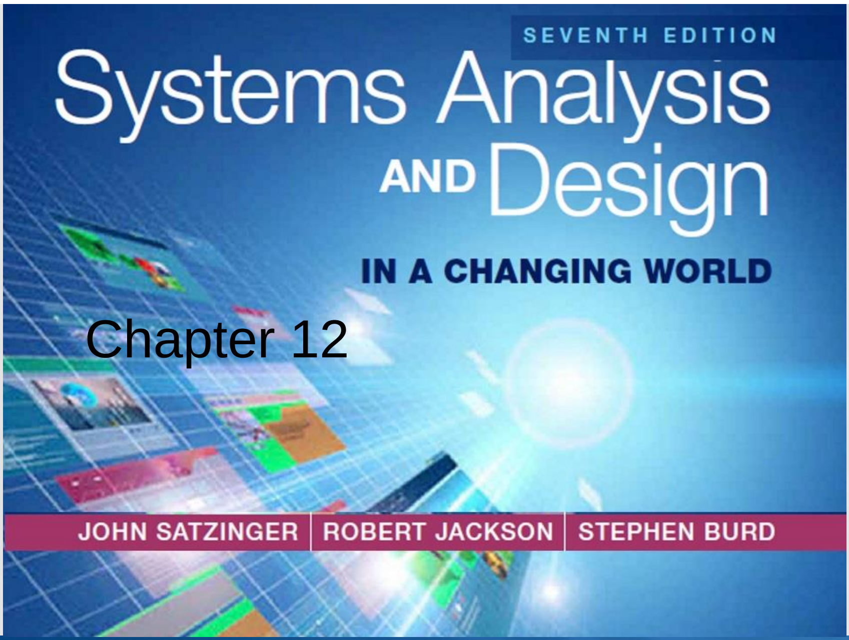


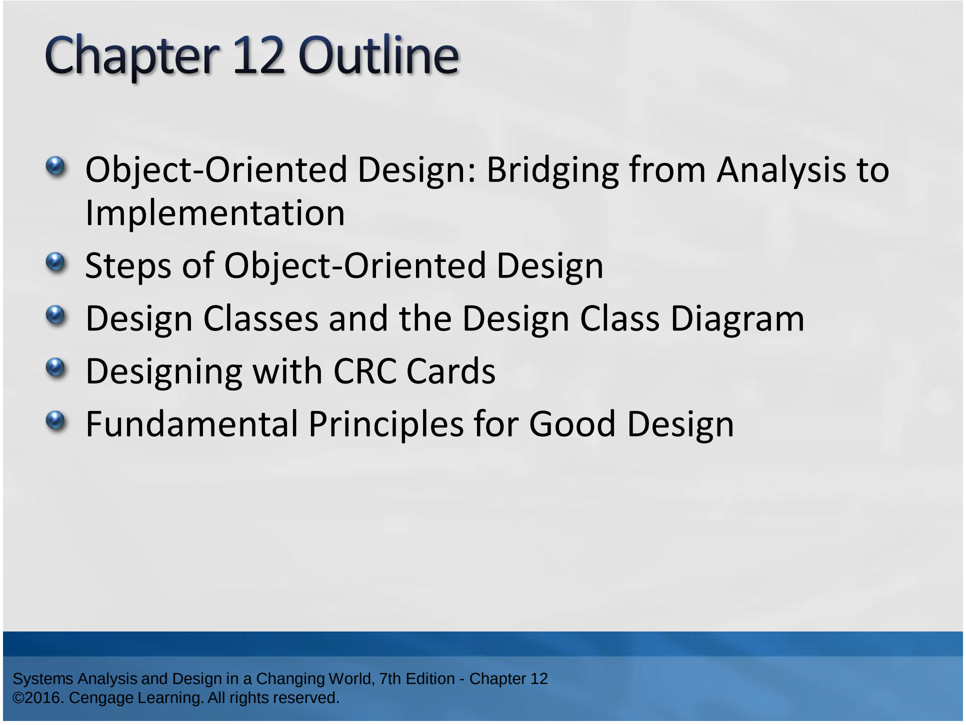
Chapter 12

Object-Oriented Design: Fundamentals

Chapter 12

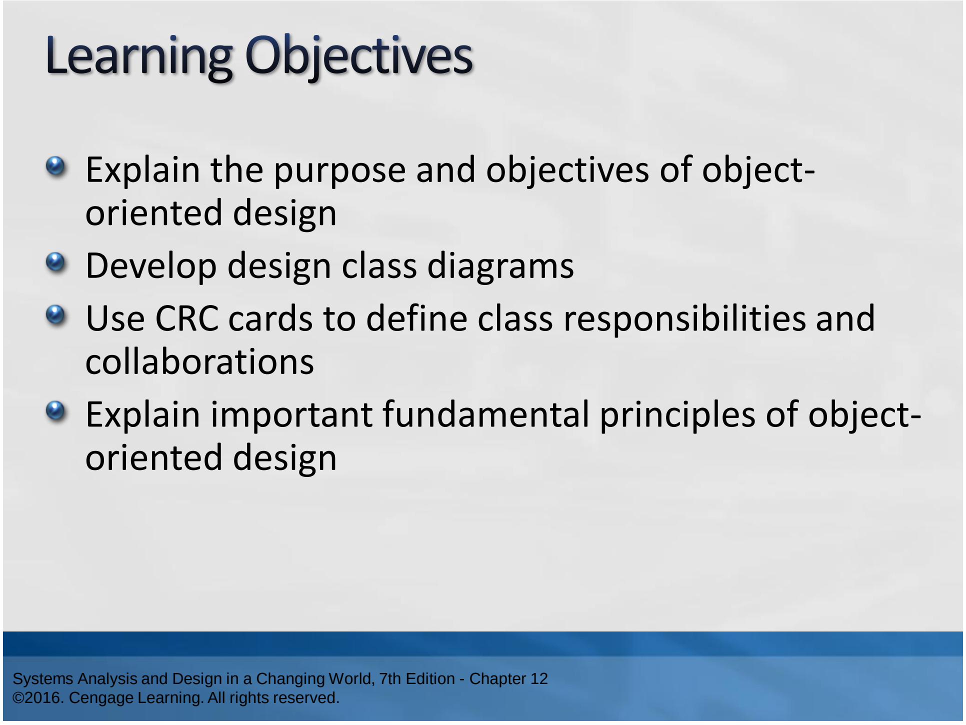
Systems Analysis and Design
in a Changing World 7th Ed

Satzinger, Jackson & Burd



Chapter 12 Outline

- Object-Oriented Design: Bridging from Analysis to Implementation
- Steps of Object-Oriented Design
- Design Classes and the Design Class Diagram
- Designing with CRC Cards
- Fundamental Principles for Good Design



Learning Objectives

- Explain the purpose and objectives of object-oriented design
- Develop design class diagrams
- Use CRC cards to define class responsibilities and collaborations
- Explain important fundamental principles of object-oriented design

Overview

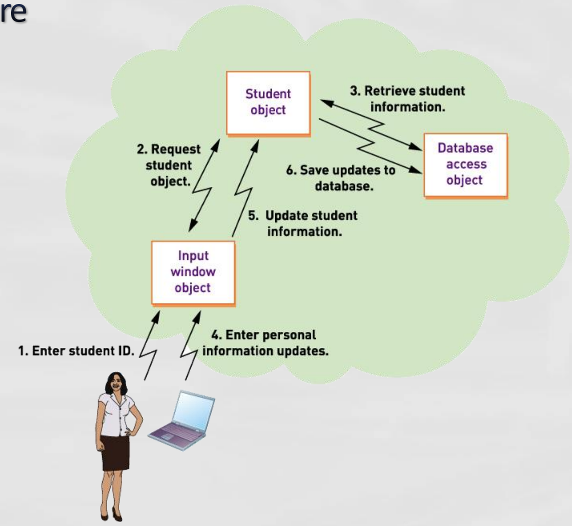
- This chapter and the next focus on designing software for the new system, at both the architectural and detailed level design
- Design models are based on the requirements models learned in Chapters 3, 4, and 5
- The steps of object-oriented design are explained
- The main model discussed is the design class diagram
- In this chapter, the CRC Cards technique is used to design the OO software
- The chapter finishes with fundamental principles of good OO design

OO Design:
Bridging from Analysis to Implementation

- OO Design: Process by which a set of detailed OO design models are built to be used for coding
- Strength of OO is requirements models from Chapters 3, 4, and 5 are extended to design models. No reinventing the wheel
- Design models are created in parallel to actual coding/implementation with iterative SDLC
- Agile approach says create models only if they are necessary. Simple detailed aspects don’t need a design model before coding

