Chapter 11
Object-oriented design use case realizations

Dr. Supakit Nootyaskool
Faculty of Information Technology
King Mongkut's Institute of Technology Ladkrabang

Outline

- Detailed Design of Multilayer Systems
- Use Case Realization with Sequence Diagrams
- Designing with Communication Diagrams
- Updating and Packaging the Design Classes
- Design Patterns

Objective

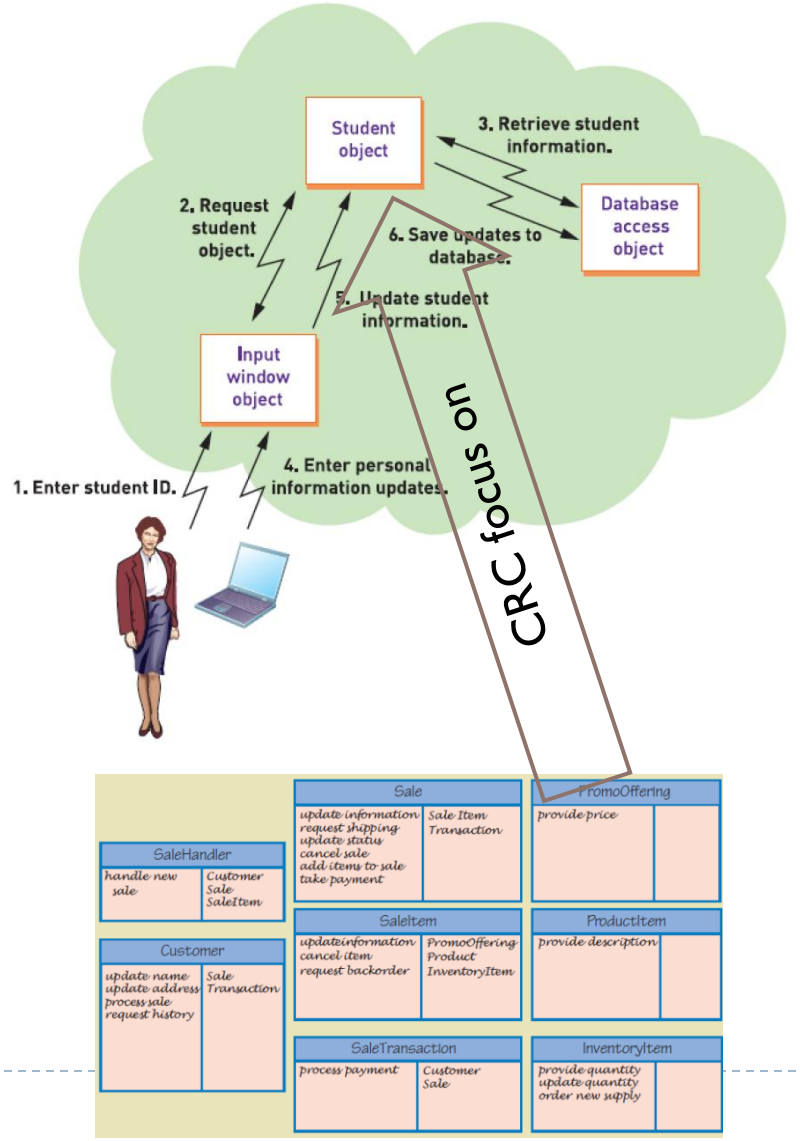
- Explain
  - The different types of objects and layers in a design
  - Design patterns and recognize various specific patterns
- Develop
  - Sequence diagrams for use case realization
  - Communication diagrams for detailed design
  - Updated design class diagrams
  - Multilayer subsystem packages

Overview

- Chapter 10 introduced software design concepts for OO programs, multi-layer design, use case realization using the CRC cards technique, and fundamental design principles
- This chapter continues the discussion of OO software design at a more advanced level
- Three layer design is demonstrated using sequence diagrams, communication diagrams, package diagrams, and design patterns
- Design is shown to proceed use case by use case, and within each use case, layer by layer

11.1 Detailed design of multilayer systems

- From Chap10.1 has three objects
  - Input windows object (Screen)
  - Student object (business logic layer)
  - Database access object
- Communication between object
- Called three layers or model-view-controller architecture.



11.1.1 Pattern and the use case controller

- Pattern = template is a set of instruction or decorative design to be followed in making an item or component.
  - Cut fabric
  - Build building
  - Sound system
  - Products
  - …
- Design pattern is standard templates for developing software that uses the concept of object-oriented design.
  - Presented in 1996, "Element of Reuseble Object-Oriented Software"
    - Gang of Four (GoF)

11.1 Detailed design of multilayer systems

- Questions that come up include
  - How do objects get created in memory?
  - How does the user interface interact with other objects?
  - How are objects handled by the database?
  - Will other objects be necessary?
  - What is the lifespan of each object?

- In the use case, the message comes from the external actor to a windows class that is electronics input form

11.1 Detailed design of multilayer systems

- The first example of a programming design pattern shown is the Controller Pattern.
  - Problem is deciding how to handle all of the messages from the view layer to classes in the problem domain layer to reduce coupling
  - Solution is to assign one class between the view layer and the problem domain layer that receives all messages and acts as a switchboard directing messages to the problem domain

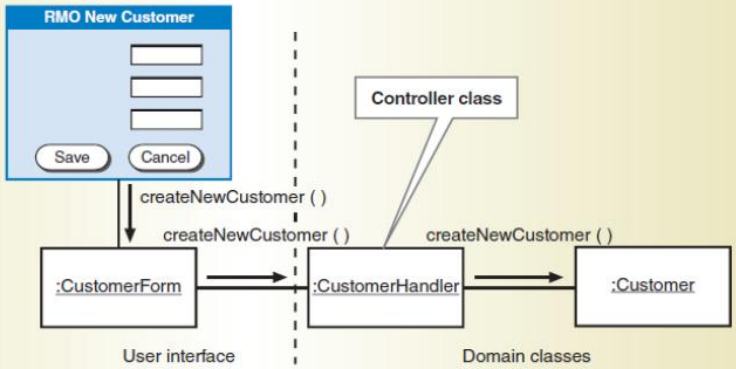
- The use case controller is the intermediate class that act as buffer between user interface and the domain classes.
  - Convert data format
  - Preparing data before calculation.

11.1 Detailed design of multilayer systems

- They are the specific of the controller pattern having five elements.
  - Pattern name
  - Problem that requires a solution
  - Solution to or explanation of the pattern
  - Example of the pattern
  - Benefits and consequences of the pattern

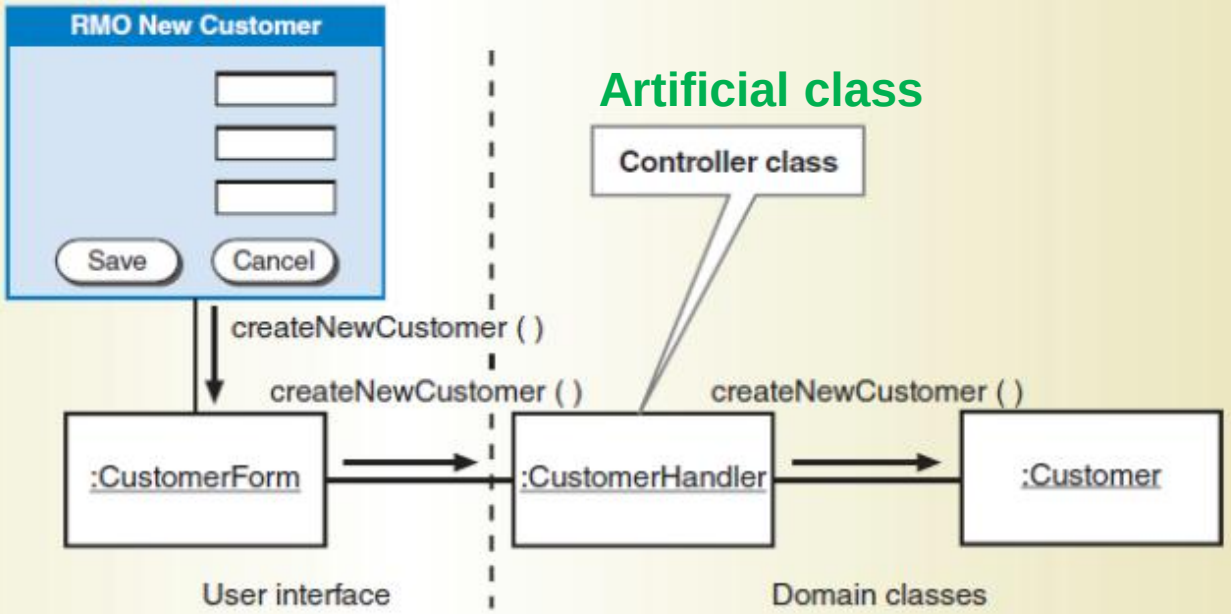
11.1 Detailed design of multilayer systems

Pattern specification for the 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 Customer account subsystem accepts inputs from a :CustomerForm window. These input messages are passed to the :CustomerHandler, which acts as the switchboard to forward the message to the correct problem domain class.  Other cases of the controller pattern will be used for each RMO use case.
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.

