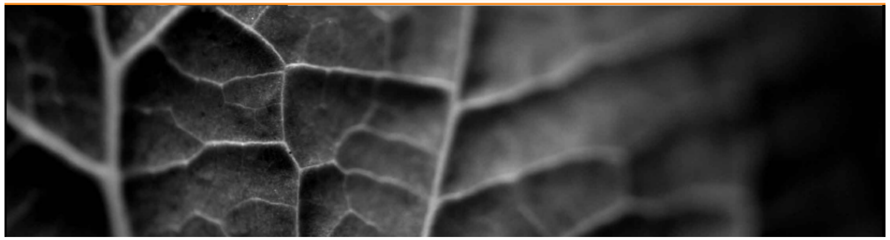
Systems Analysis and Design in a Changing World, Fifth Edition

CHAPTER 12

OBJECT-ORIENTED DESIGN: USE CASE REALIZATIONS



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

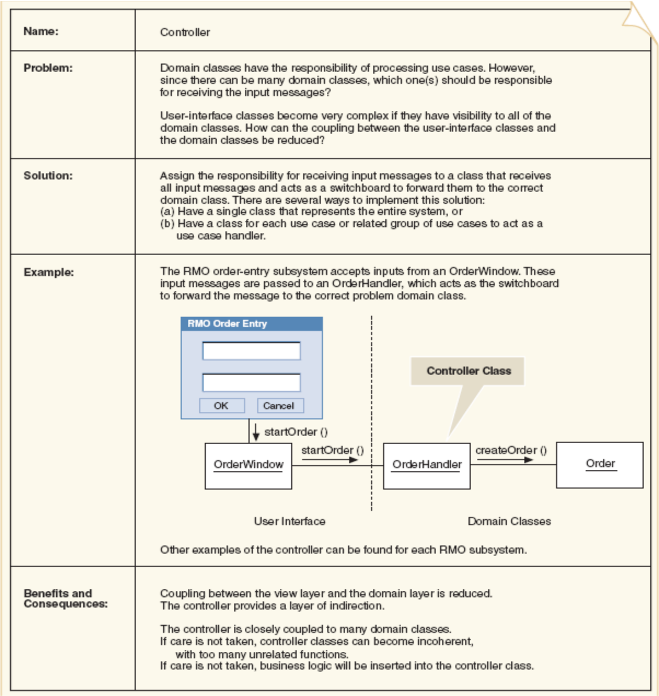
Overview

- 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
  - 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

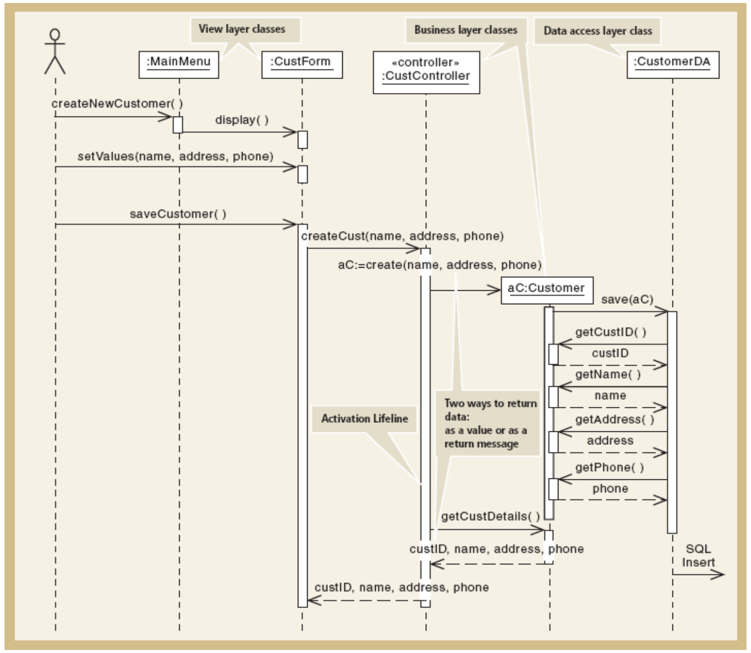


Name:	Controller
Problem:	Domain classes have the responsibility of processing use cases. However, since there can be many domain classes, which one(s) should be responsible for receiving the input messages? User-interface classes become very complex if they have visibility to all of the domain classes. 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. 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

Detailed Sequence Diagram



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

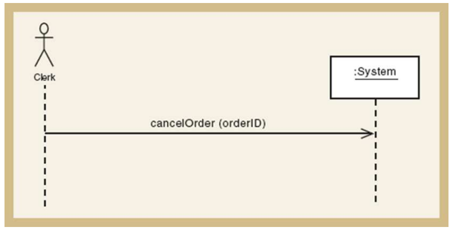
Object Responsibility

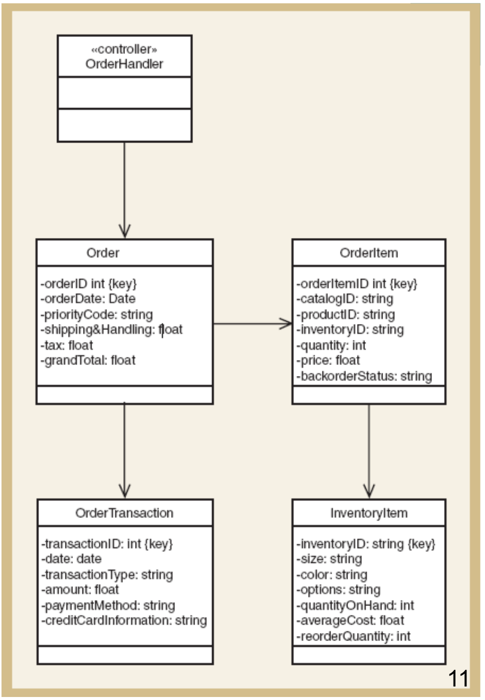
- Objects are responsible for system processing
- Responsibilities include knowing and doing
  - Knowing about object’s own data and other classes of objects with which it collaborates to carry out use cases
  - Doing activities to assist in execution of use case
    - Receive and process messages
    - 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 Sequence Diagram (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?

