# INTRODUCTION TO SYSTEMS ANALYSIS AND DESIGN:

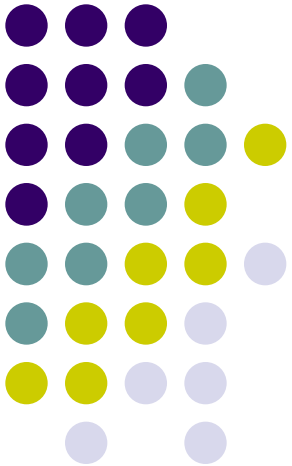
## AN AGILE, ITERATIVE APPROACH

SATZINGER | JACKSON | BURD

Chapter 11

# Object-Oriented Design: Use Case Realizations

# Chapter 11



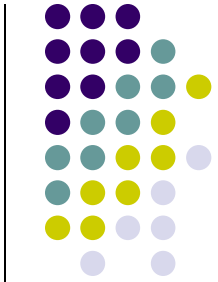
Introduction to Systems Analysis and Design: An Agile, Iteractive Approach 6th Ed

Satzinger, Jackson & Burd

# Chapter 11 Outline

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

# Learning Objectives

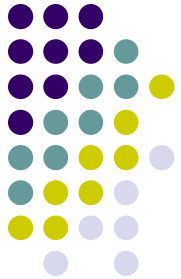


- Explain the different types of objects and layers in a design
- Develop sequence diagrams for use case realization
- Develop communication diagrams for detailed design
- Develop updated design class diagrams
- Develop multilayer subsystem packages
- Explain design patterns and recognize various specific patterns

# Overview

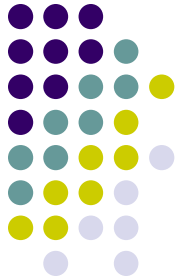
- Chapter 10 introduced software design concepts for OO programs, multi-layer design, use case realization using the CRC cards technique, and fundamental design principles
- This chapter continues the discussion of OO software design at a more advanced level
- Three layer design is demonstrated using 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

# Detailed Design of Multilayer Systems



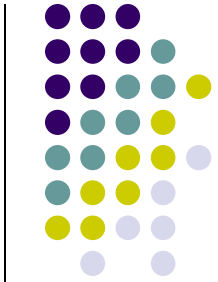
- CRC Cards focuses on the business logic, also known as problem domain layer of classes
- Three layers include view layer, business logic/problem domain layer, and data access layer
- Questions that come up include
  - How do objects get created in memory?
  - How does the user interface interact with other objects?
  - How are objects handled by the database?
  - Will other objects be necessary?
  - What is the lifespan of each object?

# Design Patterns



- Design Pattern—standard design techniques and templates that are widely recognized as good practice
- For common design/coding problems, the design pattern suggests the best way to handle the problem.
- They are written up in design pattern catalogs/references. Include:
  - Pattern name
  - Problem that requires solution
  - The pattern that solves the problem
  - An example of the pattern
  - Benefits and consequences of the a pattern

# Design Patterns

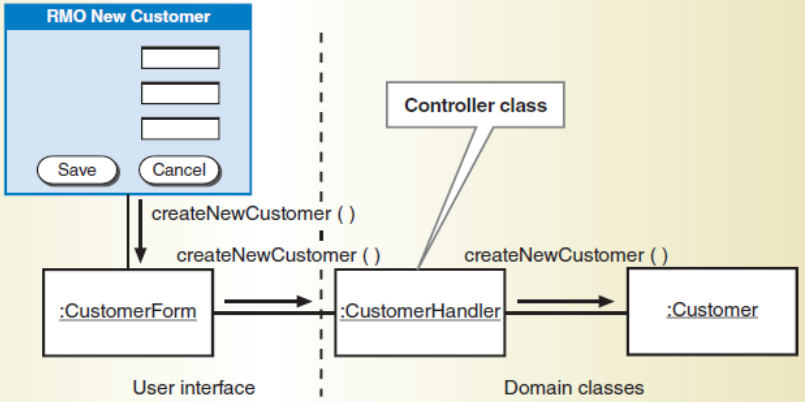


- Design patterns became widely accepted after the publication of *Elements of Reusable Object-Oriented Software*,(1996) by Gamma et al (the "Gang of Four")
- There are architectural design patterns talked about already
  - Three layer or model-view-controller architecture
- The first example of a programming 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

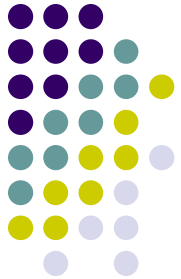
# Controller Pattern

**First step toward multilayer architecture**

**More design patterns are at the end of the chapter**

| | |
|---|---|
| **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?<br><br>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:<br>(a) Have a single class that represents the entire system, or<br>(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.<br>  <br><br>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.<br>The controller provides a layer of indirection.<br><br>The controller is closely coupled to many domain classes.<br>If care is not taken, controller classes can become incoherent, with too many unrelated functions.<br>If care is not taken, business logic will be inserted into the controller class. |

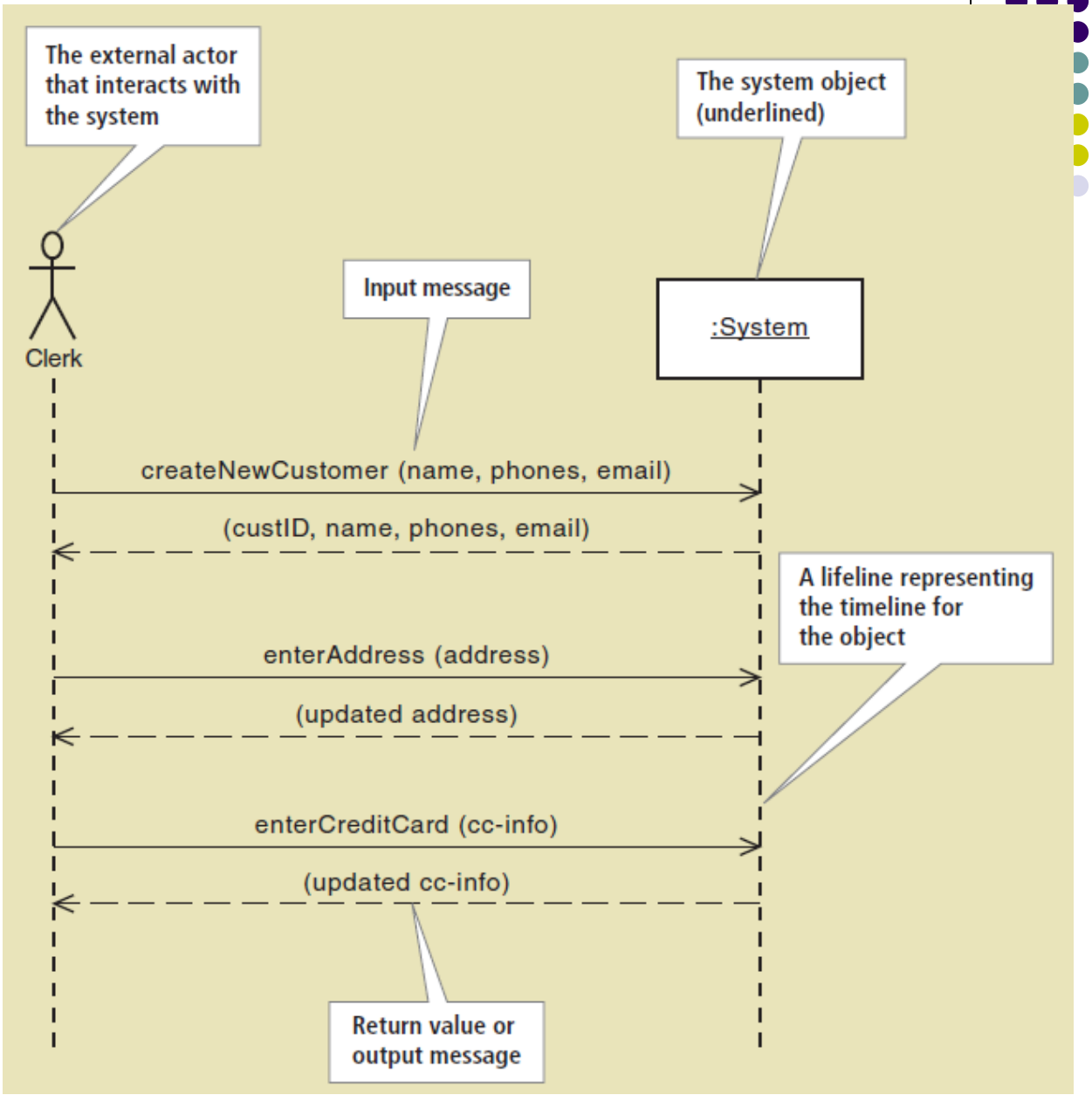
# Use Case Realization with Sequence Diagrams