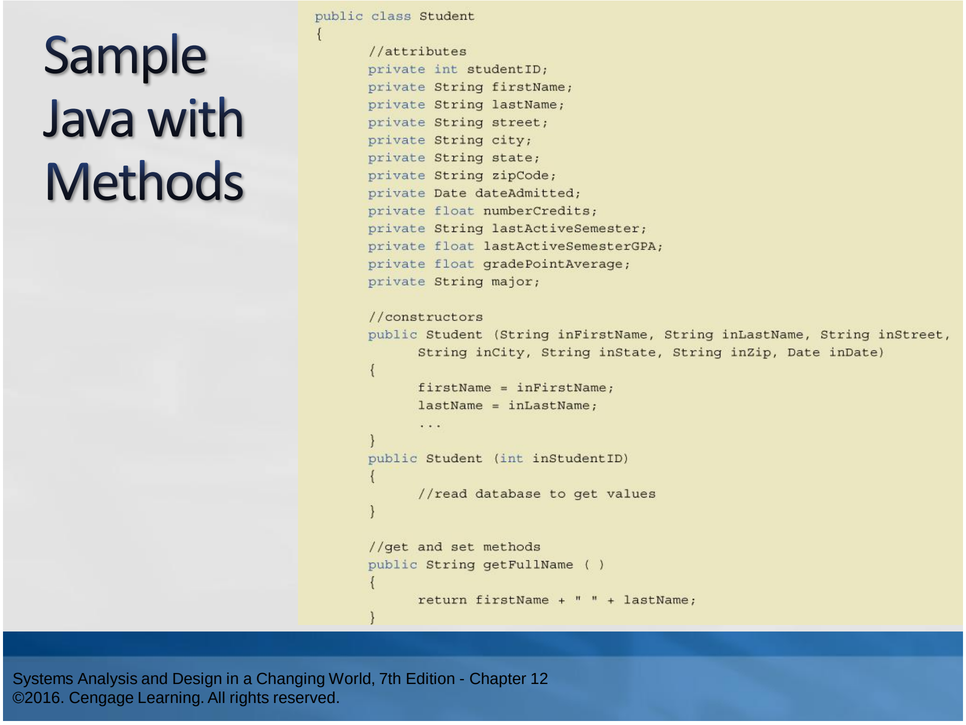
Object-Oriented Program Flow

Three Layer Architecture


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

Object-Oriented Program Flow

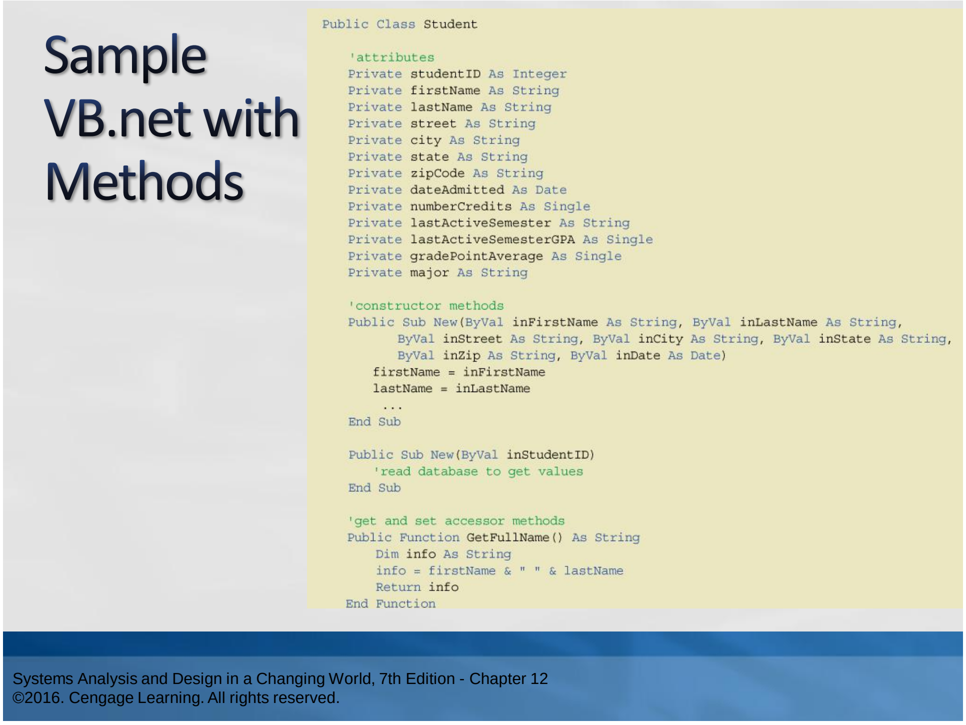
- Instantiation
  - Creation of an object in memory based on the template provided by the class
- Method
  - The function executed within an object when invoked by a message request (method call)



Sample Java with Methods

```
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;
        }
```

Systems Analysis and Design in a Changing World, 7th Edition - Chapter 12
©2016. Cengage Learning. All rights reserved.

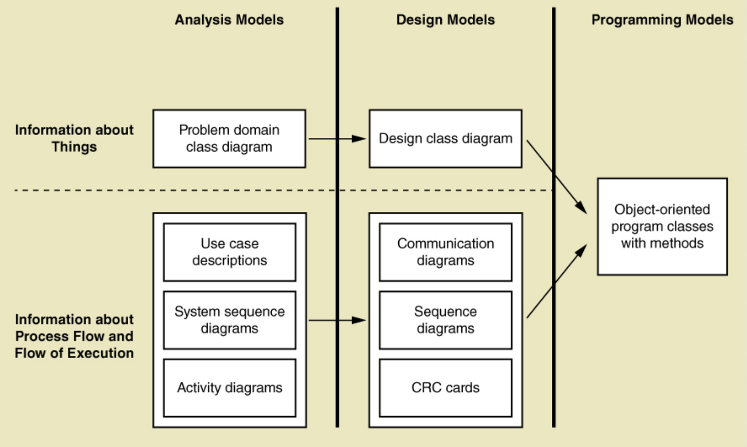


Sample VB.net with Methods

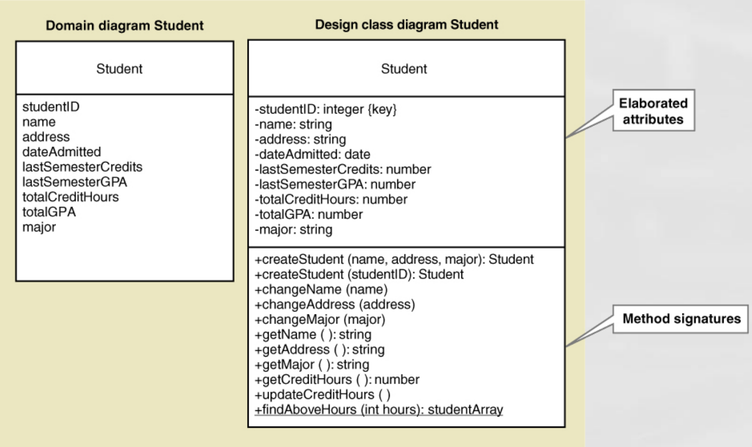
```
Public Class Student

    'attributes
    Private studentID As Integer
    Private firstName As String
    Private lastName As String
    Private street As String
    Private city As String
    Private state As String
    Private zipCode As String
    Private dateAdmitted As Date
    Private numberCredits As Single
    Private lastActiveSemester As String
    Private lastActiveSemesterGPA As Single
    Private gradePointAverage As Single
    Private major As String

    'constructor methods
    Public Sub New(ByVal inFirstName As String, ByVal inLastName As String,
           ByVal inStreet As String, ByVal inCity As String, ByVal inState As String,
           ByVal inZip As String, ByVal inDate As Date)
        firstName = inFirstName
        lastName = inLastName
         ...
    End Sub

    Public Sub New(ByVal inStudentID)
        'read database to get values
    End Sub

    'get and set accessor methods
    Public Function GetFullName() As String
        Dim info As String
        info = firstName & " " & lastName
        Return info
    End Function
```

Systems Analysis and Design in a Changing World, 7th Edition - Chapter 12
©2016. Cengage Learning. All rights reserved.