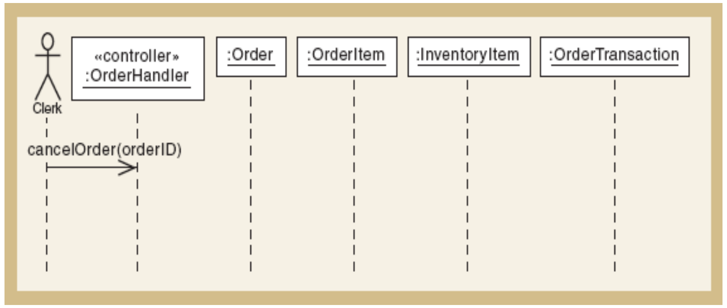
SSD for *Cancel an Order*



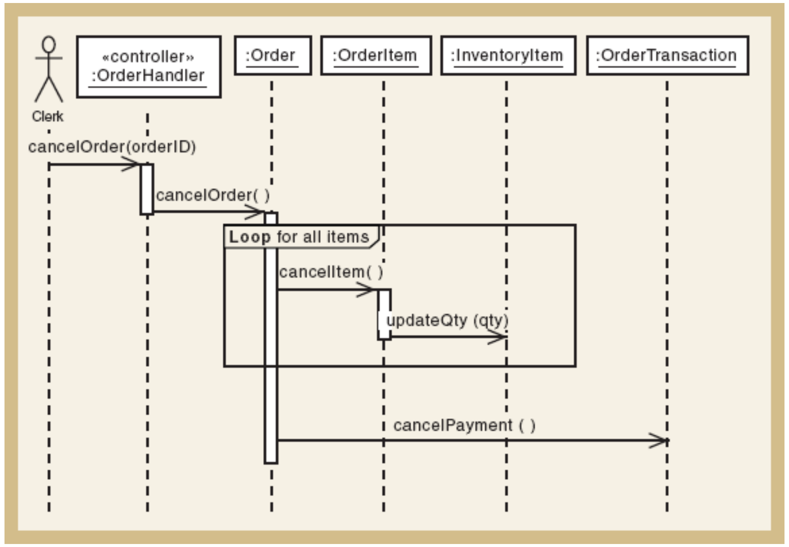


First Cut Design Class Diagram for *Cancel an Order*

Potential Objects for *Cancel an Order*



First Cut Sequence Diagram for *Cancel an Order*



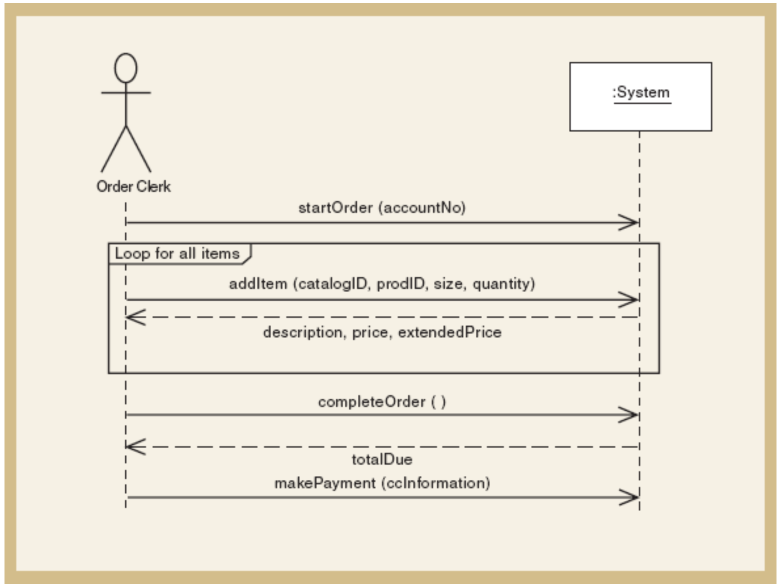
Guidelines for Sequence Diagram Development for Use Case

- 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

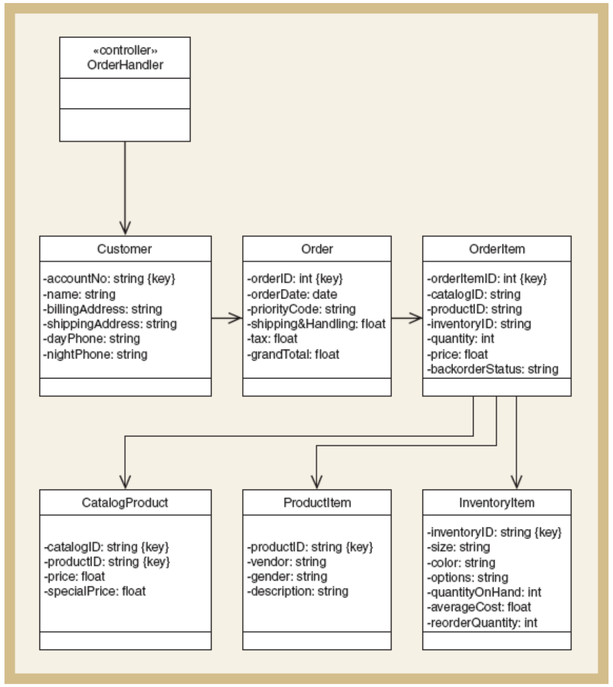
- Perfect technology assumption
  - Don’t include system controls like login/logout (yet)
- Perfect memory assumption
  - Don’t worry about object persistence (yet)
  - Assume objects are in memory ready to work
- Perfect solution assumption
  - Don’t worry about exception conditions (yet)
  - Assume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

SSD for *Create new phone order*


Order Clerk
:System
startOrder (accountNo)
Loop for all items
addItem (catalogID, prodID, size, quantity)
description, price, extendedPrice
completeOrder ( )
totalDue
makePayment (ccInformation)

12

First cut DCD for *Create new phone 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

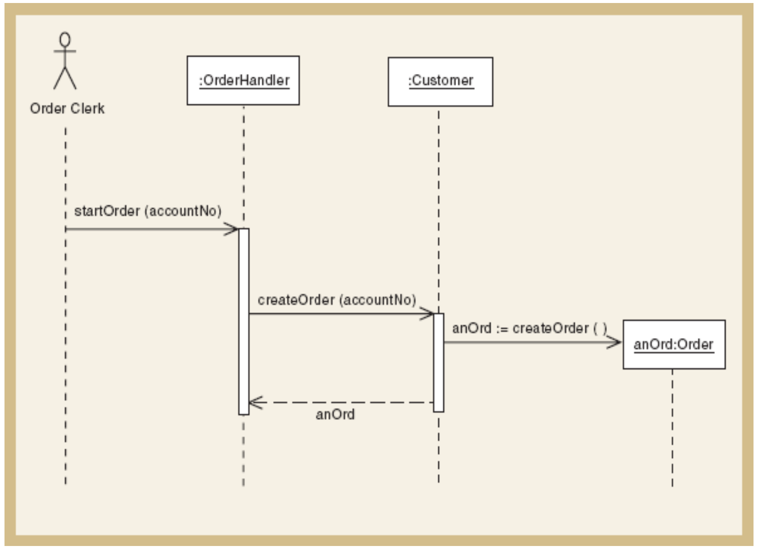
-productID: string {key}
-vendor: string
-gender: string
-description: string

