Systems Analysis and Design in a Changing World, Fourth Edition

CHAPTER

11

THE OBJECT-ORIENTED APPROACH TO DESIGN: USE CASE REALIZATION

Learning Objectives

- Explain the purpose and objectives of object-oriented design
- Develop design class diagrams
- Develop interaction diagrams based on the principles of object responsibility and use case controllers

Learning Objectives (continued)

- Develop detailed sequence diagrams as the core process in systems design
- Develop communication diagrams as part of systems design
- Document the architectural design using package diagrams

Overview

- Primary focus of this chapter is how to develop detailed object-oriented design models
- Programmers use models to code the system
- Two most important models are design class diagrams and interaction diagrams (sequence diagrams and communication diagrams)
- Class diagrams are developed for domain, view, and data access layers
- Interaction diagrams extend system sequence diagrams

Object-Oriented Design—The Bridge Between Analysis and Programming

- Bridge between users’ requirements and new system’s programming
- Object-oriented design is process by which detailed object-oriented models are built
- Programmers use design to write code and test new system
- User interface, network, controls, security, and database require design tasks and models

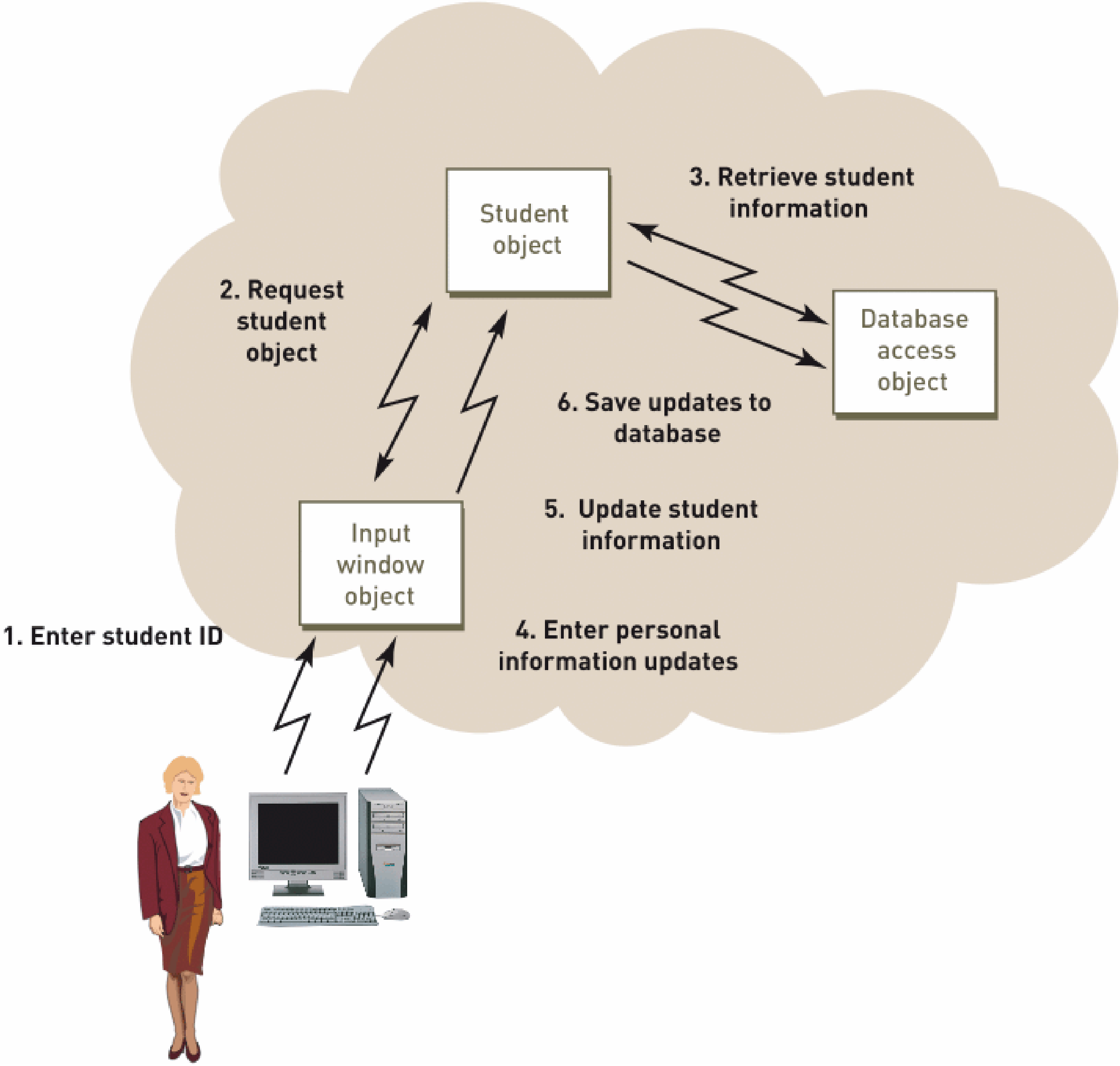
Overview of Object-Oriented Programs

- Set of objects that cooperate to accomplish result
- Object contains program logic and necessary attributes in a single unit
- Objects send each other messages and collaborate to support functions of main program
- OO systems designer provides detail for programmers
    - Design class diagrams, interaction diagrams, and some state machine diagrams

Object-Oriented Three-Layer Program

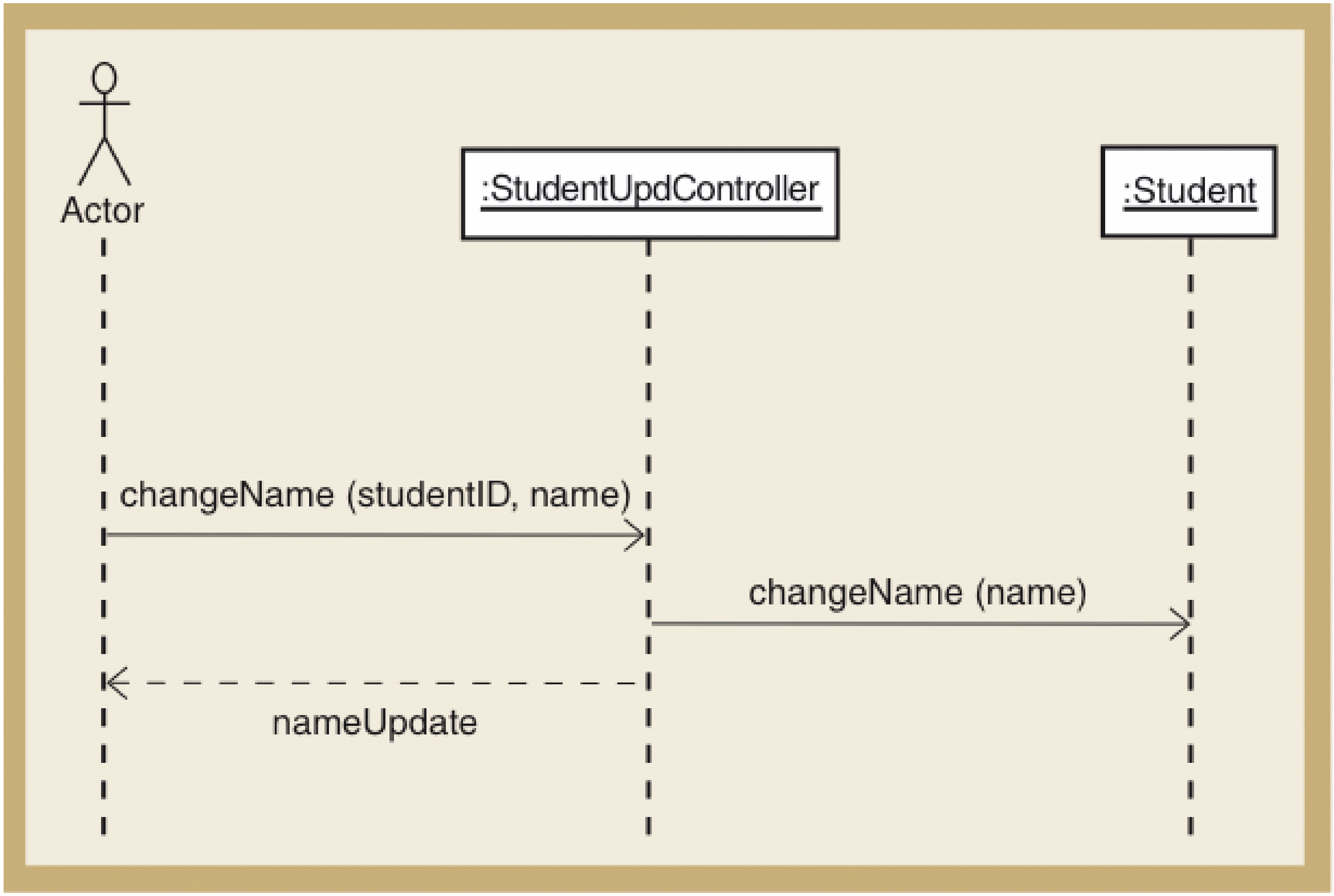
Figure 11-1

Object-oriented event-driven program flow

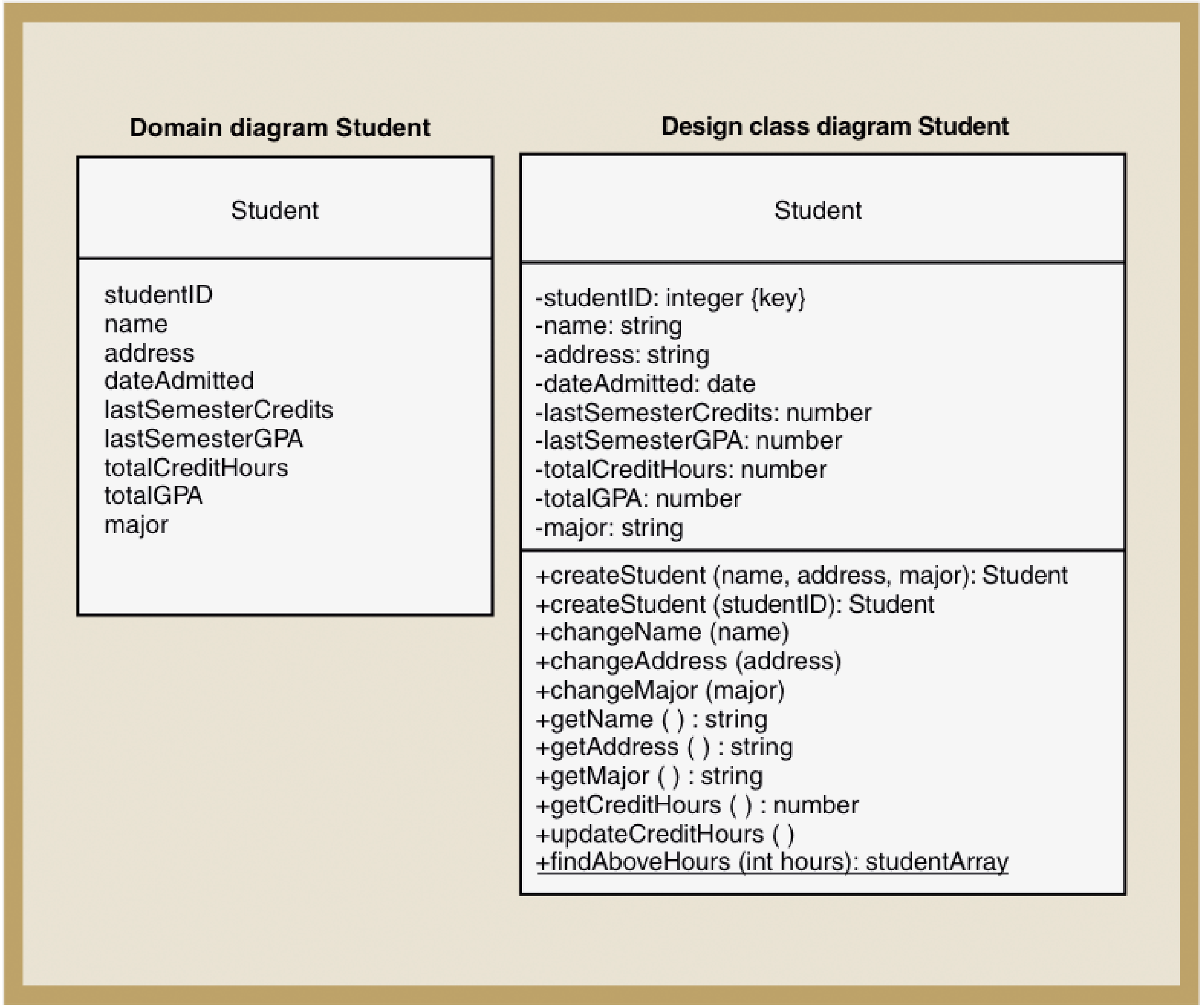

Student object
3. Retrieve student information
2. Request student object
Database access object
6. Save updates to database
5. Update student information
Input window object
1. Enter student ID
4. Enter personal information updates

Sequence Diagram for Updating Student

(Figure 11-2)



Student Class Examples for the Domain Class and the Design Class Diagrams (Figure 11-3)



Domain diagram Student

Student

studentID
name
address
dateAdmitted
lastSemesterCredits
lastSemesterGPA
totalCreditHours
totalGPA
major

Design class diagram Student

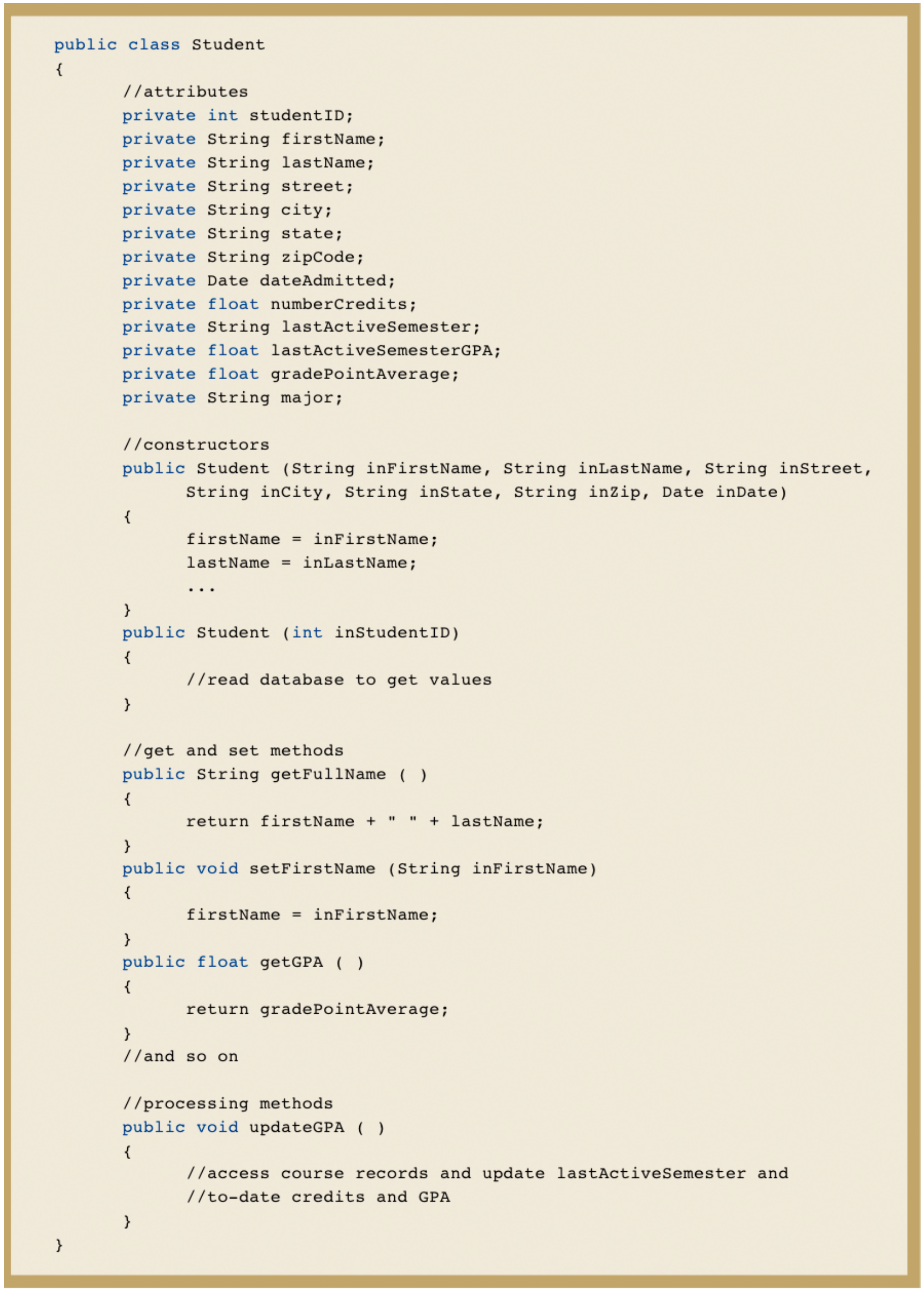
Student

-studentID: integer {key}
-name: string
-address: string
-dateAdmitted: date
-lastSemesterCredits: number
-lastSemesterGPA: number
-totalCreditHours: number
-totalGPA: number
-major: string