Analysis to Design to Implementation: Model Flow



Introduction to Design Models:
Class Diagram



Student

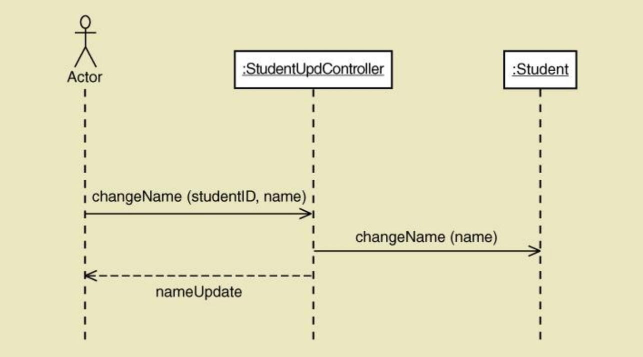
studentID
name
address
dateAdmitted
lastSemesterCredits
lastSemesterGPA
totalCreditHours
totalGPA
major

Student

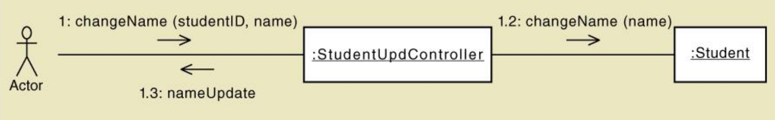
-studentID: integer {key}
-name: string
-address: string
-dateAdmitted: date
-lastSemesterCredits: number
-lastSemesterGPA: number
-totalCreditHours: number
-totalGPA: number
-major: string

+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

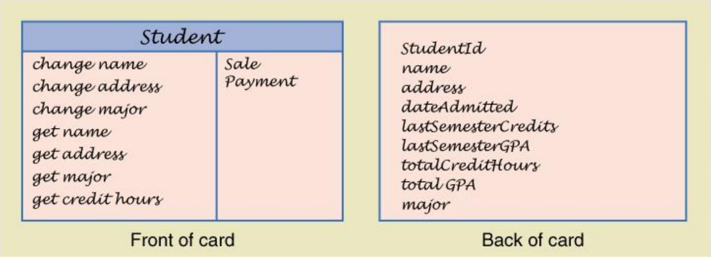
Introduction to Design Models:
Sequence Diagram



Introduction to Design Models:
Communication Diagram



Introduction to Design Models:
Class-Responsibility-Collaboration (CRC)



Student

change name
change address
change major
get name
get address
get major
get credit hours

Sale
Payment

Front of card

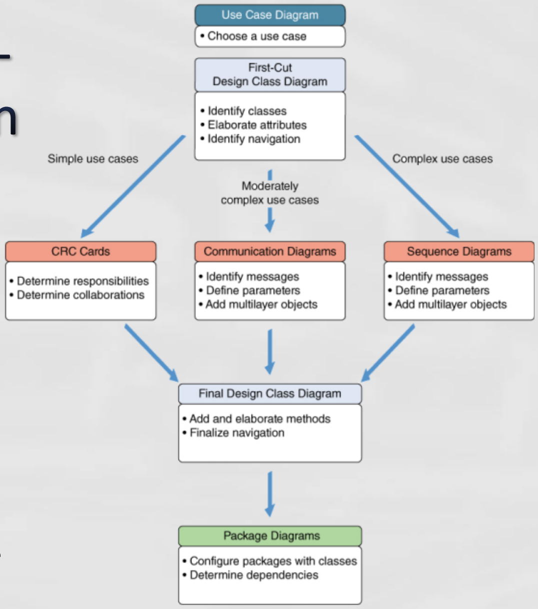
StudentId
name
address
dateAdmitted
lastSemesterCredits
lastSemesterGPA
totalCreditHours
total GPA
major

Back of card

Steps of Object-Oriented Design

- Object-oriented design
  - The process to identify the classes, their methods and the messages required for a use case
- Use case driven
  - Design is carried out use case by use case

Steps of Object-Oriented Design



- Three paths
  - Simple use case use CRC Cards
  - Medium use case use Communication Diagram
  - Complex use case use Sequence Diagram

Design Class Diagrams

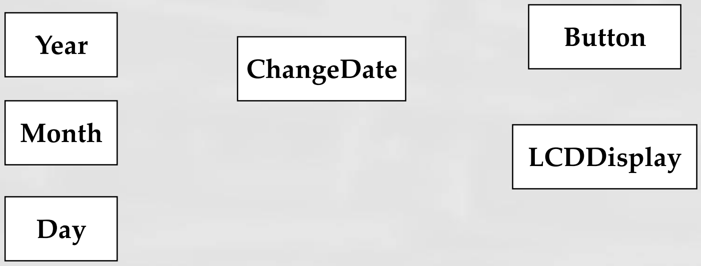
- **stereotype** a way of categorizing a model element by its characteristics, indicated by guillemots (<< >>)
- **persistent class** an class whose objects exist after a system is shut down (data remembered)
- **entity class** a design identifier for a problem domain class (usually persistent)
- **boundary class or view class** a class that exists on a system’s automation boundary, such as an input window form or Web page
- **controller 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

There are different types of Objects

- Entity Objects
  - Represent the persistent information tracked by the system (Application domain objects, also called "Business objects")
- Boundary Objects
  - Represent the interaction between the user and the system
- Control Objects
  - Represent the control tasks performed by the system.

Example: 2BWatch Modeling

To distinguish different object types in a model we can use the UML Stereotype mechanism

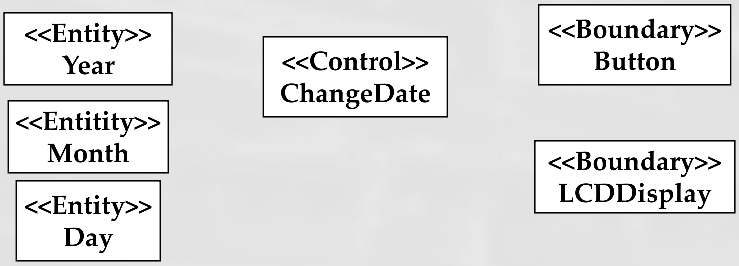


Entity Objects Control Object Boundary Objects

Naming Object Types in UML

- UML provides the stereotype mechanism to introduce new types of modeling elements
  - A stereotype is drawn as a name enclosed by angled double-quotes (“guillemets”) (<<, >>) and placed before the name of a UML element (class, method, attribute, ….)
  - Notation: <<String>>Name

Naming Object Types in UML



<<Entity>>
Year

<<Entitity>>
Month

<<Entity>>
Day

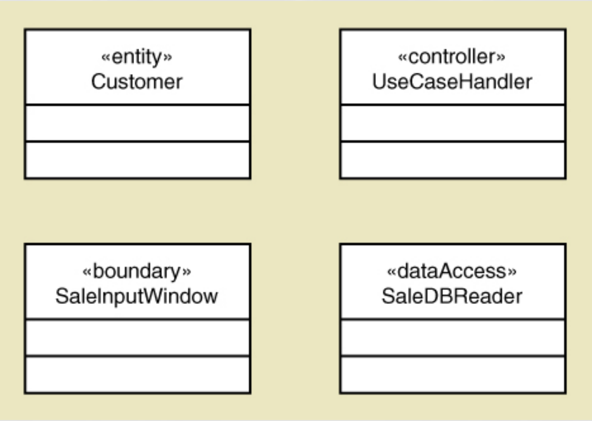
<<Control>>
ChangeDate

<<Boundary>>
Button

<<Boundary>>
LCDDisplay

Entity Object Control Object Boundary Object

Design Class Stereotypes



«entity»
Customer

«controller»
UseCaseHandler

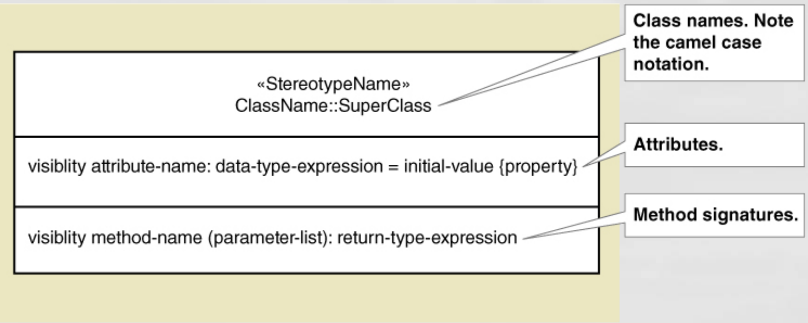
«boundary»
SaleInputWindow

«dataAccess»
SaleDBReader

Systems Analysis and Design in a Changing World, 7th Edition - Chapter 12

Notation for a Design Class

- Syntax for Name, Attributes, and Methods



«StereotypeName»
ClassName::SuperClass

visiblity attribute-name: data-type-expression = initial-value {property}

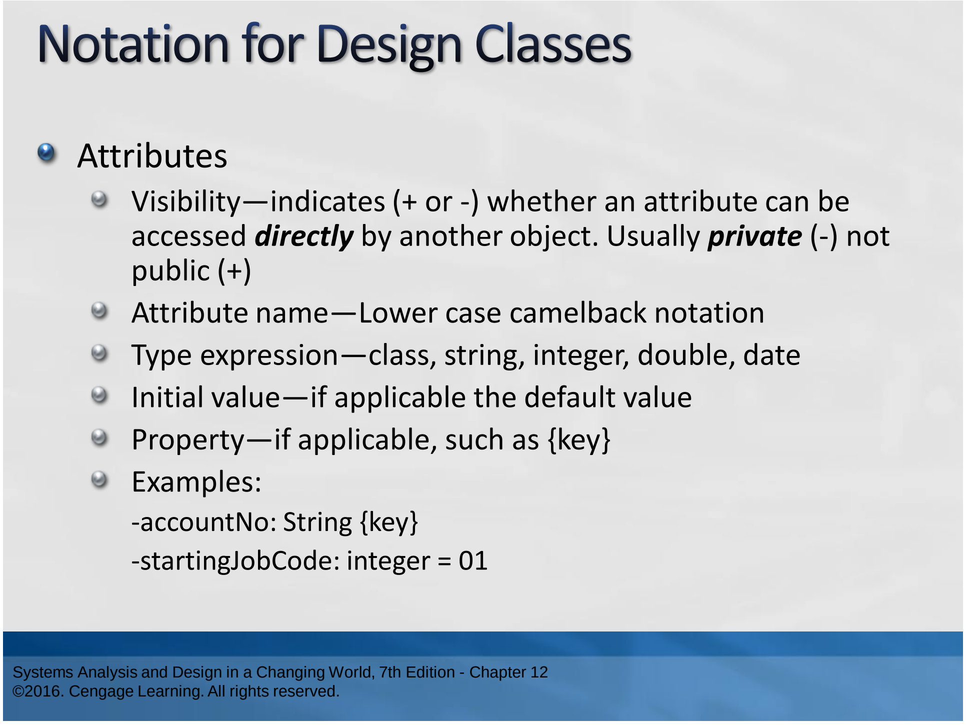
visiblity method-name (parameter-list): return-type-expression

Class names. Note the camel case notation.

