. If care is not taken, controller classes can become incoherent, with too many unrelated functions. If care is not taken, business logic will be inserted into the controller class.

Use Case Realization
with Sequence Diagrams

- Realization of use case done through interaction diagram development

- Determine what objects collaborate by sending messages to each other to carry out use case

- Sequence diagrams and communication diagrams represent results of design decisions

  - Use well-established design principles such as coupling, cohesion, separation of responsibilities

Understanding Sequence Diagrams SSDs
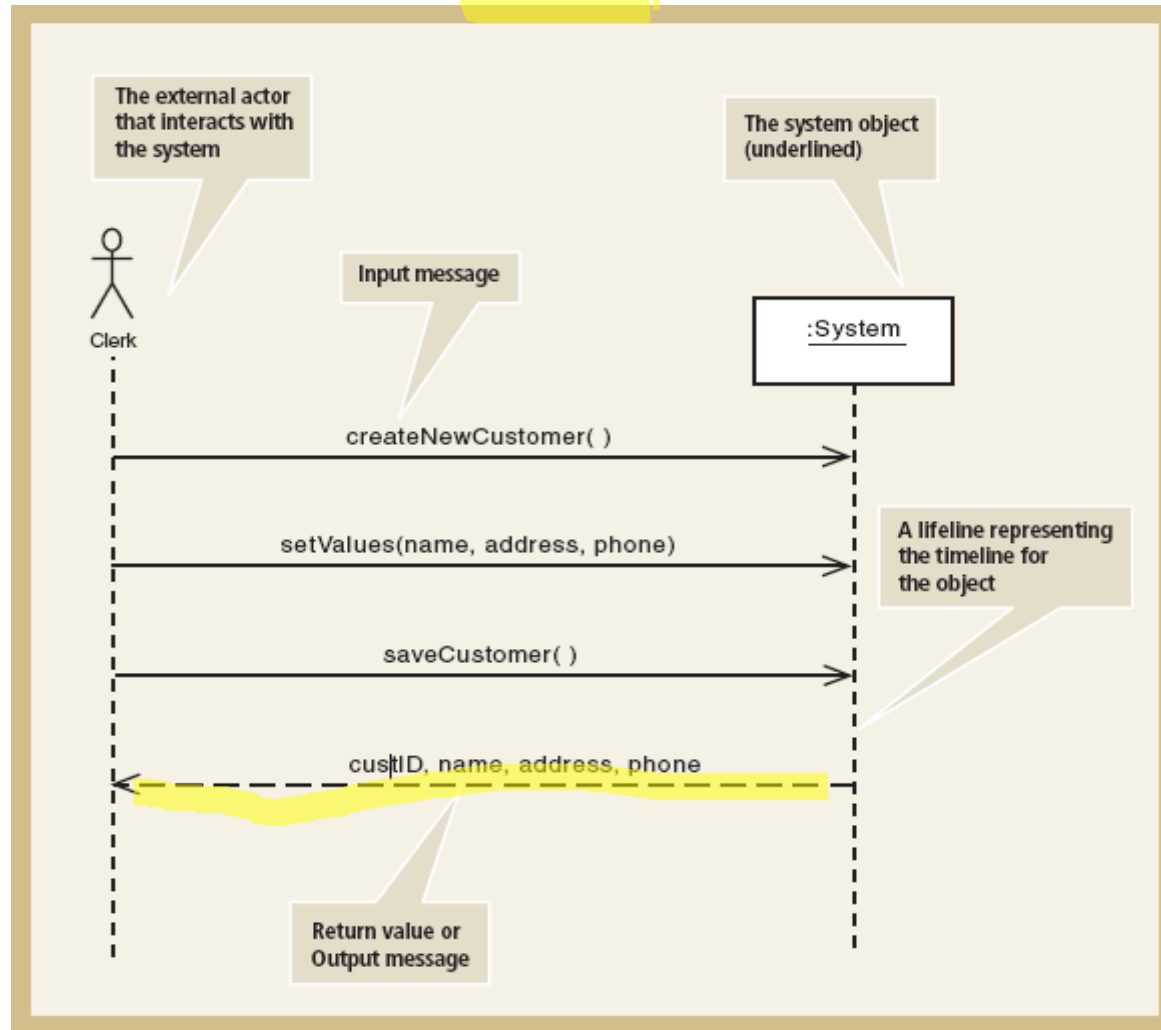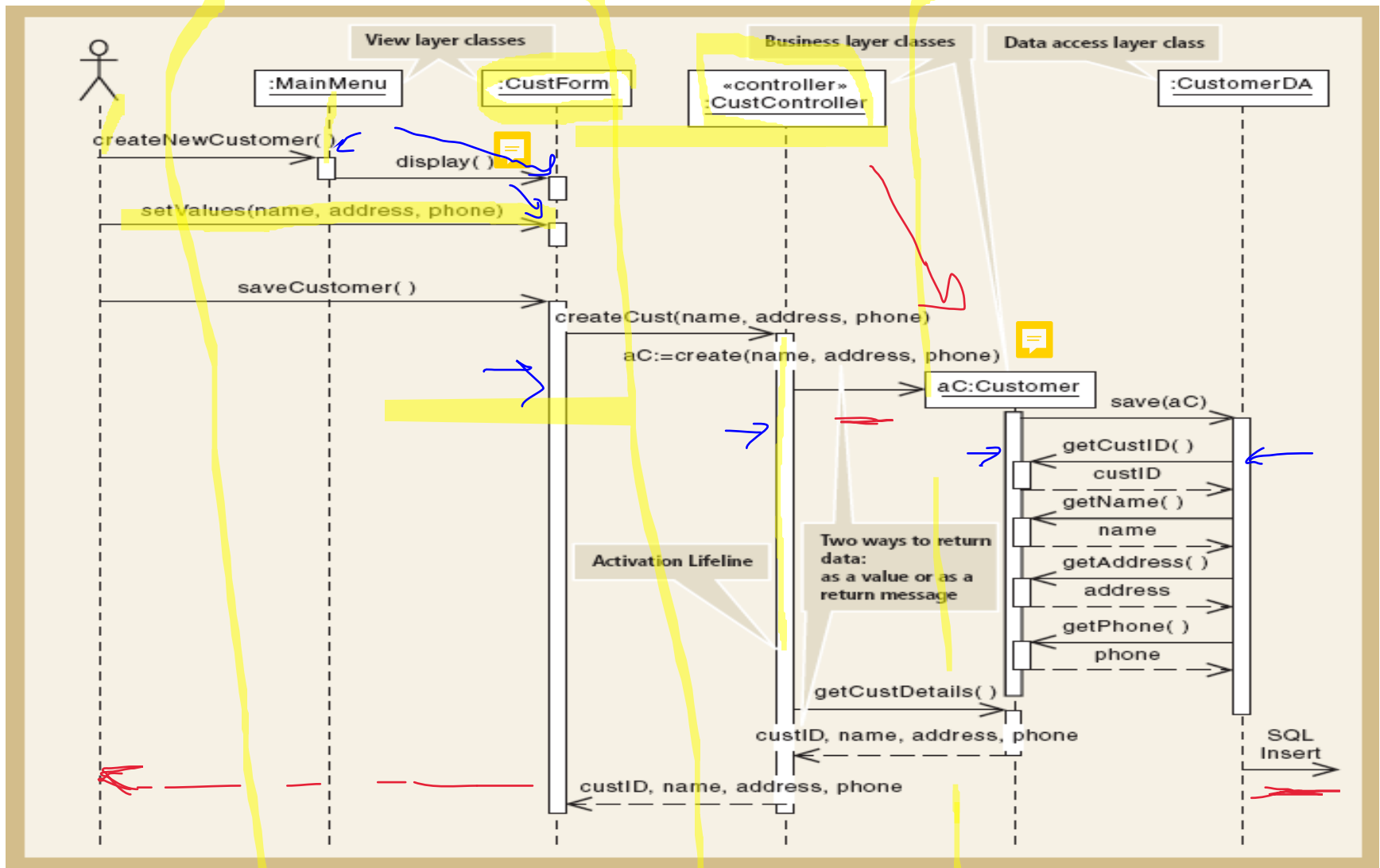
Detailed Sequence Diagram

