88823459
การวิเคราะห์และออกแบบระบบเชิงวัตถุ

พีระศักดิ์ เพียรประสิทธิ์

Outline

- The Object-Oriented Approach
- Detailed Design

วัตถุประสงค์การเรียนรู้

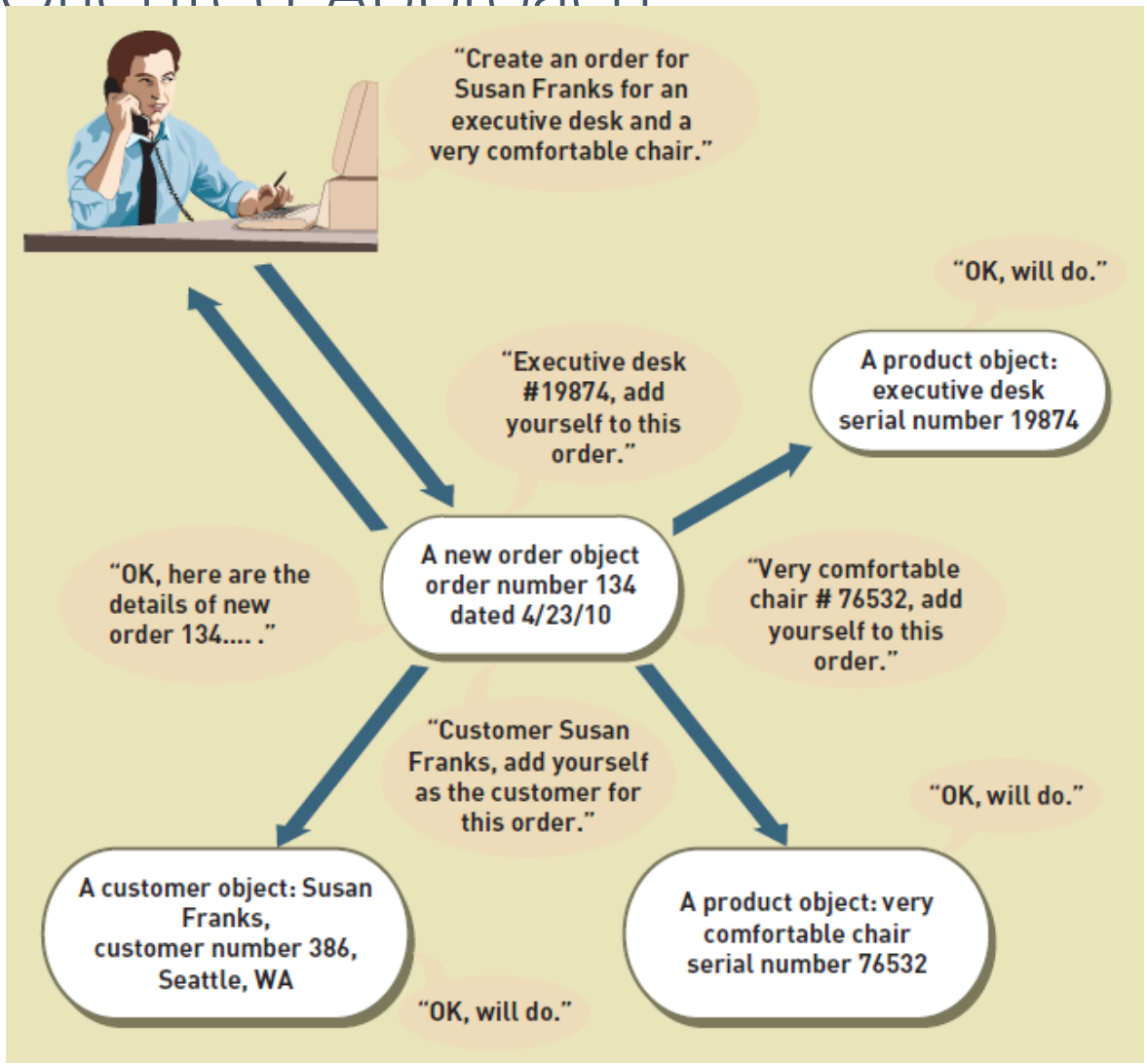
- เข้าใจรายละเอียดการออกแบบระบบเชิงวัตถุ
- กำหนดรายละเอียด วิธีการ ลำดับการสื่อสารระหว่างวัตถุ

The Object-Oriented Approach

- Object-oriented analysis (OOA)
  - The process of identifying and defining the use cases and sets of objects (classes) in the new system
- Object-oriented design (OOD)
  - Defining all of the types of objects necessary to communicate with people and devices and showing how they interact to complete tasks
- Object-oriented programming (OOP)
  - Writing statements that define the actual classes and what each object of the class does