Attributes.

Method signatures.

Systems Analysis and Design in a Changing World, 7th Edition - Chapter 12

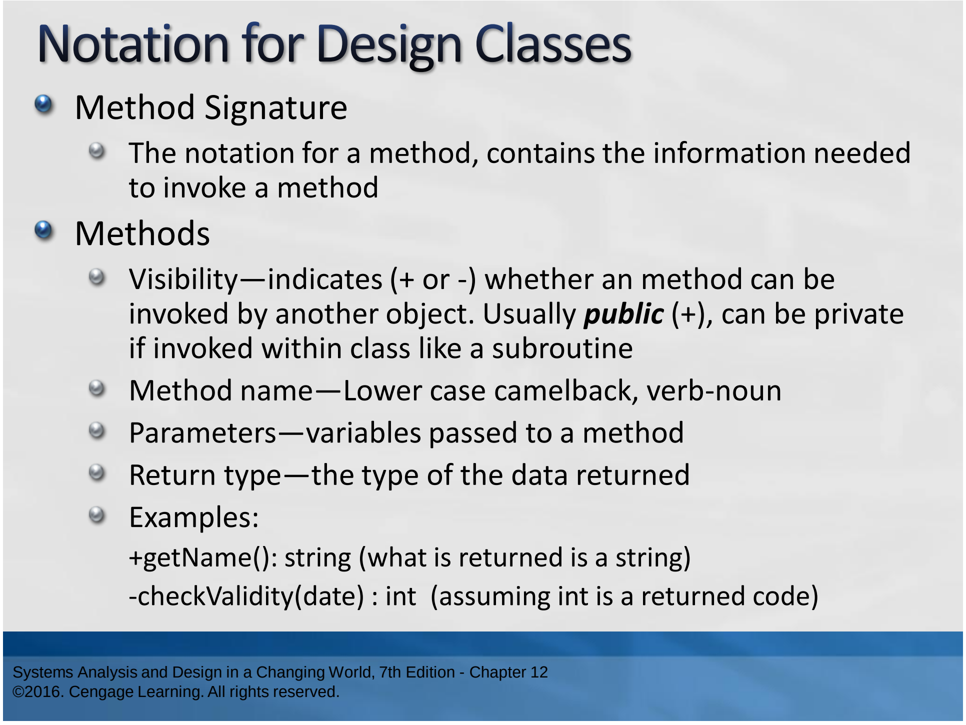


Notation for Design Classes

- 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

- Method Signature
  - The notation for a method, contains the information needed to invoke a method
- 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:

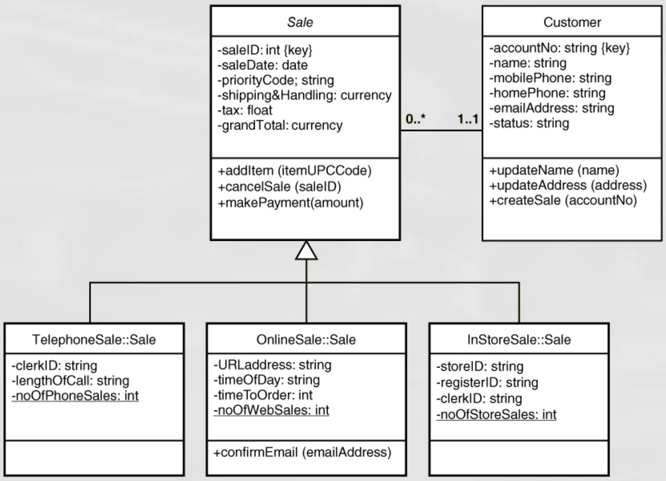
    +getName(): string (what is returned is a string)

    -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



Sale

-saleID: int {key}
-saleDate: date
-priorityCode; string
-shipping&Handling: currency
-tax: float
-grandTotal: currency

+addItem (itemUPCCode)
+cancelSale (saleID)
+makePayment(amount)

0..* 1..1

Customer

-accountNo: string {key}
-name: string
-mobilePhone: string
-homePhone: string
-emailAddress: string
-status: string

+updateName (name)
+updateAddress (address)
+createSale (accountNo)

TelephoneSale::Sale

-clerkID: string
-lengthOfCall: string
-noOfPhoneSales: int

OnlineSale::Sale

-URLaddress: string
-timeOfDay: string
-timeToOrder: int
-noOfWebSales: int

+confirmEmail (emailAddress)

InStoreSale::Sale

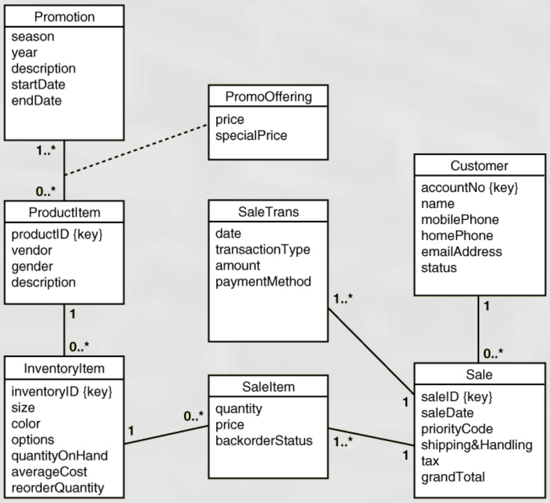
-storeID: string
-registerID: string
-clerkID: string
-noOfStoreSales: int

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



Promotion
season
year
description
startDate
endDate

PromoOffering
price
specialPrice

Customer
accountNo {key}
name
mobilePhone
homePhone
emailAddress
status

ProductItem
productID {key}
vendor
gender
description

SaleTrans
date
transactionType
amount
paymentMethod

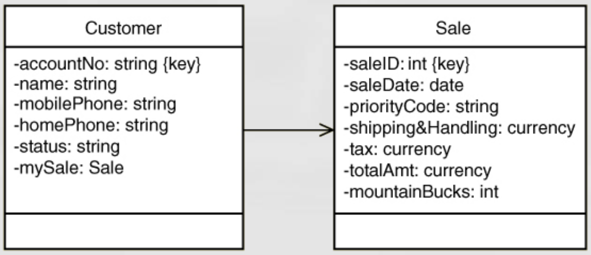
InventoryItem
inventoryID {key}
size
color
options
quantityOnHand
averageCost
reorderQuantity

SaleItem
quantity
price
backorderStatus

Sale
saleID {key}
saleDate
priorityCode
shipping&Handling
tax
grandTotal

Developing 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



Customer

-accountNo: string {key}
-name: string
-mobilePhone: string
-homePhone: string
-status: string
-mySale: Sale

Sale

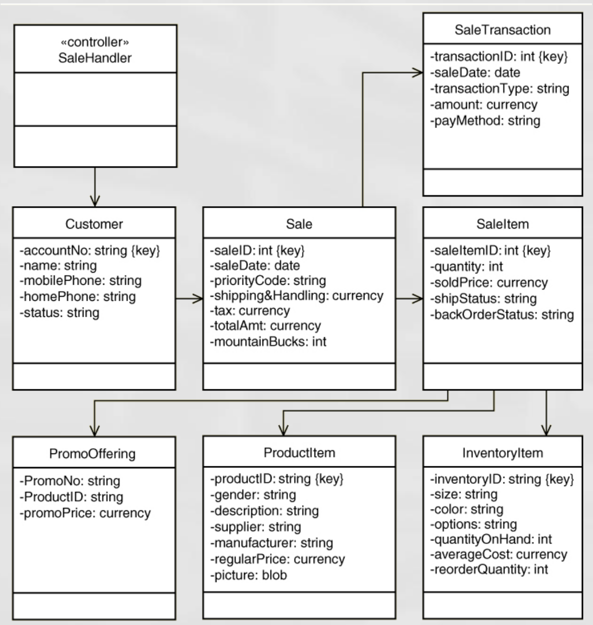
-saleID: int {key}
-saleDate: date
-priorityCode: string
-shipping&Handling: currency
-tax: currency
-totalAmt: currency
-mountainBucks: int

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.