+createStudent (name, address, major): Student
+createStudent (studentID): Student
+changeName (name)
+changeAddress (address)
+changeMajor (major)
+getName ( ) : string
+getAddress ( ) : string
+getMajor ( ) : string
+getCreditHours ( ) : number
+updateCreditHours ( )
+findAboveHours (int hours): studentArray

Example Class Definition in Java for Student Class

(Figure 11-4a)

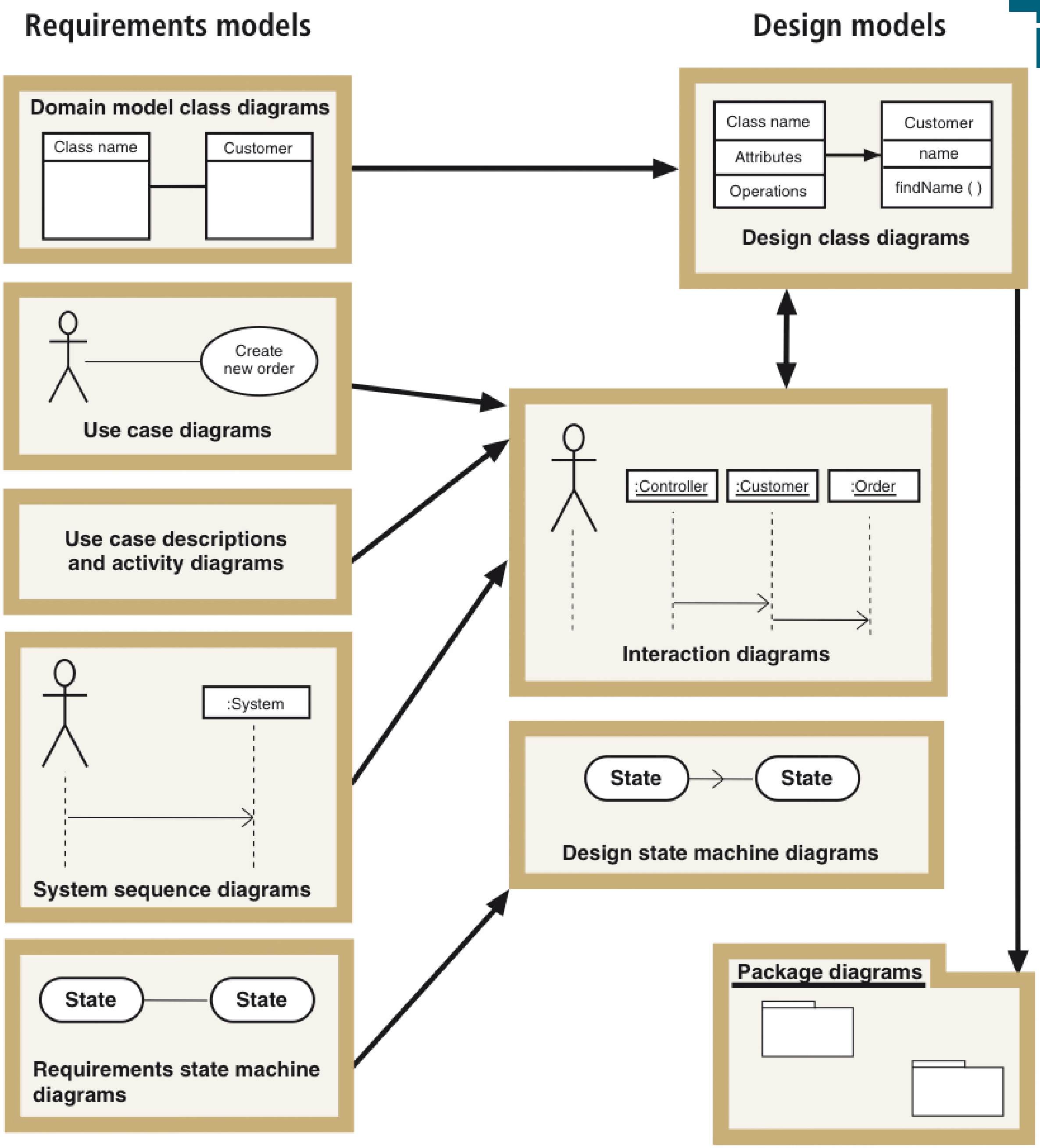


```
public class Student
{
        //attributes
        private int studentID;
        private String firstName;
        private String lastName;
        private String street;
        private String city;
        private String state;
        private String zipCode;
        private Date dateAdmitted;
        private float numberCredits;
        private String lastActiveSemester;
        private float lastActiveSemesterGPA;
        private float gradePointAverage;
        private String major;

        //constructors
        public Student (String inFirstName, String inLastName, String inStreet,
                String inCity, String inState, String inZip, Date inDate)
        {
                firstName = inFirstName;
                lastName = inLastName;
                ...
        }
        public Student (int inStudentID)
        {
                //read database to get values
        }

        //get and set methods
        public String getFullName ( )
        {
                return firstName + " " + lastName;
        }
        public void setFirstName (String inFirstName)
        {
                firstName = inFirstName;
        }
        public float getGPA ( )
        {
                return gradePointAverage;
        }
        //and so on

        //processing methods
        public void updateGPA ( )
        {
                //access course records and update lastActiveSemester and
                //to-date credits and GPA
        }
}
```

Object-Oriented Design Processes and Models

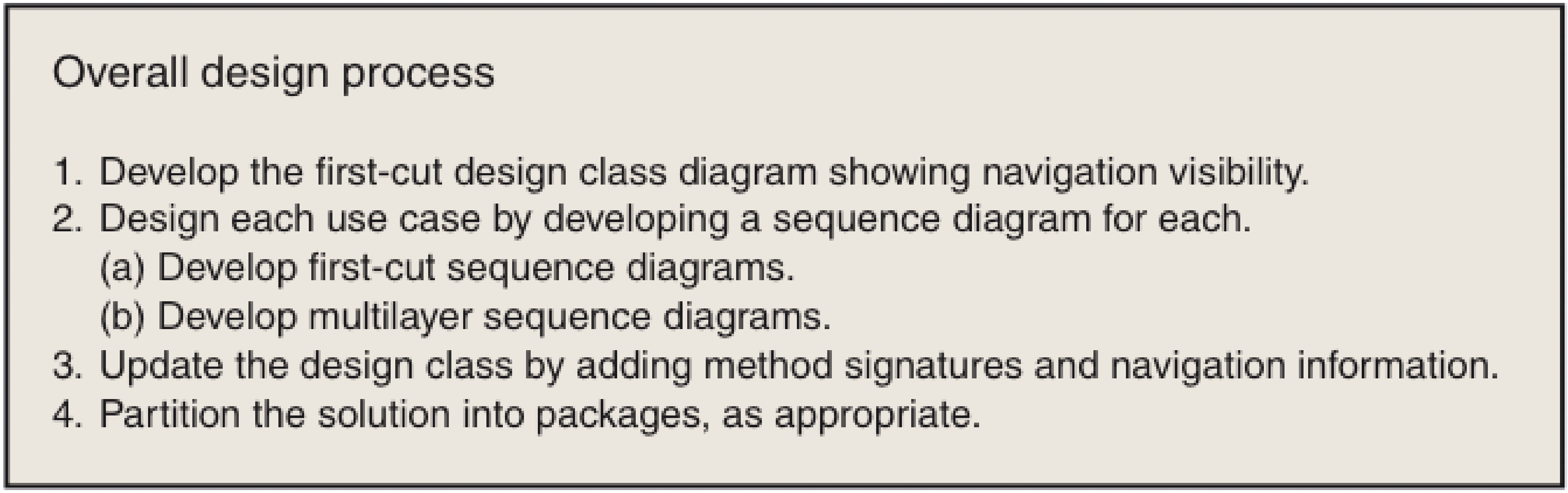
- Diagrams developed for analysis/requirements
  - Use case diagrams, use case descriptions and activity diagrams, domain model class diagrams, and system sequence diagrams
- Diagrams developed for design
  - Interaction diagrams and package diagrams
  - Design class diagrams – include object-oriented classes, navigation between classes, attribute names, method names, and properties needed for programming

Design Models with Their Respective Input Models (Figure 11-5)



Iterative Process of OO Design—Design Steps (Figure 11-6)

Realization of use case – specialization of all detailed system processing for each use case



Overall design process

1. Develop the first-cut design class diagram showing navigation visibility.
2. Design each use case by developing a sequence diagram for each.
   (a) Develop first-cut sequence diagrams.
   (b) Develop multilayer sequence diagrams.
3. Update the design class by adding method signatures and navigation information.
4. Partition the solution into packages, as appropriate.

Design Classes, Interaction, and Design Process

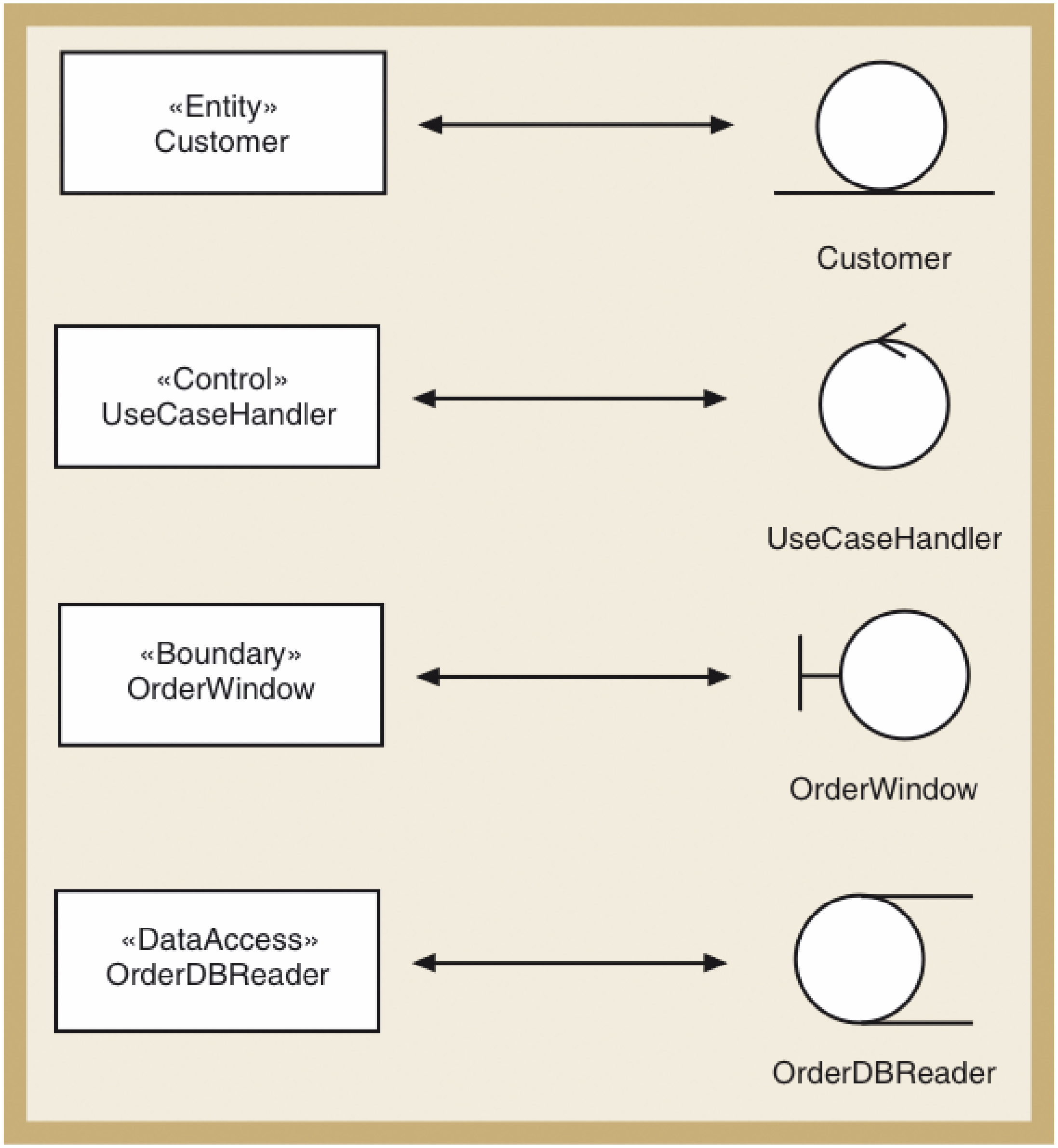
- Design class diagrams and detailed interaction diagrams
    - Use each other as inputs and are developed in parallel
- First-cut design class diagram is based on domain model and system design principles
- First-cut sequence diagram for use case is extended from system sequence diagram (SSD)
    - Shows interacting objects
- Sequence diagram is completed layer by layer
    - Problem domain, data access, and view layers
- Design class diagram is updated based on sequence diagram

Design Class Symbols

- UML does not distinguish between design class notation and domain model notation
- Domain model class diagram shows conceptual classes in users’ work environment
- Design class diagram specifically defines software classes
- UML uses stereotype notation to categorize a model element by its characteristics

Standard Stereotypes Found in Design Models

(Figure 11-7)

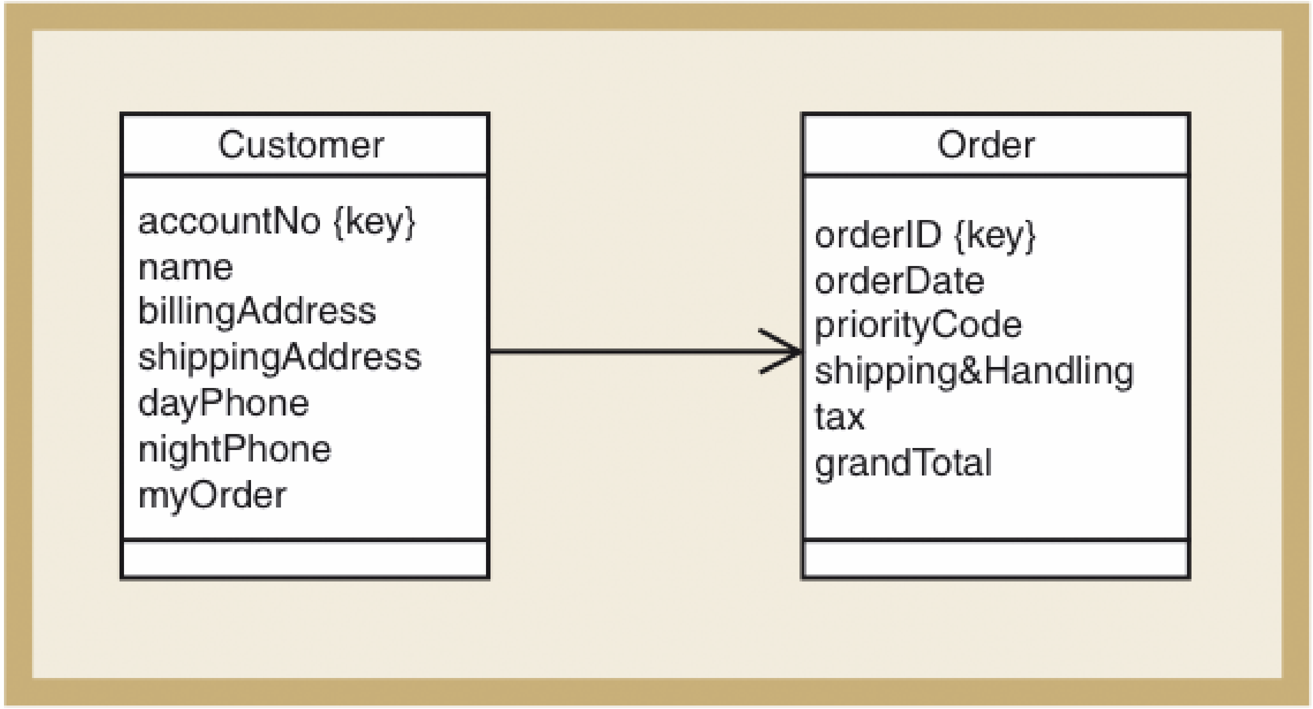


Standard Design Classes

- Entity – design identifier for problem domain class
  - Persistent class – exists after system is shut down
- Control – mediates between boundary and entity classes, between the view layer and domain layer
- Boundary – designed to live on system’s automation boundary, touched by users
  - User interface and windows classes
- Data access – retrieves data from and sends data to database

Navigation Visibility

- A design principle in which one object has reference to another object
  - Can interact with other object by sending messages