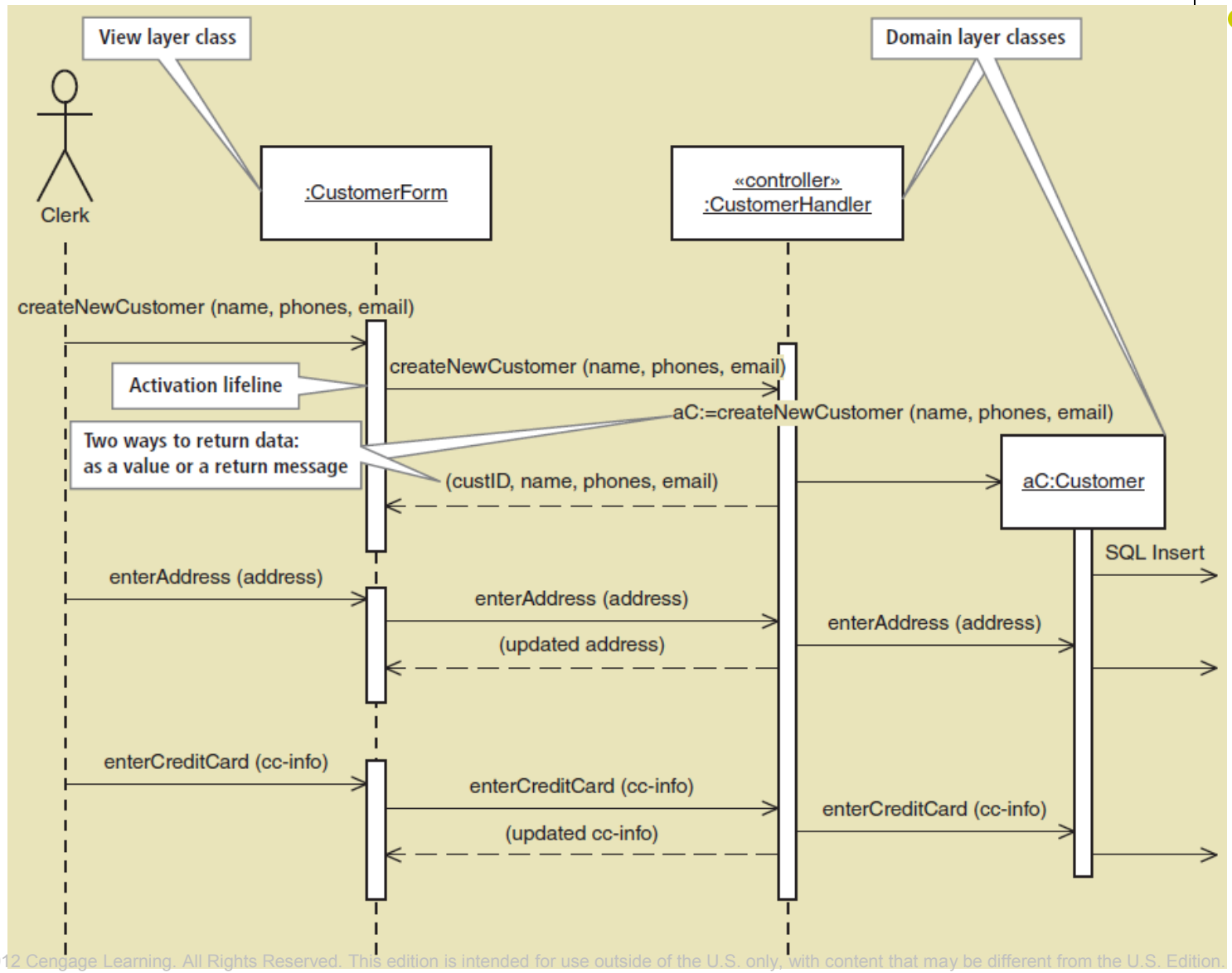
- Use case realization—the process of elaborating the detailed design of a use case with interaction diagrams
- Two types of interaction diagrams
  - UML sequence diagram (emphasized in text)
  - UML communication diagram (also introduced)
- Sequence diagrams, aka use case realization sequence diagrams, extend the system sequence diagram (SSD) to show:
  - View layer objects
  - Domain layer objects (usually done first)
  - Data access layer objects

# Start with System Sequence Diagram (SSD)

## Use case *Create customer account*



# Sequence Diagram to show View Layer and Part of Problem Domain Layer

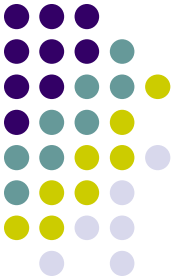


# Notes of Expanded Sequence Diagram

- 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 becuase they are objects)
- aC:Customer is an object of the Customer class known by reference variable named aC
- Note: the create message to aC:Customer points to the object symbol to indicate create

# Overview of Detailed Design Steps

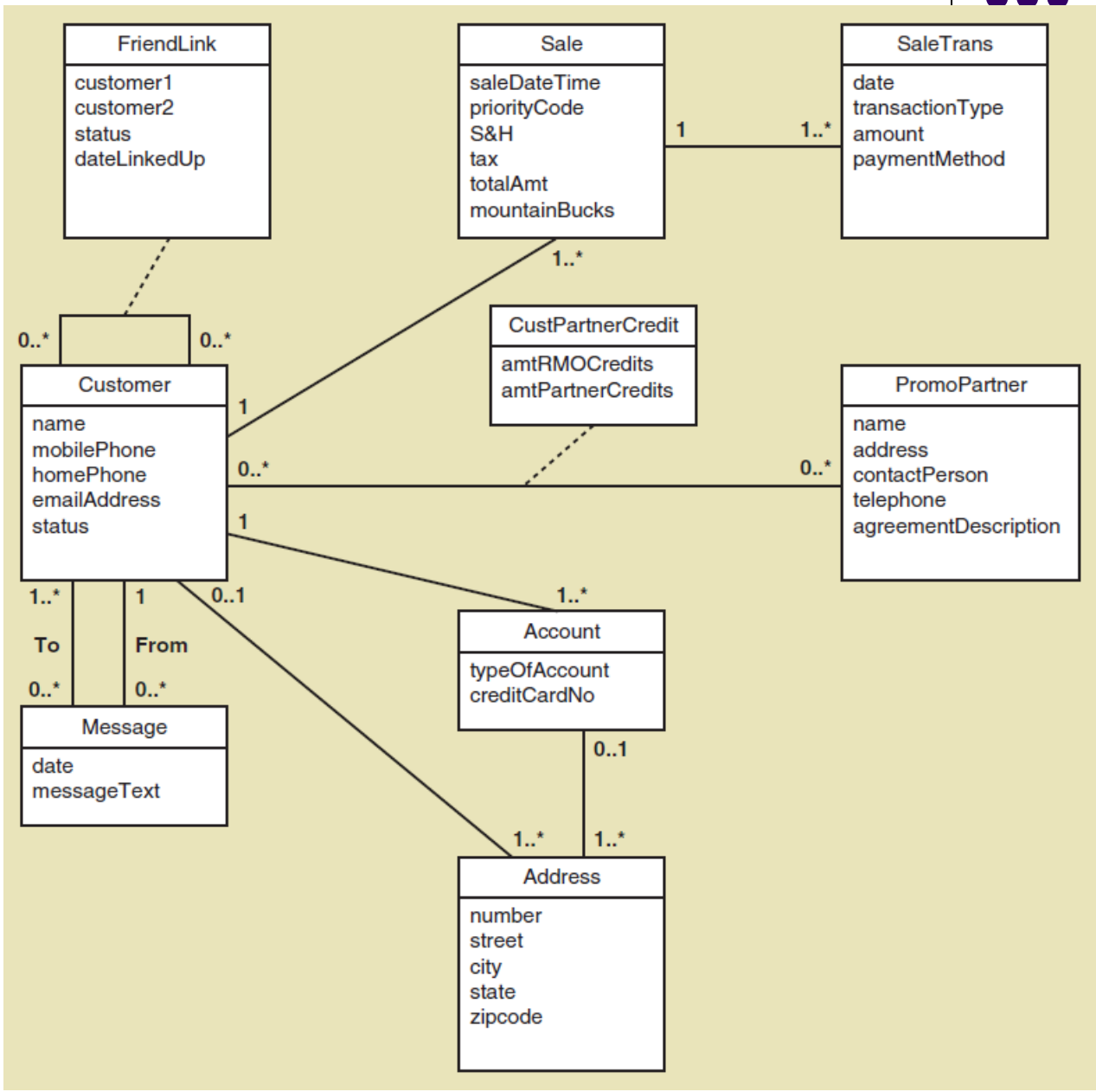
## Shown previously in Chapter 10



| Design Step | Chapter |
|---|---|
| 1. Develop the first-cut design class diagram showing navigation visibility. | 10 |
| 2. Determine the class responsibilities and class collaborations for each use case using CRC cards. | 10 |
| 3. Develop detailed sequence diagrams for each use case.<br>(a) Develop the first-cut sequence diagrams.<br>(b) Develop the multilayer sequence diagrams. | 11 |
| 4. Update the DCD by adding method signatures and navigation information. | 11 |
| 5. Partition the solution into packages, as appropriate. | 11 |

# *Create Customer Account* Use Case

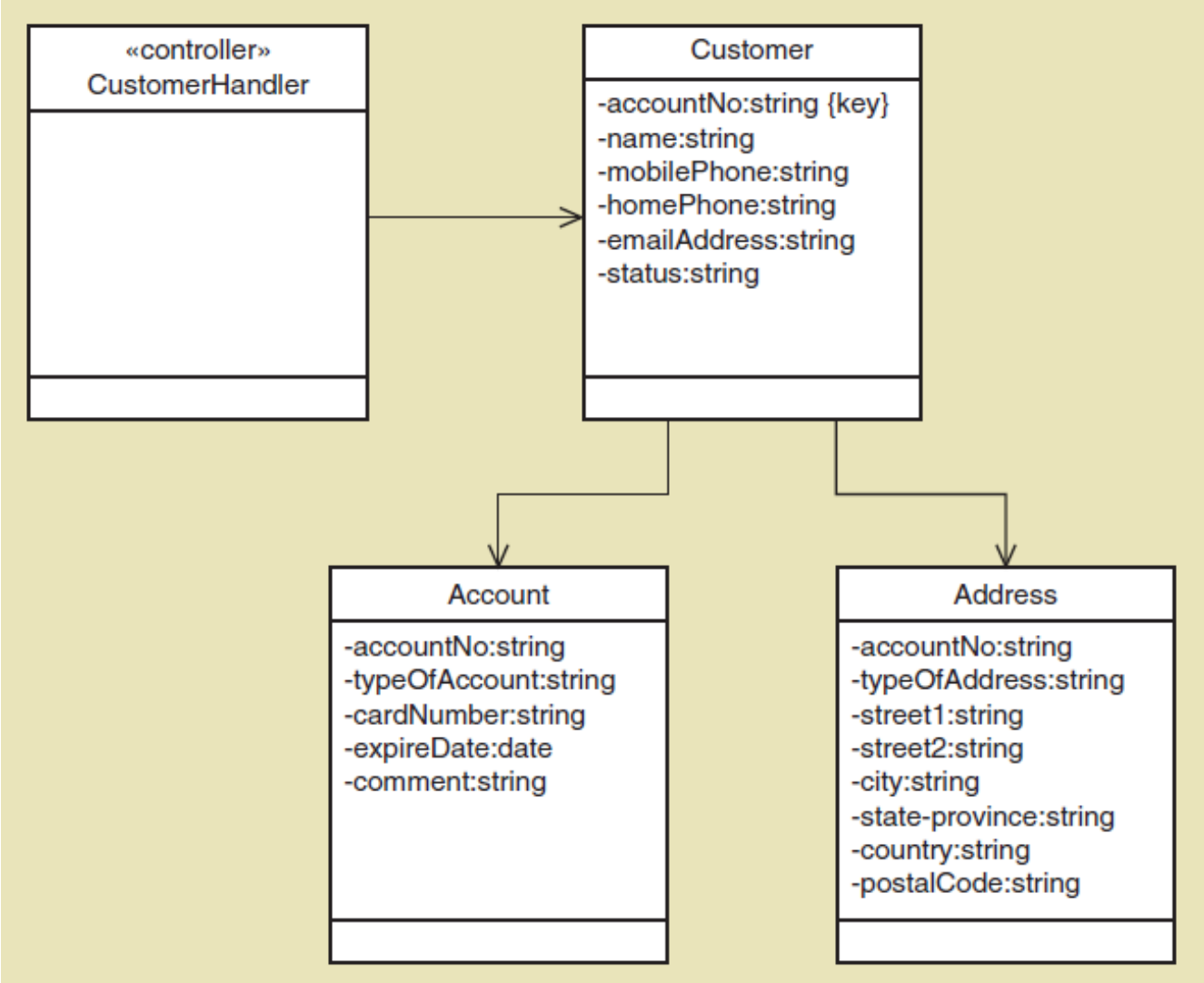
**Start with domain model for Customer Account Subsystem**