InventoryItem

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

12

Sequence Diagram for First Input Message



Order Clerk

:OrderHandler

:Customer

startOrder (accountNo)

createOrder (accountNo)

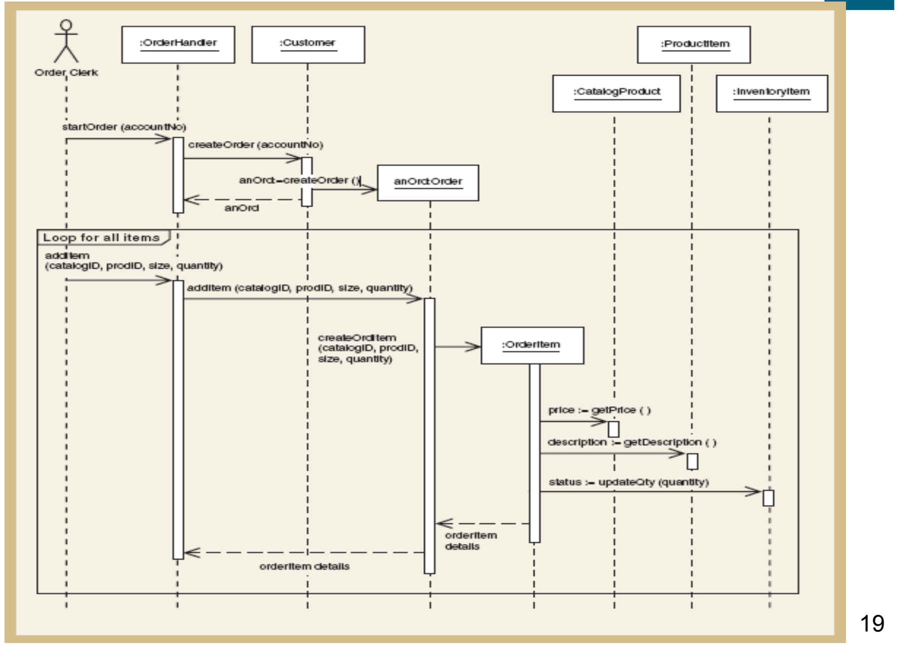
anOrd := createOrder ( )

anOrd:Order

anOrd

Sequence Diagram for First and Second Input Messages

12



Order Clerk

: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 ( )

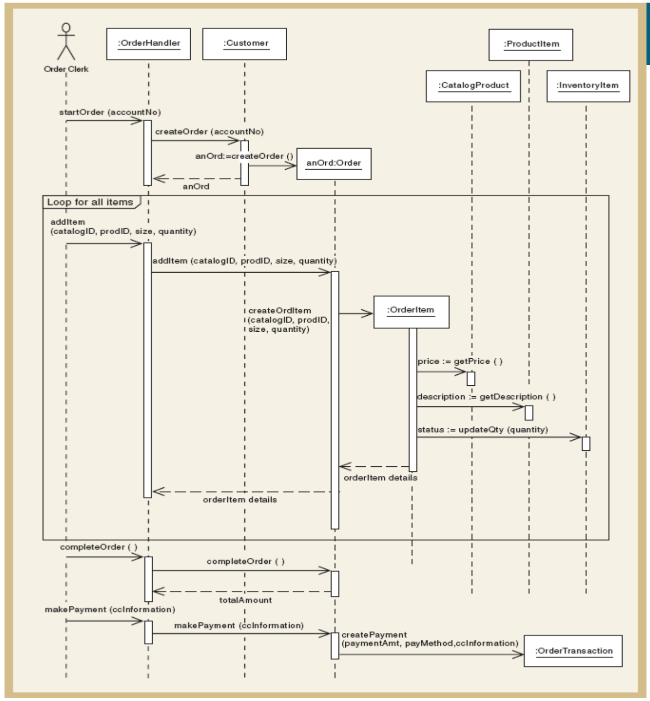
description := getDescription ( )

status := updateQty (quantity)

orderItem
details

orderItem details

Complete Seqeunce Diagram



12

Order Clerk

: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)

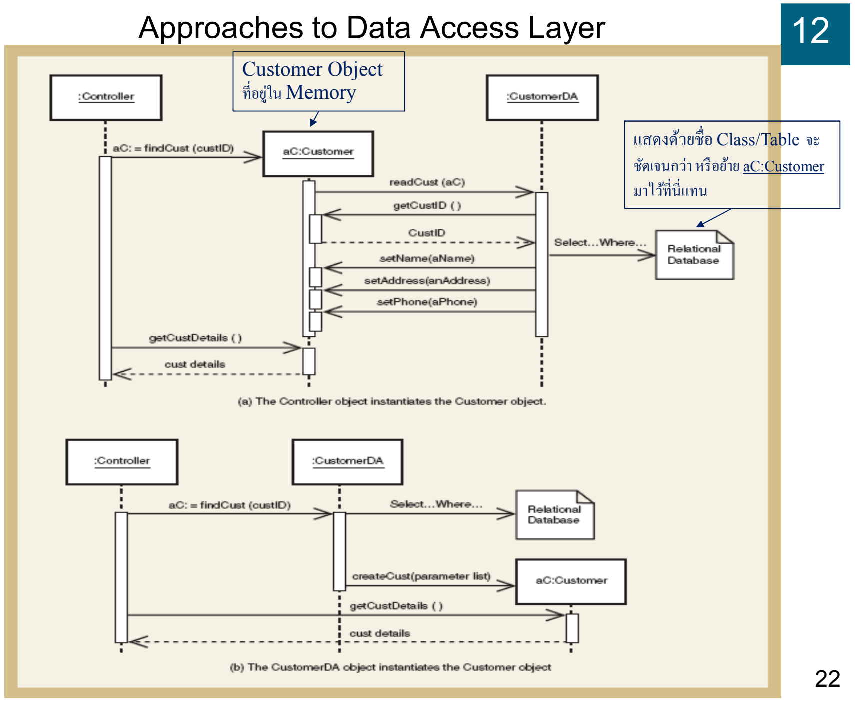
createPayment
(paymentAmt, payMethod,ccInformation)

:OrderTransaction

Developing a Multilayer Design

- First-cut sequence diagram – use case controller plus classes in domain layer
- Add data access layer – design for data access classes for separate database interaction (≡ Data Management Controller: DMC)
  - No more perfect memory assumption
  - Separation of responsibilities
- Add view layer – design for user-interface classes
  - Forms added as windows classes to sequence diagram between actor and controller

