Chapter 12 : Object Oriented Design Fundamental

Tannaz R.Damavandi
Cal Poly Pomona

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

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: bringing 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

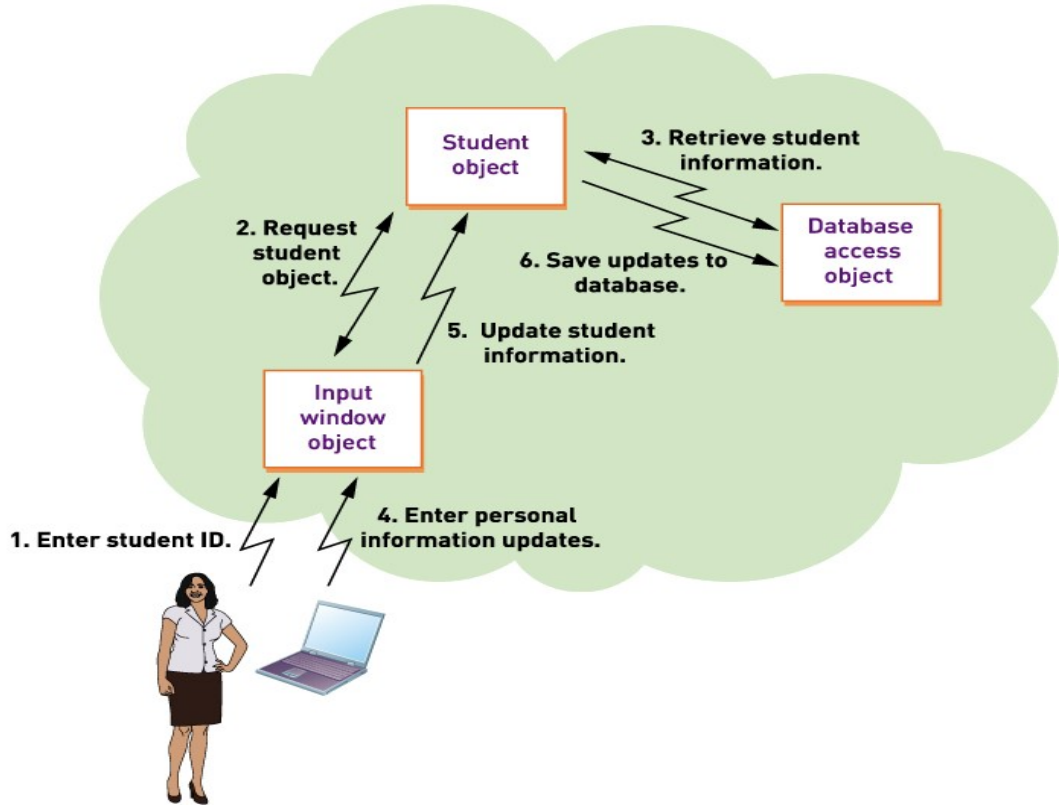
OO Program Flow: Three Layer Architecture

Instantiation

Creation of an object in memory based on the template provided by the class

Method

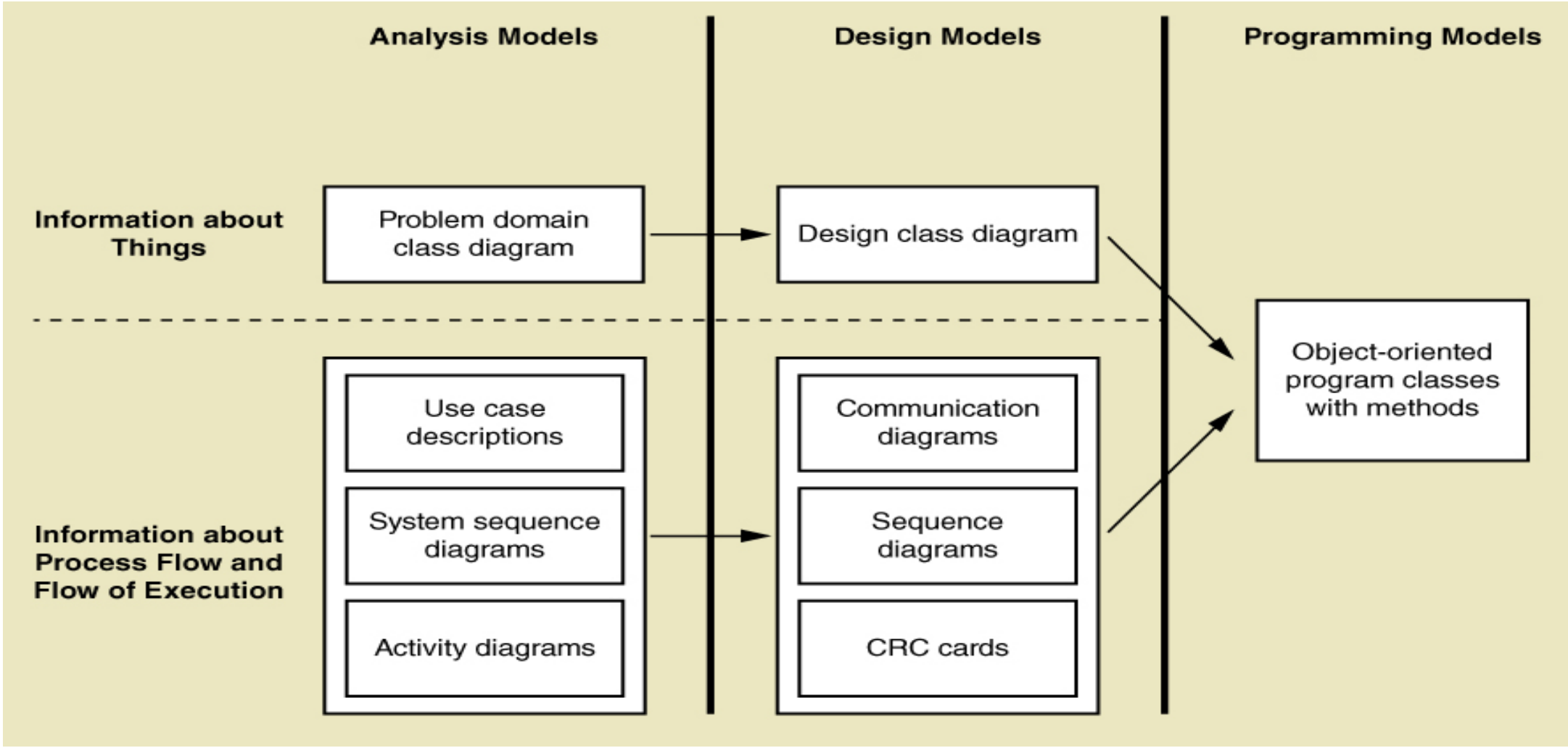
The function executed within an object when invoked by a message request (method call)



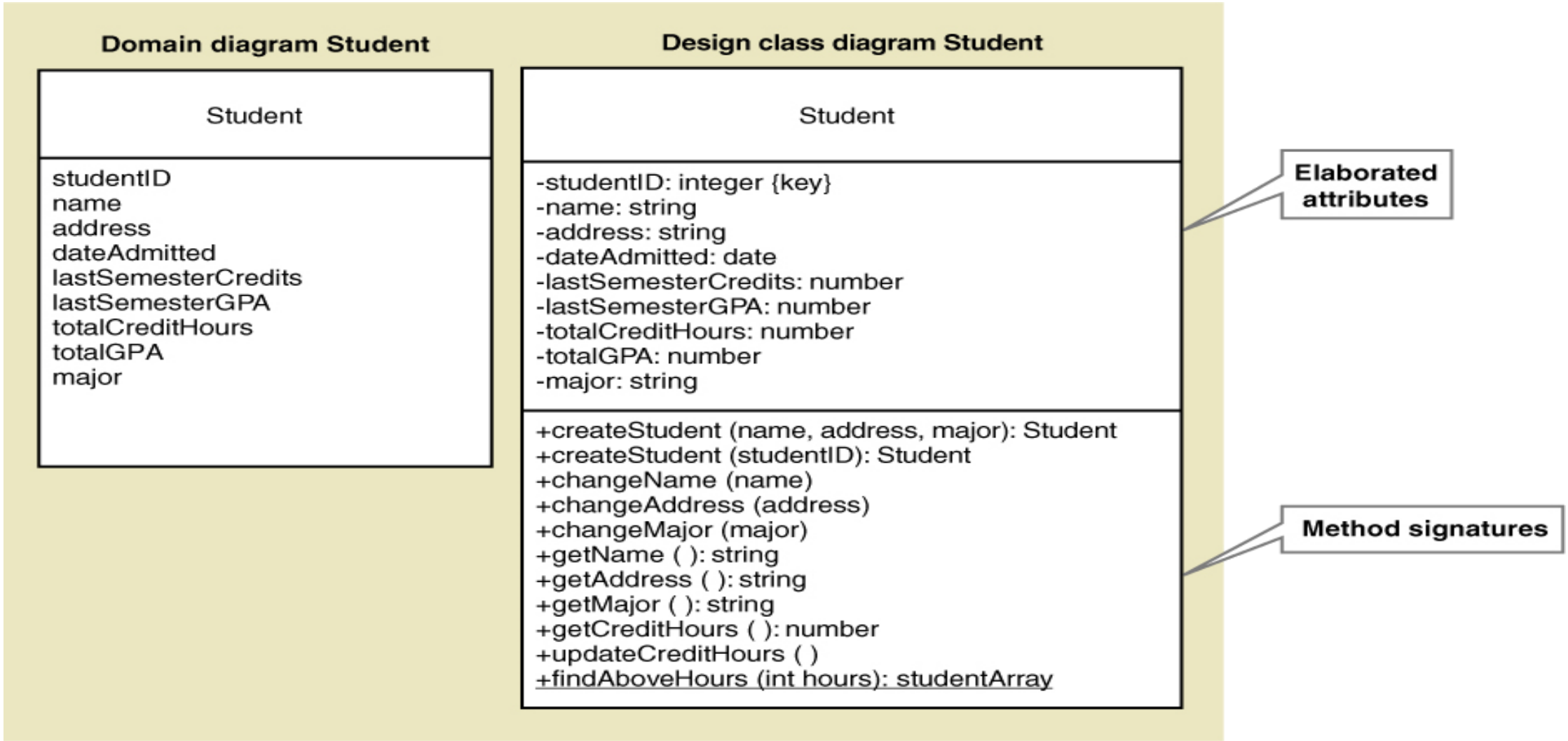
Sample Java with Methods

```
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;
	}
```