# *Create Customer Account* Use Case

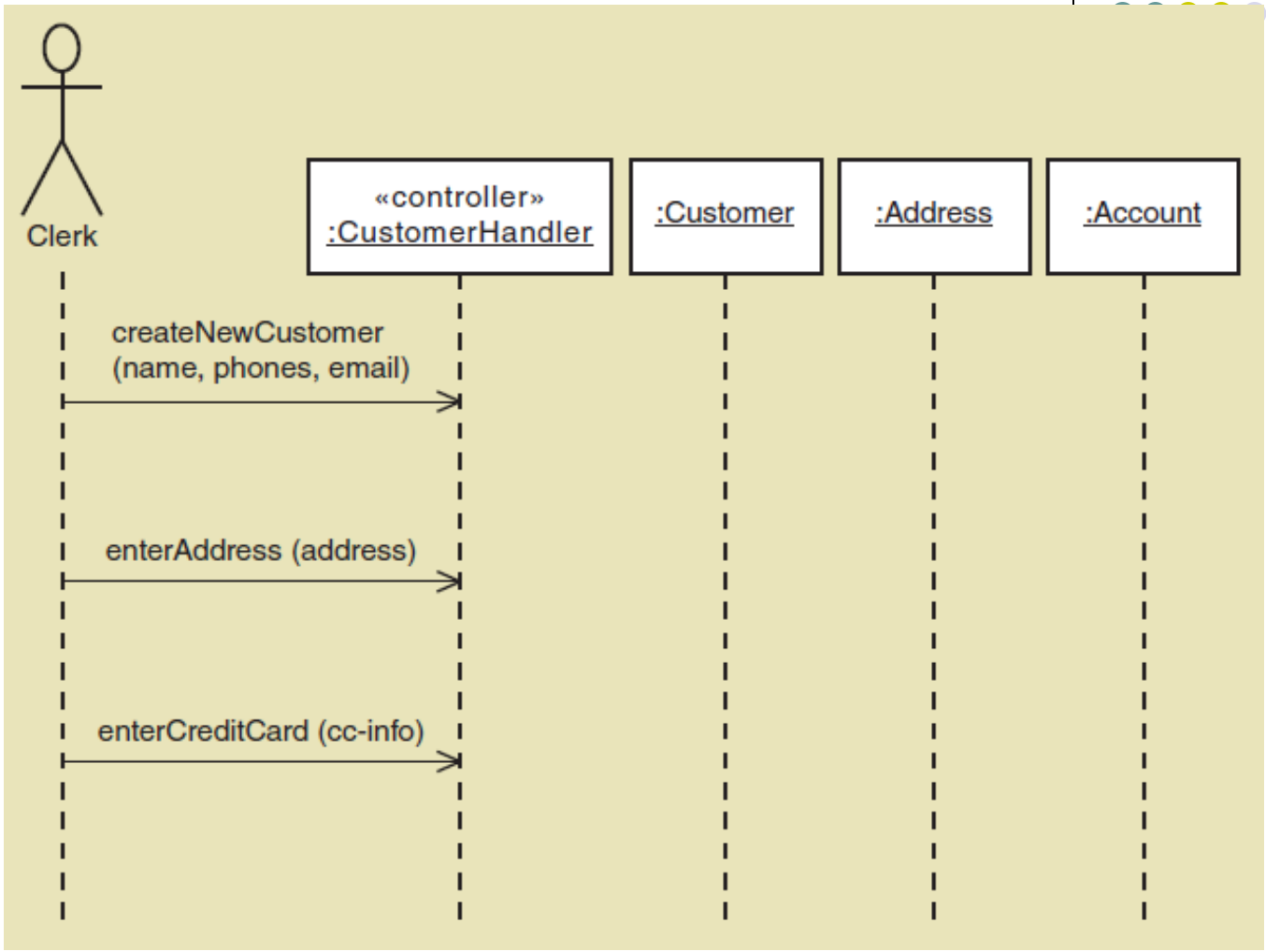
**First cut design class diagram for use case**