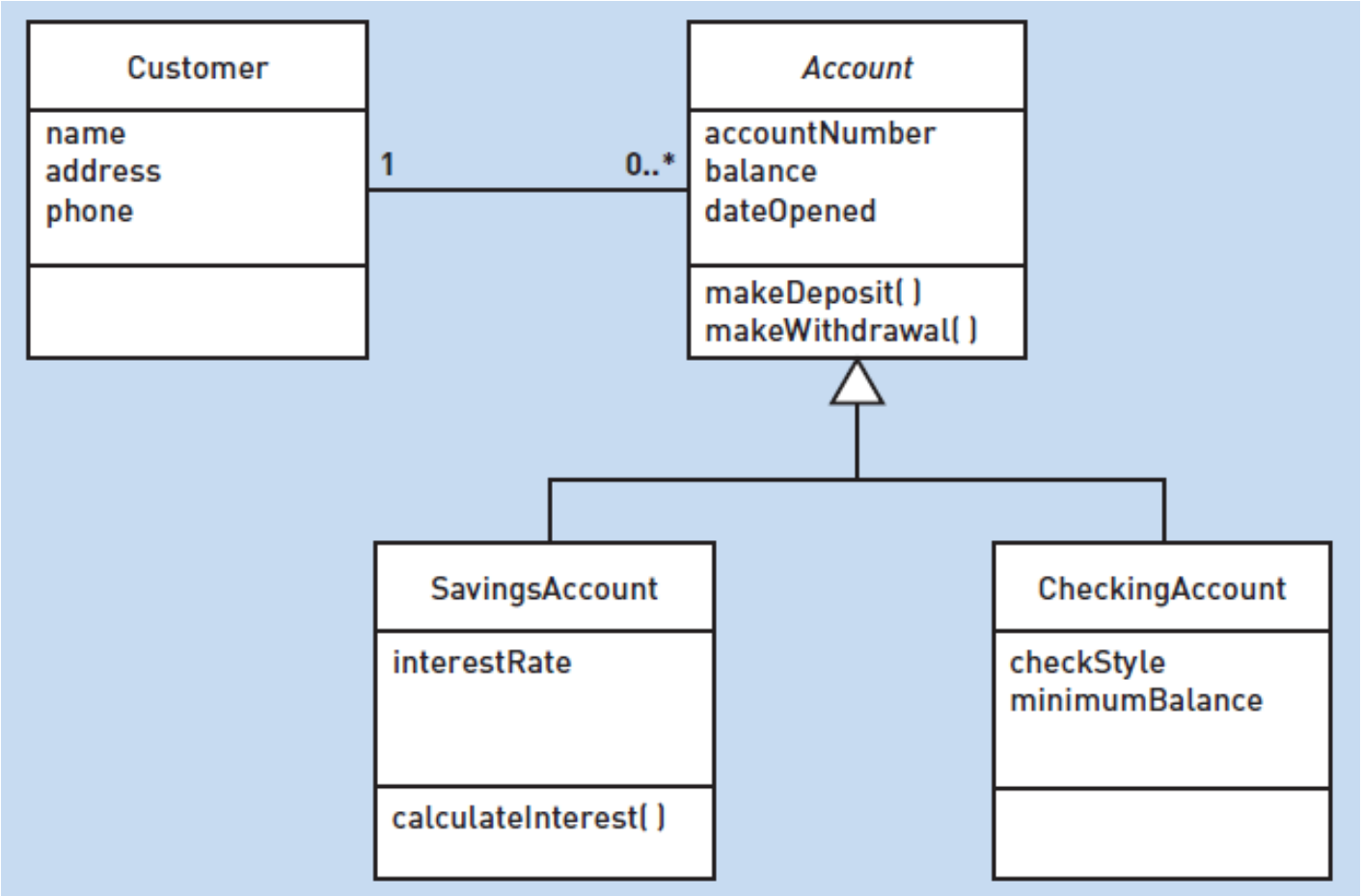
The Object-Oriented Approach

- Example showing the OO concept
- Objects collaborate to get a task done



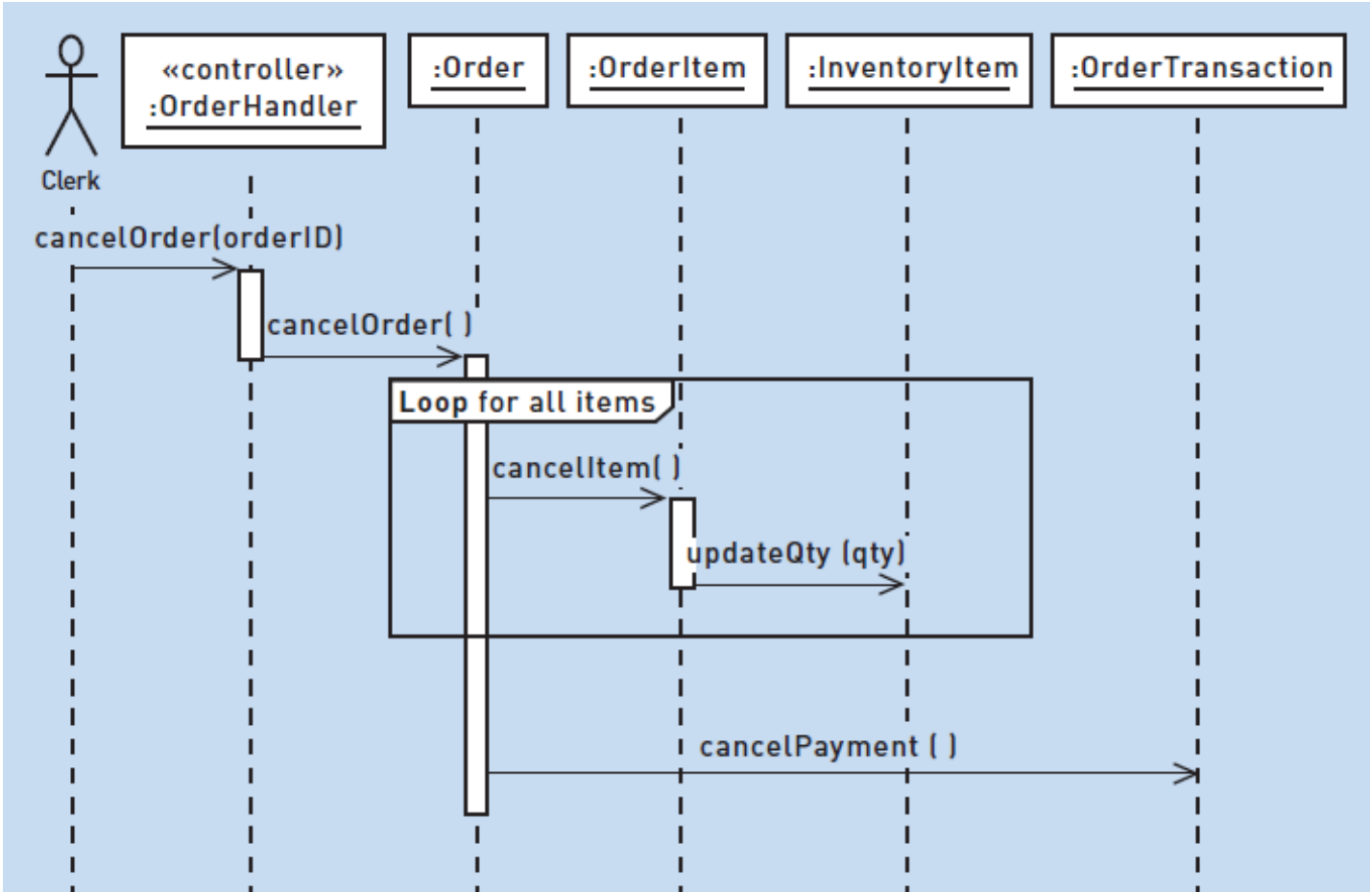
The Object-Oriented Approach

- UML Design Class Diagram



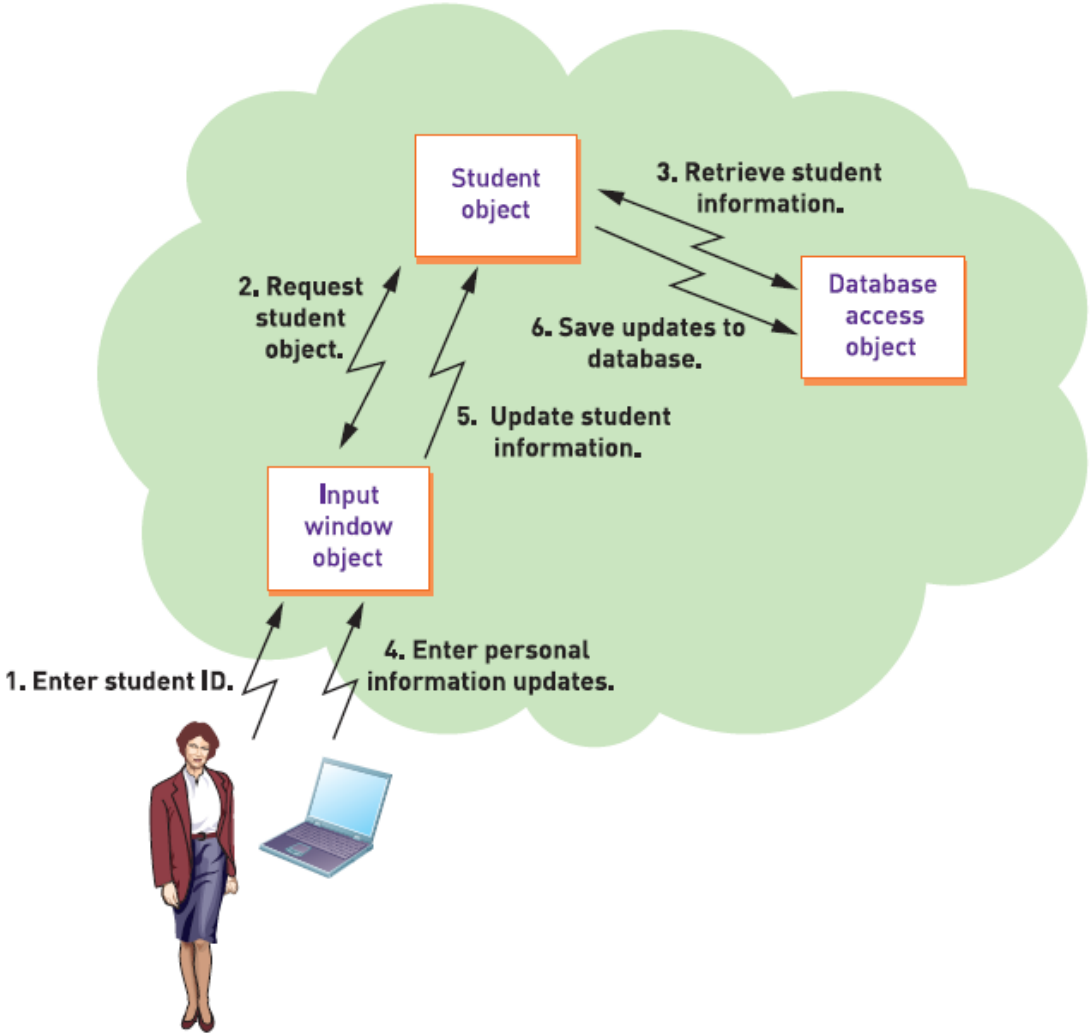
The Object-Oriented Approach

- UML Sequence Diagram



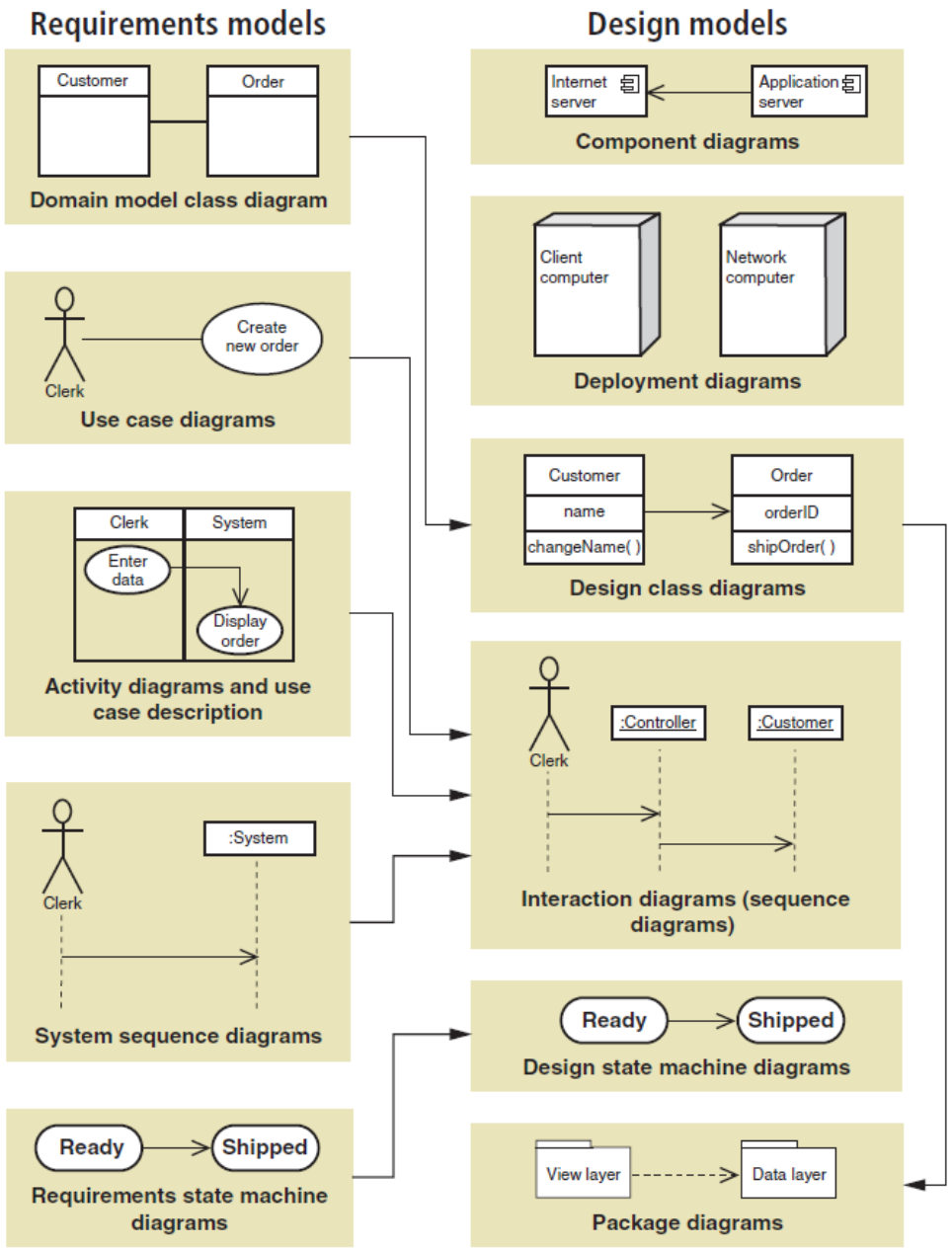
Object-Oriented Program Flow

Three Layer Architecture

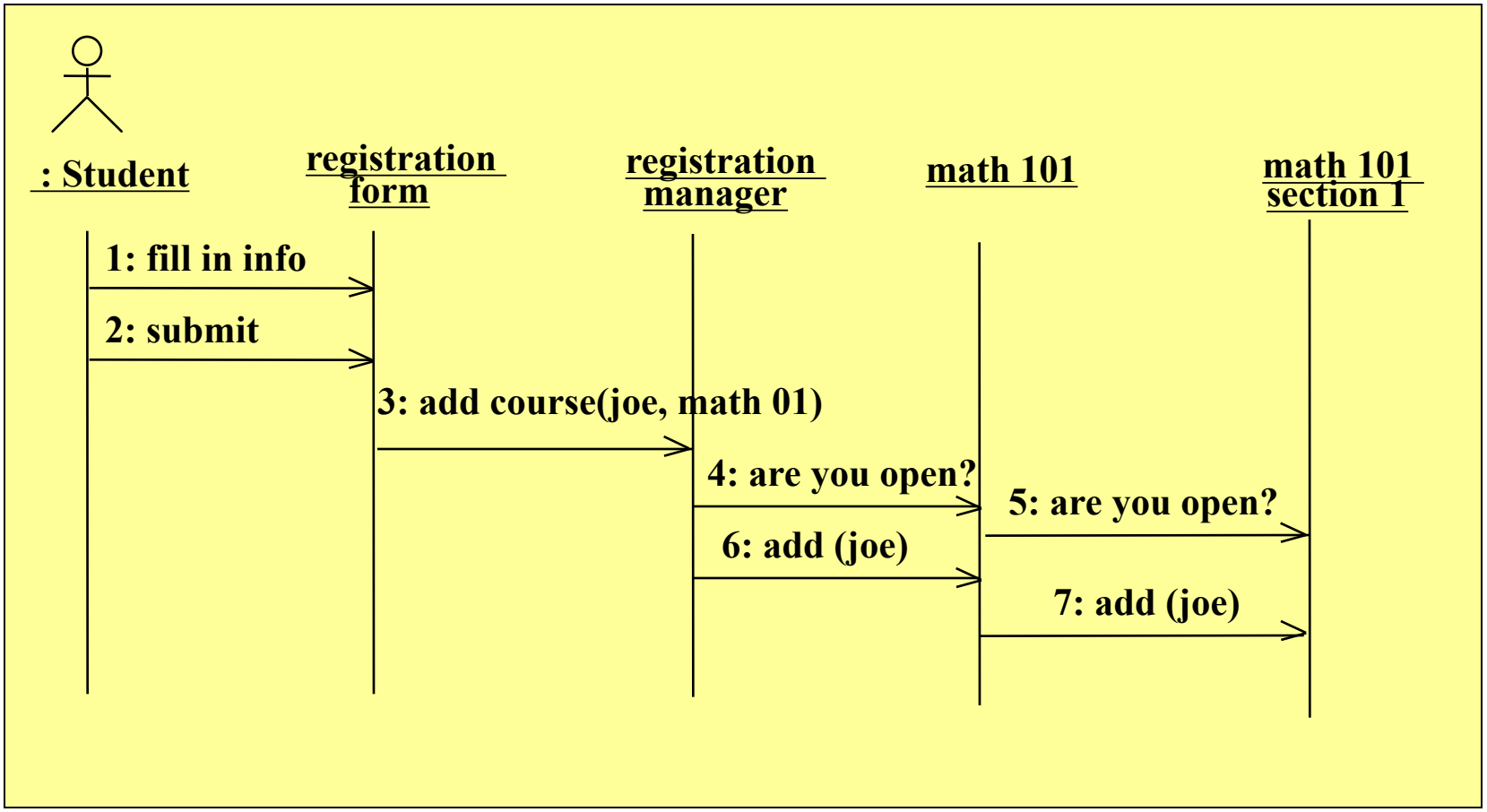


UML Requirements vs. Design Models

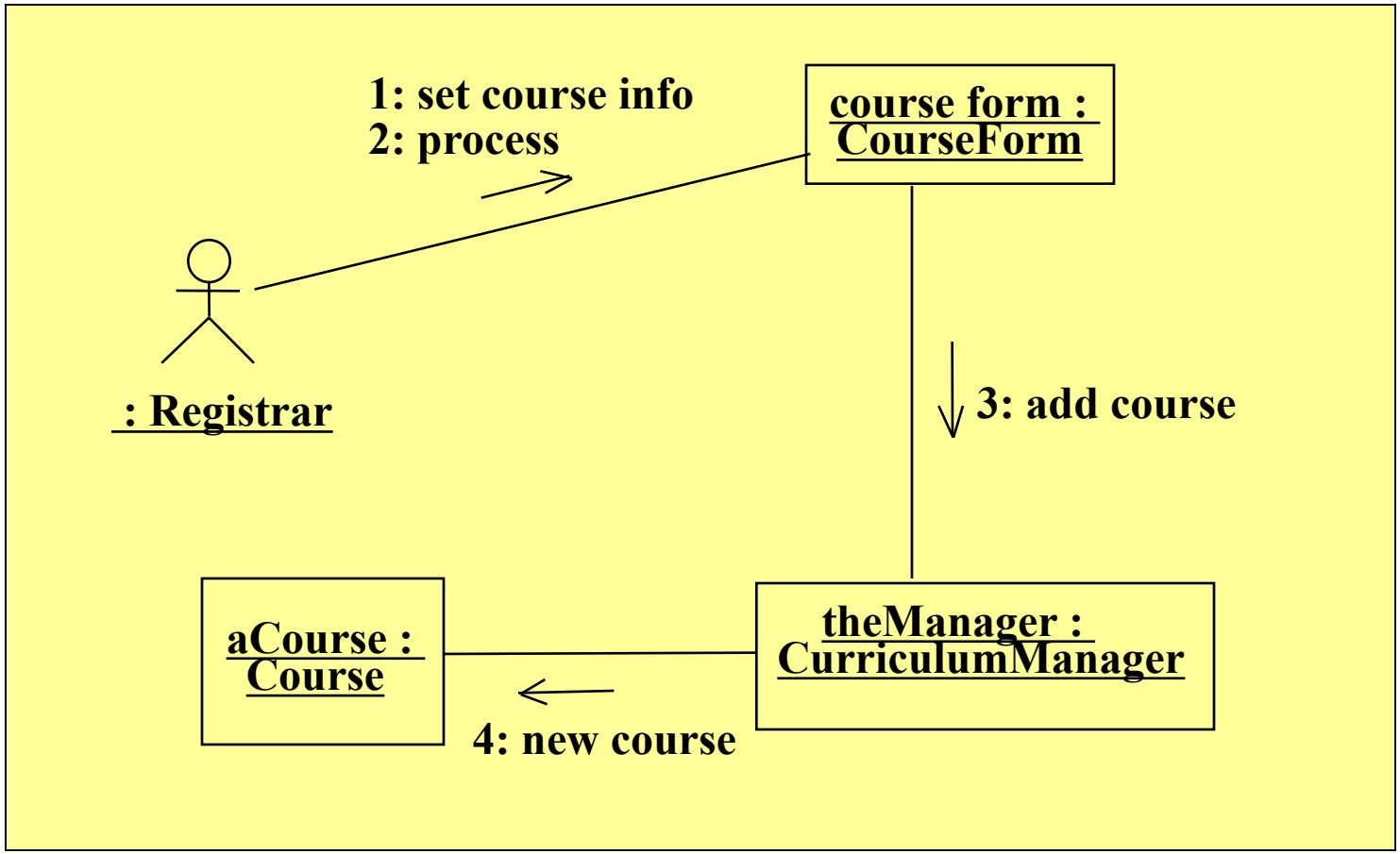
