CSCI 4050/6050
Software Engineering

Object-Oriented Design

Use-Case Realization

Dr. Eman Saleh

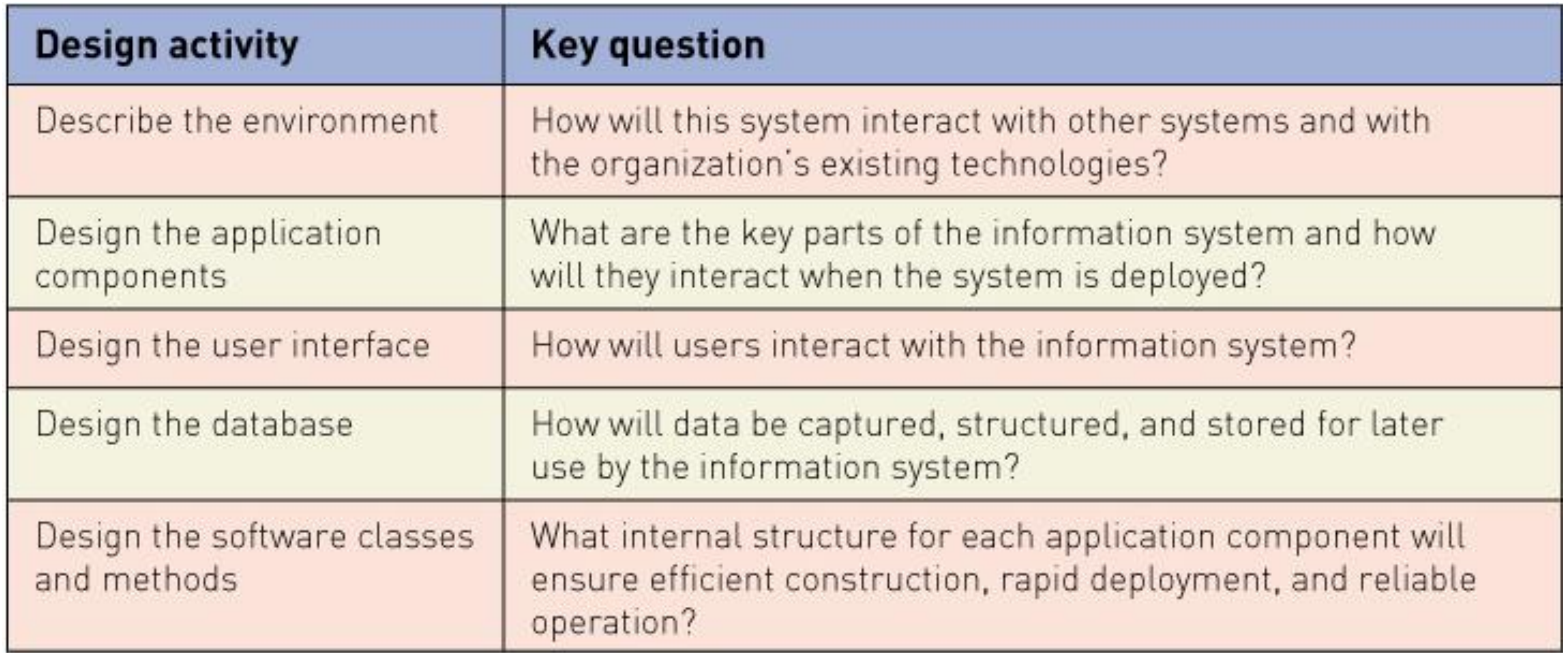
eman.saleh@uga.edu

Outline

- Object-Oriented Design: Bridging from Analysis to Implementation
- Steps of Object-Oriented Design
- Design Classes and the Design Class Diagram

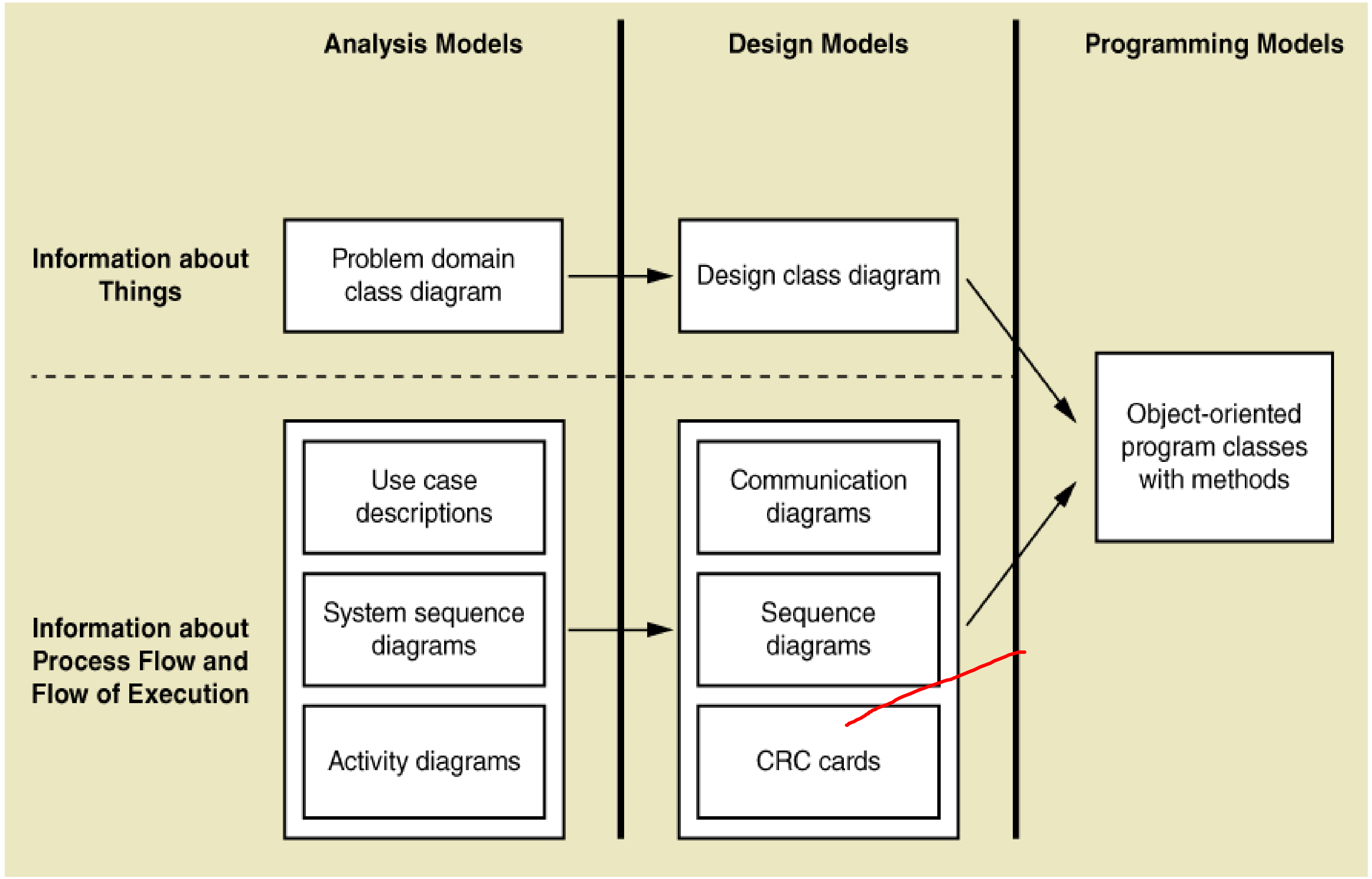
Overview

- Design models are based on the requirements models learned in previous lectures
- The steps of object-oriented design are explained
- The main model discussed here is the design class diagram



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?

Analysis to Design to Implementation: Model Flow

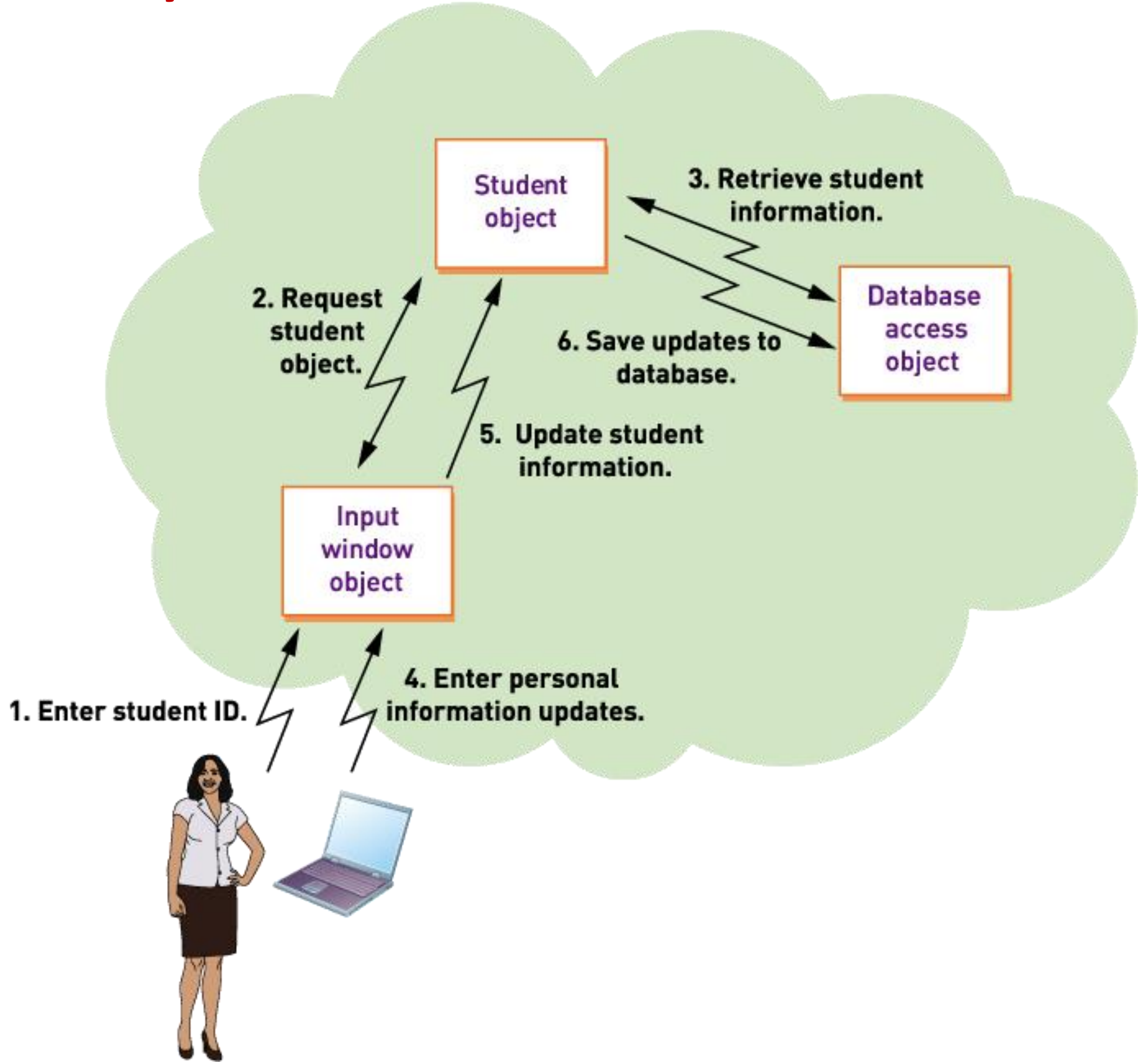


OO Design:
Bridging from Analysis to Implementation

- 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 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
4. Enter personal information updates.
1. Enter student ID.