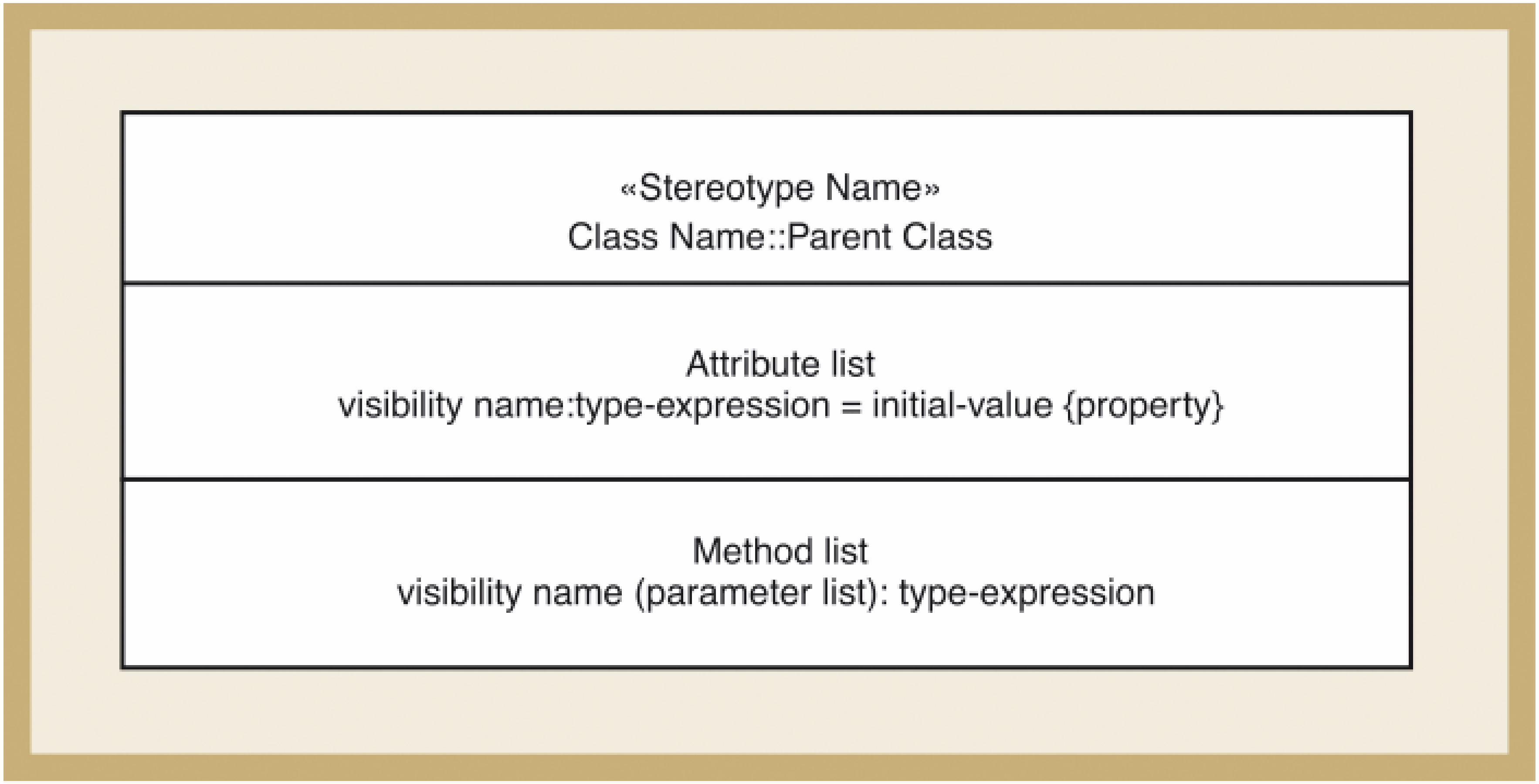


Design Class Notation

- Name – class name and stereotype information
- Attribute visibility (private or public) – attribute name, type-expression, initial-value, property
- Method signature – information needed to invoke (or call) the method
  - Method visibility, method name, type-expression (return parameter), method parameter list (incoming arguments)
  - Overloaded method – method with same name but two or more different parameter lists
  - Class-level method – method associated with class instead of each object (static or shared method), denoted by an underline

Notation Used to Define a Design Class

(Figure 11-8)

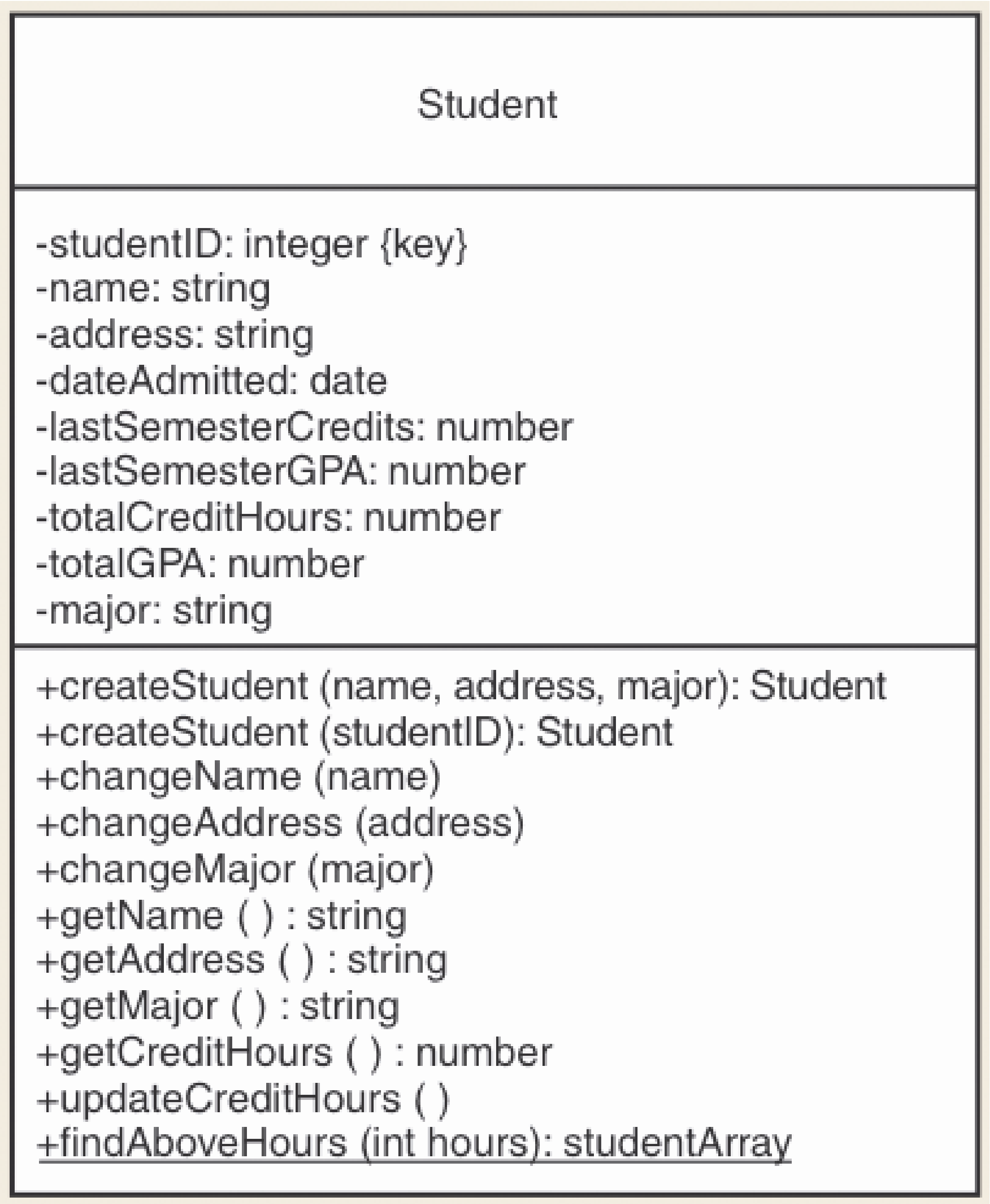


«Stereotype Name»
Class Name::Parent Class

Attribute list
visibility name:type-expression = initial-value {property}

Method list
visibility name (parameter list): type-expression

Student Design Class Example



Student

-studentID: integer {key}
-name: string
-address: string
-dateAdmitted: date
-lastSemesterCredits: number
-lastSemesterGPA: number
-totalCreditHours: number
-totalGPA: number
-major: string

+createStudent (name, address, major): Student
+createStudent (studentID): Student
+changeName (name)
+changeAddress (address)
+changeMajor (major)
+getName ( ) : string
+getAddress ( ) : string
+getMajor ( ) : string
+getCreditHours ( ) : number
+updateCreditHours ( )
+findAboveHours (int hours): studentArray

Developing the First-Cut Design Class Diagram

- Extend domain model class diagram
  - Elaborate attributes with type and initial value information
- Detailed design proceeds use case-by-use case
  - Interaction diagrams implement navigation
  - Navigation arrows are updated to be consistent
  - Method signatures are added to each class

Developing First-Cut Design Class Diagram

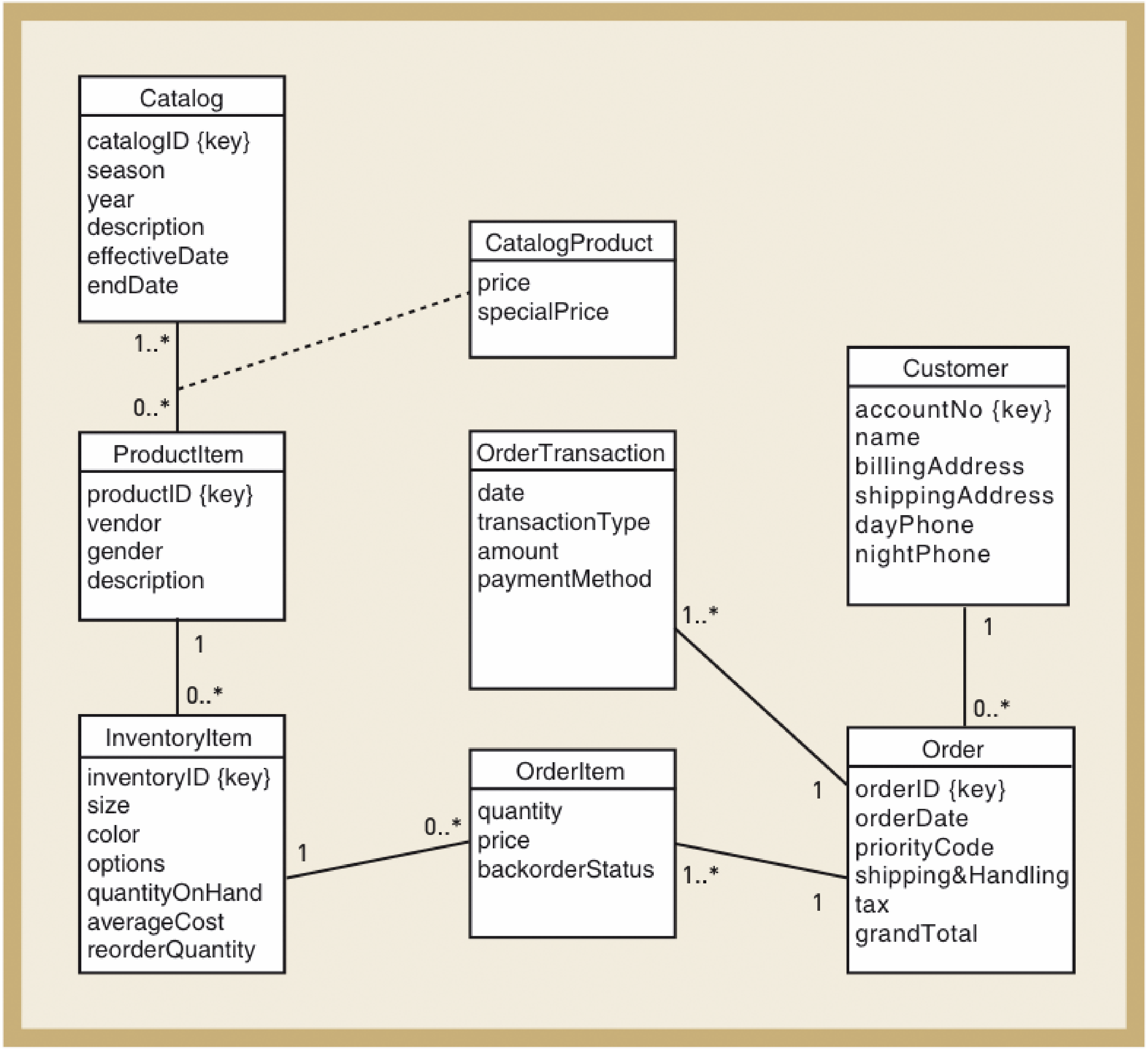
(Continued)

- Choose classes involved with the use case
- Add use case controller
- Elaborate attributes
  - Visibility, type-expression, initial-value, property
- Establish first-cut navigation visibility
  - One-to-many relationships usually navigated from superior to subordinate
  - Mandatory relationships usually navigated from independent to dependent
  - When an object needs information from another object, navigation arrow points to the object itself or to its parent in hierarchy
  - Navigation can be in both directions (arrows bidirectional)

Start with Domain Model Class Diagram

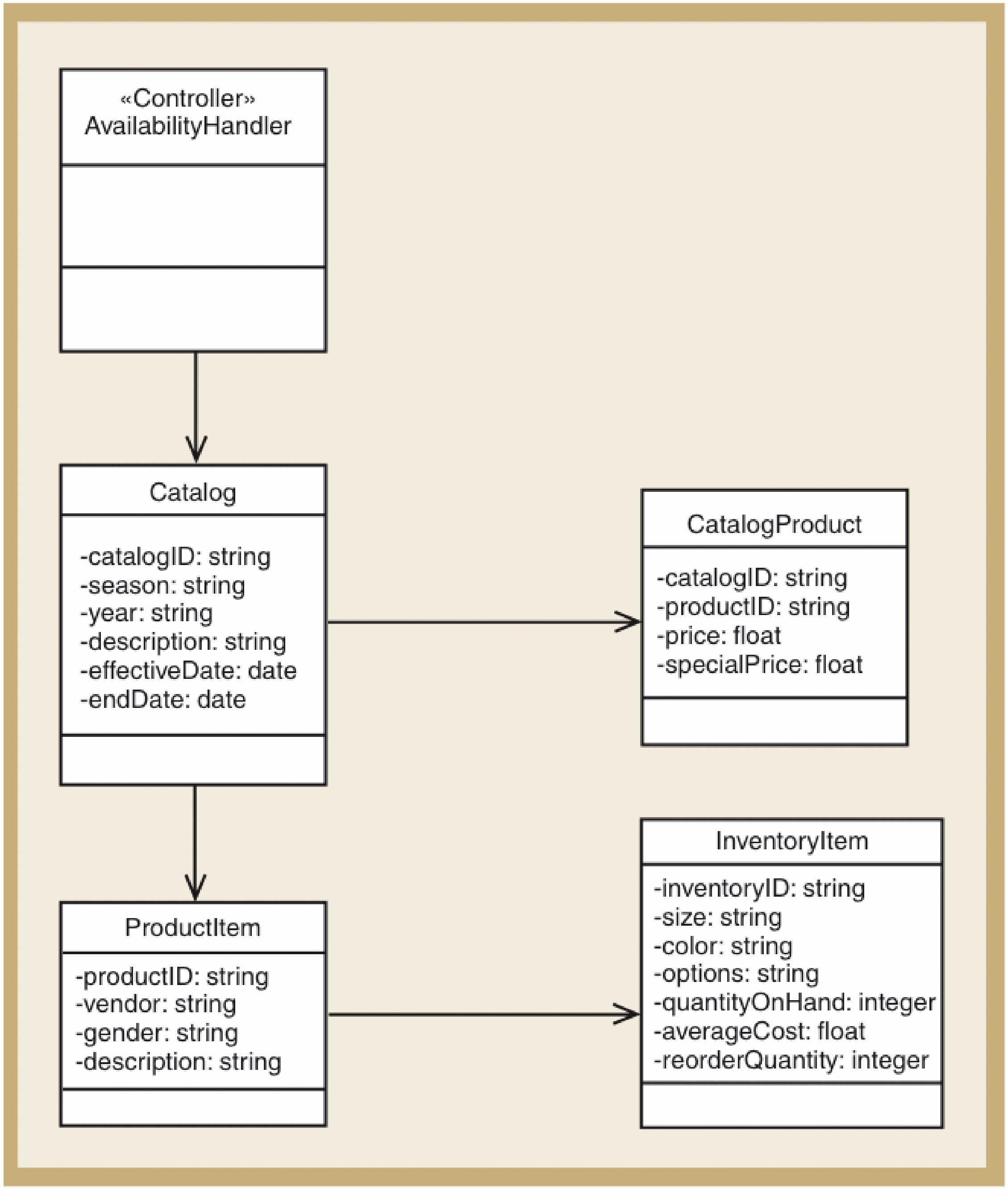
Figure 11-9

Partial RMO domain model class diagram



First-Cut RMO Design Class Diagram for *Look Up Item Availability* Use Case

(Figure 11-11)



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

Some Fundamental Design Principles

- Encapsulation – each object is self-contained unit that includes data and methods that access data
- Object reuse – designers often reuse same classes for windows components
- Information hiding – data associated with object is not visible to outside world
- Protection from variations – parts of a system that are unlikely to change are segregated from those that will
- Indirection – an intermediate class is placed between two classes to decouple them but still link them

Some Fundamental Design Principles

(Continued)

- Coupling – qualitative measure of how closely classes in a design class diagram are linked
  - Number of navigation arrows in design class diagram or messages in a sequence diagram
  - Loosely coupled – system is easier to understand and maintain
- Cohesion – qualitative measure of consistency of functions within a single class
  - Separation of responsibility – divide low cohesive class into several highly cohesive classes
  - Highly cohesive – system is easier to understand and maintain and reuse is more likely

