



Murdoch
UNIVERSITY

Software design

Topic 9

ICT284 Systems Analysis and Design



About this topic

This topic focuses on the design activity of designing the software classes and methods. The requirements models from the analysis stage are extended to a set of detailed object-oriented design models that specify the system solution in terms of collaborating objects, their attributes, and their methods.

The most important models are the *design class diagram*, which documents the classes that will be built for the system, and the *sequence diagram*, which defines the interactions between objects in order to execute a use case. These models then provide the basis for coding the solution.

Unit learning outcomes addressed in this topic

1. Explain how information systems are used within organisations to fulfil organisational needs
2. **Describe the phases and activities typically involved in the systems development life cycle**
3. Describe the professional roles, skills and ethical issues involved in systems analysis and design work
4. Use a variety of techniques for analysing and defining business problems and opportunities and determining system requirements
5. Model system requirements using UML, including use case diagrams and descriptions, activity diagrams and domain model class diagrams
6. Explain the activities involved in systems design, including designing the system environment, application components, user interfaces, database and software
7. **Represent early system design using UML, including sequence diagrams, architectural diagrams and design class diagrams**
8. Describe tools and techniques for planning, managing and evaluating systems development projects
9. Describe the key features of several different systems development methodologies
10. **Present systems analysis and design documentation in an appropriate, consistent and professional manner**

Topic learning outcomes

After completing this topic you should be able to:

- Explain the purpose and objectives of the core process 'design the software classes and methods' in the SDLC
- Briefly describe some fundamental principles of object-oriented software design
- Describe the steps in developing a design class diagram from the use case models created in analysis
- Be able to interpret first-cut and final design class diagrams
- Be able to interpret domain-level sequence diagrams to model object behaviour
- Briefly explain the different types of objects and layers in an multi-layer sequence diagram

Resources for this topic

READING

- Satzinger, Jackson & Burd, Chapter 12
Omit section 'Designing with CRC cards'
- Satzinger, Jackson & Burd, Chapter 13
Omit sections 'Use Case Realization with Communication Diagrams' and 'Design Patterns'

Except where otherwise referenced, all images in these slides are from those provided with the textbook: Satzinger, J., Jackson, R. and Burd, S. (2016) *Systems Analysis and Design in a Changing World*, 7th edition, Course Technology, Cengage Learning: Boston. ISBN-13 9781305117204

Topic outline

- Introduction
- Principles of object-oriented design
- Steps in object-oriented design
- Design class diagrams
- Sequence diagrams
- Multilayer sequence diagrams
- Package diagrams

Introduction

Design activities - reminder



Murdoch
UNIVERSITY

Design activities

- Describe the environment.
- Design the application components.
- Design user interface.
- Design the database.
- Design the software classes and methods.

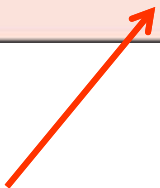
Core processes	Iterations					
	1	2	3	4	5	6
Identify the problem and obtain approval.						
Plan and monitor the project.						
Discover and understand details.						
Design system components.						
Build, test, and integrate system components.						
Complete system tests and deploy the solution.						

... in this topic we will focus on
design the software classes and methods

Key design questions for each activity



Design activity	Key question
Describe the environment	How will this system interact with other systems and with the organization's existing technologies?
Design the application components	What are the key parts of the information system and how will they interact when the system is deployed?
Design the user interface	How will users interact with the information system?
Design the database	How will data be captured, structured, and stored for later use by the information system?
Design the software classes and methods	What internal structure for each application component will ensure efficient construction, rapid deployment, and reliable operation?





Software design in the SDLC

- Bridge between users' requirements and programming of new system
- Process by which a set of detailed OO design models are built to be used for coding
- Specifies the system solution in terms of collaborating objects, their attributes, and their methods
- Use case driven: design is carried out use case by use case



Object-oriented programming

- *Objects* are responsible for carrying out system processing
- Each object has data and program logic *encapsulated* within it
- These are defined in a *class* definition
- Objects are *instantiated* from the class template when the program executes
- An OO program consists of a set of instantiated objects cooperating to accomplish a result
- The objects work together by sending each other *messages* which invoke the *methods* defined for the object

Reminder...

Example: part of a Java program

Class definition
for Student,
including
methods
getFullName,
getGPA, etc

```
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

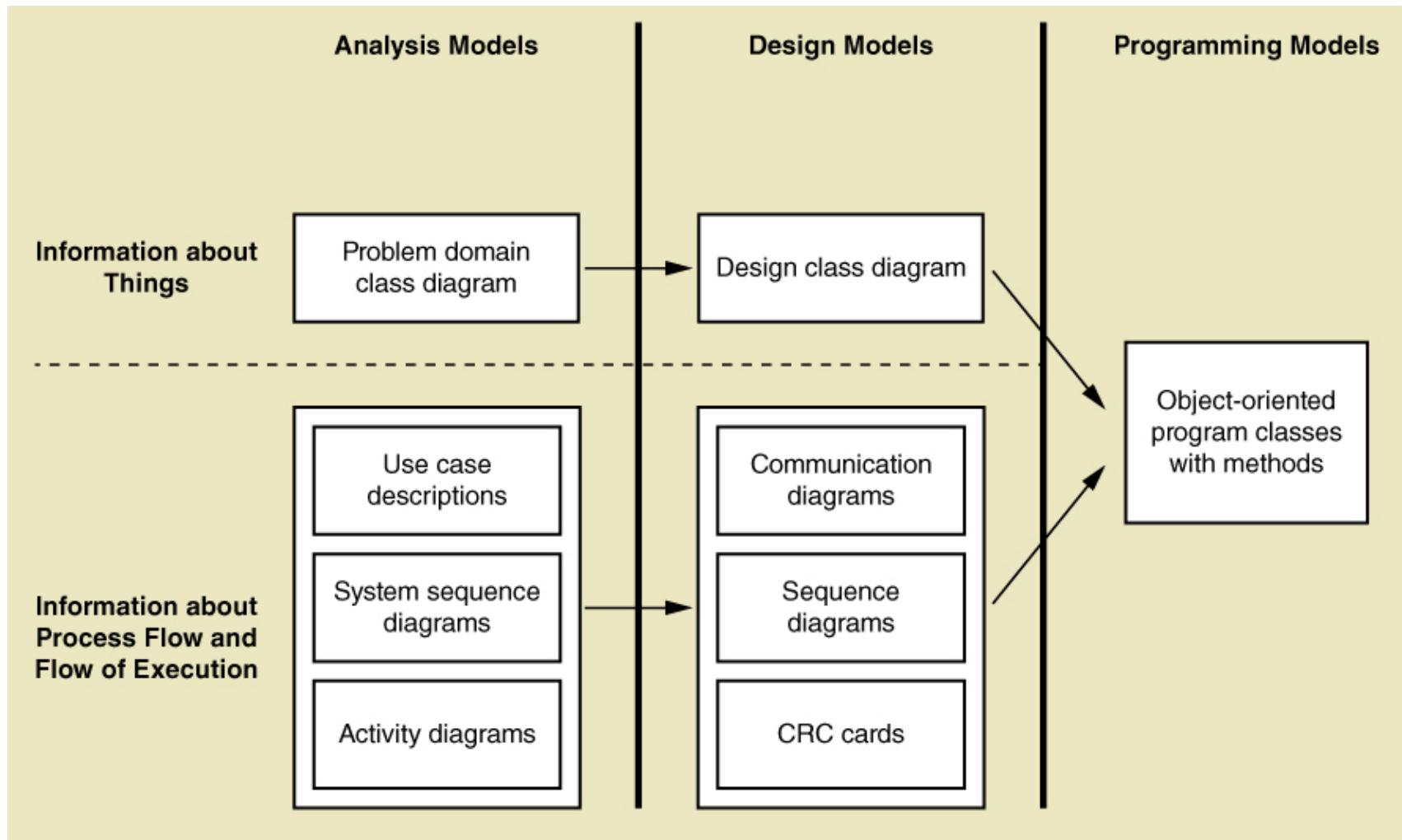
    //processing methods
    public void updateGPA ( )
    {
        //access course records and update lastActiveSemester and
        //to-date credits and GPA
    }
}
```

OO software design: Bridging from analysis to implementation



- Requirements models (from Topic 3, 4, 5) are extended to design models
- Again, there are models for both things, and processes about things
- Design models are created in parallel to actual coding/implementation with iterative SDLC
- Agile approach says create models only if they are necessary: simple aspects don't need a design model before coding