**Select needed classes, elaborate attributes, add controller, and add navigation visibility**



# *Create Customer Account* Use Case

**First cut sequence diagram expanding SSD, adding controller, and adding needed classes**

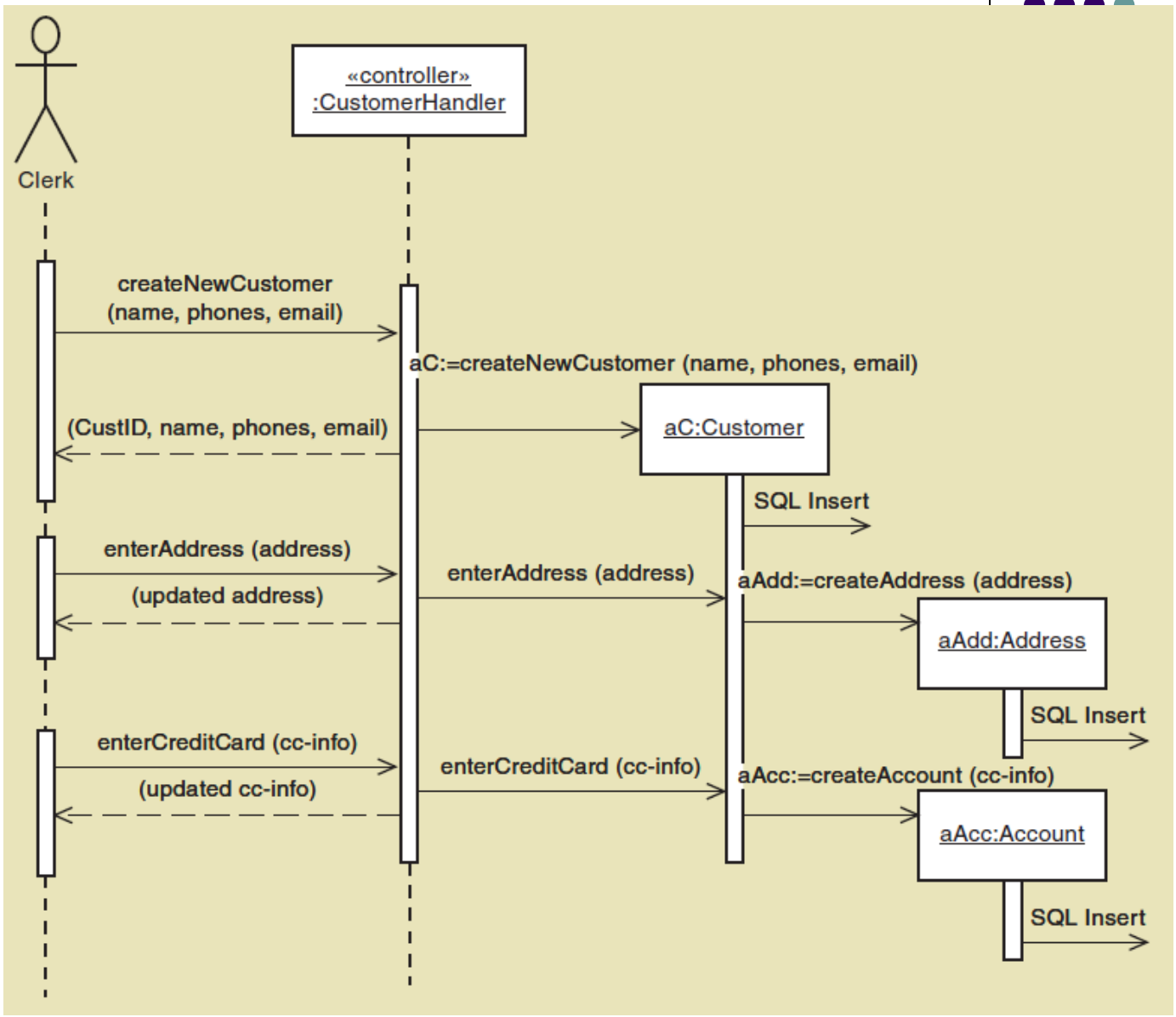


# *Create Customer Account* Use Case

**Add messages and activation to complete collaboration**

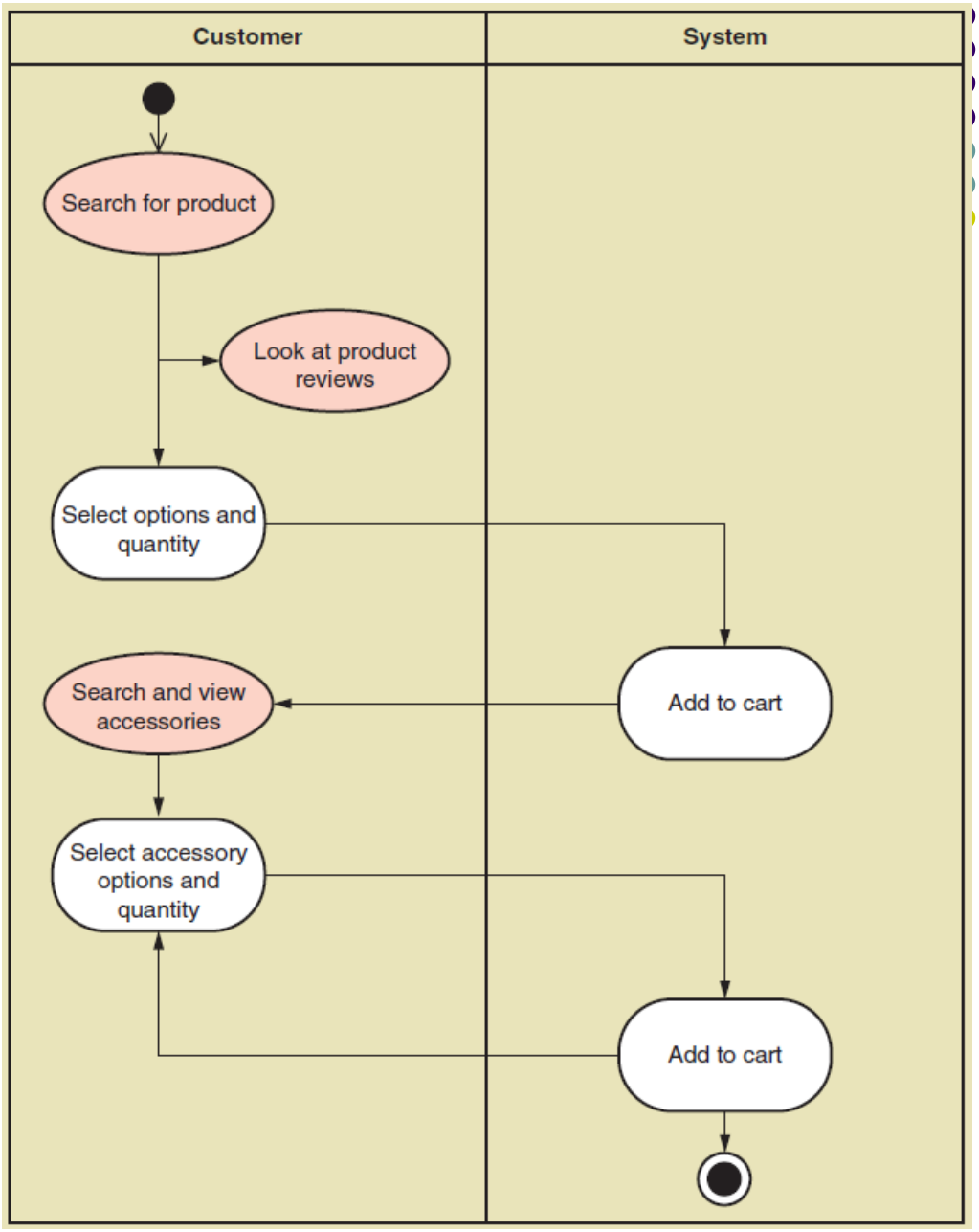
**This is just the domain layer**

**These domain classes handle data access, so this is a two layer architecture**



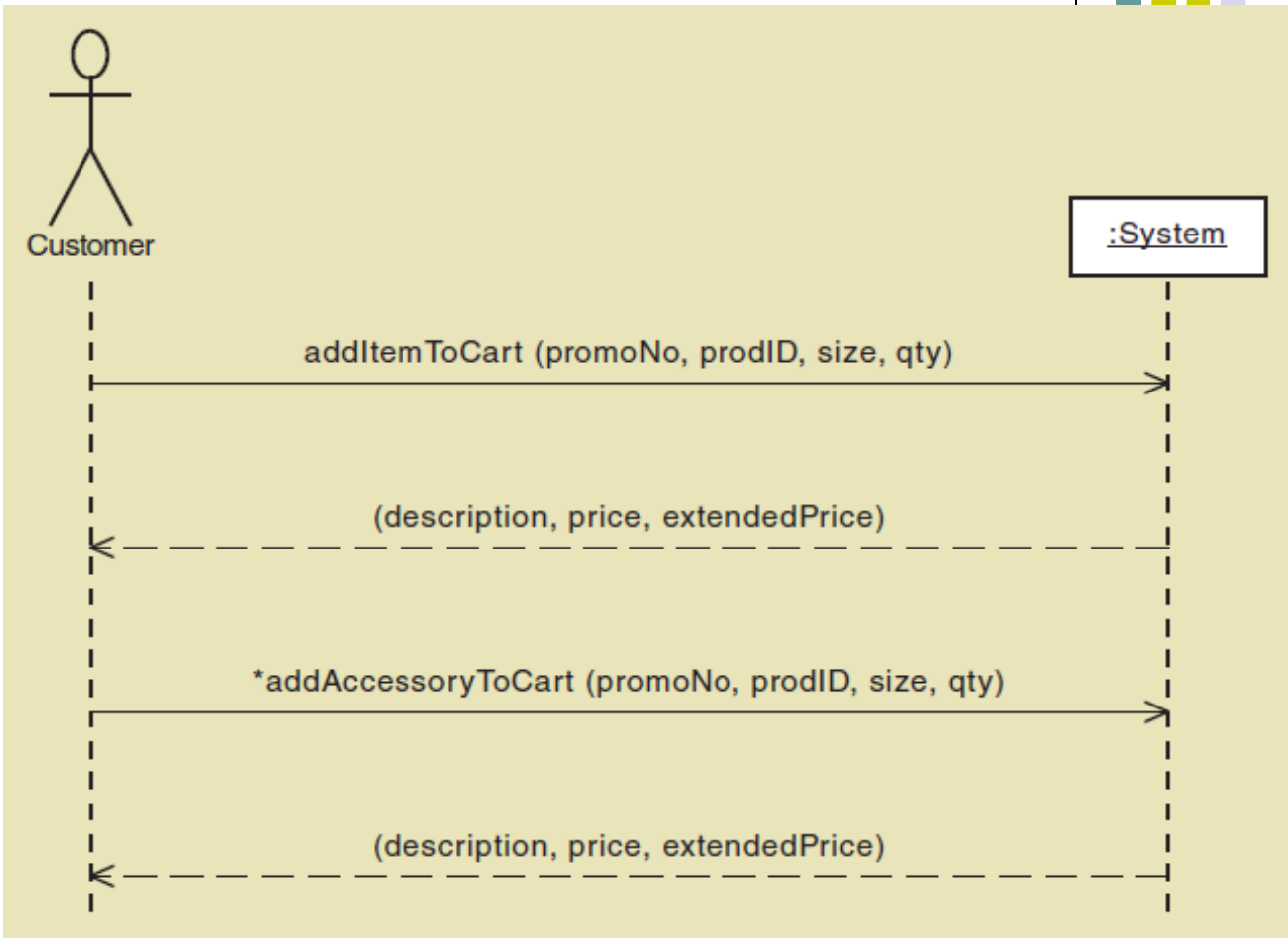
# *Fill Shopping Cart* Use Case

**Start with domain layer based on activity diagram and then SSD**



# *Fill Shopping Cart* Use Case

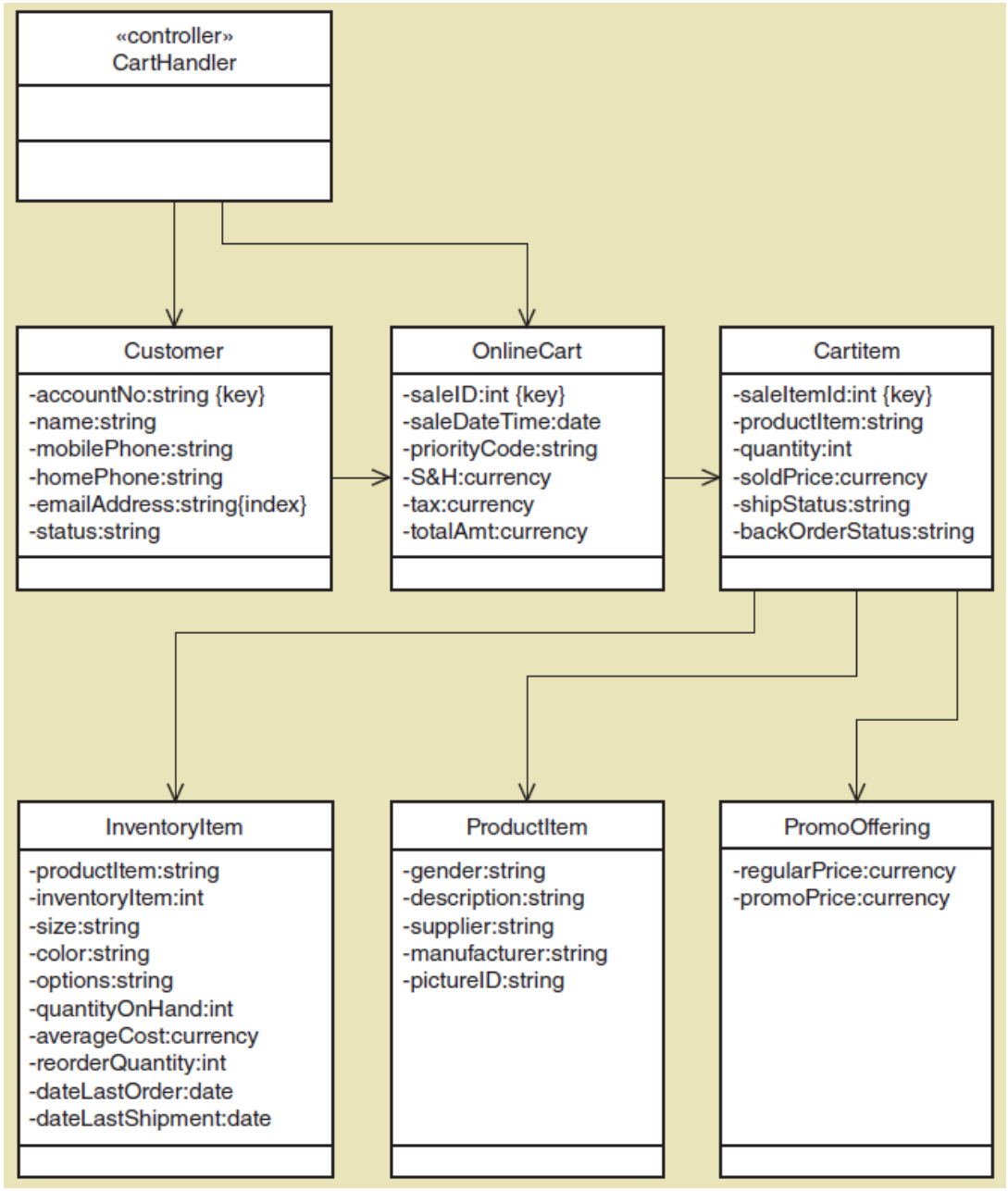
**Start with domain layer based on activity diagram and then SSD**