Sample Java code with Methods

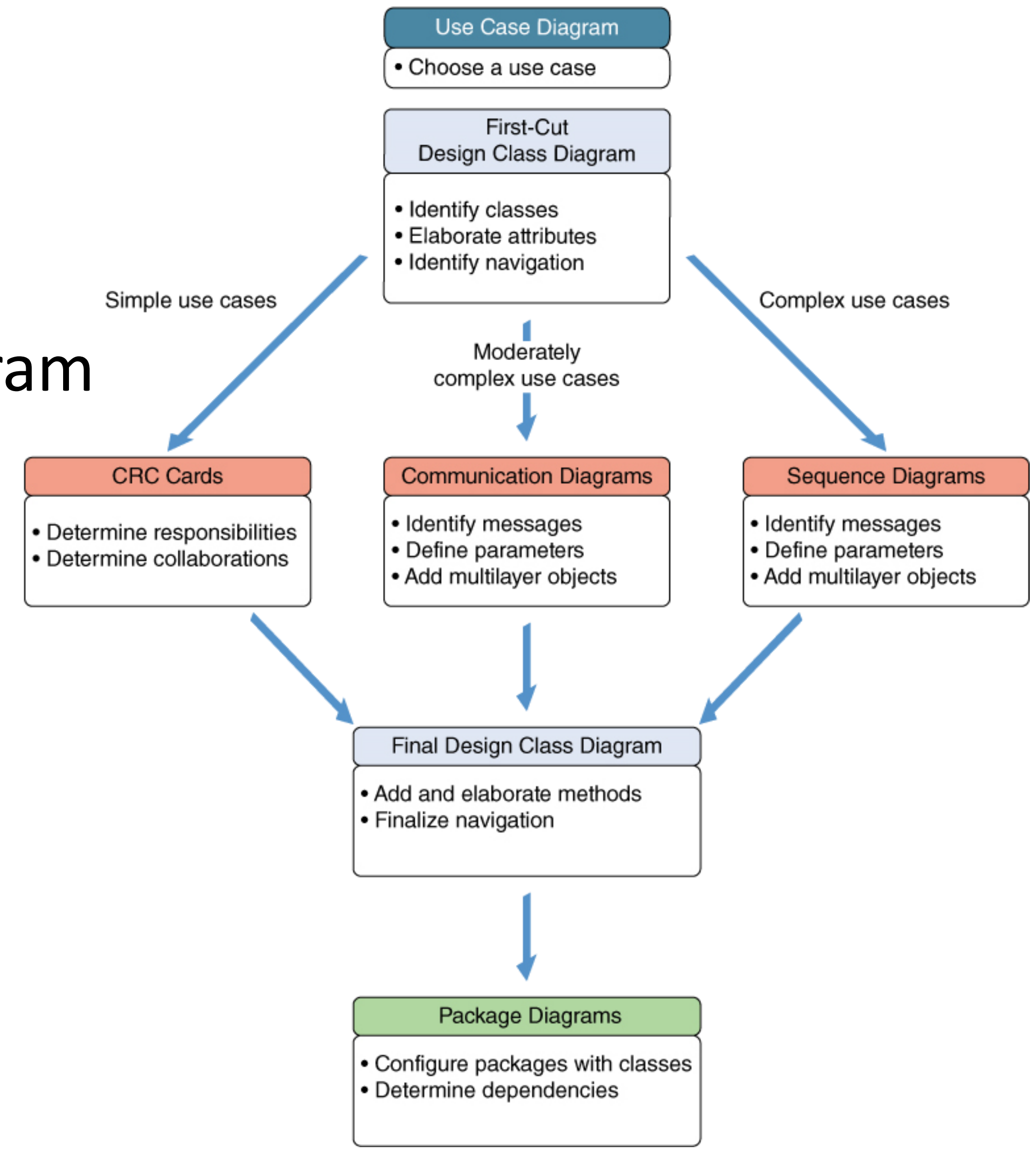
```java
public class Student
{
        //attributes
        private int studentID;
        private String firstName;
        private String lastName;
        private String street;
        private String city;
        private String state;
        private String zipCode;
        private Date dateAdmitted;
        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;
        }
```

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

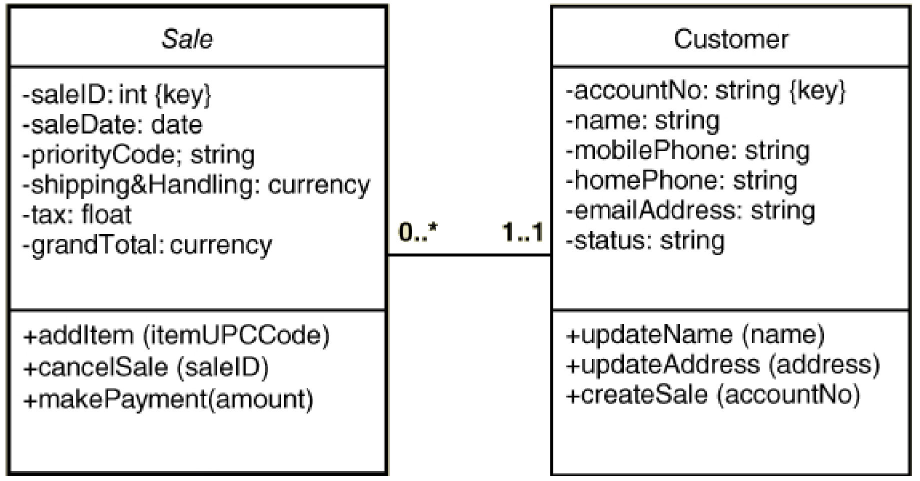
Steps of Objects Design

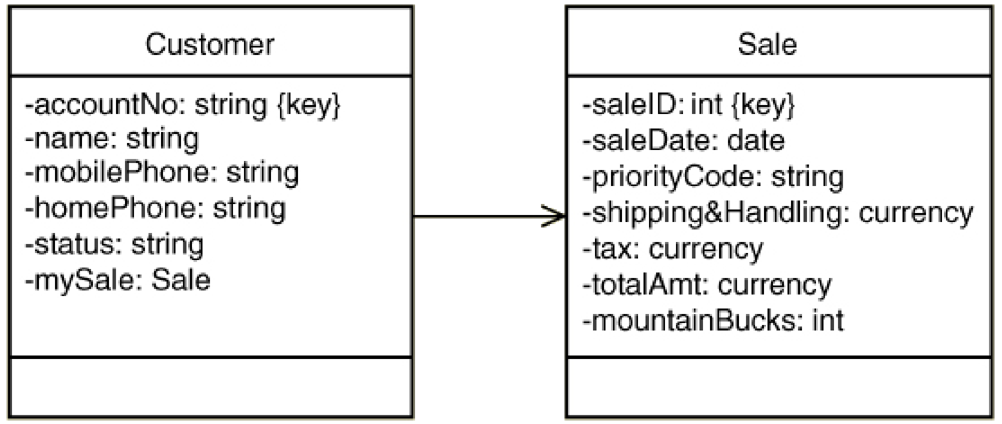
- First-cut DCD
- Extend use case with Communication Diagram
 or Sequence diagram
- Final DCD
- Package diagram



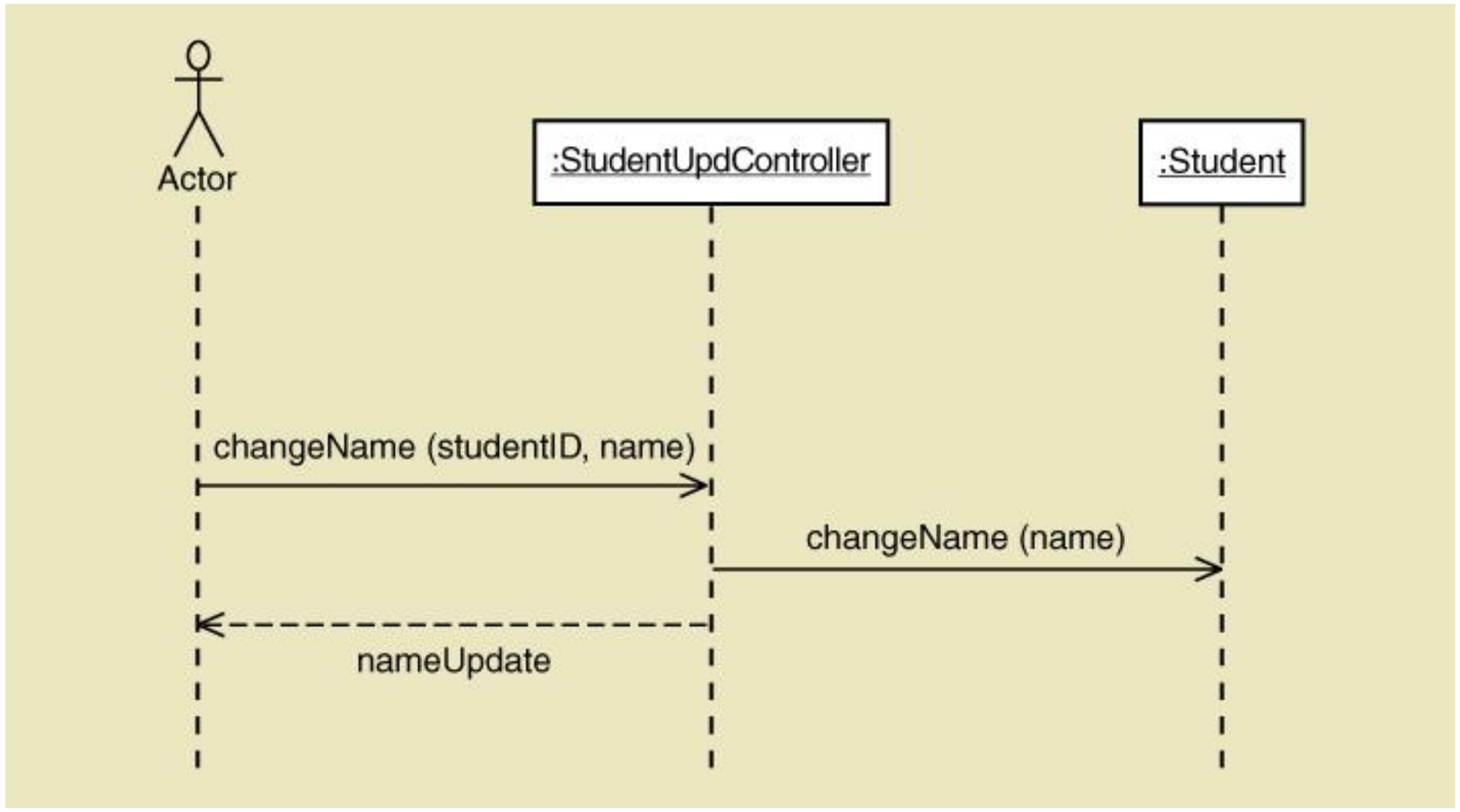
Step 1: develop the First Cut Design Class Diagram.

- **Consider one use case.**
- **Start with analysis models, and Domain class diagram and a use case description or a system sequence diagram**