Approaches to Data Access Layer



Customer Object ที่อยู่ใน Memory

แสดงด้วยชื่อ Class/Table จะชัดเจนกว่า หรือย้าย aC:Customer มาไว้ที่นี่แทน

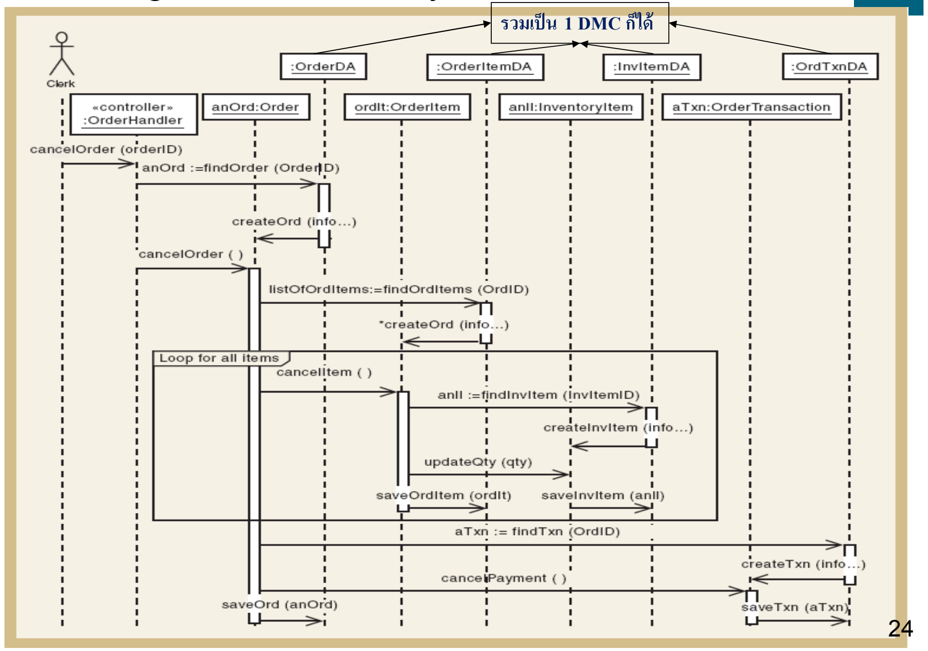
(a) The Controller object instantiates the Customer object.

(b) The CustomerDA object instantiates the Customer object

Approaches to Data Access Layer (continued)

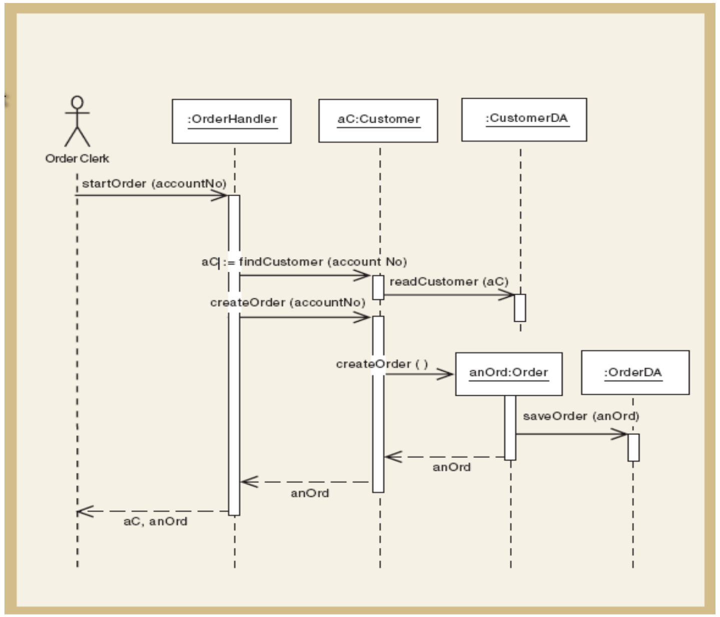
- Create data access class for each domain class
  - CustomerDA added for Customer
  - Database connection statements and SQL statements separated into data access class. Domain classes do not have to know about the database design or implementation
- Approach (a) – controller instantiates new customer aC; new instance asks DA class to populate its attributes reading from the database
- Approach (b) – controller asks DA class to instantiate new customer aC; DA class reads database and passes values to customer constructor
  - Two following examples use this approach