Analysis to design to implementation: models



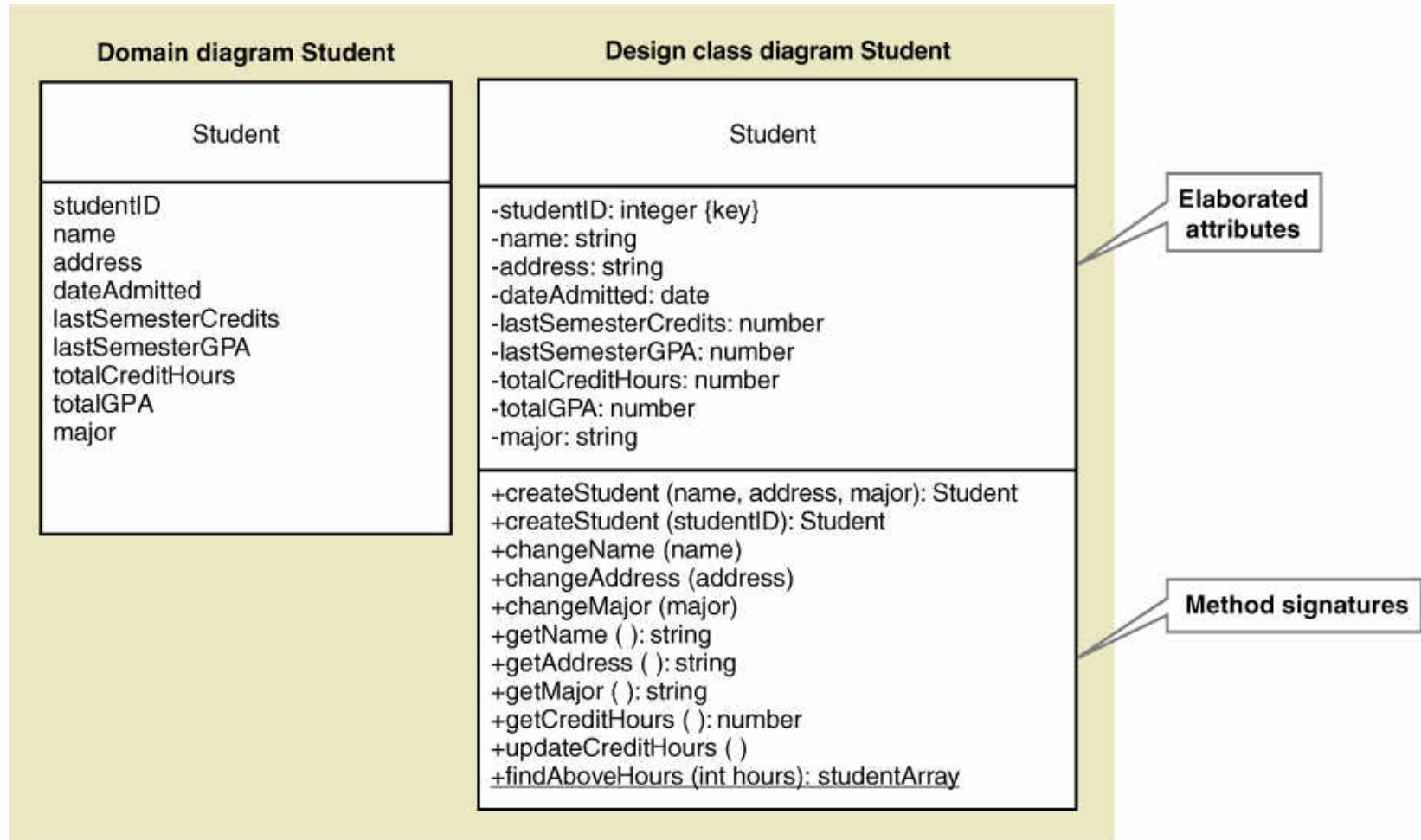


Design models

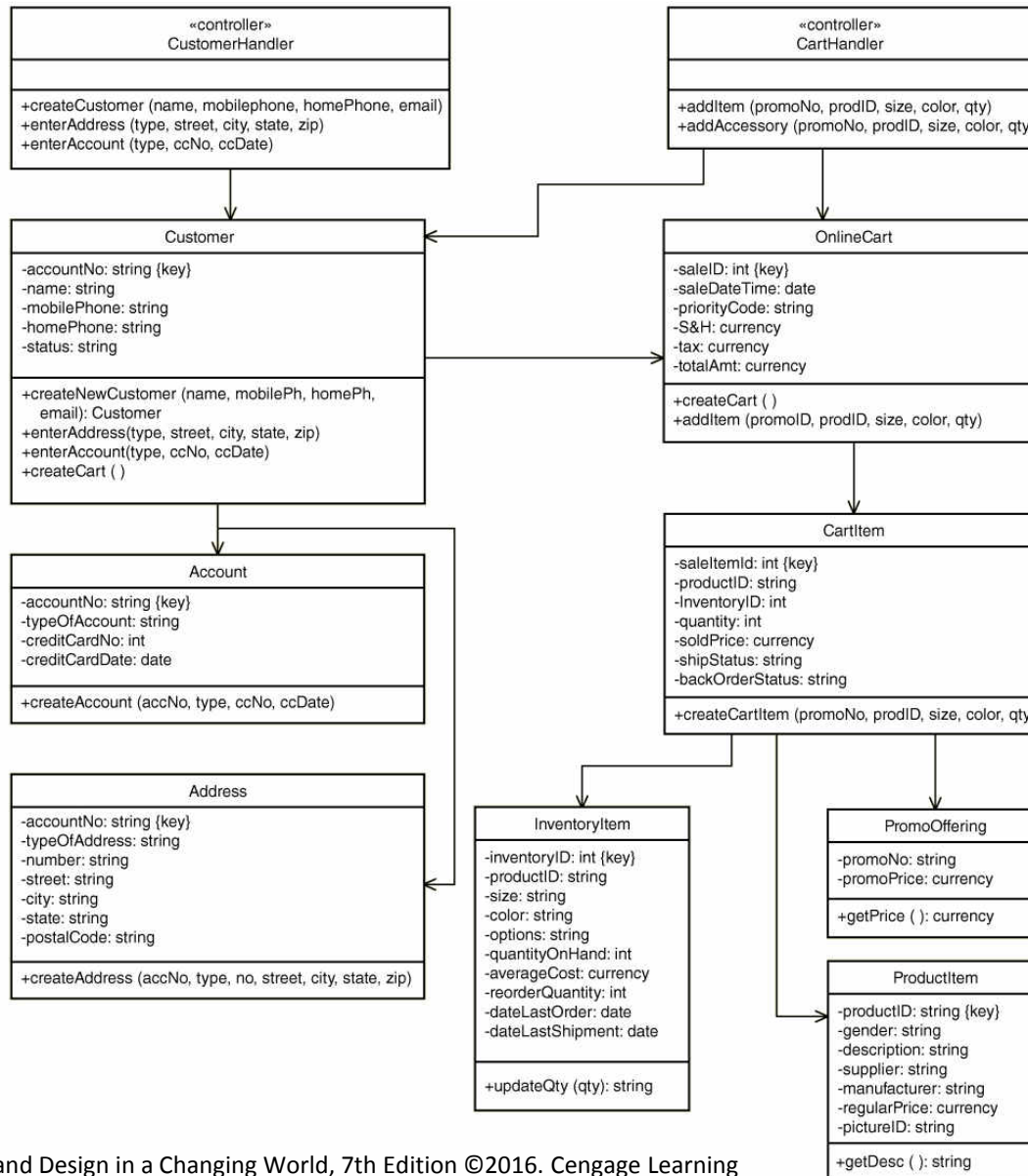
Design class diagrams and *sequence diagrams* are the most important diagrams for detailed design

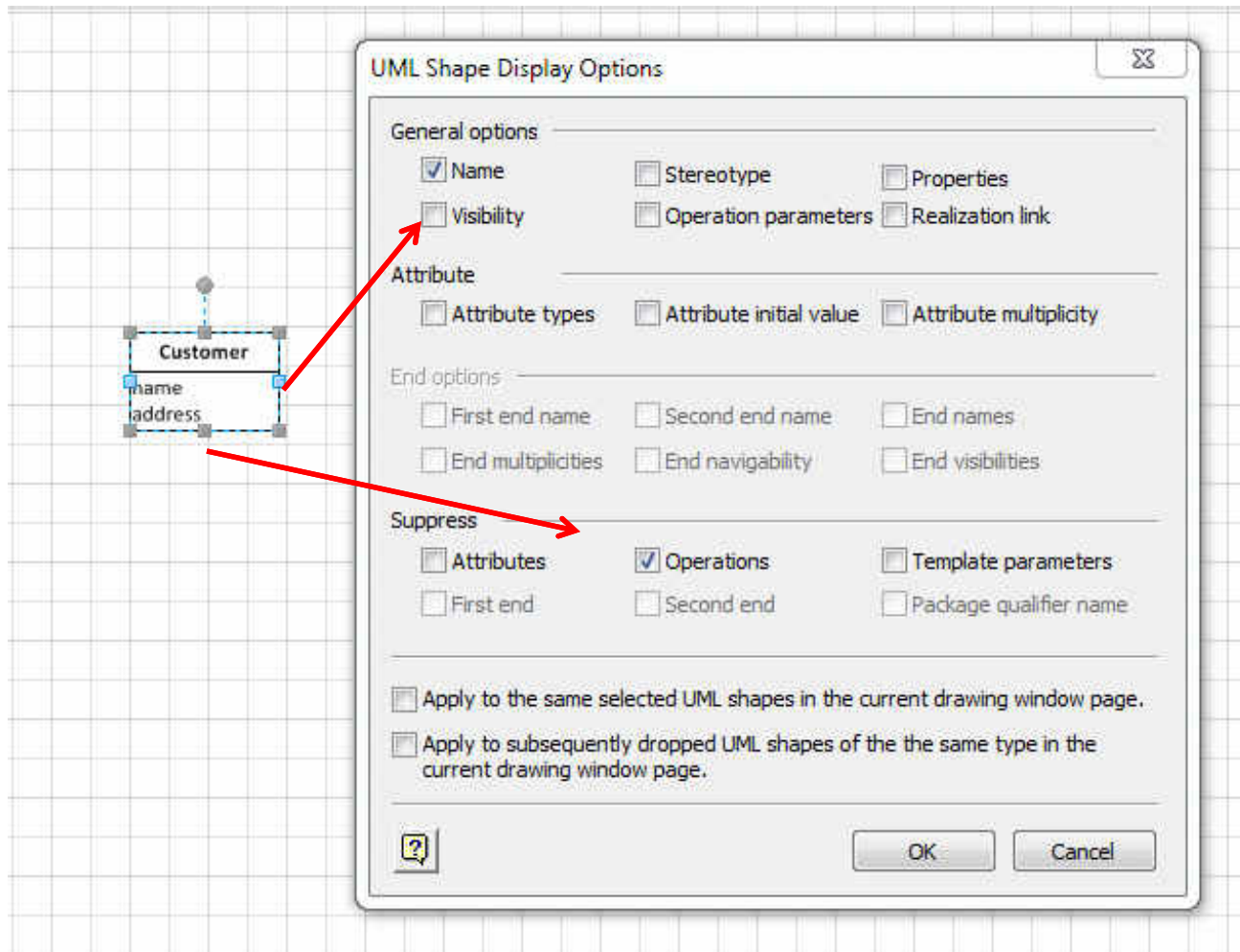
- **Design class diagrams** document the classes that will be built for the system
 - They describe the classes, navigation between the classes, attribute names and properties and method names and properties
- **Sequence diagrams** define the interactions between objects in order to execute a use case
 - Interactions are called *messages*
 - Correspond to *method calls* in programming language

Design models: Design class diagram



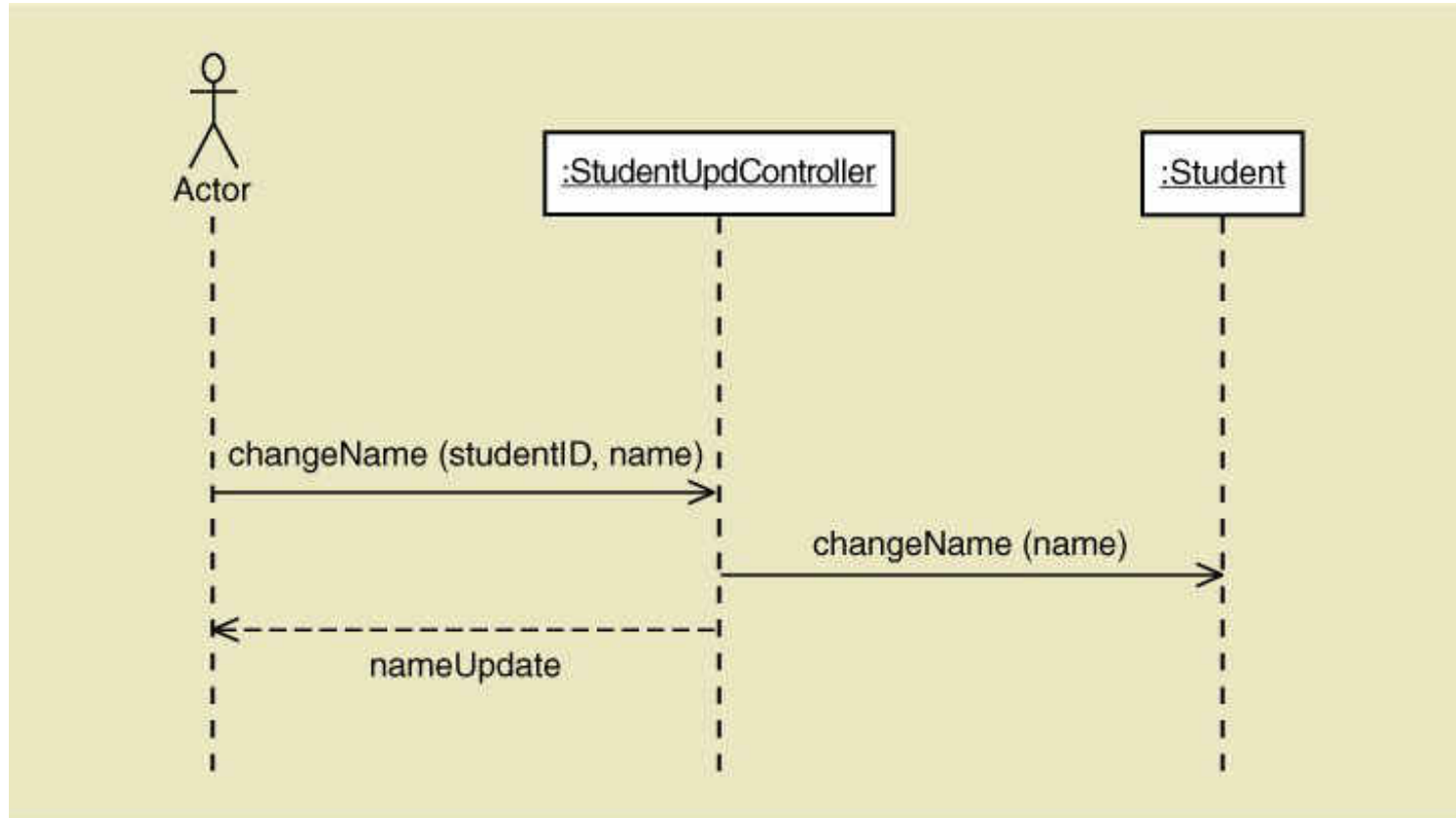
Design models: Design class diagram





Visio 2010 – design class diagram has additional features that are 'switched off' in domain class diagram

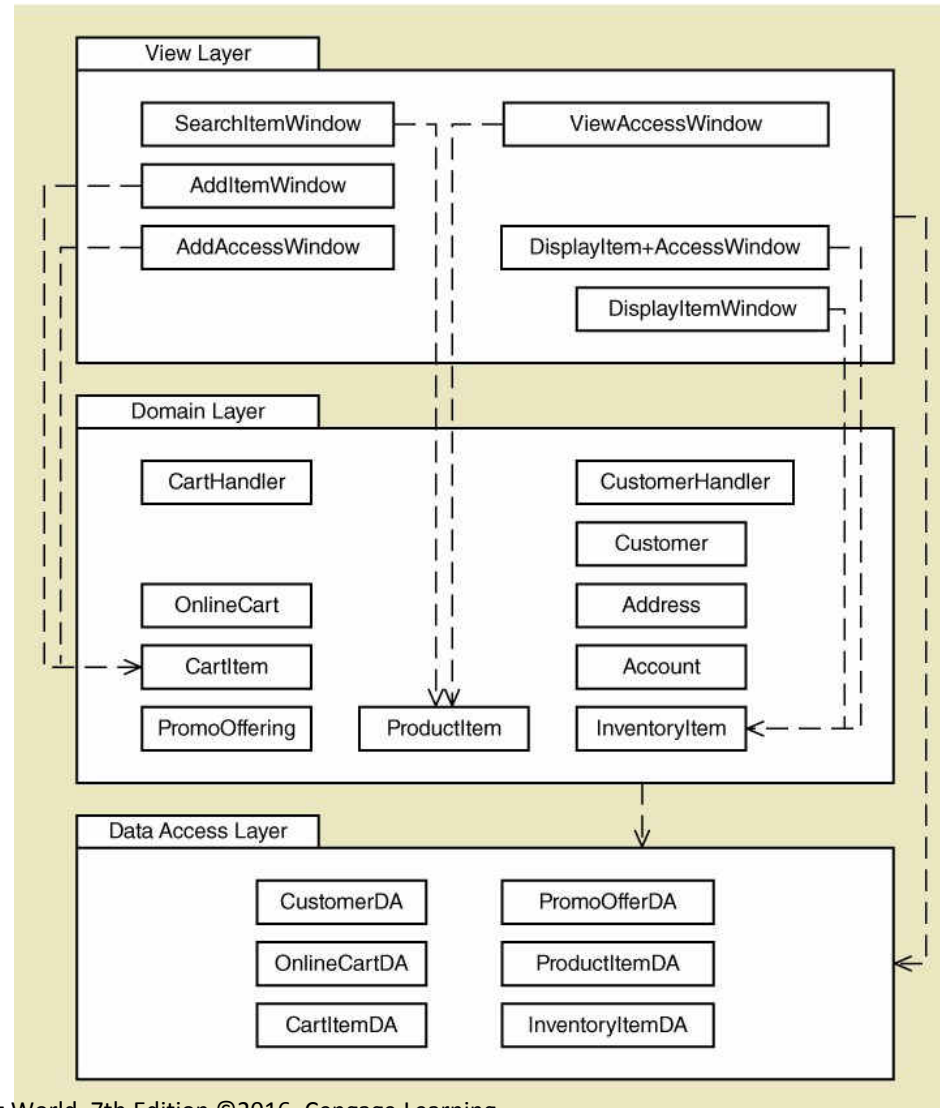
Design models: Sequence diagram





Design models: Package diagram

showing
grouping of
classes
relevant to a
use case
within a 3
layer
architecture



Summing up...