Create First Cut Design Class Diagram

Use Case *Create telephone sale* with controller added



«controller»
SaleHandler

Customer
-accountNo: string {key}
-name: string
-mobilePhone: string
-homePhone: string
-status: string

Sale
-saleID: int {key}
-saleDate: date
-priorityCode: string
-shipping&Handling: currency
-tax: currency
-totalAmt: currency
-mountainBucks: int

SaleTransaction
-transactionID: int {key}
-saleDate: date
-transactionType: string
-amount: currency
-payMethod: string

SaleItem
-saleItemID: int {key}
-quantity: int
-soldPrice: currency
-shipStatus: string
-backOrderStatus: string

PromoOffering
-PromoNo: string
-ProductID: string
-promoPrice: currency

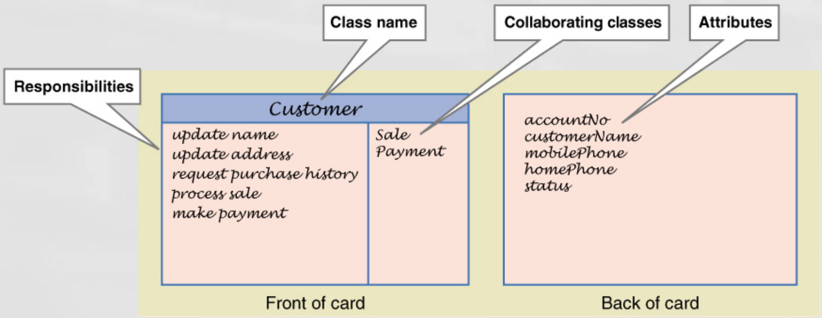
ProductItem
-productID: string {key}
-gender: string
-description: string
-supplier: string
-manufacturer: string
-regularPrice: currency
-picture: blob

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

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

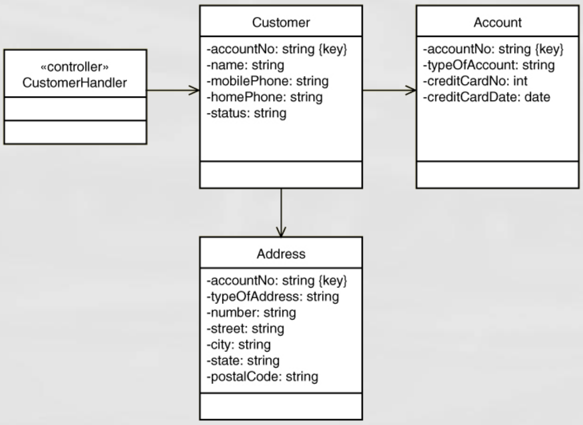
1. Because the process is to design, or realize, a single use case, start with a set of unused CRC cards. Add a controller class (Controller design pattern).
2. Identify a problem domain class that has primary responsibility for this use case that will receive the first message from the use case controller. For example, a Customer object for new sale.
3. Use the first cut design class diagram to identify other classes that must collaborate with the primary object class to complete the use case.
4. Add user-interface classes to identify inputs and outputs
5. Add any other required utility classes

CRC Cards Suggestions

- 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

Example: *Create customer Account*

- First-cut DCD



«controller»
CustomerHandler

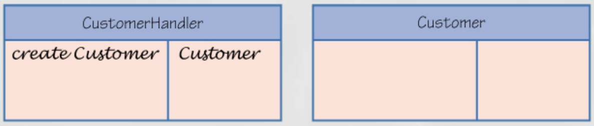
Customer
-accountNo: string {key}
-name: string
-mobilePhone: string
-homePhone: string
-status: string

Account
-accountNo: string {key}
-typeOfAccount: string
-creditCardNo: int
-creditCardDate: date

Address
-accountNo: string {key}
-typeOfAddress: string
-number: string
-street: string
-city: string
-state: string
-postalCode: string

Example: *Create customer account*

- Controller and primary domain class

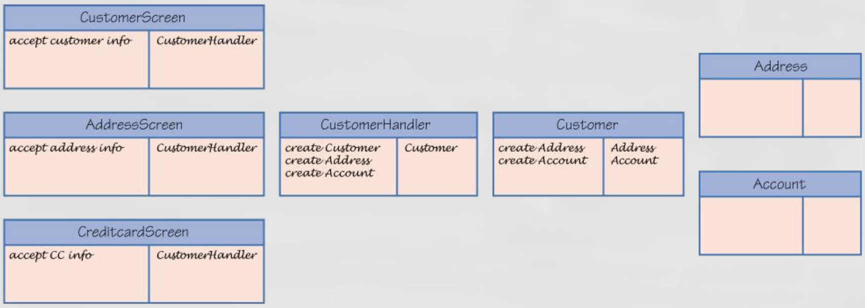


CustomerHandler	
create Customer	Customer

Customer	

Example: *Create customer account*

- Problem domain classes and user interface classes



CustomerScreen	
accept customer info	CustomerHandler

AddressScreen	
accept address info	CustomerHandler

CreditcardScreen	
accept CC info	CustomerHandler

CustomerHandler	
create Customer create Address create Account	Customer

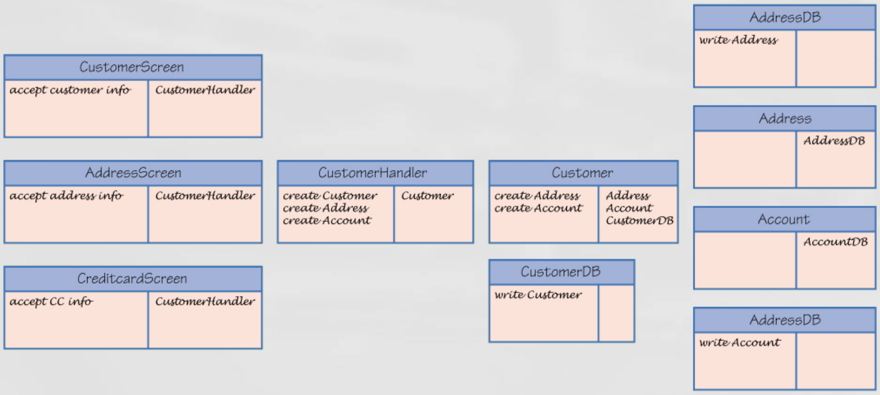
Customer	
create Address create Account	Address Account

Address	

Account	

Example: *Create customer account*

- Adding data access classes



CustomerScreen	
accept customer info	CustomerHandler

AddressScreen	
accept address info	CustomerHandler

CreditcardScreen	
accept CC info	CustomerHandler

CustomerHandler	
create Customer create Address create Account	Customer

Customer	
create Address create Account	Address Account CustomerDB

CustomerDB	
write Customer	

AddressDB	
write Address	

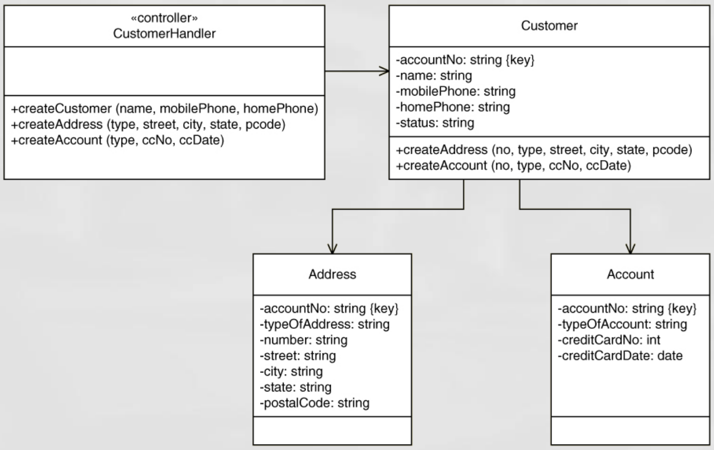
Address	
	AddressDB

Account	
	AccountDB

AddressDB	
write Account	

Example: *Create customer account*

- Final DCD with method signatures



«controller»
CustomerHandler

+createCustomer (name, mobilePhone, homePhone)
+createAddress (type, street, city, state, pcode)
+createAccount (type, ccNo, ccDate)

Customer

-accountNo: string {key}
-name: string
-mobilePhone: string
-homePhone: string
-status: string