Adding Data Access Layer for *Cancel an order*

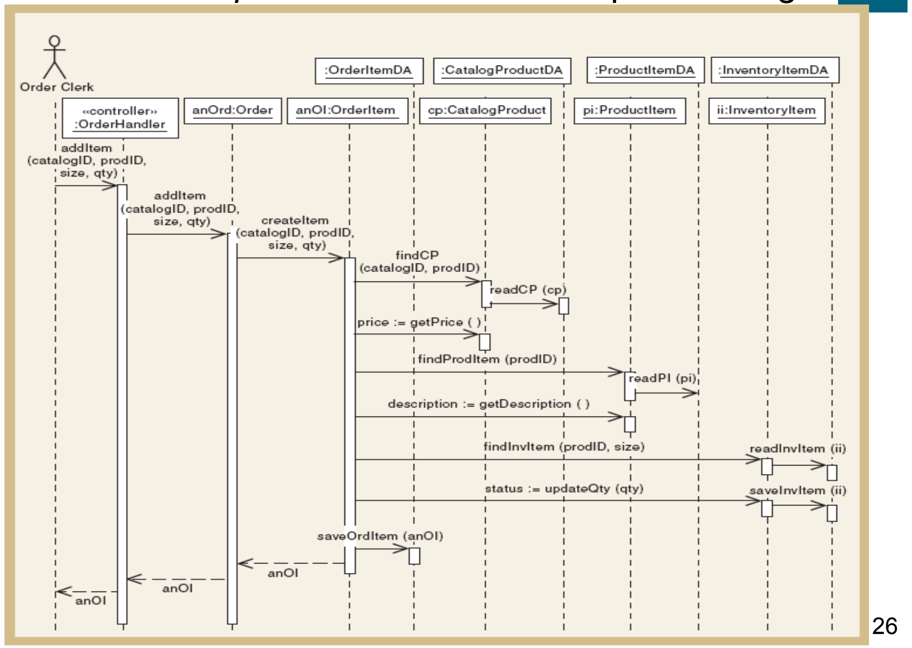


รวมเป็น 1 DMC ก็ได้

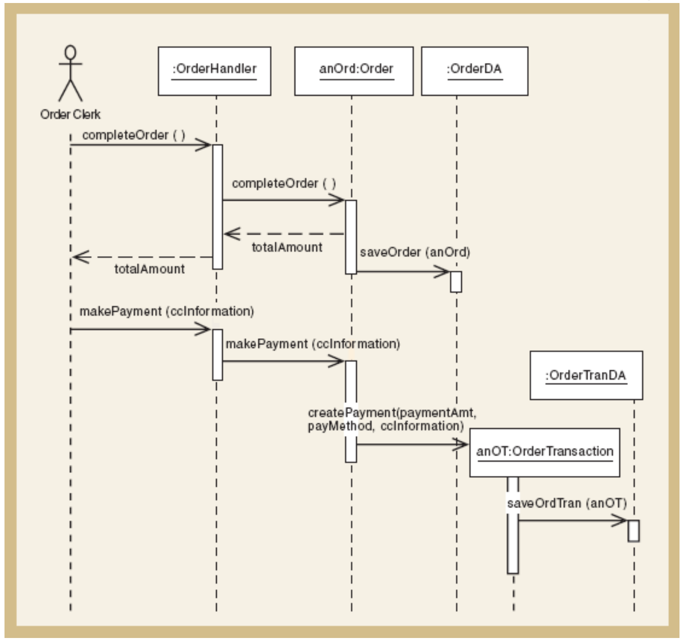
Create new phone order problem domain and data access



Create new phone order: Second input message



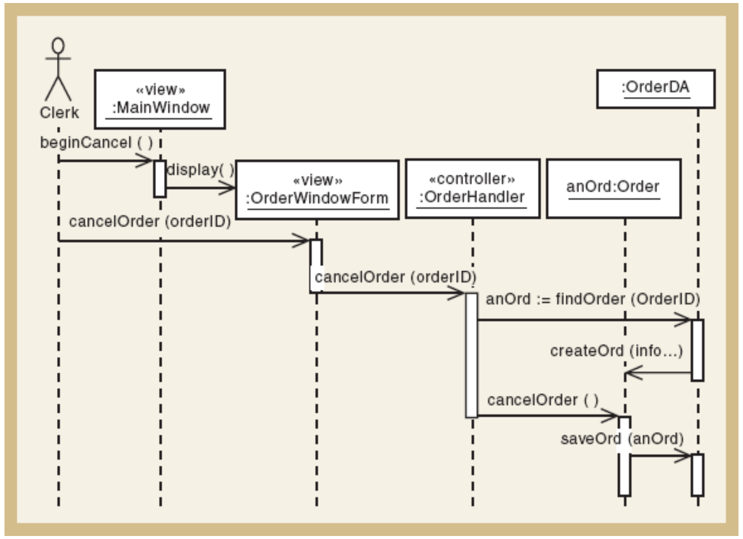
Create new phone order: Final Input Messages



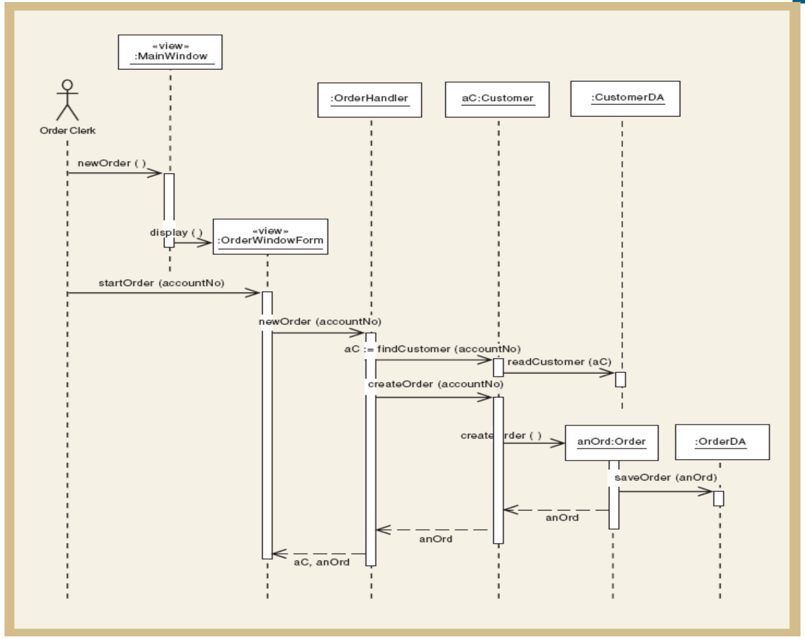
Designing the View Layer

- Add GUI forms or Web pages between actor and controller for each use case
  - Minimize business logic attached to a form
- Some use cases require only one form; some require multiple forms and dialog boxes
- View layer design is focused on high-level sequence of forms/pages – the dialog