Diagrams are enhanced and extended



A Sequence Diagram



A Collaboration Diagram



Detailed Design

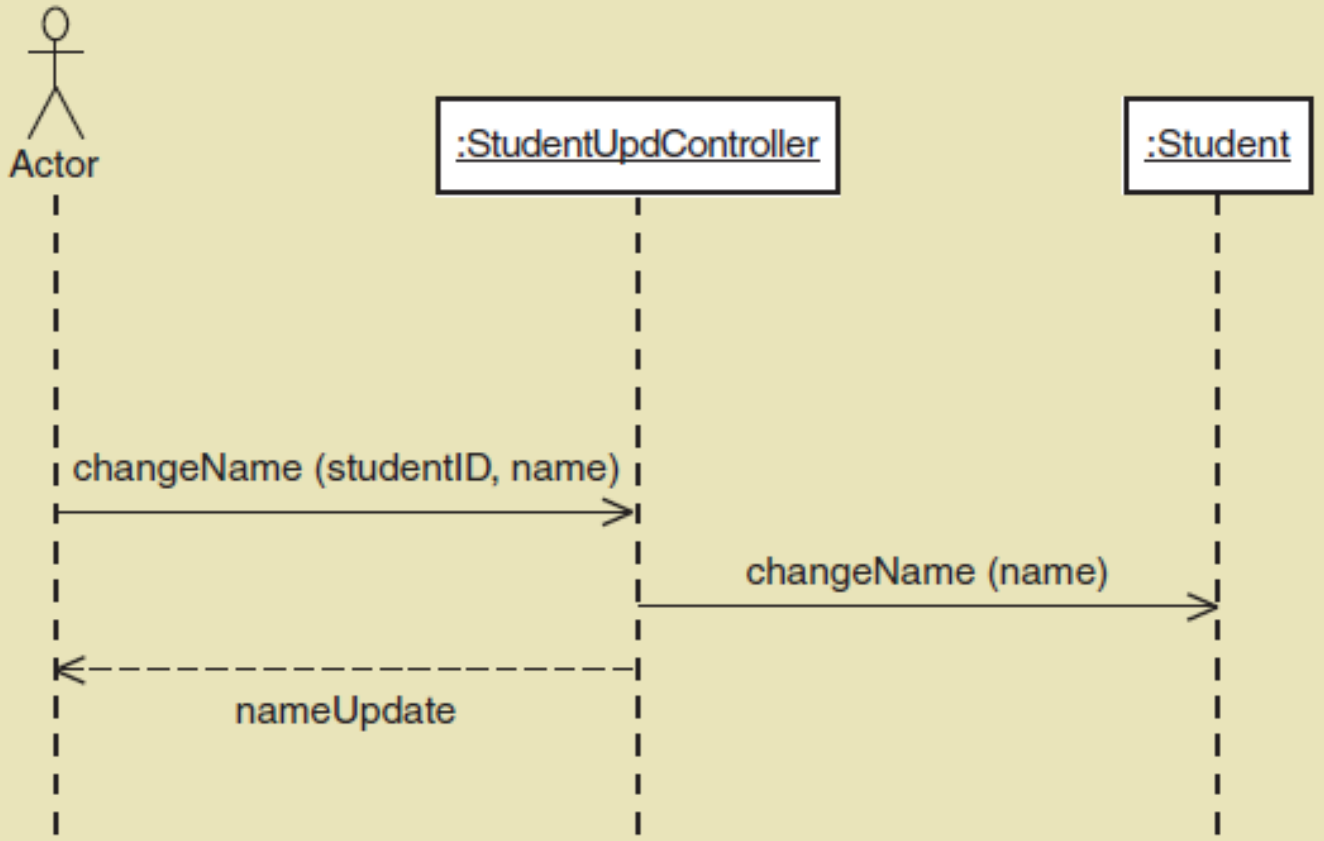
Use Case Realization

- Design and Implement Use Case by Use Case
  - Sequence Diagram—extended for system sequence diagram adding a controller and the domain classes
  - Design Class Diagram—extended from the domain model class diagram and updated from sequence diagram
    - Messages to an object become methods of the design class
  - Class Definition—written in the chosen code for the controller and the design classes