Figure 12-3

Designing with Sequence Diagrams

- Sequence diagrams used to explain object interactions and document design decisions

- Document inputs to and outputs from system for single use case or scenario

- Capture interactions between system and external world as represented by actors

- Inputs are messages from actor to system

- Outputs are return messages showing data

Object Responsibility

- Objects are responsible for system processing

- Responsibilities include knowing and doing

  - Knowing about object's own data and other classes of objects with which it collaborates to carry out use cases

  - Doing activities to assist in execution of use case

    - Receive and process messages

    - Instantiate, or create, new objects required to complete use case

- Design means assigning responsibility to the appropriate classes based on design principles and using design patterns
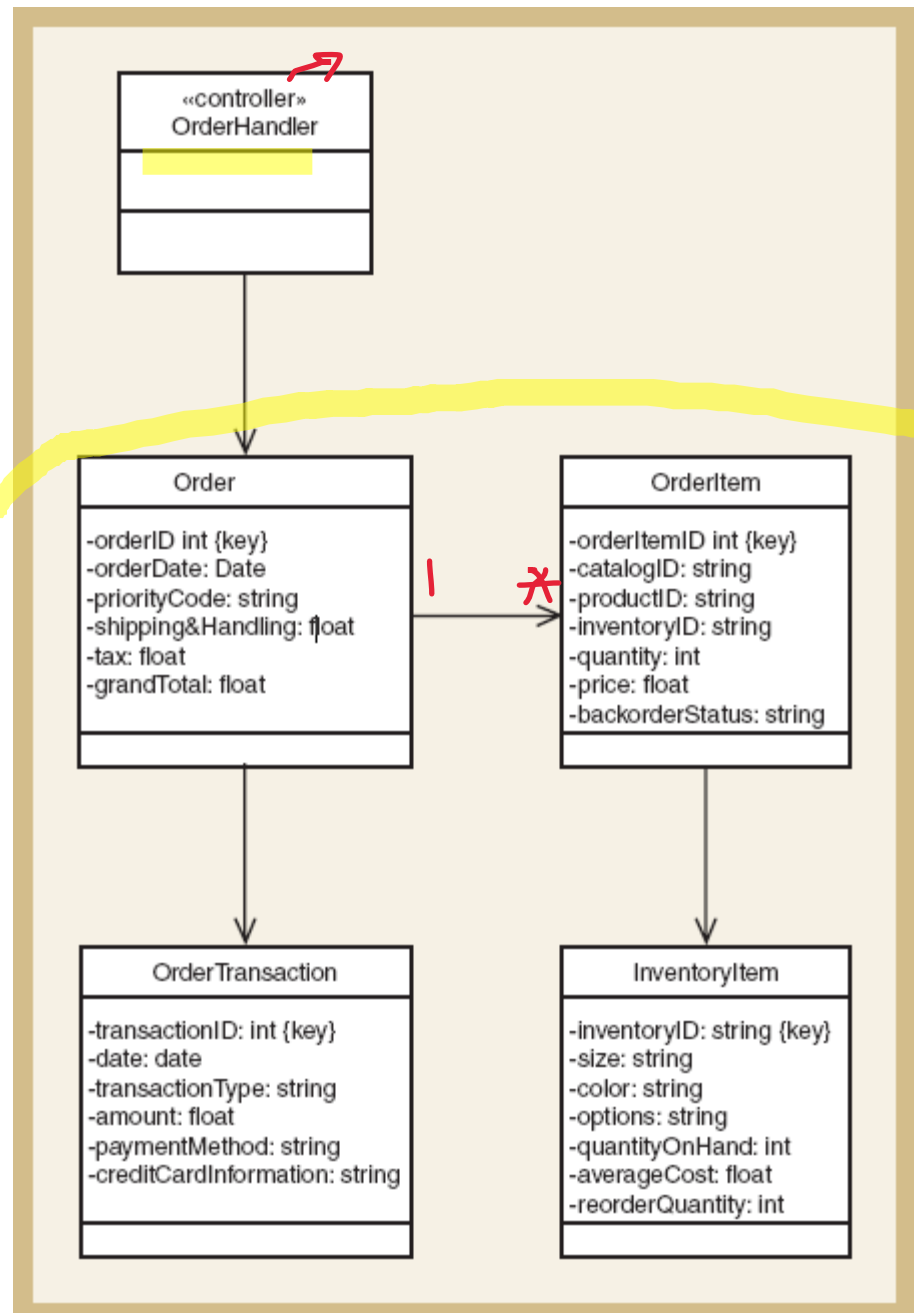
First-Cut Sequence Diagram