Analysis to Design to Implementation: Model Flow



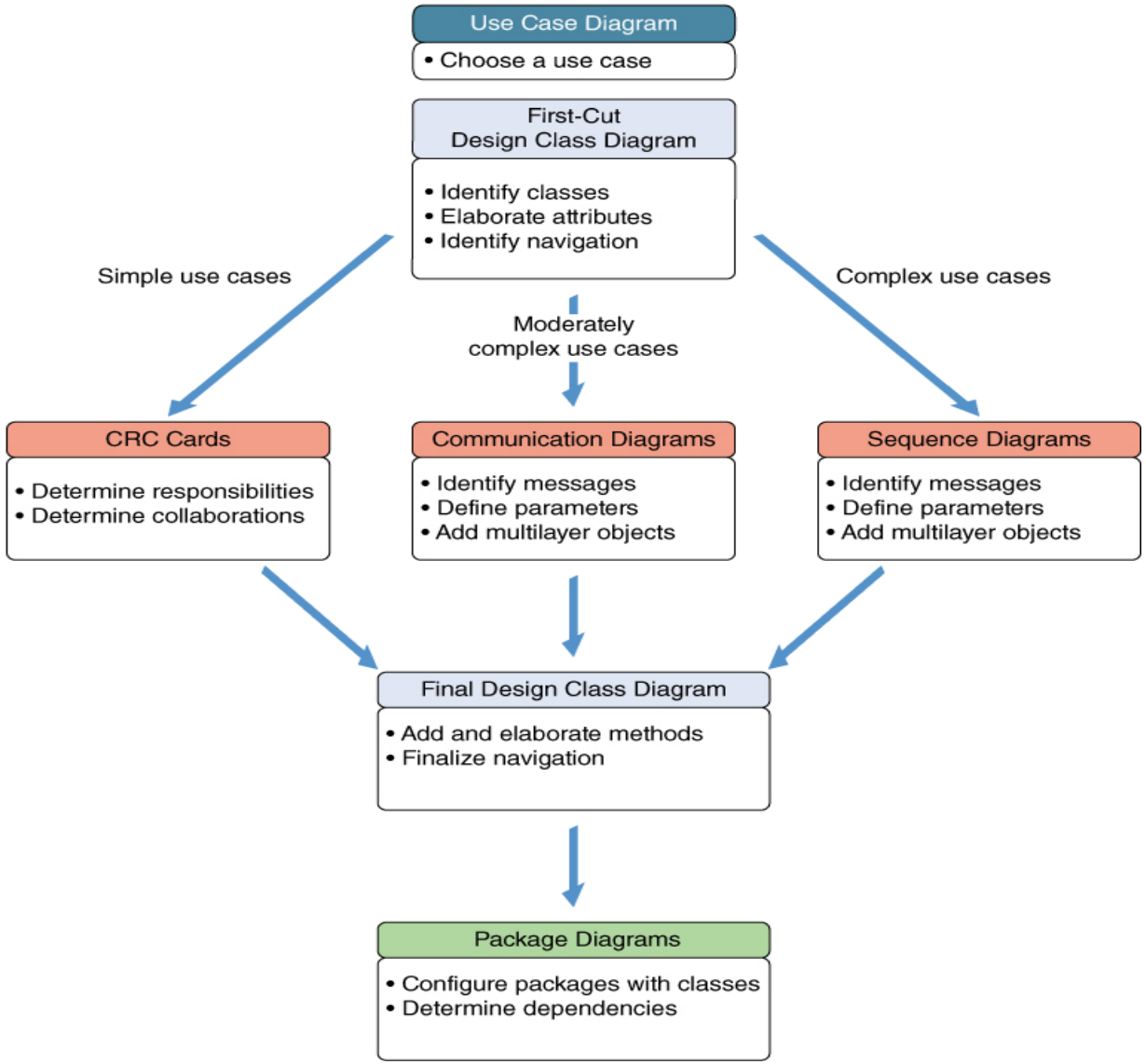
Introduction to Design Models: Class Diagram



Steps of Object-Oriented Design

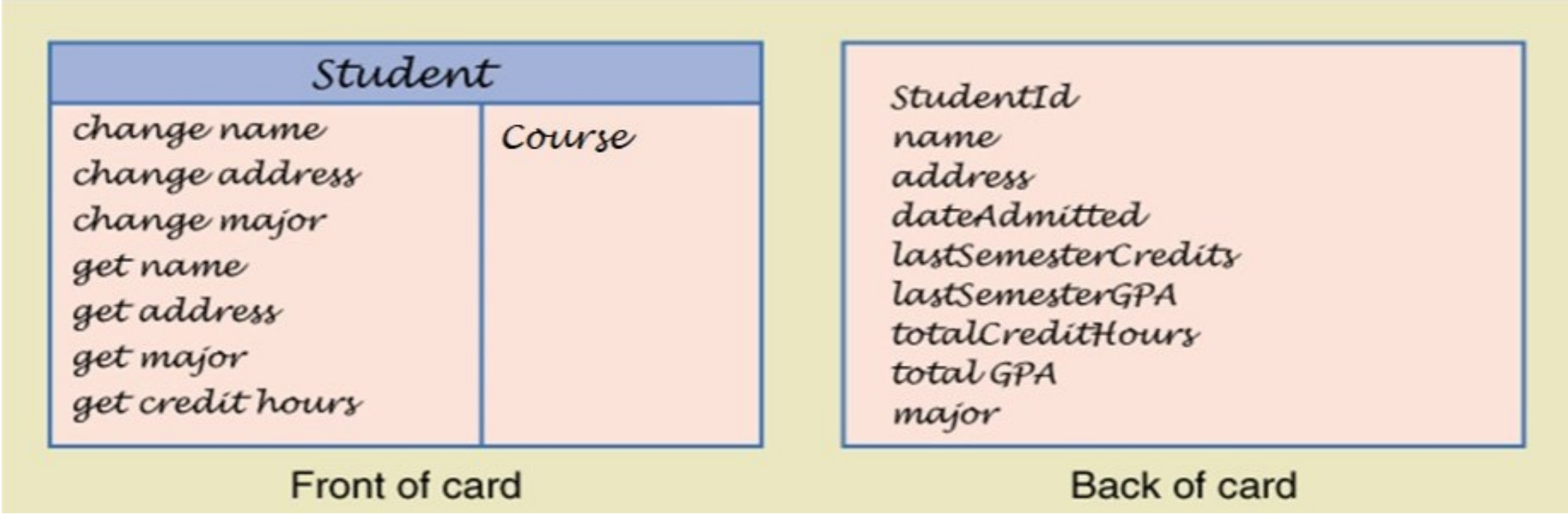
- Object-oriented design
  - The process to identify the classes, their methods and the messages required for a use case
- Use case driven
  - Design is carried out use case by use case
    - Programmer will code all of the methods across various classes that are required to implement the flow of execution of a single use case.
  - Three paths
    - Simple use case use CRC Cards
    - Medium use case use Communication Diagram
    - Complex use case use Sequence Diagram

Steps of Object-Oriented Design



Introduction to Design Models:

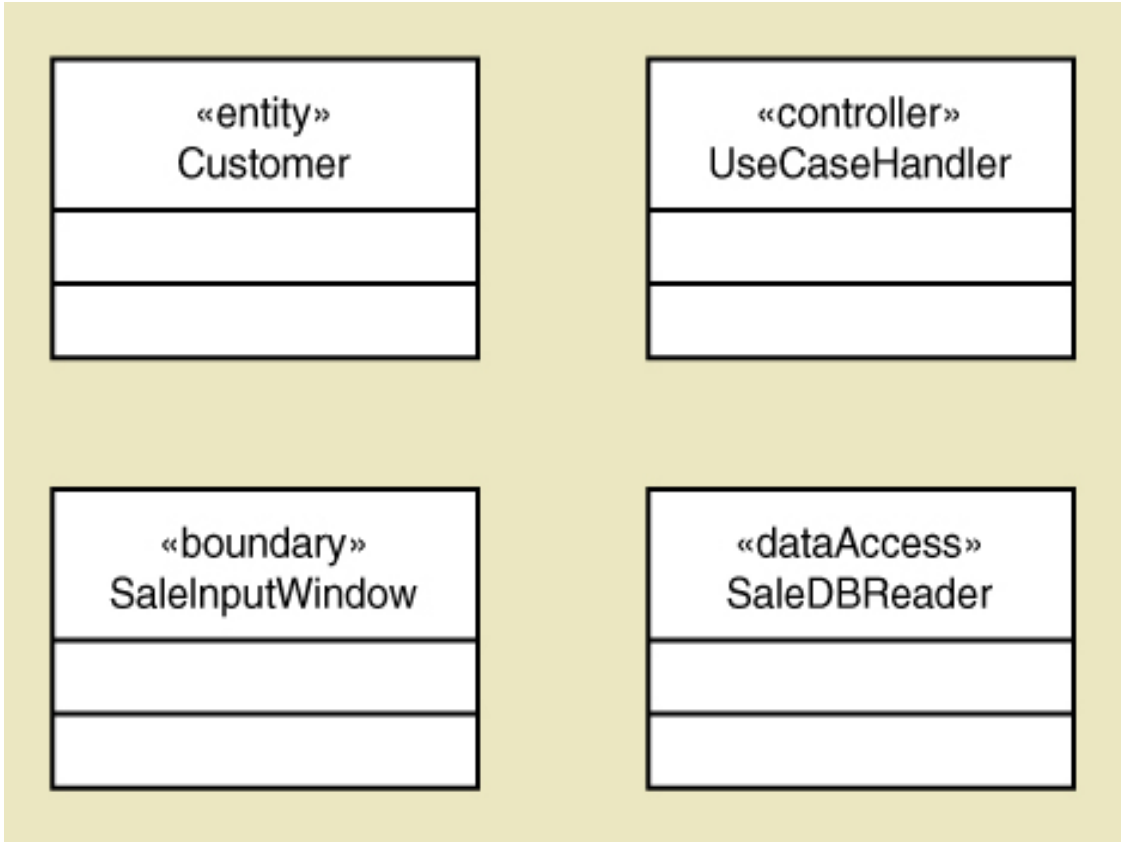
Class-Responsibility-Collaboration (CRC)



Design Class Diagrams

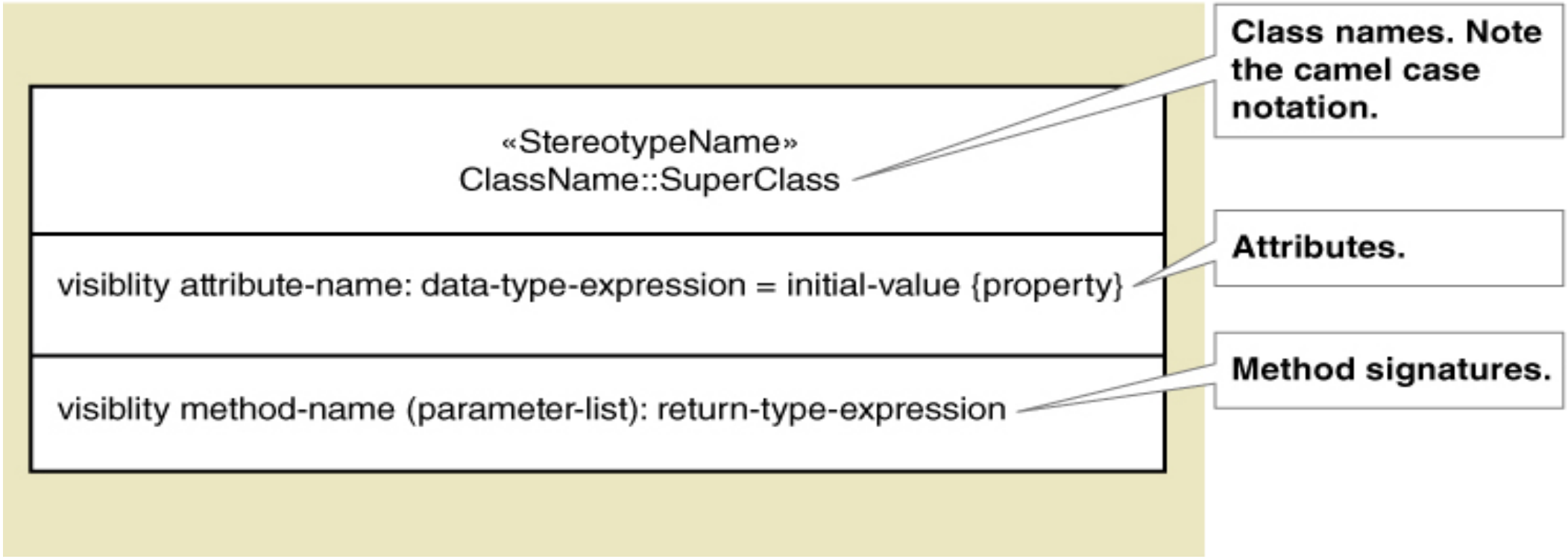
- Stereotype a way of categorizing a model element by its characteristics, indicated by guillemets (<< >>)
- Entity class is the design stereotype for a problem domain class. It typically describes something users deal with when doing their work. Objects of entity classes usually need to be *remembered and are also referred to as persistent classes*.
- Persistent class an class whose objects exist after a system is shut down (data remembered)
- Boundary class or view class a class that exists on a system's automation boundary, such as an input window form or Web page
  - A boundary stereotype is either a user or a system interface class.
- 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



Notation for a Design Class

- Syntax for Name, Attributes, and Methods



Notation for a Design Class

- 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 a Design Class

- 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:

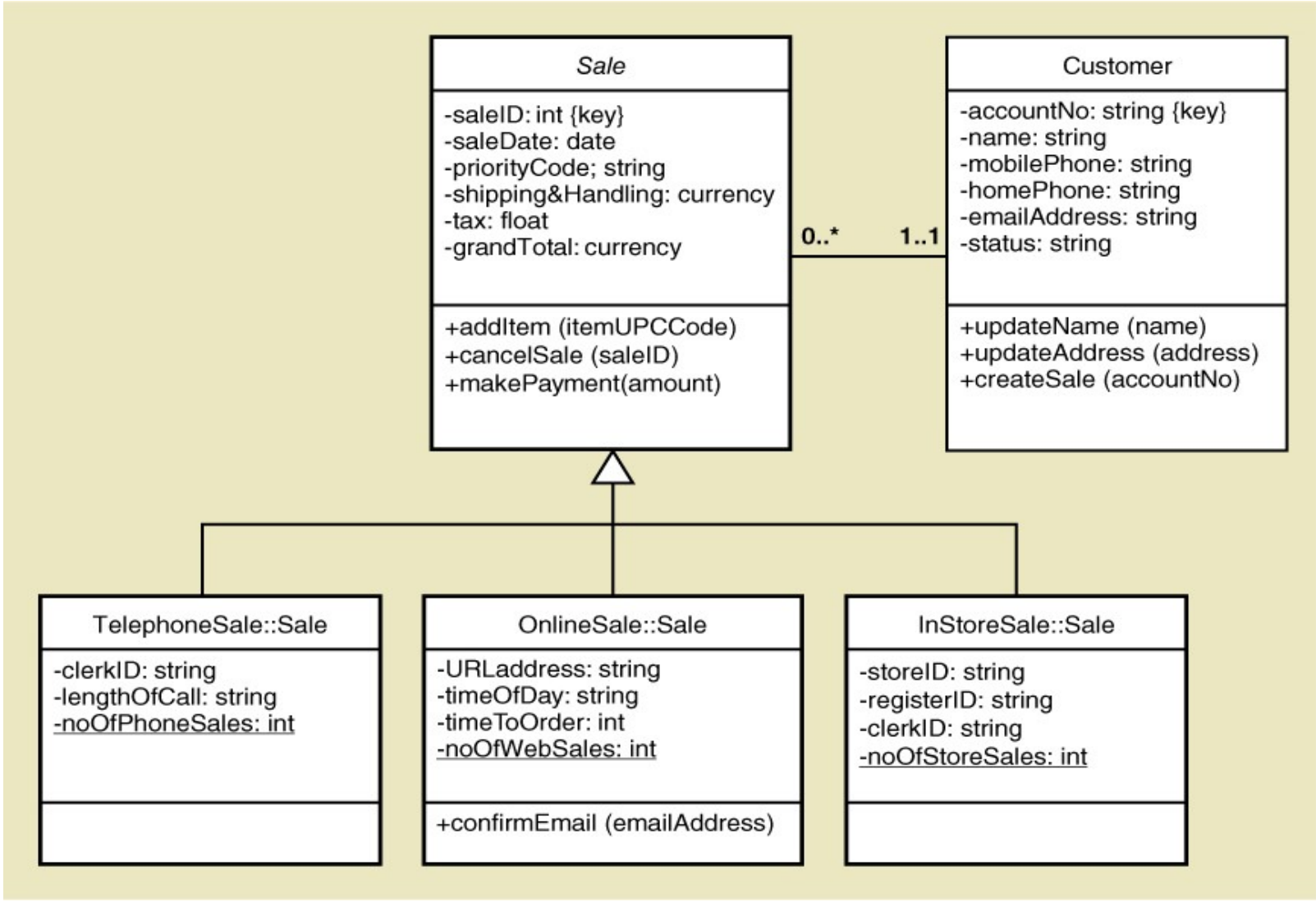
    +getName(): string (what is returned is a string)