Example:

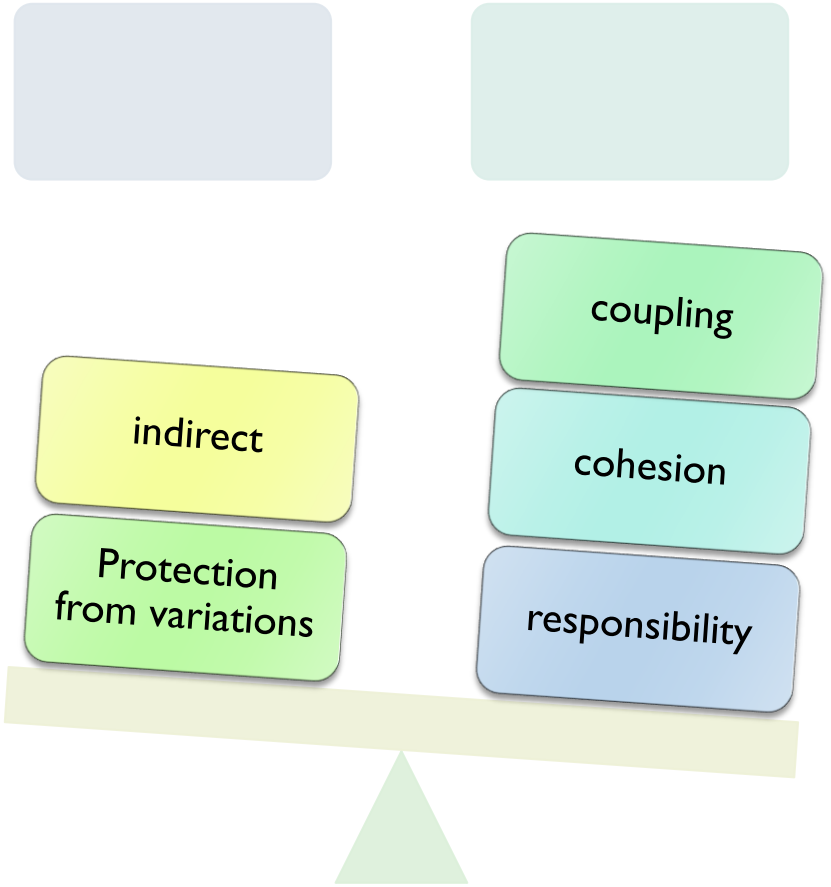
The RMO Customer account subsystem accepts inputs from a :CustomerForm window. These input messages are passed to the :CustomerHandler, which acts as the switchboard to forward the message to the correct problem domain class.



Other cases of the controller pattern will be used for each RMO use case.

11.1 Detailed design of multilayer systems

The design principles should balance many factors.

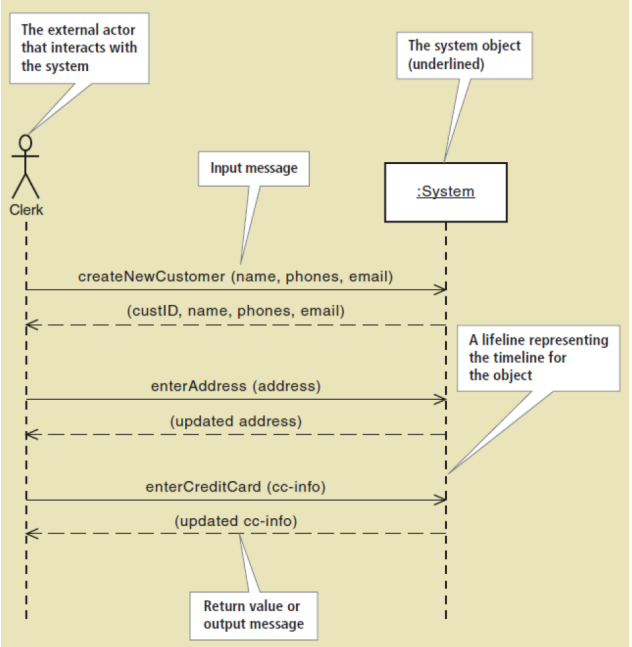


11.2 Use case realization with sequence diagrams

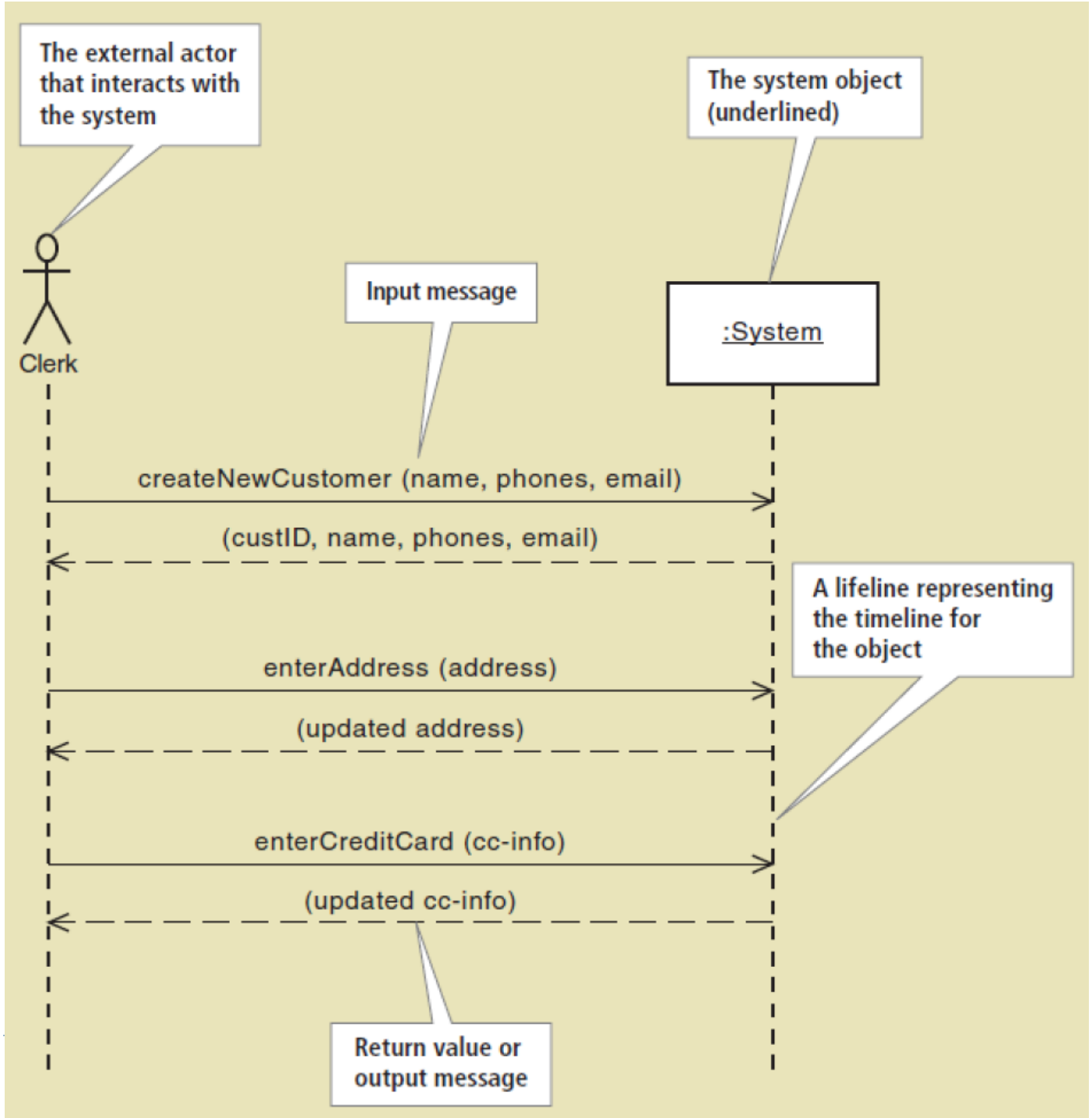
- Use case realization—the process of elaborating the detailed design of a use case with interaction diagrams
- Two types of interaction diagrams
  - UML sequence diagram emphasizes the sequence of messages sent between objects.
    - Single-layer design
    - Multilayer design
  - UML communication diagram emphasizes the sequence of message sent and receive to object.