- The final activity in the Design phase is to design the software classes and methods
- An OO program consists of a set of instantiated objects cooperating to accomplish a result
 - The objects work together by sending each other *messages* which invoke the *methods* defined for the object
- To design OO software, we continue to add detail to the requirements models:
- **Design class diagrams** document the classes and methods that will be built for the system
- **Sequence diagrams** define the interactions between objects in order to execute a use case
- Package diagrams show how classes are distributed across a three-layer architecture

OO Design principles



Murdoch
UNIVERSITY

Fundamental design principles

Decisions about OO design options are guided by some fundamental design principles:

- Object Responsibility
- Coupling
- Cohesion
- Separation of Responsibilities
- Protection from Variations
- Indirection



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



Fundamental design principles:

Coupling

- A measure of how closely related classes are linked (tightly or loosely coupled)
- Two classes are tightly coupled if 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 measure of the focus or unity of purpose within a single class (high or low cohesiveness)
- A class has *high* cohesiveness if all of its responsibilities are consistent and make sense for purpose of the class
 - e.g. a customer carries out responsibilities that naturally apply to customers
- A class has *low* cohesiveness if its responsibilities are broad or makeshift
- It is best to have classes that are **highly cohesive**

Fundamental design principles:

Separation of responsibilities



- Applies to a *group* of classes
- Segregate classes into packages or groups based on primary focus of the classes
- Basis for multi-layer design – view, domain, data
- Facilitates multi-tier computer configuration



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

Fundamental design principles:

Indirection



Murdoch
UNIVERSITY

- A design principle that states an intermediate class is placed between two classes to decouple them but still link them
 - e.g. a controller class between UI classes and problem domain classes
- Supports low coupling
- Indirection can be used to support security by directing messages to an intermediate class as in a firewall

Summing up...

Some fundamental principles guide OO design:

- Object Responsibility - **objects** are responsible for system processing
- Coupling - best to have classes that are **loosely coupled**
- Cohesion - best to have classes that are **highly cohesive**
- **Separation of Responsibilities** - Segregate classes into packages or groups based on primary focus of the classes (e.g. view, domain, data)
- **Protection from Variations** - parts of a system unlikely to change are separated (protected) from those that will
- **Indirection** - an intermediate class is placed between two classes to decouple them but still link them

Steps of OO software design



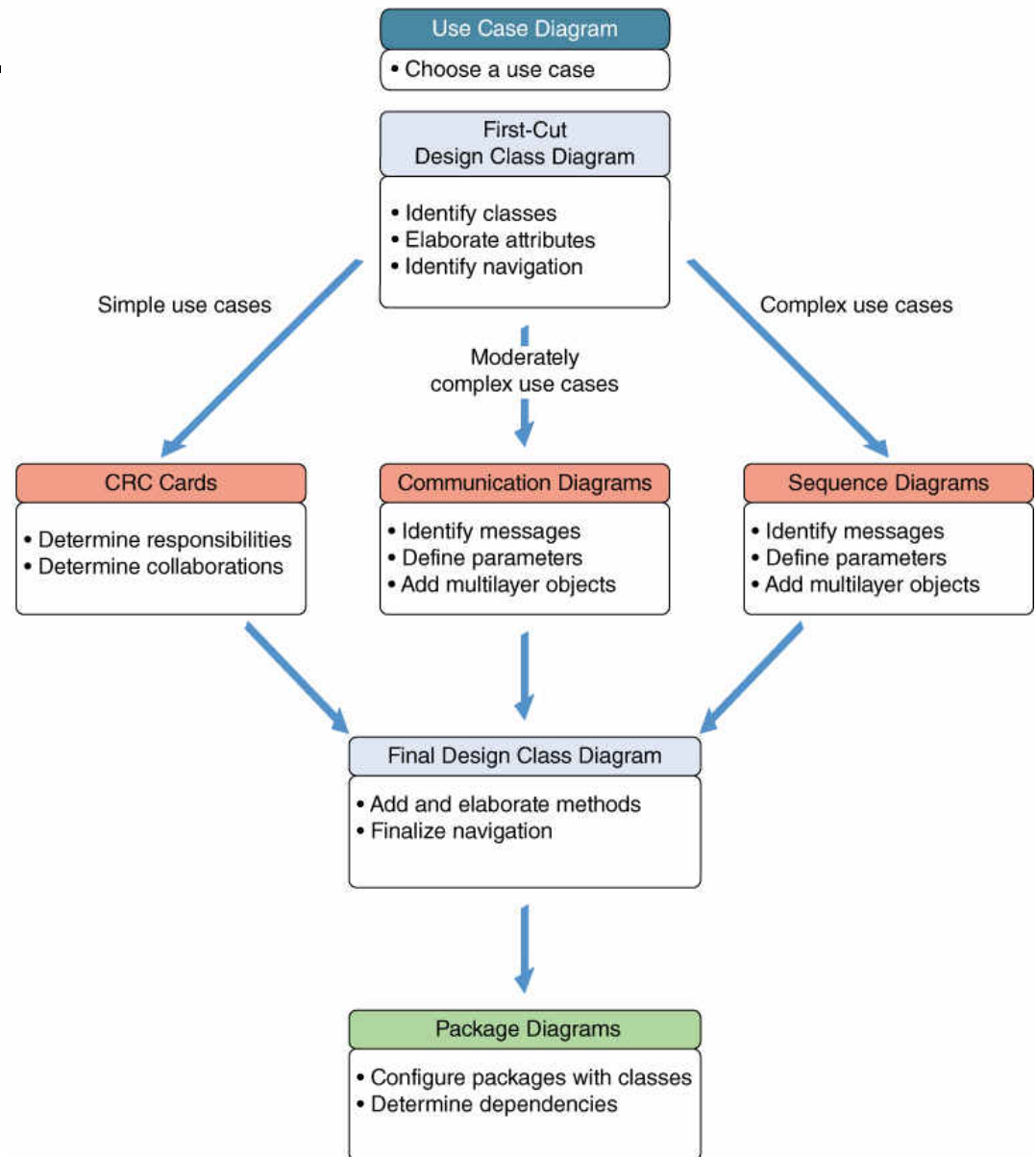
Steps of OO design

- Begin with the models from analysis
- Work with a single use case at a time
- The objective is to end up with a complete design class diagram that includes all the information and behaviour needed to support the functional requirements of the system
- Iterative – create a first cut design class diagram, then continue to update it

Steps of object-oriented design

Three paths:

- Simple use case use CRC Cards
- Medium complexity use case use Communication Diagram
- **Complex use case use Sequence Diagram**





Design steps

- Begin with the models from analysis
- Work with a single use case at a time
- Develop first-cut design class diagram
- Identify and define the methods required in each class (e.g. using sequence diagram)
 - First cut sequence diagram
 - Multilayer sequence diagram
- Update the design class diagram
- Continue for additional use cases
- Partition classes into packages as appropriate

Design class diagrams



Design class diagrams

- The design class diagram contains the final definition of each class in the OO design
- The primary source for the design class diagram is the *domain class diagram* drawn during analysis
- During design the domain classes are built on and made more precise (as we are now defining software)
- Additional classes are added that aren't in the problem domain, to handle things like user interface or data access
- Navigation is added to show how classes reference each other

Different types of design classes

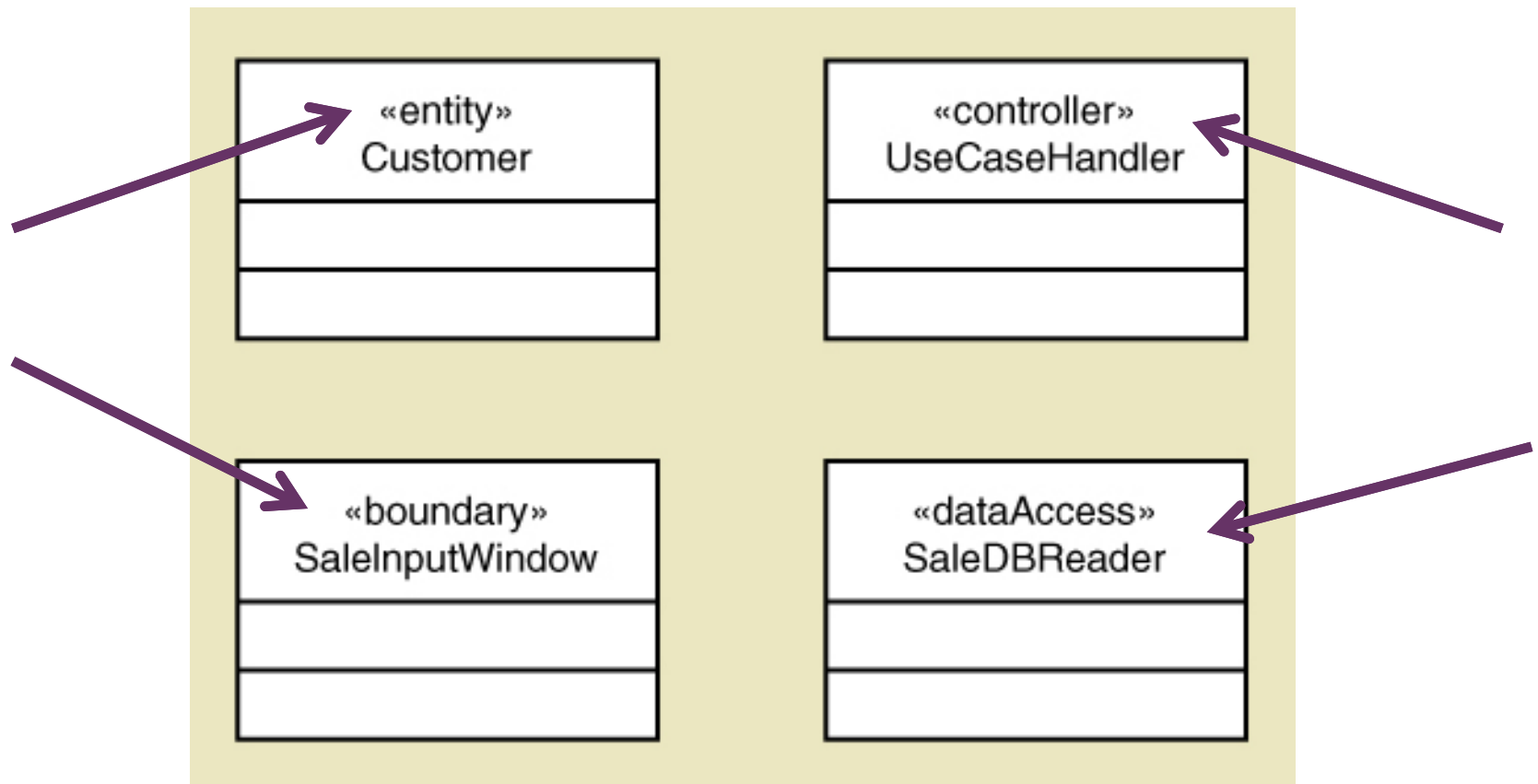
A system is structured into different types (**stereotypes**) of classes:

- **entity class** - a design stereotype for a **problem domain class**. Usually persistent
- **boundary class or view class** - provides the means by which an **actor interacts with the system**, e.g. for user interface window, dialogue box, Web page; or for system interface, an application program interface (API)
- **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**



Design class stereotypes

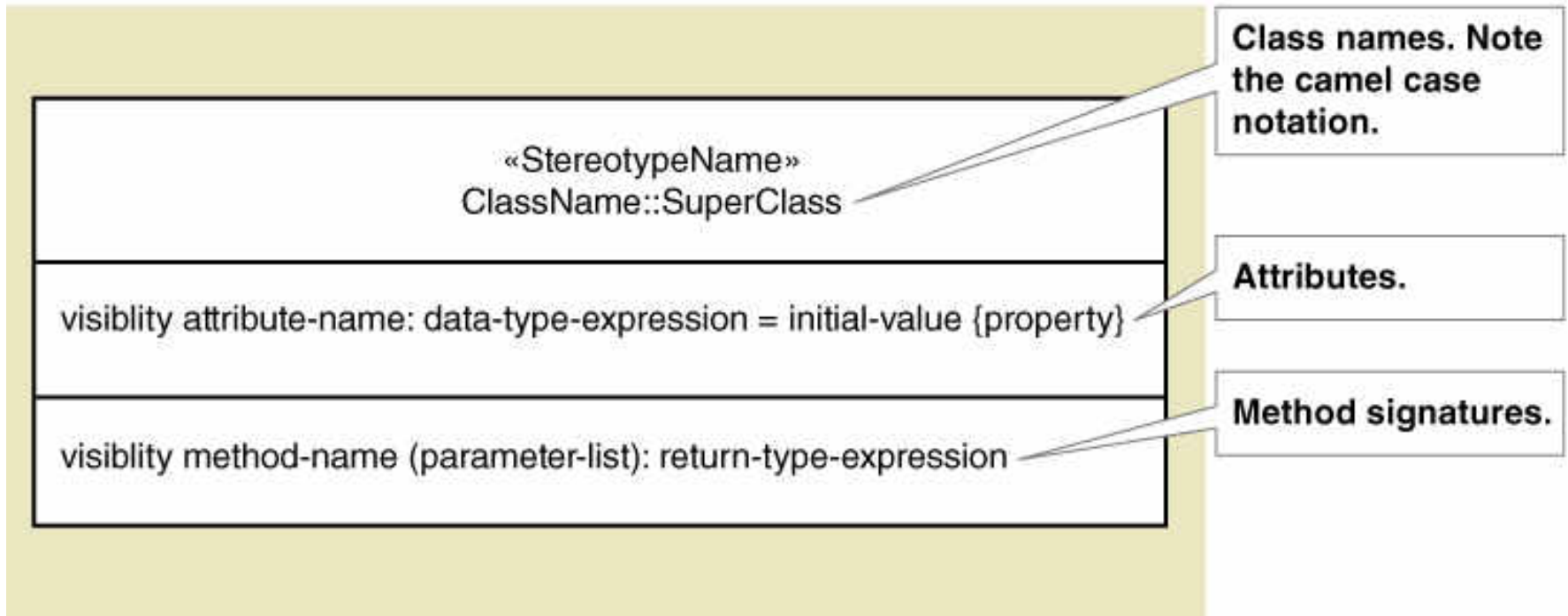
- Stereotypes indicate what type of class it is. Indicate on the diagram by << >>