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

Notation for a Design Class

- Class level method—applies to class rather than objects of class (aka static method). Underline it.
  - ++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 a Design Class



In- Class Activity #1

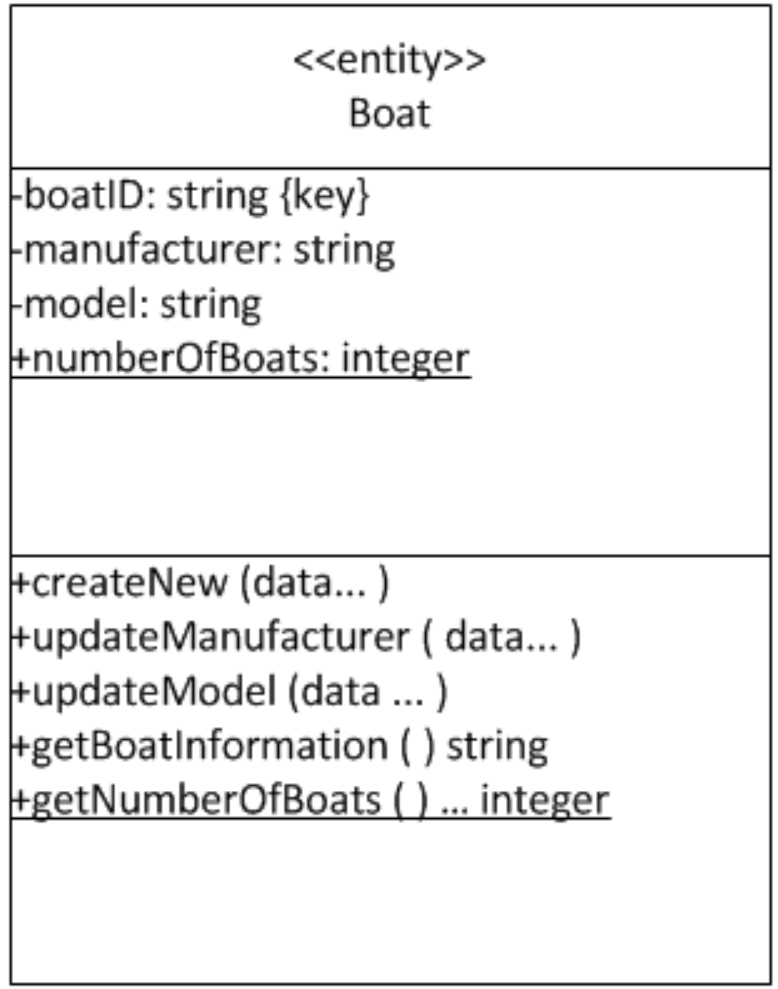
Draw a UML design class that shows the following information:

- The class name is Boat, and it is a concrete entity class.
- All three attributes are private strings with initial null values. The attribute boat identifier has the property of “key.” The other attributes are the manufacturer of the boat and the model of the boat.
- There is also an integer class-level attribute containing the total count of all boat objects that have been instantiated.
- Boat methods include creating a new instance; updating the manufacturer; updating the model; and getting the boat identifier, manufacturer, and model year.
- There is a class-level method for getting the count of all boats.

In- Class Activity #1- Solution

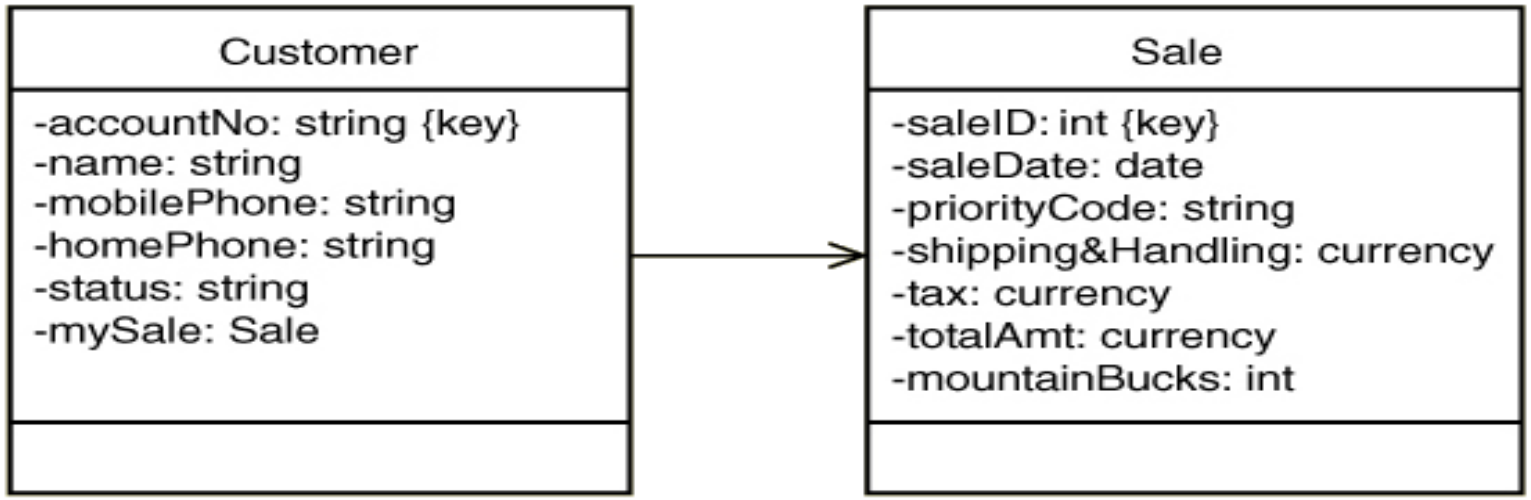
Draw a UML design class that shows the following information:

- The class name is Boat, and it is a concrete entity class.
- All three attributes are private strings with initial null values. The attribute boat identifier has the property of “key.” The other attributes are the manufacturer of the boat and the model of the boat.
- There is also an integer class-level attribute containing the total count of all boat objects that have been instantiated.
- Boat methods include creating a new instance; updating the manufacturer; updating the model; and getting the boat identifier, manufacturer, and model year.
- There is a class-level method for getting the count of all boats.



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 Guideline

Which classes need to have references to or be able to access which other classes?

- One-to-many associations that indicate a superior/subordinate relationship are usually navigated from the superior to the subordinate. —
  - i.e: Sale to SaleItem.
  - Sometimes, these relationships form hierarchies of navigation chains
    - i.e : Promotion to ProductItem to InventoryItem
- 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
  - i.e ; Customer to Sale
- When an object needs information from another object, a navigation arrow might be required
- Navigation arrows may be bidirectional.
  - i.e : a Sale object might need to send a message to its Customer object as well as the reverse.