11.2.1 Sequence diagram

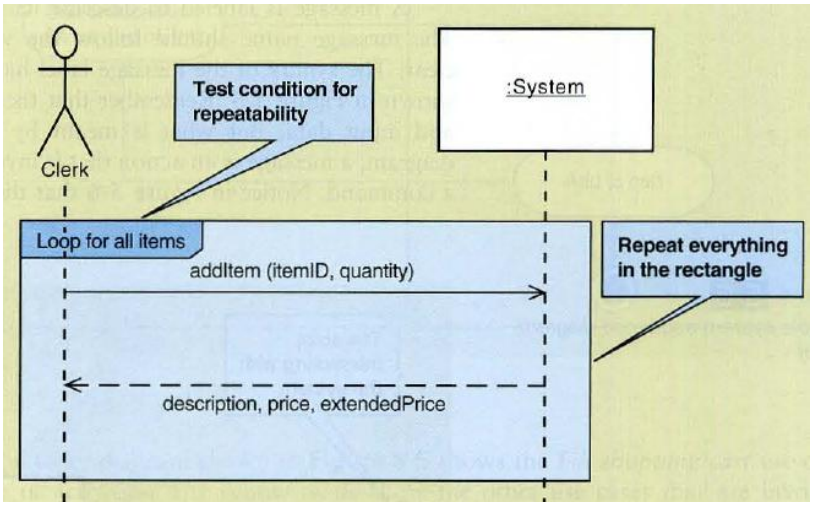
- Sequence diagrams, use case realization sequence diagrams, extend the system sequence diagram (SSD) to show:
  - View layer objects
  - Domain layer objects (usually done first)
  - Data access layer objects

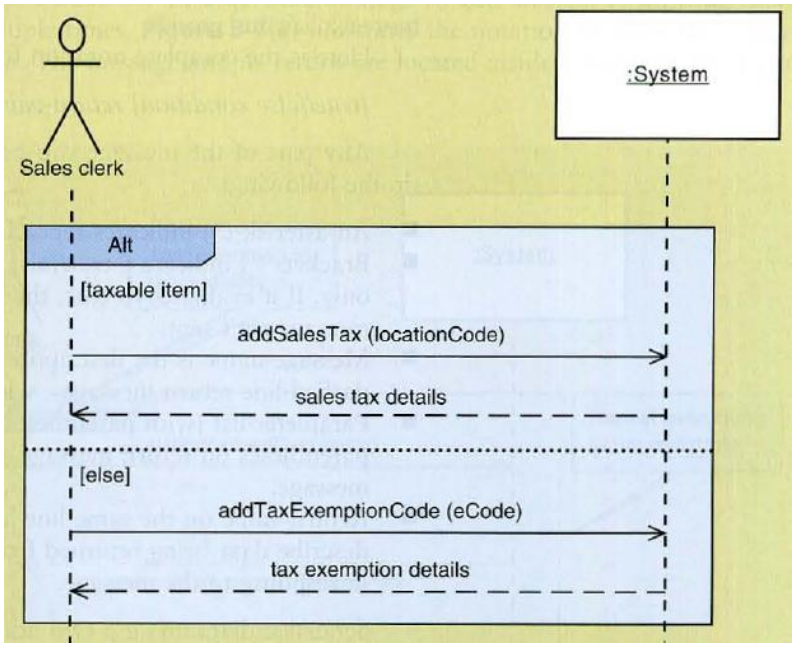


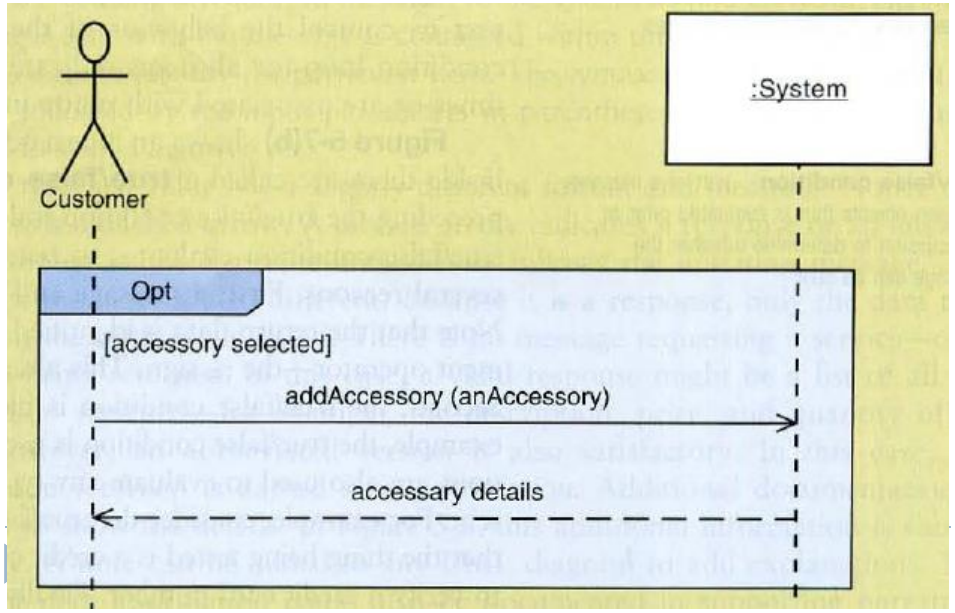
***[true/false condition] return value := message-name (parameter-list)**


The external actor that interacts with the system
The system object (underlined)
Clerk
Input message
:System
createNewCustomer (name, phones, email)
(custID, name, phones, email)
A lifeline representing the timeline for the object
enterAddress (address)
(updated address)
enterCreditCard (cc-info)
(updated cc-info)
Return value or output message