  - UI Classes—forms or pages are added to handle user interface between actor and the controller
  - Data Access Classes—are added to handle domain layer requests to get or save data to the database

Detailed Design

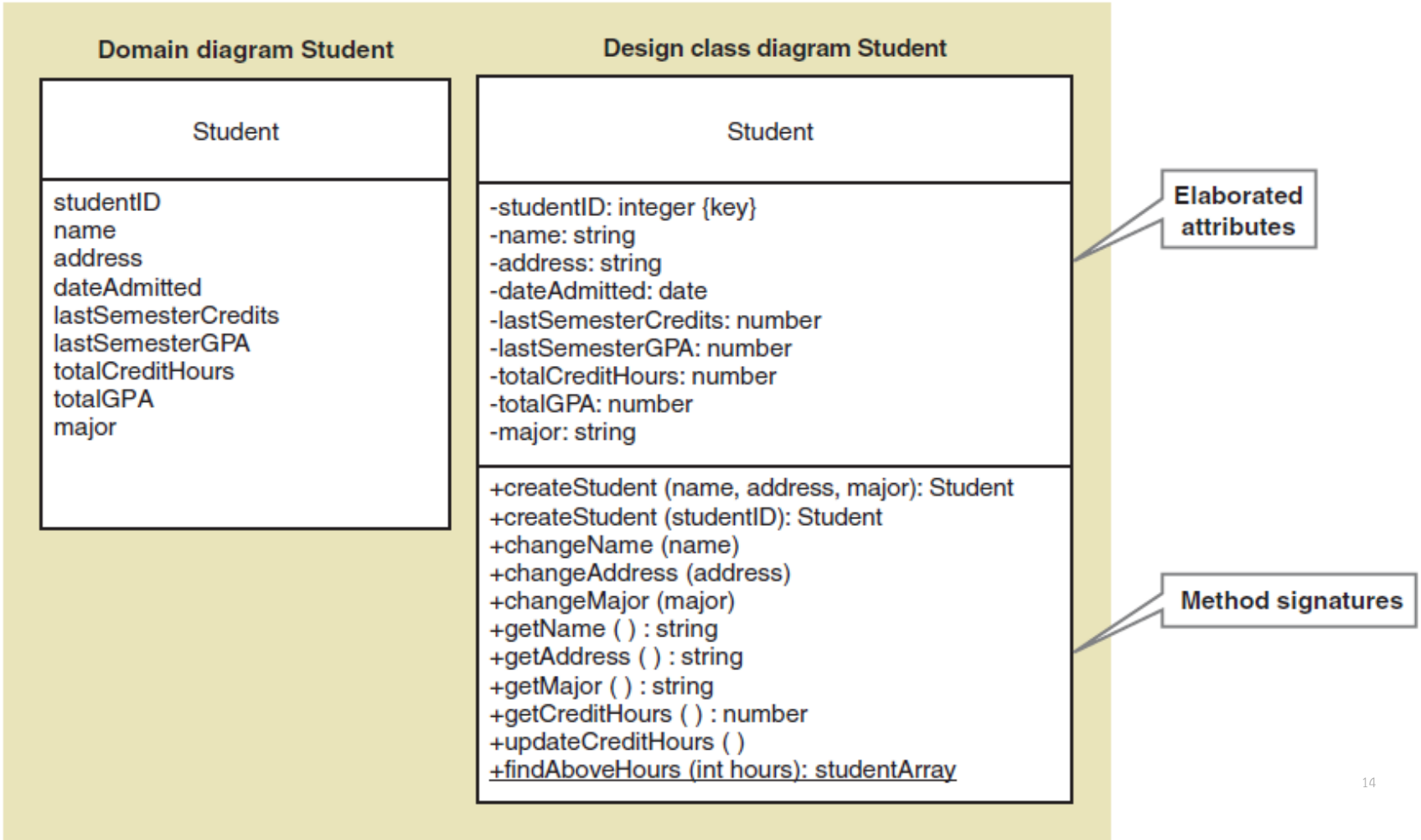
Sequence Diagram Example

- Use case *Update student name*



Design Classes in Detailed Design

- Elaborate attributes—visibility, type, properties
- Add methods and complete signatures



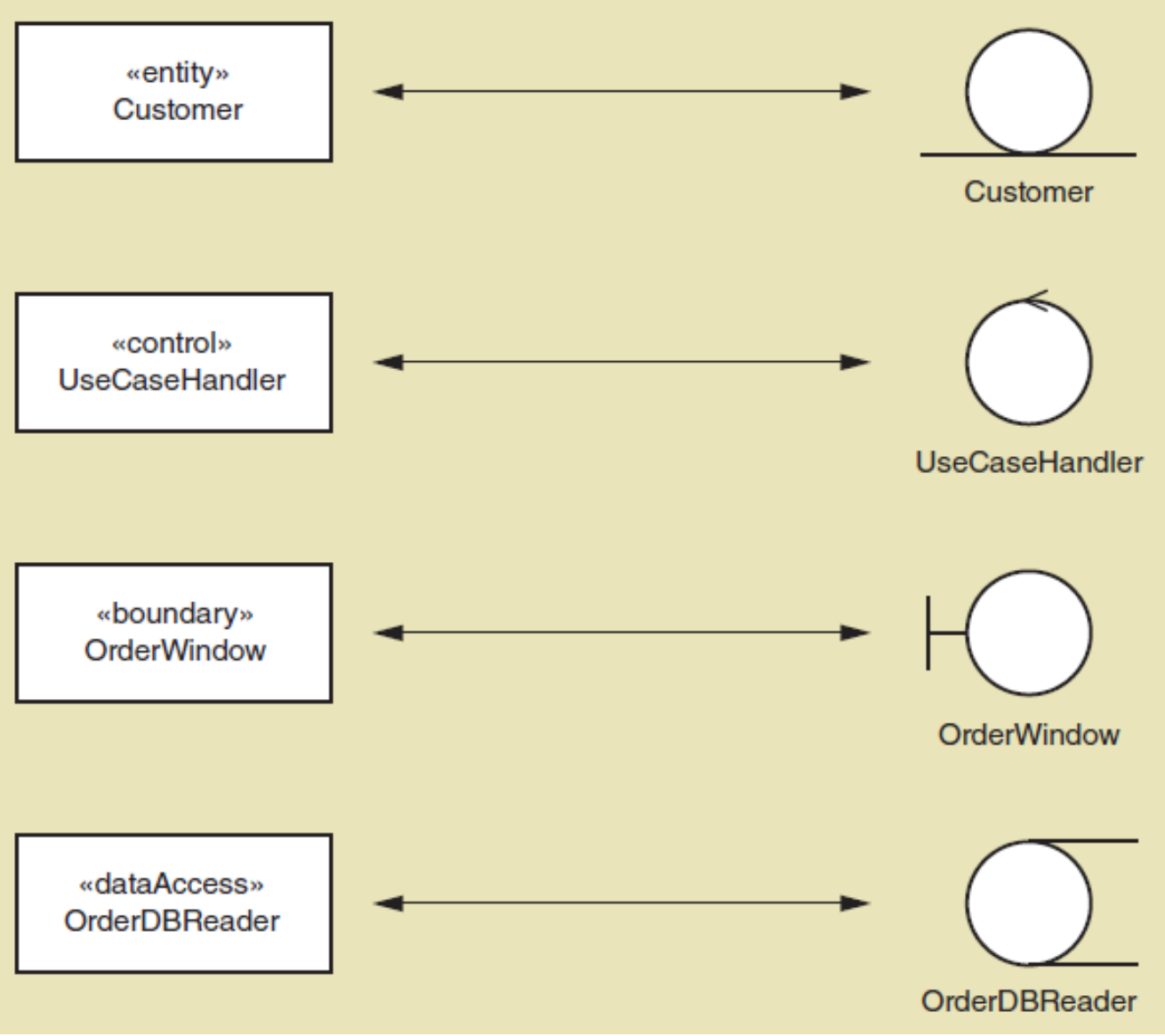
Write the Code for the Design Class to Implement

```
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
	}
}
```