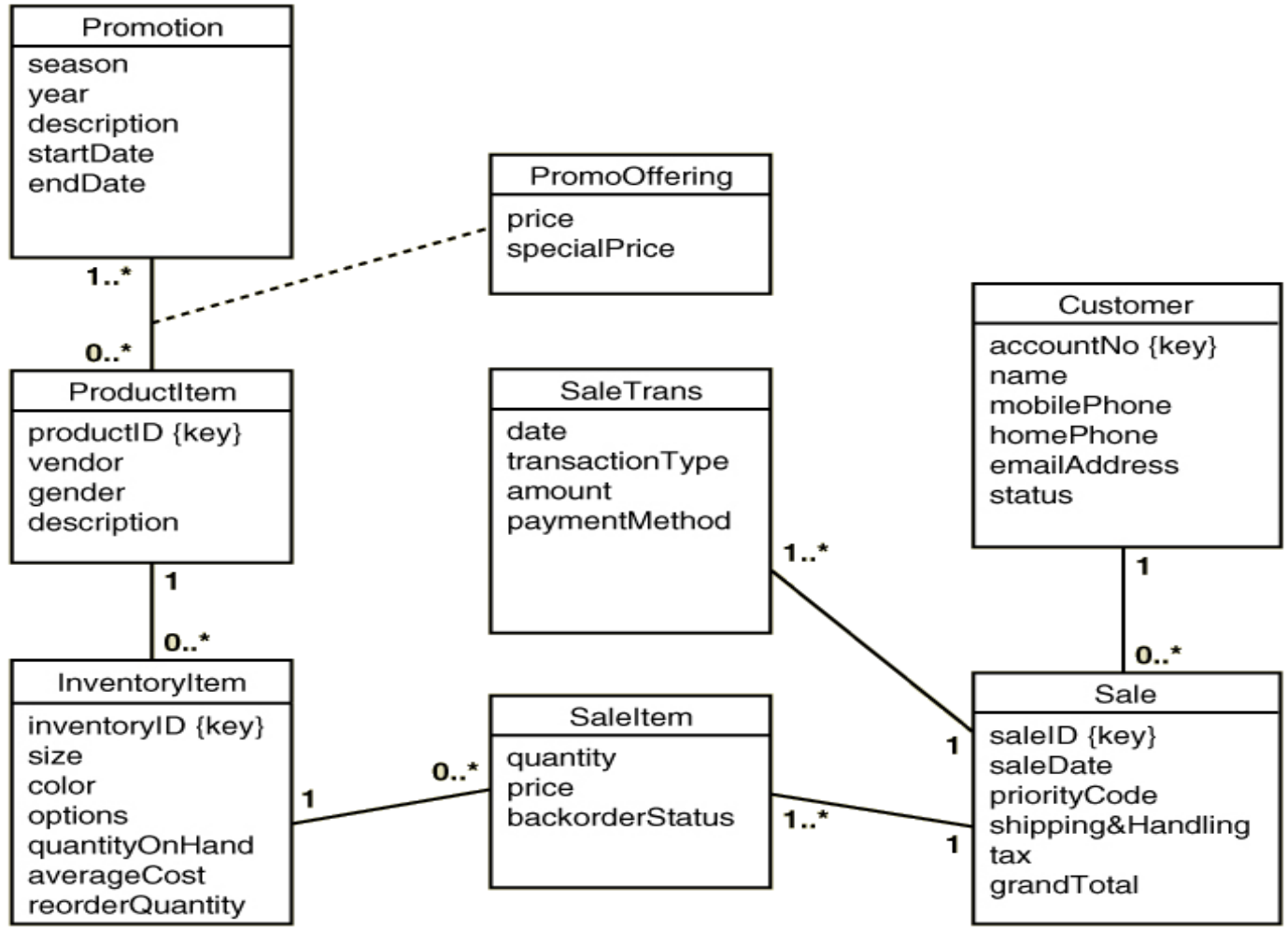
First-Cut Design Class Diagram

- 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

First-Cut Design Class Diagram- Example

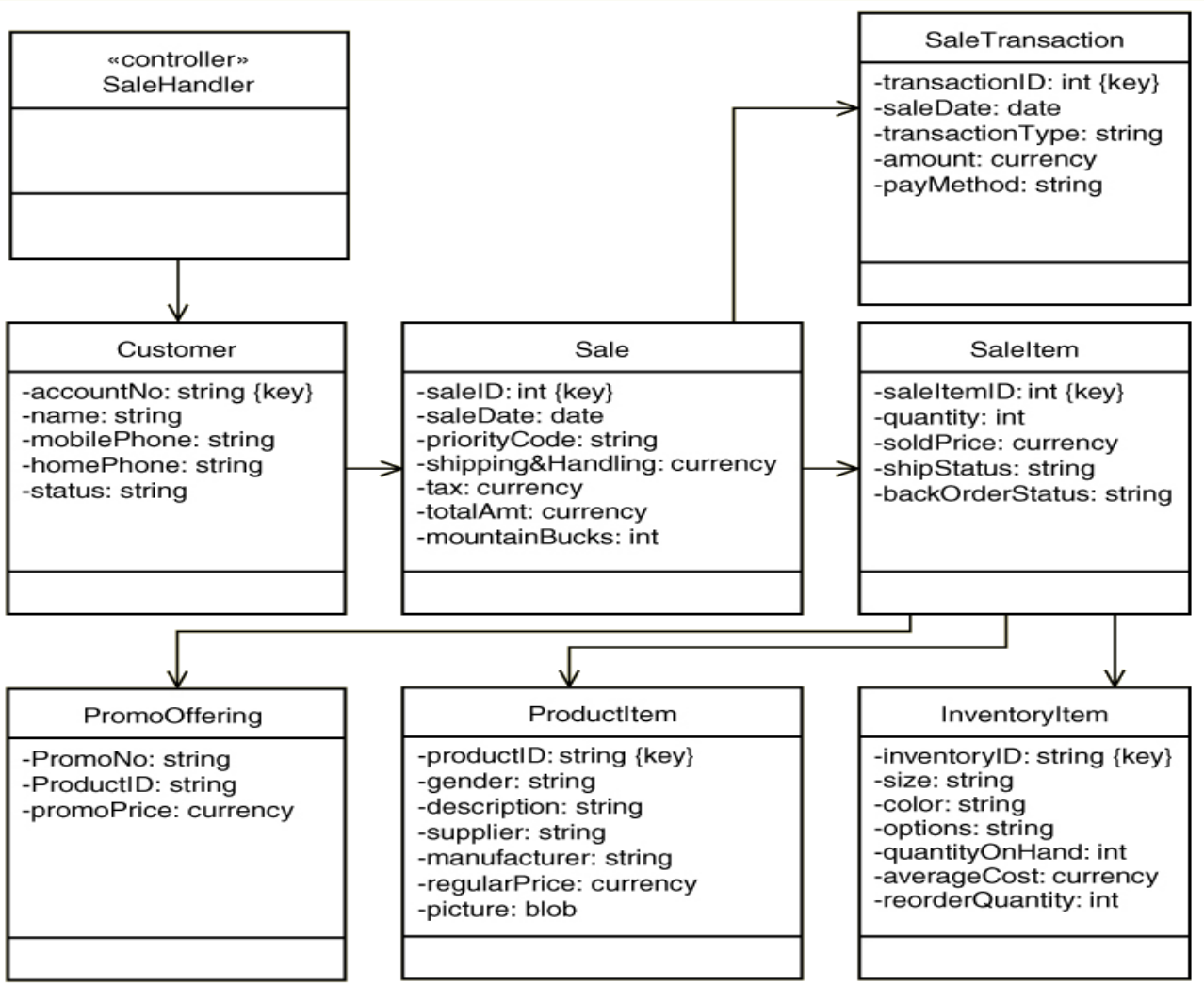
Start with domain class diagram

RMO sales sub system



Create First Cut Design Class Diagram

- Use Case "*Create telephone sale*" with controller added

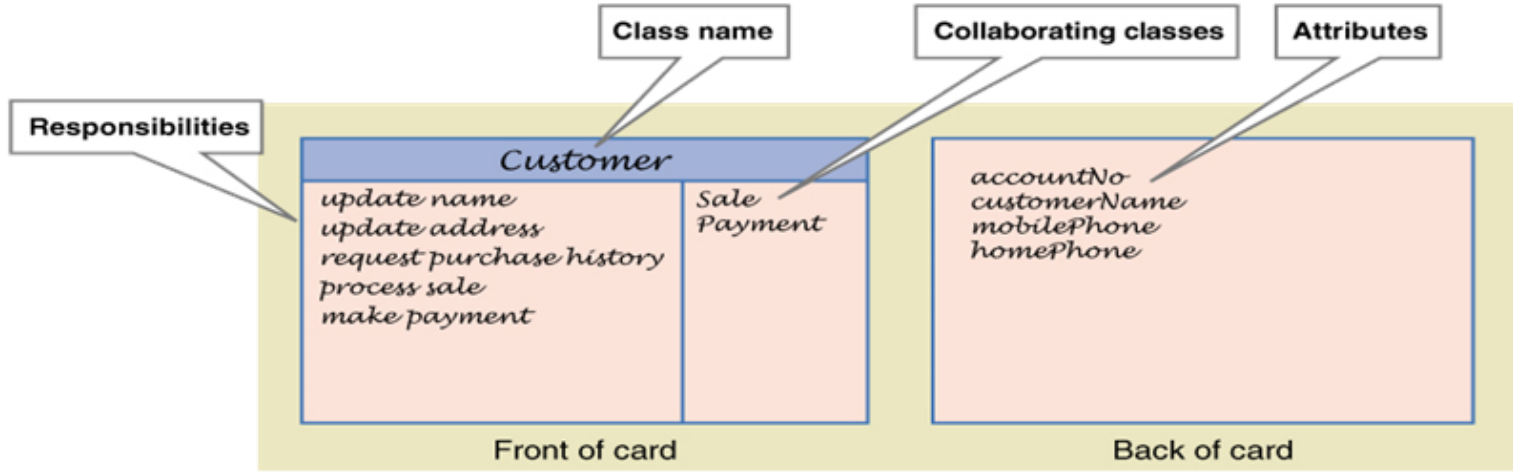


Create First Cut Design Class Diagram -

- Three points are important to note:
  - As design proceeds use case by use case, you need to ensure that the sequence diagrams support and implement the navigation visibility that was initially defined.
  - The navigation arrows need to be updated as design progresses to be consistent with design details.
  - Method signatures will be added to each class based on the design decisions made when creating the interaction diagrams for the use cases.