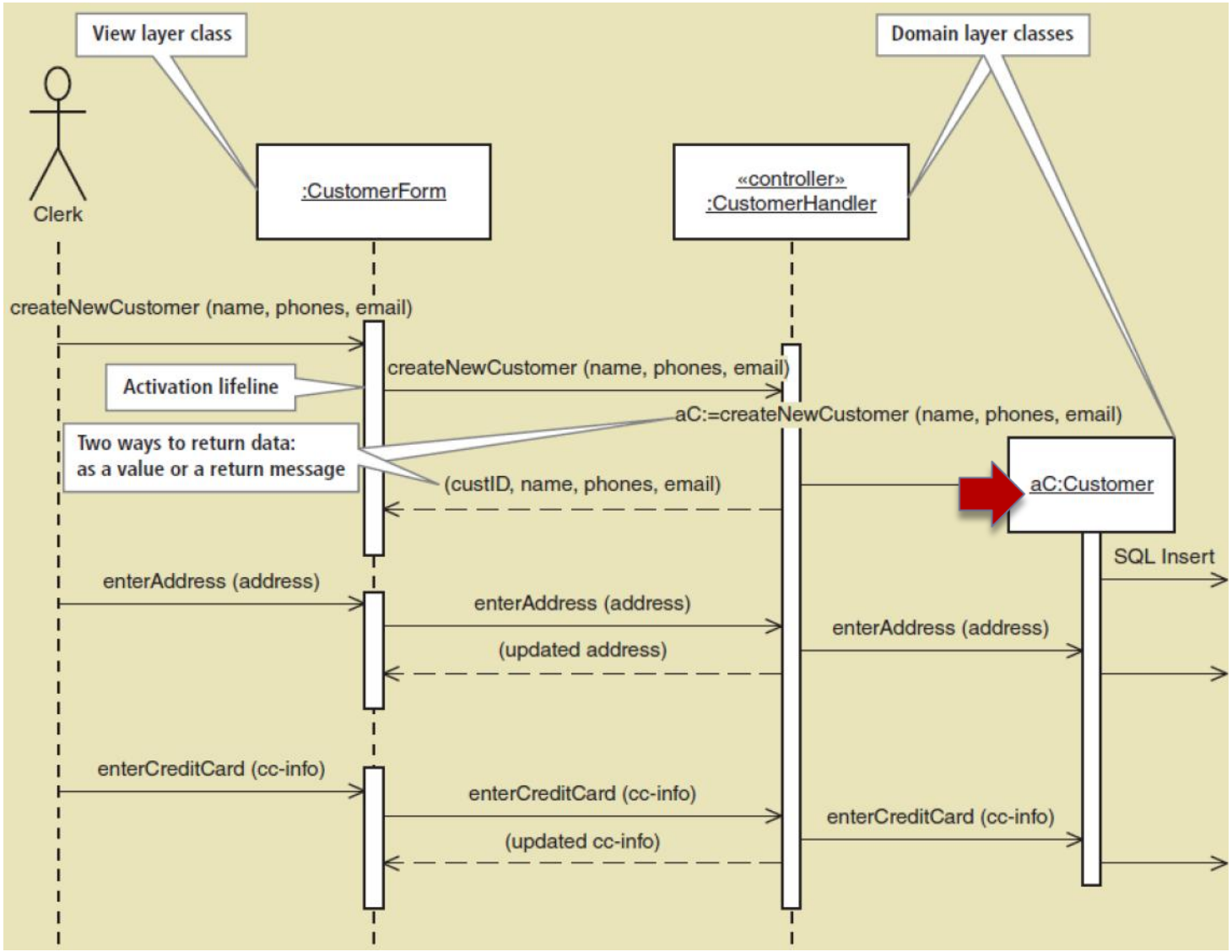
11.2.1 Sequence diagram: frames







11.2.1 Sequence diagram: Example, two-level details design

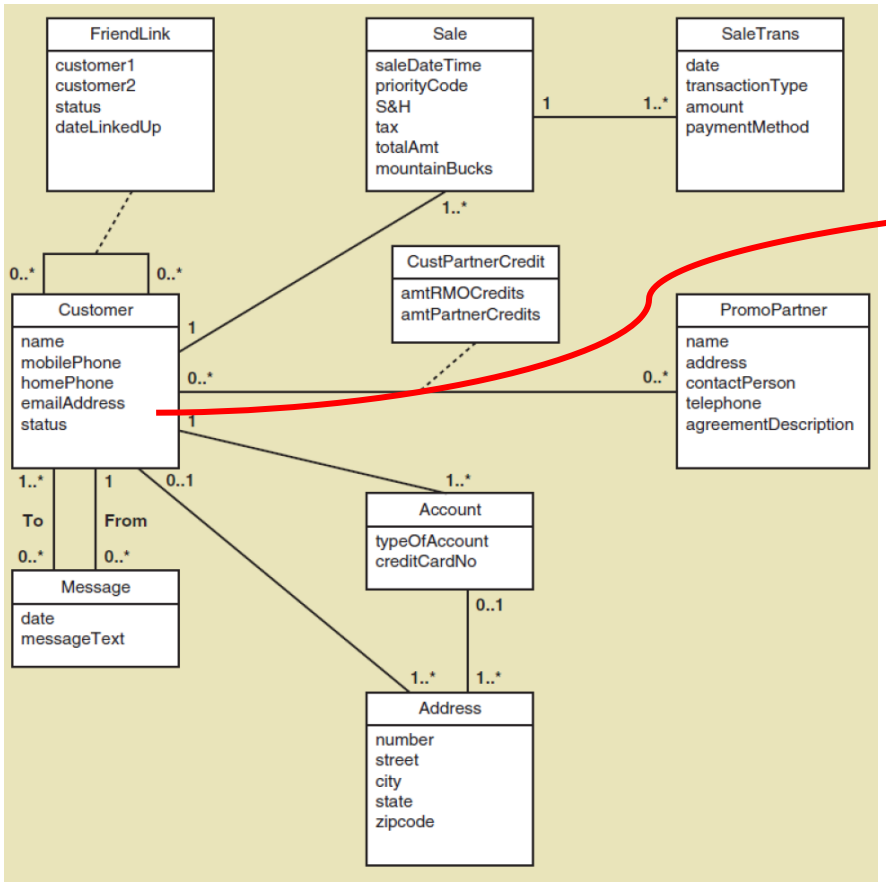


Note of expanded sequence diagram

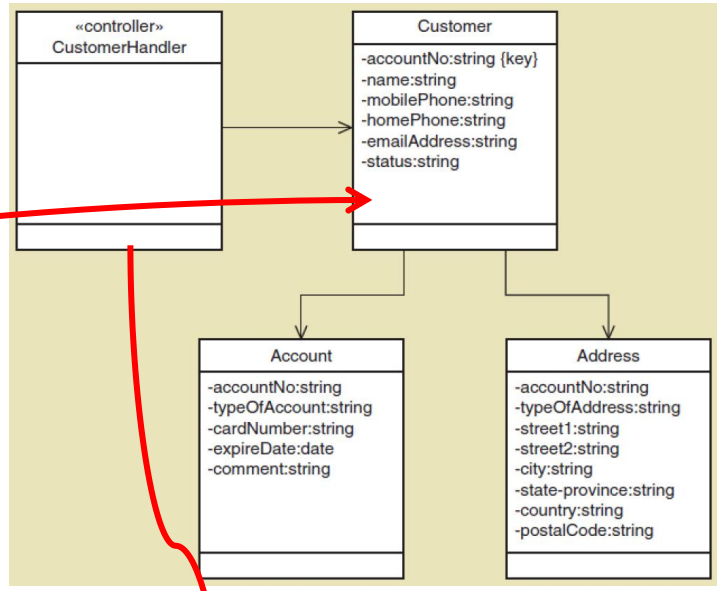
- This is a two layer architecture, as the domain class Customer knows about the database and executes SQL statements for data access
- Three layer design would add a data access class to handle the database resulting in higher cohesiveness and loose coupling
- Note :
  - CustomerForm is an object of 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

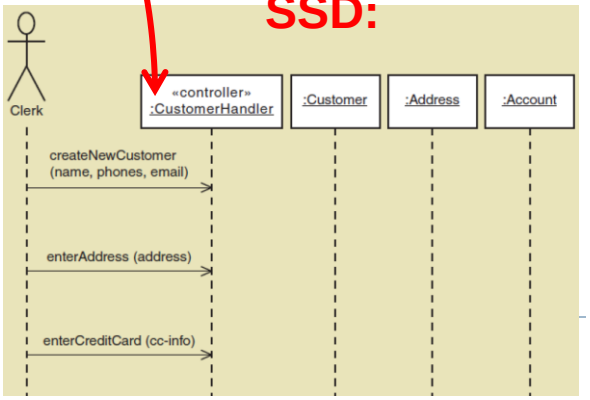
Domain model: (Chap4)
Customer account system



Design class diagram:
Create customer account use case

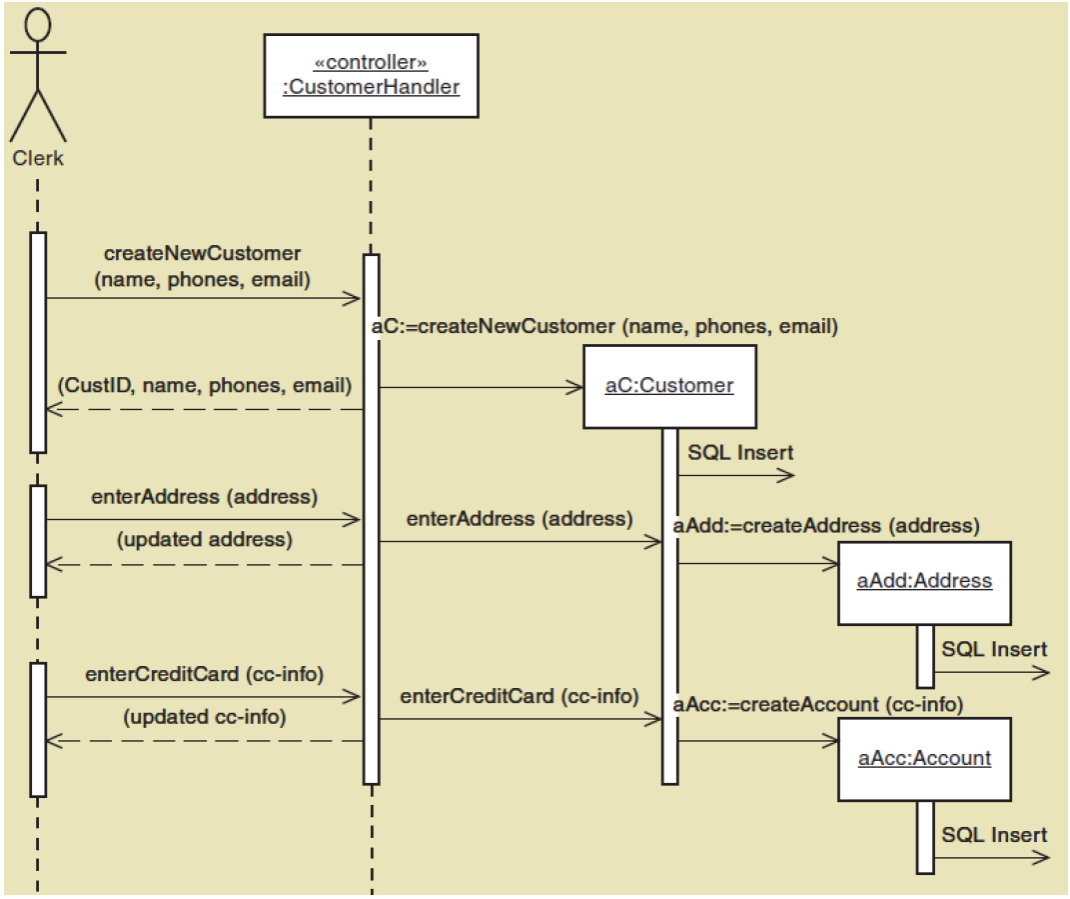


SSD:



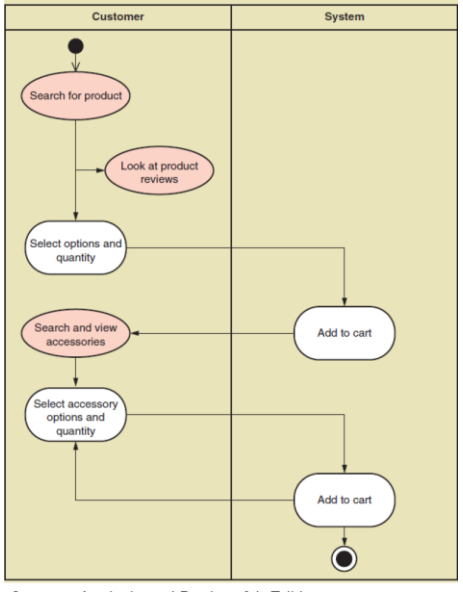
11.2.2 First-cut sequence diagram: Example create *customer account* Use case

- 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

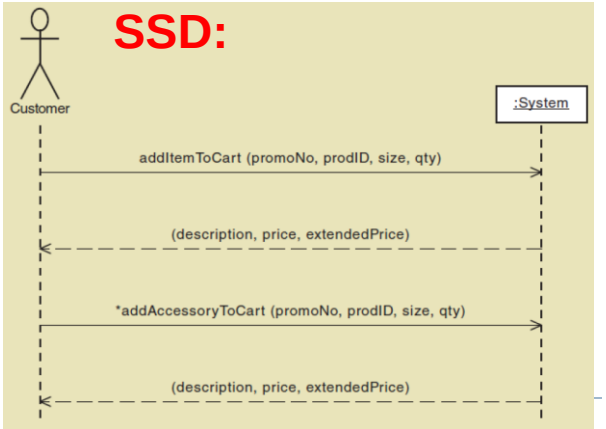


11.2.3 First-cut sequence diagram: Example Fill *shopping cart* Use case

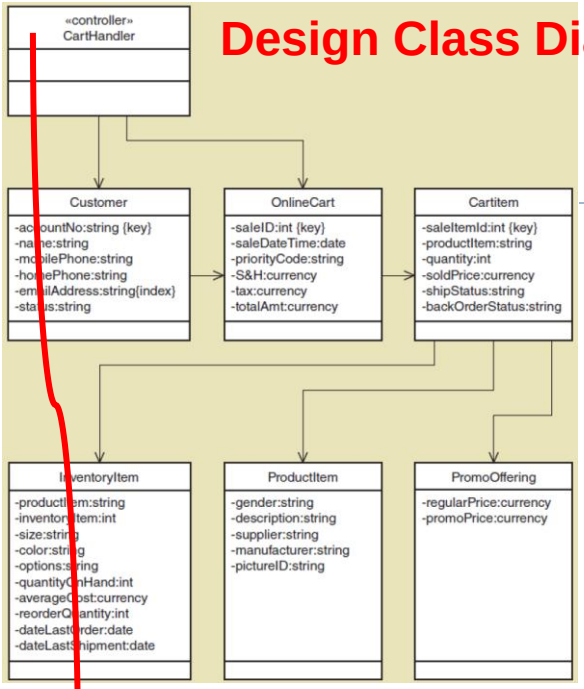
Activity diagram: (Chap3)