Realizing Use Cases and Defining Methods —Designing 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

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

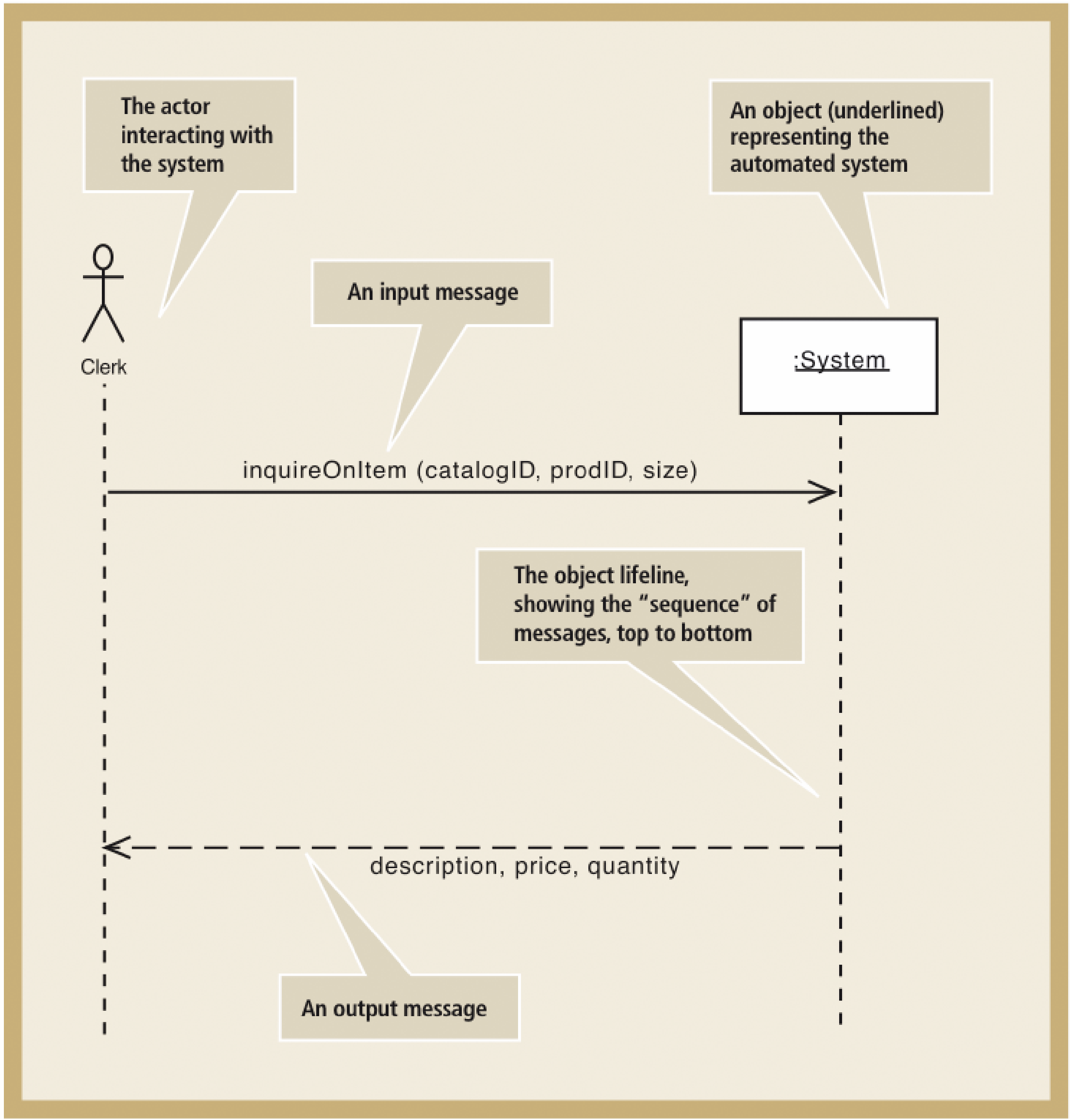
Designing with Sequence Diagrams

- 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

Annotated System Sequence Diagram (SSD) for the *Look Up Item Availability* Use Case (from Chapter 7)

Figure 11-12

SSD for the *Look up item availability* use case


The actor interacting with the system
An object (underlined) representing the automated system
An input message
Clerk
:System
inquireOnItem (catalogID, prodID, size)
The object lifeline, showing the "sequence" of messages, top to bottom
description, price, quantity
An output message

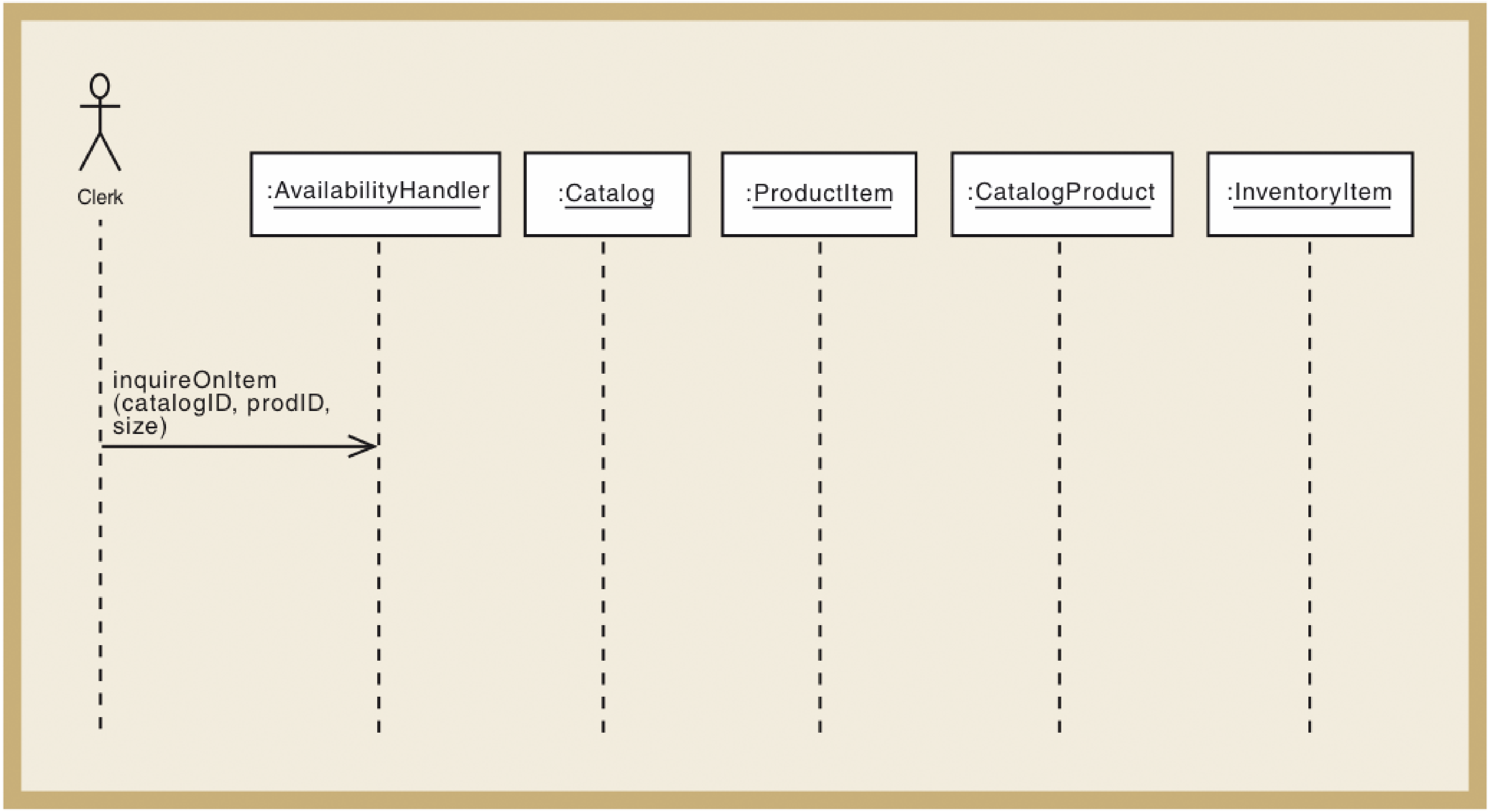
First-Cut Sequence Diagram

- Start with elements from SSD
- Replace :System 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?

Objects included in *Look Up Item Availability*

Figure 11-13

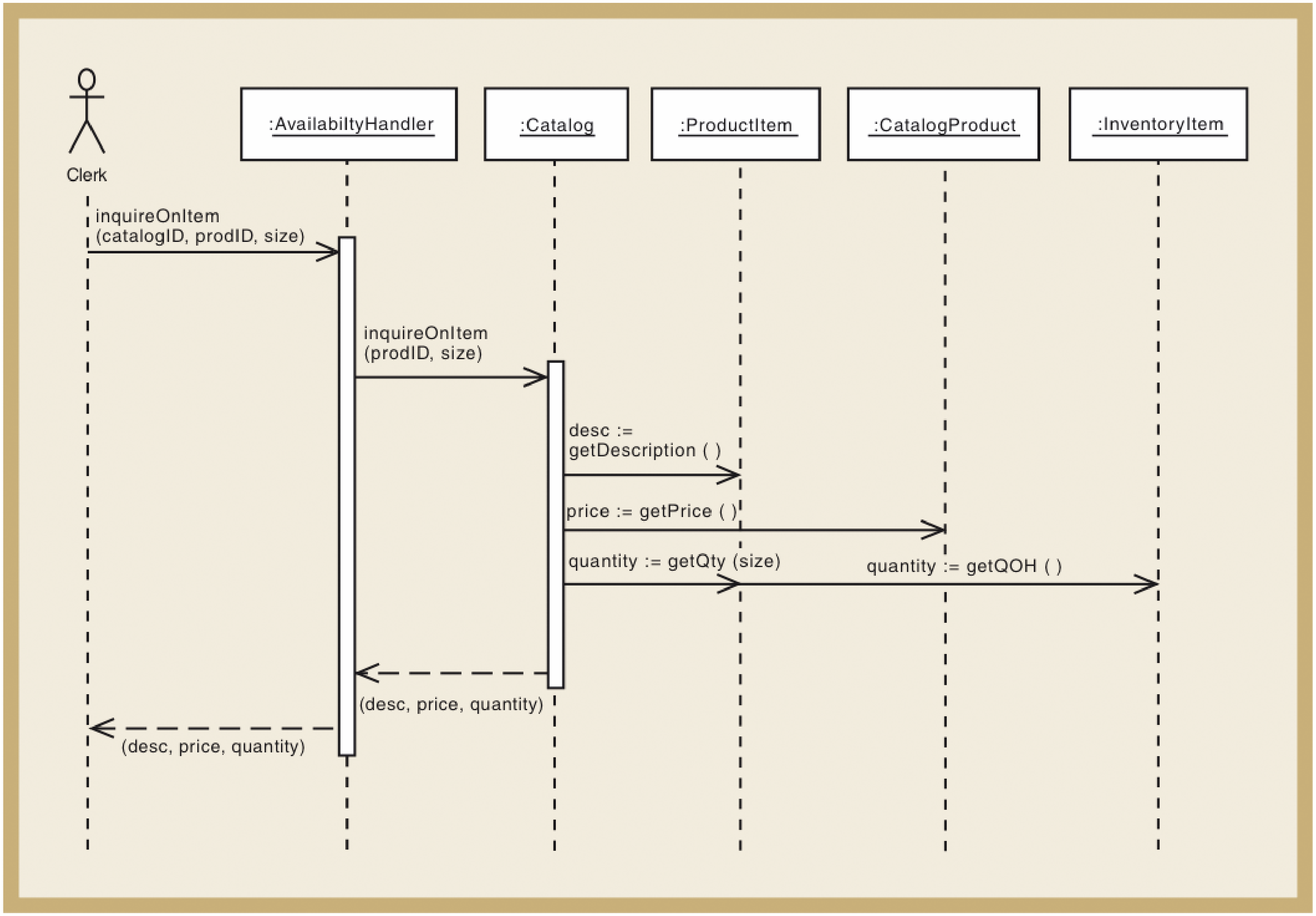
Objects included in *Look up item availability*



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

First-Cut Sequence Diagram for the *Look Up Item Availability* Use Case (Figure 11-14)



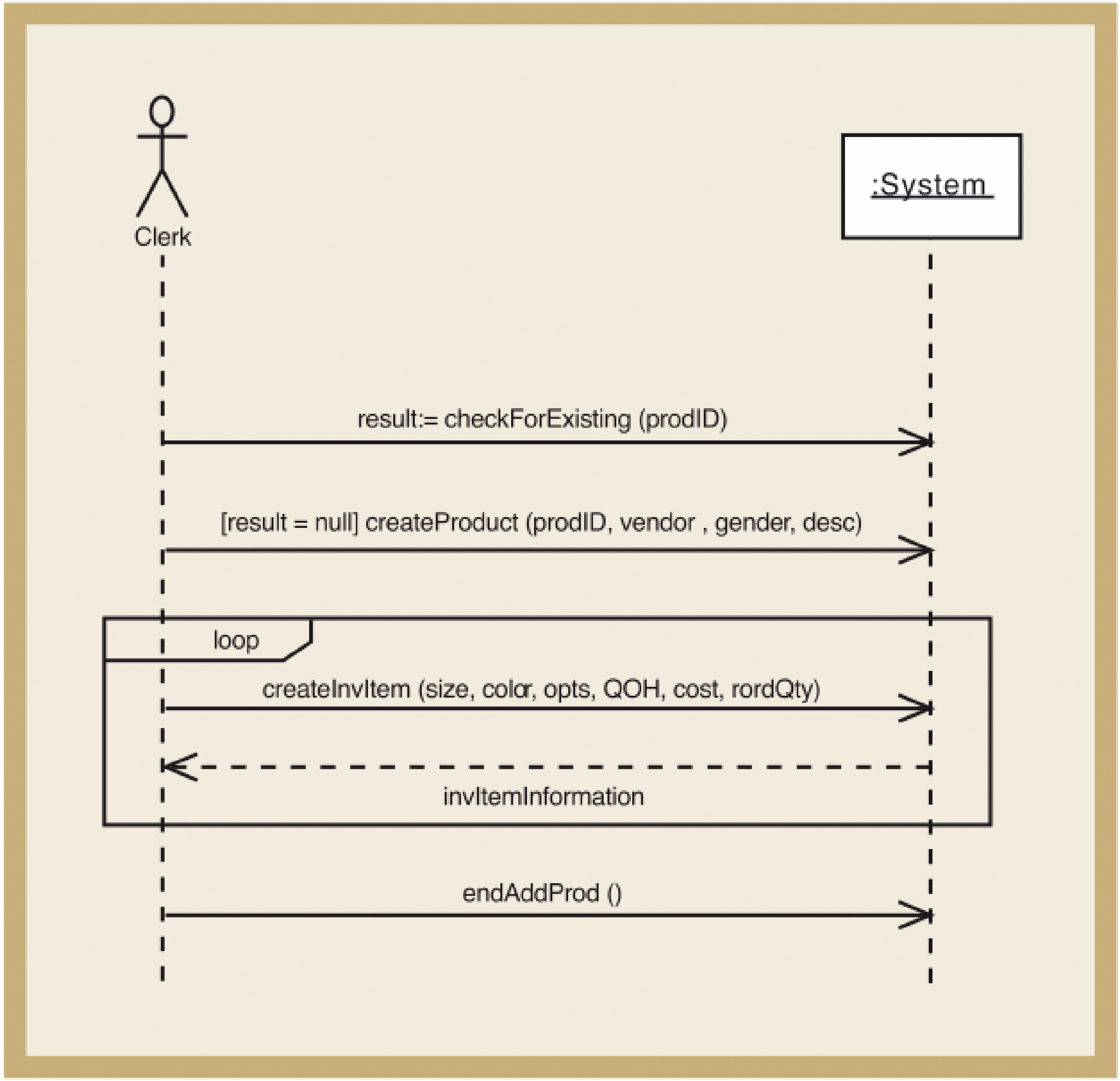
Assumptions About First-Cut Sequence Diagram