Notation for a design class

- Syntax for Name, Attributes, and Methods



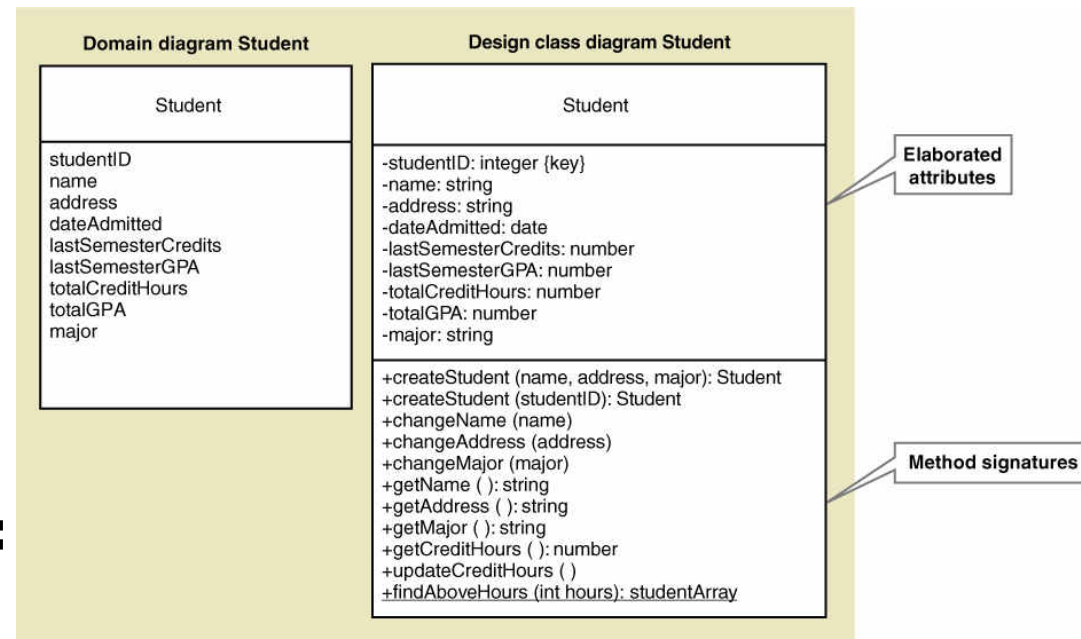
From domain class to design class

Elaborate attributes:

- Visibility
- Attribute name
- Data type
- Initial value
- Property

Add method signatures:

- Method visibility
- Method name
- Method parameter list
- Return type expression





Design steps

- Begin with the models from analysis
- Work with a single use case at a time
- **Develop first-cut design class diagram**
- Identify and define the methods required in each class (e.g. using sequence diagram)
 - First cut sequence diagram
 - Multilayer sequence diagram
- Update the design class diagram
- Continue for additional use cases
- Partition classes into packages as appropriate



Developing design classes 1: Attributes

- **Visibility**
- **Attribute name** in lower case camelback notation
- **Data Type expression** e.g. character, string, integer, number, currency, date
- **Initial value** - the default value (if applicable)
- **Property** — if applicable, such as {key}

Examples:

-accountNo: String {key}

-startingJobCode: integer = 01

Developing design classes 2:

Attribute visibility



Visibility indicates whether other objects can directly access the attribute

- + the attribute is **public** or visible
- the attribute is **private** or invisible
- Usually attributes would be private, meaning they can only be accessed via the methods in the same class

Address
-street : string
-city : string
+getStreet() : string
+getCity() : string

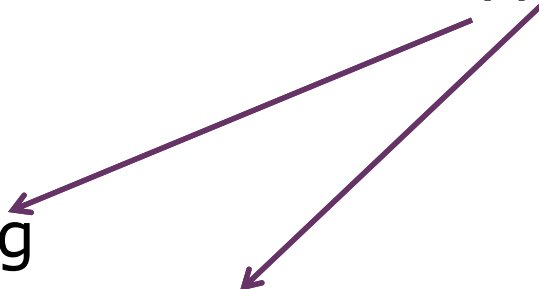


Developing design classes 3: Methods

- **Visibility**
- **Method name** - verb-noun; camelCase
- **Parameter list** (**variables passed to method**)
- **Return type expression** – the type of the data returned

Examples:

+getName(): string
-checkValidity(**date**): int



Developing design classes 4: Method visibility



Visibility indicates whether a method can be invoked by another object

- + the method is **public** or visible
- the method is **private** or invisible
- Usually methods would be **public** so that they can be invoked in response to a message sent by another

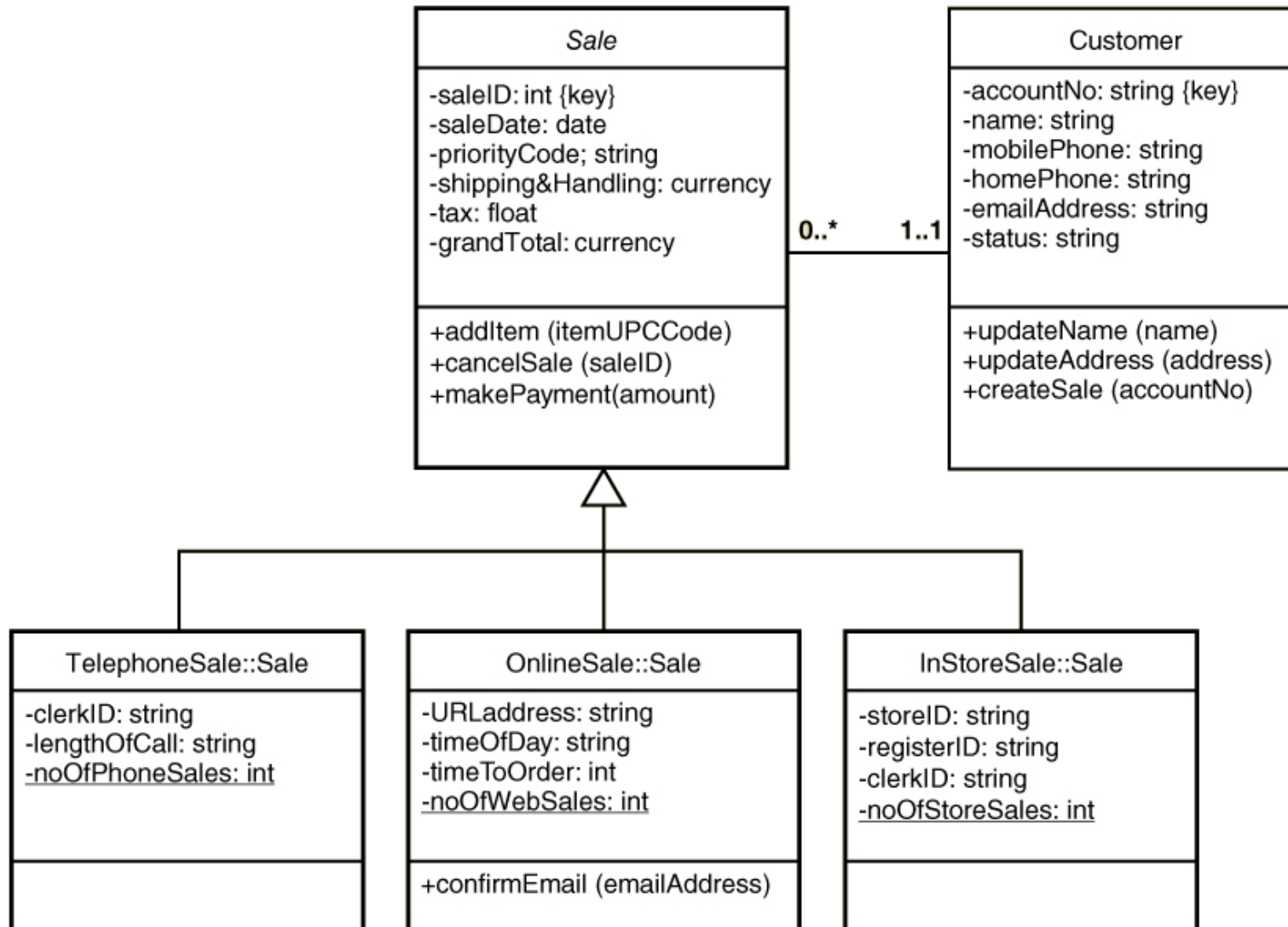
Address
-street : string
-city : string
+getStreet() : string
+getCity() : string



Developing design classes 5:

- **Class level method** applies to all instances of class rather than individual ones. Underlined.
+findStudentsAboveHours(hours): Array
+getNumberOfCustomers(): Integer
- **Class level attribute** has the same value for all objects of the class. Underlined.
-noOfPhoneSales: int
- Abstract class— class that can't be instantiated. Only for inheritance. Name in *Italics*.
- Concrete class—class that can be instantiated.

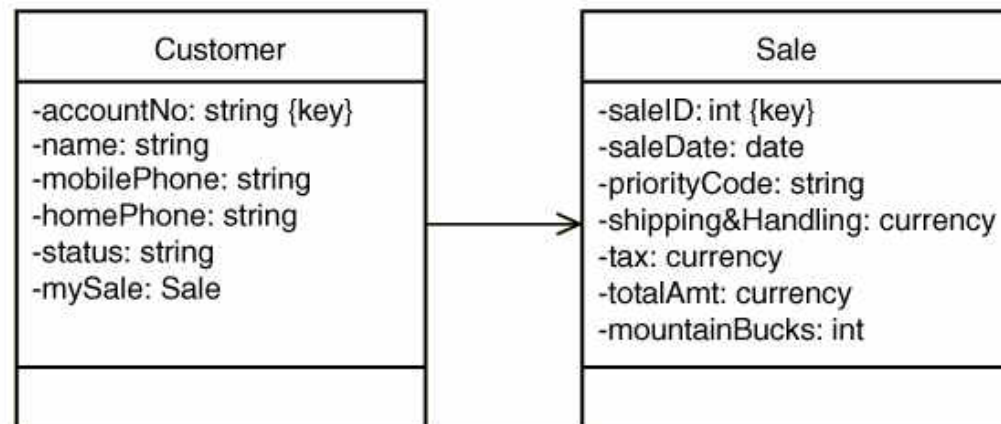
Example of design class with elaborated attributes and method





Developing design classes 6: Navigation visibility

- For one object to interact with another, the first object must be visible to the second object
- Accomplished by adding an *object reference variable* to a class (mySale below; though not always shown)
- Shown as an arrow head on the association line - Customer can find and interact with Sale because it has mySale reference variable
- This navigation is one way, others could be two-way





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 *more dependent*
- When an object needs information from another object, a navigation arrow might be required
- Navigation arrows may be bidirectional

Drawing the **First-Cut Design Class Diagram**



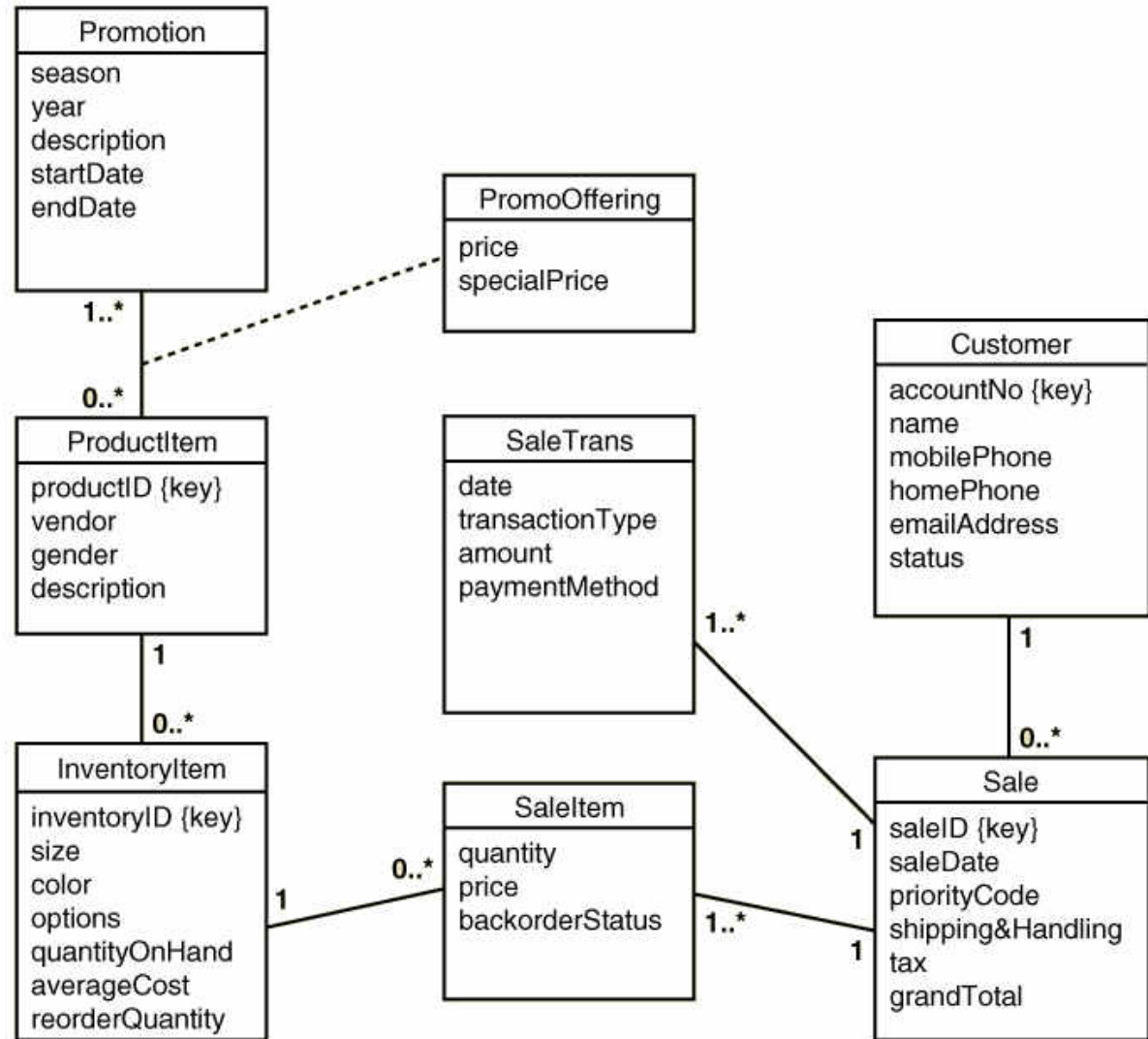
Extend the domain class diagram, by:

1. Add a **controller** class to be in charge of the use case
2. **Elaborating the attributes** of each class with visibility and type
3. Adding **navigation visibility** arrows to the diagram
 - Proceed use case by use case
 - At this point we haven't defined the methods yet, so that is left blank



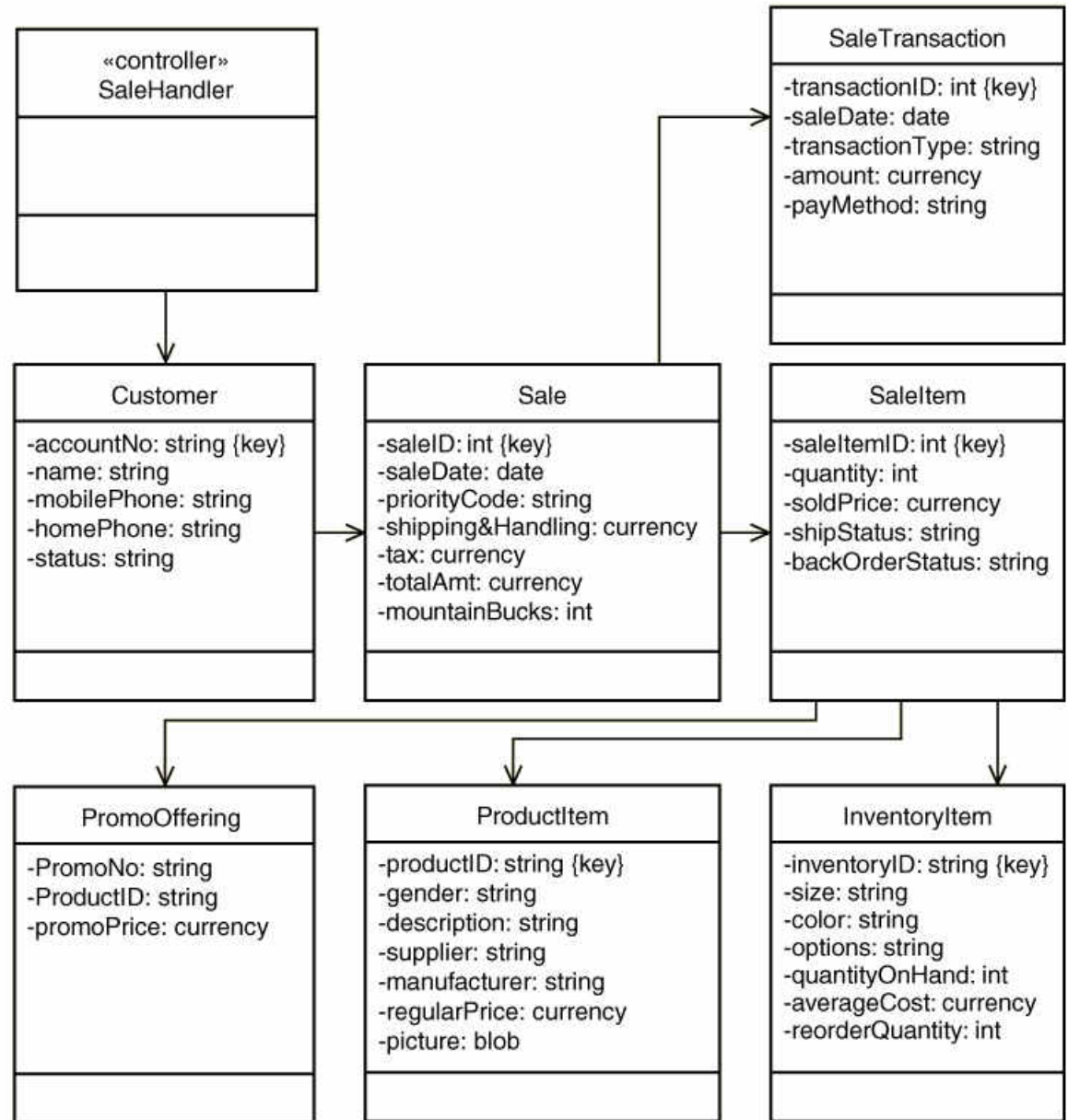
Example: RMO Sales Subsystem

Start with Domain Class Diagram...



First cut design class diagram for use case *Create telephone sale*

includes controller class and navigation visibility



Summing up...

- The Design Class Diagram is built from the Domain Model Class Diagram by adding information:
 - **Attribute elaboration** – data type, properties, visibility
 - **Method signatures** – method name, parameters, return type, visibility
 - **Navigation visibility** is also added between classes to show how objects can interact
 - **Additional classes** (boundary, controller, data access) to handle interactions with actors or between layers
- The *first cut DCD* includes controller class and navigation, but no methods as they haven't been defined yet

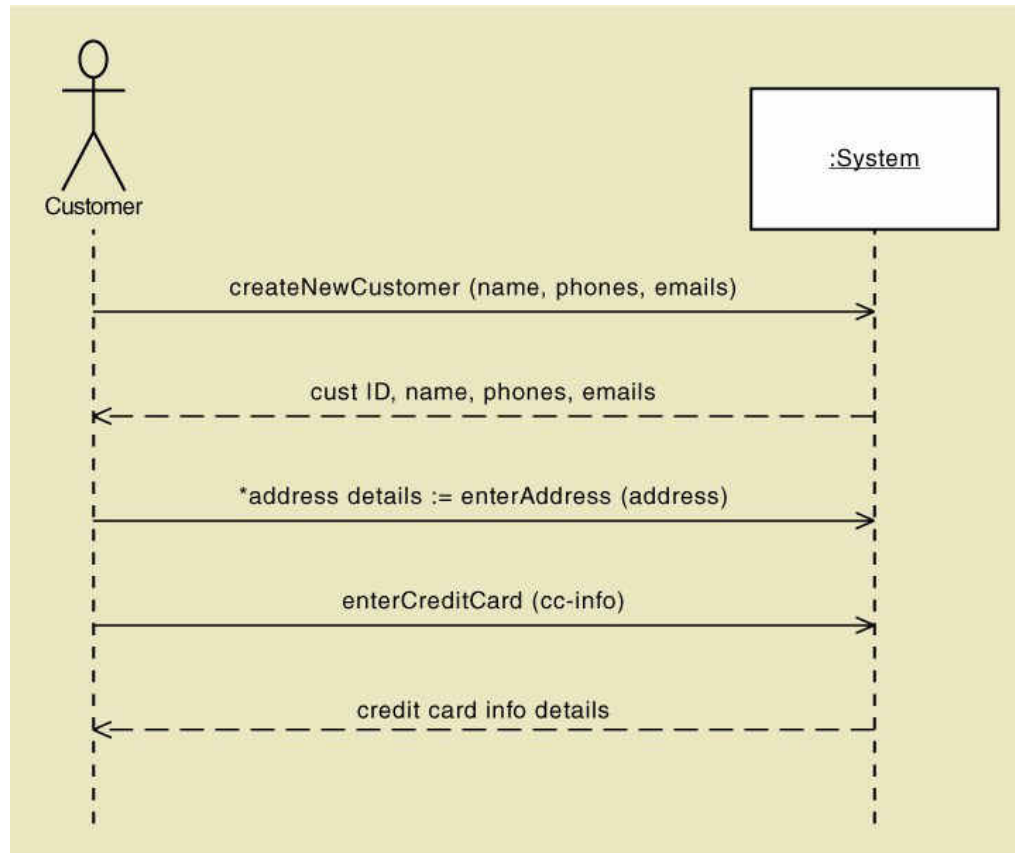
Sequence diagrams



Design steps

- Begin with the models from analysis
- Work with a single use case at a time
- Develop first-cut design class diagram
- Identify and define the methods required in each class (e.g. using sequence diagram)
 - First cut sequence diagram
 - Multilayer sequence diagram
- Update the design class diagram
- Continue for additional use cases
- Partition classes into packages as appropriate

Reminder – System Sequence Diagram (SSD)



- Shows only a single object, `:System`



Sequence diagrams

- 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
- 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



Use case Controller

- Switchboard between user-interface classes and domain layer classes
- Reduces coupling between view and domain layer
- A controller can be created for each use case, however, several controllers can be combined together for a group of related use cases
- It is a completely artificial class



Elements of Sequence Diagrams

- Lifeline
 - The dashed line under the object which serves as an origin point and a destination point for messages
- Activation lifeline
 - The vertical box on a lifeline which indicates the time period when the object is executing based on the message
- Messages have origins and destinations
 - May be on lifeline or on object box
 - Return values may be dashed message arrow, or on same message



OOD with Sequence Diagrams

- Choose a use case
 - Input models – activity diagram, SSD, classes
- Create first-cut design class diagram
- Extend input messages
 - Add all required internal messages
 - Origin and destination objects
 - Elaborate each message
- Add other layers as desired (view, data access)
- Update design class diagram



Guidelines for drawing SD

- From each input message of the use case, determine all internal messages that result
 - what is the objective of the message, what information is needed and what classes need it (destination) and what classes are source
- Determine all objects (from classes) that will be needed for the use case
- Flesh out each message with true/false conditions, parameters and returned values, etc

Assumptions for first-cut sequence diagrams

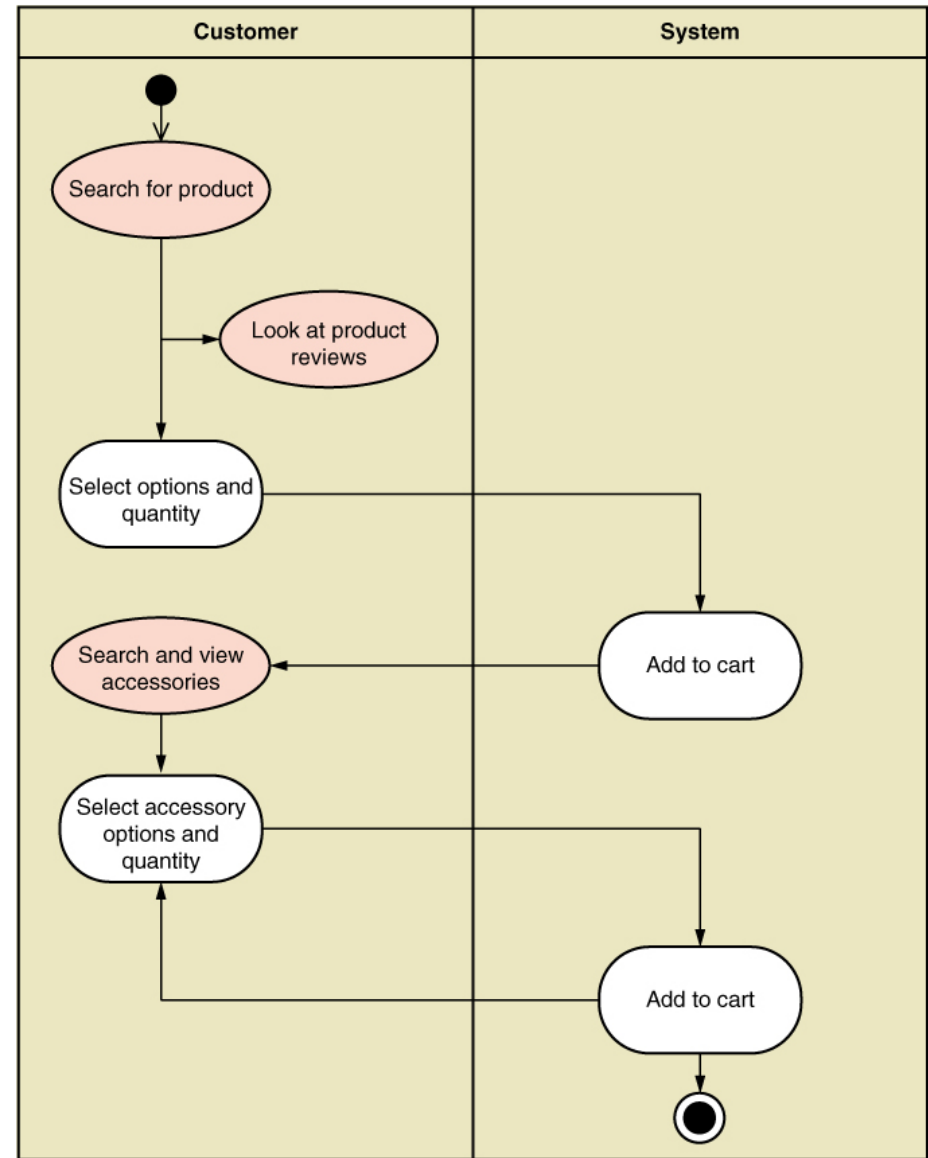


- Perfect **technology** assumption
No logon or other technical issues
- Perfect **memory** assumption
No need to read or write data
- Perfect **solution** assumption
No exception conditions, no error handling