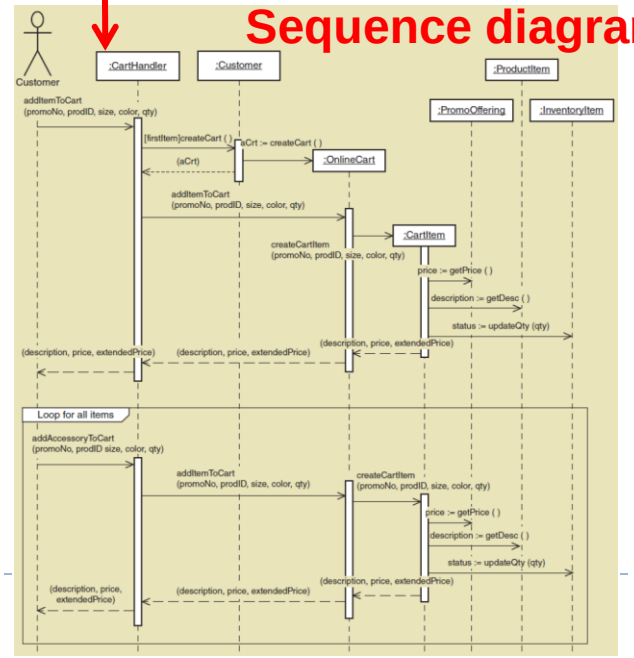
SSD:



Design Class Diagram



Sequence diagram



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.

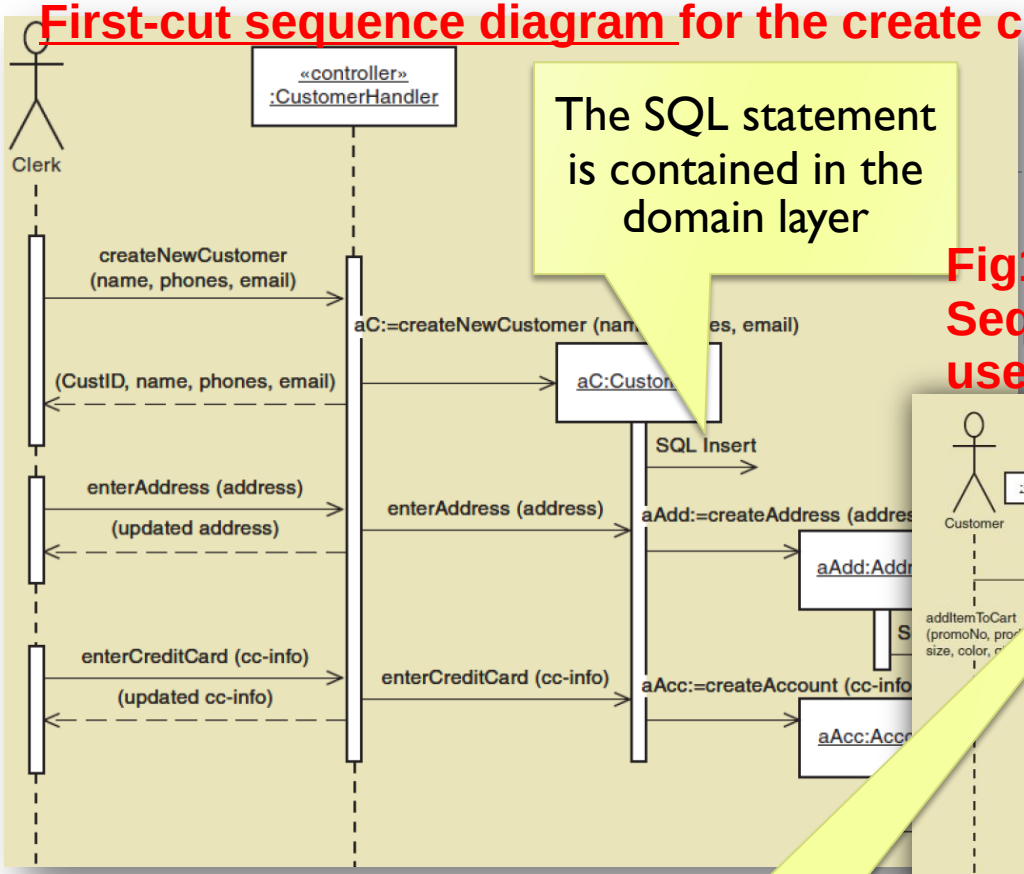
Solving

- 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

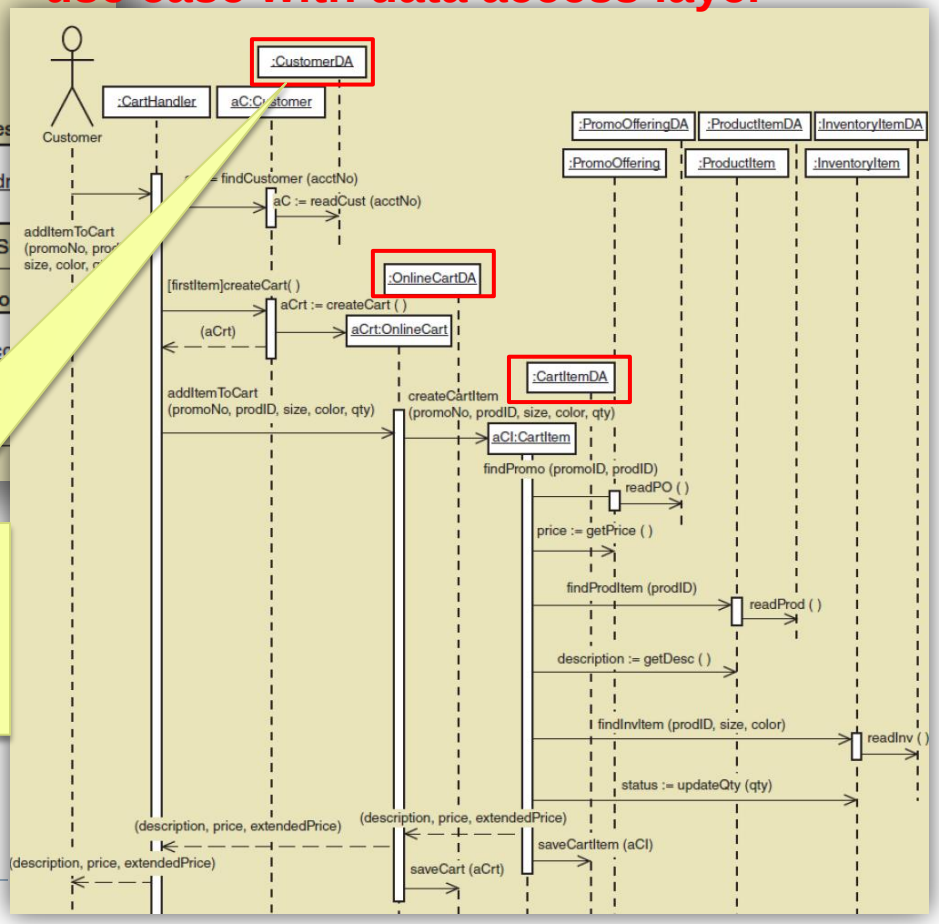
1) View layer
   - Get input data or commands
   - Show output or command responding
2) Domain layer
3) Data access layer