☐ Start with elements from SSD

☐ Replace <u>:System</u> object with use case controller

☐ Add other objects to be included in use case

  ☐ Select input message from the use case

  ☐ Add all objects that must collaborate

☐ Determine other messages to be sent

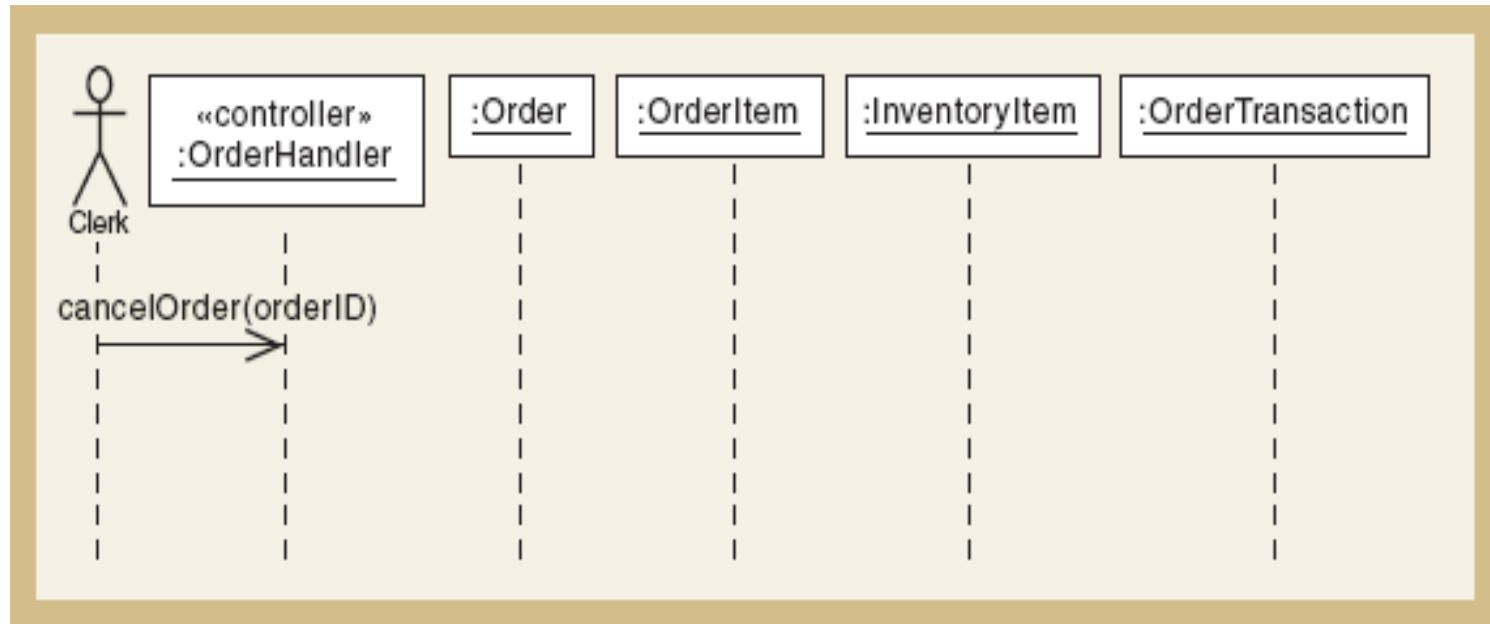 ☐ Which object is source and destination of each message?

SSD for *Cancel an Order*
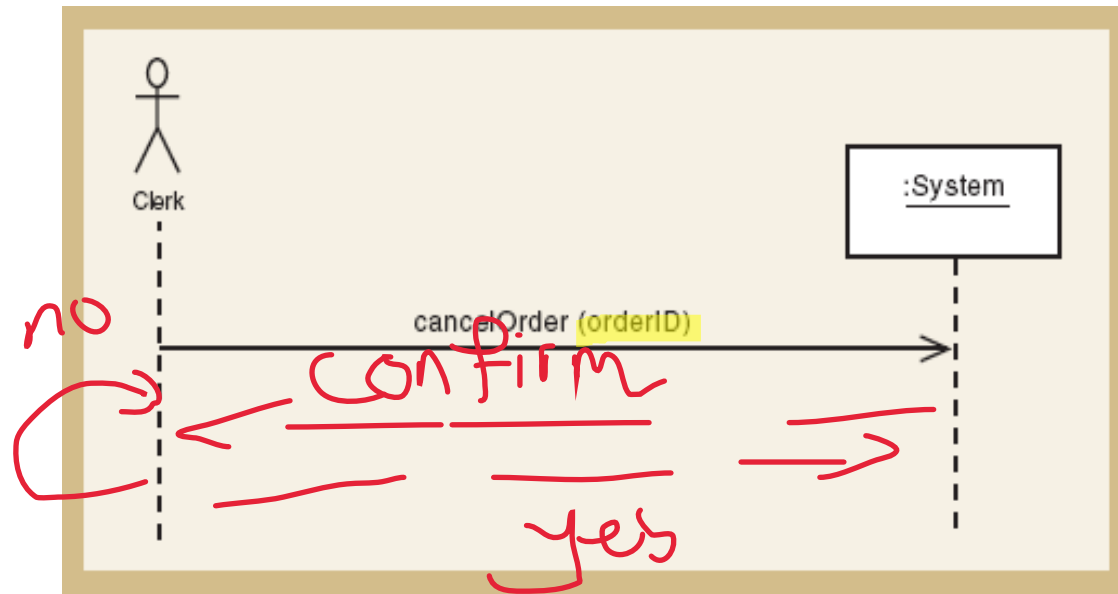
First Cut Design Class Diagram for *Cancel an Order*