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



CRC Cards Procedure

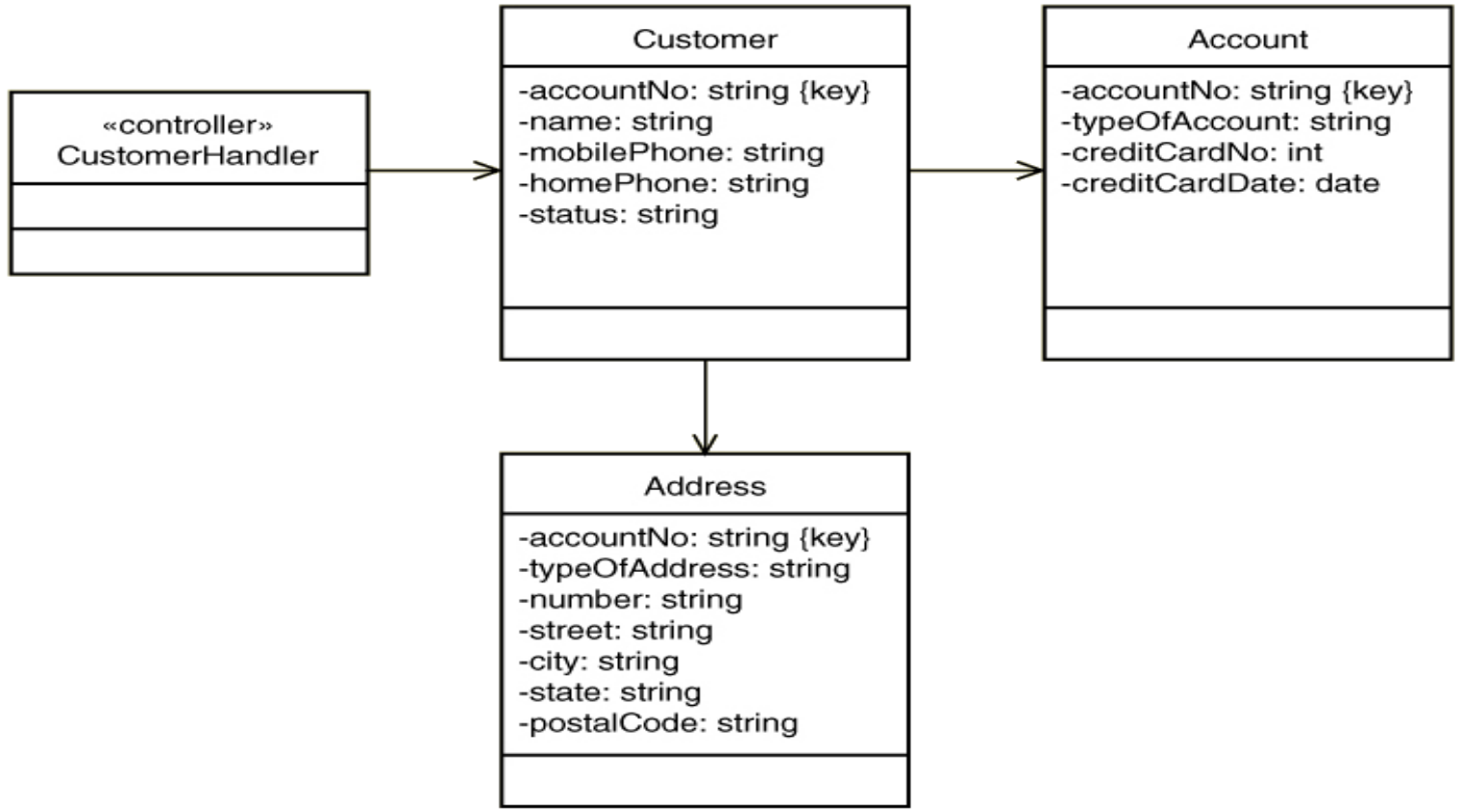
1. Because the process is to design, or realize, a single use case, start with a set of unused CRC cards. Add a controller class.
2. Identify a problem domain class that has primary responsibility for this use case that *will receive the first message from the use case controller*. For example, a Customer 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