Example: *Fill Shopping Cart*

Input activity diagram

- Note the activity flows that cross the system boundary

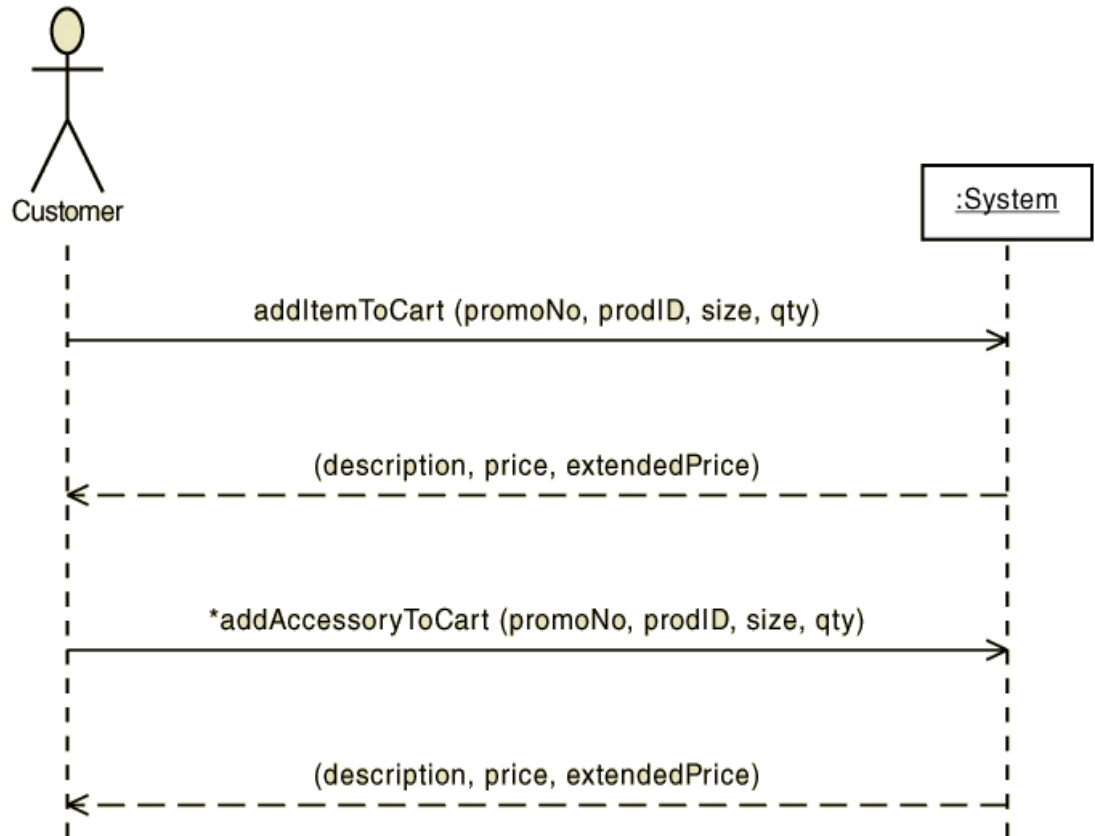


Example:

Fill Shopping Cart

SSD

- Note the input and return messages
- Note the repeating message

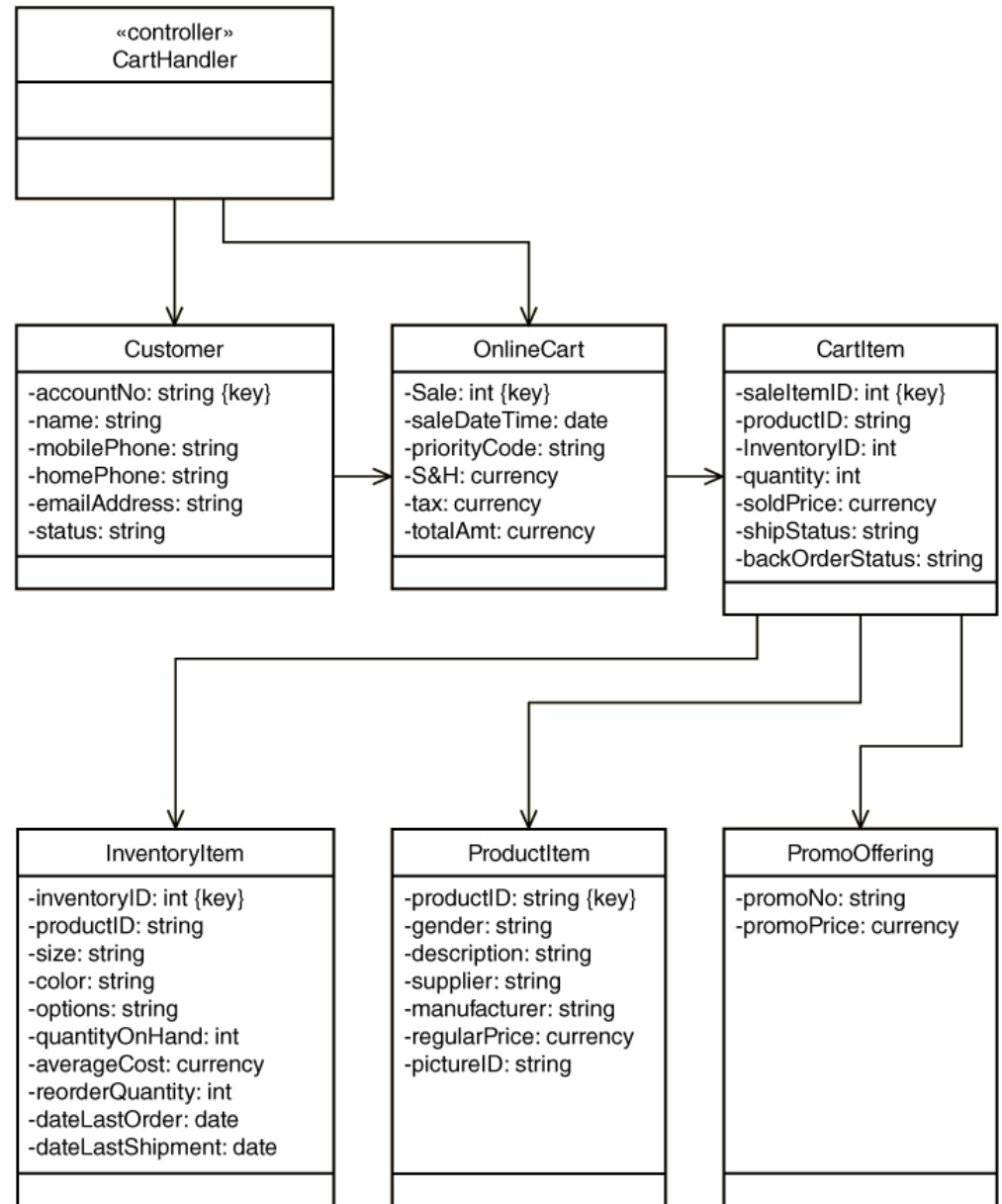


Adds an item and (multiple) accessory items to the shopping cart

Example: *Fill Shopping Cart*

To carry out this use case need to know about Customer, OnlineCart, CartItem; and also information about ProductItem (what it is), InventoryItem (is it in stock), PromoOffering (price)

... this gives us the First Cut DCD

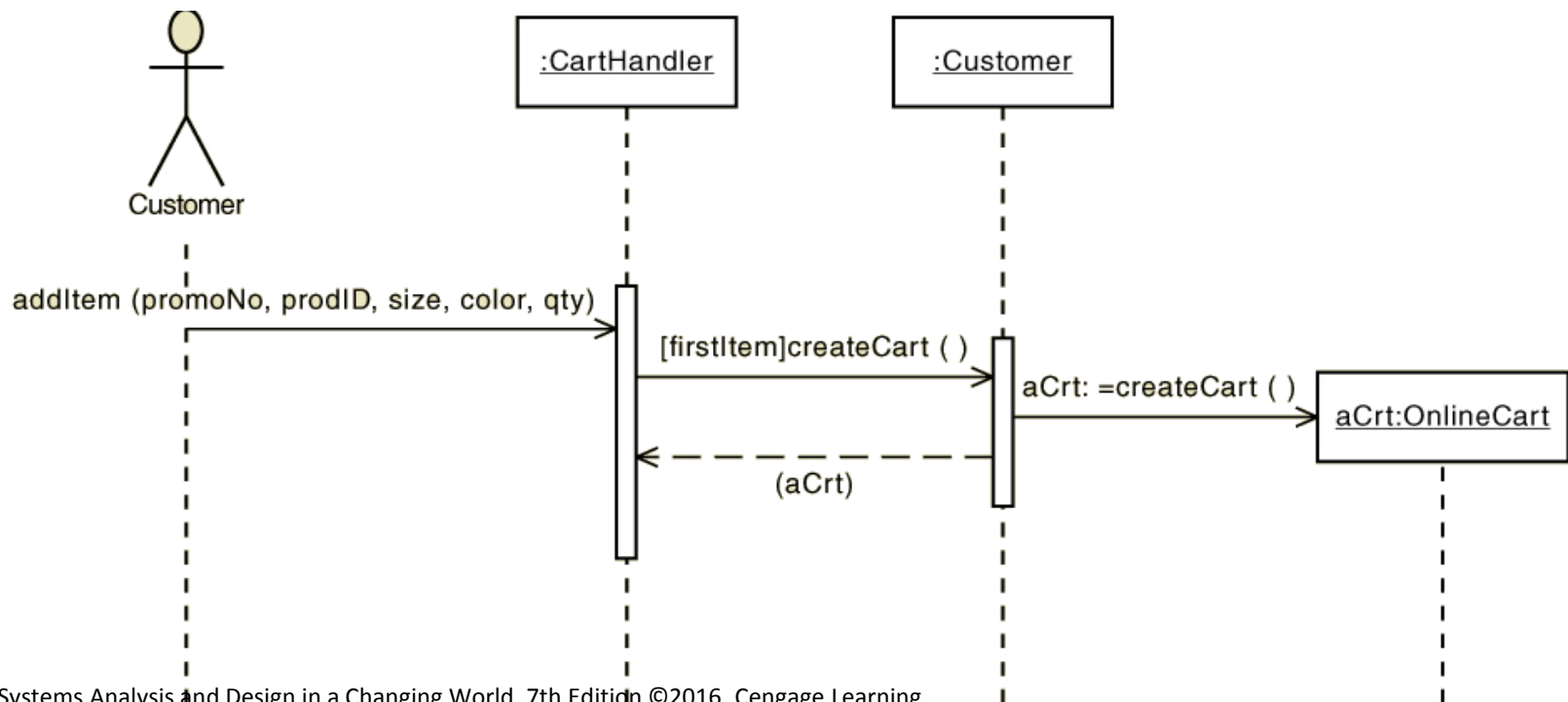




Example: *Fill Shopping Cart*

- First step in *addItem* message - Create a cart if first item in the purchase

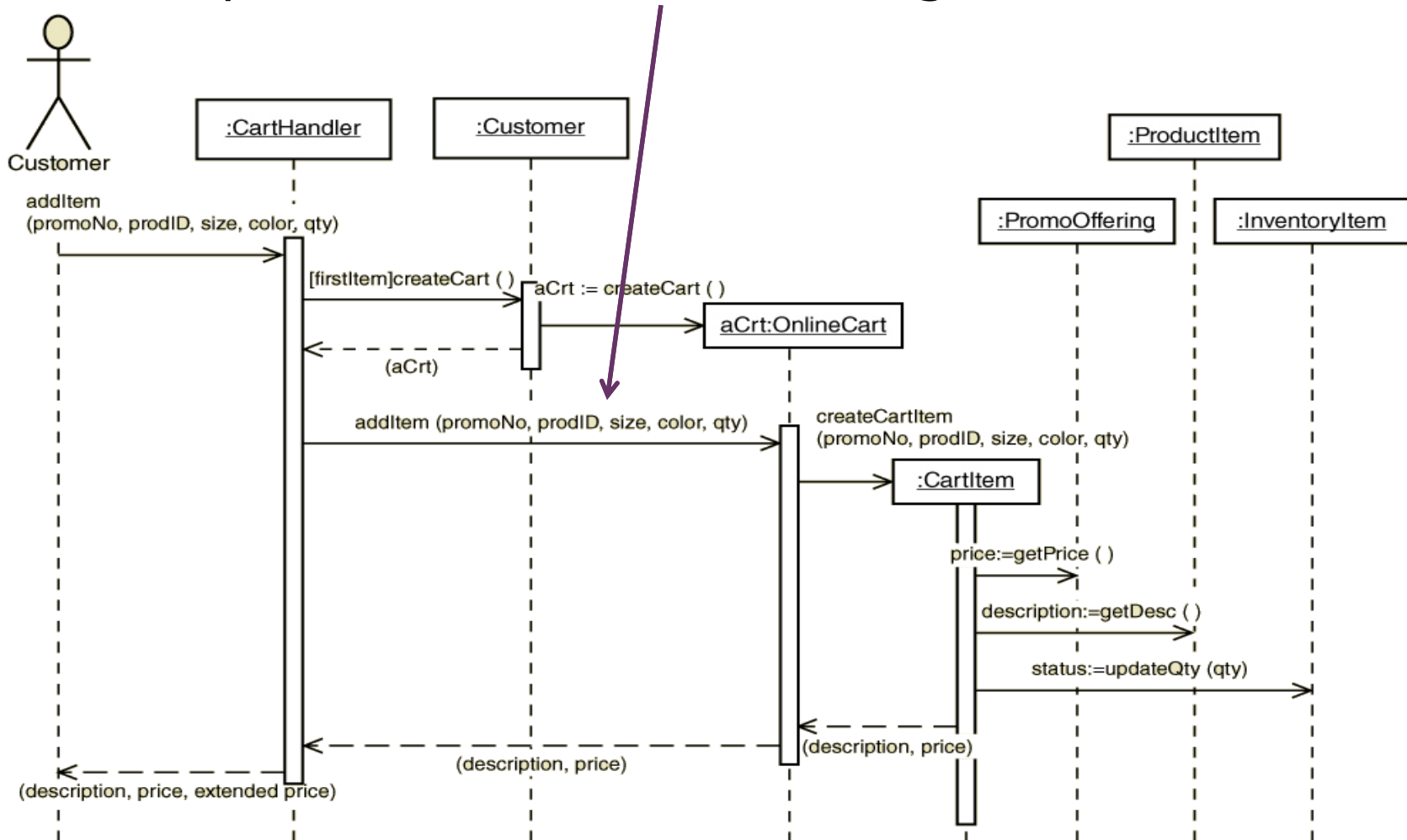
Note true/false test for firstItem, two ways of returning aCrt values, named objects, activation lifelines, message origins and destinations