# *Fill Shopping Cart* Use Case

**First cut design class diagram**

**Select classes, elaborate attributes, add navigation visibility**

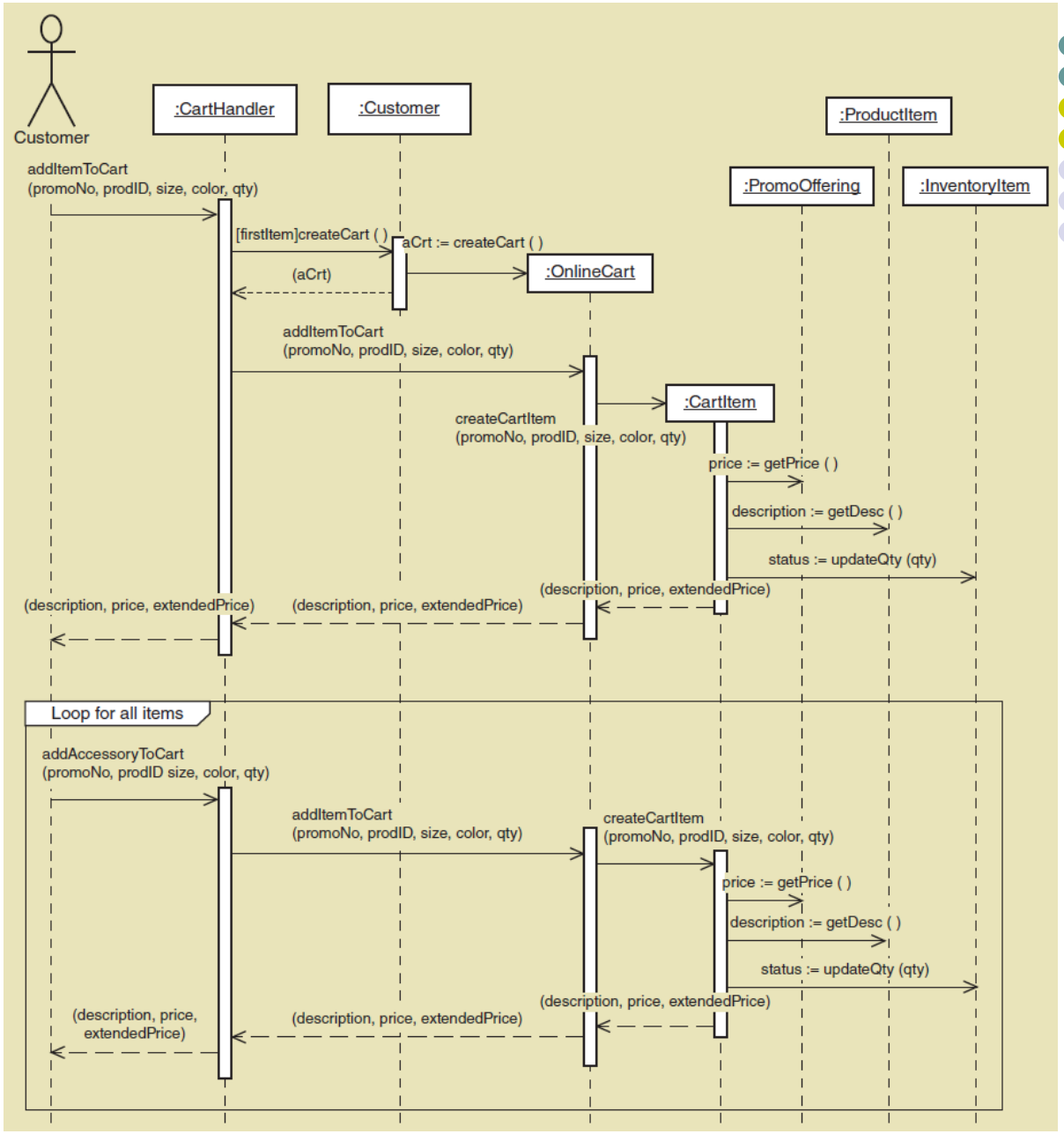


# *Fill Shopping Cart* Use Case

**First cut sequence diagram for domain layer. Will be three layer architecture.**

**:CartHandler is controller, other domain classes are included.**

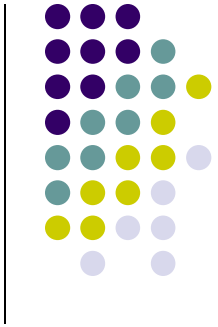
**Note loop frame for adding each item to cart. The SSD used * notation**



# Assumptions

- 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

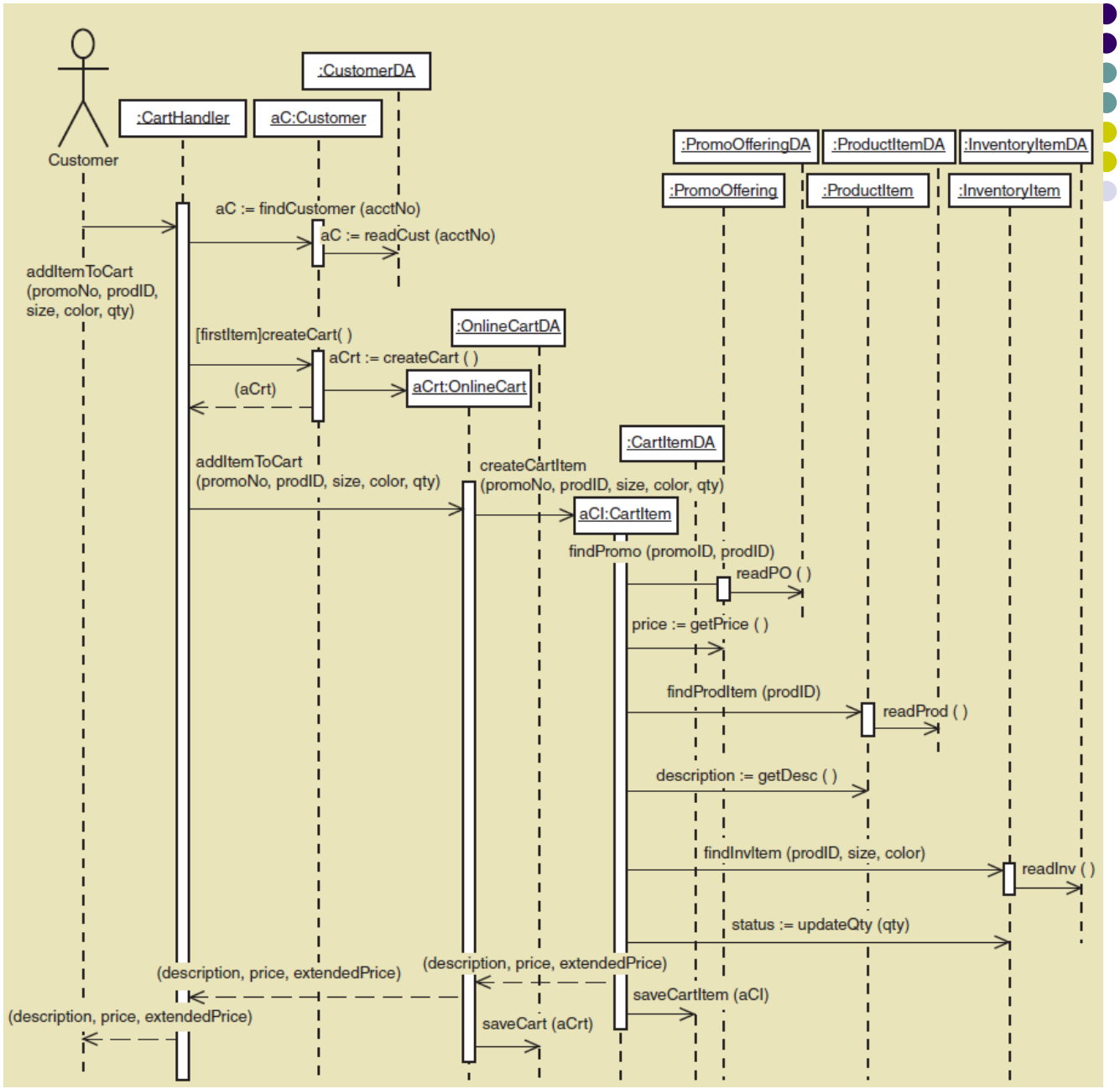
# Developing a Multilayer Design



- View Layer
  - With users, there is always a view layer. Add forms or pages
- Data Access Layer
  - Persistent classes need some mechanism for storing and retrieving their state from a database
  - Separate layer required when the business logic is complex and there is a need to separate connection to database and SQL from the domain classes.
  - *Create customer account* use case might not need a separate layer as domain classes can handle data
  - *Fill shopping cart* is much more complex and does need it
  - For each domain class, add a data access class to get data and save data about an instance