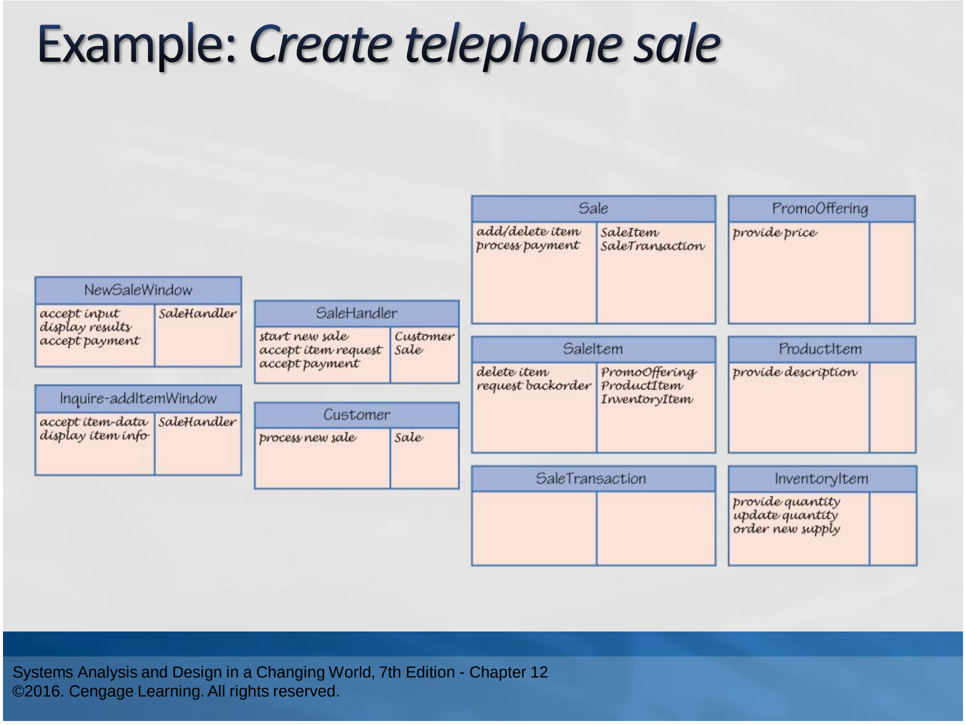
+createAddress (no, type, street, city, state, pcode)
+createAccount (no, type, ccNo, ccDate)

Address

-accountNo: string {key}
-typeOfAddress: string
-number: string
-street: string
-city: string
-state: string
-postalCode: string

Account

-accountNo: string {key}
-typeOfAccount: string
-creditCardNo: int
-creditCardDate: date



Example: *Create telephone sale*

NewSaleWindow
accept input
display results
accept payment
SaleHandler

Inquire-addItemWindow
accept item-data
display item info
SaleHandler

SaleHandler
start new sale
accept item request
accept payment
Customer
Sale

Customer
process new sale
Sale

Sale
add/delete item
process payment
SaleItem
SaleTransaction

PromoOffering
provide price

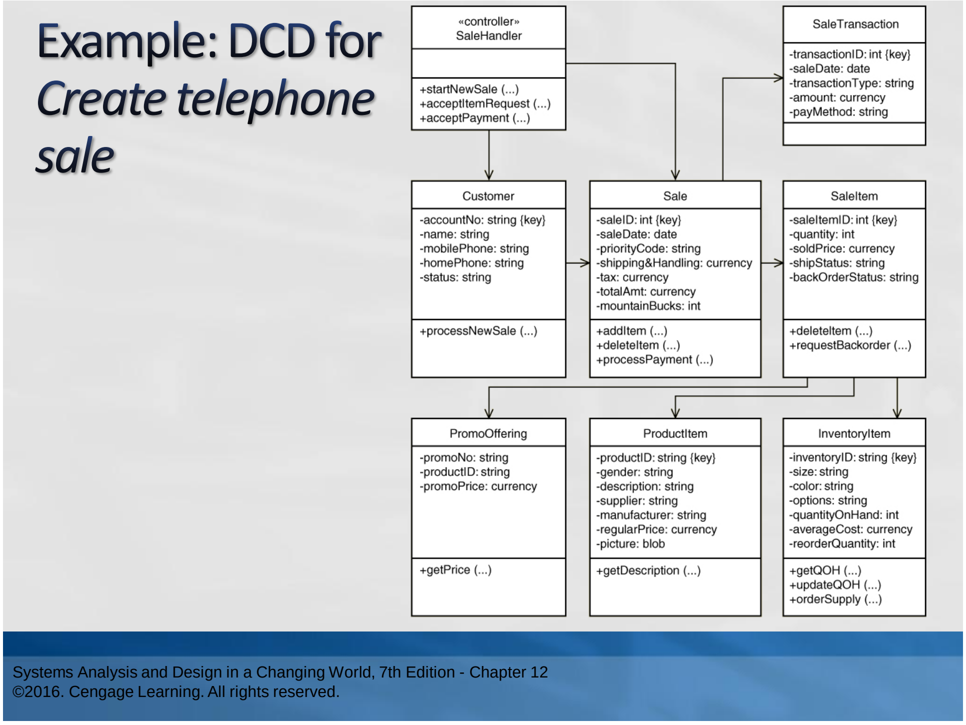
SaleItem
delete item
request backorder
PromoOffering
ProductItem
InventoryItem

ProductItem
provide description

SaleTransaction

InventoryItem
provide quantity
update quantity
order new supply

Systems Analysis and Design in a Changing World, 7th Edition - Chapter 12
©2016. Cengage Learning. All rights reserved.



Example: DCD for *Create telephone sale*

«controller»
SaleHandler
+startNewSale (...)
+acceptItemRequest (...)
+acceptPayment (...)

SaleTransaction
-transactionID: int {key}
-saleDate: date
-transactionType: string
-amount: currency
-payMethod: string

Customer
-accountNo: string {key}
-name: string
-mobilePhone: string
-homePhone: string
-status: string
+processNewSale (...)

Sale
-saleID: int {key}
-saleDate: date
-priorityCode: string
-shipping&Handling: currency
-tax: currency
-totalAmt: currency
-mountainBucks: int
+addItem (...)
+deleteItem (...)
+processPayment (...)

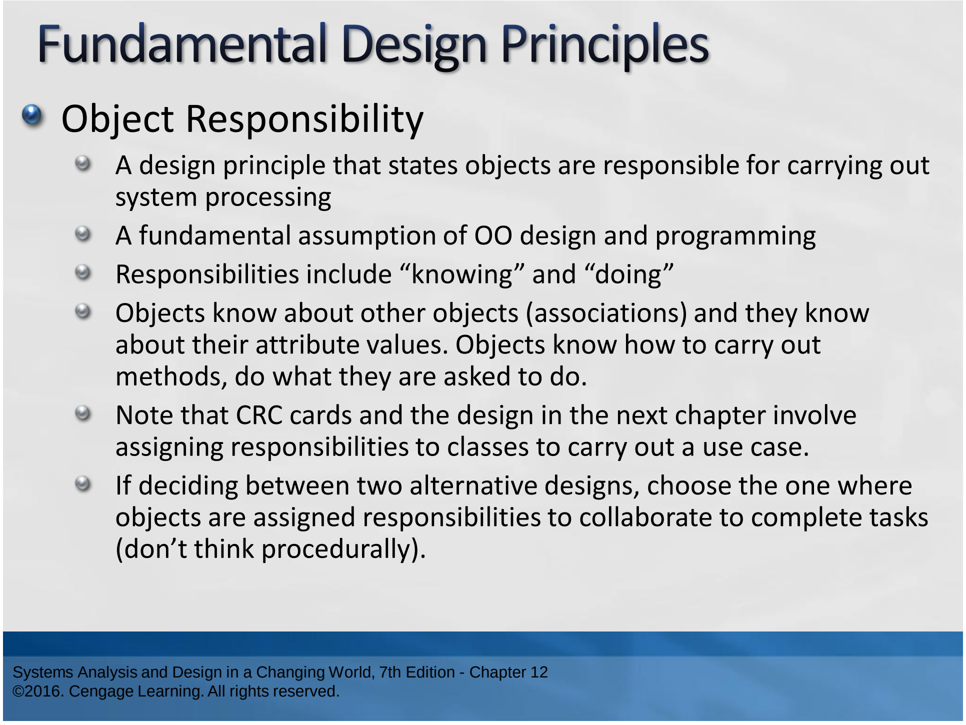
SaleItem
-saleItemID: int {key}
-quantity: int
-soldPrice: currency
-shipStatus: string
-backOrderStatus: string
+deleteItem (...)
+requestBackorder (...)

PromoOffering
-promoNo: string
-productID: string
-promoPrice: currency
+getPrice (...)

ProductItem
-productID: string {key}
-gender: string
-description: string
-supplier: string
-manufacturer: string
-regularPrice: currency
-picture: blob
+getDescription (...)