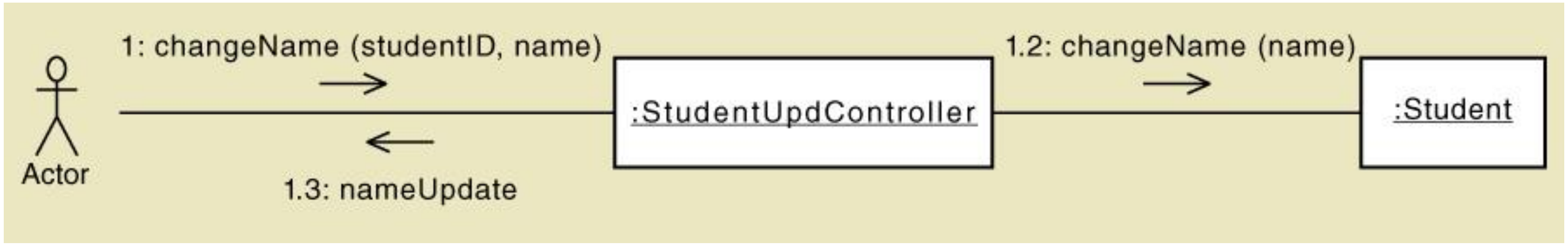




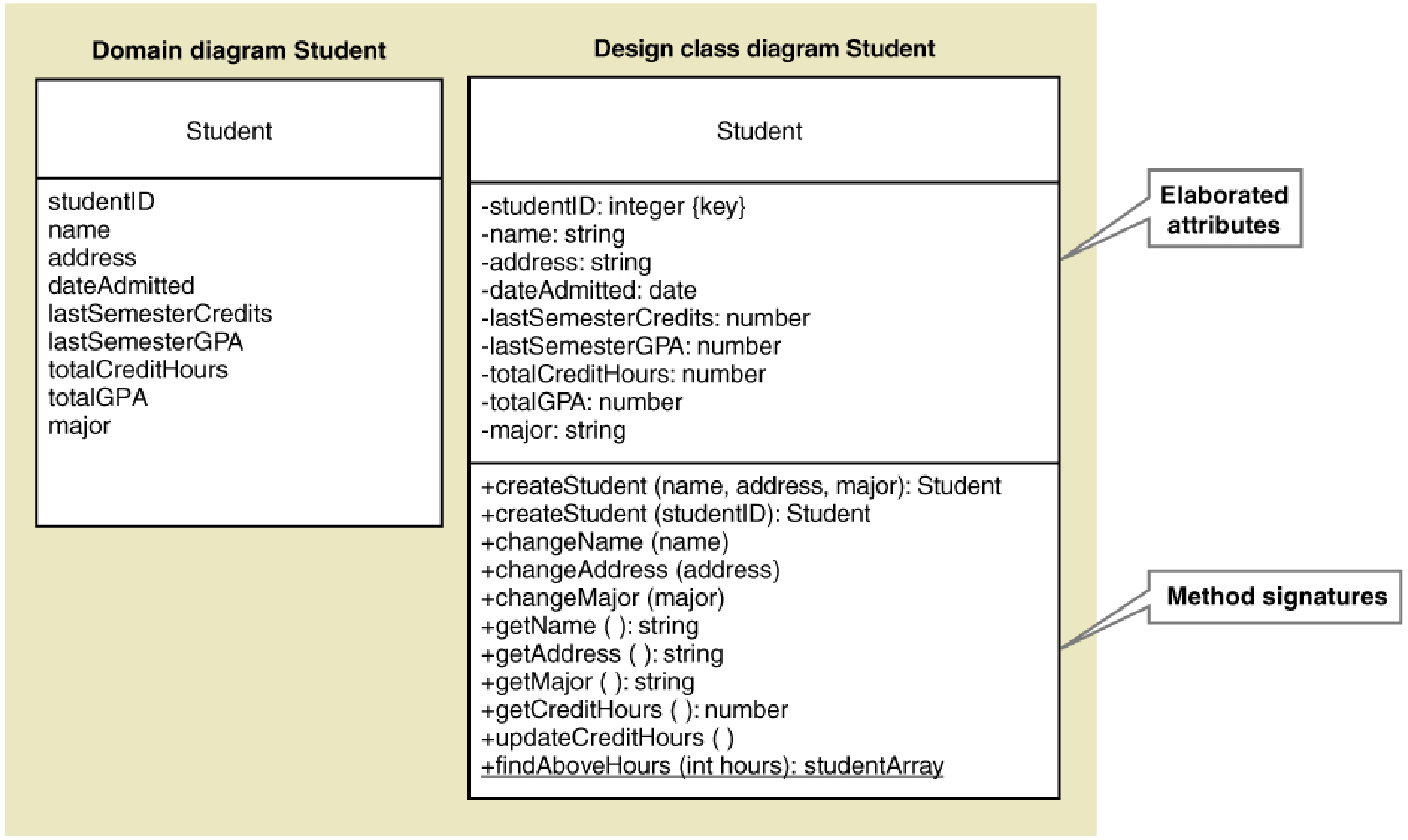
Step2: Analyze with an interaction diagram(Sequence Diagram or a communication diagram)



A Communication Diagram



Step 3: Develop the Design Class Diagram.

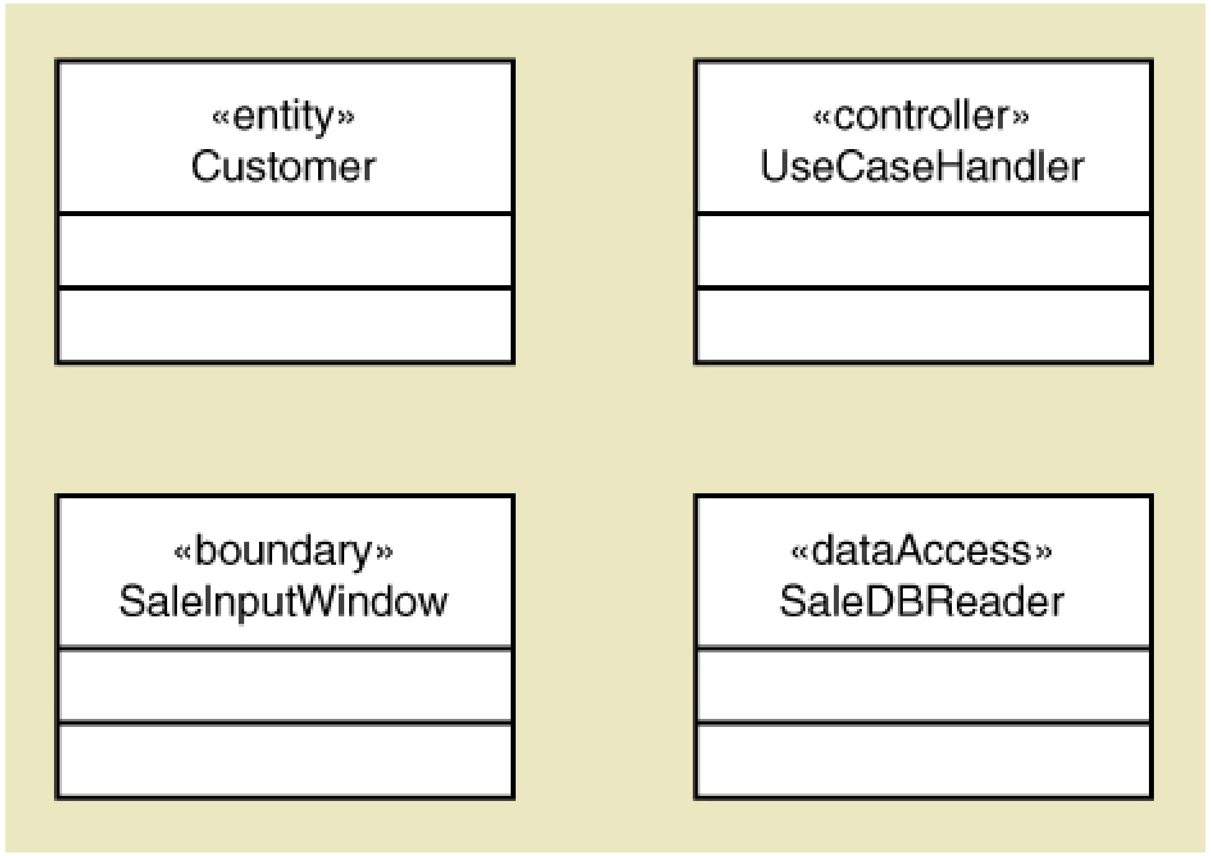


Design Class Diagrams (revisited)

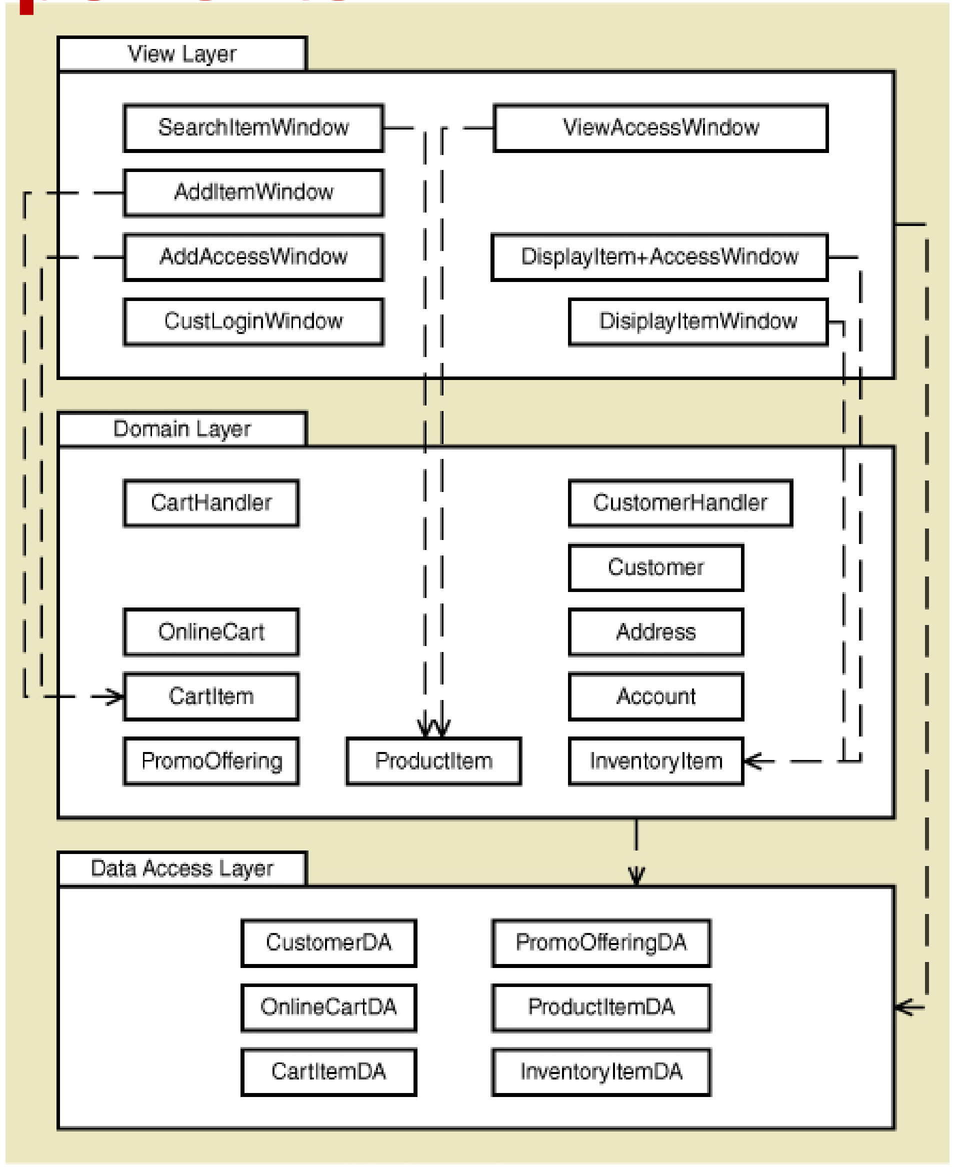
stereotype a way of categorizing a model element by its characteristics, indicated by guillemots (<< >>)

- **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

Design Class Stereotypes



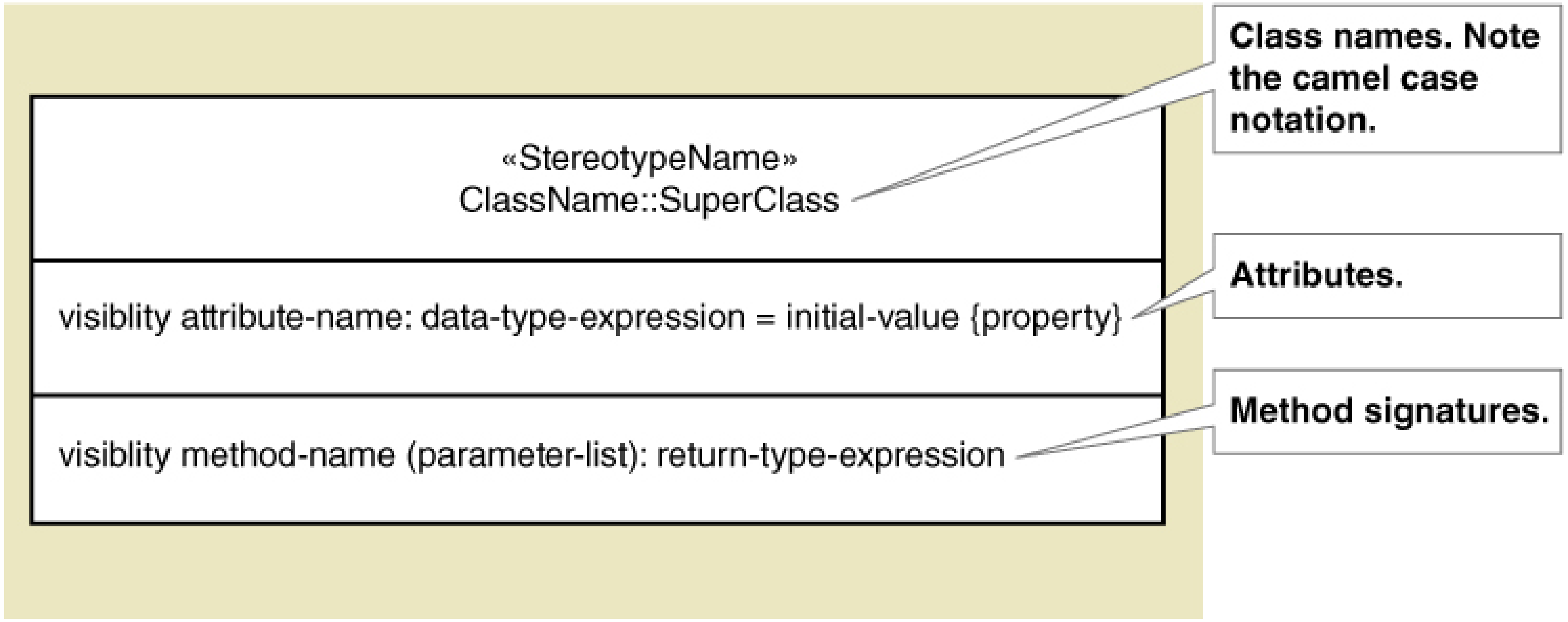
Typical models for defining application components