Potential Objects for *Cancel an Order*

First Cut Sequence Diagram
for *Cancel an Order*
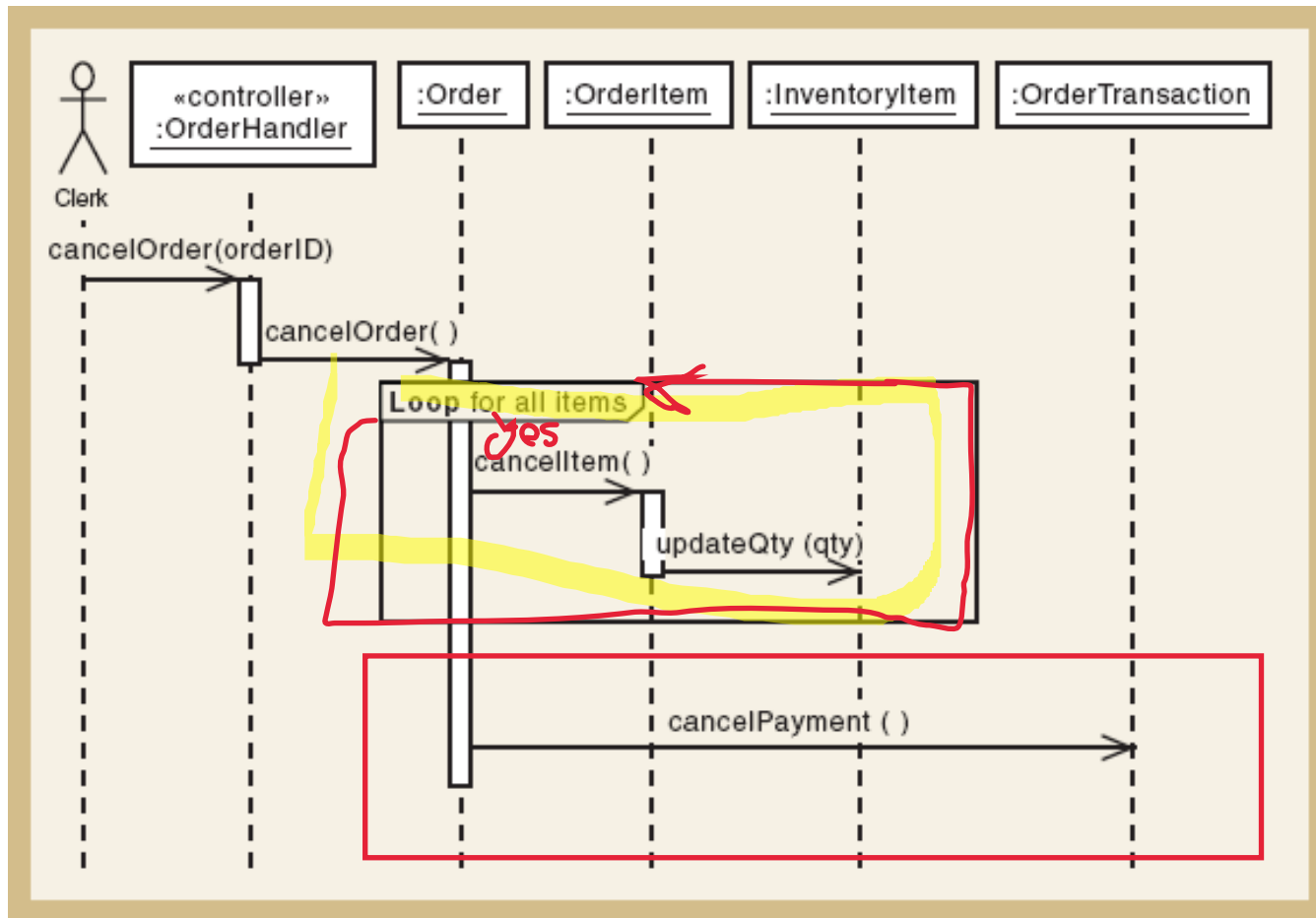
Guidelines for Sequence Diagram Development for Use Case

- Take each input message and determine internal messages that result from that input
  - For that message, determine its objective
  - Needed information, class destination, class source, and objects created as a result
  - Double check for all required classes
- Flesh out components for each message
  - Iteration, guard-condition, passed parameters, return values

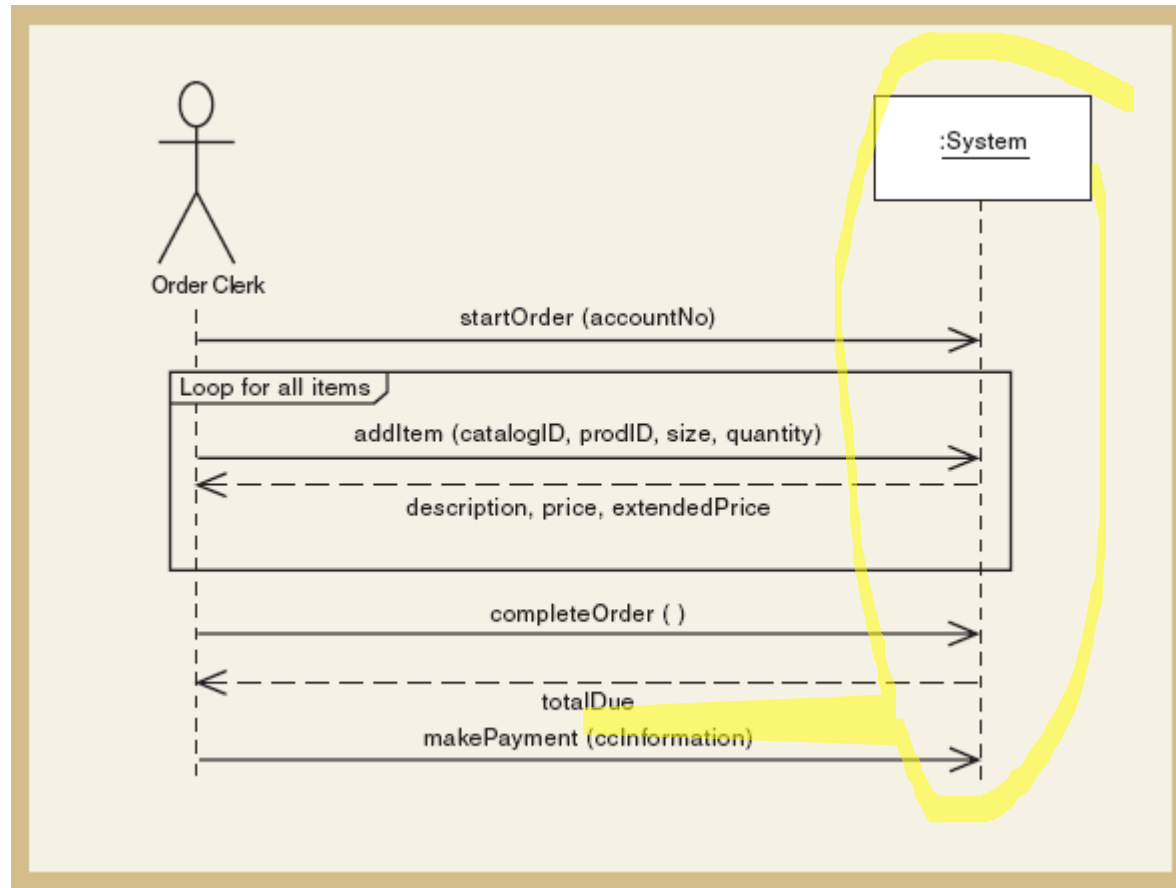# Assumptions About First-Cut Sequence Diagram

- Perfect technology assumption
  - Don't include system controls like login/logout (yet)
- Perfect memory assumption
  - Don't worry about object persistence (yet)
  - Assume objects are in memory ready to work
- Perfect solution assumption
  - Don't worry about exception conditions (yet)
  - Assume happy path/no problems solution