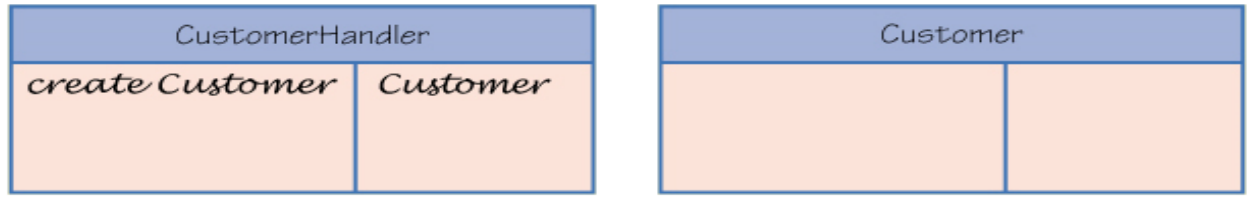
Example – Create Customer Account

- First cut Design Class Diagram

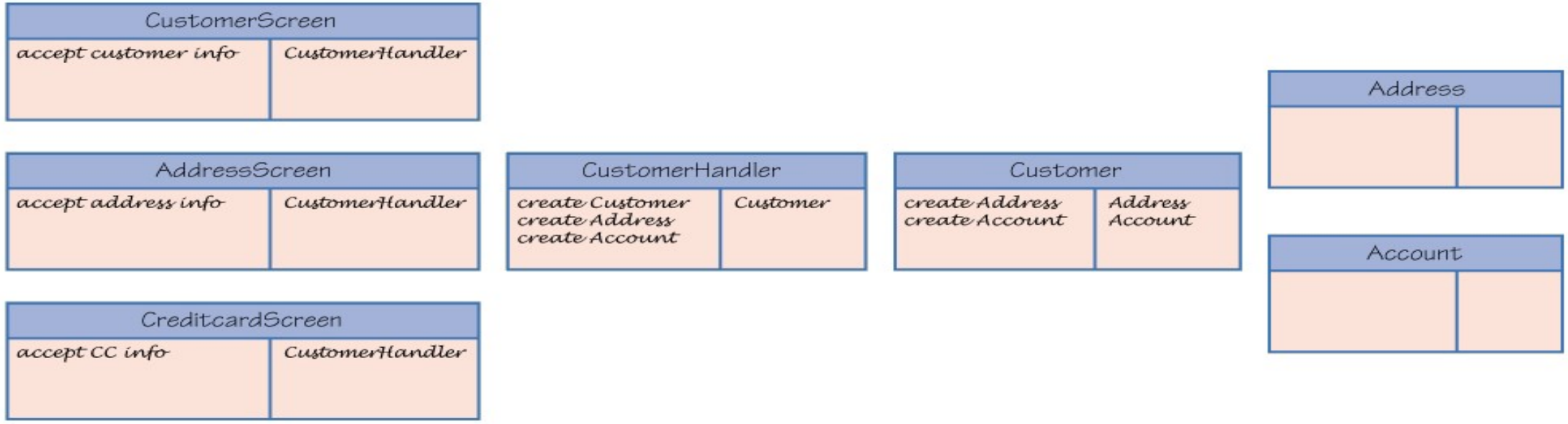


Example – Create Customer Account

- Controller and primary domain class

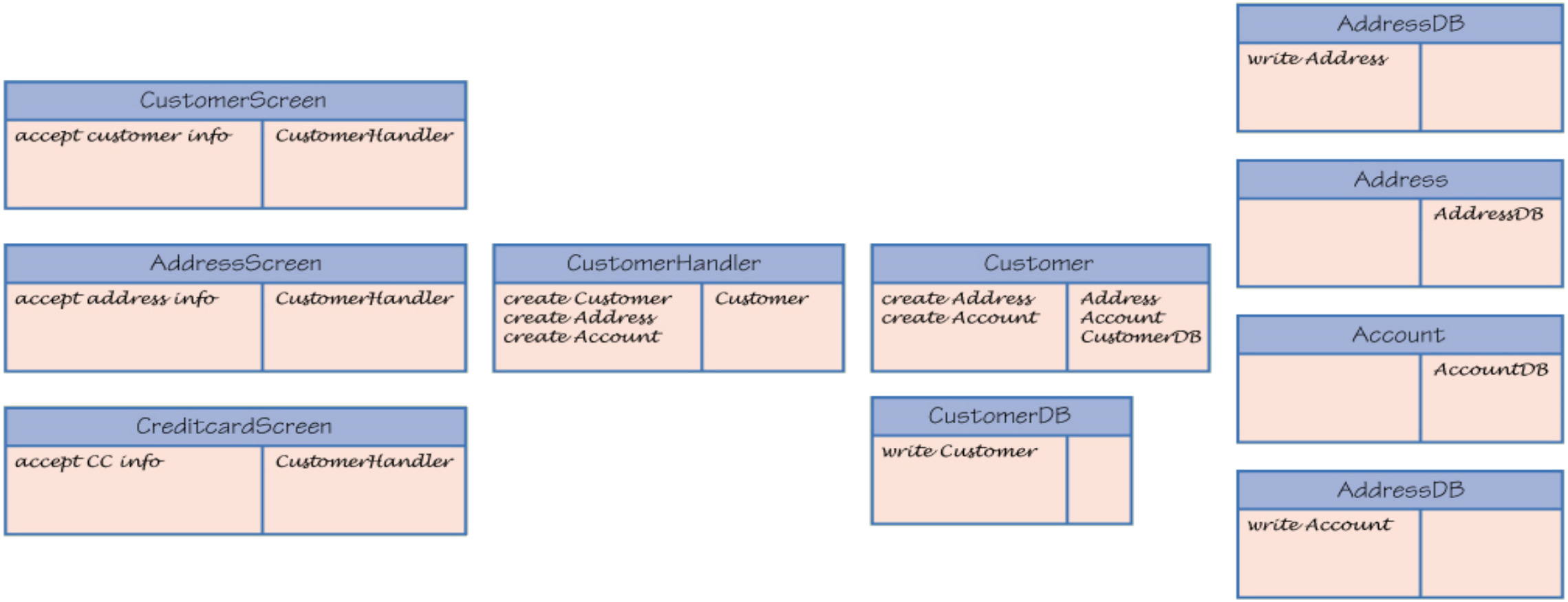


- Adding domain classes and user interface classes



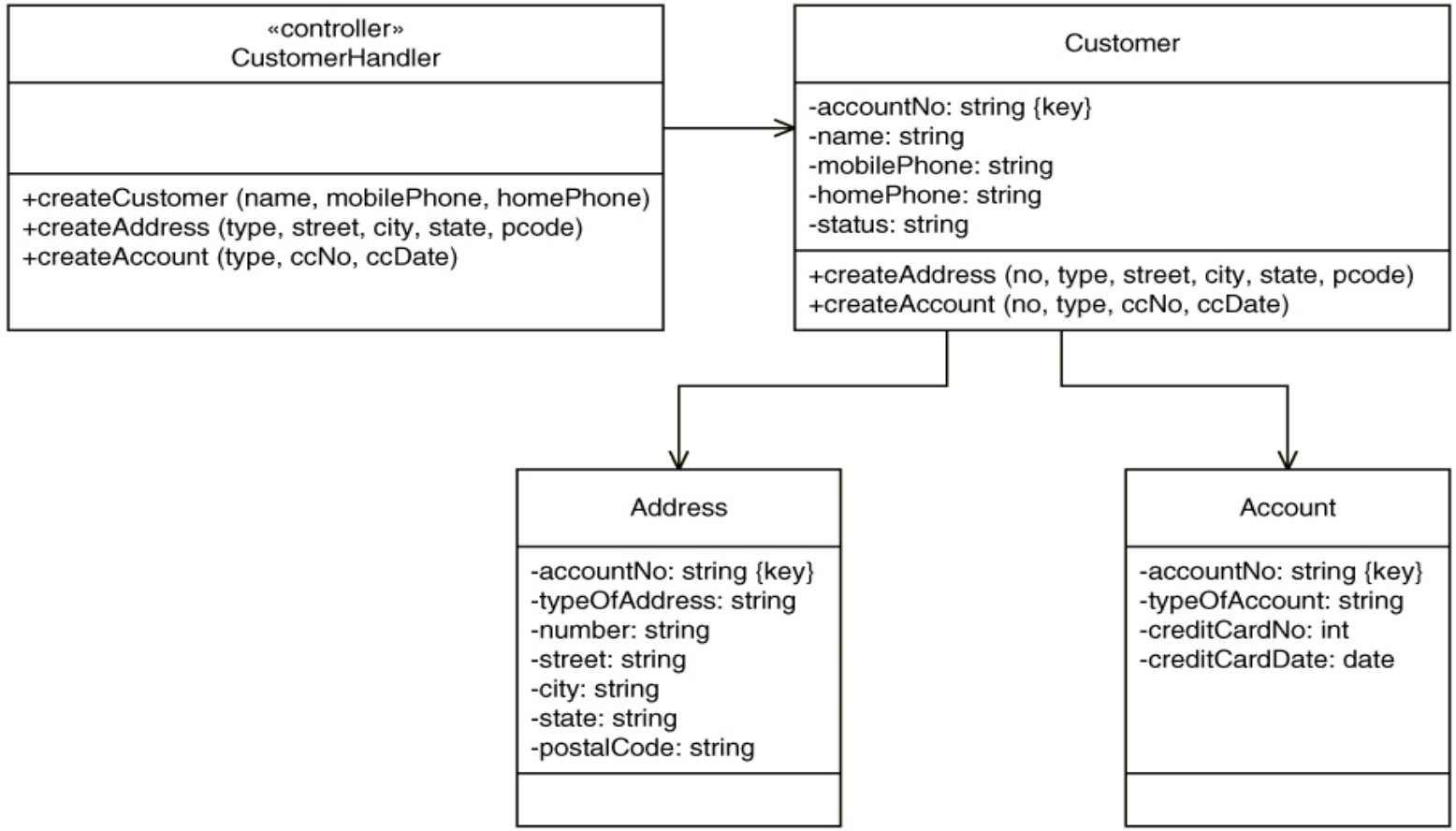
Example – Create Customer Account

- Adding data access classes



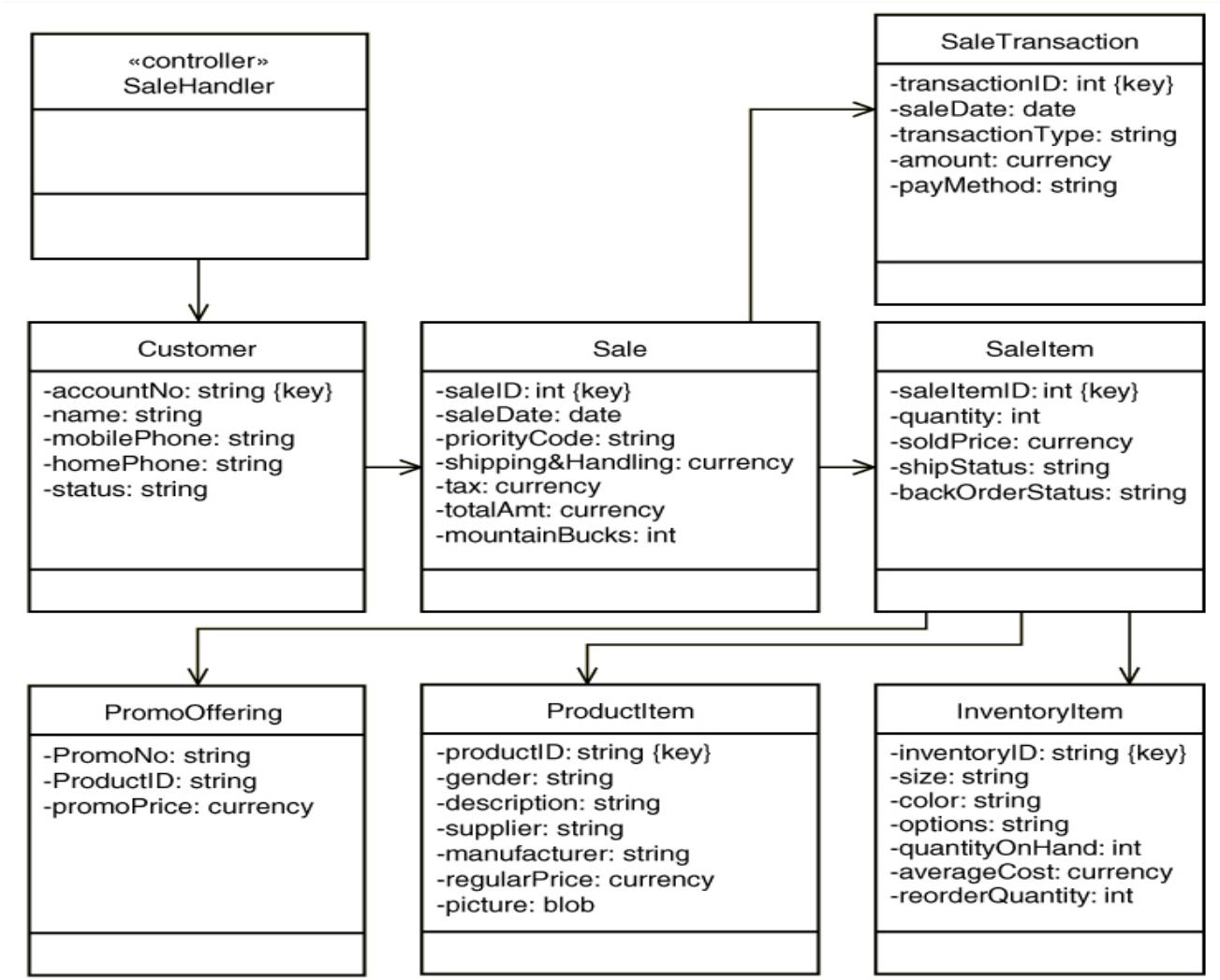
Example – Create Customer Account

- Final Design Class Diagram (DCD) with method signatures

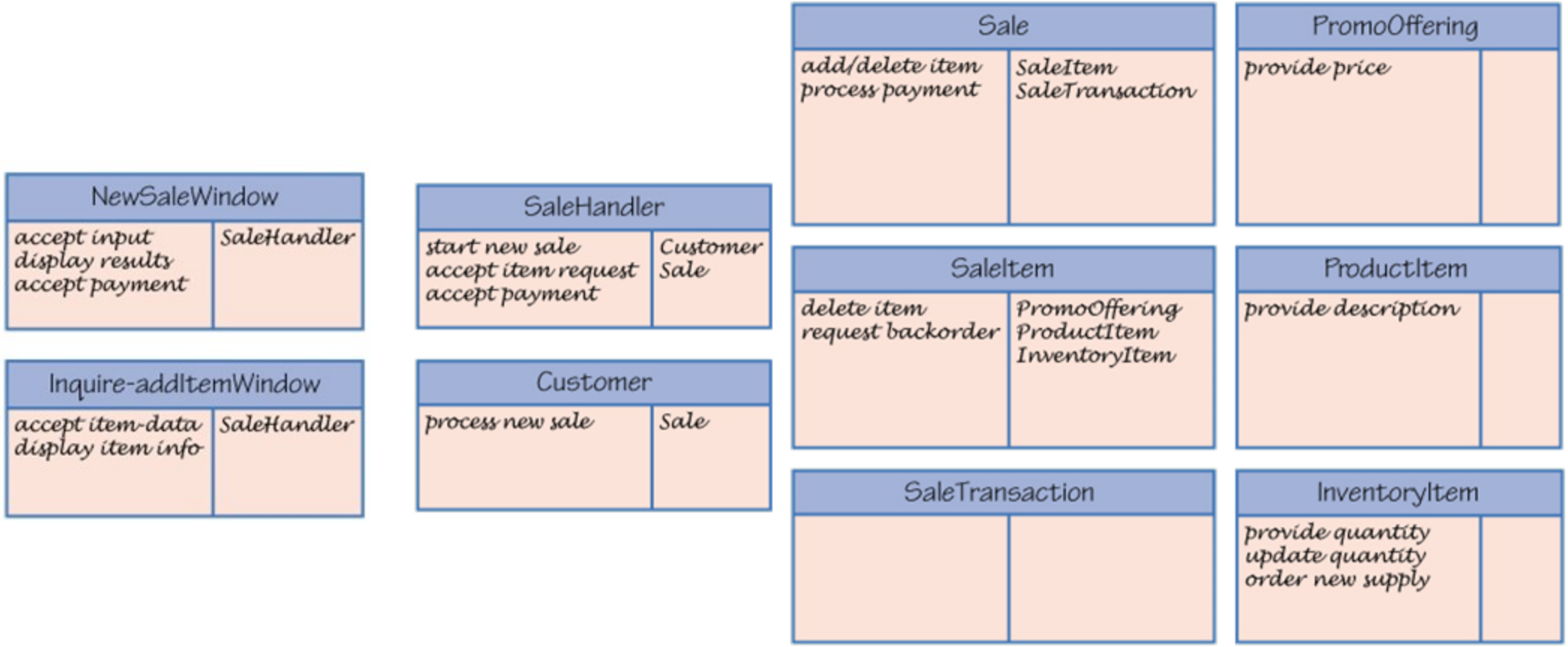


In-Class Activity #2

- Create a set of CRC cards and final DCD for *Telephone* Sale use case based on its first cut design.