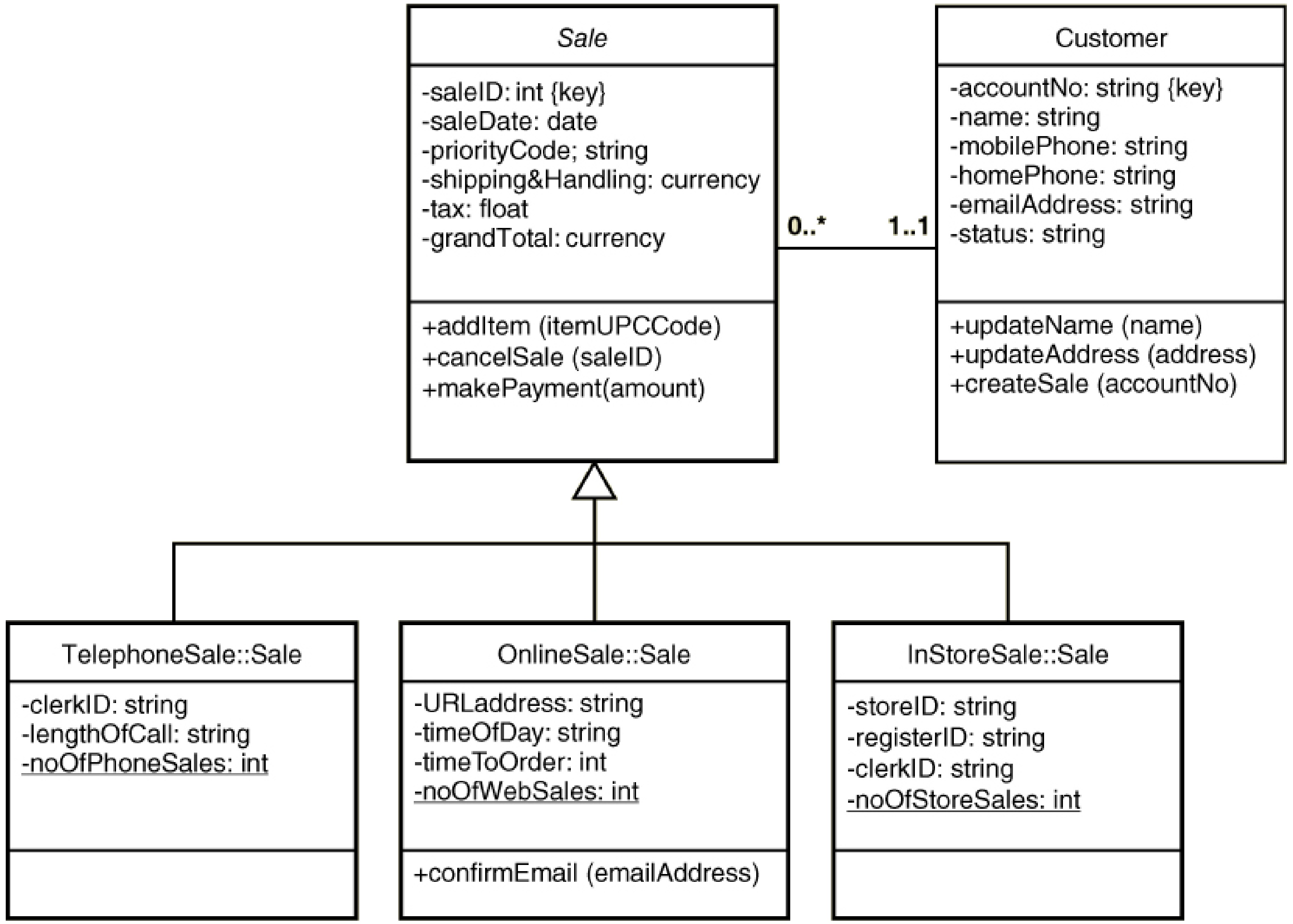
Package diagram

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.

Notation for Design Classes

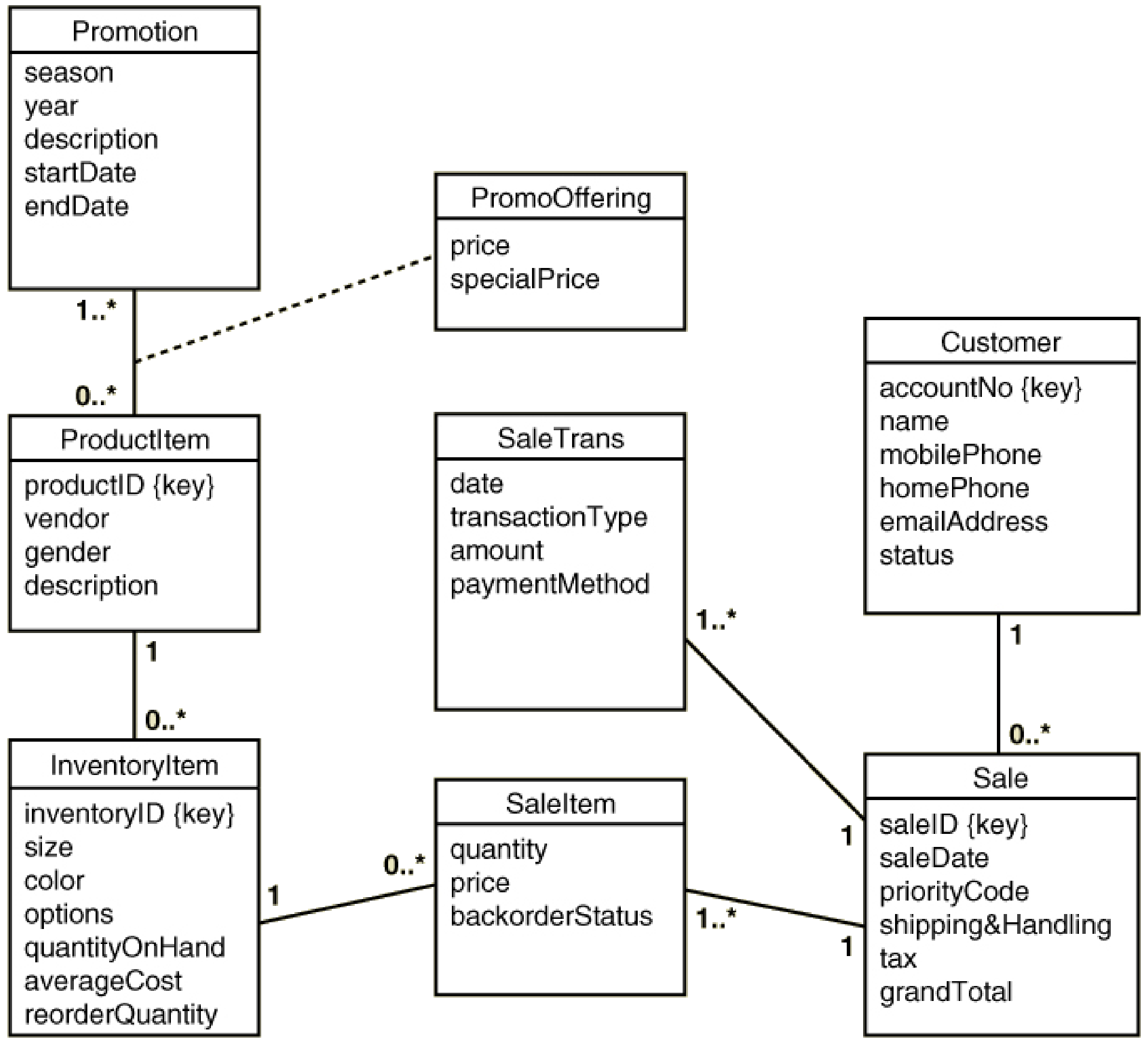


Step 1: First-Cut Design Class Diagram

- 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

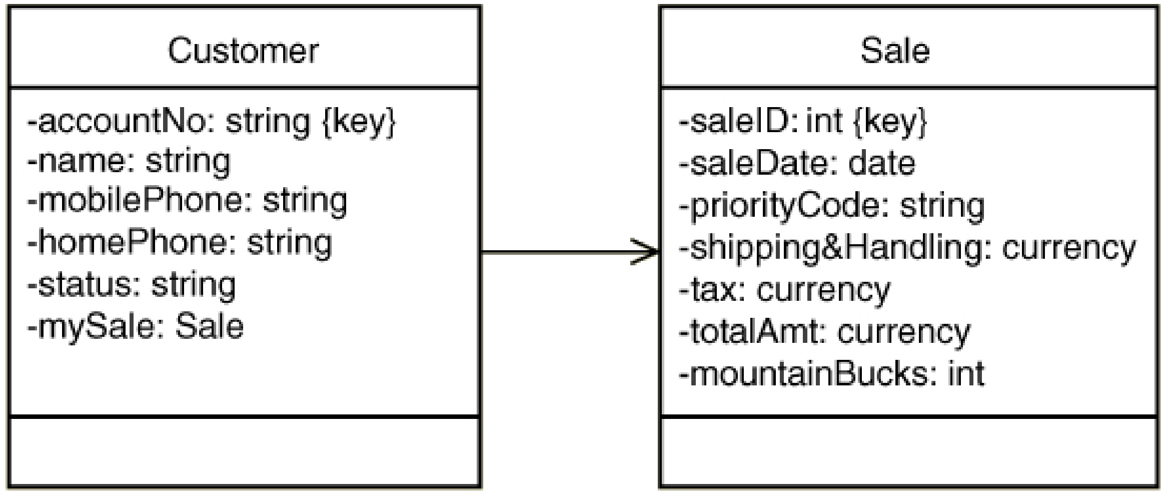
Start with Domain Class Diagram

RMO Sales Subsystem



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

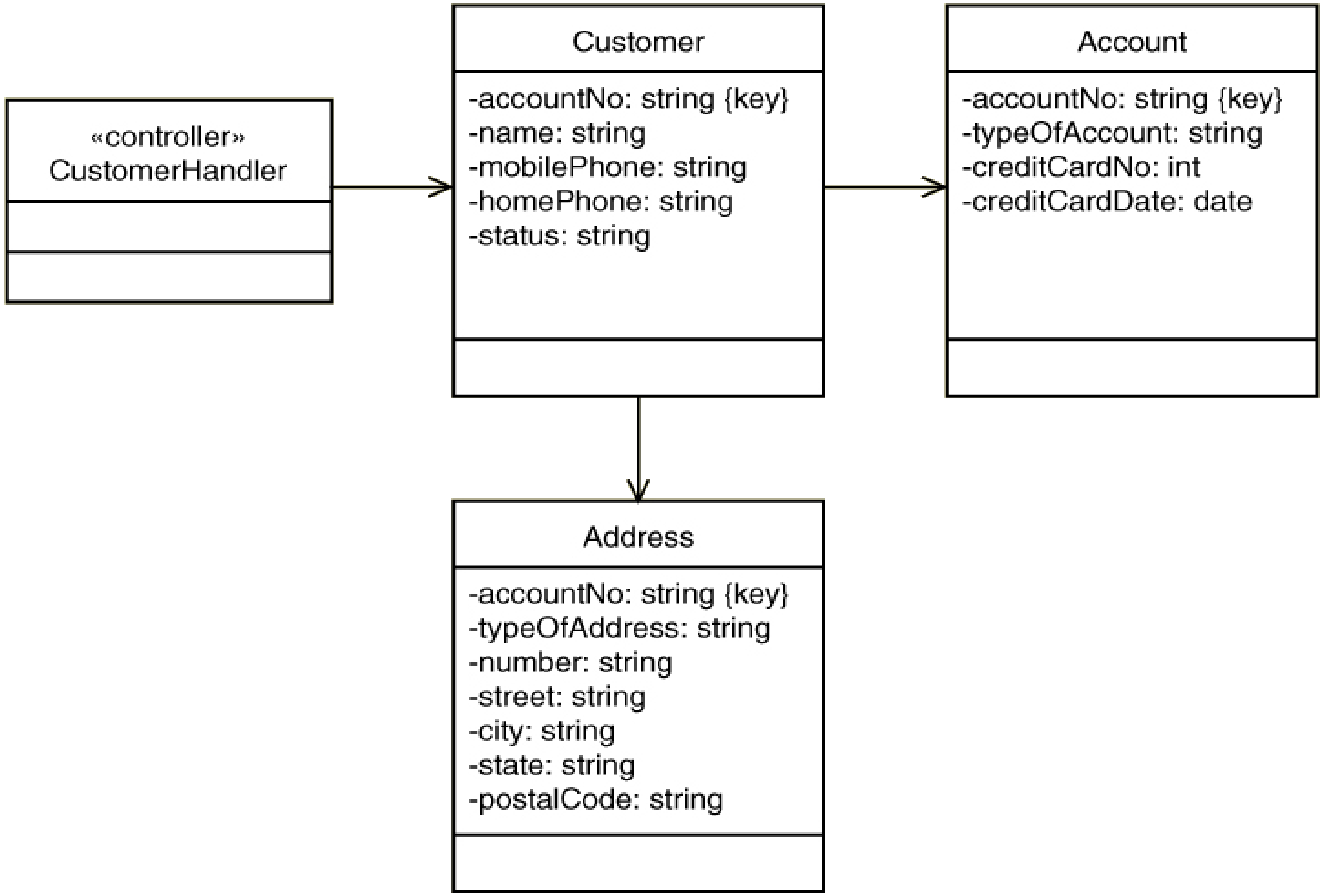


Navigation Visibility Guidelines

- **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.