SSD for *Create new phone order*
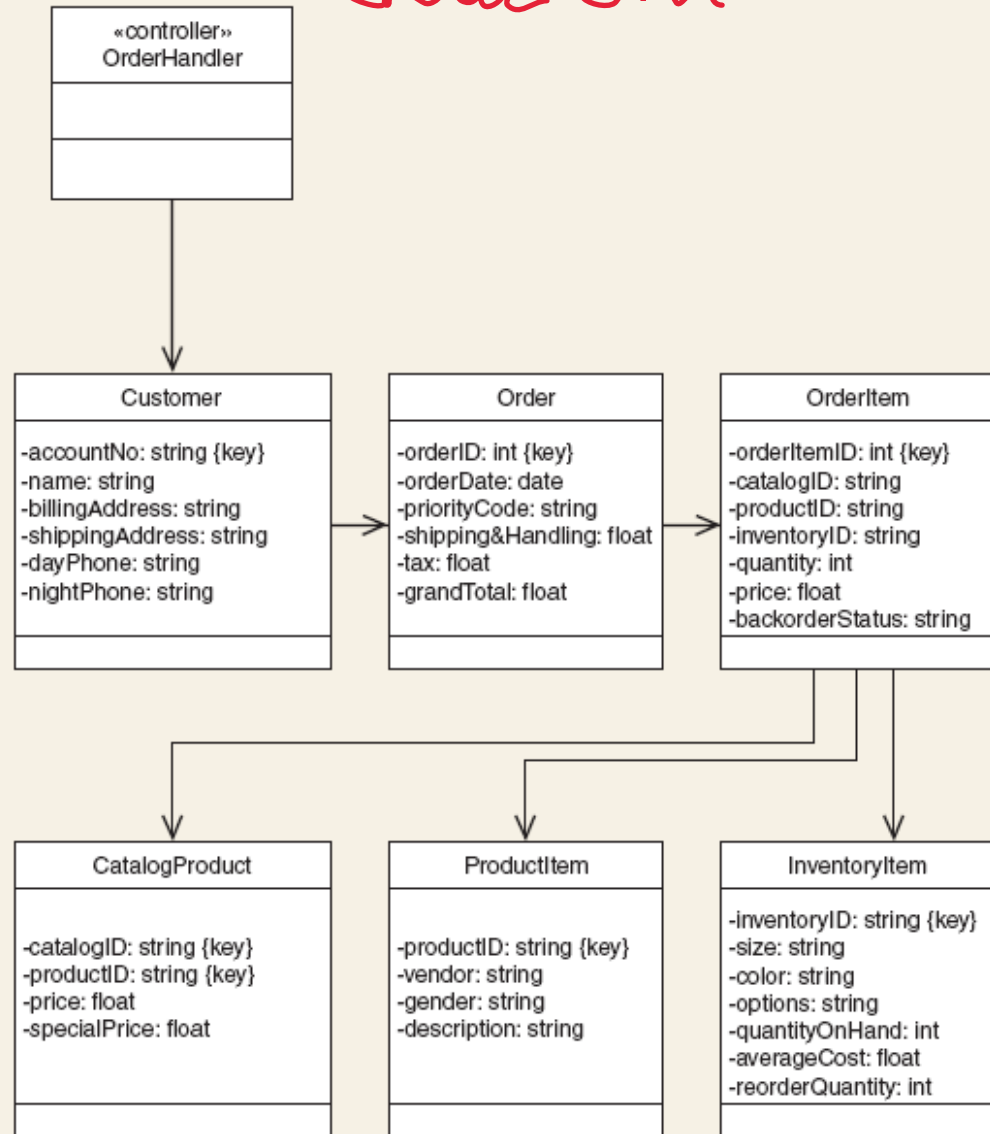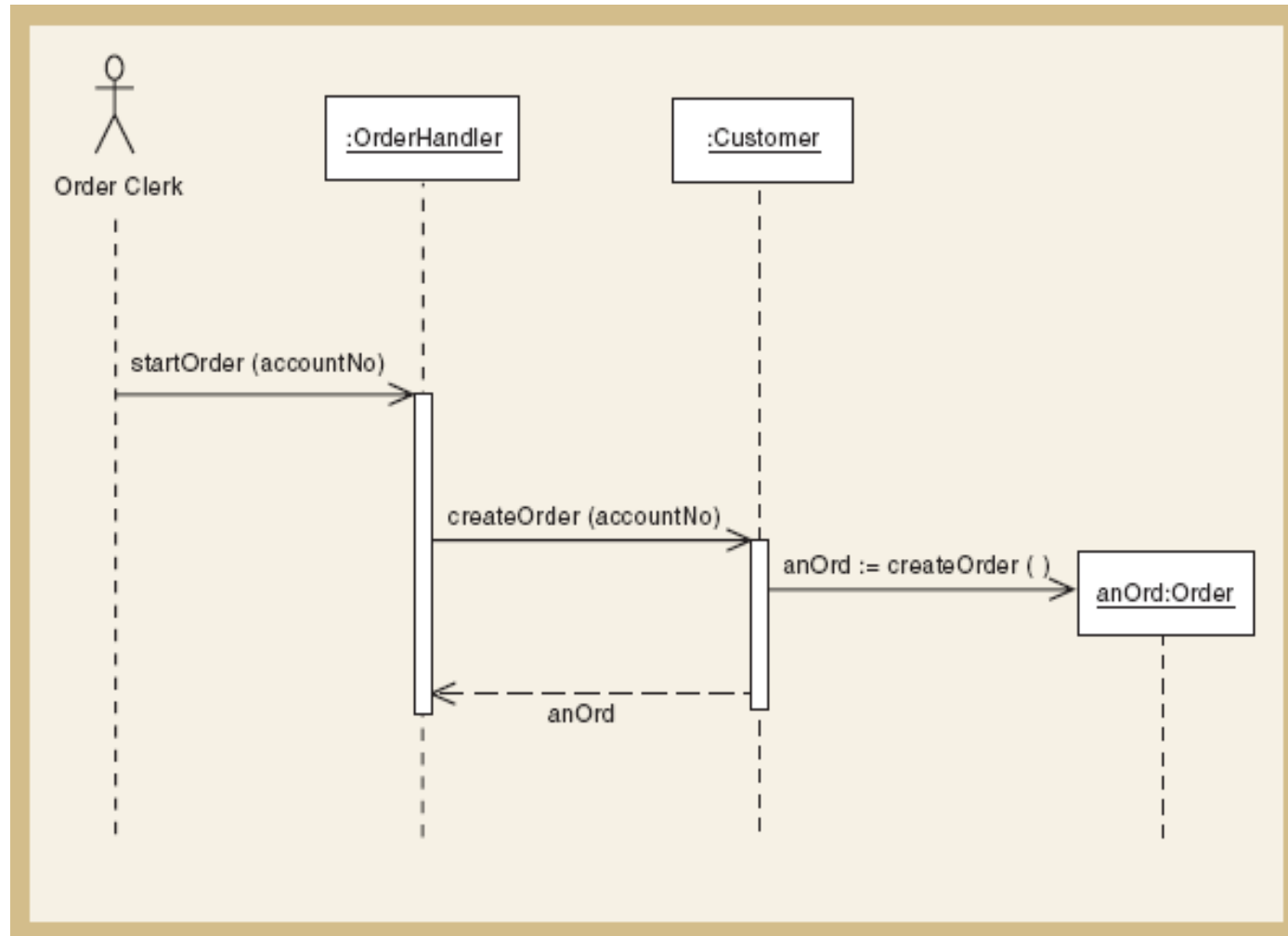
sys →

seg ↗

1 unit no time

Order Clerk

:System

startOrder (accountNo)