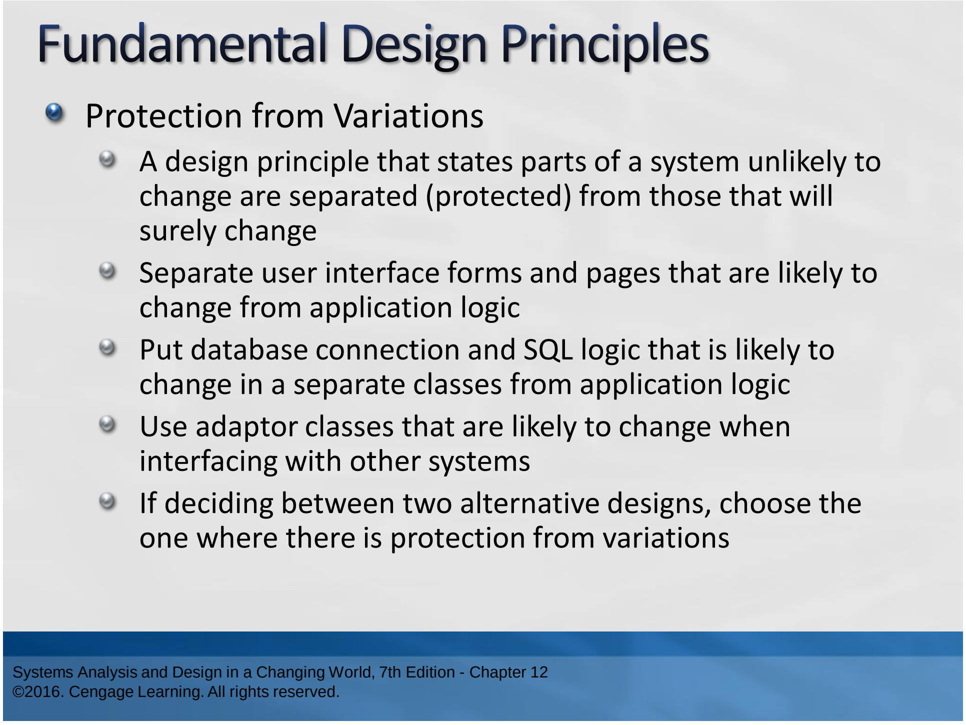
InventoryItem
-inventoryID: string {key}
-size: string
-color: string
-options: string
-quantityOnHand: int
-averageCost: currency
-reorderQuantity: int
+getQOH (...)
+updateQOH (...)
+orderSupply (...)

Systems Analysis and Design in a Changing World, 7th Edition - Chapter 12
©2016. Cengage Learning. All rights reserved.



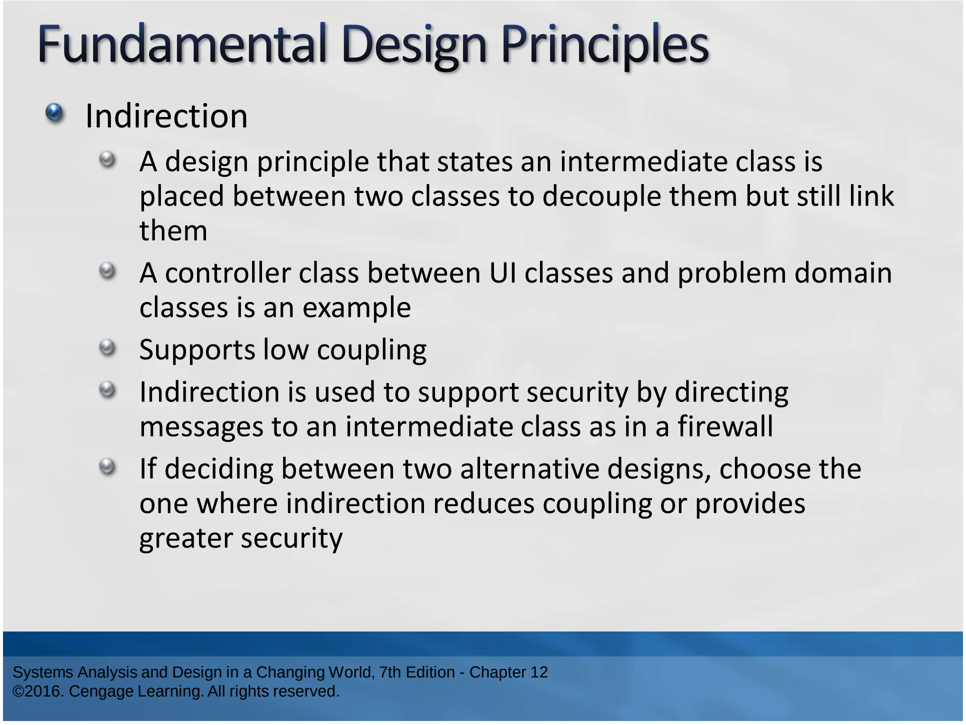
Fundamental Design Principles

- 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).



Fundamental Design Principles

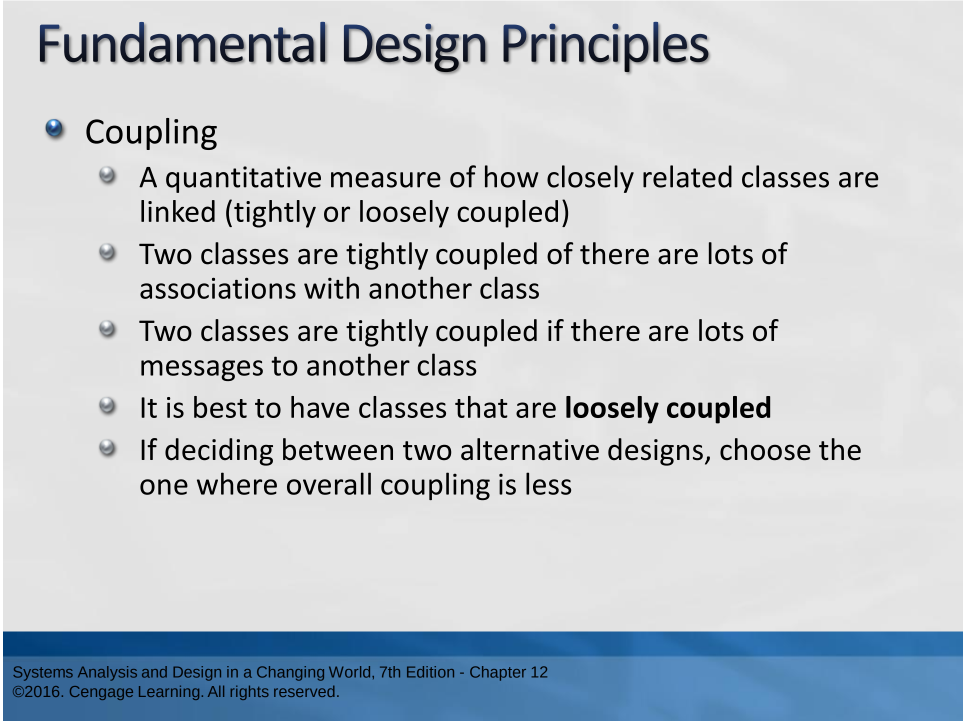
- Protection from Variations
  - A design principle that states parts of a system unlikely to change are separated (protected) from those that will surely change
  - Separate user interface forms and pages that are likely to change from application logic
  - Put database connection and SQL logic that 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

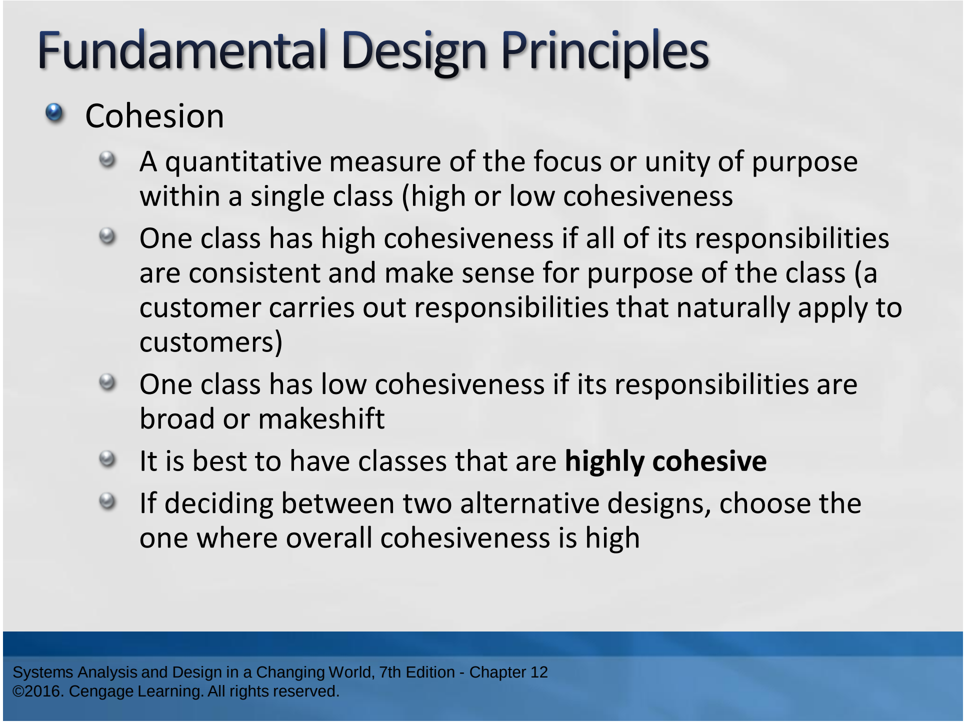
Systems Analysis and Design in a Changing World, 7th Edition - Chapter 12