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

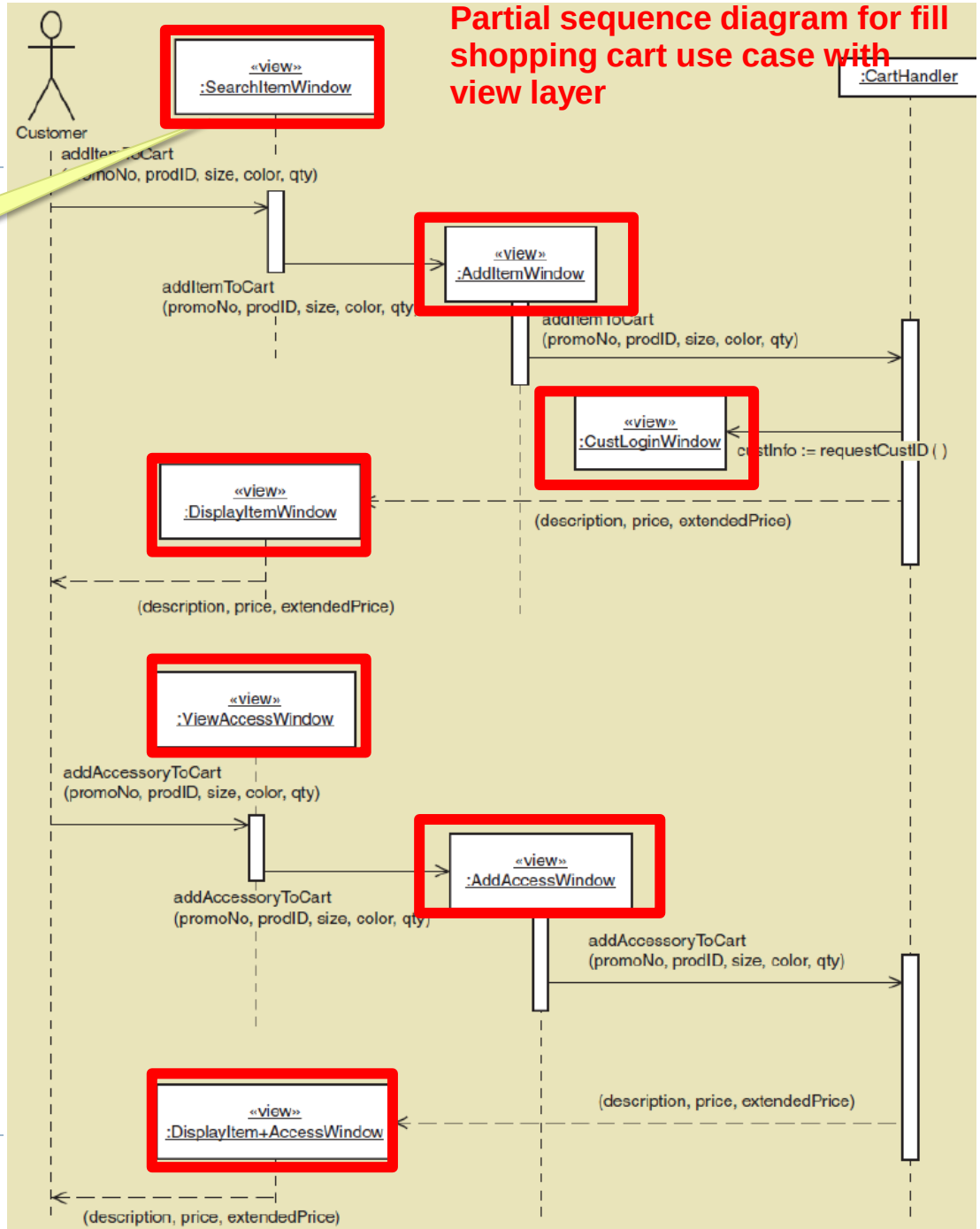
Data access notes

- :CartHandler findCustomer(acctN0) message to Customer class means Customer should create a new instance named aC, send a message to :CustomerDA asking it to read info from the database for a customer with that account number, and then populate the new customer instance with the attribute values from the database.
- The other times a domain object is needed, a similar pattern is used, such as when needing information from :PromoOffering, :ProductItem, :InventoryItem from each :CartItem to display in the :OnLineCart. :PromoOfferingDA, :ProductItemDA, and :InventoryItemDA are asked to find the data and populate the instances.
- :CartItem and :OnlineCart ask DA classes to save them

11.2.4 Developing a multilayer design: View layer

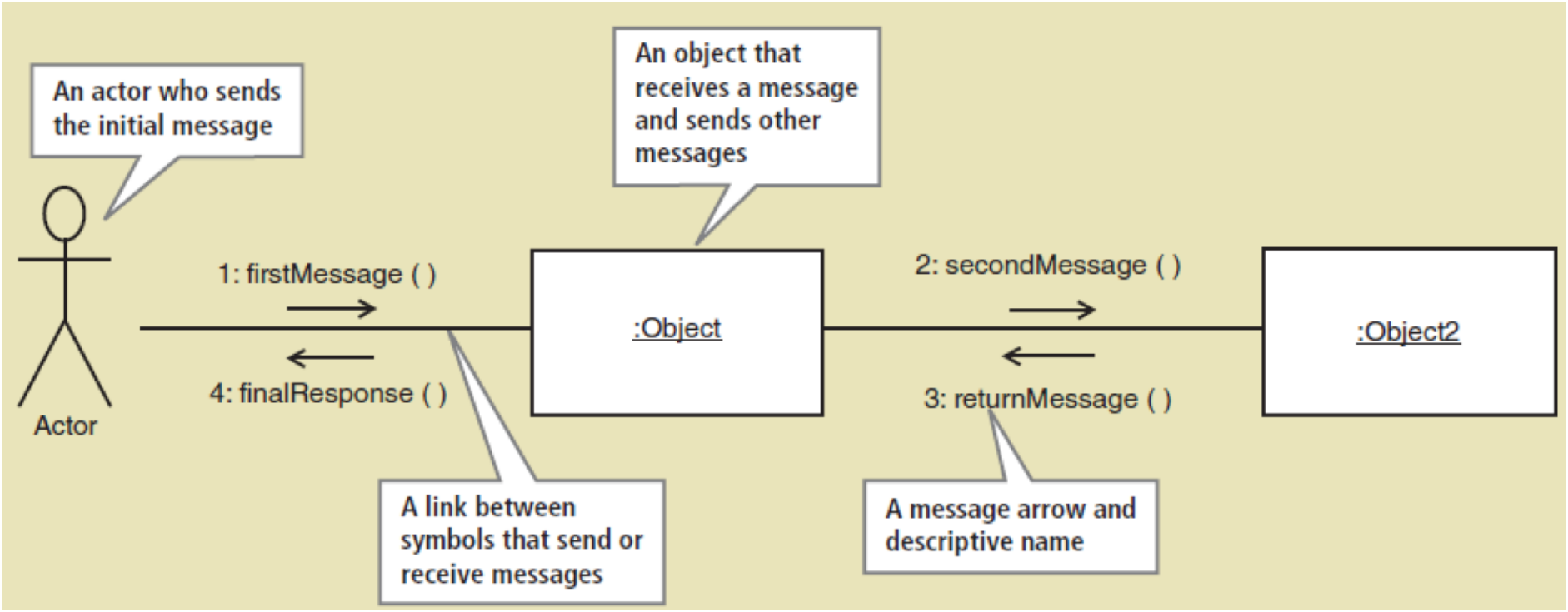
View layer

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



11.3 Designing communication diagrams

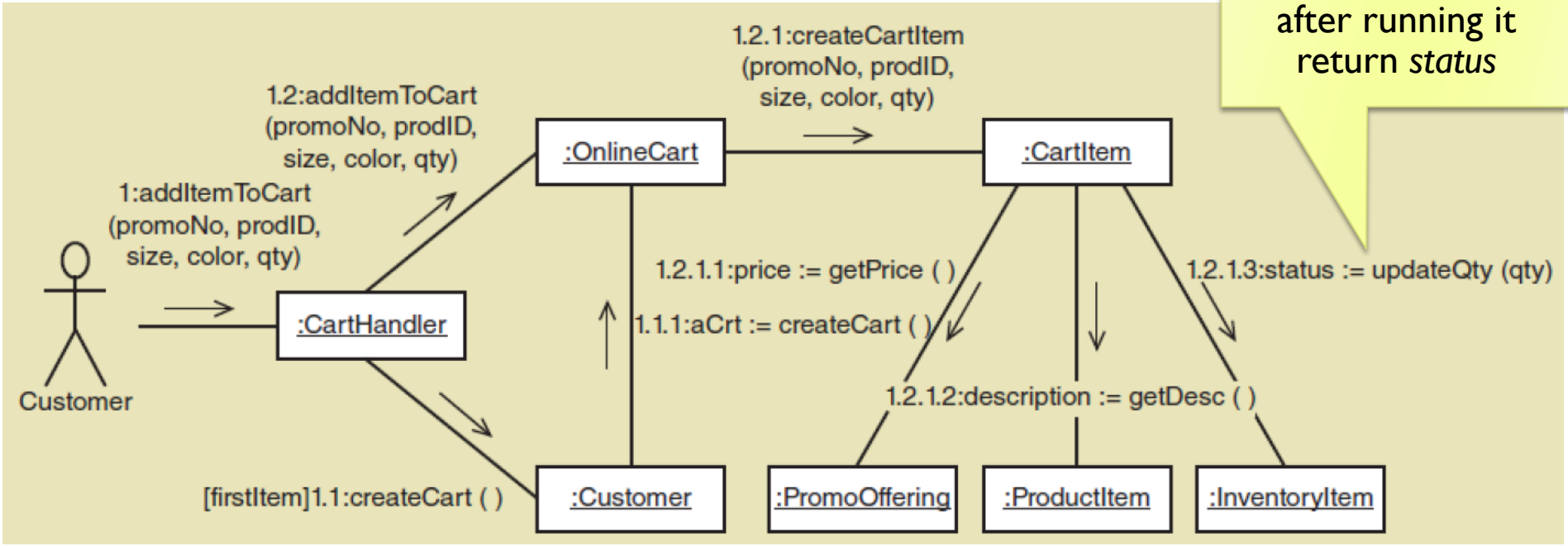
- Shows the same information as a sequence diagram
- Symbols used in a communication diagram:



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

11.3 Designing communication diagrams Example Fill Shopping Cart use case

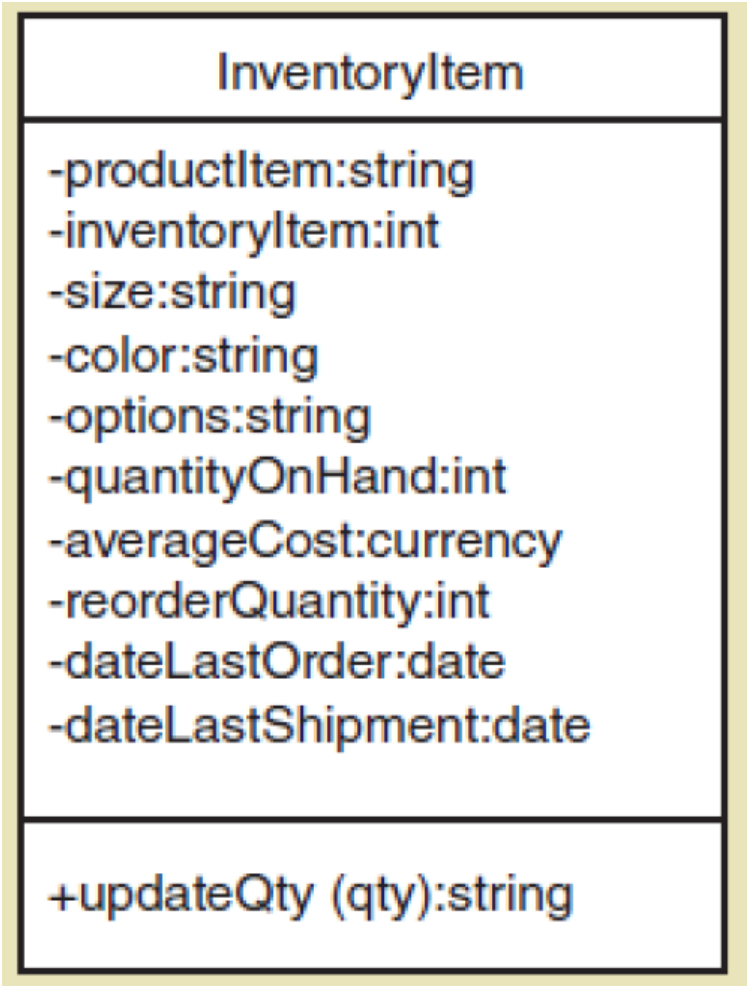
- This diagram should math the domain layer sequence diagram shown earlier.
- Many people prefer them for brainstorming



11.4 Updating and packaging the design classes

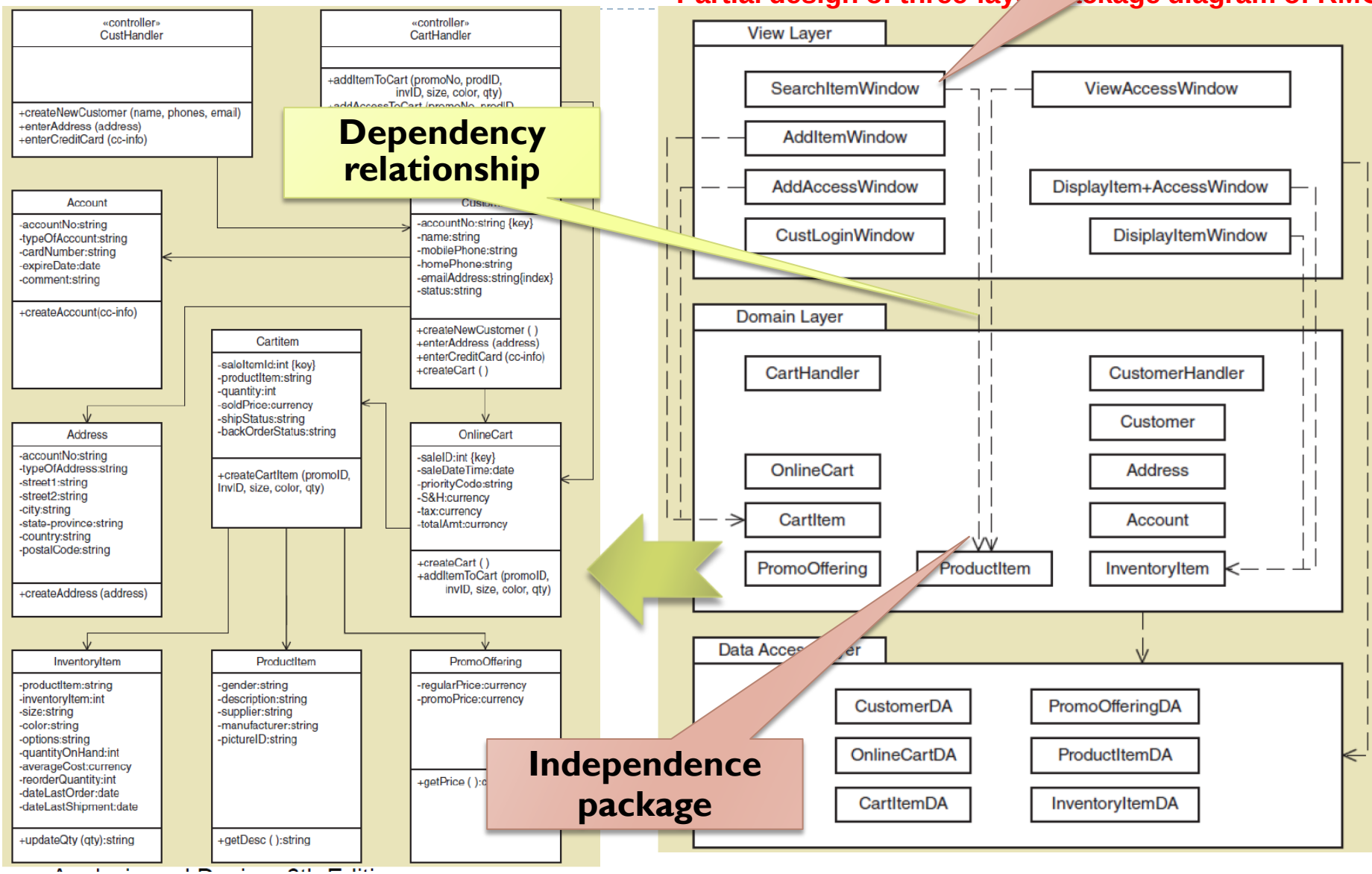
- Design class diagram (DCD) focuses on domain layer

- The first step in updating DCD is to add the three-method signatures
  1) Constructor methods
  2) Data-get/set method
  3) Use case-specific method