Cancel an order with view layer



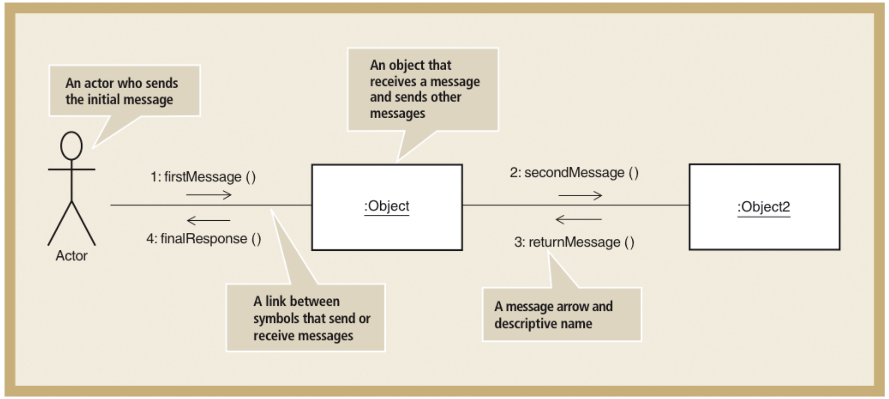
Create new phone order with view layer



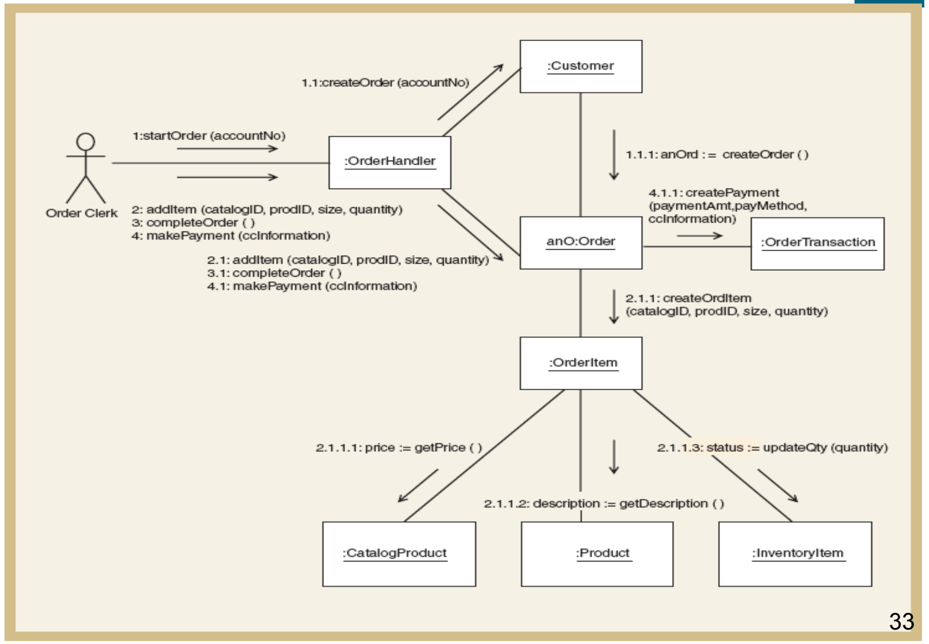
Designing with Communication Diagrams

- 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


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

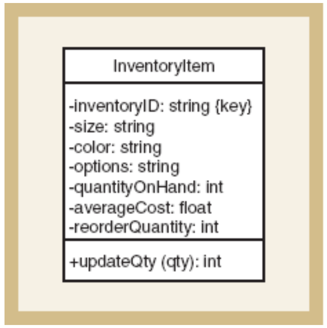
A Communication Diagram for *Create new phone order*


:Customer
1.1:createOrder (accountNo)
1:startOrder (accountNo)
:OrderHandler
Order Clerk
1.1.1: anOrd := createOrder ( )
4.1.1: createPayment
(paymentAmt,payMethod,
ccInformation)
2: addItem (catalogID, prodID, size, quantity)
3: completeOrder ( )
4: makePayment (ccInformation)
anO:Order
:OrderTransaction
2.1: addItem (catalogID, prodID, size, quantity)
3.1: completeOrder ( )
4.1: makePayment (ccInformation)
2.1.1: createOrdItem
(catalogID, prodID, size, quantity)
:OrderItem
2.1.1.1: price := getPrice ( )
2.1.1.3: status := updateQty (quantity)
2.1.1.2: description := getDescription ( )
:CatalogProduct
:Product
:InventoryItem

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

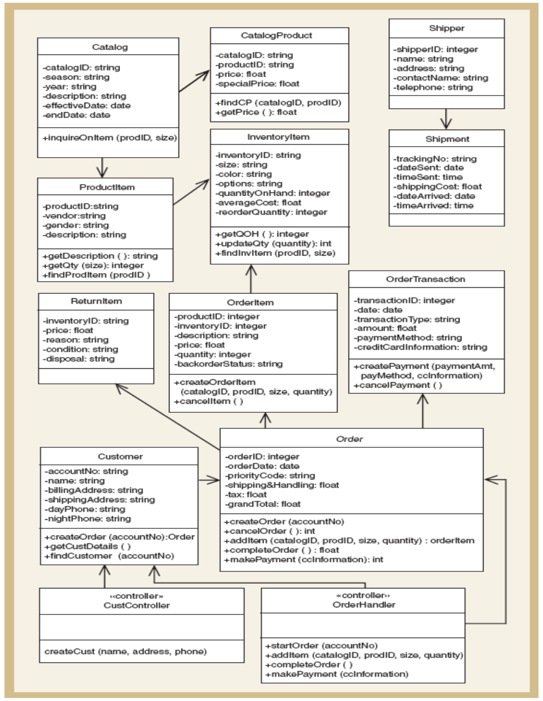


InventoryItem

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

+updateQty (qty): int

Updated Design Class Diagram for the Domain Layer



Catalog
-catalogID: string
-season: string
-year: string
-description: string
-effectiveDate: date
-endDate: date
+inquireOnItem (prodID, size)

CatalogProduct
-catalogID: string
-productID: string
-price: float
-specialPrice: float
+findCP (catalogID, prodID)
+getPrice ( ): float

Shipper
-shipperID: integer
-name: string
-address: string
-contactName: string
-telephone: string

ProductItem
-productID:string
-vendor:string
-gender: string
-description: string
+getDescription ( ): string
+getQty (size): integer
+findProdItem (prodID)

InventoryItem
-inventoryID: string
-size: string
-color: string
-options: string
-quantityOnHand: integer
-averageCost: float
-reorderQuantity: integer
+getQOH ( ): integer
+updateQty (quantity): int
+findInvItem (prodID, size)

Shipment
-trackingNo: string
-dateSent: date
-timeSent: time
-shippingCost: float
-dateArrived: date
-timeArrived: time

ReturnItem
-inventoryID: string
-price: float
-reason: string
-condition: string
-disposal: string

OrderItem
-productID: integer
-inventoryID: integer
-description: string
-price: float
-quantity: integer
-backorderStatus: string
+createOrderItem
(catalogID, prodID, size, quantity)
+cancelItem ( )

OrderTransaction
-transactionID: integer
-date: date
-transactionType: string
-amount: float
-paymentMethod: string
-creditCardInformation: string
+createPayment (paymentAmt,
payMethod, ccInformation)
+cancelPayment ( )

Customer
-accountNo: string
-name: string
-billingAddress: string
-shippingAddress: string
-dayPhone: string
-nightPhone: string
+createOrder (accountNo):Order
+getCustDetails ( )
+findCustomer (accountNo)

Order
-orderID: integer
-orderDate: date
-priorityCode: string
-shipping&Handling: float
-tax: float
-grandTotal: float
+createOrder (accountNo)
+cancelOrder ( ): int
+addItem (catalogID, prodID, size, quantity) : orderItem
+completeOrder ( ) : float
+makePayment (ccInformation): int

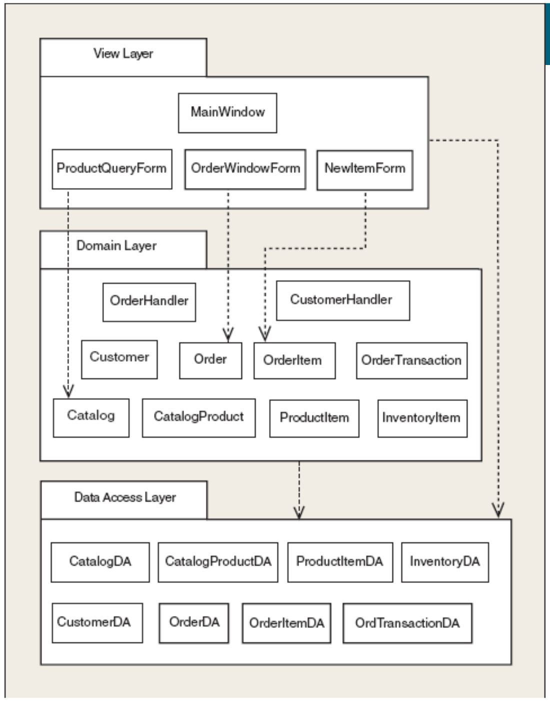
«controller»
CustController
createCust (name, address, phone)