- 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

Maintain Product Information Use Case—Start with SSD

Figure 11-15

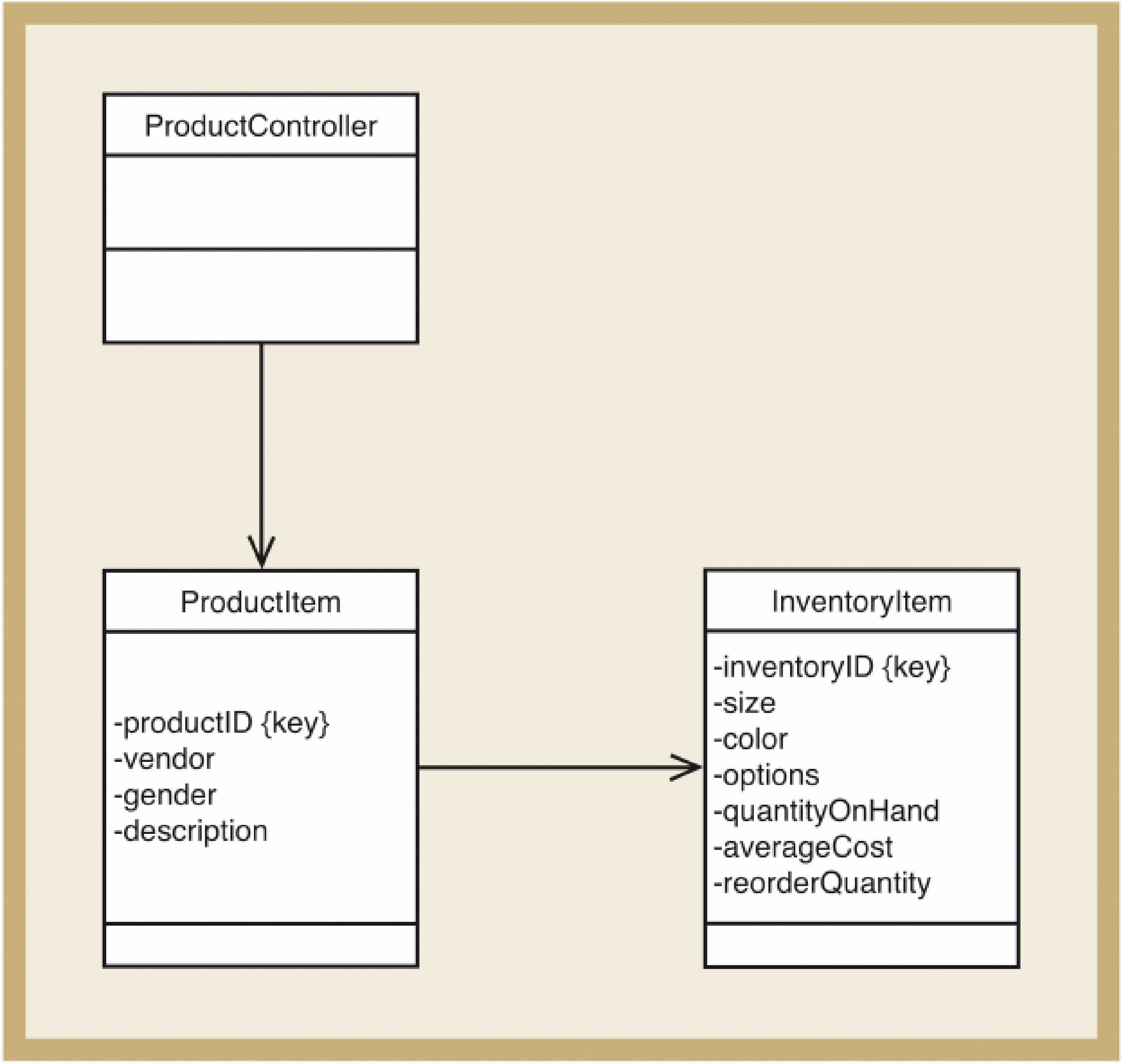
SSD for the *Maintain product information* use case


Clerk
:System
result:= checkForExisting (prodID)
[result = null] createProduct (prodID, vendor , gender, desc)
loop
createInvItem (size, color, opts, QOH, cost, rordQty)
invItemInformation
endAddProd ()

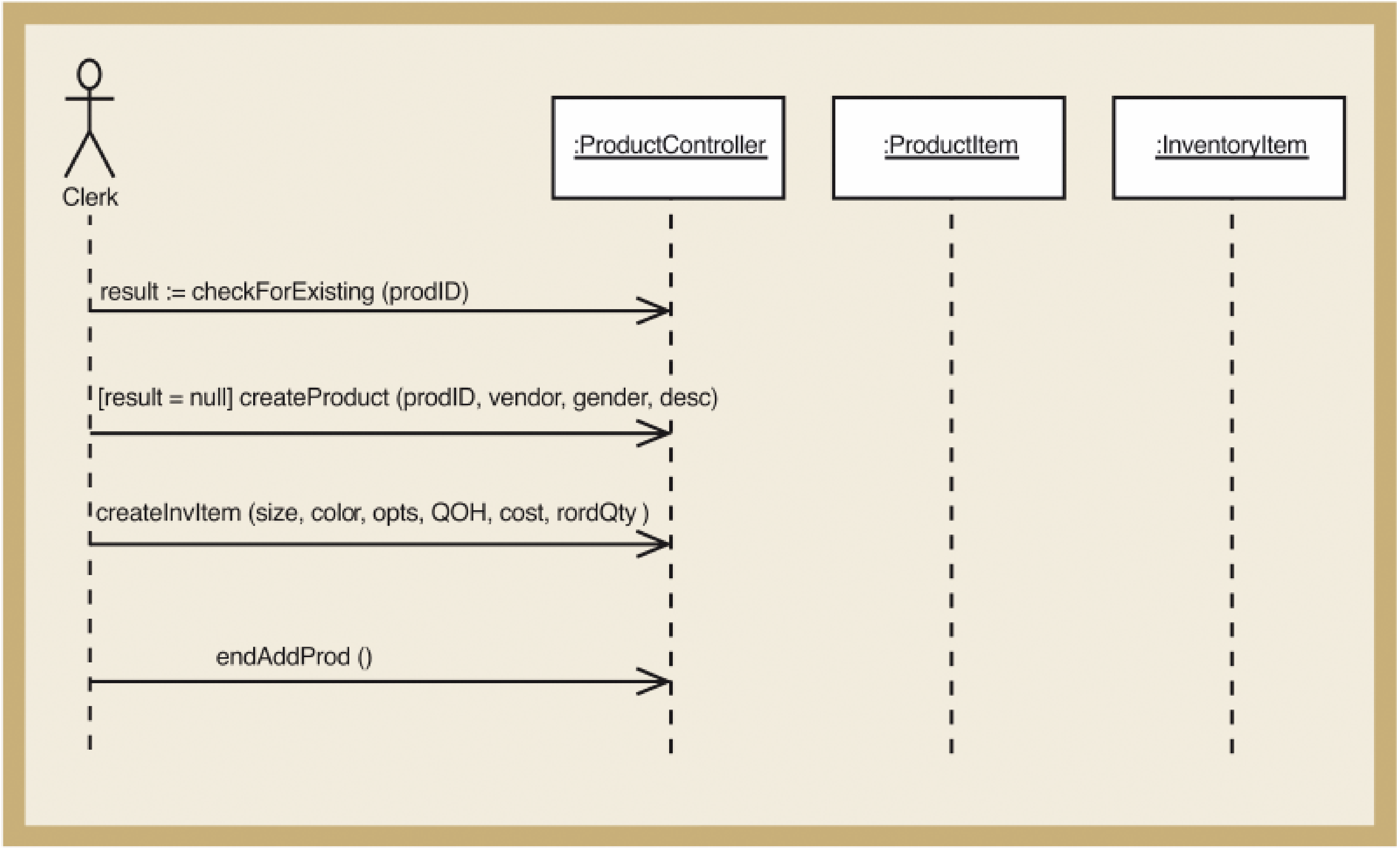
Add Controller and Identify Domain Classes and Navigation Visibility

Figure 11-16

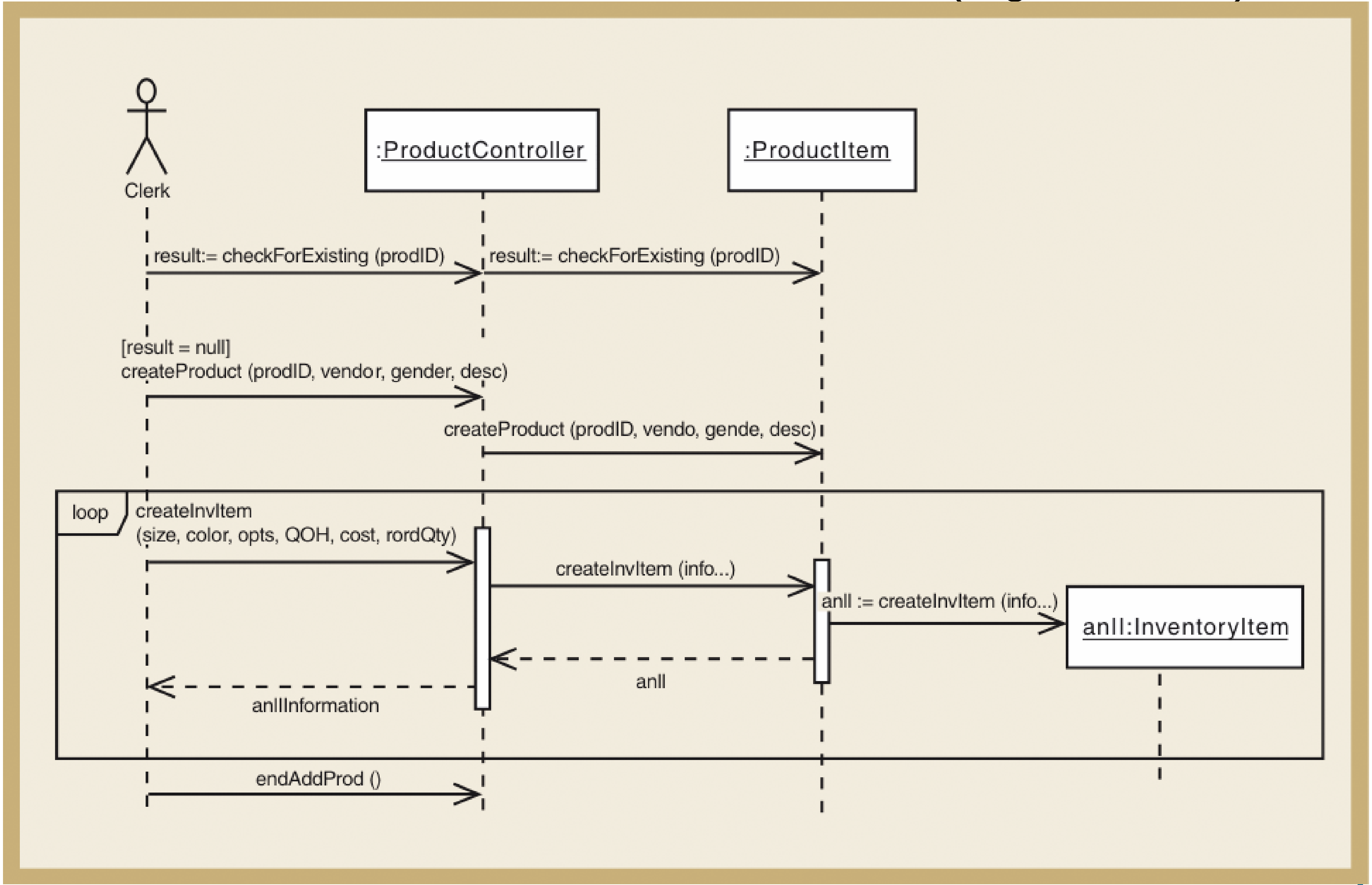
First-cut design class diagram for the *Maintain product information* use case



Replace :System Object in SSD with Controller and Domain Objects (Figure 11-17)


Clerk
:ProductController
:ProductItem
:InventoryItem
result := checkForExisting (prodID)
[result = null] createProduct (prodID, vendor, gender, desc)
createInvItem (size, color, opts, QOH, cost, rordQty)
endAddProd ()

First-Cut Sequence Diagram for *Maintain Product Information* Use Case (Figure 11-18)


Clerk
:ProductController
:ProductItem
result:= checkForExisting (prodID)
result:= checkForExisting (prodID)
[result = null]
createProduct (prodID, vendor, gender, desc)
createProduct (prodID, vendo, gende, desc)
loop
createInvItem
(size, color, opts, QOH, cost, rordQty)
createInvItem (info...)
anII := createInvItem (info...)
anII:InventoryItem
anII
anIIInformation
endAddProd ()

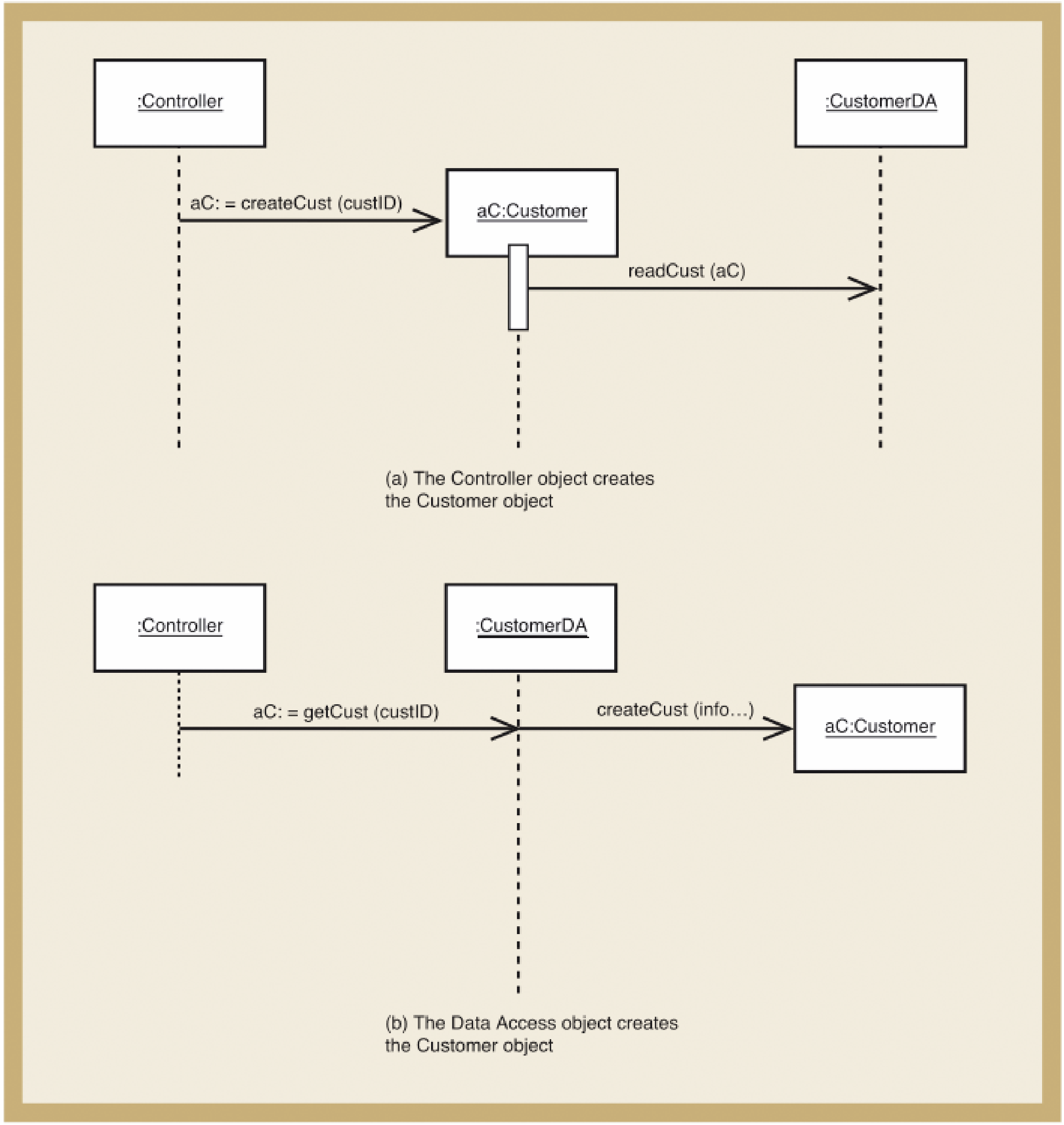
Developing a Multilayer Design

- 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
  - 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