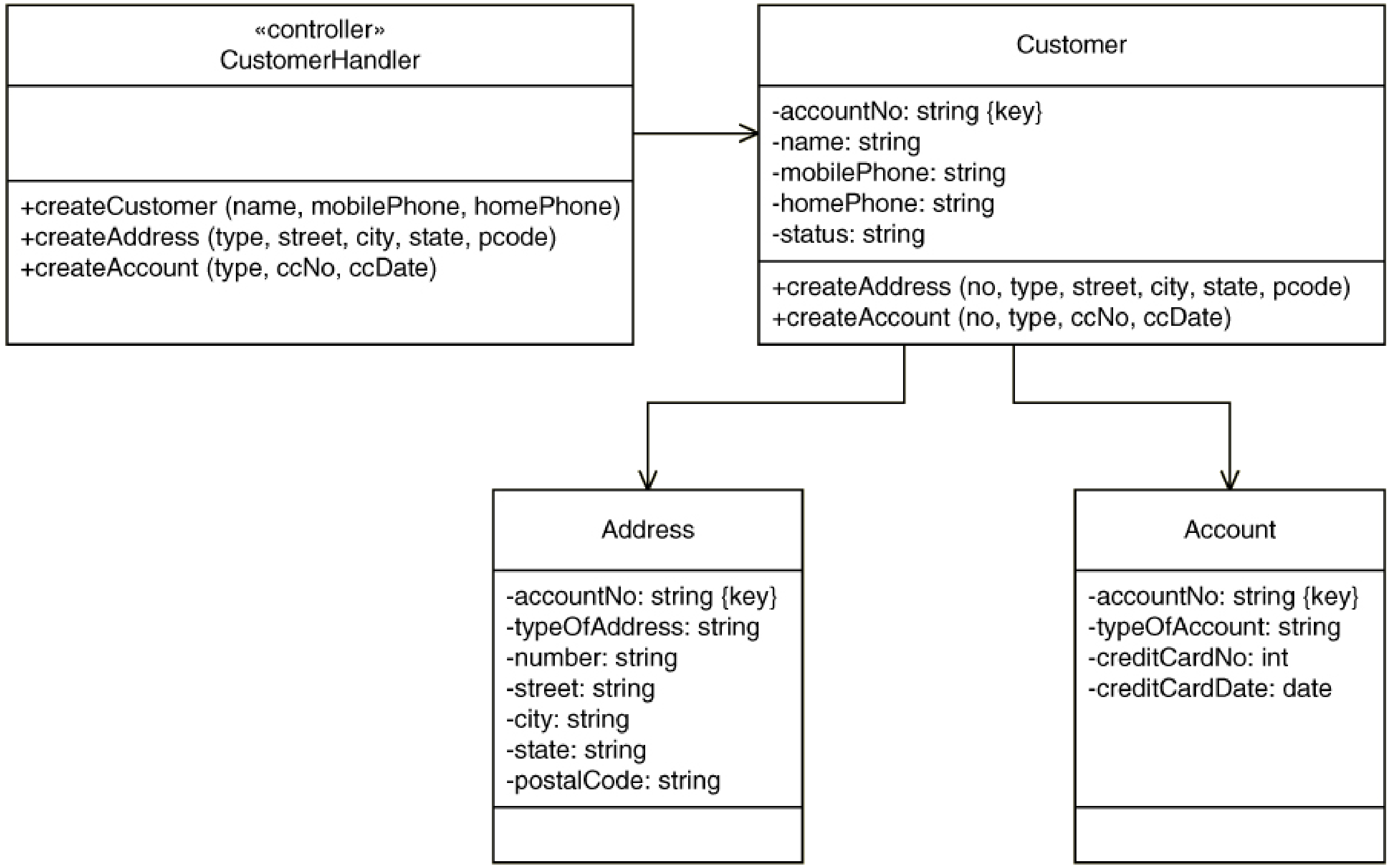
Example: *Create customer Account*

- First-cut DCD



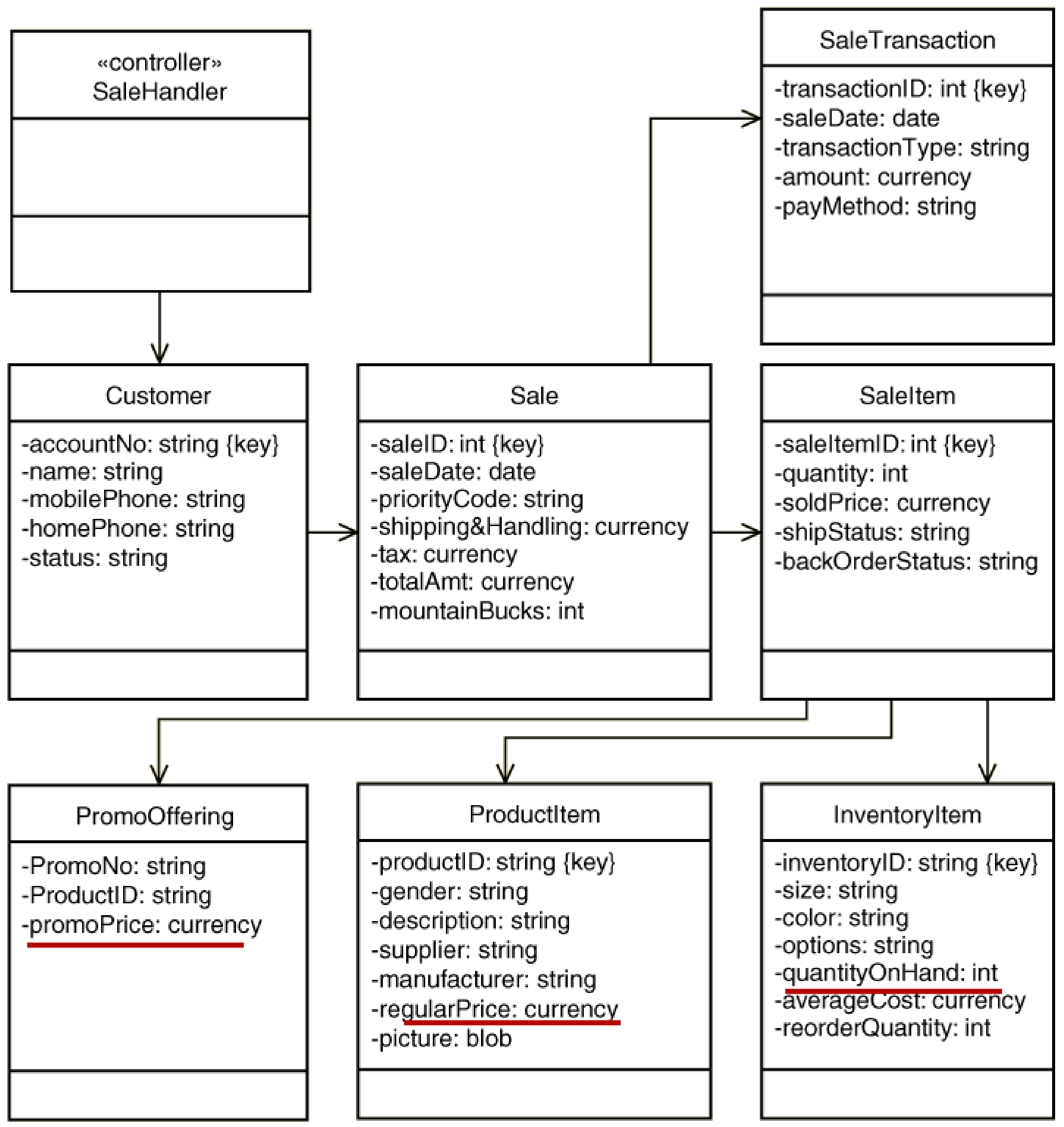
Example: *Create customer account*

- Design Class Diagram (DCD), Finfing navigation arrows



Create First Cut Design Class Diagram

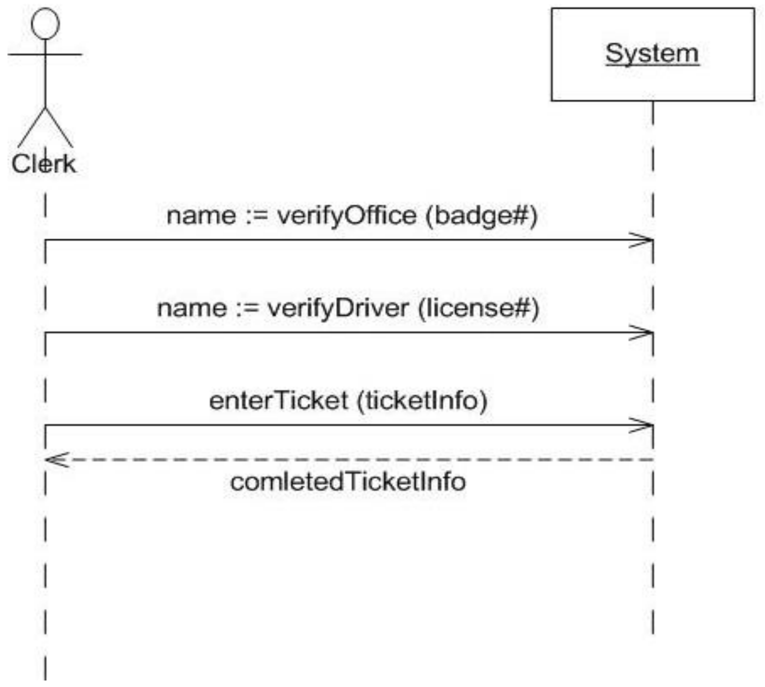
Use Case *Create telephone sale* with controller added