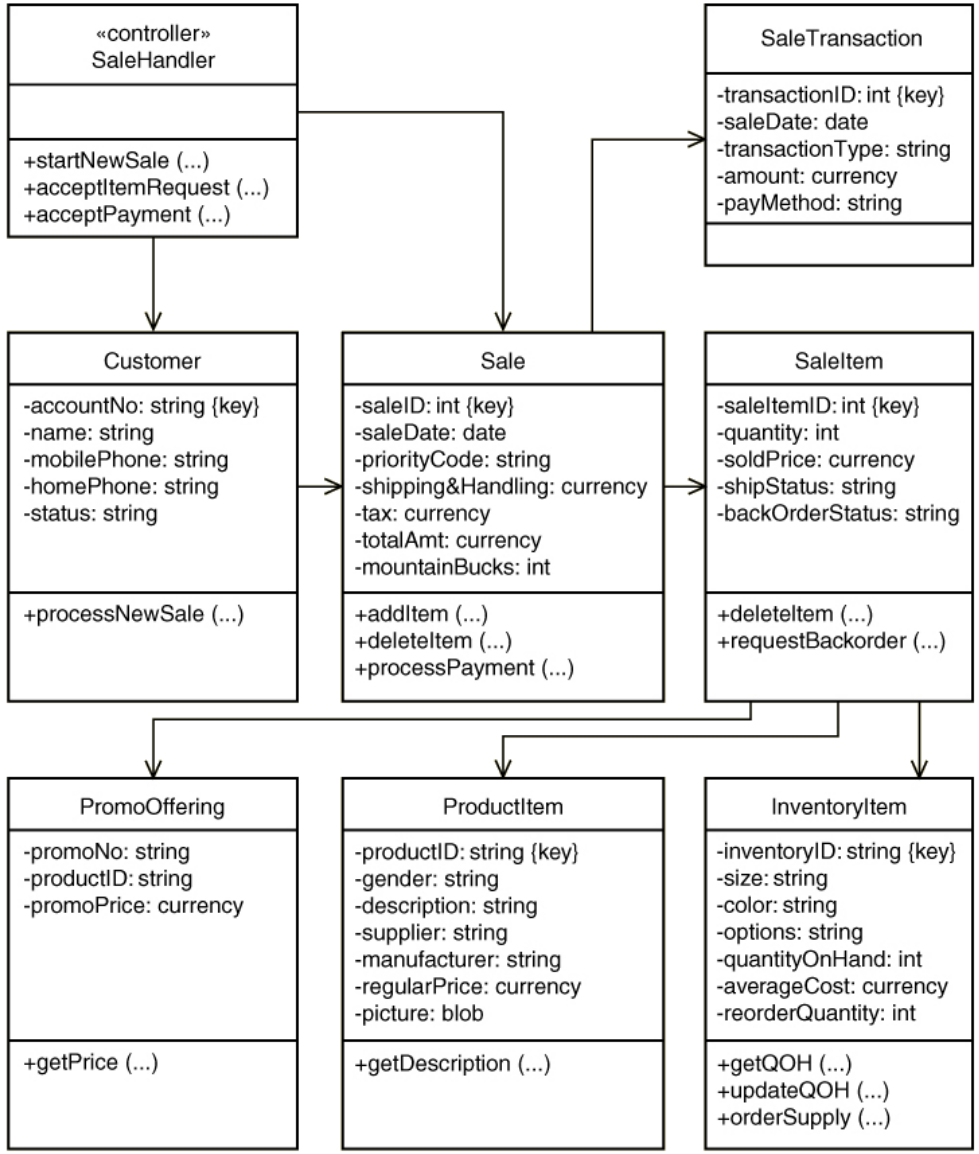


In-Class Activity #2 - Create Telephone Sale - Solution



The SaleTransaction process the payment by instantiating a new SaleTransaction object.

DCD for Create Telephone Sale



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
- 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

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 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
  - If deciding between two alternative designs, choose the one where there is protection from variations
- 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

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.
- 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.
- 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.