Design Class Diagrams

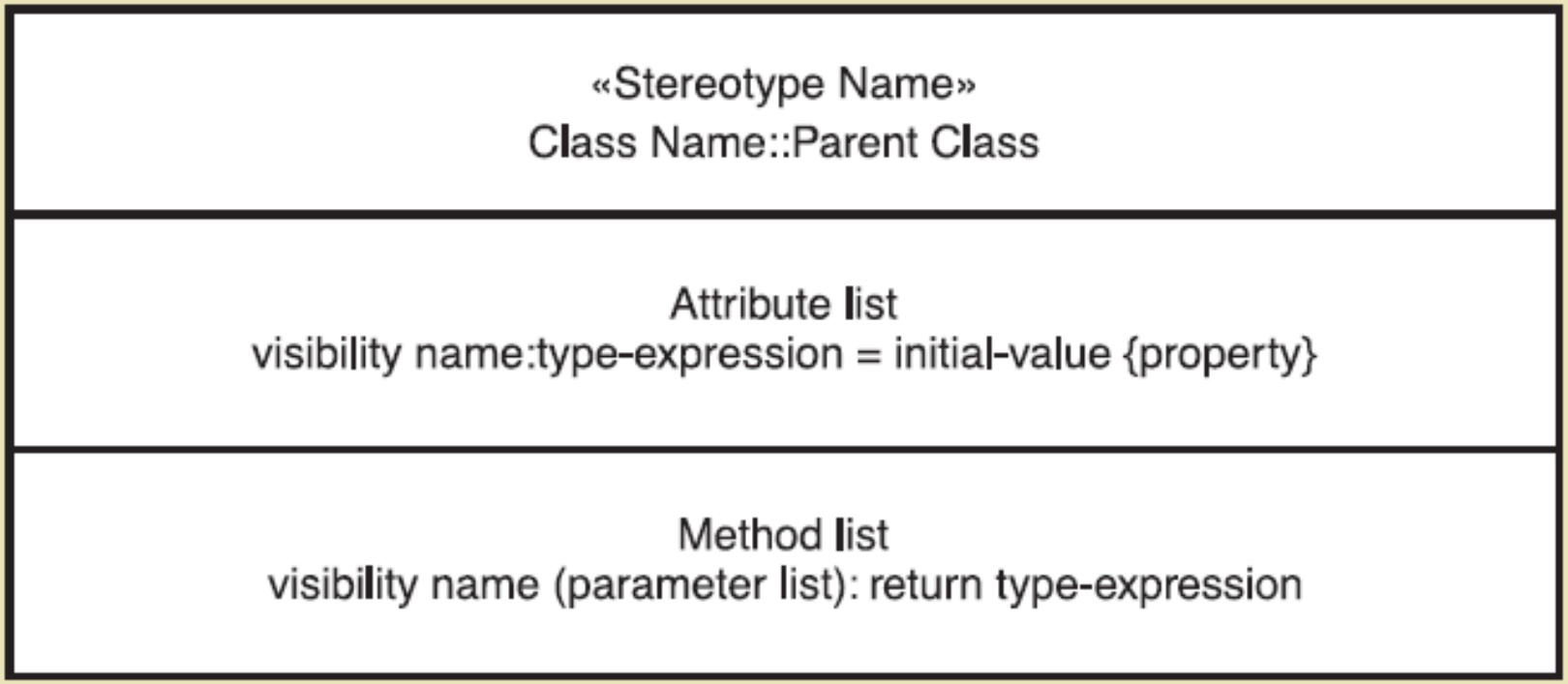
- **stereotype** a way of categorizing a model element by its characteristics, indicated by guillemots (<< >>)
- **persistent class** an class whose objects exist after a system is shut down (data remembered)
- **entity class** a design identifier for a problem domain class (usually persistent)
- **boundary class or view class** a class that exists on a system's automation boundary, such as an input window form or Web page
- **control class** a class that mediates between boundary classes and entity classes, acting as a switchboard between the view layer and domain layer
- **data access class** a class that is used to retrieve data from and send data to a database

Class Stereotypes in UML



Notation for a Design Class

- Syntax for Name, Attributes, and Methods



Notation for Design Classes

- Attributes
  - Visibility—indicates (+ or -) whether an attribute can be accessed ***directly*** by another object. Usually ***private*** (-) not public (+)
  - Attribute name—Lower case camelback notation
  - Type expression—class, string, integer, double, date
  - Initial value—if applicable the default value
  - Property—if applicable, such as {key}
  - Examples:

    -accountNo: String {key}

    -startingJobCode: integer = 01

Notation for Design Classes

- Methods
  - Visibility—indicates (+ or -) whether an method can be invoked by another object. Usually ***public*** (+), can be private if invoked within class like a subroutine
  - Method name—Lower case camelback, verb-noun
  - Parameters—variables passed to a method
  - Return type—the type of the data returned
  - Examples:

    +setName(fName, lName) : void (void is usually let off)

    +getName(): string (what is returned is a string)

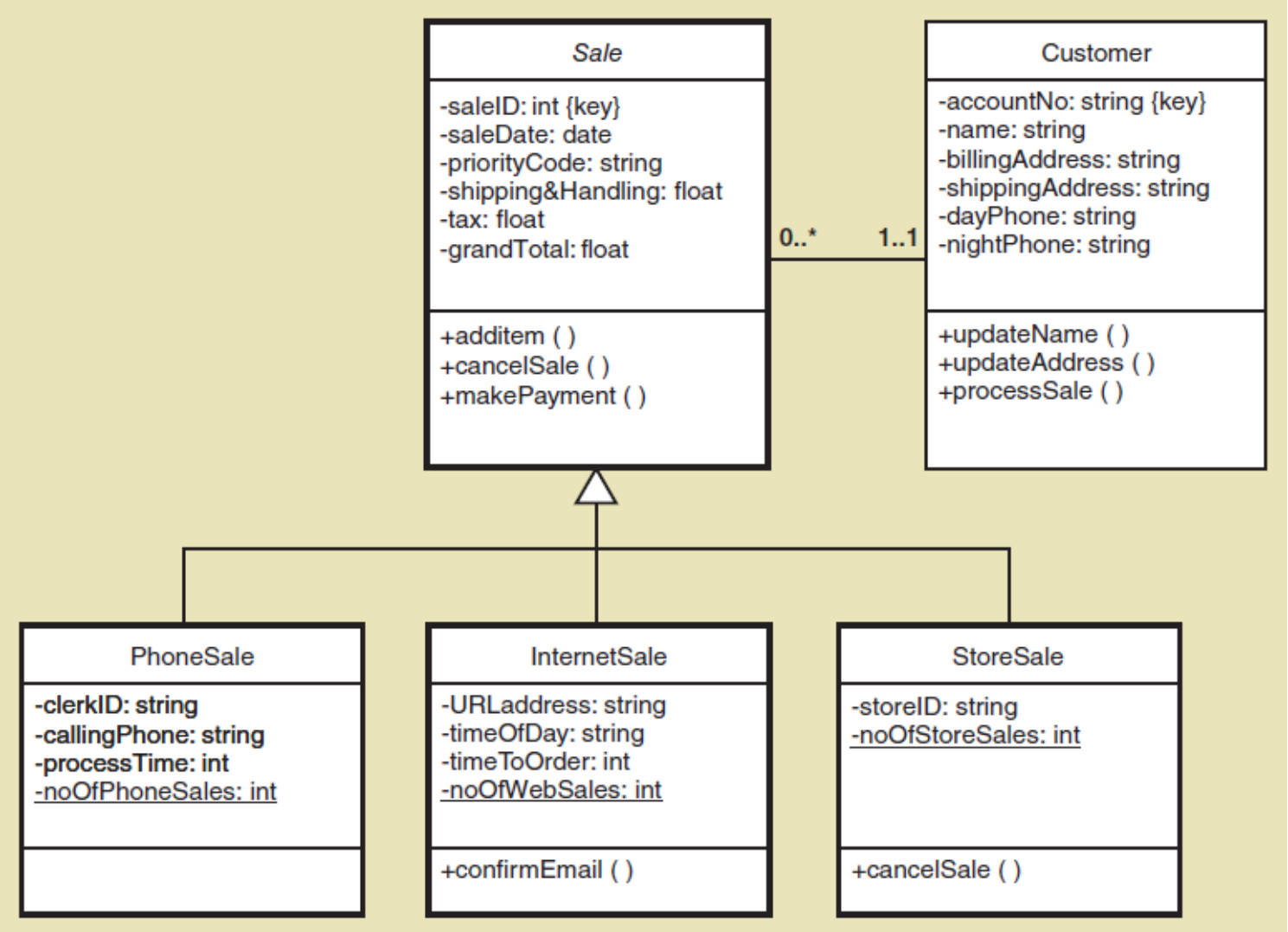
    -checkValidity(date) : int (assuming int is a returned code)

Notation for Design Classes

- Class level method—applies to class rather than objects of class (aka static method). Underline it.
  - +findStudentsAboveHours(hours): Array
  - +getNumberOfCustomers(): Integer
- Class level attribute—applies to the class rather than an object (aka static attribute). Underline it.
  - -noOfPhoneSales: int
- Abstract class– class that can't be instantiated.
  - Only for inheritance. Name in *Italics*.
- Concrete class—class that can be instantiated.