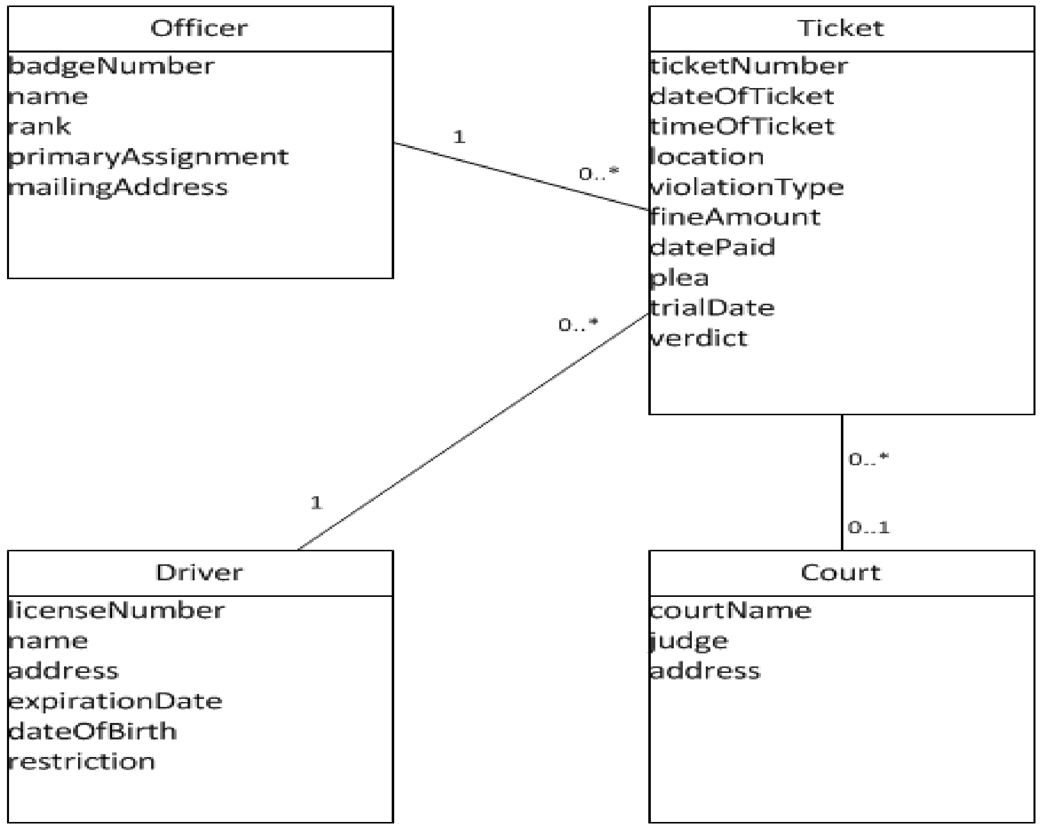


State Patrol System

Draw a first cut design class diagram for use case Record new Ticket

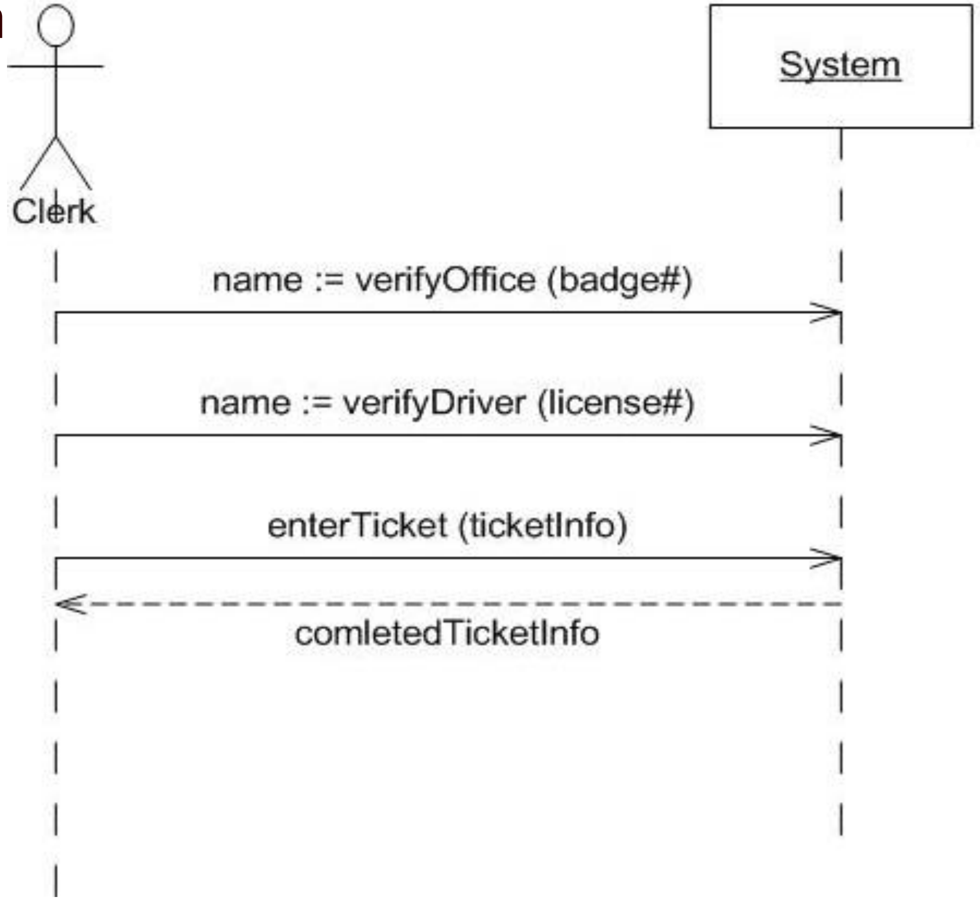
Use your previous model(s): Use case description, System sequence diagram or activity diagram and the domain model

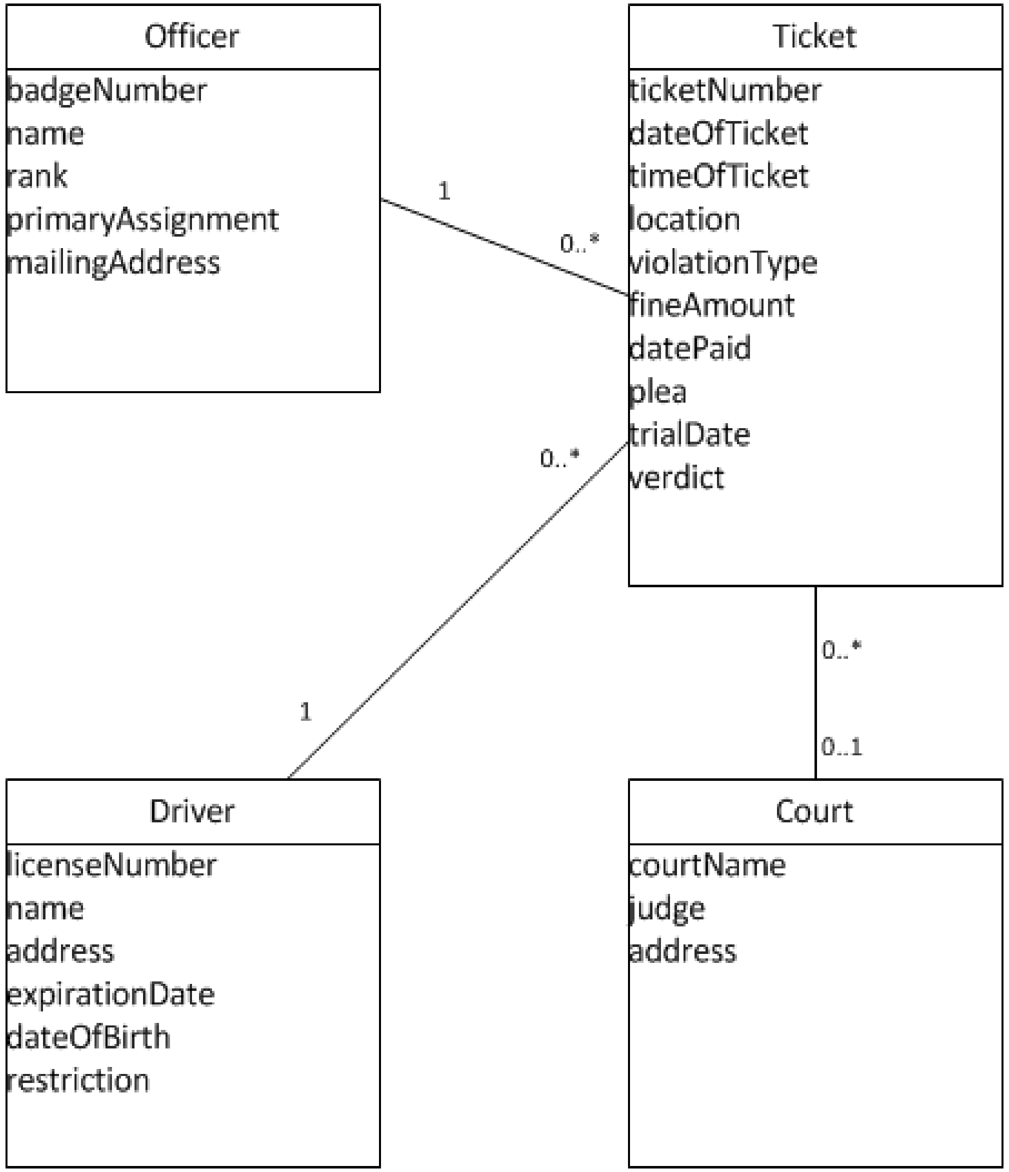

Clerk
System
name := verifyOffice (badge#)
name := verifyDriver (license#)
enterTicket (ticketInfo)
comletedTicketInfo



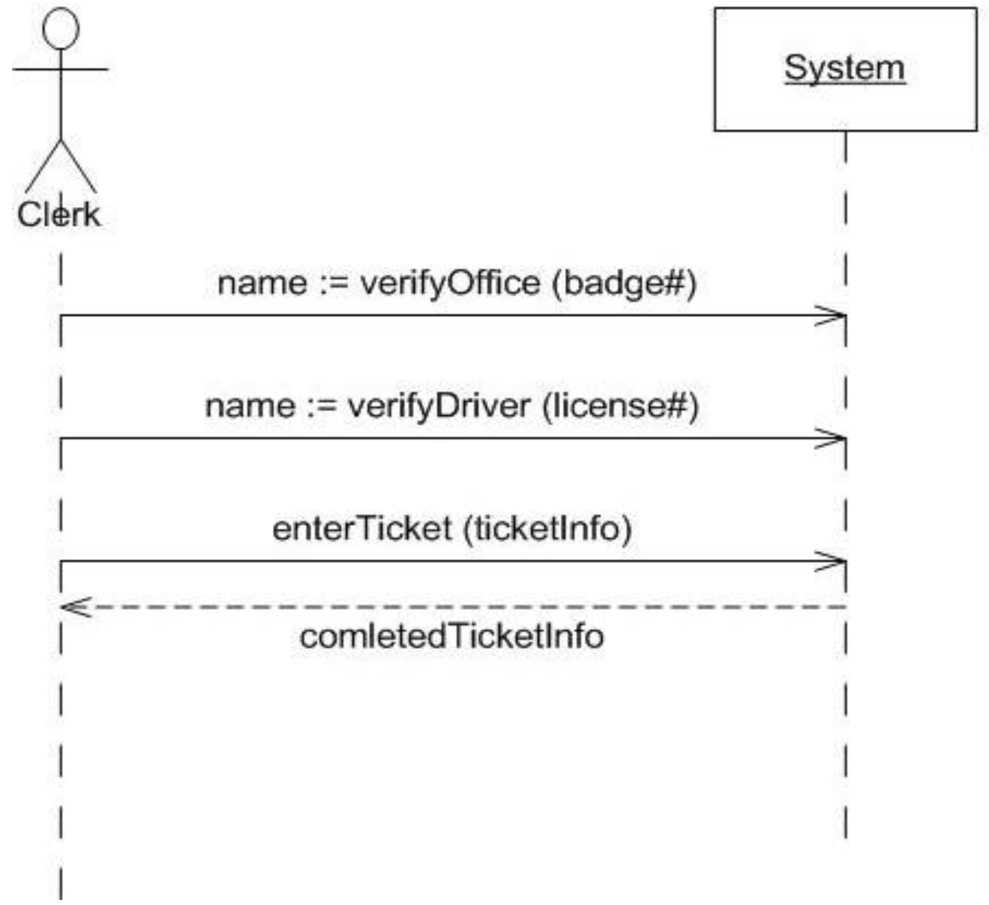
State Patrol System
Step 1:

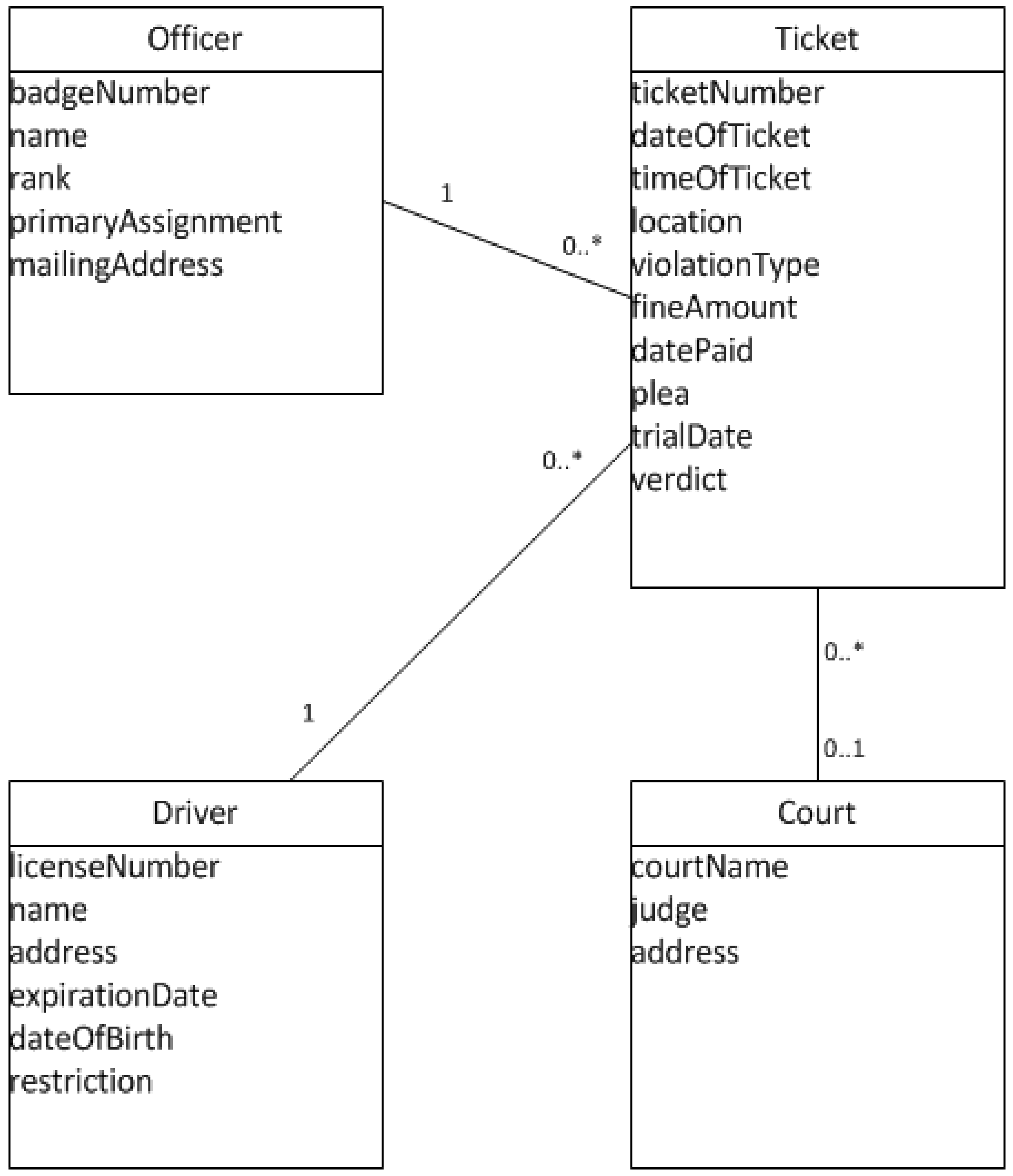
- Step1: First cut design class diagram
Add a controller and the navigation
a


Clerk
System
name := verifyOffice (badge#)
name := verifyDriver (license#)
enterTicket (ticketInfo)
comletedTicketInfo



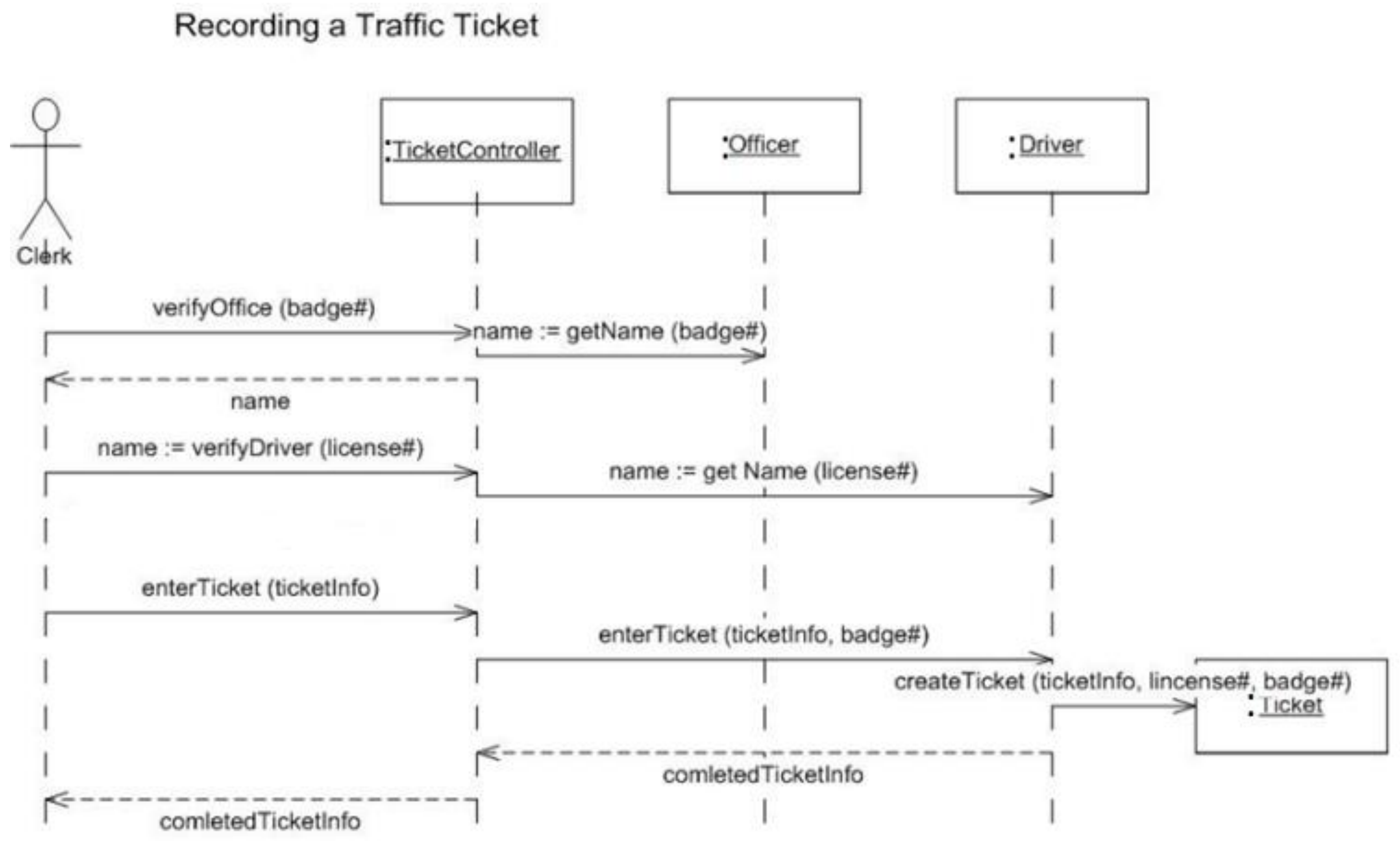
State Patrol System
Step 2: Develop interaction diagram


Clerk
System
name := verifyOffice (badge#)
name := verifyDriver (license#)
enterTicket (ticketInfo)
comletedTicketInfo

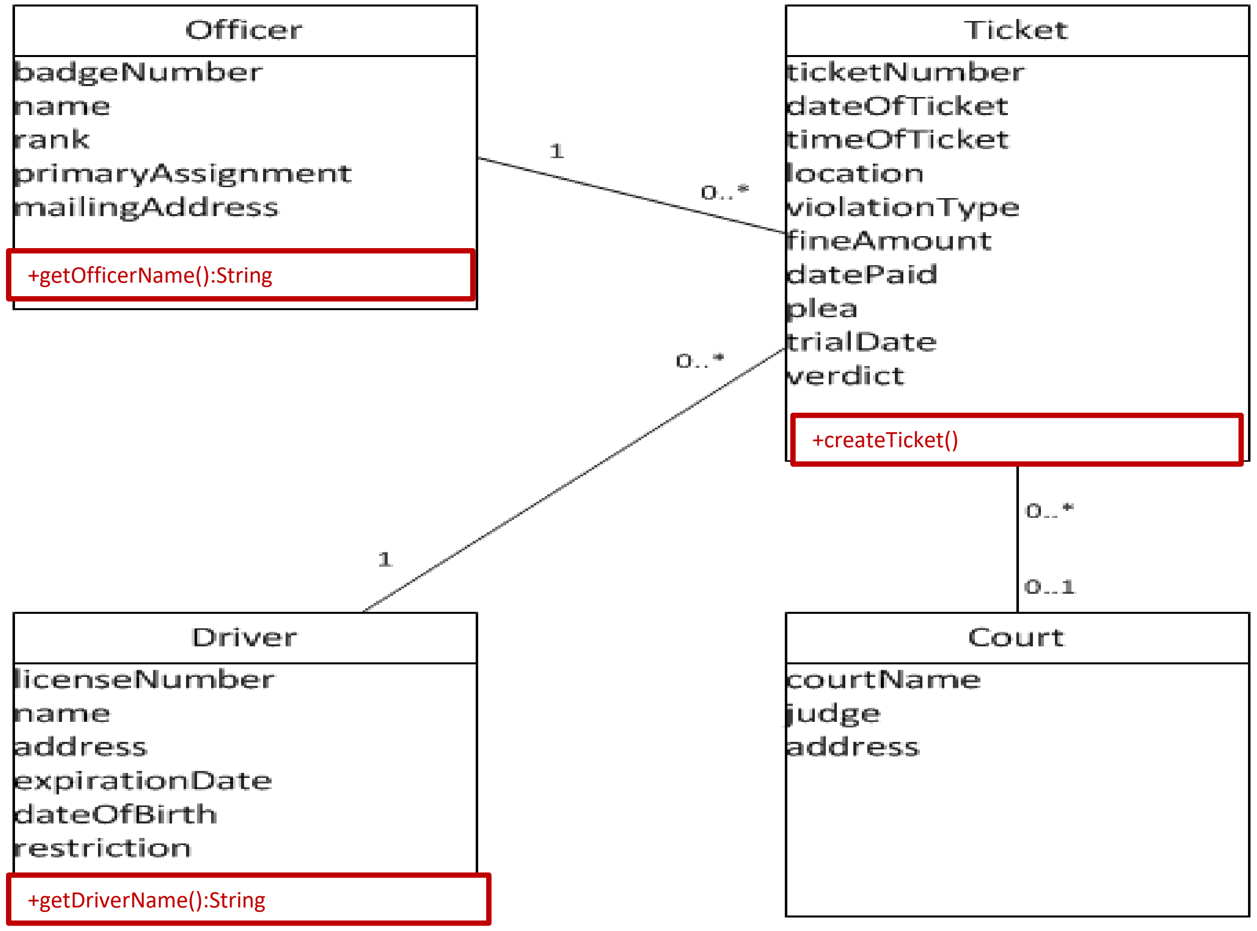


State Patrol System :Step 2: Develop the first cut sequence diagram

First cut sequence diagram

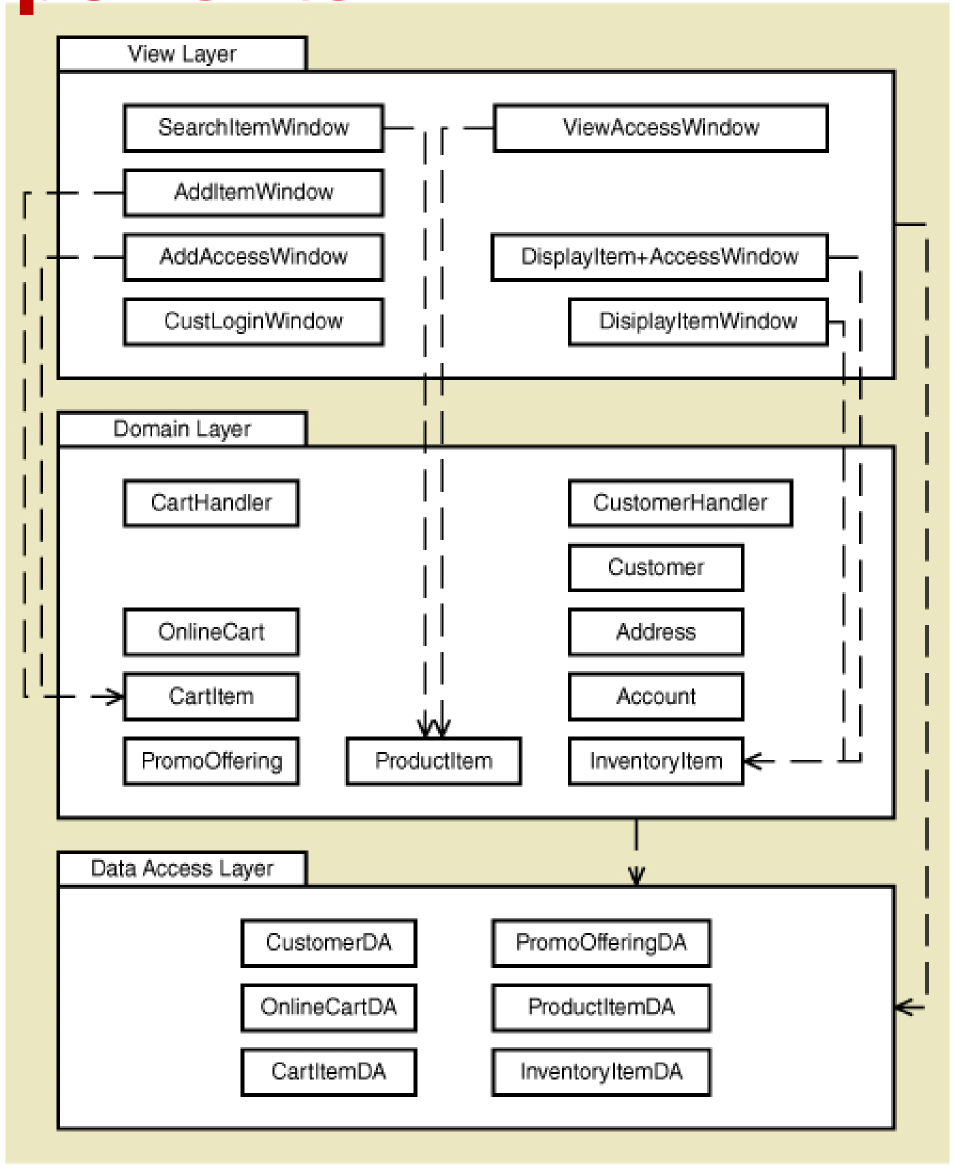

Recording a Traffic Ticket
Clerk
:TicketController
:Officer
:Driver
verifyOffice (badge#)
name := getName (badge#)
name
name := verifyDriver (license#)
name := get Name (license#)
enterTicket (ticketInfo)
enterTicket (ticketInfo, badge#)
createTicket (ticketInfo, lincense#, badge#)
:Ticket
comletedTicketInfo
comletedTicketInfo