Loop for all items

addItem (catalogID, prodID, size, quantity)

description, price, extendedPrice

completeOrder ( )

totalDue

makePayment (ccInformation)

First cut DCD for *Create new phone order*

create order

«controller»
OrderHandler

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

Order
- -orderID: int {key}
- -orderDate: date
- -priorityCode: string
- -shipping&Handling: float
- -tax: float
- -grandTotal: float

OrderItem
- -orderItemID: int {key}
- -catalogID: string
- -productID: string
- -inventoryID: string
- -quantity: int
- -price: float
- -backorderStatus: string

CatalogProduct
- -catalogID: string {key}
- -productID: string {key}
- -price: float
- -specialPrice: float

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

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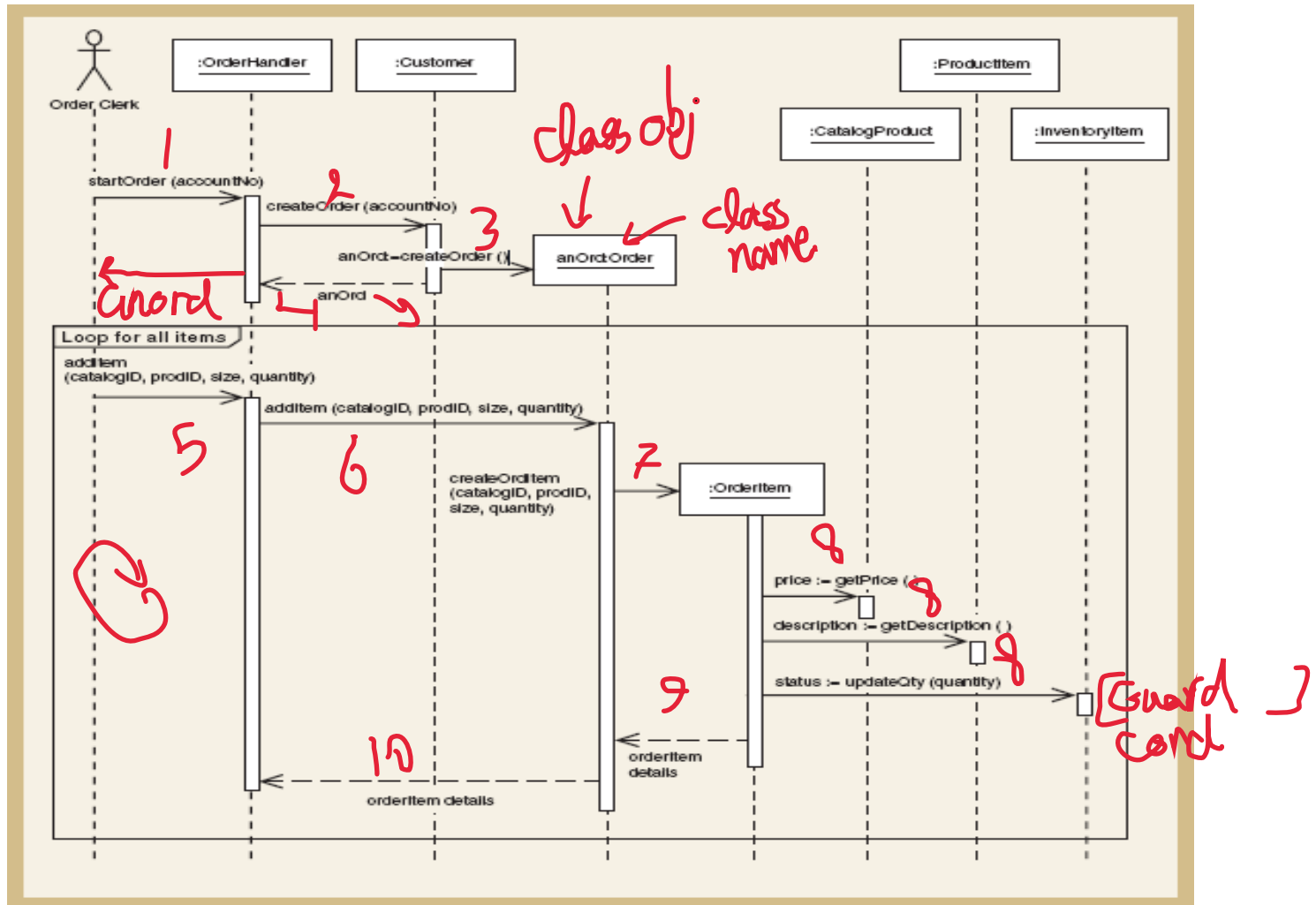- -quantityOnHand: int
- -averageCost: float
- -reorderQuantity: int