«controller»
OrderHandler
+startOrder (accountNo)
+addItem (catalogID, prodID, size, quantity)
+completeOrder ( )
+makePayment (ccInformation)

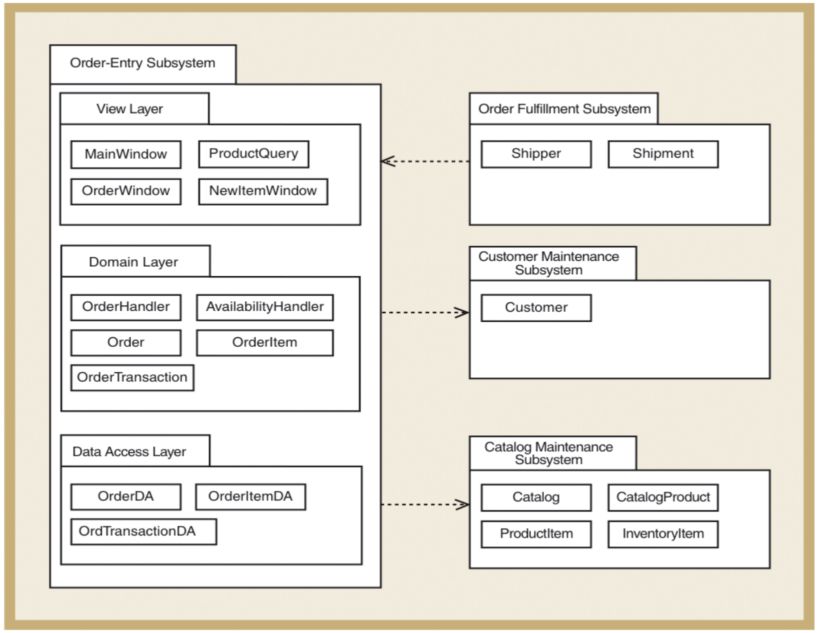
Package Diagram—Structuring the Major Components

- High-level diagram in UML to associate classes of related groups
- Identifies major components of a system and dependencies
- Determines final program partitions for each layer
  - View, domain, data access
- Can divide system into subsystem and show nesting within packages

Partial Design of Three-Layer Package Diagram for RMO



RMO Subsystem Packages



Implementation Issues for Three-Layer Design

- Construct system with programming
  - Java or VB .NET or C# .NET
  - IDE tools (Visual Studio, Rational Application Developer, JBuilder)
- Integration with user-interface design, database design, and network design
- Use object responsibility to define program responsibilities for each layer
  - View layer, domain layer, data access layer

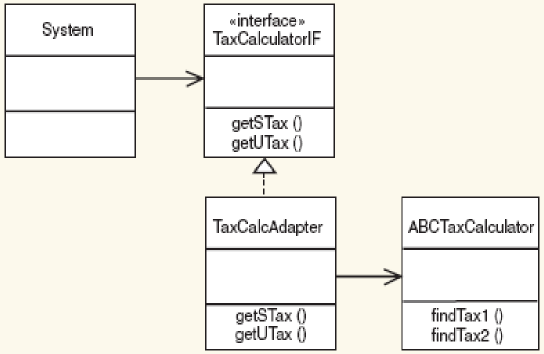
Design Patterns

Scope of pattern	Type of pattern		
	Creational	Structural	Behavioral
Class-level patterns	Factory method	Adapter	Interpreter Template Method
Object-level patterns	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Adapter Pattern

Example:

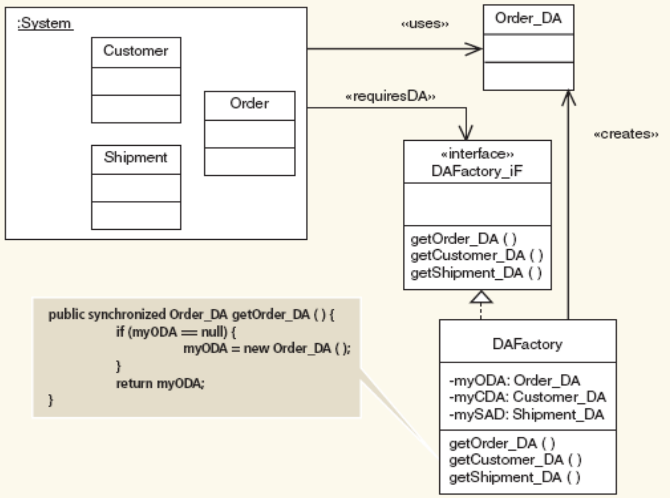
There are several places in the RMO system where class libraries were purchased to provide special processing. These purchased libraries provide specialized services such as tax calculations and shipping and postage rates. From time to time, these service libraries are updated with new versions. Sometimes a service library is even replaced with one from an entirely different vendor. The RMO systems staff applies protection from variations and indirection design principles by placing an adapter in front of each replaceable class.



Factory or Factory Method Pattern

Example:

Several places in the RMO system need to get data from an Order object and need to have a reference to an Order_DA [data access] object. The Order_DA object may or may not already have been instantiated. A data access factory is defined and an interface is created. The requesting object uses the methods defined in the interface to request the reference to the Order_DA object. It then can read the database of orders.



```
public synchronized Order_DA getOrder_DA ( ) {
            if (myODA == null) {
                        myODA = new Order_DA ( );
            }
            return myODA;
}
```

Singleton Pattern

Example:

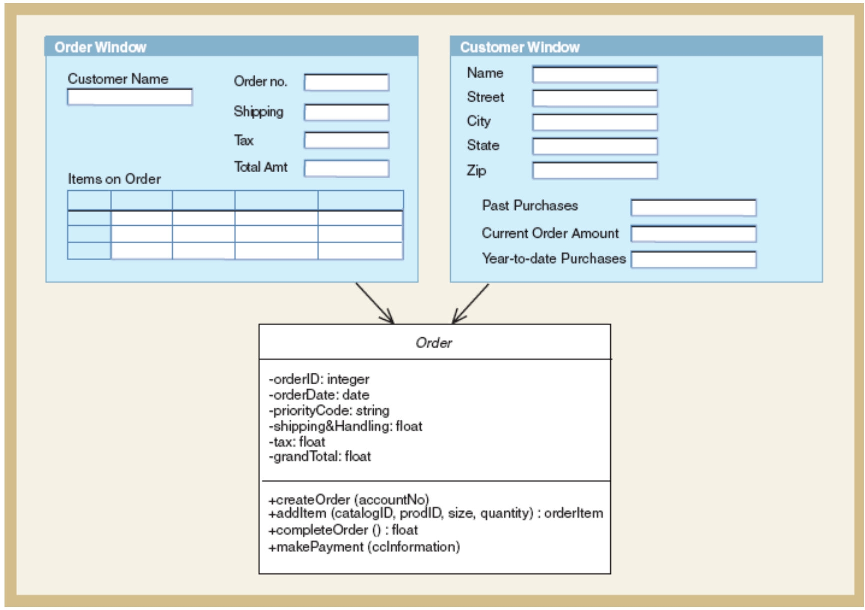
In RMO's system, the connection to the database is made through a class called Connection. However, for efficiency, we want each desktop system to open and connect to the database only once, and to do so as late as possible. Only one instance of Connection, that is, only one connection to the database, is desired. The Connection class is coded as a singleton. The following coding example is similar to C# and Java.

```
Class Connection
{
private static Connection conn = null;
public synchronized static getConnection ( )
     {
       if (conn == null) {
      conn = new Connection ( ) ;}
       return conn;
     }
}
```

Another example of a singleton pattern is a utilities class that provides services for the system, such as a factory pattern. Since the services are for the entire system, it causes confusion if multiple classes provide the same services.

An additional example might be a class that plays audio clips. Since only one audio clip should be played at one time, the audio clip manager will control that. However, for this to work, there must be only one instance of the audio clip manager.

Create new order -- Observers



Order Window

Customer Name

Order no.

Shipping

Tax

Total Amt

Items on Order

Customer Window

Name

Street

City

State

Zip

Past Purchases

Current Order Amount

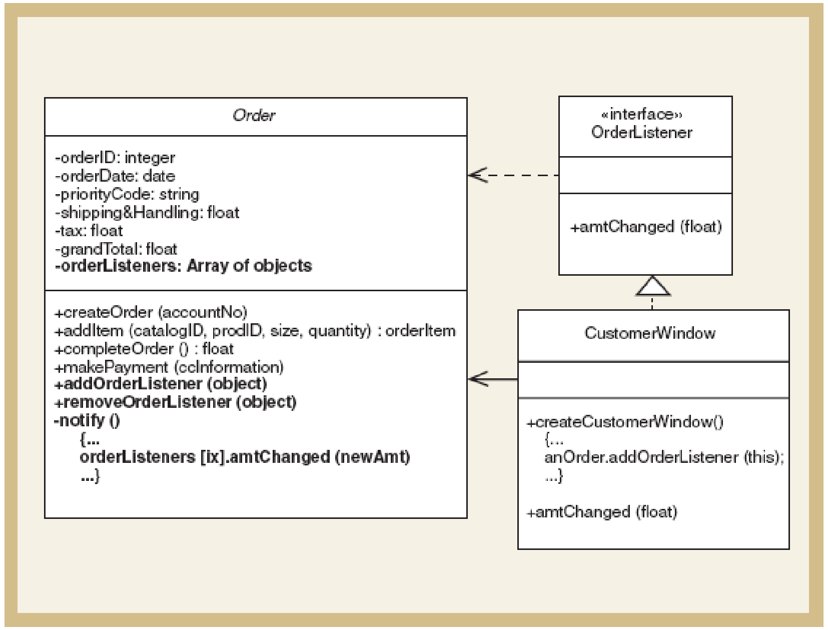
Year-to-date Purchases

Order

-orderID: integer
-orderDate: date
-priorityCode: string
-shipping&Handling: float
-tax: float
-grandTotal: float

+createOrder (accountNo)
+addItem (catalogID, prodID, size, quantity) : orderItem
+completeOrder () : float
+makePayment (ccInformation)

Implementation of Observer Pattern



Order

-orderID: integer
-orderDate: date
-priorityCode: string
-shipping&Handling: float
-tax: float
-grandTotal: float
-orderListeners: Array of objects

+createOrder (accountNo)
+addItem (catalogID, prodID, size, quantity) : orderItem
+completeOrder () : float
+makePayment (ccInformation)
+addOrderListener (object)
+removeOrderListener (object)
-notify ()
{...
orderListeners [ix].amtChanged (newAmt)
...}

«interface»
OrderListener

+amtChanged (float)

CustomerWindow

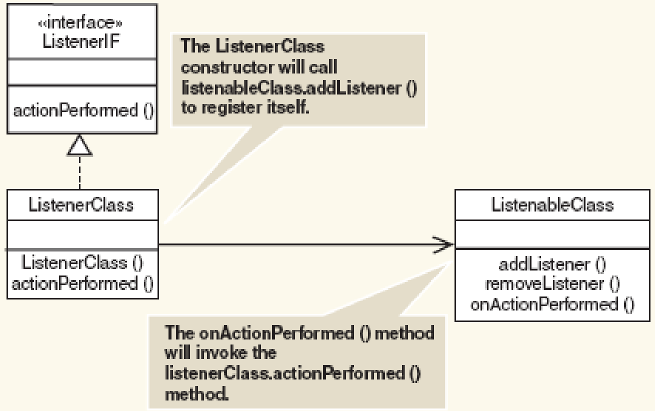
+createCustomerWindow()
{...
anOrder.addOrderListener (this);
...}

+amtChanged (float)

Observer Pattern

Example:

The entire windows event-handling system is based on this pattern. Windows components, such as text boxes and buttons, should not be directly coupled to classes and methods that process events, such as text entry or button clicks. Each component will have a method such as addListener () to allow other objects to register themselves. In windows, the listener method that notifies the listeners when an event occurs is often called *actionPerformed ().*



«interface»
ListenerIF

actionPerformed ()

ListenerClass

ListenerClass ()
actionPerformed ()

ListenableClass

addListener ()
removeListener ()
onActionPerformed ()

The ListenerClass constructor will call listenableClass.addListener () to register itself.

The onActionPerformed () method will invoke the listenerClass.actionPerformed () method.

Summary

- Systems design is driven by use cases, design class diagrams, and sequence diagrams
  - Domain class diagrams are transformed into design class diagrams
  - Sequence diagrams are extensions of system sequence diagrams (SSDs)
- System Sequence Diagrams
  - Capture method signatures
  - Show navigation visibility
- Package Diagrams show subsystem organization and dependencies
- Design Patterns are useful solutions to standard programming problems
  - Use Case Controller (MVC pattern)
  - Adpater
  - Factory and Singleton
  - Observer