Example: *Fill Shopping Cart*

- Completion of *addItem* message





Example: *Fill Shopping Cart*

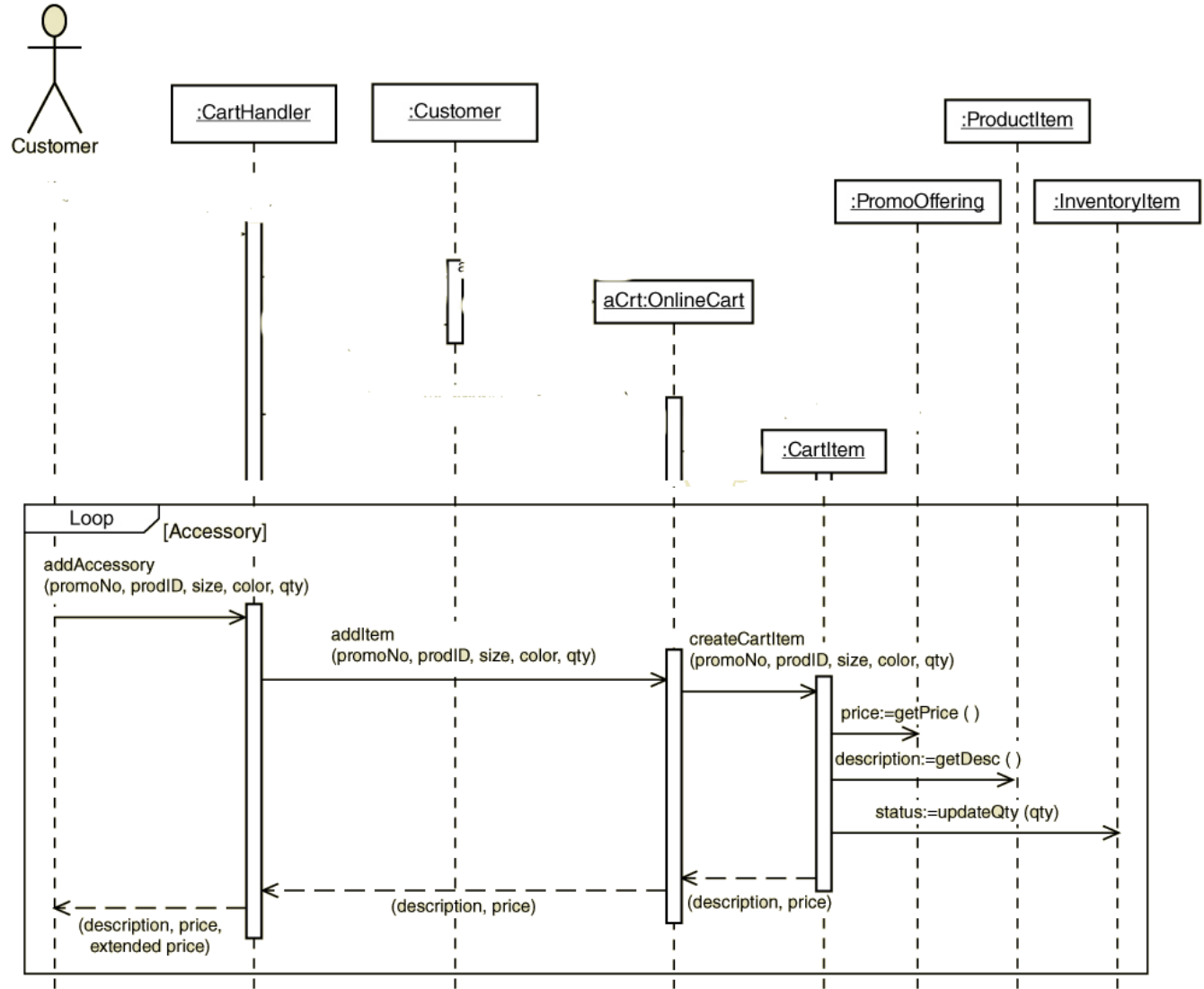
Note origin and destination objects and visibility

- createCart () – cart handler knows if first item
- aCrt:=createCart() – customer owns onlineCart
- addItem() – forwarded message
- createCartItem() – cartItem responsible for creating itself and getting values
- getPrice() – just returns the price
- getDescription() – just returns description
- updateQty(qty) – initiates updates

... these can now be added to the design class diagram



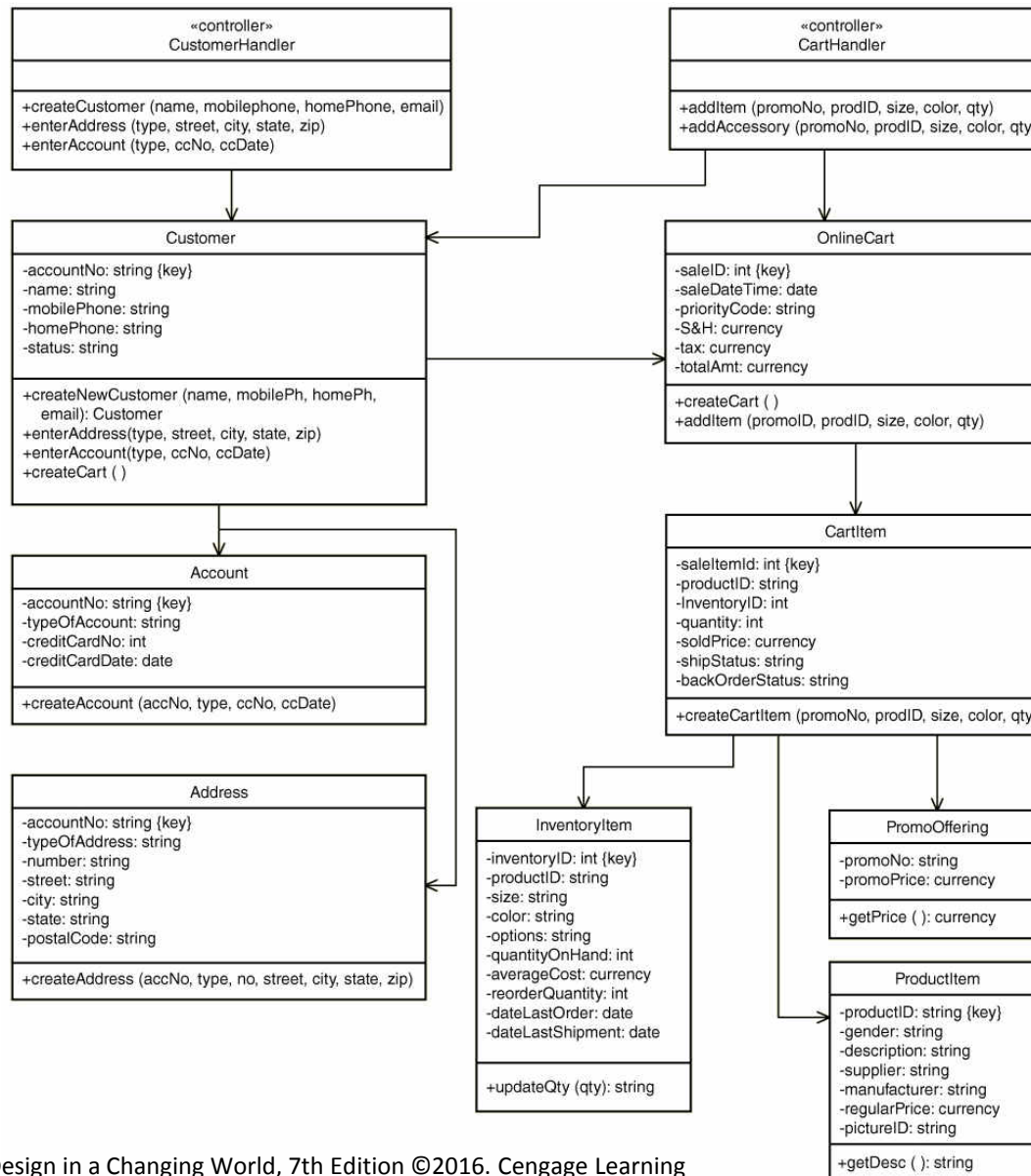
Example: *Fill Shopping Cart*



addAccessory
message

(top part of
SD as before)

Example - Design class diagram updated with methods



Summing up...

- To define the **methods** that are required in the design class diagram, each use case is considered in turn:
- The systems sequence diagram (SSD) from analysis is expanded into a first-cut **sequence diagram** by adding a use case controller and then the domain classes for the use case
- The internal messages and returns that result from the user case are determined and added to the SD
- The methods are then added to the design class diagram

Multilayer Sequence Diagrams



Design steps

- Begin with the models from analysis
- Work with a single use case at a time
- Develop first-cut design class diagram
- Identify and define the methods required in each class (e.g. using sequence diagram)
 - First cut sequence diagram
 - **Multilayer sequence diagram**
- Update the design class diagram
- Continue for additional use cases
- Partition classes into packages as appropriate

Multi-layer Sequence Diagrams



Murdoch
UNIVERSITY

- The first-cut sequence diagram shows the classes relevant to the business classes of the use case – the domain layer
- We can add classes that explicitly handle the view and data layers as well
- Add the *view layer* for input screens to handle input messages
- Add *data access layer* to read and write the data



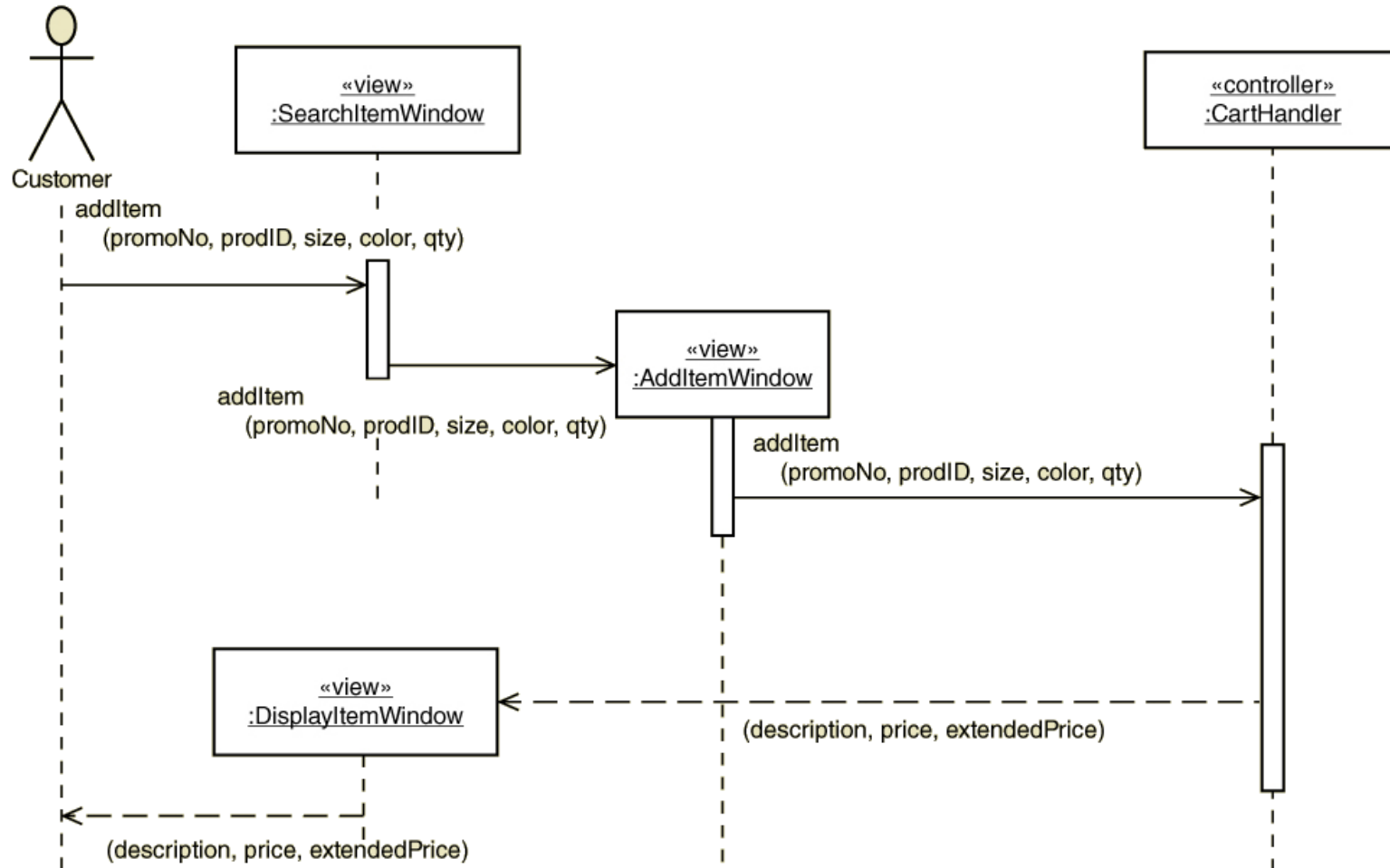
View layer class responsibilities

- Display electronic forms and reports
- Capture input events such 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