Figure 11-19

Two methods for accessing the database and instantiating objects



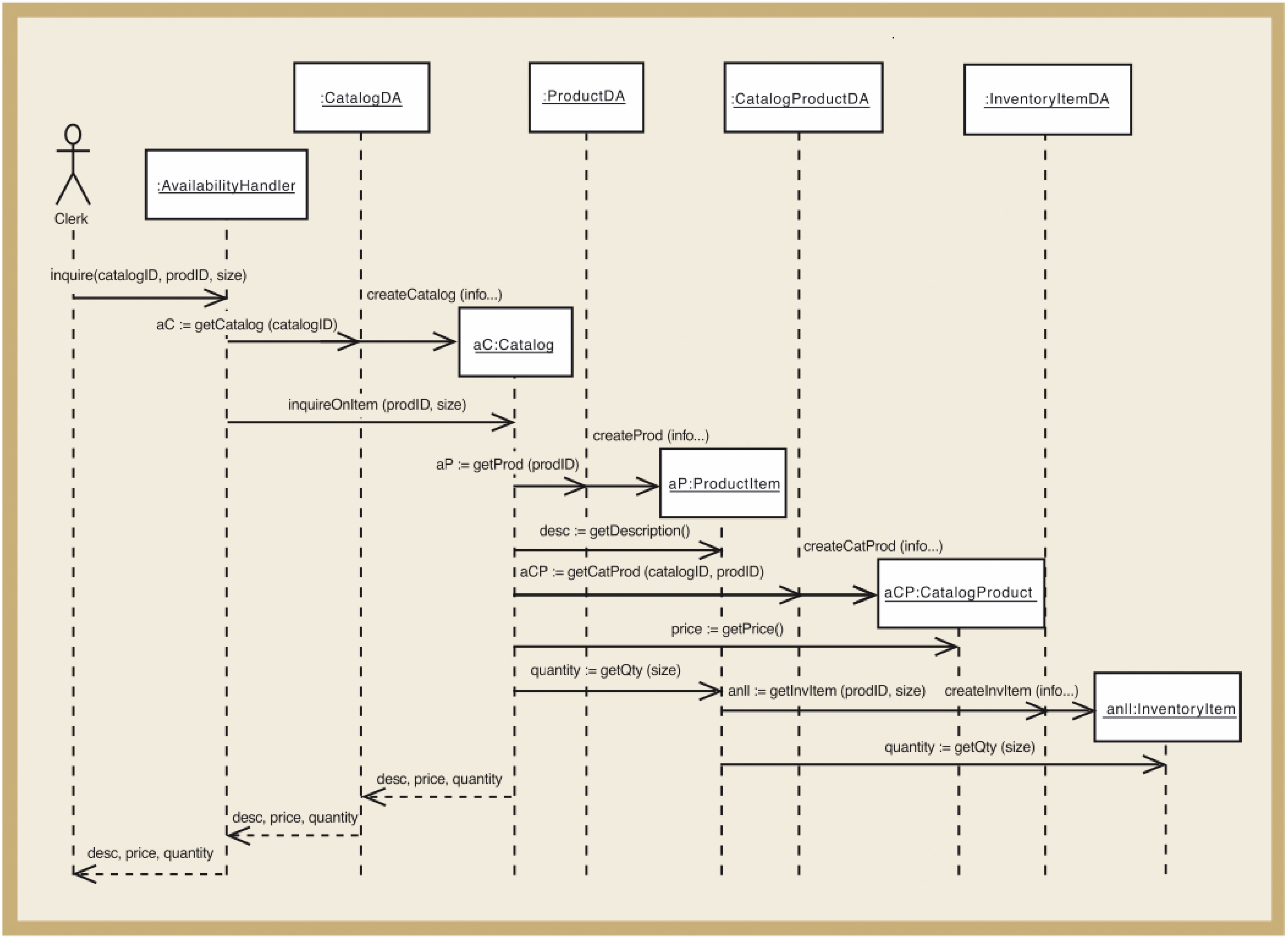
(a) The Controller object creates the Customer object

(b) The Data Access object creates the Customer object

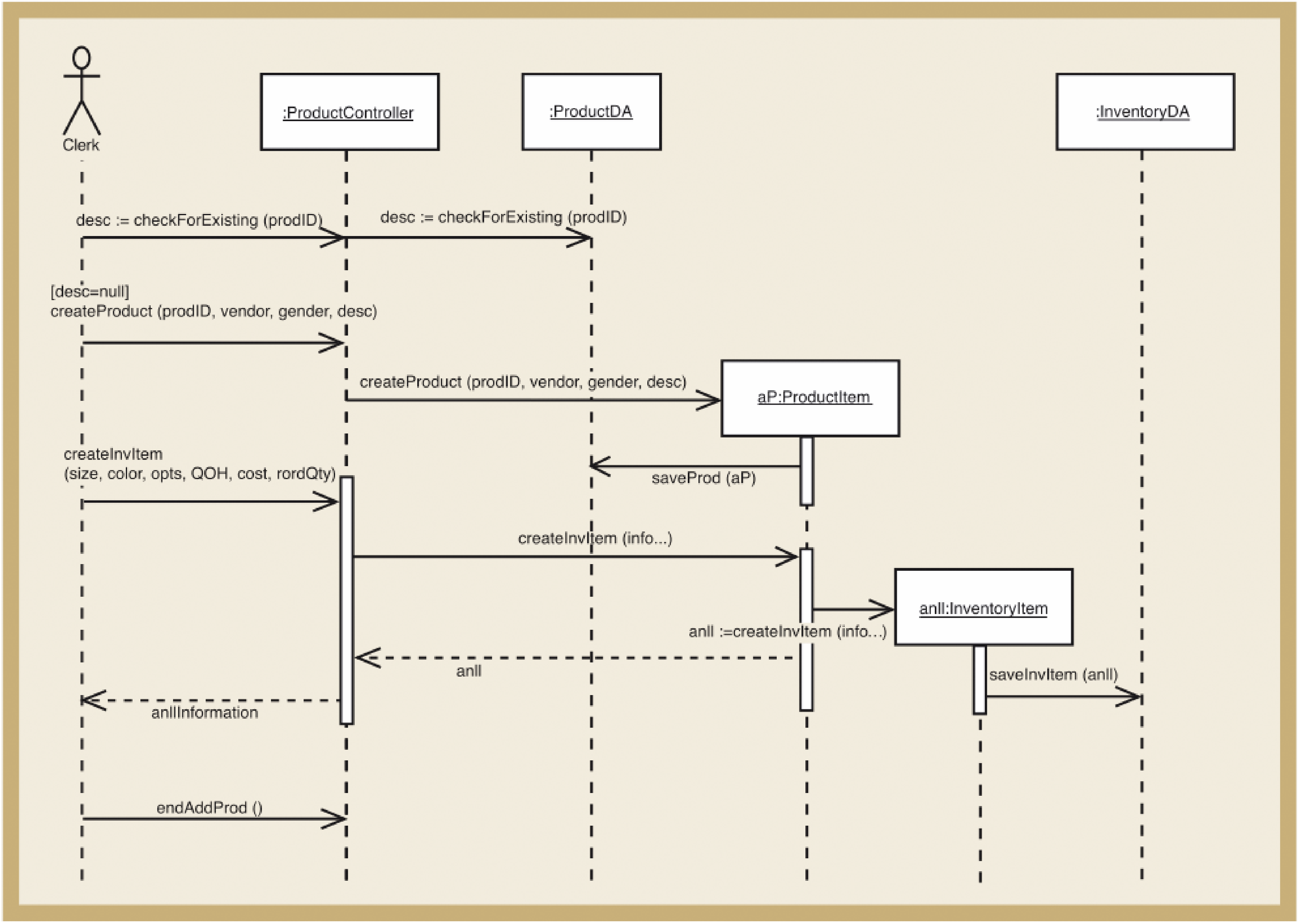
Approaches to Data Access Layer (Continued)

- 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 *Look Up Item Availability* Use Case (Figure 11-20)



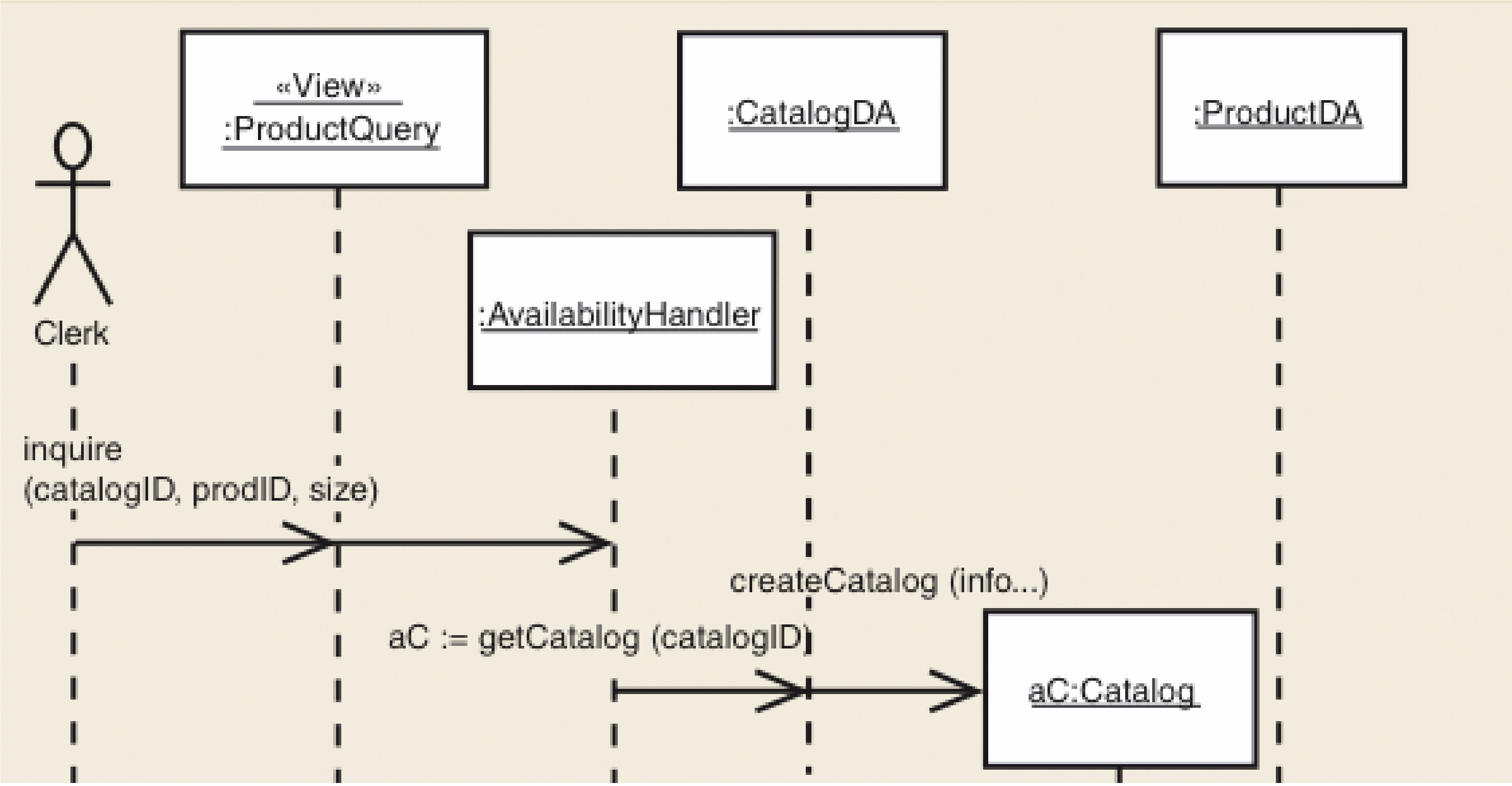
Adding Data Access Layer for *Maintain Product Information* Use Case (Figure 11-21)


Clerk
:ProductController
:ProductDA
:InventoryDA
desc := checkForExisting (prodID)
desc := checkForExisting (prodID)
[desc=null]
createProduct (prodID, vendor, gender, desc)
createProduct (prodID, vendor, gender, desc)
aP:ProductItem
saveProd (aP)
createInvItem
(size, color, opts, QOH, cost, rordQty)
createInvItem (info...)
anII:InventoryItem
anII :=createInvItem (info...)
anII
saveInvItem (anII)
anIIInformation
endAddProd ()

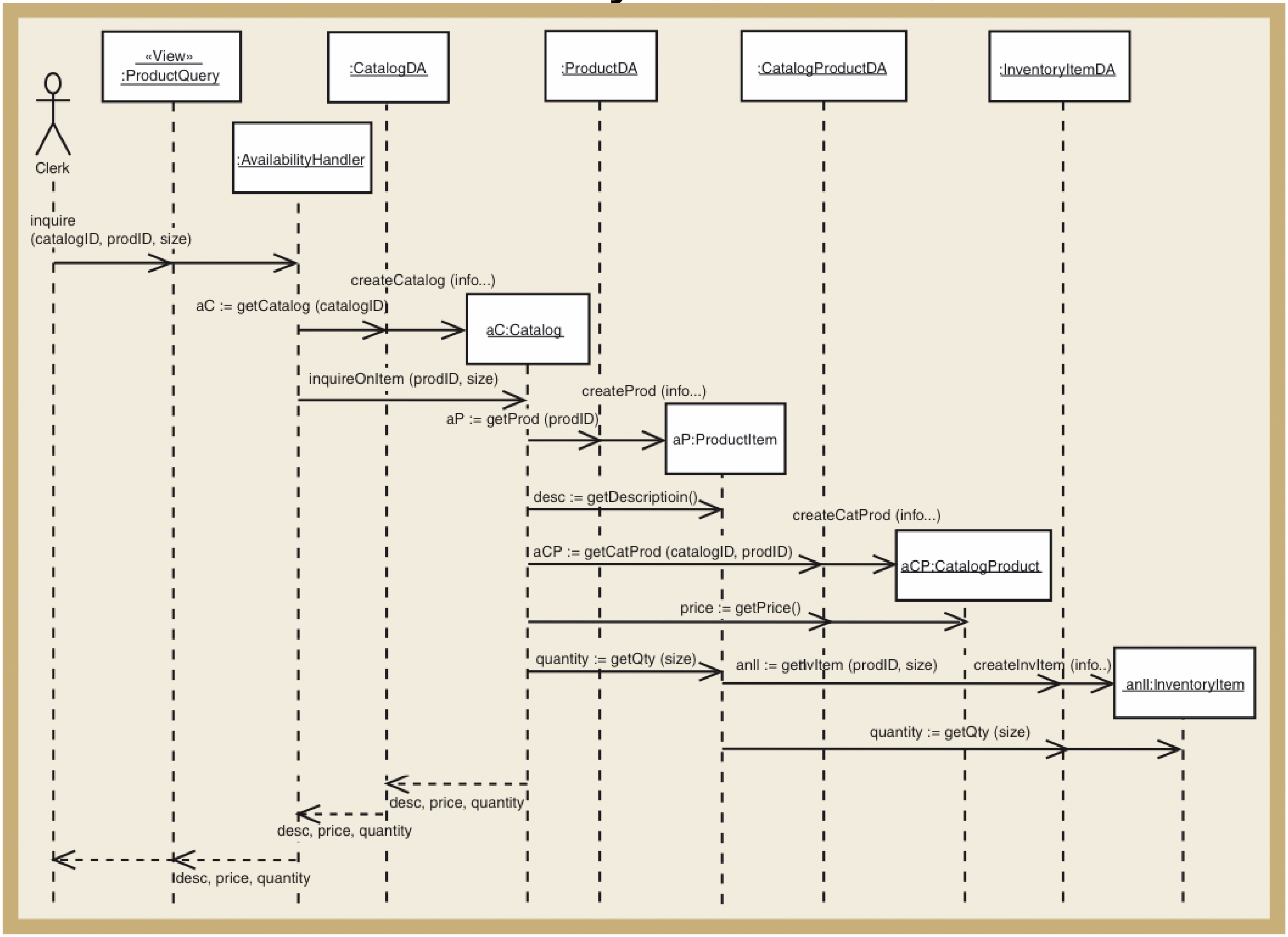
Designing the View Layer

- 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
- Details of interface design and HCI in Chapters 13 and 14

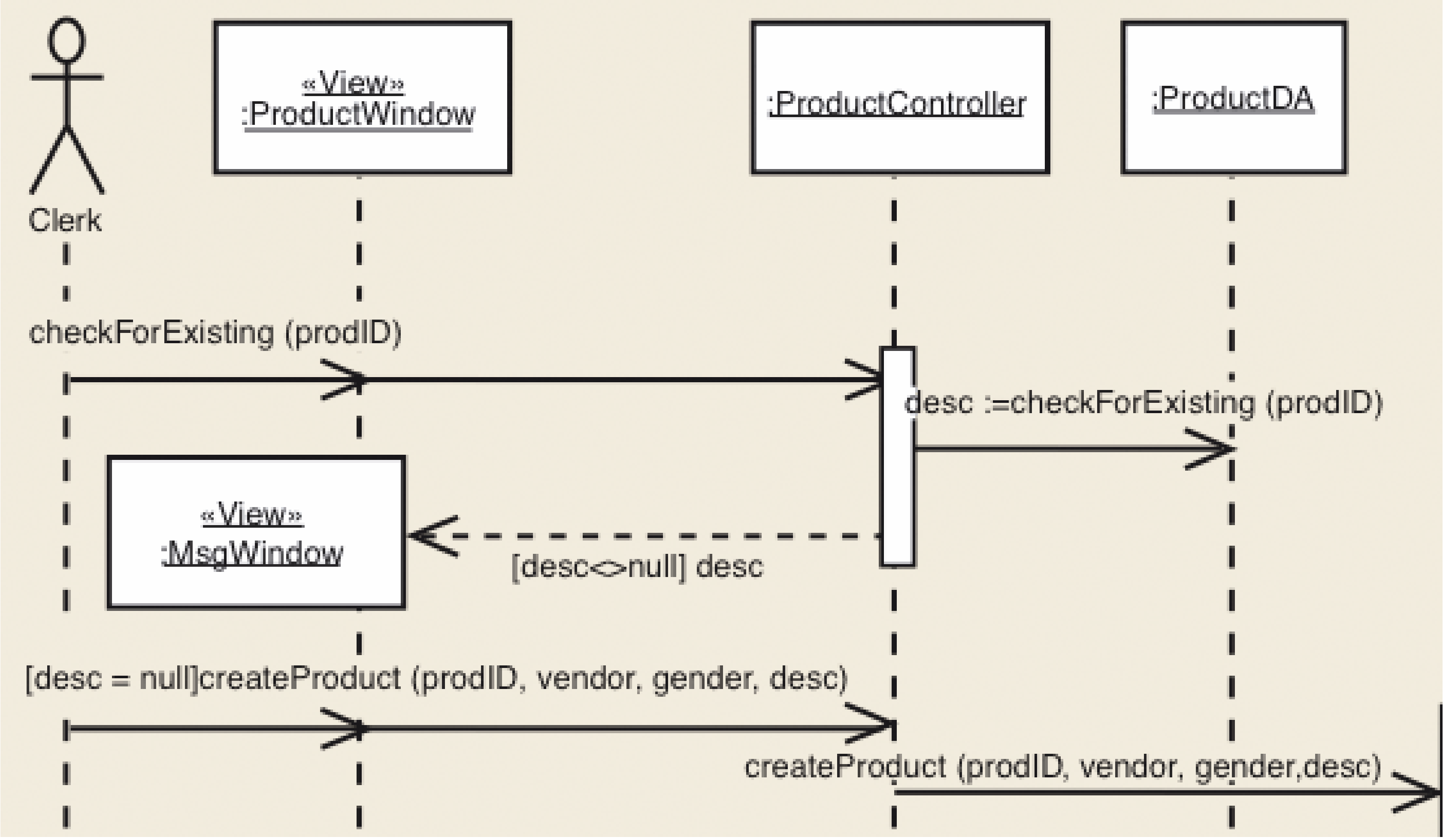
<<View>> ProductQuery Form Added for *Look Up Item Availability* Use Case