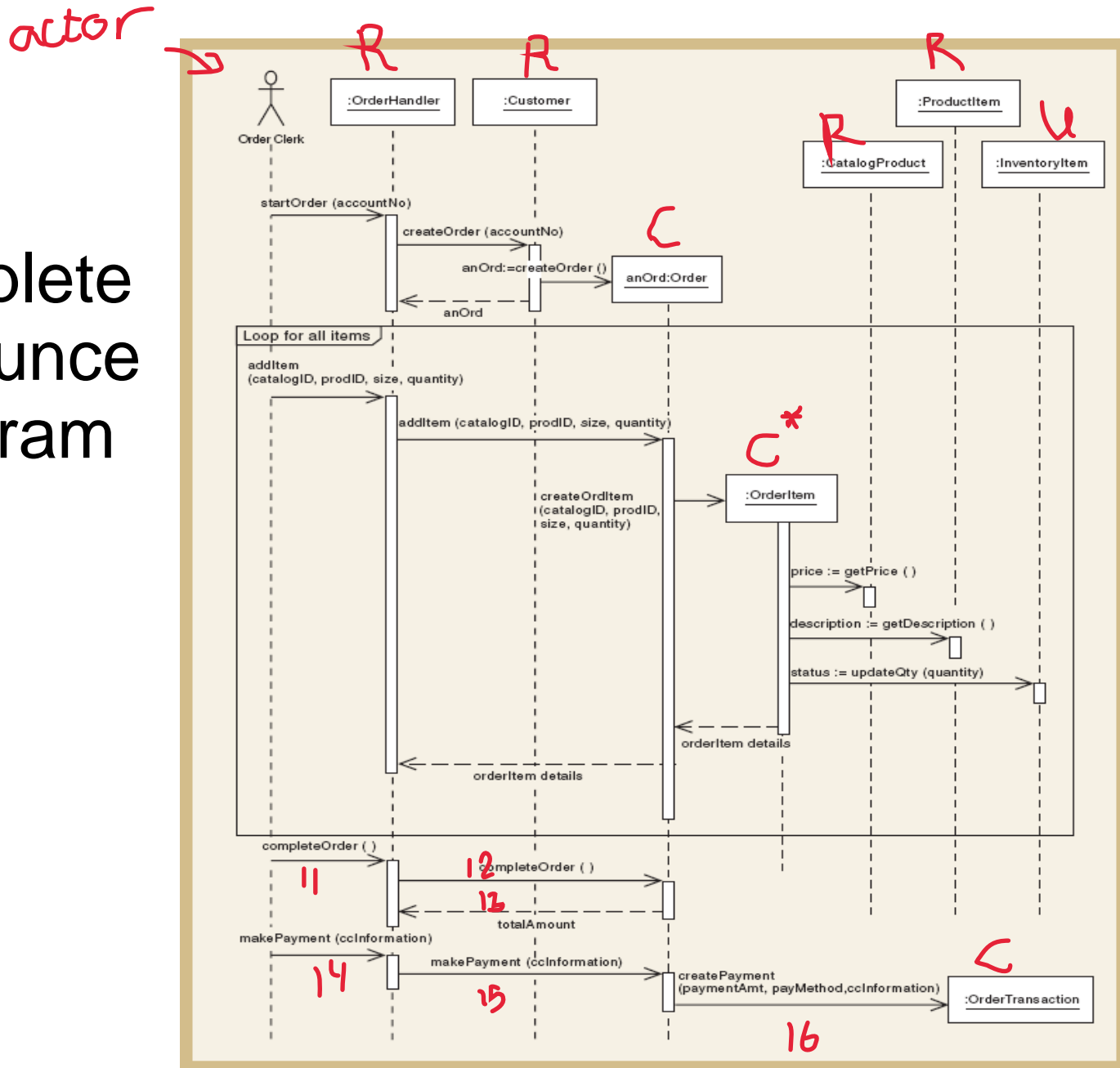# Sequence Diagram for First Input Message

Sequence Diagram for First and Second Input Messages

Complete Seqeunce Diagram

Complete Sequence Diagram

Annotations (handwritten in red):
- actor (pointing to Order Clerk)
- R (above :OrderHandler)
- R (above :Customer)
- R (above :ProductItem)
- U (above :InventoryItem)
- R (above :CatalogProduct)
- C (near anOrd:Order)
- C* (near :OrderItem)
- C (near :OrderTransaction)
- 11, 12, 13, 14, 15, 16

Diagram text:

- Order Clerk (actor)
- :OrderHandler
- :Customer
- :ProductItem
- :CatalogProduct
- :InventoryItem

- startOrder (accountNo)
- createOrder (accountNo)
- anOrd:=createOrder ()
- anOrd:Order
- anOrd

- Loop for all items
- addItem (catalogID, prodID, size, quantity)
- addItem (catalogID, prodID, size, quantity)
- createOrdItem (catalogID, prodID, size, quantity)
- :OrderItem
- price := getPrice ( )
- description := getDescription ( )
- status := updateQty (quantity)
- orderItem details
- orderItem details

- completeOrder ( )
- completeOrder ( )
- totalAmount
- makePayment (ccInformation)
- makePayment (ccInformation)
- createPayment (paymentAmt, payMethod, ccInformation)
- :OrderTransaction
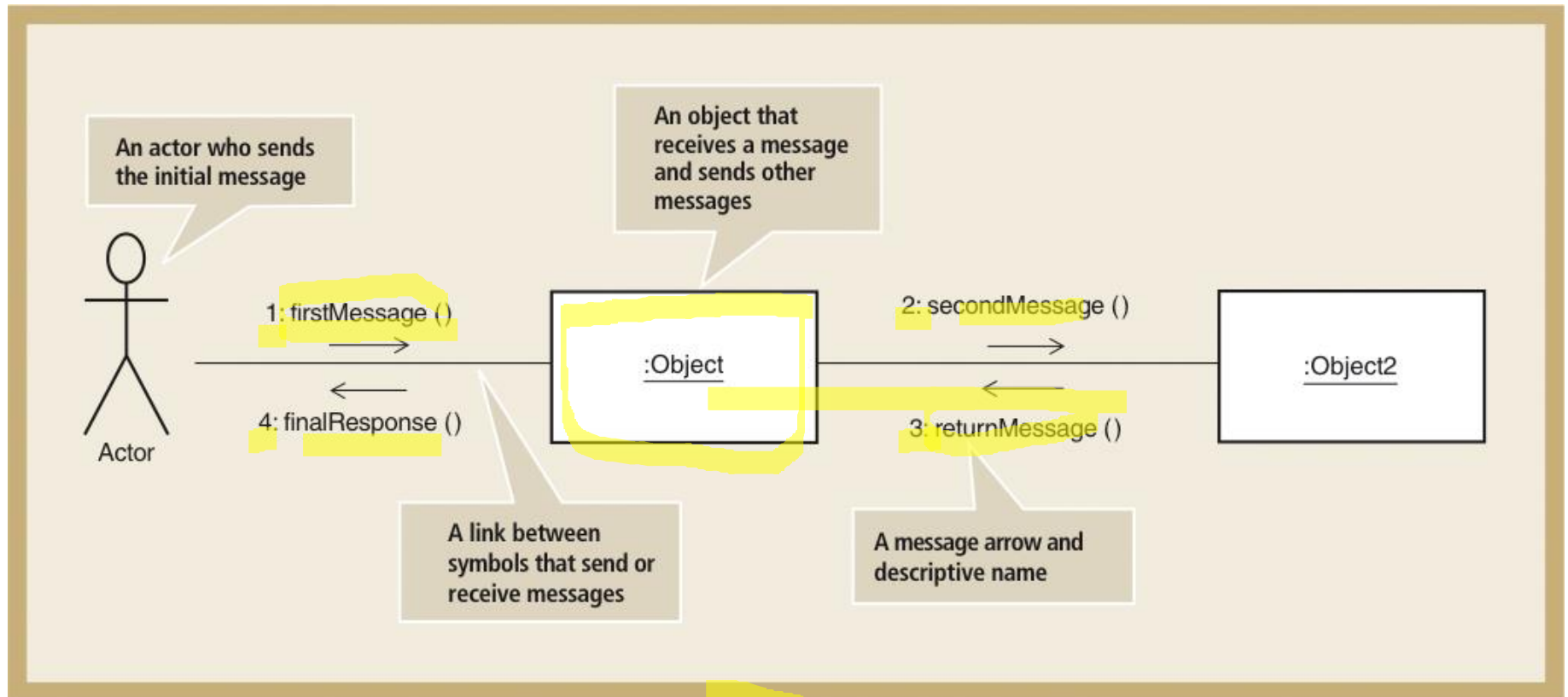
Designing with Communication Diagrams