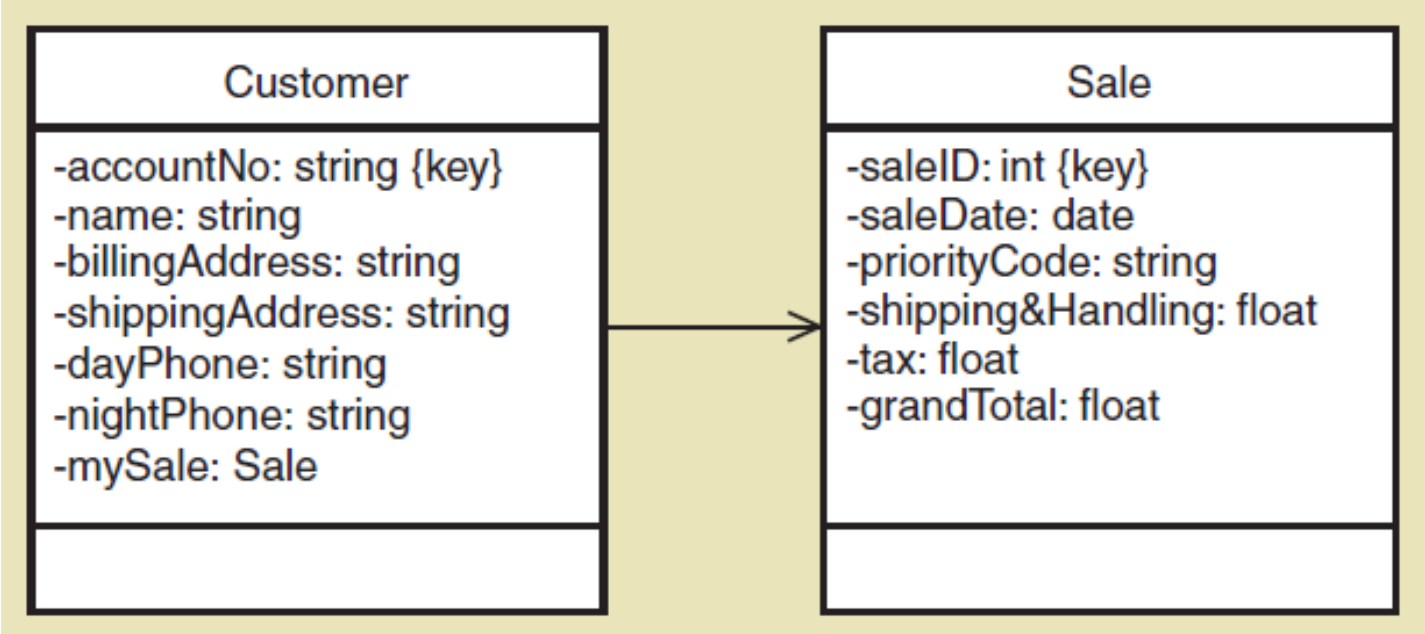
Notation for Design Classes

method arguments and return types not shown



Notation for Design Classes

- Navigation Visibility
  - The ability of one object to view and interact with another object
  - Accomplished by adding an object reference variable to a class.
  - Shown as an arrow head on the association line—customer can find and interact with sale because it has mySale reference variable



Navigation Visibility Guidelines

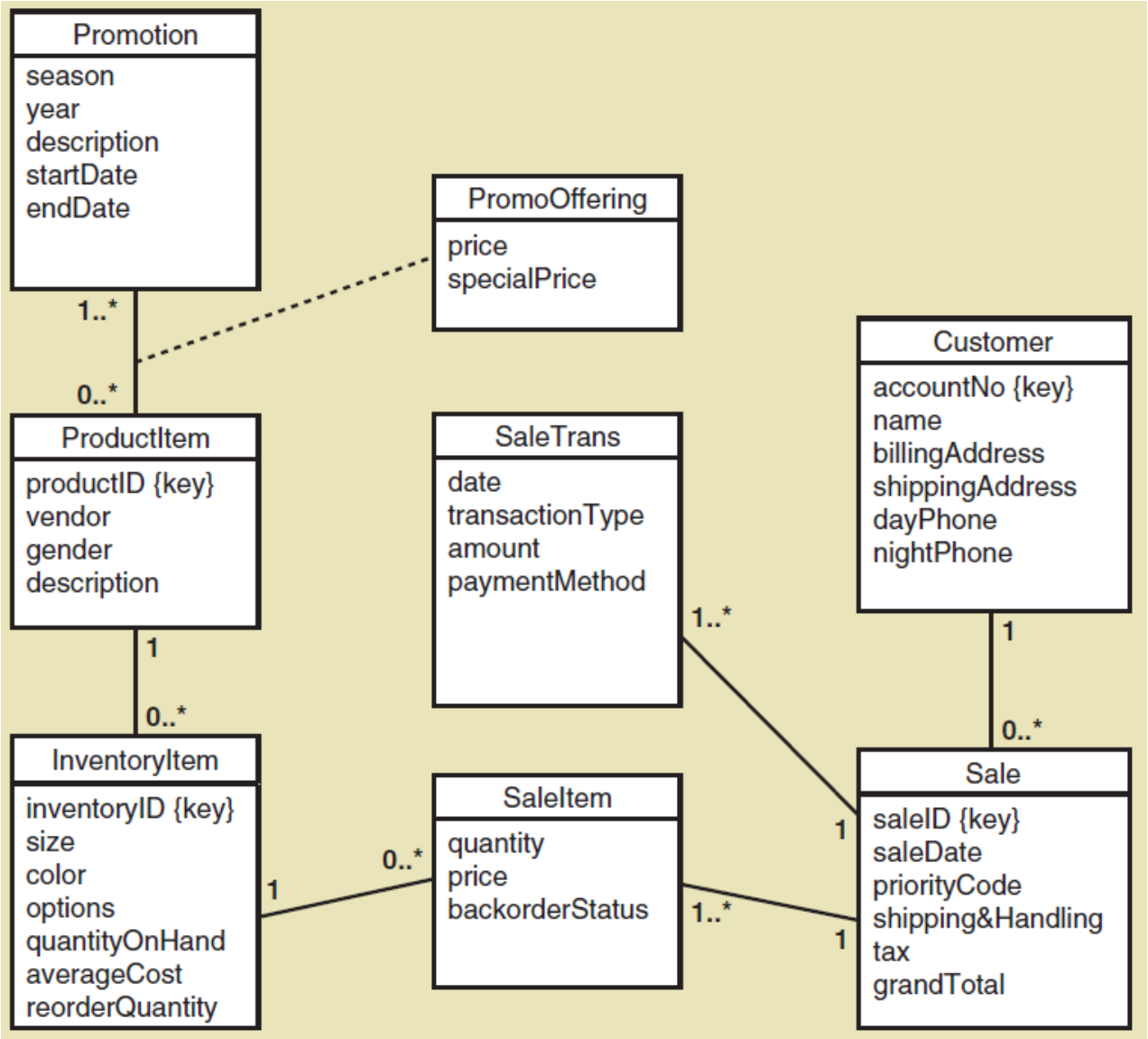
- One-to-many associations that indicate a superior/subordinate relationship are usually navigated from the superior to the subordinate
- Mandatory associations, in which objects in one class can’t exist without objects of another class, are usually navigated from the more independent class to the dependent
- When an object needs information from another object, a navigation arrow might be required
- Navigation arrows may be bidirectional.

First Cut Design Class Diagram

- Proceed use case by use case, adding to the diagram
- Pick the domain classes that are involved in the use case (see preconditions and post conditions for ideas)
- Add a controller class to be in charge of the use case
- Determine the initial navigation visibility requirements using the guidelines and add to diagram
- Elaborate the attributes of each class with visibility and type
- Note that often the associations and multiplicity are removed from the design class diagram as in text to emphasize navigation, but they are often left on