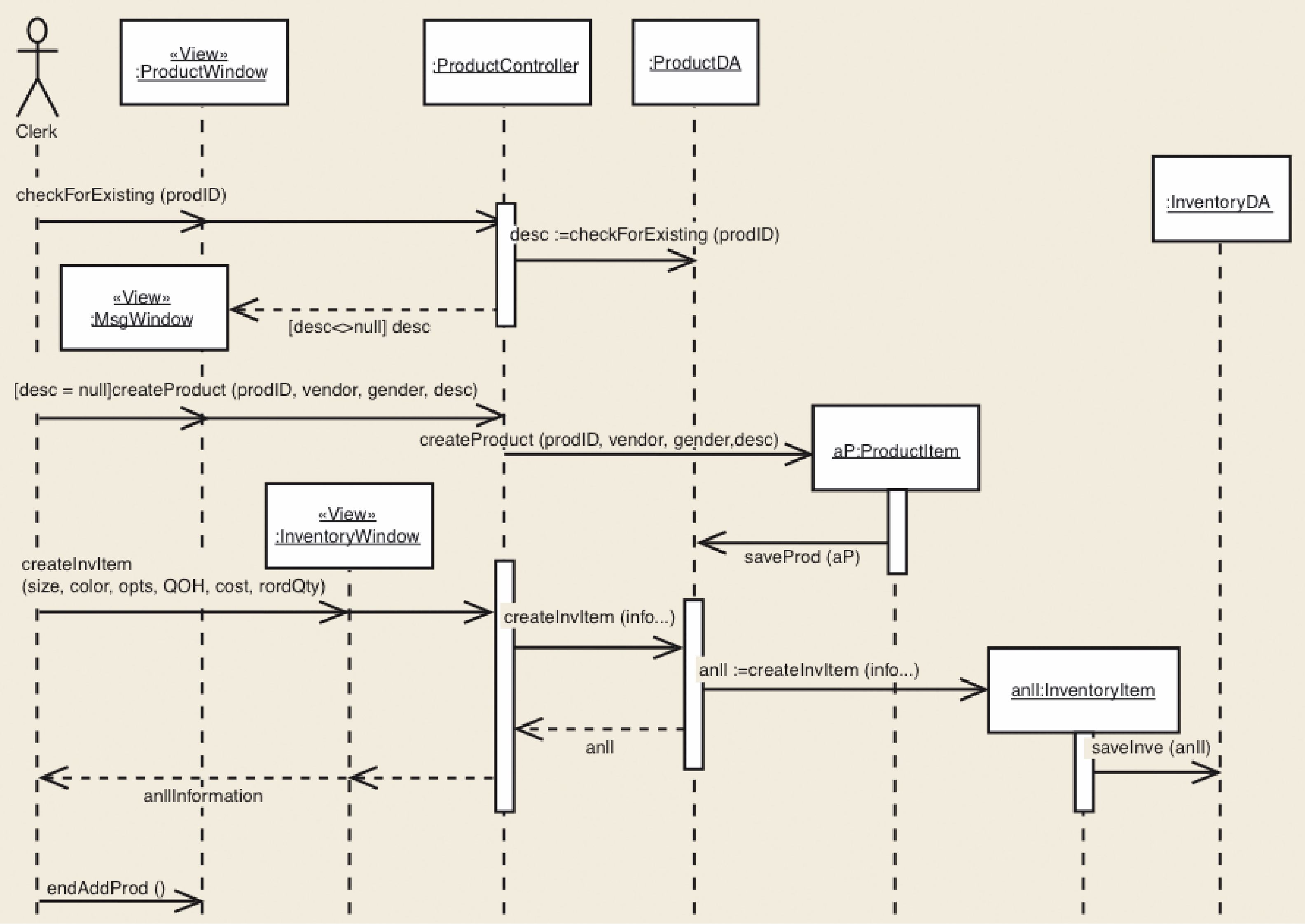
Complete *Look Up Item Availability* Use Case with View Layer (Figure 11-22)



ProductWindow and MsgWindow for *Maintain Product Information* Use Case


Clerk
«View»
:ProductWindow
:ProductController
:ProductDA
checkForExisting (prodID)
desc :=checkForExisting (prodID)
«View»
:MsgWindow
[desc<>null] desc
[desc = null]createProduct (prodID, vendor, gender, desc)
createProduct (prodID, vendor, gender,desc)

Complete *Maintain Product Information* Use Case Use Case with View Layer (Figure 11-23)

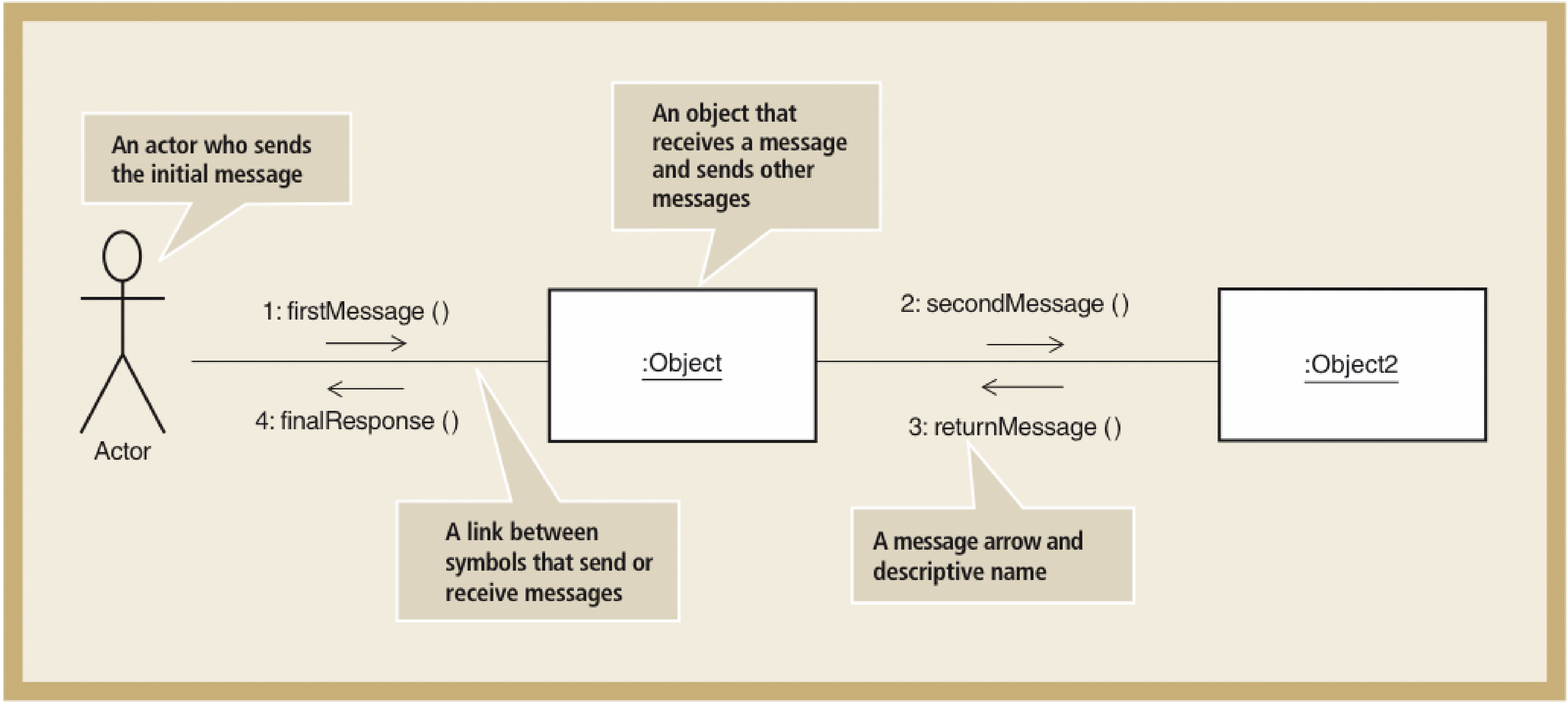


Designing with Communication Diagrams

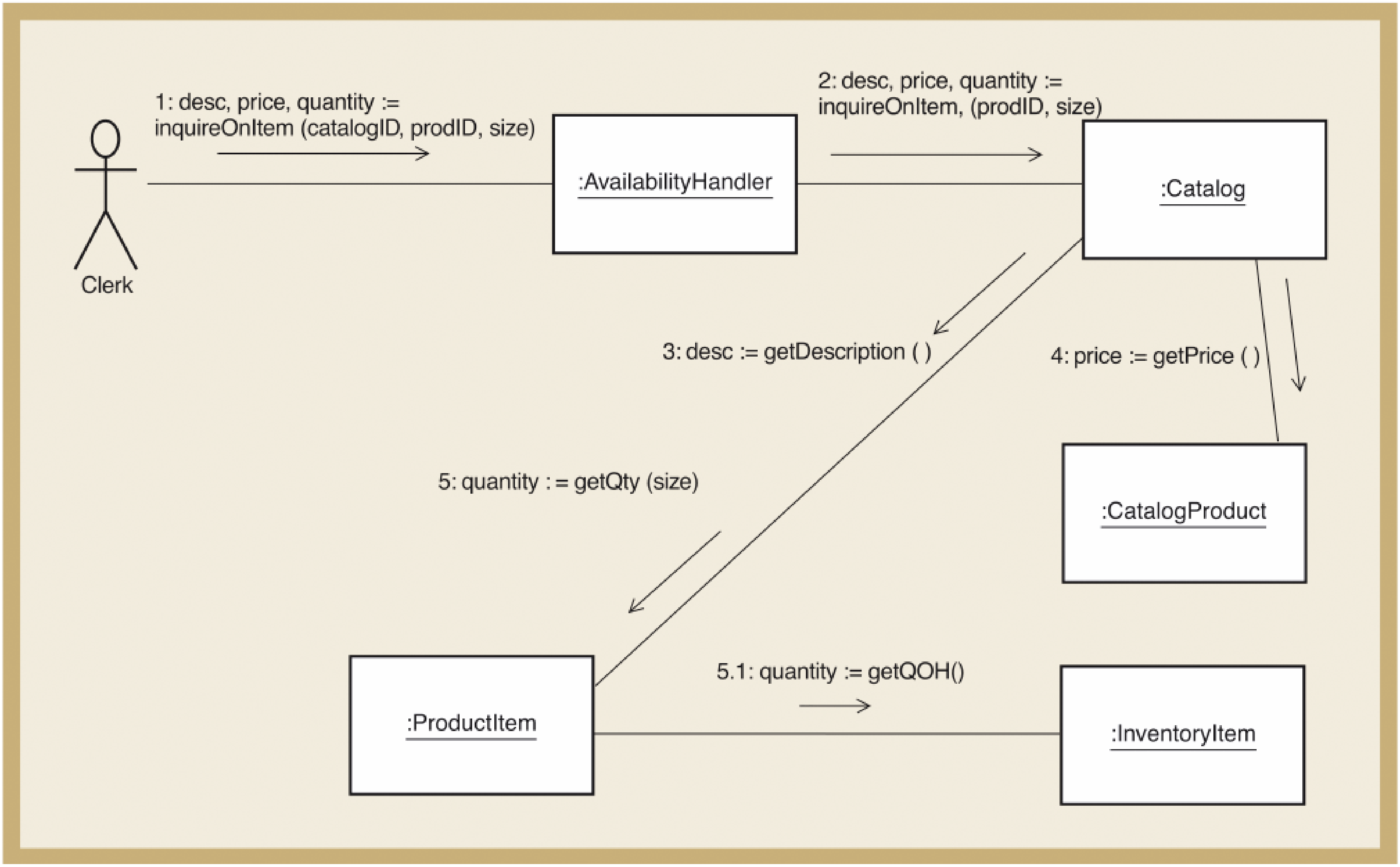
- Communication diagrams and sequence diagrams
  - Both are interaction diagrams
  - Both capture same information
  - Process of designing is same for both
- Model used is designer’s personal preference
  - Sequence diagram – use case descriptions and dialogs follow sequence of steps
  - Communication diagram – emphasizes coupling

The Symbols of a Communication Diagram

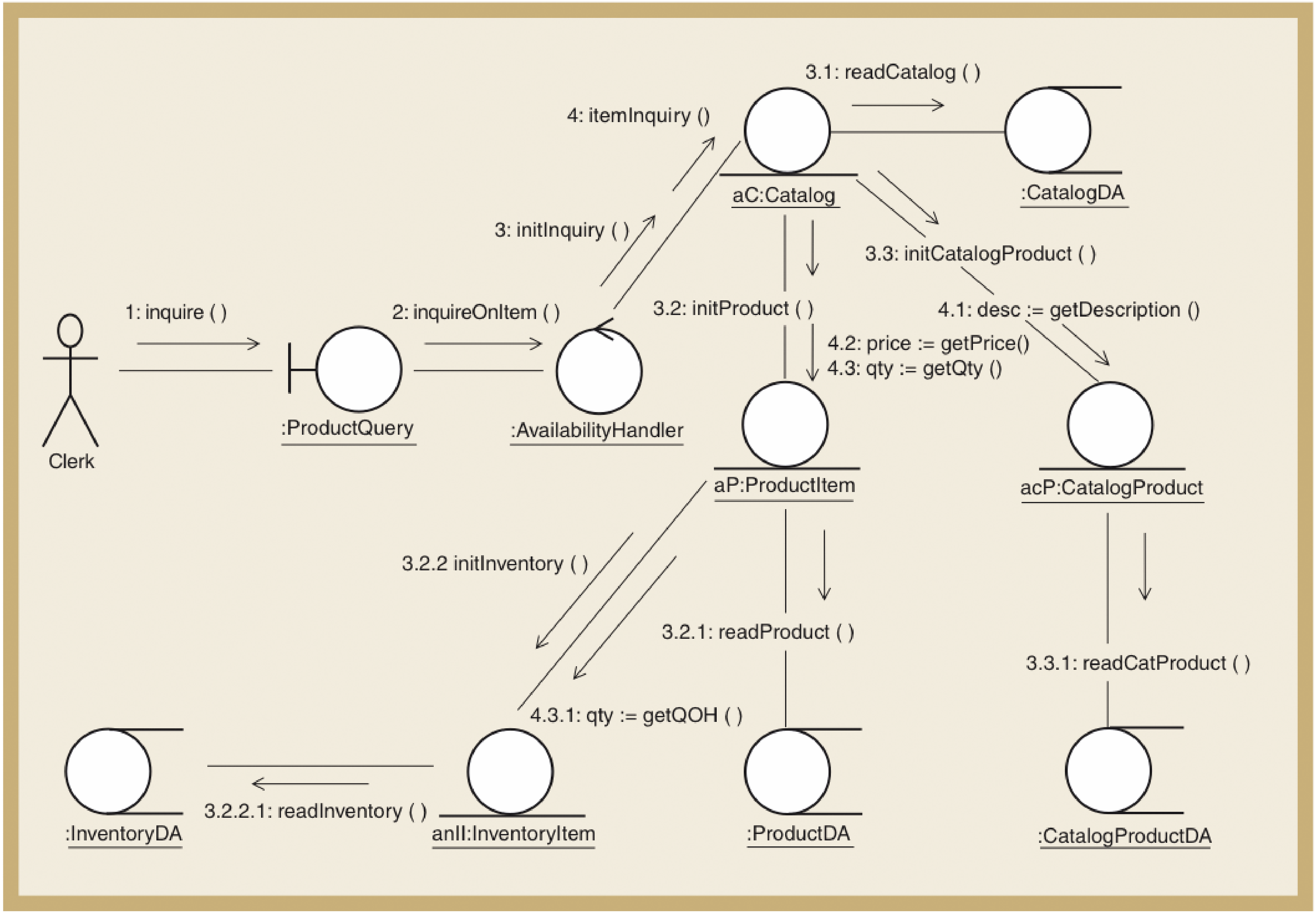
(Figure 11-24)


An actor who sends the initial message
An object that receives a message and sends other messages
1: firstMessage ()
:Object
2: secondMessage ()
:Object2
Actor
4: finalResponse ()
3: returnMessage ()
A link between symbols that send or receive messages
A message arrow and descriptive name

A Communication Diagram for *Look Up Item Availability* (Figure 11-25)