Add discovered methods to the class diagram



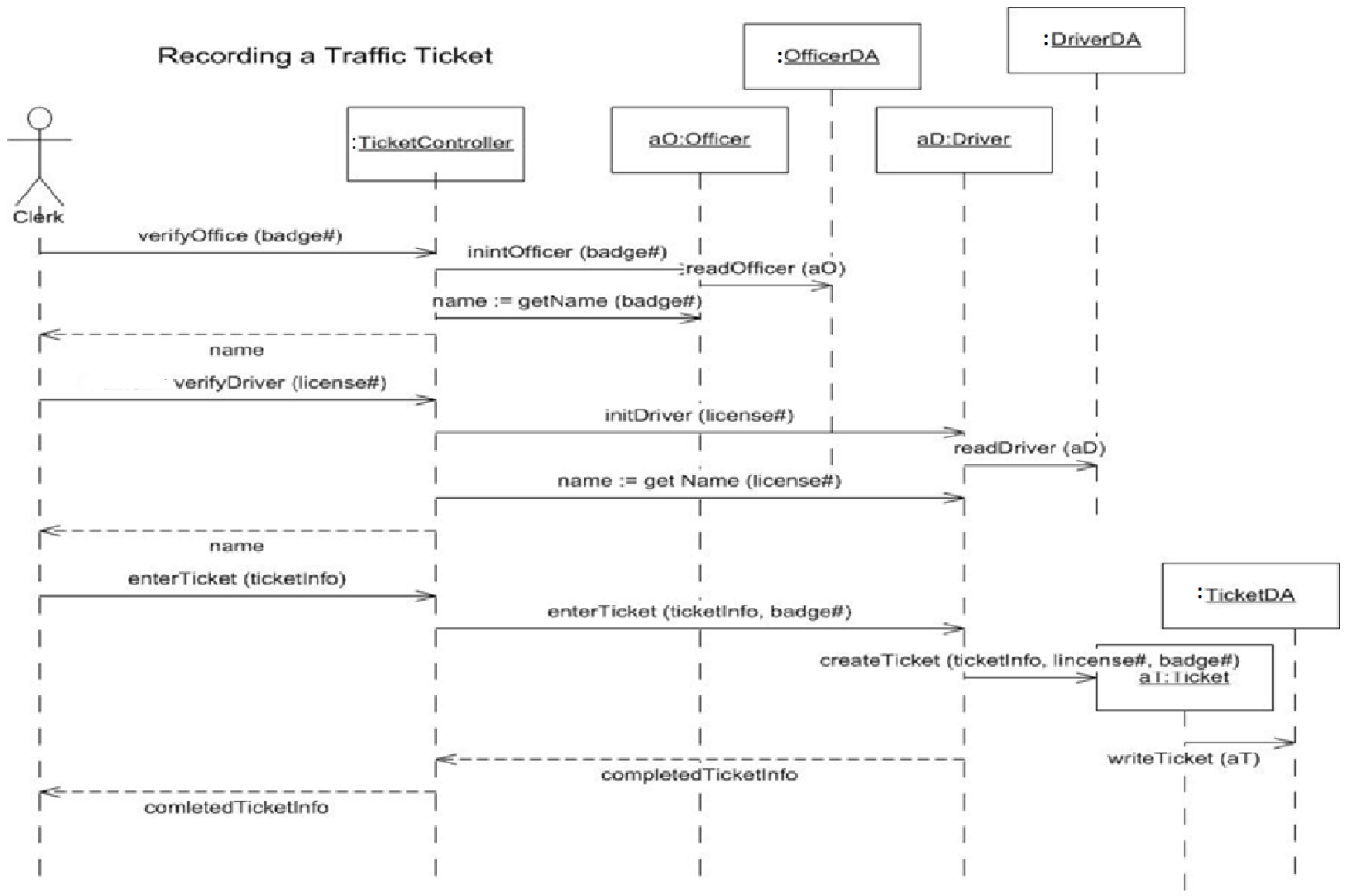
To develop a 3 layer Sequence diagram: Add a DA layer and a View Layer

Typical models for defining application components

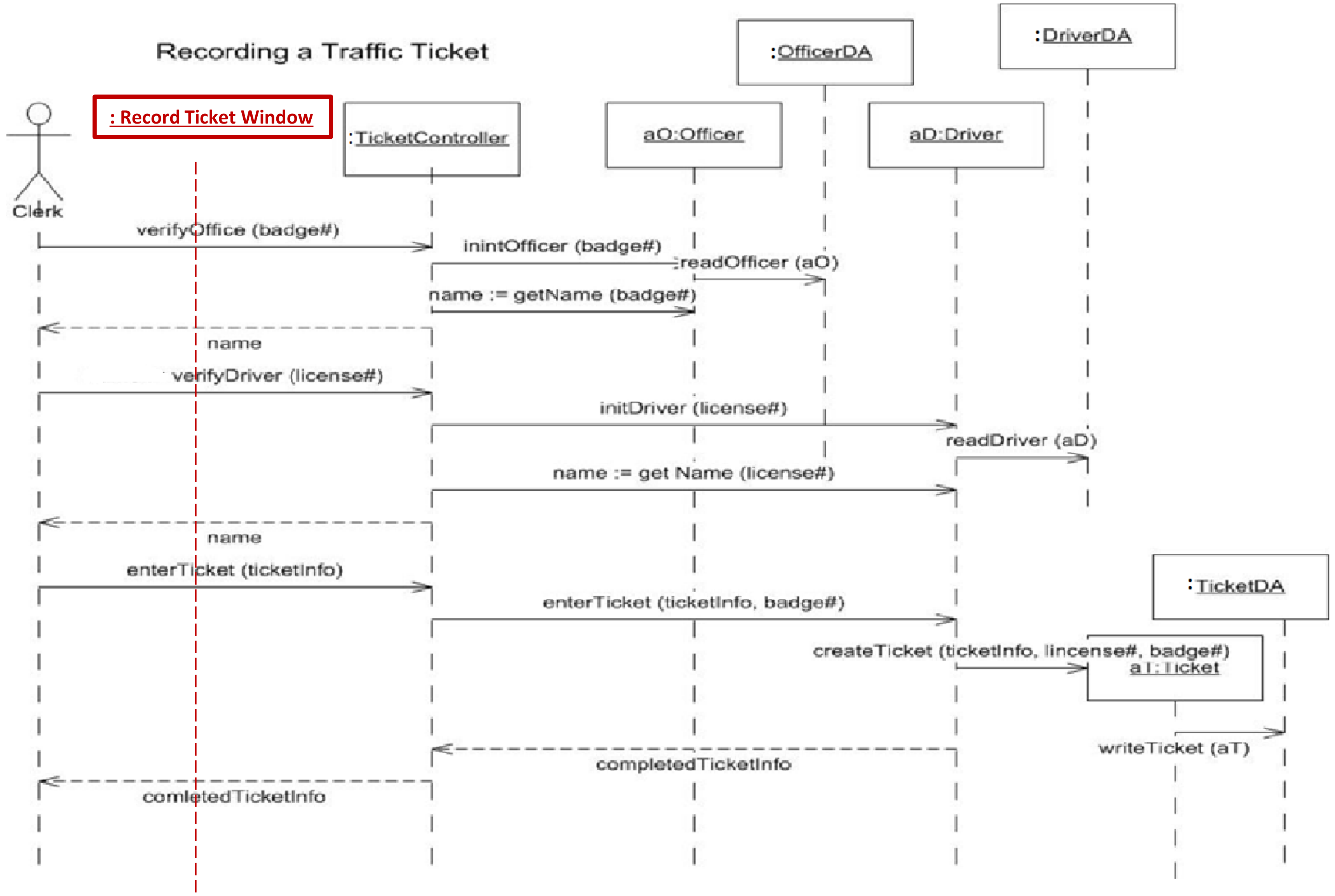


Package diagram

Add a DA layer



Add A view layer



CSCI 4050/6050
Software Engineering

Object-Oriented Design

Fundemental Design principles

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.
  - 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

- **Separation of Responsibilities**
  - AKA Separation of Concerns
  - Applied 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

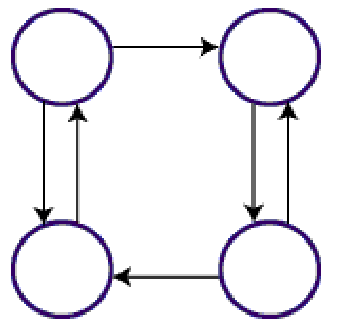
- **Protection from Variations**
  - A design principle states that parts of a system unlikely to change are separated (protected) from those that will surely change.
  - Separate user interface forms and pages that are likely to change from application logic
  - Put database connection and SQL logic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

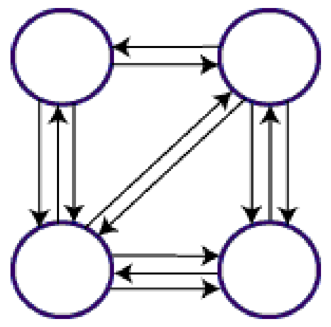
- **Indirection**
  - A design principle that states an intermediate class is placed between two classes to decouple them but still link them
  - A controller class between UI classes and problem domain classes is an example
  - Supports low coupling
  - Indirection is used to support security by directing 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:

- **Coupling**
  - A quantitative 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



Loosely Coupled:
Some dependencies



Highly Coupled:
Many dependencies

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

Coupling	Cohesion
Coupling is also called Inter-Module Binding.	Cohesion is also called Intra-Module Binding.
Coupling shows the relationships between modules.	Cohesion shows the relationship within the module.
Coupling shows the relative **independence** between the modules.	Cohesion shows the module's relative **functional** strength.
While creating, you should aim for low coupling, i.e., dependency among modules should be less.	While creating you should aim for high cohesion, i.e., a cohesive component/ module focuses on a single function (i.e., single-mindedness) with little interaction with other modules of the system.
In coupling, modules are linked to the other modules.	In cohesion, the module focuses on a single thing.

Levels of Coherence

- we can think about classes as having:
- very low cohesion
- low cohesion
- medium cohesion
- high cohesion.

Remember, high cohesion is the most desirable

Levels of Coherence (cont.)

- An example of very low cohesion is a class

that has responsibility for services in different functional areas, such as a class that accesses the Internet and a database.

- An example of low cohesion is a class that has different responsibilities but in related functional areas—for example, one that does all database access for every table in the database. It would be better to have different classes to access customer information, order information, and inventory information.

Levels of Coherence (cont.)

- An example of medium cohesion is a class that has closely related responsibilities, such as a single class that maintains customer information and customer account information.
- Two highly cohesive classes could be defined: one for customer information, such as names and addresses, and another class or set of classes for customer accounts, such as balances, payments, credit information, and all financial activity.
- If the customer information and the account information are limited, they could be combined into a single class with medium cohesiveness.
- Either medium or highly cohesive classes can be acceptable in systems design.

In-Class Activity
The first five principles of OO design

- What is are the S.O.L.I.D Object Oriented design principles.

Define each briefly (1 to 2 lines)

Upload your answer to activity 7 on eLC (word, txt, or PDF document)

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)

- Decisions about design options are guided by some fundamental design principles. Some of these are coupling, cohesion, protection from variations, indirection, and object responsibility.