View layer



- Partial example: *Fill shopping cart* use case



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

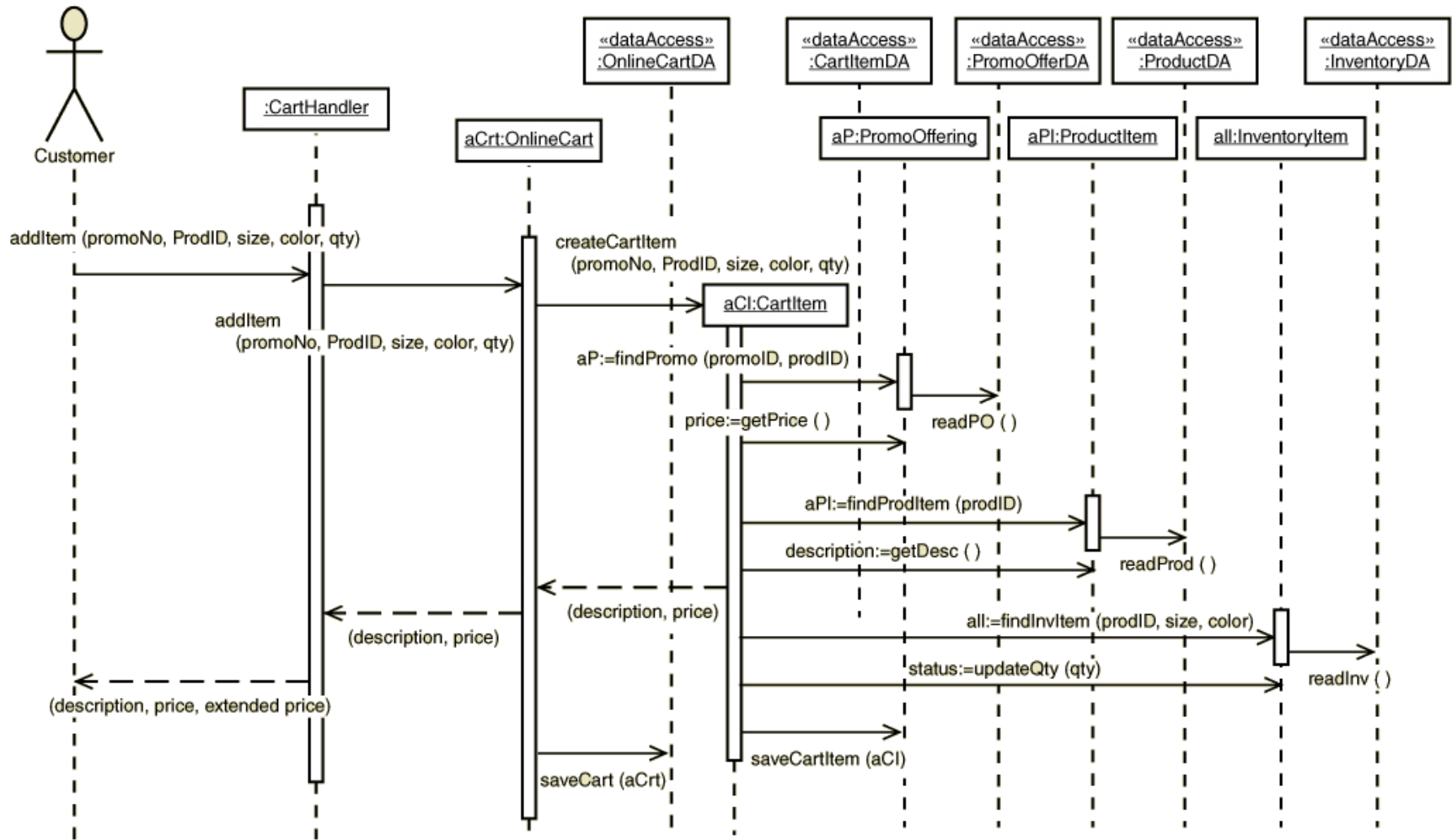


- Establish and maintain connections to the database
- Contain all SQL statements
- Process result sets (the results of SQL executions) into appropriate domain objects
- Disconnect gracefully from the database

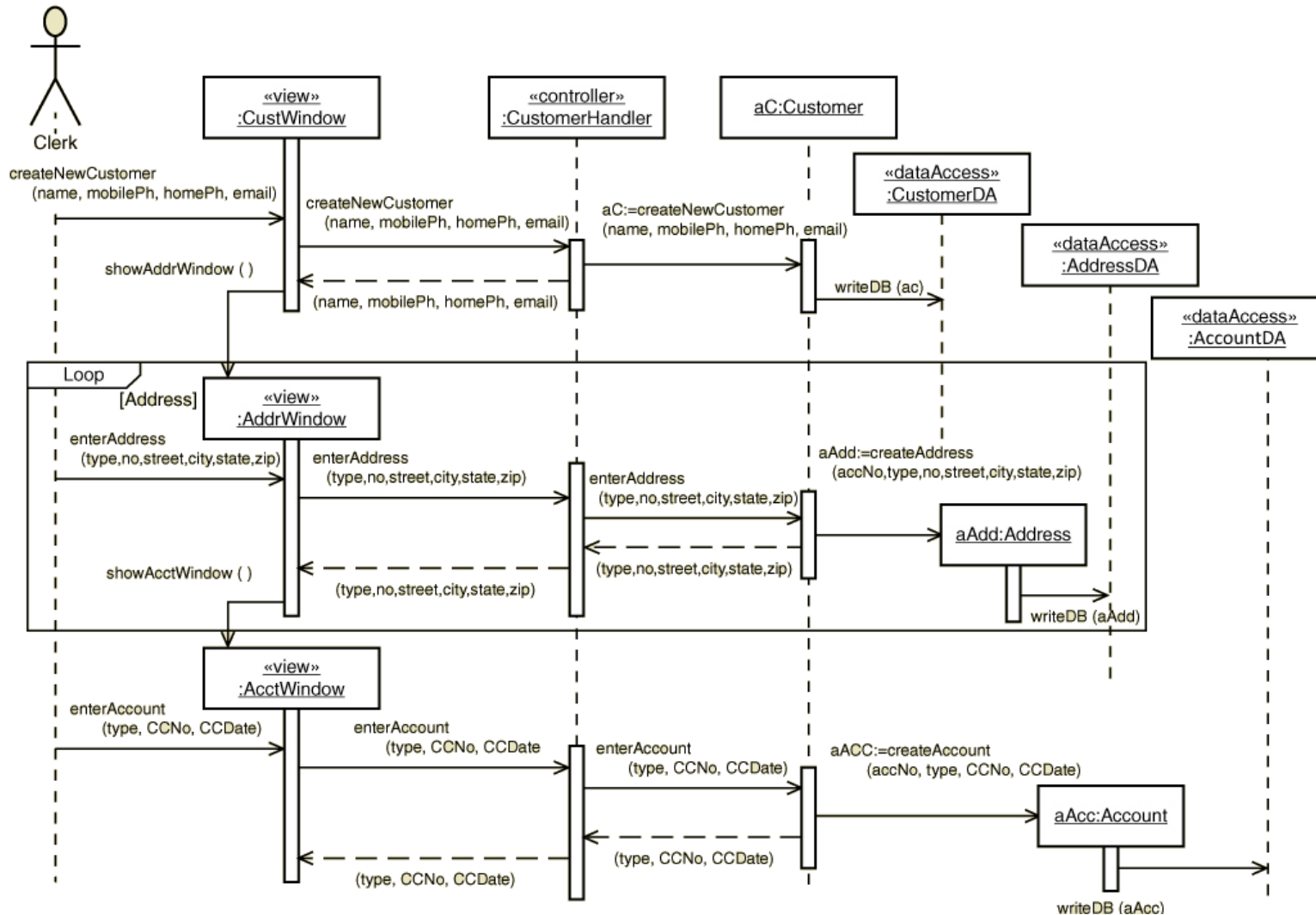
Data access layer



- Partial example: *Fill shopping cart* use case



View layer, domain layer, data access layer



Summing up...

- The first-cut sequence diagram shows the classes relevant to the business classes of the use case – the domain layer
- We can add classes that explicitly handle the view (boundary) and data layers of a three-layer architecture as well:
 - Add the *view layer* for input screens to handle input messages
 - Add *data access layer* to read and write the data

Package diagrams



Design steps

- Begin with the models from analysis
- Work with a single use case at a time
- Develop first-cut design class diagram
- Identify and define the methods required in each class (e.g. using sequence diagram)
 - First cut sequence diagram
 - Multilayer sequence diagram
- Update the design class diagram
- Continue for additional use cases
- **Partition classes into packages as appropriate**

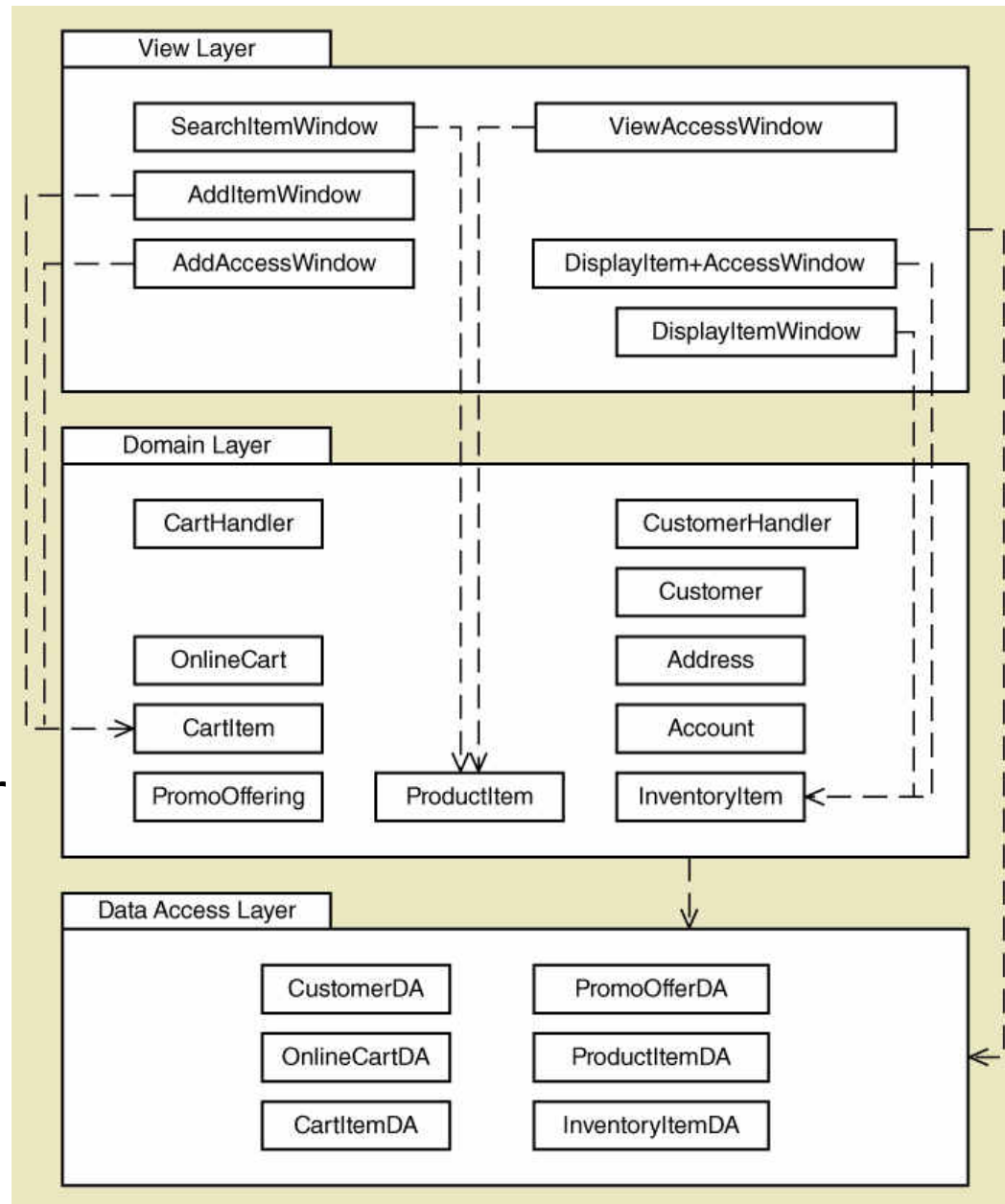


Package diagrams

- Can be used to define formal packages such as subsystems
- Can be used informally to group classes together for understanding
- Dependency relationship:
 - A relationship between packages (or classes within a package) in which a change of the independent component may require a change in the dependent component
 - Indicated by dashed line with arrow
 - Arrow head points to independent element, i.e. $A \rightarrow B$ means A depends on B

Package Diagram

- Three-layer package diagram of classes in previous slide
- Dependencies:
 - View layer depends on Data Access layer
 - SearchItemWindow depends on ProductItem
- etc.



Topic learning outcomes

After completing this topic you should be able to:

- Explain the purpose and objectives of object-oriented design
- Explain how object-oriented programs work by interacting objects sending messages
- Develop first-cut and final design class diagrams
- Develop domain-level sequence diagrams to model object behaviour
- Explain the different types of objects and layers in an object-oriented design
- Briefly describe some fundamental principles of object-oriented design

What's next?

We've now concluded our coverage of the activities involved in systems design. In the next topic, we'll look at activities relating to building and testing the system, and deploying the completed system in the organisation.