1: desc, price, quantity :=
inquireOnItem (catalogID, prodID, size)
2: desc, price, quantity :=
inquireOnItem, (prodID, size)
Clerk
:AvailabilityHandler
:Catalog
3: desc := getDescription ( )
4: price := getPrice ( )
5: quantity : = getQty (size)
:CatalogProduct
5.1: quantity := getQOH()
:ProductItem
:InventoryItem

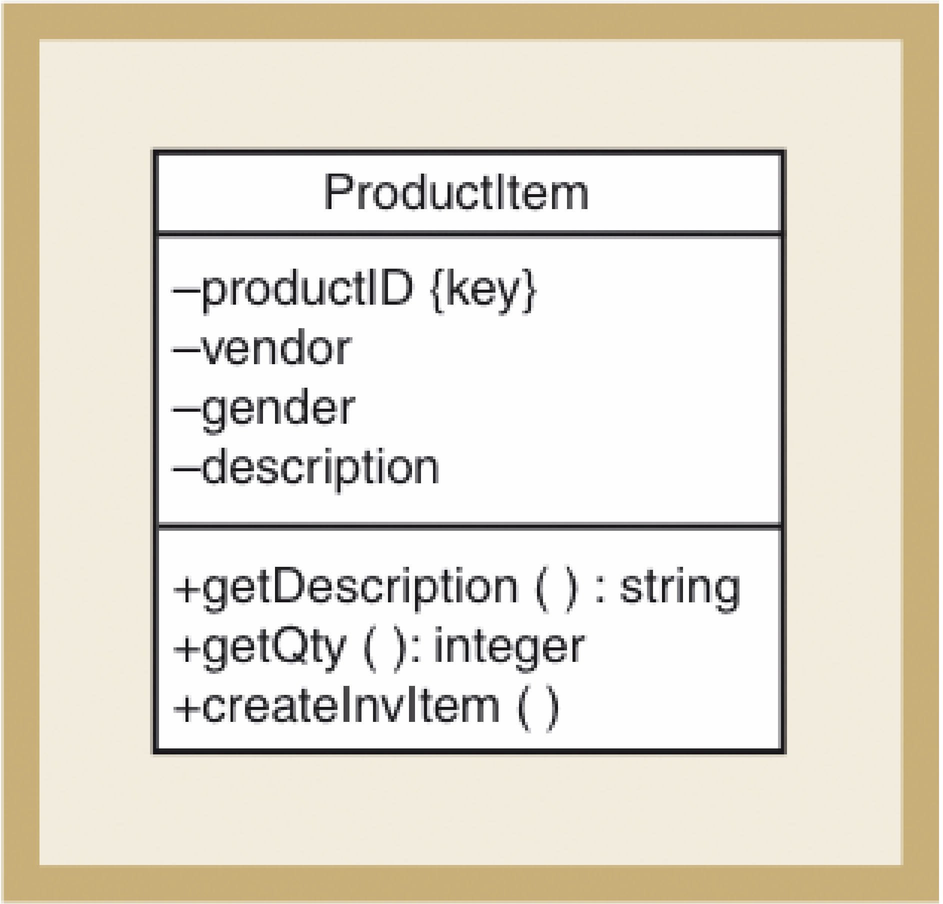
Look Up Item Availability Use Case Using Iconic Symbols (Figure 11-26)



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 ProductItem Class (Figure 11-27)



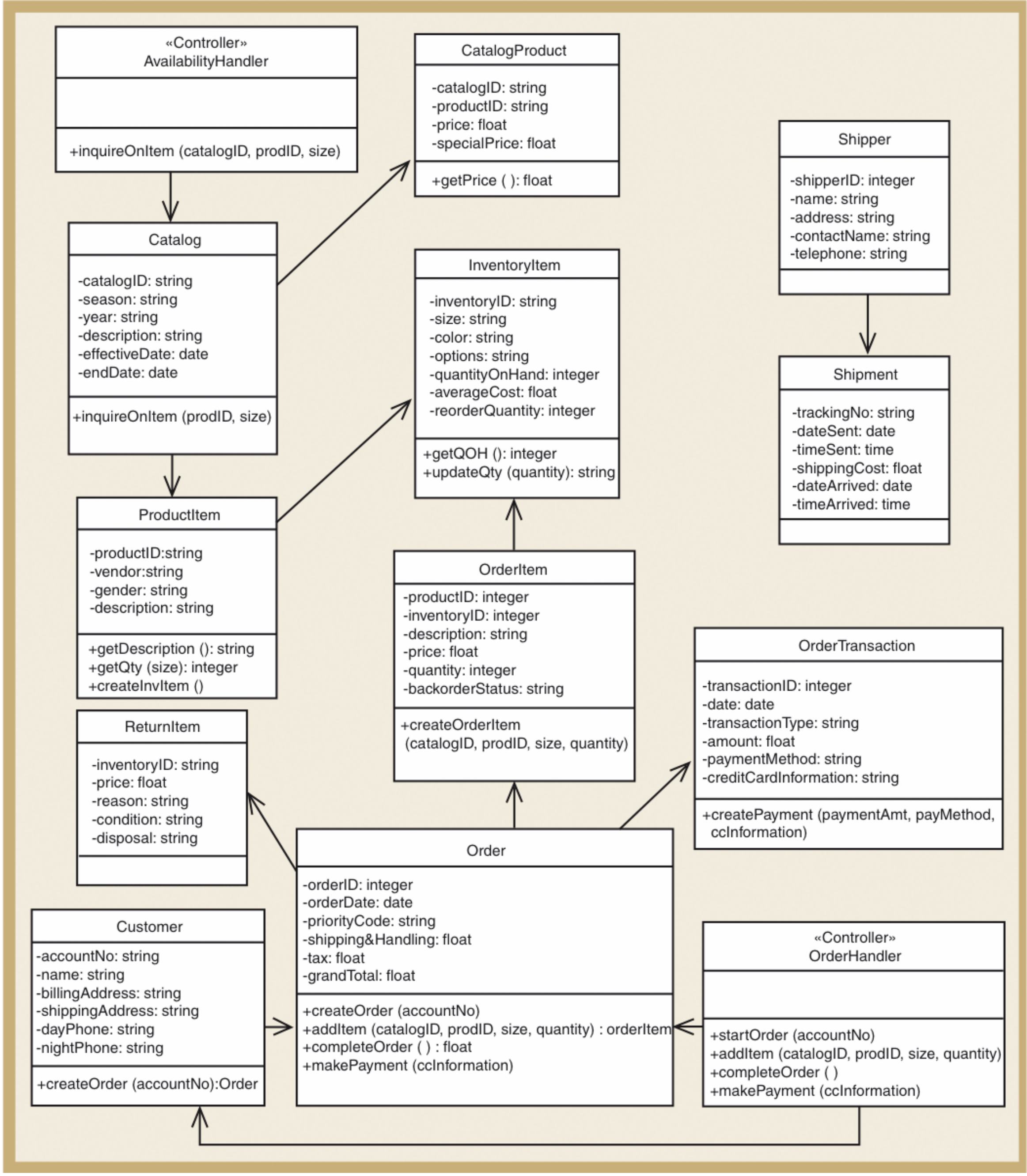
ProductItem

–productID {key}
–vendor
–gender
–description

+getDescription ( ) : string
+getQty ( ): integer
+createInvItem ( )

Updated Design Class Diagram for the Domain Layer

(Figure 11-28)



«Controller»
AvailabilityHandler

+inquireOnItem (catalogID, prodID, size)

CatalogProduct

-catalogID: string
-productID: string
-price: float
-specialPrice: float

+getPrice ( ): float

Shipper

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

Catalog

-catalogID: string
-season: string
-year: string
-description: string
-effectiveDate: date
-endDate: date

+inquireOnItem (prodID, size)

InventoryItem

-inventoryID: string
-size: string
-color: string
-options: string
-quantityOnHand: integer
-averageCost: float
-reorderQuantity: integer

+getQOH (): integer
+updateQty (quantity): string

Shipment

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

ProductItem

-productID:string
-vendor:string
-gender: string
-description: string

+getDescription (): string
+getQty (size): integer
+createInvItem ()

OrderItem

-productID: integer
-inventoryID: integer
-description: string
-price: float
-quantity: integer
-backorderStatus: string

+createOrderItem
(catalogID, prodID, size, quantity)

OrderTransaction

-transactionID: integer
-date: date
-transactionType: string
-amount: float
-paymentMethod: string
-creditCardInformation: string

+createPayment (paymentAmt, payMethod, ccInformation)

ReturnItem

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

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)

Customer

-accountNo: string
-name: string
-billingAddress: string
-shippingAddress: string
-dayPhone: string
-nightPhone: string

+createOrder (accountNo):Order

«Controller»
OrderHandler

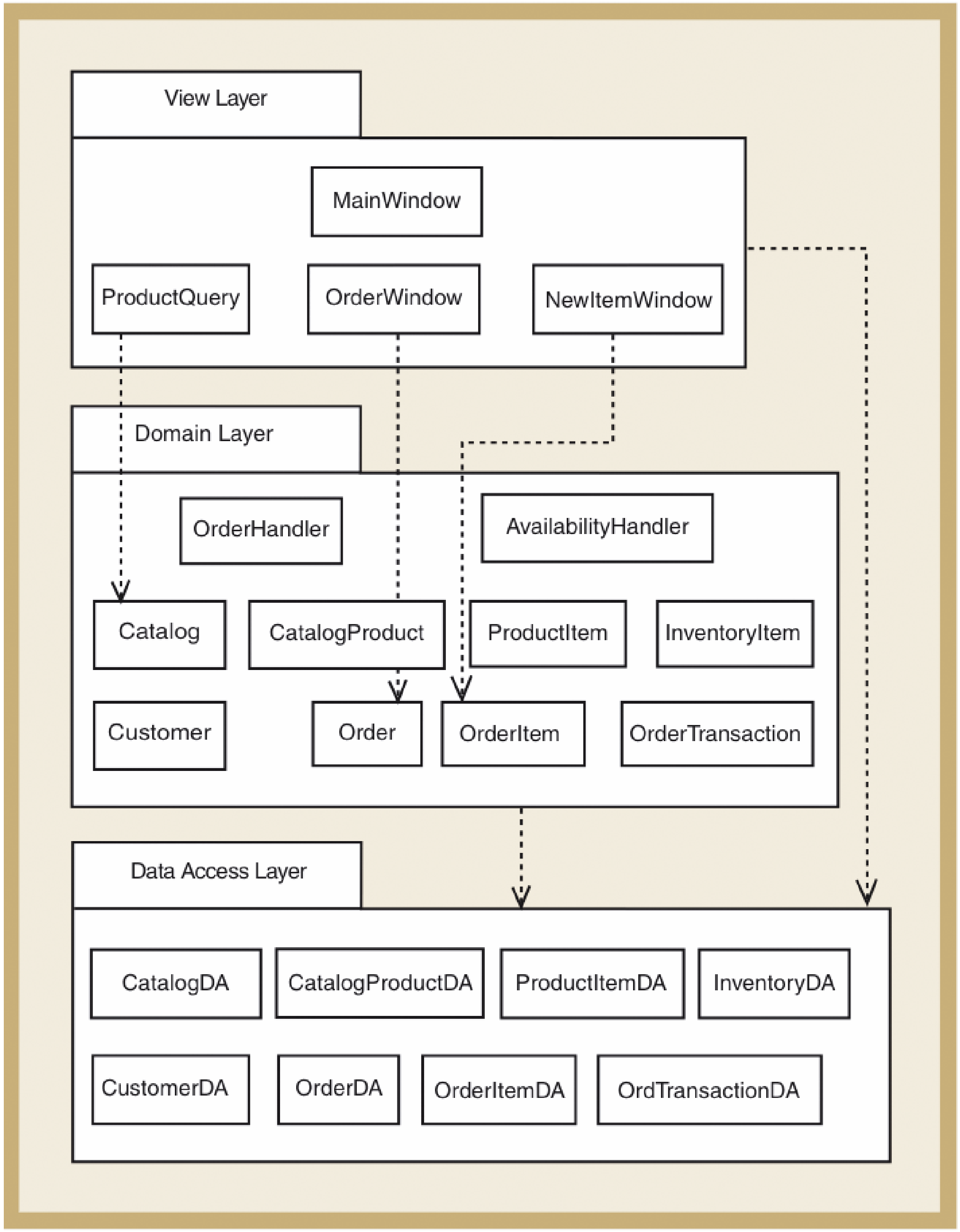
+startOrder (accountNo)
+addItem (catalogID, prodID, size, quantity)
+completeOrder ( )
+makePayment (ccInformation)

Package Diagram—Structuring the Major Components

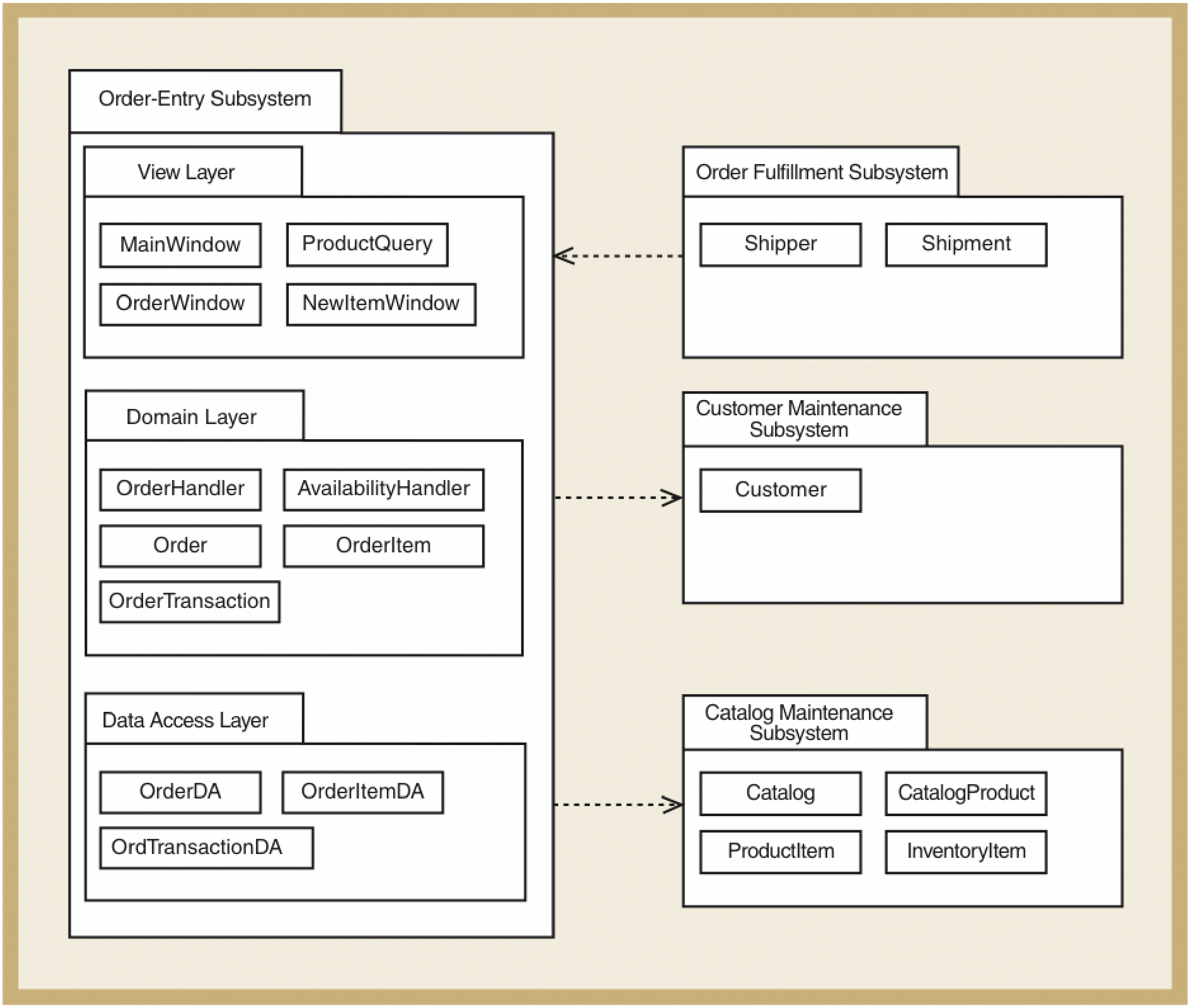
- High-level diagram in UML to associate classes of related groups
- Identifies major components of a system and dependencies
- Determines final program partitions for each layer
  - View, domain, data access
- Can divide system into subsystem and show nesting within packages

Partial Design of Three-Layer Package Diagram for RMO

(Figure 11-29)



RMO Subsystem Packages (Figure 11-30)



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

Summary

- Object-oriented design is the bridge between user requirements (in analysis models) and final system (constructed in programming language)
- 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)

Summary (continued)

- Object-oriented design principles must be applied
  - Encapsulation – data fields are placed in classes along with methods to process that data
  - Low coupling – connectivity between classes
  - High cohesion – nature of an individual class
  - Protection from variations – parts of a system that are unlikely to change are segregated from those that will
  - Indirection – an intermediate class is placed between two classes to decouple them but still link them
  - Separation navigation – access classes have to other classes
- Three-layer design is used because maintainable