Fundamental Design Principles

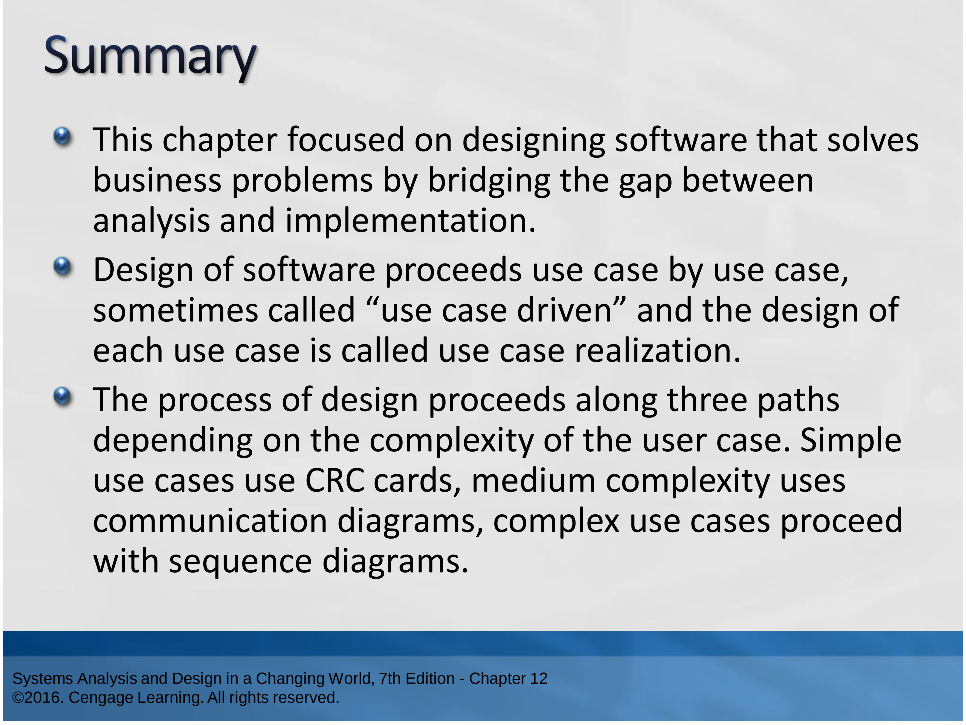
- Coupling
  - A quantitative measure of how closely related classes are linked (tightly or loosely coupled)
  - Two classes are tightly coupled of there are lots of associations with another class
  - Two classes are tightly coupled if there are lots of messages to another class
  - It is best to have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

Systems Analysis and Design in a Changing World, 7th Edition - Chapter 12



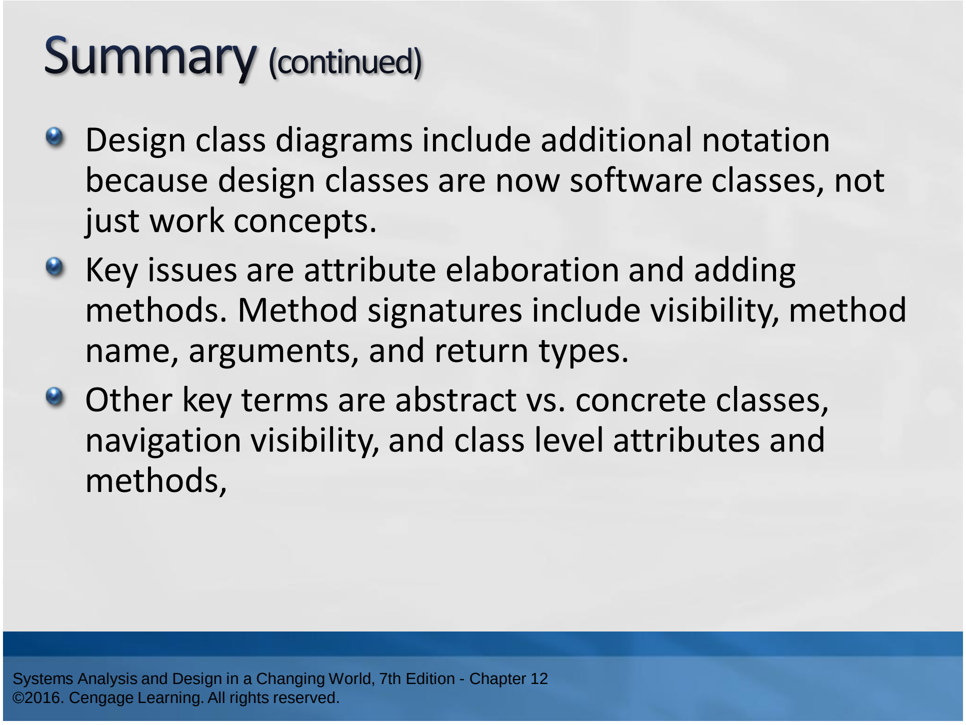
Fundamental Design Principles

- 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



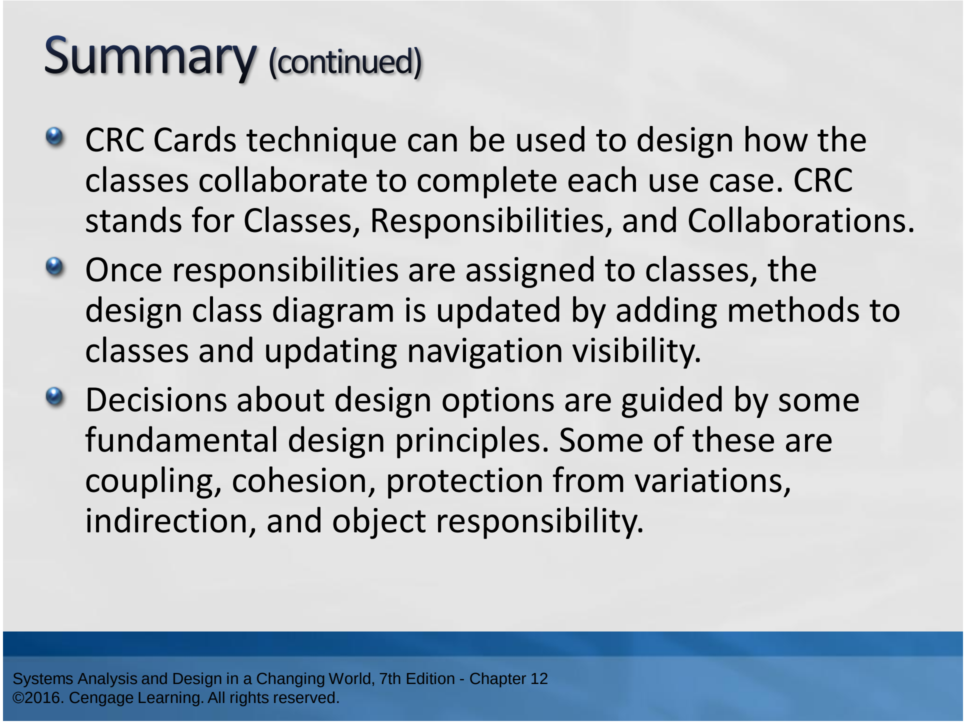
Summary

- This chapter focused on designing software that solves business problems by bridging the gap between analysis and implementation.
- Design of software proceeds use case by use case, sometimes called “use case driven” and the design of each use case is called use case realization.
- The process of design proceeds along three paths depending on the complexity of the user case. Simple use cases use CRC cards, medium complexity uses communication diagrams, complex use cases proceed with sequence diagrams.



Summary (continued)

- Design class diagrams include additional notation because design classes are now software classes, not just work concepts.
- Key issues are attribute elaboration and adding methods. Method signatures include visibility, method name, arguments, and return types.
- Other key terms are abstract vs. concrete classes, navigation visibility, and class level attributes and methods,



Summary (continued)

- CRC Cards technique can be used to design how the classes collaborate to complete each use case. CRC stands for Classes, Responsibilities, and Collaborations.
- Once responsibilities are assigned to classes, the design class diagram is updated by adding methods to classes and updating navigation visibility.
- Decisions about design options are guided by some fundamental design principles. Some of these are coupling, cohesion, protection from variations, indirection, and object responsibility.