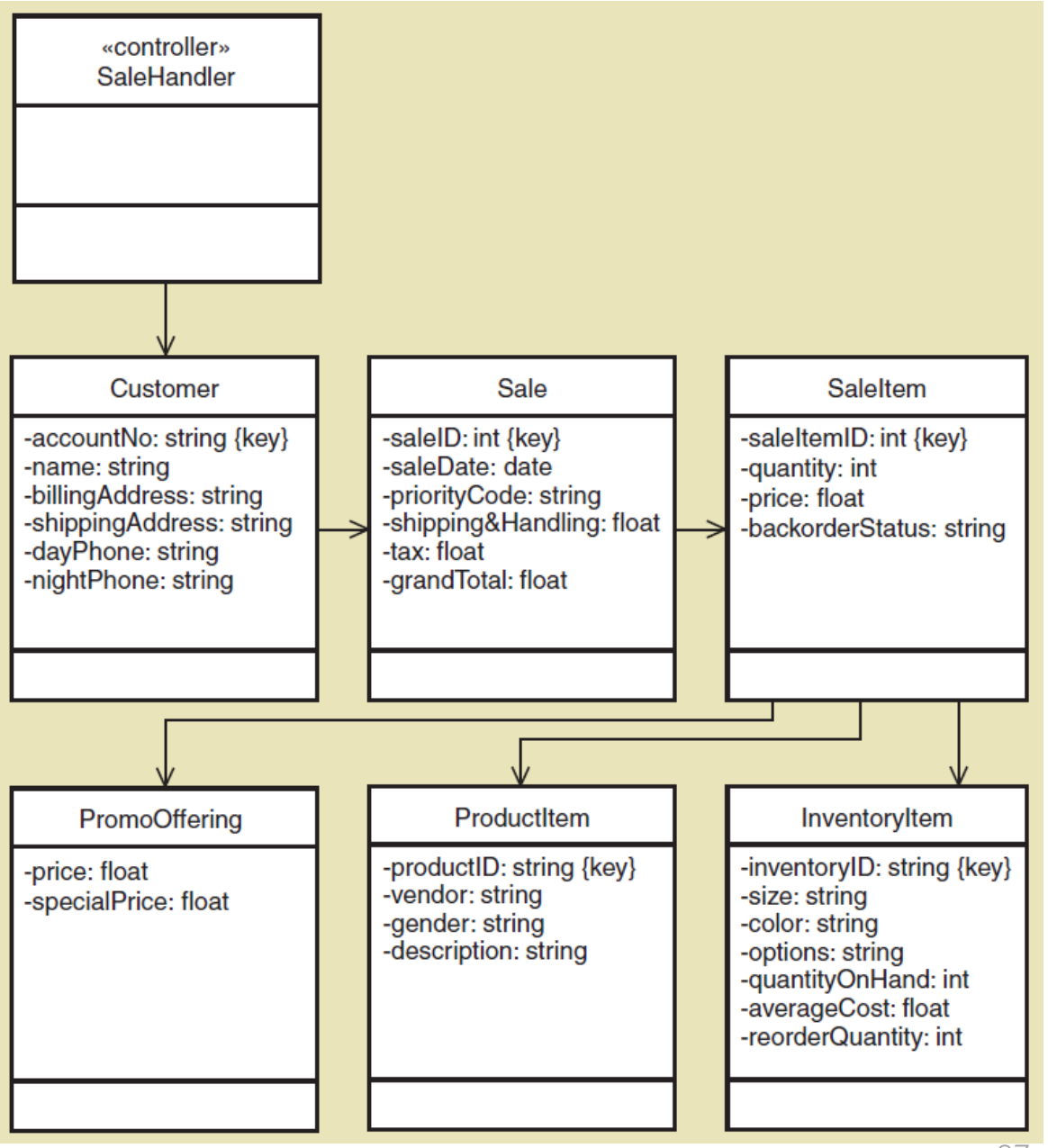
Start with Domain Class Diagram

RMO Sales Subsystem



Create First Cut Design Class Diagram

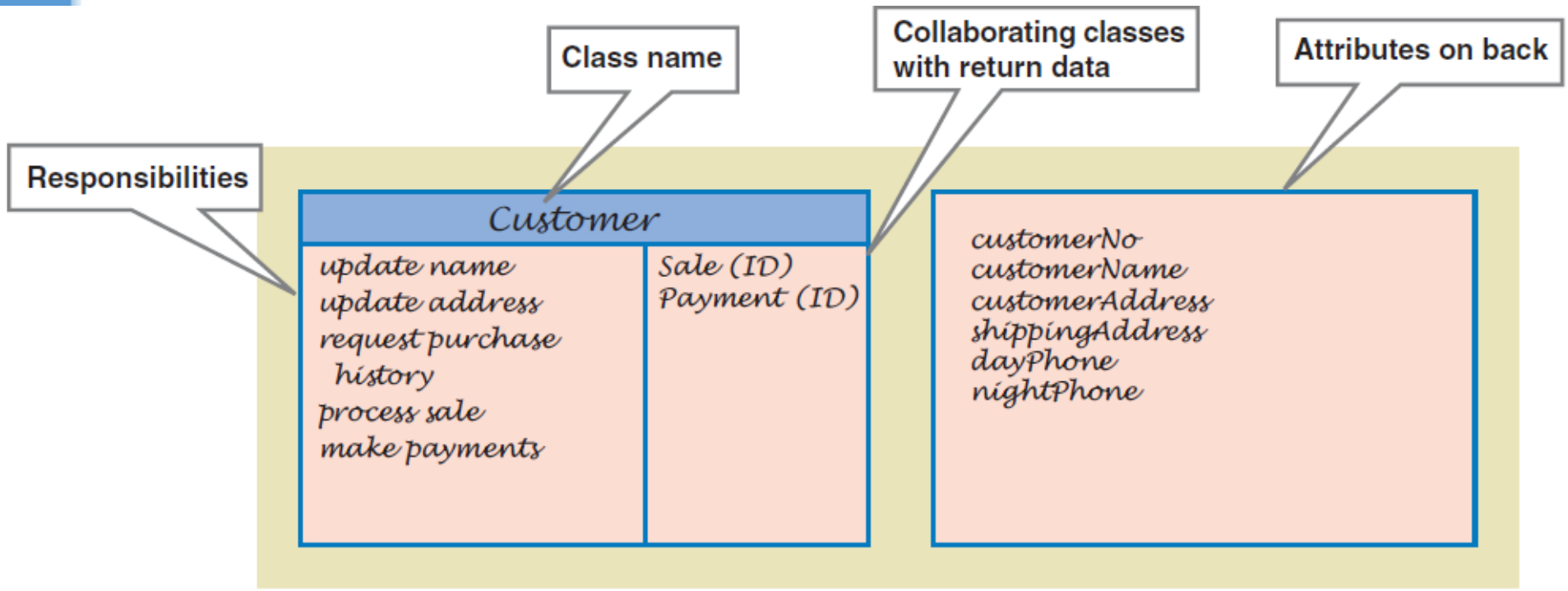
Use Case *Create phone sale* with controller added



Designing With CRC Cards

- CRC Cards—Classes, Responsibilities, Collaboration Cards
- OO design is about assigning Responsibilities to Classes for how they Collaborate to accomplish a use case
- Usually a manual process done in a brainstorming session
  - 3 X 5 note cards
  - One card per class
  - Front has responsibilities and collaborations
  - Back has attributes needed

Example of CRC Card



CRC Cards Procedure

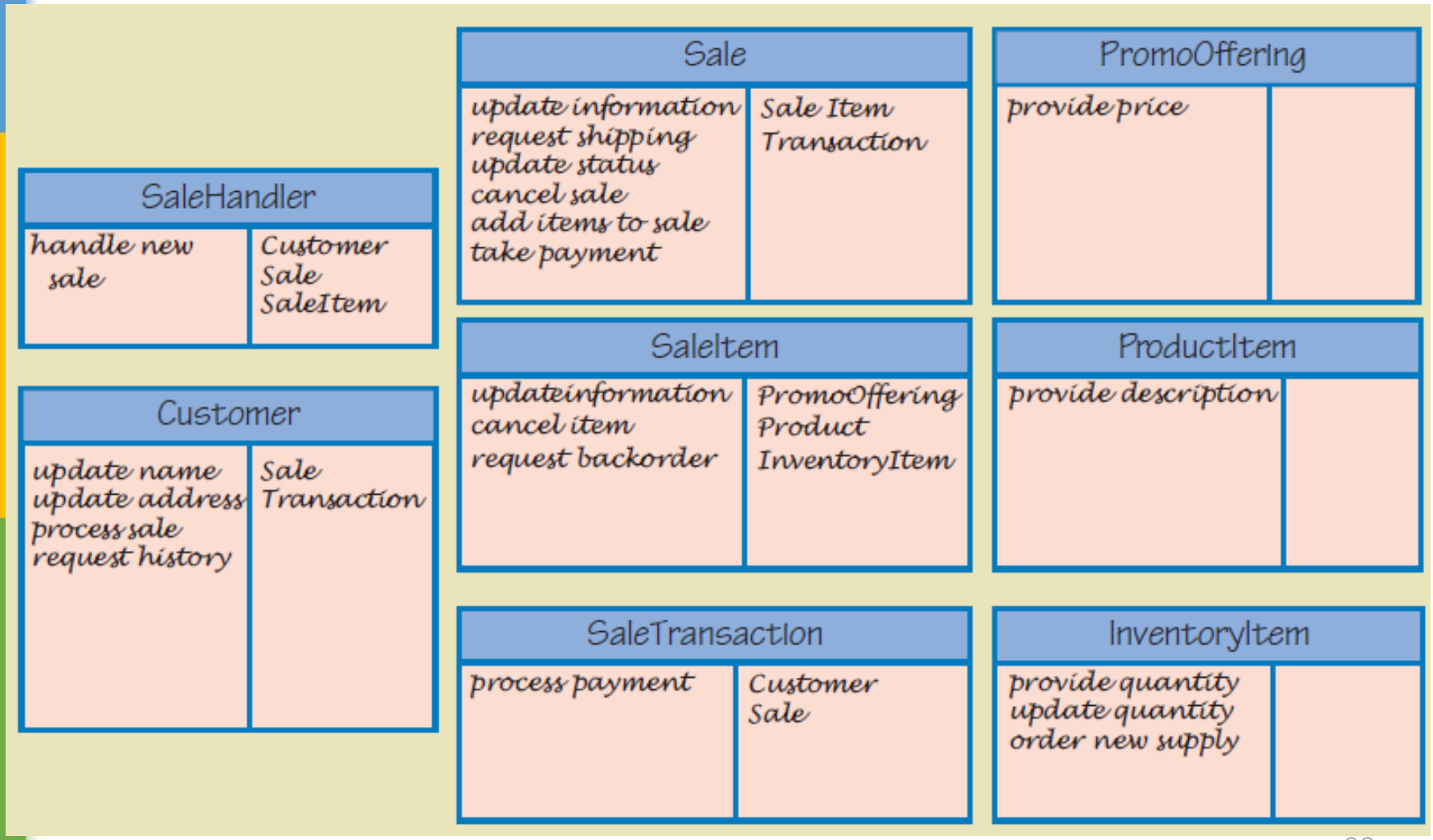
- Because the process is to design, or realize, a single use case, start with a set of unused CRC cards. Add a controller class (Controller design pattern).
- Identify a problem domain class that has primary responsibility for this use case that will receive the first message from the use case controller. For example, a Customer object for new sale.
- Use the first cut design class diagram to identify other classes that must collaborate with the primary object class to complete the use case.
- Have use case descriptions and SSDs handy

CRC Cards Procedure (continued)

- Start with the class that gets the first message from the controller. Name the responsibility and write it on card.
- Now ask what this first class needs to carry out the responsibility. Assign other classes responsibilities to satisfy each need. Write responsibilities on those cards.
- Sometimes different designers play the role of each class, acting out the use case by verbally sending messages to each other demonstrating responsibilities
- Add collaborators to cards showing which collaborate with which. Add attributes to back when data is used
- Eventually, user interface classes or even data access classes can be added