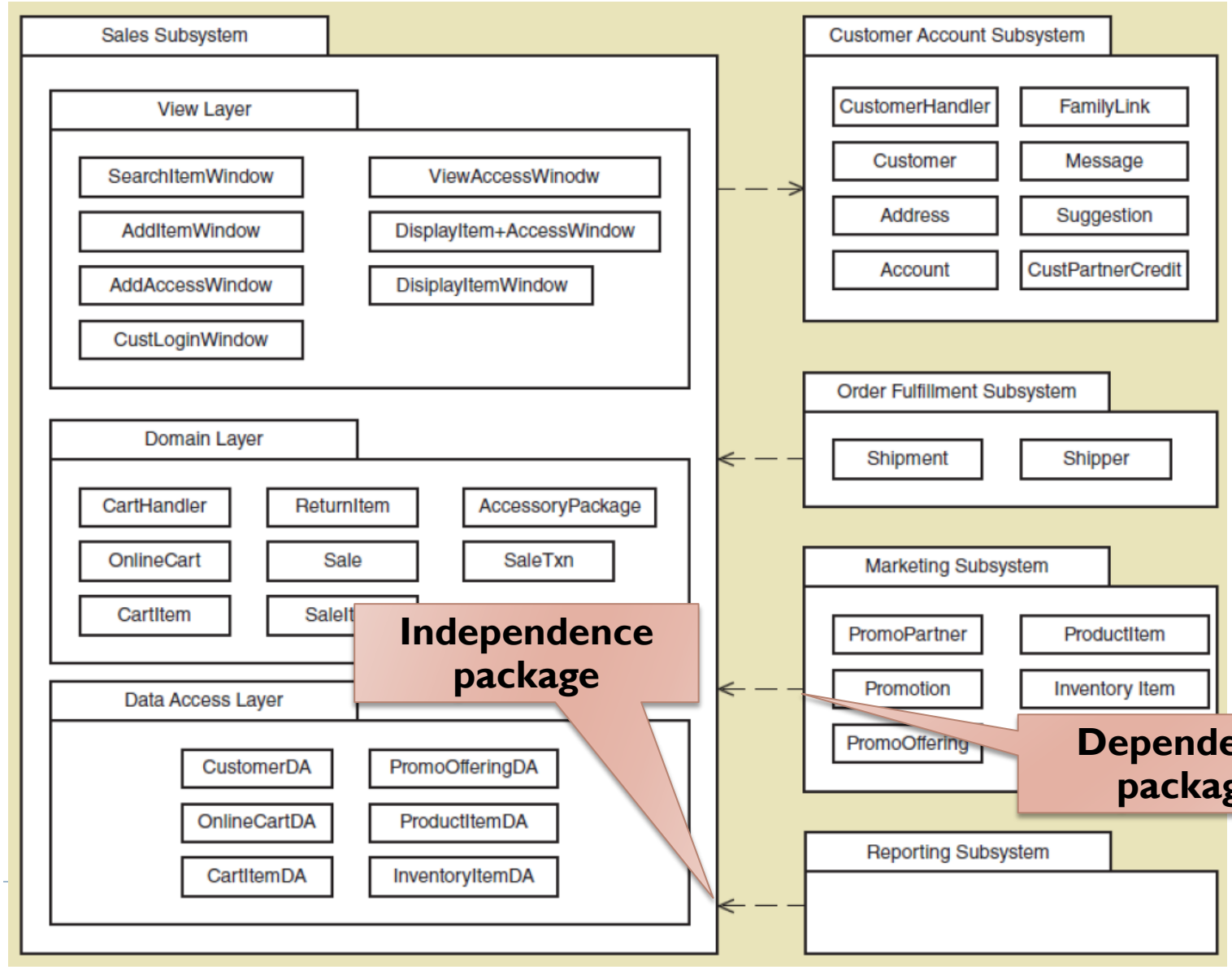


11.4 Update design class diagram for domain layer

Fig11-19
Partial design of three-laye ckage diagram of RMO



11.4 Update design class diagram for domain layer RMO subsystem package diagram



11.4.1 Implementation issues
Three layer design

- View Layer Class Responsibilities
  - Display electronic forms and reports.
  - Capture such input events as clicks, rollovers, and key entries.
  - Display data fields.
  - Accept input data.
  - Edit and validate input data.
  - Forward input data to the domain layer classes.
  - Start and shut down the system.

11.4.1 Implementation issues
Three layer design (2)

- Domain Layer Class Responsibilities
  - Create problem domain (persistent) classes.
  - Process all business rules with appropriate logic.
  - Prepare persistent classes for storage to the database.
- Data Access Layer Class Responsibilities
  - Establish and maintain connections to the database.
  - Contain all SQL statements.
  - Process result sets (the results of SQL executions) into appropriate domain objects.
  - Disconnect gracefully from the database.

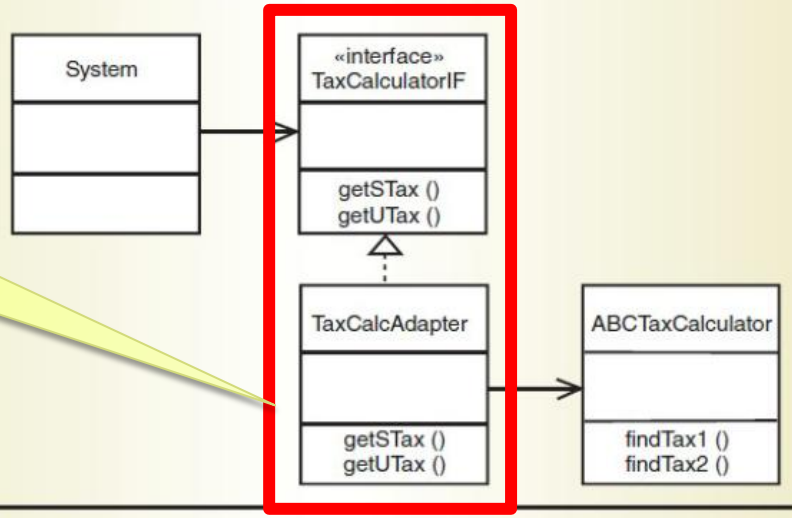
11.5 Design patterns

- Adapter
  - Like an electrical adapter
  - The adapter class is inserted to convert the method calls from within the system to the method names in the external class.

- Factory
  - Encapsulation data to protect client's object access or modify source of the data.
  - Use factor class when creation logic is complex for a set of classes
- Singleton
  - Use when only one instance should exist at a time and is shared

11.5.1 Adapter design pattern

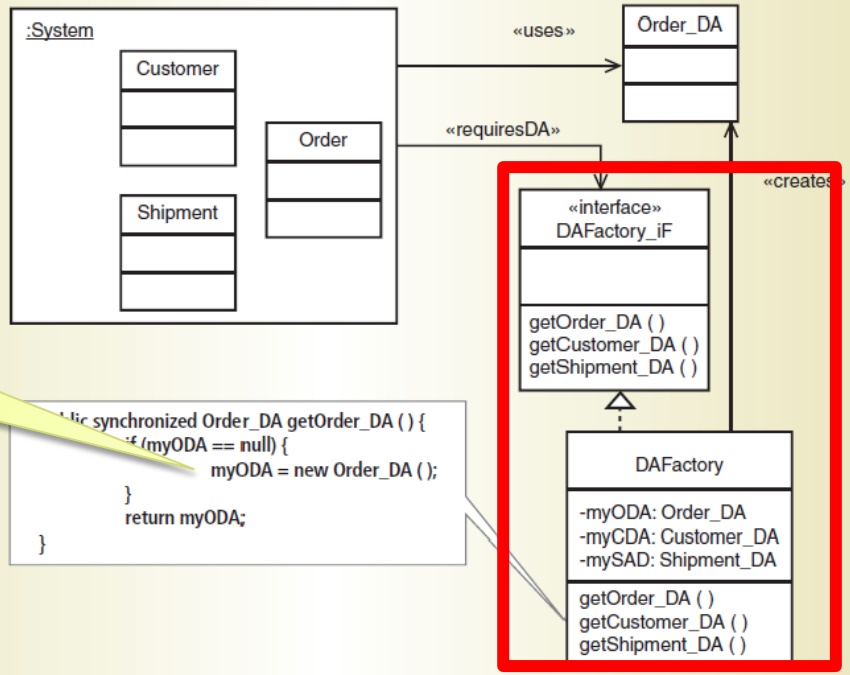
Name:	Adapter
Problem:	A class must be replaced, or is subject to being replaced, by another standard or purchased class. The replacing class already has a predefined set of method signatures that are different from the method signatures of the original class. How do you link in the new class with a minimum of impact so that you don't have to change the names throughout the system to the method names in the new class?
Solution:	Write a new class, the adapter class, which serves as a link between the original system and the class to be replaced. This class has method signatures that are the same as those of the original class (and the same as those expected by the system). Each method then calls the correct desired method in the replacement class with the method signature. In essence, it "adapts" the replacement class so that it looks like the original class.
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.
Benefits and consequences:	The adaptee class can be replaced as desired. Changes are confined to the adapter class and do not ripple through the system. Two classes are defined, an interface class and the adapter class. Passed parameters may add more complexity, and it is difficult to limit changes to the adapter class.



Adapter class

11.5.2 Factory design pattern

Name:	Factory or Factory Method
Problem:	Who should be responsible for creating utility type objects that do not specifically belong to the problem domain classes? These utility objects may also be accessed from various places within the system, so a given object may need to be instantiated from several classes.
Solution:	Create an artifact that is a factory class. Its responsibility is only to instantiate utility classes. In many situations, only one instance of a particular utility class is allowed. Hence, all classes that need access to the class come through the factory. The factory ensures that only one instance is created.
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.
Benefits and Consequences:	Higher cohesion of problem domain classes Less coupling between business logic layer and data layer Smaller, more maintainable classes



Create new object of Order_DA

11.5.3 Singleton design pattern

Name:	Singleton
Problem:	Only one instantiation of a class is allowed. The instantiation (new) can be called from several places in the system. The first reference should make a new instance, and later attempts should return a reference to the already instantiated object. How do you define a class so that only one instance is ever created?
Solution:	A singleton class has a static variable that refers to the one instance of itself. All constructors to the class are private and are accessed through a method or methods, such as getInstance(). The getInstance() method checks the variable; if it is null, the constructor is called. If it is not null, then only the reference to the object is returned.

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. Because 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.

Benefits and consequences:	There are other times when only one instance of an object is needed, but if it is instantiated from only one place, then a singleton may not be required. The singleton object controls itself and ensures that only one instance is created—no matter how many times it is called and wherever the call occurs in the system. The code to implement the singleton is very simple, which is one of the desirable characteristics of a good design pattern.

Summary

- This chapter went into more detail about use case realization and three layer design to extend the design techniques from last chapter
- Three layer design is an architectural design pattern, part of the movement toward the use of design principles and patterns.
- Use case realization is the design of a use case, done with a design class diagram and sequence diagrams. Using sequence diagrams allows greater depth and precision than using CRC cards.
- Use case realization proceeds use case by use case (use case driven) and then for each use case, it proceeds layer by layer

Summary (2)

- Starting with the business logic/domain layer, domain classes are selected and an initial design class diagram is drawn.
- The systems sequence diagram (SSD) from analysis is expanded by adding a use case controller and then the domain classes for the use case.
- Messages and returns are added to the sequence diagram as responsibilities are assigned to each class.
- The design class diagram is then updated by adding methods to the classes based on messages they receive and by updating navigation visibility.
- Simple use case might be left with two layers if the domain classes are responsible for database access. More complex systems add a data access layer as a third layer to handle database access

Summary (3)

- The view layer can also be added to the sequence diagram to show how multiple pages or forms interact with the use case controller.
- The UML communication diagram is also used to design use case realization and it shows the same information as a sequence diagram.
- The UML package diagram is used to structure the classes into packages, usually one package per layer. The package diagram can also be used to package layers into subsystems.
- Design patterns are a standard solutions or templates that have proven to be effective approaches to handling design problems. The design patterns in this chapter include controller, adapter, factory, and singleton