# Adding Data Access Layer

***Fill shopping Cart* use case**



# Data Access Notes

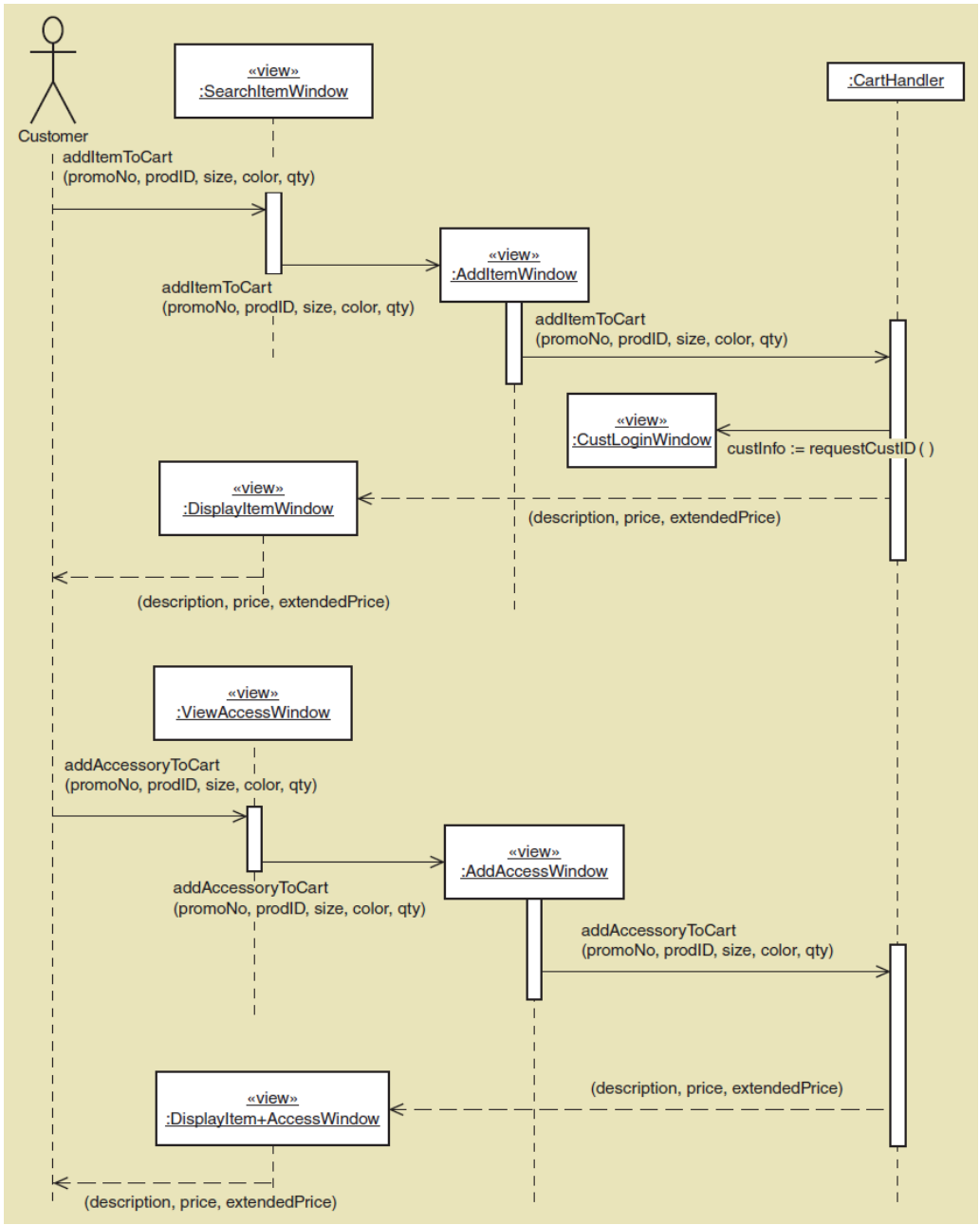
- :CartHandler findCustomer(acctN0) message to Customer class means Customer should create a new instance named aC, send a message to :CustomerDA asking it to read info from the database for a customer with that account number, and then populate the new customer instance with the attribute values from 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

# Adding View Layer

***Fill shopping Cart* use case**

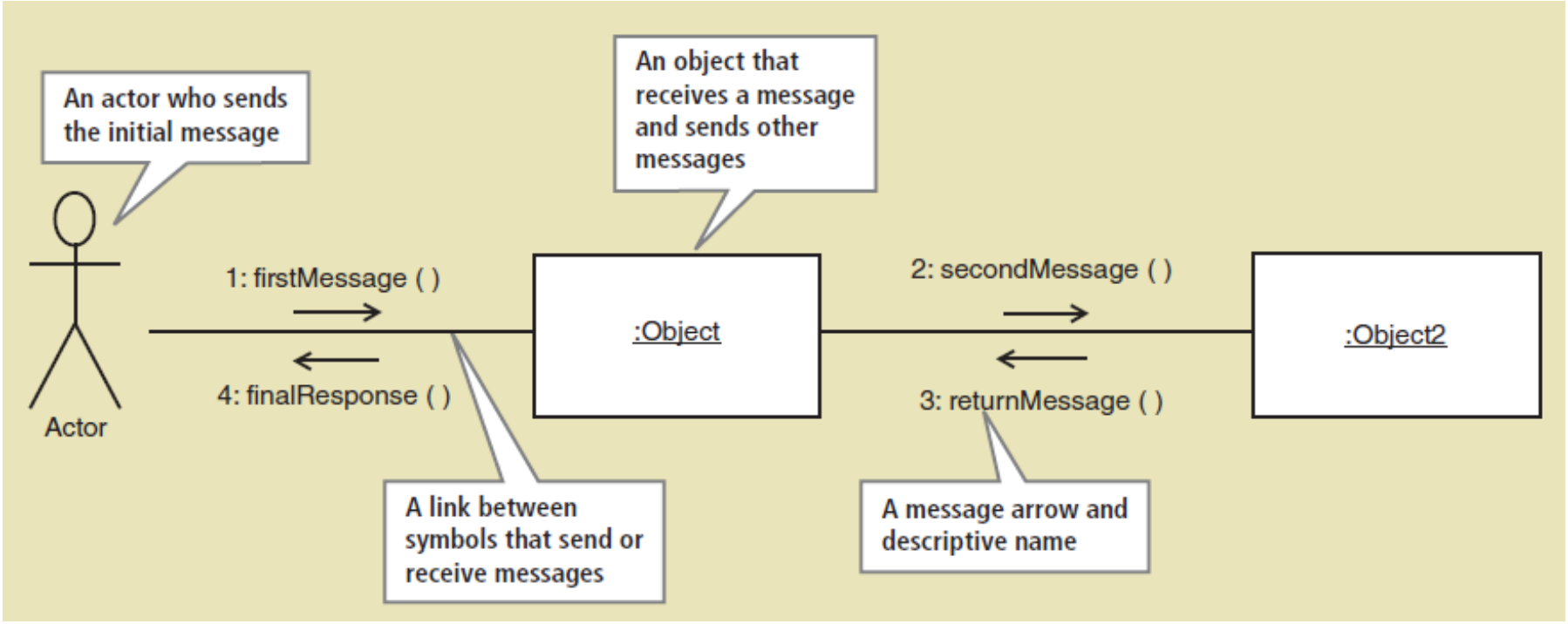
**Sequence diagram just shows View interaction with the controller**

**View classes don't need to even know about the domain or data access layer**



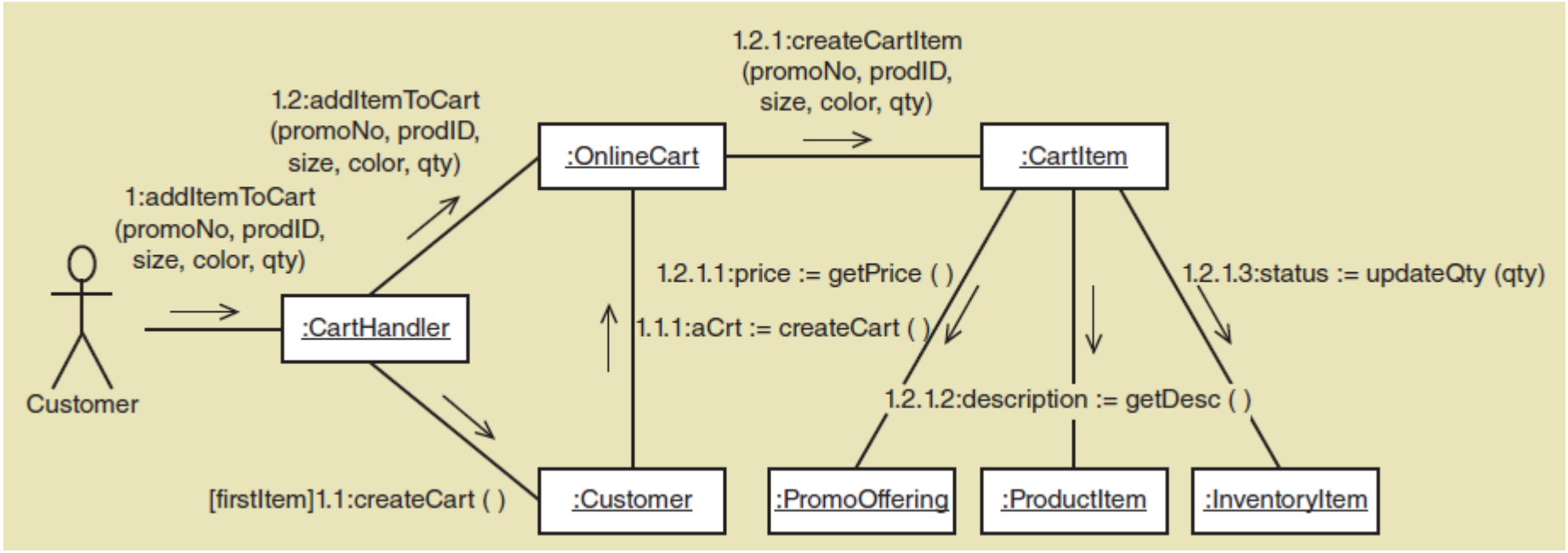
# Designing with UML Communication Diagrams