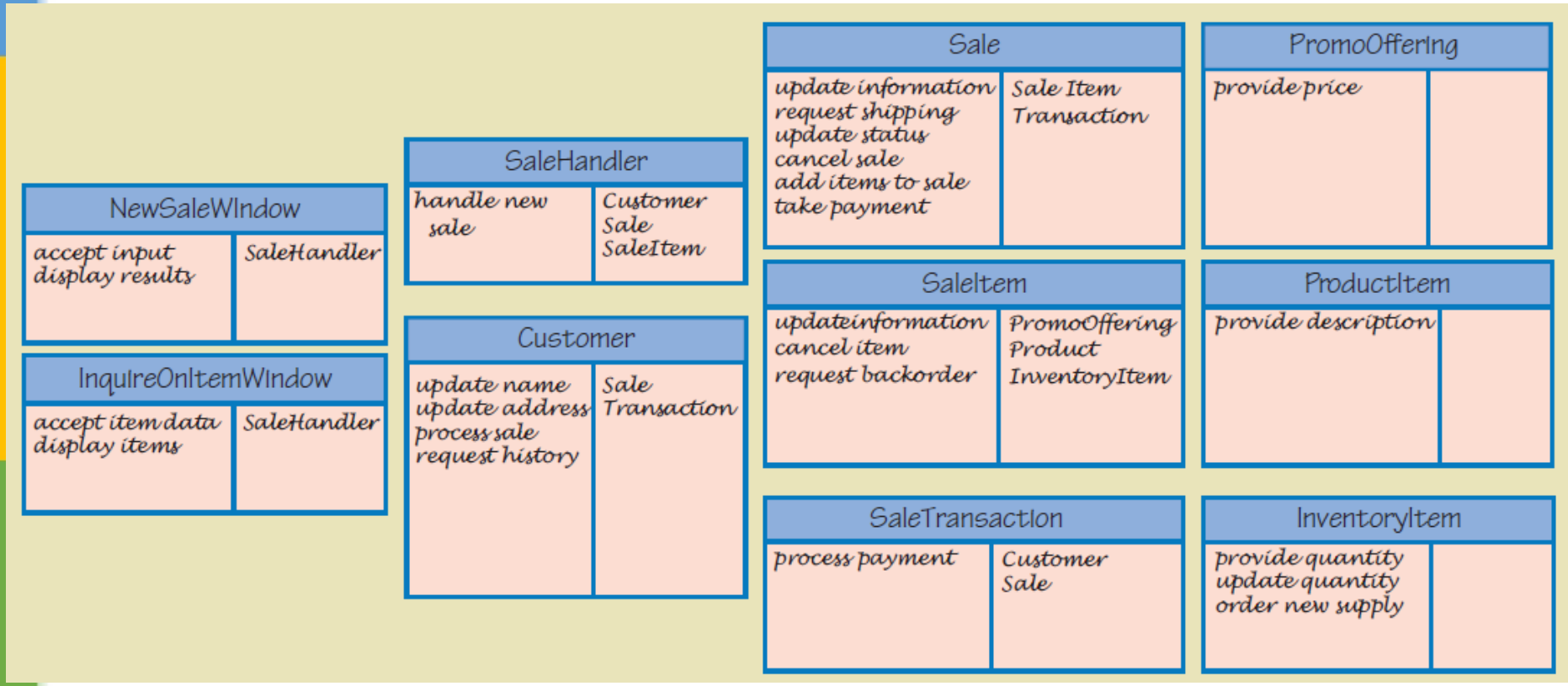
CRC Cards Results

Several Use Cases



CRC Cards Results

Adding In User Interface Layer

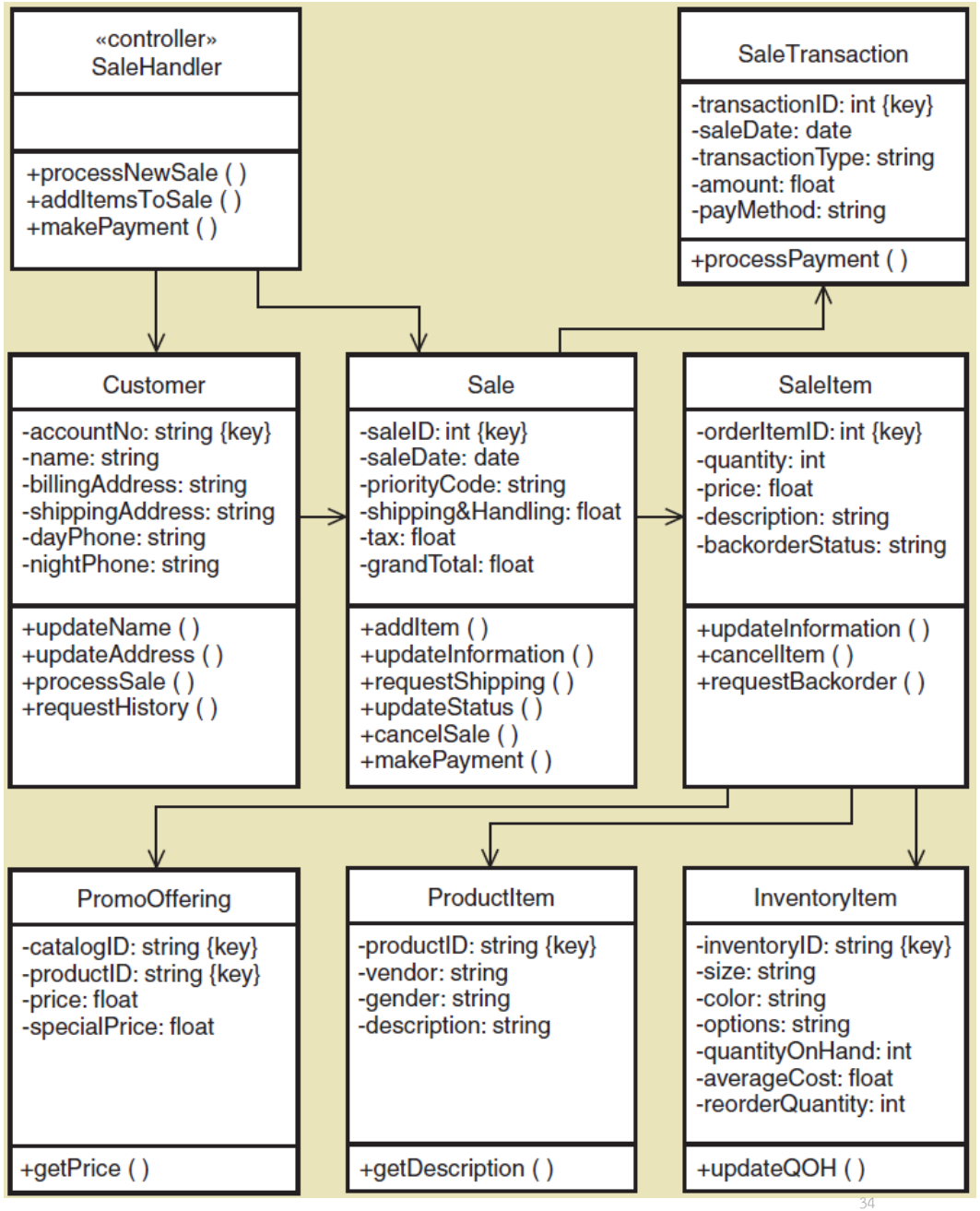


CRC Cards

Update design class diagram based on CRC results

Responsibilities become methods

Arguments and return types not yet added



Fundamental Design Principles

- Coupling
  - A quantitative measure of how closely related classes are linked (tightly or loosely coupled)
  - Two classes are tightly coupled of there are lots of associations with another class
  - Two classes are tightly coupled if there are lots of messages to another class
  - It is best to have classes that are **loosely coupled**
  - If deciding between two alternative designs, choose the one where overall coupling is less

Fundamental Design Principles

- Cohesion
  - A quantitative measure of the focus or unity of purpose within a single class (high or low cohesiveness
  - One class has high cohesiveness if all of its responsibilities are consistent and make sense for purpose of the class (a customer carries out responsibilities that naturally apply to customers)
  - One class has low cohesiveness if its responsibilities are broad or makeshift
  - It is best to have classes that are **highly cohesive**
  - If deciding between two alternative designs, choose the one where overall cohesiveness is high
  - **loosely coupled and highly cohesive** คือพยายามเขียนโปแกรมให้สัมพันธ์กันภายในคลาสให้มากที่สุด และพยายามลดความสัมพันธ์ระหว่าง class ให้น้อยที่สุด

Fundamental Design Principles

- Protection from Variations
  - A design principle that states parts of a system unlikely to change are separated (protected) from those that will surely change
  - Separate user interface forms and pages that are likely to change from application logic
  - Put database connection and SQL logic that is likely to change in a separate classes from application logic
  - Use adaptor classes that are likely to change when interfacing with other systems
  - If deciding between two alternative designs, choose the one where there is protection from variations

Fundamental Design Principles

- Indirection
  - A design principle that states an intermediate class is placed between two classes to decouple them but still link them
  - A controller class between UI classes and problem domain classes is an example
  - Supports low coupling
  - Indirection is used to support security by directing messages to an intermediate class as in a firewall
  - If deciding between two alternative designs, choose the one where indirection reduces coupling or provides greater security

Fundamental Design Principles

- Object Responsibility
  - A design principle that states objects are responsible for carrying out system processing
  - A fundamental assumption of OO design and programming
  - Responsibilities include “knowing” and “doing”
  - Objects know about other objects (associations) and they know about their attribute values. Objects know how to carry out methods, do what they are asked to do.
  - Note that CRC cards and the design in the next chapter involve assigning responsibilities to classes to carry out a use case.
  - If deciding between two alternative designs, choose the one where objects are assigned responsibilities to collaborate to complete tasks (don’t think procedurally).

Question