- Communication diagrams and sequence diagrams

  - Both are interaction diagrams

  - Both capture same information

  - Process of designing is same for both

- Model used is designer's personal preference

  - Sequence diagram – use case descriptions and dialogs follow sequence of steps

  - Communication diagram – emphasizes coupling

The Symbols of a Communication Diagram

A Communication Diagram for *Create new phone order*
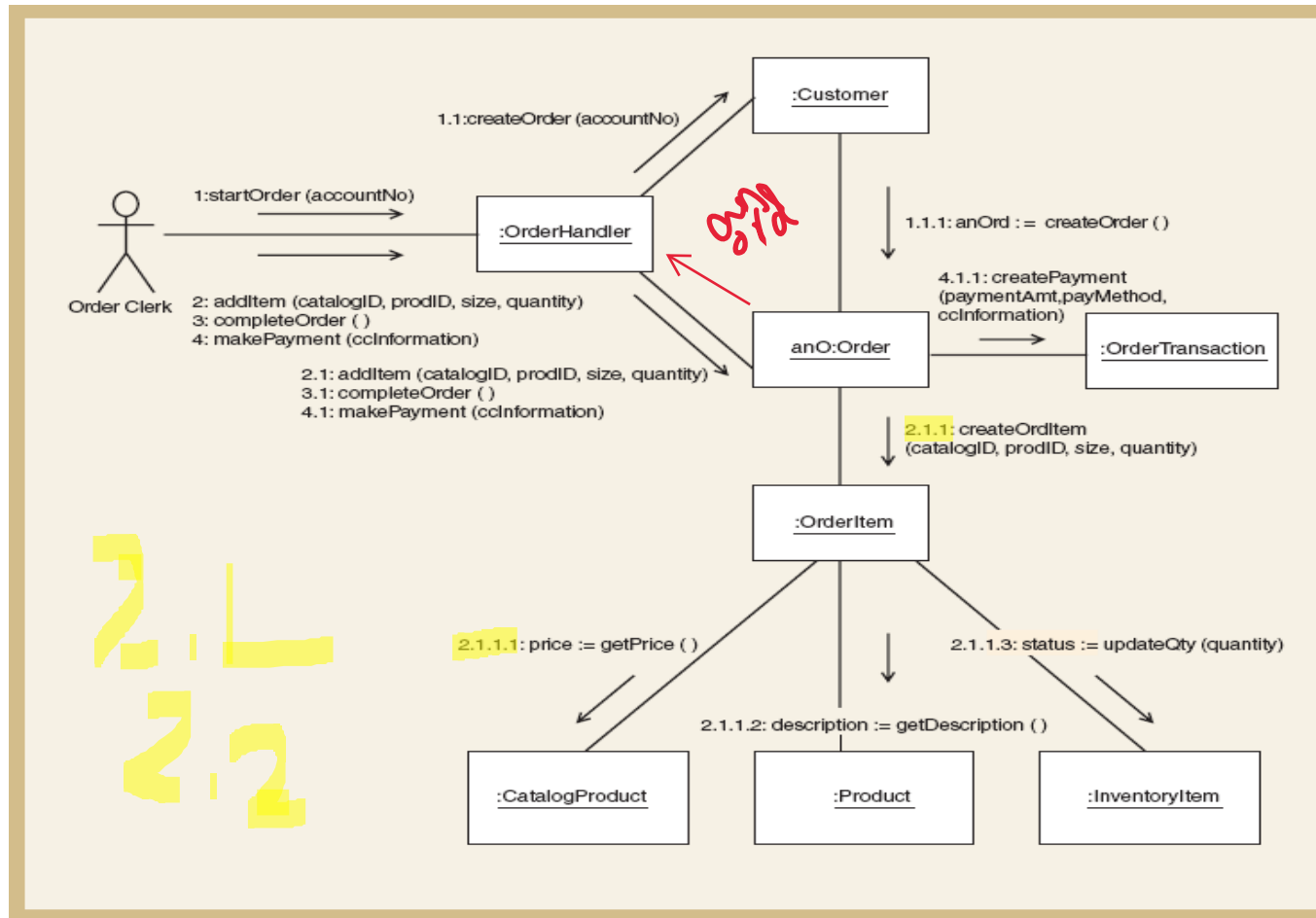
Updating the Design Class Diagram

- Design class diagrams developed for each layer

  - New classes for view layer and data access layer

  - New classes for domain layer use case controllers

- Sequence diagram's messages used to add methods

  - Constructor methods

  - Data get and set method

  - Use case specific methods

Design Class with Method Signatures, for the InventoryItem Class
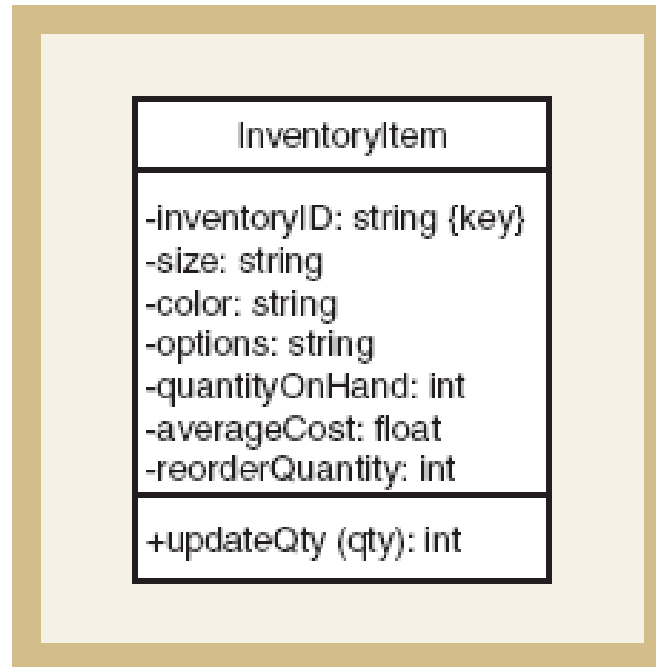
Updated Design Class Diagram for the Domain Layer