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



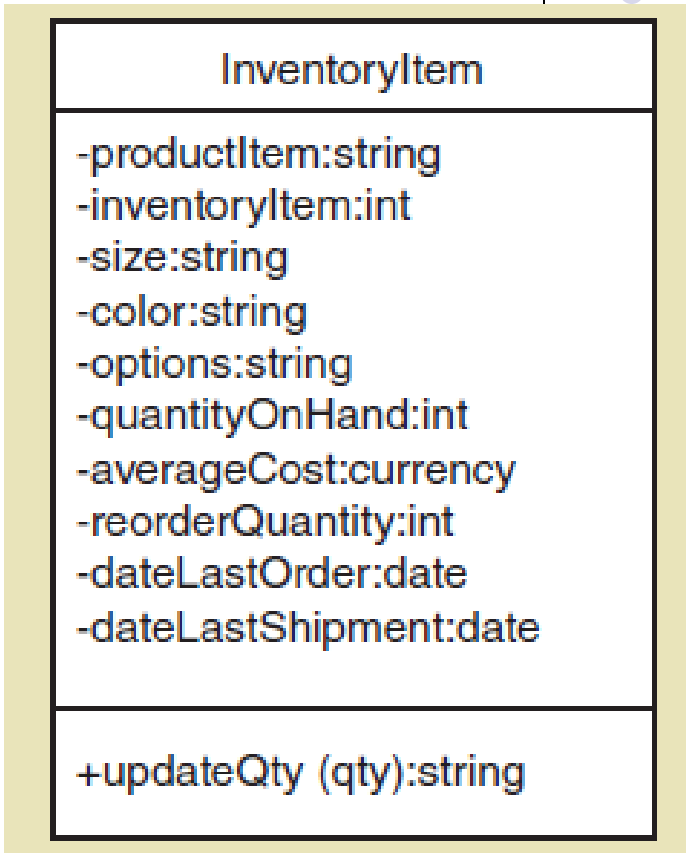
# Communication Diagram for *Fill Shopping Cart* Use Case

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



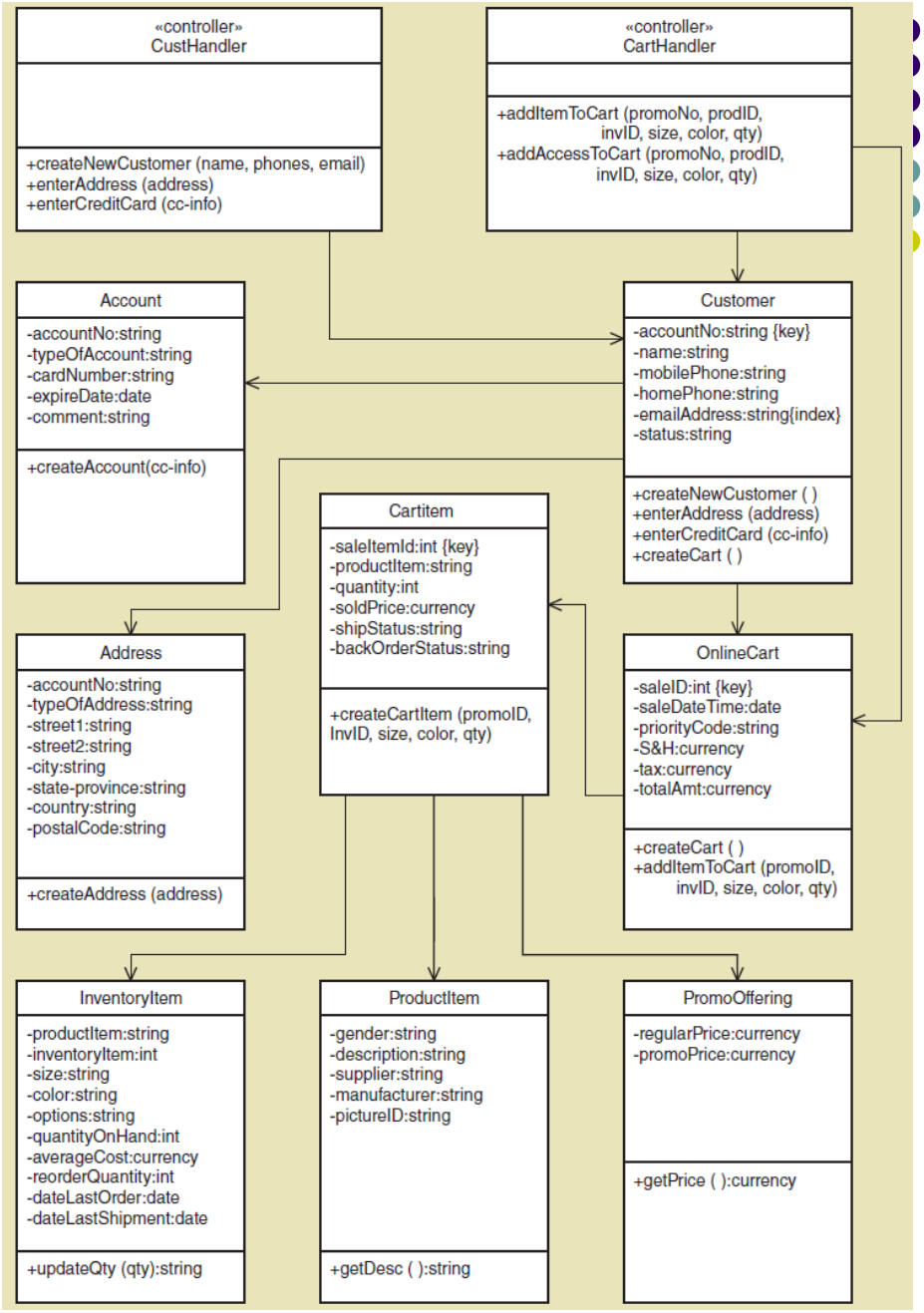
# Updating the Design Classes

- Design class diagram (DCD) focuses on domain layer
- When an object of a class ***receives*** a message, that message ***becomes a method*** in the class on the DCD
- If :InventoryItem receives the message updateQty(qty), then a method signature is added to InventoryItem:



# Updated Design Class Diagram for Domain Layer

**After *Create customer account* use case and *Fill shopping cart* use case**

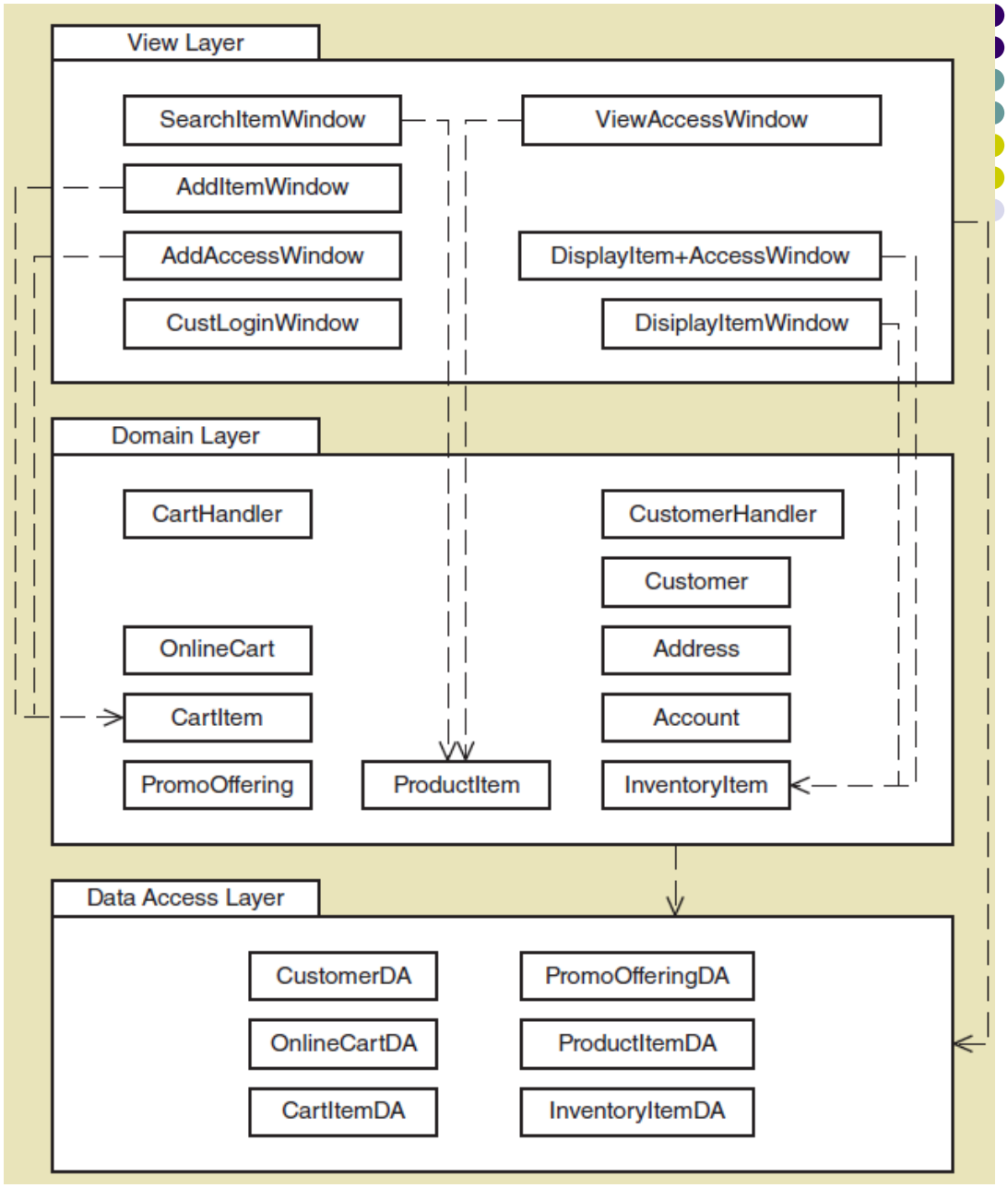


# Structuring Components with UML Package Diagram

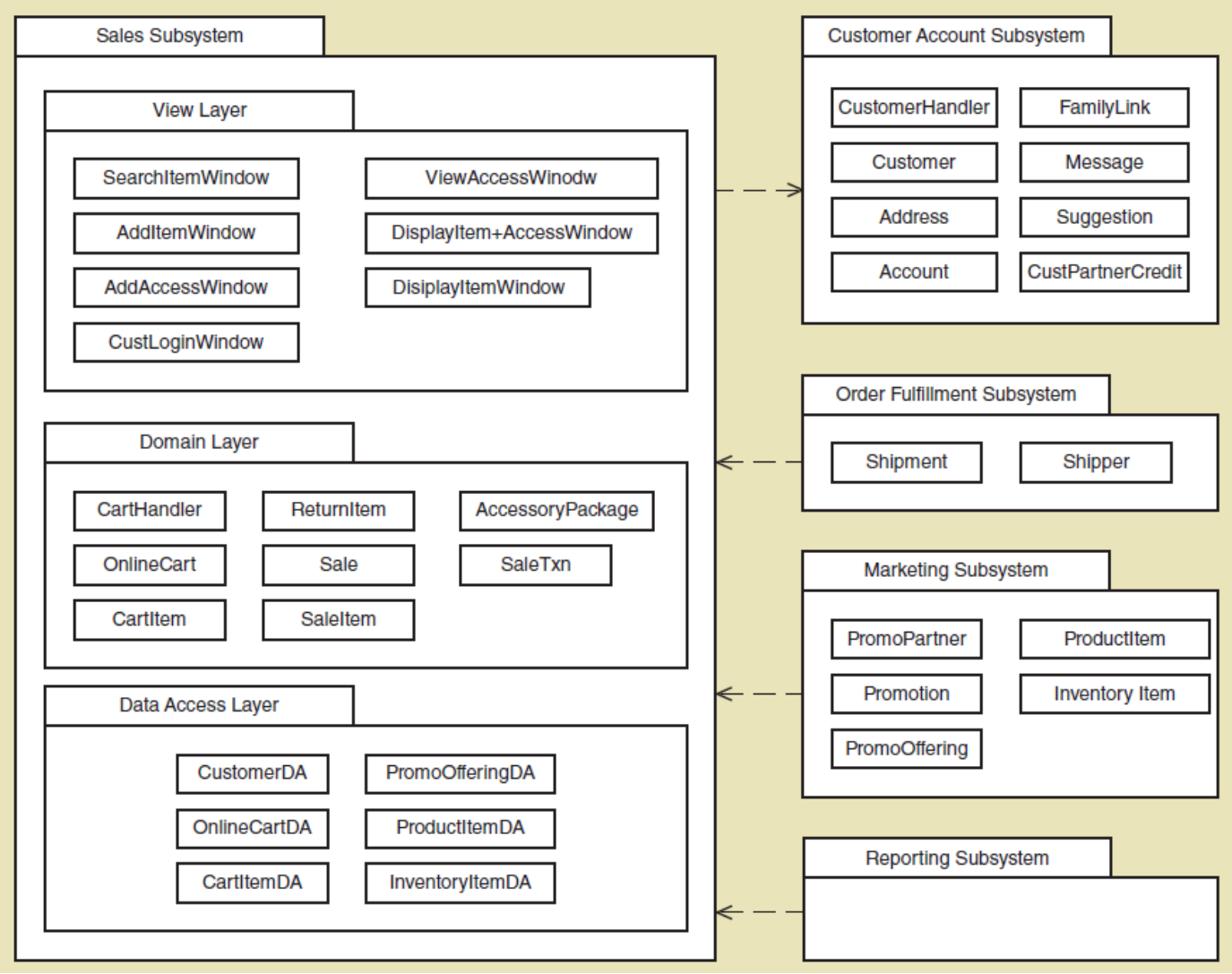
**Three layer package diagram after two RMO use cases**

**Dashed line is dependency**

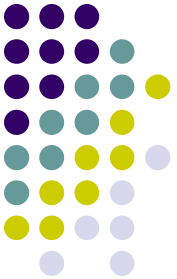
**Note: Java package or .NET namespace**



# RMO Subsystem Package Diagram

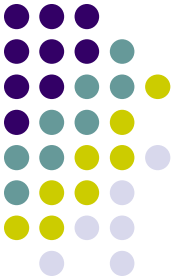


# Implementation Issues
## Three Layer Design



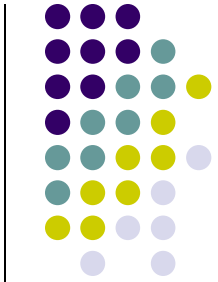
- View Layer Class Responsibilities
  - Display electronic forms and reports.
  - Capture such input events as clicks, rollovers, and key entries.
  - Display data fields.
  - Accept input data.
  - Edit and validate input data.
  - Forward input data to the domain layer classes.
  - Start and shut down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

# Implementation Issues
## Three Layer Design



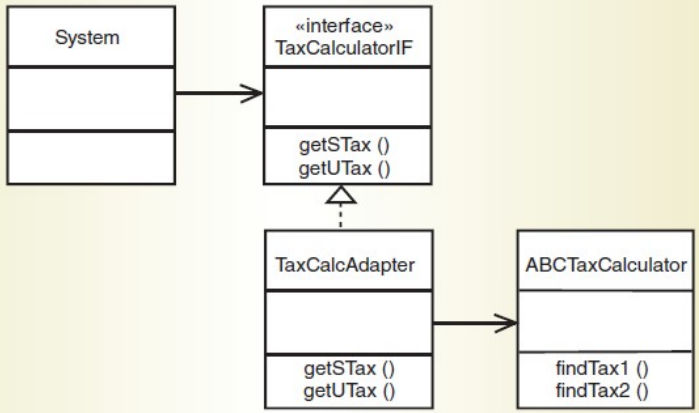
- Domain Layer Class Responsibilities
  - Create problem domain (persistent) classes.
  - Process all business rules with appropriate logic.
  - Prepare persistent classes for storage to the database.
- Data Access Layer Class Responsibilities
  - Establish and maintain connections to the database.
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

# More Design Patterns



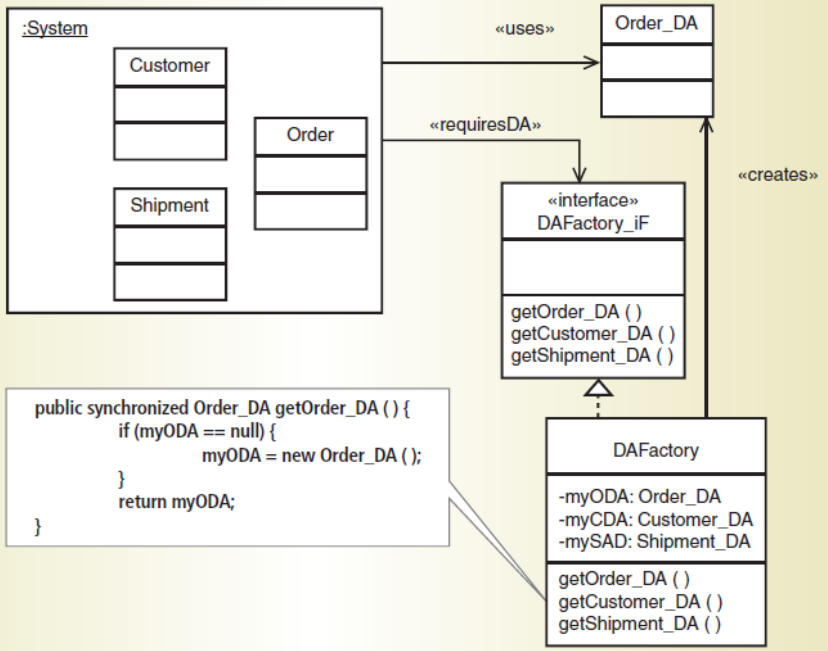
- Adapter
  - Like an electrical adapter
  - Place an adapter class between your system and an external system
- Factory
  - Use factor class when creation logic is complex for a set of classes
- Singleton
  - Use when only one instance should exist at a time and is shared

# 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.<br><br>Two classes are defined, an interface class and the adapter class.<br><br>Passed parameters may add more complexity, and it is difficult to limit changes to the adapter class. |

# Factory Design Pattern

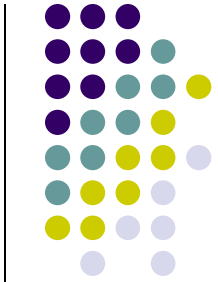
| | |
|---|---|
| **Name:** | Factory or Factory Method |
| **Problem:** | Who should be responsible for creating utility type objects that do not specifically belong to the problem domain classes? These utility objects may also be accessed from various places within the system, so a given object may 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<br>Less coupling between business logic layer and data layer<br>Smaller, more maintainable classes |



# Singleton Design Pattern

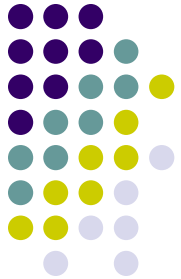
| | |
|---|---|
| **Name:** | Singleton |
| **Problem:** | Only one instantiation of a class is allowed. The instantiation (new) can be called from several places in the system. The first reference should make a new instance, and later attempts should return a reference to the already instantiated object. 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:<br><br>`Class Connection`<br>`{`<br>`private static Connection conn = null;`<br>`public synchronized static getConnection ( )`<br>`{`<br>`if (conn == null) {`<br>`conn = new Connection ( );}`<br>`return conn;`<br>`}`<br>`}`<br><br>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.<br><br>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.<br><br>The code to implement the singleton is very simple, which is one of the desirable characteristics of a good design pattern. |

# Summary



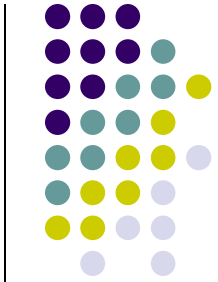
- This chapter went into more detail about use case realization and three layer design to extend the design techniques from last chapter
- Three layer design is an architectural design pattern, part of the movement toward the use of design principles and patterns.
- Use case realization is the design 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 (continued)



- 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 (continued)



- 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